Implantation de PPart sous SmallTalk : Synthèse de circuits hiérarchiques

L. Lemarchand

7 mai 2002

Table des matières

Chapitre 1

Introduction

Le système PPart [Lem99] a pour but principal la synthèse logique distribuée de circuits représentés au niveau RTL. Une lacune importante de l'approche réside dans le fait que PPart, pour effectuer la distribution des données, ne se base que sur les informations structurelles du circuit, sans tenir compte de la nature éventuellement hiérarchique de celui-ci.

La complexité du développement de PPart, et notamment les problèmes posés par la manipulation des réseaux booléens, nous ont conduit à réimplanter le logiciel dans un environnement objet. La modélisation des circuits, et en particulier l'introduction de la notion de sous-réseaux, nous ont permis d'étendre plus facilement PPart pour la prise en compte des circuits hiérarchiques.

Ce document contient des informations de plusieurs natures. C'est tout d'abord un manuel technique, à destination des développeurs, explicitant les classes mises en œuvre de manière relativement détaillée. C'est ensuite un document de recherche, présentant les algorithmes utilisés de manière plus théorique. Ces algorithmes (et la simplicité de mise en œuvre) sont une illustration des possibilités de l'environnement OO-PPart pour la mise au point et le test de nouvelles approches distribuées pour la synthèse de circuits au niveau RTL. Enfin, nous essayons de procurer un jeu de scripts de synthèse à titre d'exemple. C'est l'aspect "manuel d'utilisation" du document.

Les premières parties sont consacrées à un descriptif détaillé des classes mises en œuvre dans OO-PPart, et notamment à la modélisation et la manipulation des données. Chaque aspect décrit ci-dessus (partitionnement, synthèse locale avec SIS, synthèse distribuée, conversion VHDL) fait l'objet d'une sous-partie. Nous explicitons ensuite les couplages envisagés entre partitionnement et exploitation de la hiérarchie pour la synthèse distribuée de circuits. Une autre possibilité pour extraire du parallélisme consiste, lorsque les algorithmes de synthèse ne concernent que la partie combinatoire des circuits, à extraire ces parties combinatoires pour les synthétiser de manière indépendante en parallèle. Enfin, nous donnons quelques exemples de scripts de synthèse possibles.

Nous présentons tout d'abord dans cette parties les différentes composantes du logiciel et un exemple détaillé de son utilisation.

1.1 Composantes de OO-PPart

PPart inclut plusieurs composantes :

- Un module de manipulation de réseaux booléens, au travers de leur représentation BLIF.
- Un module de partitionnement optimisé, exploitant le logiciel de partitionnement d'hypergraphes HMetis.
- Un module de synthèse, qui consiste en une interface vers SIS [Sa92].

– Un module de synthèse distribuée, s'appuyant sur le précédent, et sur PVM [GBD+94] pour la partie distribution des données et collecte des résultats

Chacun de ces modules a été réimplanté sous VisualWorks 3.1 :

Synthèse externe avec SIS : PPart exploite SIS pour la synthèse des circuits. Une interface vers SIS permettant le chargement/extraction de circuits dans un processus SIS, ainsi que l'exécution de scripts de synthèse a été développée sous SmallTalk. La modélisation des circuits prend maintenant en compte les circuits convertis vers une librairie technologique. Une interface supplémentaire pour la gestion des librairies au format GENLIBa également été mise en place. SIS ne gère pas les circuits hiérarchiques, mais effectue une mise à plat de ceux-ci en lecture. L'interface permet la synthèse par niveau ou avec mise mise a plat. Cela autorise la préservation des informations hiérarchiques stockées dans le modèle lors de la synthèse, qu'elle soit distribuée ou locale.

Partitionnement de circuits : une interface directe (via DLLC-Connect) permet de partitionner les circuits logiques en tant que hypergraphes. Le partitionnement n'est plus vu comme une information externe au circuit (création de sous-circuits indépendants) mais comme une hiérarchisation de celuici. Cela permet entre autres de mixer les opérations de partitionnement avec d'autres opérations de manipulation du circuit.

Distribution du processus de synthèse : OO-PPart inclut une interface vers PVM pour la création de processus à distance, et la gestion de communications entre processus (ce qui permet, entre autre de se passer de NFS pour la communication des données, comme c'était le cas dans PPart). Cette interface n'est pas spécifique à la synthèse de circuits et peut donc être exploitée pour d'autres applications distribuées sous SmallTalk.

Interface vers VHDL Les circuits hiérarchiques, ou tout au moins les informations concernant la nature hiérarchique du circuit, sont le plus souvent déduites ou inhérentes à la vue de haut niveau correspondant à la synthèse architecturale de ce circuit. Les outils de synthèse de haut niveau produisent des modules RTL inter-connectés, le plus souvent décrits individuellement en VHDL structurel. OO-PPart inclut une interface de lecture/écriture d'une sous partie de VHDL correspondant aux descriptions logiques fonctionnelles de niveau RTL et aux descriptions structurelles RTL. La hiérarchie de modules décrite en VHDL est préservée en lecture et écriture. Un convertisseur permet de repasser en description BLIF.

Interface vers EDIF Une interface vers EDIF a été ajoutée. Elle permet de lire et d'écrire de l'EDIF gnrique, et/ou de l'EDIF correspondant à des formats particuliers (EDIF du banc de test LGSynth 93, EDIF de description des circuits Xilinx xc6200).

La figure 1.1 résume les différentes modélisation de circuits implantées dans OO-PPart, ainsi que les possibilités de lecture et d'écriture de fichiers associées.

1.2 Un exemple

Le code ci-dessous illustre un processus de synthèse :

– Lecture et conversion d'un multiplieur de Wallace, décrit par un ensemble de fichiers représentant une hiérarchie de modules :

```
VHDLParser BlifFileFromVHDL: #('mult32x32' 'add60' 'fa4x1' 'mult4x1'
'mult4x1acc' 'mult4x4' 'wal11' 'wal13' 'wal15' 'wal9' 'wal3x4'
'wal5' 'wal5x4' 'wal7' 'wal7x4' 'wallace16x16'  'wallace32x32'
```

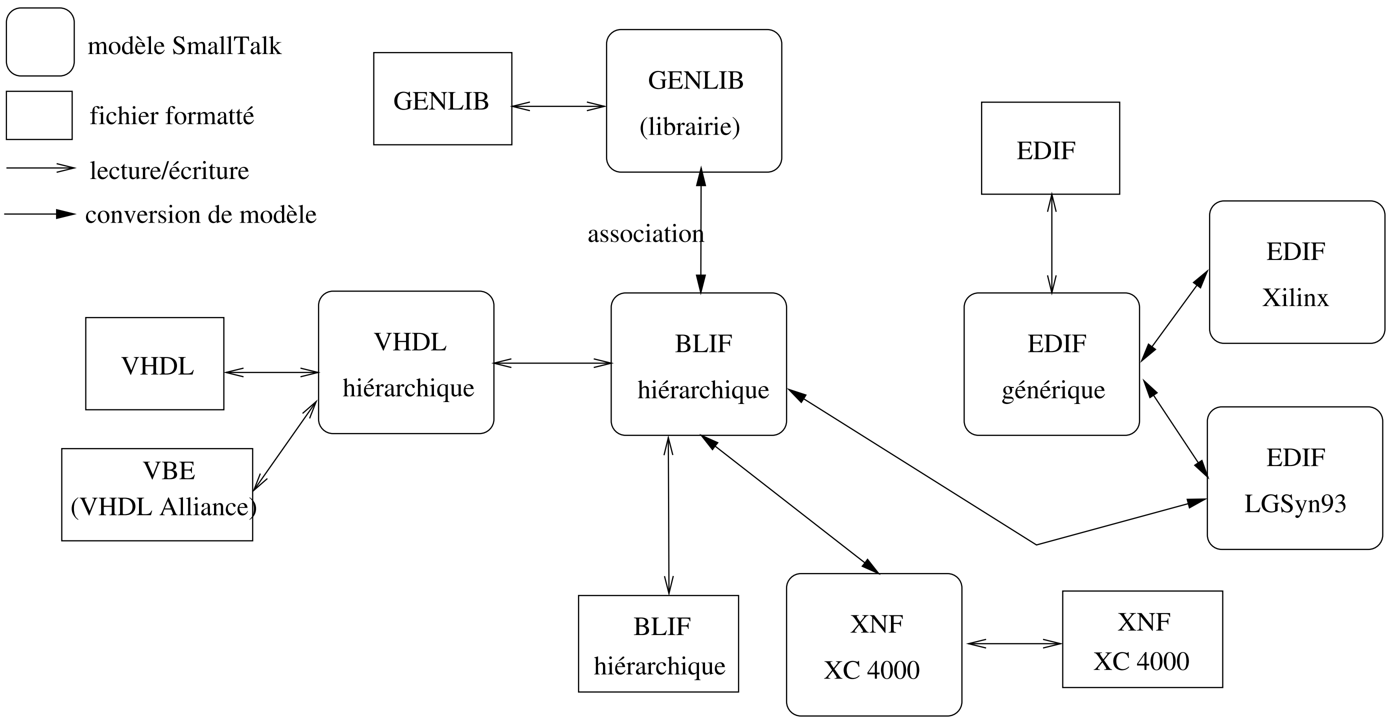

FIG. 1.1 – *Les différents modèles de données utilisés dans* OO-PPart *et l'importation/exportation de données*

```
'wallace8x8' 'fa' 'mult1x1' 'wal3' 'add3' )
inDir:'VHDL/WALLACE32'
toFile:'wallace32.blif'
```

Le résultat est un fichier BLIF hiérarchique.

– Mise en place d'une machine parallèle sur réseau de stations pour réaliser la synthèse du circuit.

```
|p g|
PPart initialize.
g := BLIFModel fromFile:'wallace32.blif'.
p := g ppart.
p start.
p add: 'eckmull'.
p add:'pontusval'.
p graph equilibrateDeepFirst: 7000.
```

La variable `g` contient un circuit, la variable `p` une interface OO-PPart associée à `g` avec 3 processeurs disponibles (le processeur local + 2 processeurs distants). La dernière ligne correspond à une réorganisation de la hiérarchie du circuit, pour obtenir des sous-réseaux de taille (en nombre de littéraux dans la représentation SOP) inférieure ou égale à un seuil.

– Synthèse distribuée des différents sous-circuits. Ceux-ci sont éventuellement re-partitionnés avant d'être envoyés à l'outil de synthèse SIS. La synthèse elle-même (`source script.rugged`) est une technique standard d'optimisation logique incluse dans SIS.

```
p optimizeByLevel: 'source script.rugged' maxSize:3000.
```

– Conversion technologique vers une libraire standard.

```
p graph setLibraryFromFile: 'VHDL/Library/lib2.genlib'.

p optimizeByLevel: 'sweep; map -m 0; phase -g' maxSize:2500.
```

```
p graph equilibrateDeepFirst:5000.
```

```
p optimizeByLevel: 'phase -g' maxSize:4000.
```

  Le circuit est lié à une librairie technologique, puis la conversion technologique est réalisée en distribué et en 2 phases. Un rééquilibrage de la hiérarchie a lieu entre les deux phases.

- Arrêt de la machine virtuelle PVM.

```
p stop.
```

- Sauvegarde sur disque de la hiérarchie obtenue (*netlist*) à la fois aux formats BLIF et VHDL.

```
p graph printVhdlToDir: 'ESSAIPAR'.
p graph printBlifToFile: 'verif.blif'.
```

Nous allons détailler dans les parties suivantes la modélisation des données et les différents modules constituant l'environnement.

Chapitre 2

Circuits hiérarchiques

La notion de *hiérarchie* est ambigüe car elle recouvre deux types de hiérarchies en ce qui concerne les circuits au niveau RTL :

- il existe des dépendances *fonctionnelles* entre circuits, généralement symbolisées par des appels à des sous-circuits (avec ou sans paramètres génériques dans un langage comme VHDL).
- les dépendances structurelles sont également traduites par des appels à des sous-circuits, mais elles reflètent uniquement l'imbrication des sous-circuits entre eux.

L'exemple ci-dessous est une description d'une partie d'un additioneur écrit au format BLIF. Chaque circuit est décrit comme un *modèle* nommé avec la directive `.model` et comportent des entrées/sorties, des fonctions logiques (ou des appels aux composants d'une librairie) ainsi que des appels à des sous-modèles (directive `.subckt`) :

```
.model 4bitadder
.inputs a3 a2 a1 a0 b3 b2 b1 b0 Cin
.outputs Cout s3 s2 s1 s0
.subckt fulladder a=a0 b=b0 cin=Cin s=s0 cout=carry1
.subckt fulladder a=a1 b=b1 cin=carry1 s=s1 cout=carry2
.subckt fulladder a=a2 b=b2 cin=carry2 s=s2 cout=carry3
.subckt fulladder a=a3 b=b3 cin=carry3 s=s3 cout=Cout
.end

.model fulladder
.inputs a b cin
.outputs s cout
.names a b k
10 1
01 1
.names k cin s
10 1
01 1
.names a b cin cout
11- 1
1-1 1
-11 1
.end
```

Il y a dans l'exemple 4 appels au circuit `fulladder` à partir du circuit principal `4bitadder`, avec des paramètres d'I/O différents. La figure 2.1 est une représentation des deux types de dépendances :

Du point de vue de la synthèse globale du circuit, si les deux circuits sont considérés comme étant indépendants, il suffit de les synthétiser séparément. De plus une seule application du logiciel

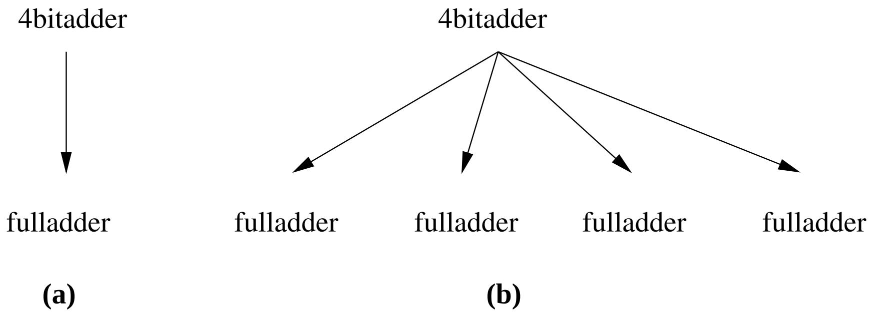

FIG. 2.1 – *(a) Dépendance fonctionnelle et (b) dépendance structurelle entre deux circuits*

de synthèse est nécessaire pour le circuit `fulladder`. En cas de développement (mise à plat) complet de la hiérarchie (avant ou après optimisation), la vue fonctionnelle est suffisante, et permet d'éviter la synthèse multiple d'un même sous-circuit. Par contre si des manipulations hiérarchiques sont envisagées (par exemple réinjection d'un des appels dans le circuit principal avant optimisation, partitionnement du circuit), les appels doivent alors être différenciés entre eux, en considérant la hiérarchie de façon structurelle. Le même problème survient si des paramètres génériques sont inclus dans les appels (VHDL). Les sous-circuits doivent alors être partiellement évalués (suivant leurs paramètres génériques) pour obtenir une description complètement structurelle du circuit.

2.1 Modélisation

La modélisation des circuits hiérarchiques s'appuie, au niveau RTL sur une représentation sous la forme de réseau booléen de ceux-ci.

Un circuit est un ensemble de sommets et de signaux reliant les sommets entre eux. Un signal est une valeur booléen-ne, ayant comme source un sommet, et une ou plusieurs destinations, chaque destinations étant elle-même un sommet. Chaque composant (sommet) du circuit est relié aux autres via un ensemble de terminaux d'entrées/sorties.

La structuration des données est la suivante :

- Un *modèle* est un ensemble de *sommets* et de *signaux*.
- Chaque sommet (composant) possède un ensemble de *terminaux* unidirectionnels.
- Chaque terminal appartient à la fois à un sommet et à un signal du modèle.
- Un *signal* est un ensemble de terminaux.
- Un modèle, en plus de ses sommets, possède un ensemble de sous-modèles (éventuellement vide), associés à certains de ses sommets.

Chaque composant et chaque signal sont associés au modèle auquel ils appartiennent.

2.1.1 Classes de composants

La liste des composants possibles est la suivante :

- un port d'entrée ou de sortie (*pad*),
- une fonction logique à une sortie,
- une bascule décrite par sa fonction logique,
- une porte logique (appartenant à une librairie),
- une bascule (appartenant à une librairie),
- un appel à un sous-modèle (dans ce cas, le sous-modèle est associé au circuit)

La figure 2.2 montre les différentes classes et sous-classes utilisées dans la modélisation. Les classes sans *super classe* explicite héritent de `Object`. La figure 2.3 montre une vue structurelle de l'imbrication des objets dans la modélisation d'un circuit. Chaque objet contient des informations spécifiques non représentées dans l'exemple, mais détaillées par la suite.

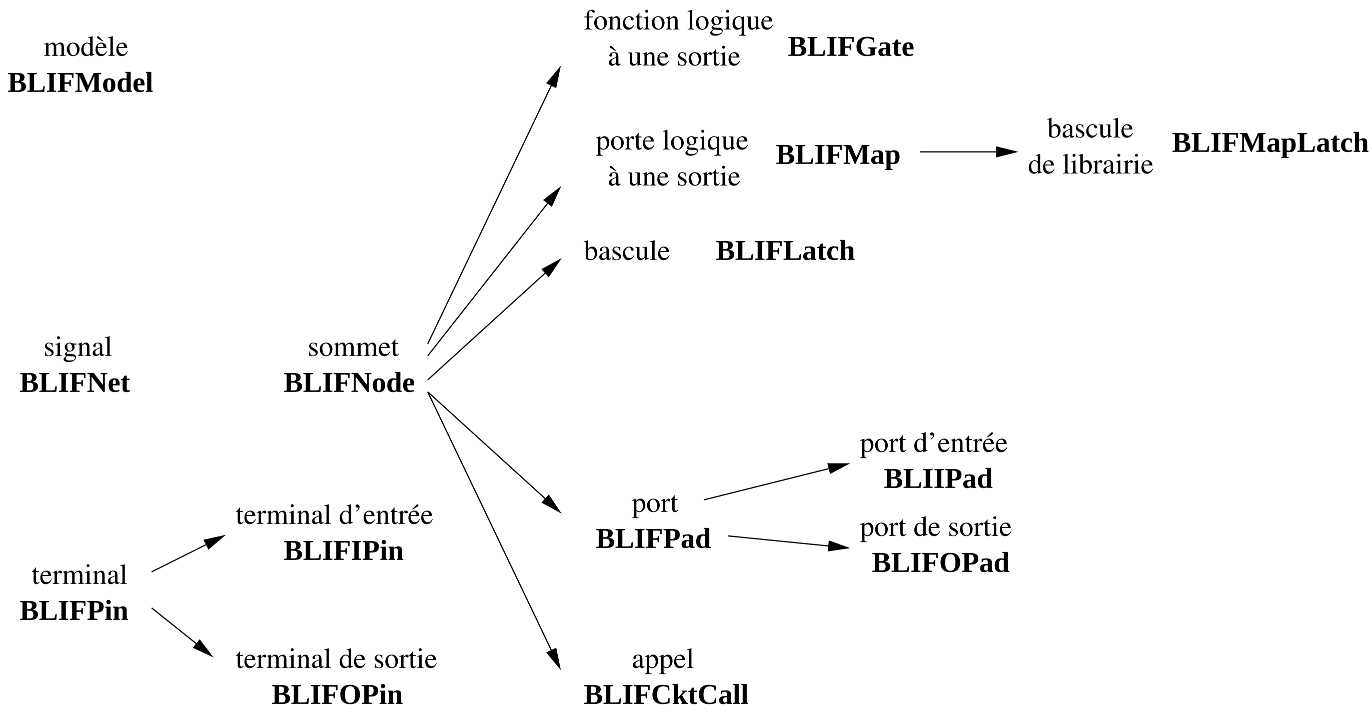

FIG. 2.2 – *Hiérarchie des classes utilisées pour la modélisation de circuits au niveau* RTL *(les noms des classes sont en* **gras***).*

Cette modélisation, et notamment le choix des différentes classes de composants, s'inspire très largement de la représentation des circuits au format BLIF. Les classes Genlib et GenlibLatch ne sont pas représentées dans les figures 2.2 et 2.3. Elles sont utilisées pour associer une librairie technologique à un circuit. Les éléments du circuits correspondant à une conversion technologique (BLIFMap, BLIFMapLatch) sont liés aux éléments correspondant dans la librairie.

2.1.2 Stockage des informations

Chaque objet possède un nom unique (composants et signaux) pour un circuit donné, mais les sont locaux aux composants du circuit (un même nom peut être réutilisé dans un sous-modèle de manière totalement transparente). Chaque composant comporte de plus une variable d'indexage, et une variable *code*, contenant les informations spécifiques au composant. Les terminaux possèdent également un nom, interprété différemment suivant le type de composant auquel appartient le terminal. Nous détaillons en premier le stockage du code associé à chaque composant du modèle, avant d'expliciter le nomage des terminaux et sa sémantique.

Contenu du code de chaque classe de composant. Voici, suivant la classe des composants, le contenu de leur variable code :

- pour les appels à des sous-modèles (classe BLIFCktCall) : le nom du sous-modèle utilisé. Utiliser simplement le nom comme référence évite les problèmes de cohérence des données lors de la modification du modèle appelé (notamment si celui-ci est exporté pour être modifié hors de l'environnement puis réimporté une fois optimisé).
- pour les fonctions logiques (classe BLIFGate) : la description textuelle du PLA de la fonction (voir le format BLIF) (voir également la partie sémantique des terminaux pour la reconstruction de la liste ordonnée des entrées de la fonction logique, liste significative pour la représentation PLA utilisée dans le format BLIF).
- pour les fonctions représentant des bascules (classe BLIFLatch) : la description textuelle des arguments du latch (valeur initiale, nom du signal de contrôle, ...) (voir le format BLIF).

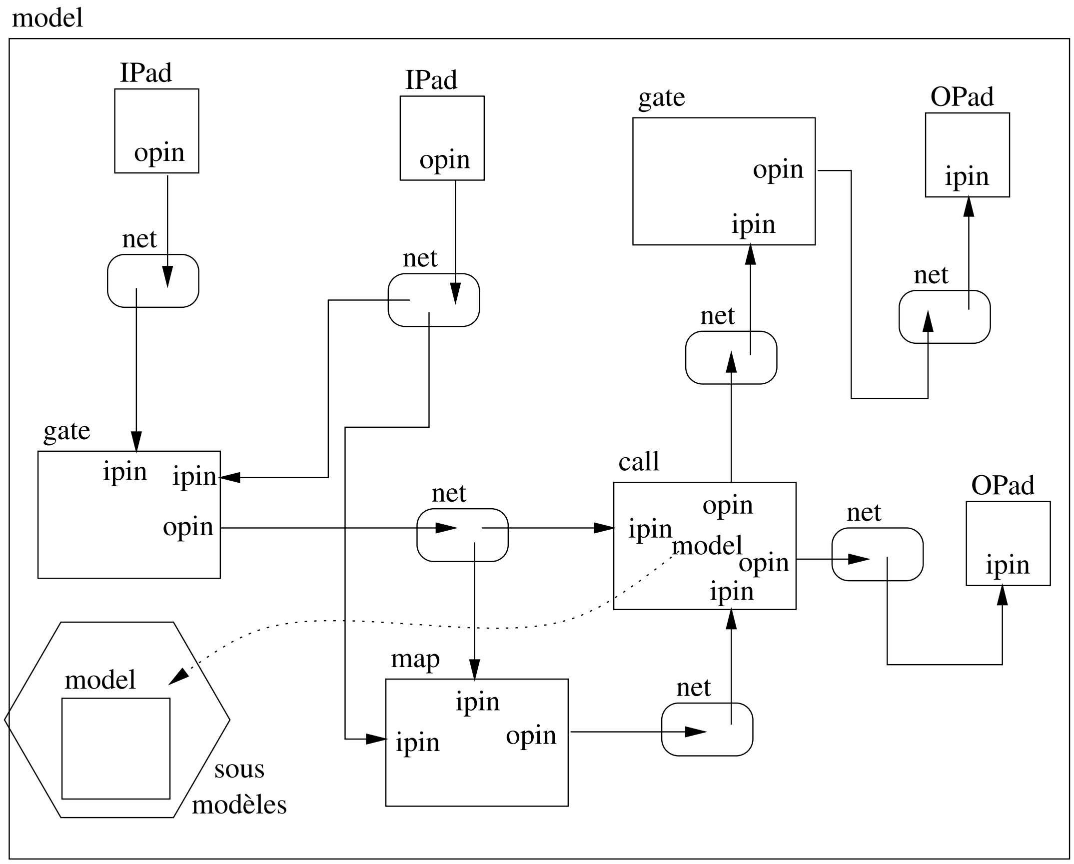

FIG. 2.3 – *Vue structurelle d'un circuit.*

- pour les portes ou bascules issues d'une librairie (classes BLIFMap et BLIFLatchMap) : le nom de la porte utilisée. Cela permet de manipuler le circuit sans forcement avoir lié celui-ci à une librairie. Lors de la liaison avec une librairie, la porte de la librairie est substituée au nom.
- terminaux d'entrée/sorties : vide.

Sémantique du nomage des terminaux Les terminaux contenus dans chaque composant servent à relier celui-ci aux composants dont il dépend (ou aux composants dépendant de lui). La liste des terminaux est non ordonnée. Le nom associé à chaque objet d'une des deux classes de terminaux est *non significative*, notamment lors de la production d'une description textuelle du circuit. Les dépendances entre composants (correspondant aux identités de noms dans les formats de descriptions textuels) sont marquées par les signaux eux-mêmes. Lors de la production d'une description textuelle, c'est le nom du signal (et non le nom du terminal produisant ce signal) qui est exploité pour spécifier les dépendances entre composants. Ce choix a été fait pour faciliter le Re-nomage de signaux lors des manipulations de la hiérarchie des circuits (un nom est local à un modèle, et peut donc être exploité dans un sous-modèle, les deux signaux étant complètement distincts). Les noms de terminaux n'ont de signification que dans les cas suivants :

- Pour les fonctions logiques (classe BLIFGate) : les terminaux d'entrée n'étant pas ordonnés mais le code (le PLA) de la fonction reposant sur un ordre, les noms de terminaux sont `pin1, pin2, ..., pinN`. Cela permet de retrouver l'ordonnancement original des signaux d'entrée de la porte en cas de besoin (notamment, description textuelle au format BLIF).
- Pour les appels à des sous-modèles (classe BLIFCktCall) : les appels correspondent à l'affectation des signaux d'entrées-sorties du sous-modèle à des signaux du modèle. Le nom de chaque terminal (connecté à un signal du modèle) est celui de l'entrée (ou de la sortie) dans le sous-modèle.

2.1.3 Méthodes d'accès

Nous ne détaillons pas ici les différentes méthodes correspondant à l'accès aux informations relatives à un circuit. De manière générale, ils existe des méthodes de sélection des différents types de composants, d'accès, pour un composant donné à son signal de sortie, aux composants dont il dépend, ... Sont aussi disponibles des méthodes de copie simple, de copie avec duplication des sous-modèles, de remplacement de modèle, d'ajout de signaux ou de composants (avec vérification de l'unicité des noms). Il existe également des méthodes de vérification de la cohérence des données (tous les signaux possèdent bien un seul terminal de sortie), de simplification du modèle (suppression des signaux pendants, des sous-modèles non utilisés ...), de recherche d'un composant par son nom, ...

Les entrées/sorties textuelles sont principalement réalisées au format BLIF (format ayant inspiré la modélisation présentée ici) à l'aide des méthodes `printBlifModelOn: aStream`, et `printBlifOn: aStream` qui retournent respectivement une description BLIF d'un modèle et d'un modèle et de tous ses sous-modèles.

La création d'un modèle (ou d'une hiérarchie complète) peut être réalisée à partir de la classe BLIFModel ou directement à partir de la classe BLIFParser, utilisant une grammaire *Lex & Yacc* [Lem01] pour analyser les fichiers source au format BLIF. (voir la catégorie *initializing* de la classe `BLIFParser` pour les différentes méthodes de création).

D'autres formats d'entrées/sorties sont disponibles via l'interface vers SIS. Un sous-ensemble du format VHDL structurel est également utilisable pour la spécification de hiérarchies de circuits (voir section 4.1).

2.2 Sous-circuits et Extraction

OO-PPart possède des méthodes de manipulation de circuits pré-définis (parcours de la hiérarchie, affichage, etc) mais inclut également des possibilités de modification de la hiérarchie d'un circuit. Ces possibilités de manipulation reposent sur deux opérations :

- la *création d'un sous-modèle* à partir d'un ensemble de composants d'un modèle, et
- l'*extraction* (ou *réinjection*) d'un appel à un sous-modèle dans le modèle appelant.

Ces deux opérations permettent de modifier la structure hiérarchique d'un circuit. Elles respectent la cohérence du modèle et sont exploitées ensuite pour la réalisation d'opération de plus haut niveau (mise à plat, partitionnement, synthèse distribuée). Nous détaillons dans cette partie les contraintes et les algorithmes des deux opérations.

2.2.1 Création d'un sous-modèle

La méthode subCkt: aNodeSet name: aName permet de créer un sous-modèle dans le modèle receveur du message. `aNodeSet` est un ensemble de sommets (composants) appartenant au modèle (y compris éventuellement des appels à d'autres sous-modèles). `aName` est le nom du futur modèle qui va être créé à partir de l'ensemble de composants `aNodeSet`.

L'opération est réalisée de la manière suivante :
- création d'un composant d'appel dans le modèle (`newCall`),
- création d'un nouveau modèle (`newModel`) et déplacement de tous les composants de `aNodeSet` dans `newModel`.
- les ports d'entrées/sorties éventuellement présents dans `aNodeSet` ne sont pas déplacés dans `newModel`
- déplacement des signaux internes à `aNodeSet`, *i.e.,* connectés uniquement à des sommets de `aNodeSet`, dans `newModel`

- pour les signaux non internes ayant leur source dans `aNodeSet` (et donc utilisés en entrée d'au moins un des composants n'appartenant pas à `aNodeSet`), duplication du signal dans `newModel`. Le nouveau signal est connecté à tous les sommets de `aNodeSet` qui l'exploitaient dans le modèle original. Un nouveau port de sortie est créé dans `newModel`. Le sommet de `aNodeSet` à l'origine du signal est supprimé de la liste des terminaux dans le modèle original, et remplacé par une connexion à un terminal de sortie de `newCall`.
- à l'inverse, pour les signaux non internes exploités dans `aNodeSet`, un port d'entrée est créé dans `newModel` et connecté à une copie du signal. Un terminal d'entrée est ajouté au signal et connecté à `newCall` dans le modèle original. Les sommets de `aNodeSet` utilisant le signal sont connectés à la copie.
- le modèle de chaque appel inclus dans `aNodeSet` est ajouté aux sous-modèles de `newModel`, et éventuellement retiré de la liste de sous-modèles du circuit original (dans le cas ou l'appel était l'unique appel au modèle dans l'original).
- les informations concernant d'éventuelles librairies sont recopiées dans `newModel`

Cette méthode est générale, préserve la cohérence du modèle, et permet une hiérarchisation au niveau le plus fin du modèle, *i.e.,* pour chaque composant.

2.2.2 Extraction d'un appel

L'opération inverse consiste à remplacer, dans un modèle (`model`), un appel (`aCktCall`) par les composants du modèle appelé `modelCalled`. Les paramètres formels d'entrées/sorties de `modelCalled` sont remplacés lors de la substitution dans des connexions aux signaux correspondant aux paramètres effectifs de l'appel.

La méthode expandCktCall: aCktCall implante l'algorithme :
- suppression du composant `cktCall` de `model` (le receveur)
- ajout d'une copie de tous les composants et tous les signaux internes de `modelCalled` à `model` (avec renomage éventuel en cas de conflit).
- ajout (sans doublons) de tous les sous-modèles de `modelCalled` à `model`.
- mise à jour des informations sur les librairies de `model` à partir de celles de `modelCalled`.
- pour les signaux exportés de `modelCalled` vers `model`, suppression du port de sortie de la liste des terminaux du signal, puis inclusion de cette liste dans la liste des terminaux du signal correspondant dans `model`.
- *idem* pour les signaux importés dans `modelCalled`.
- Si `modelCalled` n'est plus appelé dans `model` (`cktCall` était l'unique appel à `modelCalled`) suppression de `modelCalled` de la liste des sous-modèles de `model`.

Les deux opérations sont exploitées dans des méthodes de plus haut niveau explicitées ci-après.

2.2.3 Opérations sur la hiérarchie

Mise à plat

Il est possible de supprimer un niveau de hiérarchie expédiant le message flatten à un modèle (extraction de tous les appels du modèle). Le message flattenAll effectue une mise à plat récursive (cela correspond à ce qui est réalisé dans SIS lors de la lecture d'un circuit hiérarchique).

Partitionnement disjoint

Si une méthode de partitionnement quelconque (voir la section 2.3.1 *Partitionnement structurel*) permet de réaliser un partitionnement disjoint des composants internes d'un modèle. Ce partitionnement se traduit par l'ajout d'un niveau hiérarchique au modèle. De nouveaux modèles correspondant à chaque partie sont créés. Le niveau de hiérarchie racine n'est plus constitué que par des appels aux sous-circuits créés.

Extraction des parties combinatoires

Il peut être intéressant, pour paralléliser la synthèse de la partie combinatoire d'un circuit hiérarchique séquentiel de restructurer celui-ci pour en exhiber les parties combinatoires et séquentielles. Le chemin de données de la figure 2.4 est décrit par deux circuits, le second étant appelé par le premier en 2 endroits. Les rectangles dénotent des parties purement combinatoires des deux circuits, les barres, des bancs de registres.

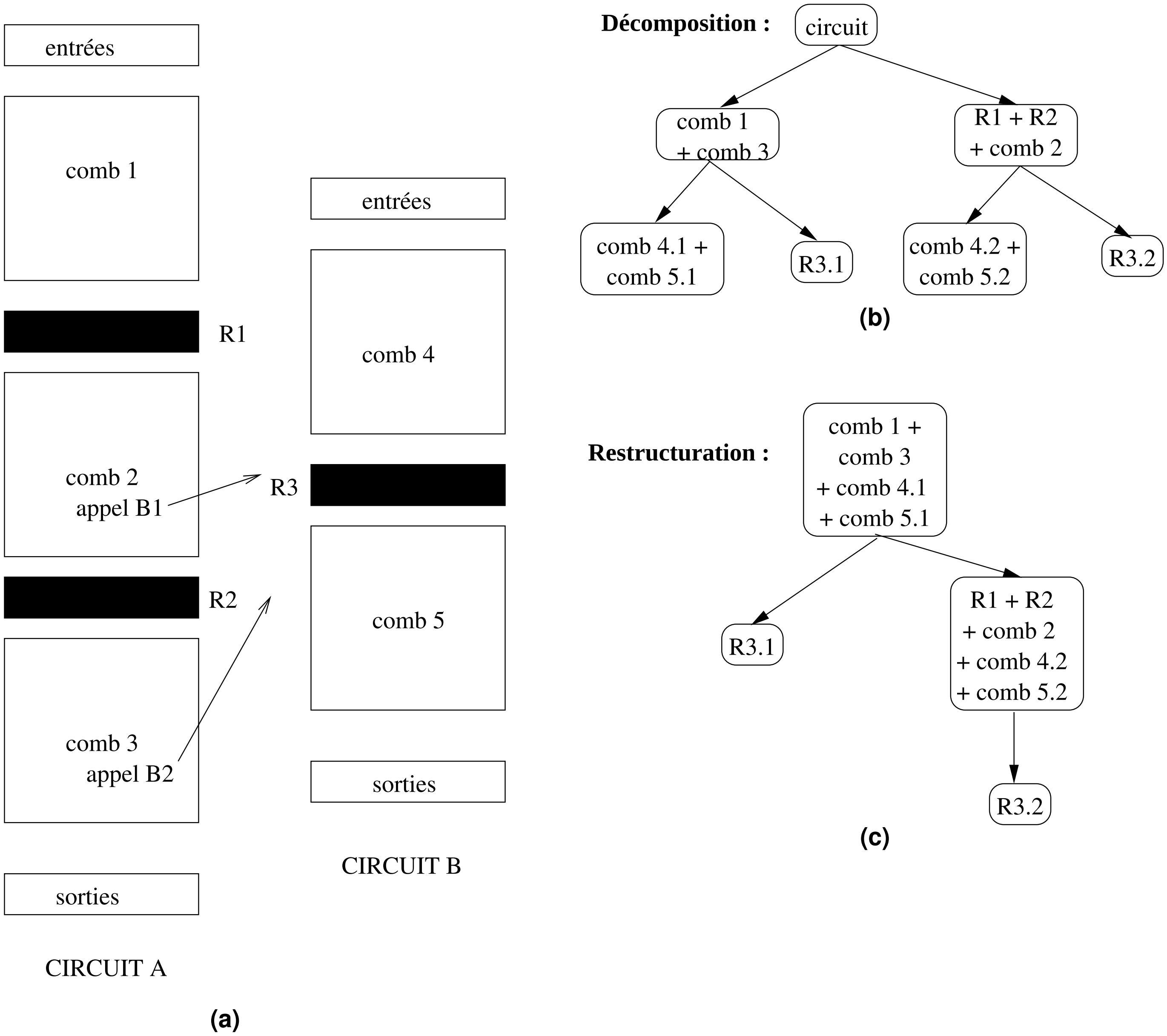

FIG. 2.4 – *Réorganisation d'un circuit (a). (b) Décomposition hiérarchique du circuit en parties combinatoires et séquentielles. (c) Réorganisation du circuit*

La restructuration du circuit fonctionne en deux phases :

- Identification des composants joignables par un chemin combinatoire à partir des entrées-sorties. Les sommets non-atteignables sont isolés par des composants séquentiels du reste du circuit. Ces sommets, ainsi que les bascules forment la partie séquentielle, et est intégrée dans un sous-circuit.
- L'opération précédente est réalisée pour tous les modèles qui composant le circuit. Chaque partie combinatoire est ensuite ré-injectée dans son modèle appelant, récursivement à partir des feuilles de l'arborescence de dépendances. Le circuit hiérarchique obtenu est composé de modèles purement combinatoires, avec des appels à des sous-circuits possédant éventuellement des composants

séquentiels connectés directement aux entrées-sorties.

Étant donnée la méthodologie de restructuration du circuit employée, la synthèse indépendante des parties combinatoire des réseaux obtenues peut être réalisée de manière indépendante sans perte de qualité pour des algorithmes de synthèse se limitant aux éléments combinatoires. L'algorithme est implanté par la méthode combAndSeqHierarchy de la classe BLIFModel.

2.3 Partitionnement de circuits hiérarchiques

La synthèse par partitionnement développée dans PPart a pour but d'accélérer le processus de synthèse en découpant les circuits en modèles de taille réduite. Cela permet à la fois des gains en temps, grâce à la réduction de la taille des problèmes traités par les outils de synthèse, mais il est également possible de paralléliser le processus de synthèse sur réseau de stations. PPart ne prend pas en compte les circuits hiérarchiques. L'adaptation de l'algorithme de partitionnement au cas de ces circuits hiérarchiques a demandé quelques adaptations au niveau algorithmique, en plus du redéveloppement de l'algorithme dans l'environnement objet SMALLTALK.

Nous redonnons brièvement le principe du partitionnement d'hypergraphe utilisé pour la découpe du circuit avant de détailler l'interface vers HMetis mise en place pour résoudre ce problème de partitionnement. Nous montrons ensuite comment il est possible de mixer ce partitionnement structurel du circuit avec le partitionnement hi'erarchique du circuit initial pour obtenir un circuit hiérarchique équilibré, *i.e.,* dans lequel chaque sommet de la hiérarchie ne dépasse pas un certain poids.

2.3.1 Partitionnement structurel

Le but d'un partitionnement structurel est de créer un ensemble de sous-circuits, de taille moins importante que le circuit initial, en perdant un minimum d'informations sur la connectivité entre composants. Les résultats du partitionnement structurel du circuit sont exploités pour restructurer celui-ci en différentes parties, reflétant la distribution des composants en différents blocs. Le circuit initial peut être hiérarchique ou non. Le circuit final est une hiérarchie à au moins 2 niveaux :

- Un niveau racine, ne comportant que des appels à des sous-circuits distincts (les différentes parties).
- Les différentes parties : si une hiérarchie existait déjà avant le partitionnement, cette hiérarchie est préservée : les appels à des sous circuits (composants d'appels) sont traités comme des composants *normaux* par l'outil de partitionnement, et les sous-modèles correspondants sont inclus lors de la construction de la hiérarchie découlant du partitionnement.

Problème de partitionnement

Afin de minimiser la perte de qualité engendrée par le partitionnement, l'objectif de l'outil est de minimiser le nombre de signaux ayant des terminaux dans différentes parties après partitionnement. Pour cela, un problème de *k-partitionnement avec minimisation de la coupe* est résolu sur un hypergraphe déduit de la représentation du circuit sous la forme d'une hiérarchie de réseaux booléens. L'objectif de HMetis [KAKS97] est de minimiser la coupe (les hyper-signaux déconnectés par le partitionnement), tout en équilibrant la taille des différentes parties obtenues. Pour cela un poids est attribué à chacun des sommets de l'hypergraphe. La mesure de la coupe engendrée par la déconnexion d'un hypersignal prend en compte le nombre de terminaux déconnectés ainsi que le nombre de partitions où le signal déconnecté possède des terminaux.

La section suivante montre comment la pondération de l'hypergraphe correspondant au circuit est réalisée. Cet hypergraphe est ensuite envoyé à l'interface HMetis, présentée ensuite.

Pondération

Le but du partitionnement est de réaliser la synthèse des parties, éventuellement en parallèle, en minimisant le coût CPU des opérations de synthèse. Pour cela, un poids doit être attribuer à chacun des sommets, poids reflétant le coût en temps CPU de la synthèse de ce sommet. Trop de facteurs influent sur les temps d'exécution des algorithmes pour pouvoir évaluer *a priori* correctement les temps de synthèse effectifs. La solution retenue dans PPart correspond pour les composants logiques au nombre de littéraux dans la forme SOP de la fonction logique associée au composant, à une constante pour les composants séquentiels et les entrées-sorties. Cette formulation est reprise dans OO-PPart[1]. Pour la partie hiérarchique des circuits, le poids d'un appel correspond au poids du modèle appelé.

La pondération des composants est réalisée à l'aide des méthodes suivantes, implantées dans chaque classe de composants :

- setWeight réalise le calcul du poids d'un composant. Si le composant est un appel, le poids du modèle appelé est renvoyé. Pour un modèle, tous les poids des modèles dépendants sont calculés récursivement.
- weight retourne le poids (stocké dans une variable d'instance) d'un composant.
- ownWeight, uniquement pour les modèles. Le poids propre d'un modèle, sans prendre en compte le poids des appels à ses sous-modèles. Cette méthode est utilisée lors de l'équilibrage de la taille de circuits dépendant les uns des autres.

Le mécanisme de partitionnement s'applique aux circuits hiérarchiques sans modifier les hiérarchies existantes afin de les équilibrer. Le partitionnement ne fait que rajouter une couche hiérarchique supplémentaire, reflétant le résultat du partitionnement des composants.

Interface HMetis

Le partitionnement est réalisé en faisant un appel à la librairie HMetis, via DLLC-CONNECT. La fonction de génération d'hypergraphe (tableaux C) est implantée dans la méthode hmetis. La résolution du problème de partitionnement lui-même est réalisée par la méthode partition: nbParts de la classe BLIFModel, avec le nombre de parties souhaitées en argument. L'interface vers la librairie est la classe ElHMetis de la catégorie *Metis*. La résolution d'un problème, avec notamment tous les arguments par défaut transmis à la librairie est implantée dans la classe HMetis de la même catégorie. La méthode solve résout le problème de partitionnement, et retourne une collection dont le premier élément est la taille de la coupe obtenue. Tous les éléments suivants sont des collections, chacune d'entre elles contenant les composants du hypergraphe associés à l'une des parties.

2.3.2 Équilibrage (partitionnement hiérarchique)

Les différents modèles décrivant un circuit peuvent être de taille très diverse. Du point de vue des outils de synthèse, cela signifie que le temps d'exécution du processus de synthèse pour les différents nœuds de la hiérarchie peuvent être très différents. Cela est préjudiciable au temps global de la synthèse si chacun des modèles de la hiérarchie est considéré de manière indépendante. Nous partons ici du postulat suivant :

Les temps de synthèse *acceptables* dépendent à la fois de l'outil de synthèse utilisé et de la configuration matérielle disponible. La taille acceptable pour un circuit à synthétiser peut être plus ou moins importante en fonction des critères pr'ec'edement cités. Par contre, décomposer le circuit en

1. Le poids d'une porte est évalué par la complexité de sa représentation BLIF. Pour connaître le poids SOP réel d'un modèle BLIFModel, lui envoyer le message sisStats qui affiche les statistiques obtenues par SIS sur le circuit dans le Transcript.

modèles de tailles trop faibles induit des pertes importantes au niveau qualitatif pour le circuit optimisé résultant de la synthèse indépendante de ses sous-parties.

Cette problématique est la même que celle développée dans PPart pour la synthèse avec partitionnement de circuits non-hiérarchiques. Nous considérons que l'utilisateur est capable de quantifier la taille des circuits acceptables par l'outil de synthèse, en fonction des délais attendus en terme de temps de synthèse et du degré d'optimisation souhaité. Le fait d'équilibrer la taille des modèles à traiter repose sur cette quantification. Les méthodes exposées ci-dessous ont donc comme argument la taille maximale souhaitée pour un sous-circuit à traiter de manière indépendante par un outil de synthèse. Cette taille, exprimée en nombre de littéraux dans la forme SOP du circuit (voir PPart) est notée `maxLit` par la suite.

Pour un circuit donné, 2 cas peuvent se présenter :
- Le poids de sa forme SOP, hors coût SOP de ses appels à des sous-modèles (voir section 2.3.1) dépasse `maxLit`.
- Le poids de sa forme SOP est inférieure à `maxLit`

Dans le premier cas, un partitionnement structurel permet de réduire la taille du modèle. Le choix du nombre de parties est fonction de `maxLit` et de la taille du modèle.

Dans le second cas, il est possible d'augmenter la taille du circuit en ré-injectant dans celui-ci certains des appels à des sous-modèles qu'il effectue (si il en existe). Le choix des sous-modèles à ré-injecter est réalisé de la façon suivante :

- Les appels sont classés par ordre décroissant du nombre d'entrées–sorties qu'ils contiennent. L'idée est de favoriser les Re-connections de signaux comportant des terminaux dans différents modèles afin de favoriser les optimisations du circuit basées sur la propagation des valeurs des signaux entre composants.
- Les appels classés sont examinés dans l'ordre. Tant qu'il est possible de ré-injecter un appel sans dépasser pour l'appelant la taille limite `maxLit`, l'appel est ré-injecté.

La procédure est appliquée à l'ensemble de l'arborescence de manière récursive. Deux parcours sont implantés, dans les méthodes suivantes :

- equilibrateDeepFirst: maxLit commence les ré-injections en partant des feuilles de l'arborescence de composants pour aller vers la racine.
- equilibrateBreathFirst: maxLit exécute les ré-injections en effectuant un parcours en largeur d'abord de l'arbre.

Les partitionnement structurels (cas des circuits de taille trop importantes) sont dans ces deux méthodes également effectuées dans le même ordre. Noter que, pour un circuit non hiérarchique, les deux méthodes sont équivalentes, et reviennent à un partitionnement structurel des circuits de taille trop importante (cas de PPart).

Chapitre 3

Synthèse

Nous présentons dans cette partie la modélisation du processus de synthèse d'un circuit, localement au processeur exécutant SMALLTALK (avec SIS, première partie) puis de manière distribuée avec un logiciel de synthèse quelconque en s'appuyant sur une modélisation de PVM sous SMALLTALK.

3.1 Synthèse locale

Nous explicitons dans cette partie les différentes méthodes de synthèse possibles d'une hiérarchie de circuits, à l'aide d'un outil de synthèse externe, SIS.

Les circuits sont manipulés sous forme hiérarchique à l'intérieur du système OO-PPart, mais synthétisés en externe par des outils standard et performants. Le principe est de communiquer via des shells avec les outils de synthèse externes, en leur envoyant les données et les commandes de synthèse sous forme textuelle, de manière interactive. De cette façon, le couplage avec différents outils de synthèse est aisé à réaliser.

3.1.1 L'interface vers SIS

Une seule interface est actuellement implantée, vers SIS [Sa92], comme c'était le cas dans PPart. Le processus externe peut être vu comme une boite noire, qui communique via un pipeline en entrée et en sortie avec SMALLTALK. Pour des raisons de bufférisation des entrées/sorties standard, nous avons été obligés de modifier légèrement le code de SIS (ajout d'un mot-clé `.endoffile` pour marquer la fin d'une lecture de circuit au format BLIF en mode interactif).

L'interface est implantée dans la classe Sis, qui permet de créer un processus SIS (la variable de classe `sisExec` contient l'exécutable) puis de lui envoyer des commandes (en mode textuel) et de recevoir les résultats affichés sur la sortie standard du processus. Toutes les communications sont réalisées en se basant sur la séparation des commandes dans SIS par le prompt `sis>`, intercalés entre l'exécution de chaque commande dans SIS en mode interactif. Ce même mode d'exécution est repris dans tous les outils de synthèse (shell interne) et le même modèle de communication pourrait être utilisé pour créer des interfaces vers ces outils à partir de SMALLTALK. La communication des données (circuits) se fait par lecture sur l'entrée standard des circuits envoyés au format BLIF à partir de SMALLTALK. De même, la récupération des circuits optimisés est fait par affichage sur la sortie standard du processus SIS puis lecture du circuit en mode textuel dans SMALLTALK.

Les méthodes principales sont les suivantes :

- initialize : lancement d'un processus SIS
- release : terminaison du processus
- sendCmd: aText : envoi d'une commande

- rcvResult : réception du texte résultant de la dernière commande envoyée.
- sendGraph: aGraph : envoi d'un circuit (y compris ses sous-modèles, qui seront mis à plat dans SIS).
- rcvGraph : récupération du circuit courant du processus SIS.

Les commandes release, sendGraph: et rcvGraph sont basées sur sendCmd: et rcvResult. Le processus créé est local au processeur exécutant SMALLTALK.

Cette interface est utilisée pour la synthèse hiérarchique des circuits à partir de SMALLTALK. Ce point est explicité dans la partie suivante.

3.1.2 Synthèse d'un nœud de la hiérarchie

La figure 3.1 montre la hiérarchie structurelle d'un circuit simple.

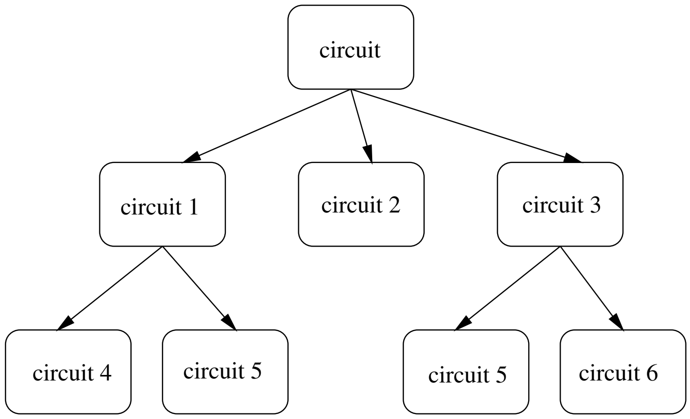

FIG. 3.1 – *La hiérarchie structurelle d'un circuit simple*

Il existe deux possibilités de synthèse partielle de ce circuit (figure 3.2) :

(a) Synthèse d'un nœud et de ses dépendants (mise à plat récursive du nœud, puis synthèse de celui-ci

(b) Synthèse d'un nœud sans ses dépendants : dans ce cas, le circuit doit être restructuré avant synthèse, l'outil SIS n'acceptant pas des circuits partiellement définis.

Dans le premier cas, il suffit d'envoyer le nœud et ses modèles dépendants à l'outil de synthèse. Celui-ci effectuera la mise à plat du circuit, et le résultat de la synthèse sera la version optimisée du sous arbre dont la racine est le composant envoyé. Noter que les sous-modèles exploités par d'autres nœuds sont alors dupliqués (les appels dans le sous-arbre sont ré-injectés et optimisés, les appels hors du sous-arbre restent inchangés).

Dans le second cas, il faut disjoindre les composants du nœud à synthétiser (partie A du nœud) et les appels à des sous-modèles, vus comme des boites noires par l'outil de synthèse (partie B du nœud). La synthèse du nœud correspond à la synthèse de la partie A. Après synthèse, le nœud B et le nœud A' (résultat de la synthèse de A) sont re-fusionnés.

Les méthodes implantées pour la réalisation des deux types de synthèses sont explicitées dans la partie suivante. Le méthodes de synthèse globales du circuit, basées sur les précédentes sont détaillées ensuite. L'ensemble de ces méthodes est disponible dans la catégorie de méthodes *local synthesis* de la classe BLIFModel.

Méthodes de base

Synthèse d'un sous-arbre : (cas (a) de la section précédente).

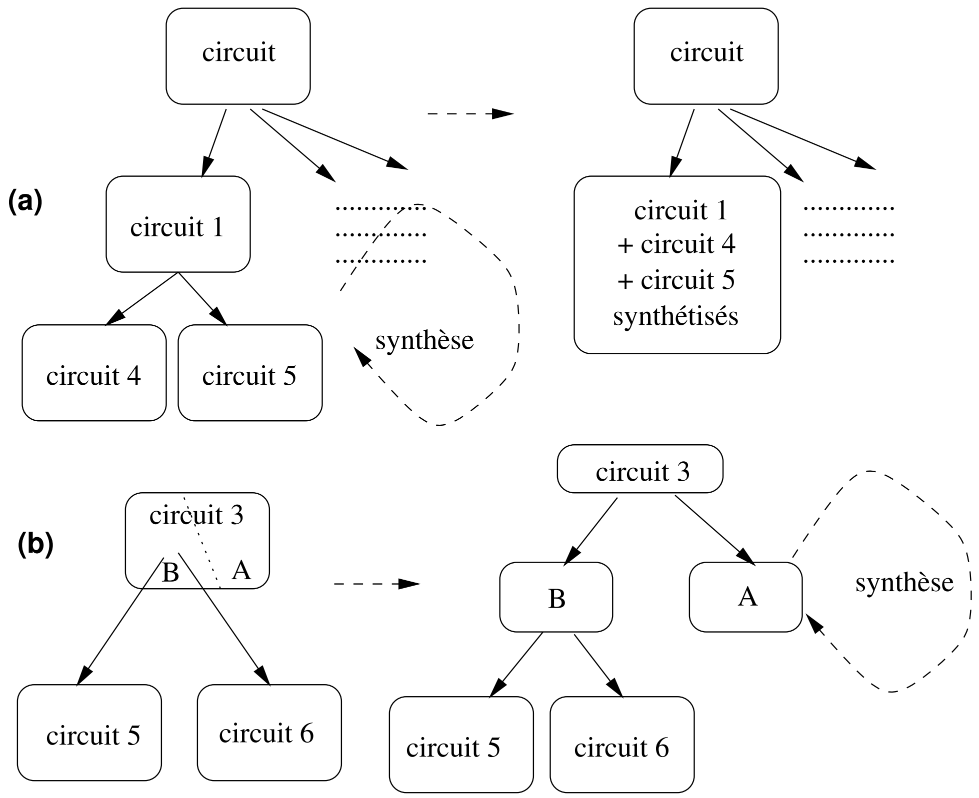

FIG. 3.2 – *(a) synthèse d'un sous-arbre et (b) synthèse d'un nœud seul*

La méthode localOptimize: command, où `command` est un texte représentant une (des) commande(s) de synthèse acceptée par SIS, lance un processus local SIS, lui envoie le circuit receveur (ainsi que les sous-modèles dépendants), effectue la synthèse, puis remplace le *contenu du receveur* par le circuit résultant. Sis le receveur est un sous-arbre d'un circuit englobant, la modification du sous-arbre est donc prose en compte automatiquement dans l'appelant.

La méthode est en fait basée sur un appel à la méthode générique localOptimizeAndReplace: circuits command: cmd, qui synthétise par la commande `cmd` un ensemble de circuits `circuits` en les envoyant un par un à un processus SIS local au processeur exécutant SMALLTALK.

Synthèse un nœud : (cas (b) de la section précédente).

La méthode localOptimizeLevel0: command, où `command` est un texte représentant une (des) commande(s) de synthèse acceptée par SIS, effectue le partitionnement du circuit receveur en partie hiérarchique et composants non-hiérarchiques, synthétise la partie non-hiérarchique, puis ré-injecte celle-ci et la partie hiérarchique dans le receveur. Là encore, si le circuit receveur correspondant à un appel dans un ou plusieurs circuits parents dans une hiérarchie englobante, les modifications sont prises en compte car le contenu du receveur est remplacé par le résultat de la synthèse.

La méthode de synthèse localOptimize: command est utilisée pour la synthèse de la partie non-hiérarchique, elle-même produite à partir du receveur par la méthode disconnectAll (catégorie de méthodes *masking*) qui effectue le partitionnement du circuit.

Synthèse d'une hiérarchie complète

La méthode localOptimizeByLevel: cmd optimise chaque nœud de la hiérarchie de manière indépendante, sans synthétiser plusieurs fois les modèles appelés à de multiples reprises.

Une extension de cette méthode est la méthode localOptimizeByLevel: cmd maxSize: maxLit qui, pour les nœuds de taille (coût SOP) supérieur à la borne `maxLit`, effectue un partitionnement structurel du nœud et synthétise les parties obtenues de manière indépendante. Les parties sont ré-injectées dans le circuit final, qui présente donc la même hiérarchie que le receveur initial du message. Là encore, le circuit receveur est remplacé par sa version optimisée.

Un exemple typique de synthèse consiste à équilibrer une hiérarchie, puis à en synthétiser chaque nœud :

```
| circuit |
  circuit := BLIFModel fromFile: 'essai.blif'.
  circuit equilibrateDeepFirst: 5000.
  circuit localOptimizeByLevel: 'source script'
```

Un second exemple : pour un circuit séquentiel dont on cherche à synthétiser les parties combinatoires (le `source script` correspond à une optimisation logique des parties combinatoires d'un circuit) :

```
| circuit |
  circuit := BLIFModel fromFile: 'essai.blif'.
  circuit combAndSeqHierarchy.
  circuit localOptimizeByLevel: 'source script' maxSize: 5000
```

3.1.3 Liaison avec une librairie

Les composants d'un circuit peuvent correspondre à des portes logiques ou à des portes provenant d'une librairie (voir partie 2.1).

Les classes Genlib et GenlibLatch modélisent des composants de librairie d'après le format GENLIB exploité dans SIS. Un circuit comportant des composants de librairie (classes BLIFMap et BLIFMapLatch) peut être lié à la librairie correspondante : la variable `library` contient éventuellement pour un circuit une collection de portes de librairie. Si tel n'est pas le cas, les composants de librairie restent simplement repérés par leur nom. La méthode codeName appliquée à un composant d'un circuit renvoie, suivant le cas le composant de librairie si il existe, ou son nom si le circuit n'est pas lié à une librairie.

Il n'est pas possible dans SIS de lire de manière interactive le contenu d'une librairie : celle-ci doit être chargée à partir d'un fichier. C'est pourquoi lors de la liaison d'un circuit avec une librairie, le nom du fichier d'où est issue la description de la librairie est stockée en même temps que les composants de la librairie eux-mêmes.

Nous présentons tout d'abord les méthodes de liaison entre un circuit et une librairie avant de montrer comment la conversion technologique vers une telle librairie est possible.

Attachement d'une librairie GENLIB

La classe GenlibParser permet d'analyser le contenu d'un fichier au format GENLIB et retourne une liste de composants de librairie (un GenlibSet), avec notamment des informations sur la surface et les délais des portes et bascules extraites du fichier. Cet analyseur, écrit avec l'interface *Lex & Yacc* développée sous SMALLTALK, est exploité pour lier un circuit à une librairie grâce à la méthode setLibraryFromFile: aFileName. Cette méthode permet de lier un modèle et tous ses sous-modèles à la librairie lue. La méthode linkToLibrary (resp. linkAllToLibrary) permet ensuite de relier individuellement chacun des composants du circuit (resp. et de ses sous-modèles) à la porte ou la bascule correspondante (*via* les noms des portes) de la librairie utilisée.

Une fois lié à une librairie, les méthodes suivantes sont disponibles pour examiner le circuit :

- mappedSize, ownMappedSize, le nombre de portes utilisés par le nœud et ses descendants ou son nombre de portes propre.
- area, ownArea, les surfaces correspondantes (d'après les informations technologiques stockées pour chaque porte de la librairie).
- oneOfEachMap renvoie une collection des portes de la librairie utilisées pour la conversion technologique du circuit.

Il est donc possible de pondérer les composants d'un circuit converti vers une technologie cible avec les informations de surface incluses dans la librairie choisie, pour ensuite découper (partitionnement structurel) en parties égales (par exemple pour l'affectation des parties à plusieurs ressources programmables de capacité unitaire fixe).

Synthèse avec conversion technologique

Lorsque un circuit comporte des éléments correspondant à des portes de librairie, celle-ci doit être chargée dans SIS avant la lecture du circuit. Cela est réalisé automatiquement si le circuit a été lié à une librairie dans SMALLTALK avant la demande de synthèse. De même, un circuit résultant d'une conversion technologique est relu dans SMALLTALK en préservant les informations de conversion.

Même si il est possible de changer de librairie durant la synthèse, aucune vérification n'est faite sur ce point dans l'environnement OO-PPart. Cela peut donc mener à des incohérences de données. Il est donc recommander de lier le circuit à une et une seule librairie, gardée durant tout le processus de synthèse du circuit.

nota : *Les transferts de librairie sont, contrairement aux transferts de circuits, réalisés par fichier entre* SMALLTALK *et* SIS *(*SIS *ne charge une librairie* GENLIB *que à partir d'un fichier, et ne peut le faire en mode interactif : les pipelines ne sont donc pas utilisables). Il est donc préférable, lors du chargement de la librairie, d'indiquer le chemin absolu vers son fichier descriptif. En cas de synthèse en parallèle (voir la partie 3.2 sur la synthèse en parallèle), il est nécessaire que ce chemin corresponde à un disque partagé via* NFS*, ou que le fichier soit disponible dans l'environnement de synthèse* SIS*.*

Exemple : Le code ci-dessous montre comment réaliser en deux phases successives la synthèse d'un circuit avec conversion technologique.

```
| circuit |
circuit := BLIFModel fromFile:'wallace32.blif'.
circuit setLibraryFromFile: '/Library/lib2.genlib'.

circuit equilibrateDeepFirst:7000.
circuit optimizeByLevel: 'sweep; map -m 0; phase -g' maxSize:2500.

circuit optimizeByLevel: 'phase -g' maxSize:4000.

circuit printBlifToFile: 'mapped.blif'.
circuit printLibraryToFile: 'lib.genlib'.
Transcript show: circuit area printString; cr.
```

La première phase, de conversion proprement dite, est réalisée sur des parties de petite taille (SOP de 2500 max). La seconde, de post-optimisation est nettement moins coûteuse en temps CPU, Elle est donc réalisée avec un partitionnement du circuit à plus gros grain.

La synthèse du circuit par nœud est aisement parallélisable, en distribuant les tâches de synthèse sur réseau de stations. La partie suivante présente le mécanisme de parallélisation implanté dans OO-PPart, ainsi que les différents algorithmes (basés sur diverses sources de parallélisation possibles liées à la nature hiérarchique des circuits traités).

Librairie par défaut Il est possible de lier un circuit à une librairie standard par défaut. La méthode mapToDefaultLibrary lie le circuit à la librairie par défaut et réalise la conversion technologique vers cette librairie. Le fichier GENLIB de la librairie par défaut est accessible à l'aide de la méthode defaultLibraryName de la catégorie *library* des méthodes d'instance de BLIFModel. C'est actuellement `$OOPPART/library/minimal.genlib` avec les portes suivantes :

```
GATE ZERO 0 O=CONST0;
GATE ONE 0 O=CONST1;
GATE NAND2 2 O=!(a*b); PIN * INV 1 999 1.0 0.2 1.0 0.2
GATE NOR2 2 O=!(a+b); PIN * INV 1 999 1.0 0.2 1.0 0.2
GATE AND2 2 O=in0*in1; PIN * INV 1 999 1.0 0.2 1.0 0.2
GATE AND3 2 O=in0*in1*in2; PIN * INV 1 999 1.0 0.2 1.0 0.2
GATE AND4 2 O=in0*in1*in2*in3; PIN * INV 1 999 1.0 0.2 1.0 0.2
GATE AND5 2 O=in0*in1*in2*in3*in4; PIN * INV 1 999 1.0 0.2 1.0 0.2
GATE BUF1 1 O=a; PIN * INV 1 999 0.9 0.3 0.9 0.3
GATE INV1 1 O=!in0; PIN * INV 1 999 0.9 0.3 0.9 0.3

# D-type latches and flip-flops are necessary for sequential
# technology mapping.
# WARNING: area and delay parameters are arbitrary.
LATCH DFF 10 Q = D;
PIN D NONINV 2 999 1.0 0.1 1.0 0.1
SEQ Q ANY RISING_EDGE
CONTROL CLOCK 2 999 1.0 0.1 1.0 0.1
CONSTRAINT D 0.1 0.1

LATCH DLATCH 8 Q = D;
PIN D NONINV 2 999 1.0 0.1 1.0 0.1
SEQ Q ANY ACTIVE_HIGH
CONTROL CLOCK 2 999 1.0 0.1 1.0 0.1
CONSTRAINT D 0.1 0.1
```

3.2 Parallélisation des opérations

Le but principal de PPart est d'accélérer le processus de synthèse des circuits de taille importante en extrayant du parallélisme de ce processus. La parallélisation est basée sur un partitionnement structurel des données représentant le circuit. Le problème principal réside dans la perte d'optimalité des résultats engendrée par la découpe en sous-circuits.

Avec le système OO-PPart, les circuits à traiter sont déjà pré-partitionnés : la hiérarchie fournit un partitionnement *naturel*, pré-défini. Ce partitionnement, dû en général à la manière dont le circuit a été spécifié à haut niveau (outils de synthèse architecturaux) puis compilé au niveau RTL, est lui aussi source de perte d'optimalité si chacun des composants est traité de manière indépendante par les outils de synthèse. Cependant, cette perte d'optimalité peut être moins importante que celle engendrée avec PPart (ce point, intuitif, demanderait à être vérifié). La nature hiérarchique des circuits est une source naturelle de parallélisme dans le processus de synthèse. Il peut être combiné, dans OO-PPart, avec une approche par partitionnement structurel comme celle exploitée dans PPart.

Nous nous intéressons dans cette partie à la parallélisation du processus de synthèse, toujours avec une approche par partitionnement de circuits, similaire à celle de PPart. Cependant les possibilités de partitionnement prennent en compte la nature hiérarchique des circuits.

Nous présentons tout d'abord le mécanisme de synthèse parallèle implanté dans l'environnement SmallTalk. La parallélisation est basée comme dans PPart sur l'exploitation de la librairie pvm [GBD+94]. Une interface de l'environnement objet vers pvm a été mise en place. Elle est explicitée dans la première partie. Cette interface est indépendante de OO-PPart et peut être exploitée pour la parallélisation d'autres processus à partir de SmallTalk.

Les différentes techniques de parallélisation sont mises en œuvre dans la classe PPart de l'environnement OO-PPart. Cette classe fait l'objet de la seconde partie. Nous examinons également dans cette partie comment le déroulement du processus de synthèse en parallèle est réalisé.

La troisième partie est consacrée aux algorithmes de haut niveau développés en se basant sur les mécanismes explicités dans la partie précédente. Les algorithmes développés correspondent en fait à la parallélisation de la version locale de l'interface de synthèse présentée dans la partie 3.1.

Enfin, dans la dernière partie, nous explicitons une généralisation de l'interface de synthèse de OO-PPart pour l'application d'outils de synthèse quelconques : le mécanisme de mise en œuvre de processus sis à distance est généralisé et appliqué à l'utilisation de bop, l'outil de synthèse logique du logiciel Alliance.

3.2.1 Interface pvm

pvm [GBD+94] est un environnement de développement d'applications parallèles basé sur le modèle des *processus communicants*. Une application est composée d'un ensemble arbitraire de processus, exécutés par un ensemble de processeurs (éventuellement un processeur unique) constituant une *machine virtuelle*.

La librairie d'interface contient un ensemble de fonctions permettant la gestion des processus (création, destruction), la gestion des hôtes (ajout, destruction, arrêt de la machine virtuelle), et le passage de messages entre processus. Une application est exécutée après lancement d'une machine virtuelle définissant les hôtes (stations, calculateurs parallèles, etc) disponibles pour la machine virtuelle.

L'interface SmallTalk vers pvm reprend les fonctionnalités de gestion de ressources matérielles et de processus. Elle permet également les communications de messages entre SmallTalk et les processus créés (mais pas la communication inter-processus, qui est à la charge du programmeur des processus. Un processus est créé par exécution d'un exécutable sous Unix.

La première partie est consacrée aux classes mises en œuvre pour la modélisation objet des hôtes, processus et communications de/vers SmallTalk des processus créés. L'exécution et la coordination entre processus est assurée par un gestionnaire, lui-même modélisé par une classe SmallTalk. L'implantation réalisée, notamment en ce qui concerne le passage de messages vers les processus à partir de SmallTalk est également détaillée dans cette partie.

La seconde partie liste les principales méthodes disponibles, et en donne un exemple complet d'utilisation.

Modélisation de pvm sous SmallTalk

pvm est modélisé sous SmallTalk à l'aide de trois classes :

- La classe PvmManager est le gestionnaire de processus et de machine virtuelle pvm,
- La classe PvmMsg modélise les messages,

– La classe PvmTask modélise les processus.

Le lancement d'une machine virtuelle, l'exécution de processus, et la communication de messages entre SMALLTALK (une instance de PvmManager) sont réalisés en passant par l'intermédiaire d'un programme C, **pvmManager** qui est le premier processus exécuté par la machine virtuelle (et qui de fait lance celle-ci). La communication entre ce gestionnaire externe et le gestionnaire interne à SMALLTALK est réalisé par un pipeline d'entrée-sortie sur les I/O standard du processus externe. Le processus de communication est explicité sur la figure 3.3.

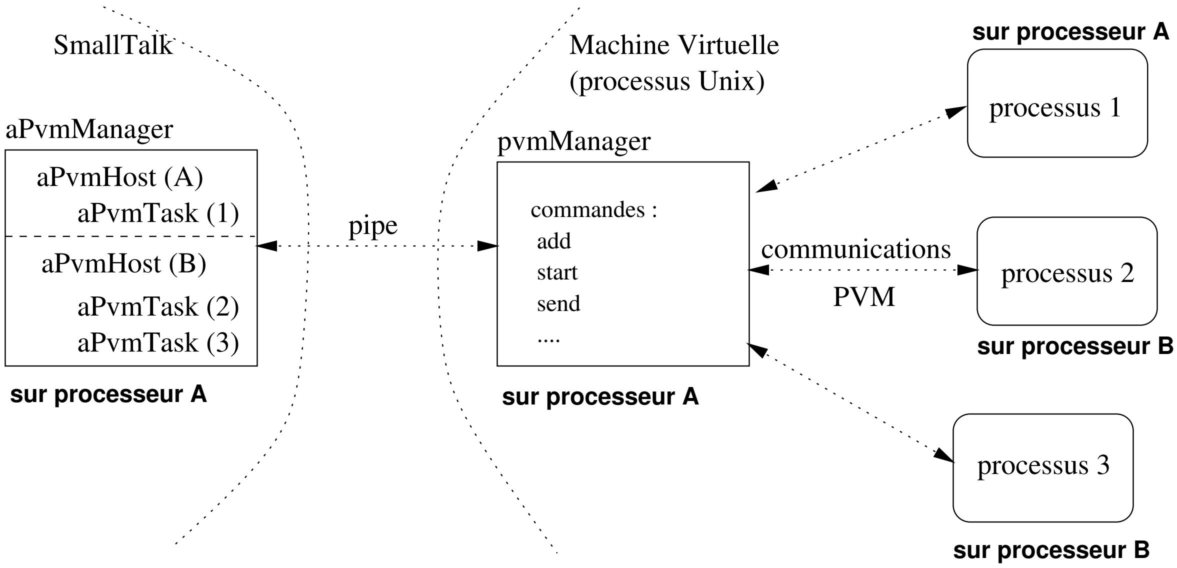

FIG. 3.3 – *Les différents constituants de l'interface* SMALLTALK / PVM

Le protocole de communication entre les deux processus (SMALLTALK et le processus gestionaire Unix) comprend les commandes suivantes (dont les arguments ne sont pas détaillés ici) :

- `start` pour démarrer la machine virtuelle. Au départ, celle-ci n'est composée que du processeur local.
- `stop` pour arrêter la machine virtuelle (ce qui termine tous les processus s'exécutant sur celle-ci).
- `add` pour ajouter une machine à la machine virtuelle (existante).
- `delete` pour supprimer une machine
- `kill` pour stopper un processus particulier
- `spawn` pour exécuter un processus (le programme à exécuter est donné en argument).
- `send` pour envoyer des données à un processus cible.
- `receive` pour recevoir des données d'un processus cible.

Les communications de/vers le gestionnaire interne (classe PvmManager) sont typées par un entier quelconque (comme c'est le cas dans PVM), ce qui permet de discriminer les messages envoyés et reçus.

Chacun des processus est modélisé en interne par une instance de PvmTask. Lors du lancement d'un processus, la machine hôte utilisée et les informations sur le processus sont stockées dans l'objet représentant celui-ci. Les messages sont de même modélisés en interne à SMALLTALK (classe PvmMsg). Toutes les opérations sur les processus et les communications sont traduites en commandes suivant le protocole décrit ci-dessus puis envoyées pour exécution au processus gestionnaire externe.

Méthodes disponibles

Les principales méthodes implantés pour le gestionnaire SMALLTALK sont les suivantes :

- add: aHostName et delete: aHostName : ajout et retrait d'un processeur à/de la machine virtuelle

- start et stop : démarrage et arrêt de la machine virtuelle
- spawn: aTask : lancement d'une tâche, caractérisée par son exécutable et éventuellement une machine hôte
- kill: aTask : arrêt d'une tâche
- sendMsg: aMsg : envoi d'un message, caractérisé par son type (un entier), son destinataire task et son contenu text, une suite d'octets.
- receiveMsg: aMsg : reçoit un message de la tâche stockée dans aMsg (ou de n'importe quelle tâche si le champ est nil) et stocke le résultat obtenu dans le champ text de aMsg. Le message reçu est du type de aMsg ou de n'importe quel type si le type de aMsg est -1.

Exemples Les exemples présentés ici montrent comment la synthèse parallèle de circuit est implantée dans OO-PPart (classe PPart). Le processus parallèle (variable manager de classe PvmManager) est composé d'un ensemble de tâches, une par processeur disponible. Les tâches correspondent à l'exécution d'un SIS distant *via* l'exécutable sisExec.

• L'ajout d'une machine pour la synthèse :

```
add: aHost
   | t |
   self manager add: aHost.
   t := PvmTask name: SisFork manager: self manager host: aHost.
   self manager spawn: t.
```

• L'envoi d'une commande au processus SIS, suivi de la réception d'un résultat :

```
sendCmd: aCmd to: aTaskOrHost
   | m t |
   m := PvmMsg new.
   m type: 1.  "Sis command"
   m text: aCmd , '
'.
   t := aTaskOrHost class == PvmTask ifFalse: [self taskByHost: aTaskOrHost]
            ifTrue: [aTaskOrHost].
   m task: t.

   self manager sendMsg: m

rcvResultFrom: aTaskOrHost
   | m t |
   m := PvmMsg new.
   m type: 2.  "Sis result"
   t := aTaskOrHost class == PvmTask ifFalse: [self taskByHost: aTaskOrHost]
            ifTrue: [aTaskOrHost].
   m task: t.
   self manager receiveMsg: m.

   ^m text
```

• La récupération du circuit courant utilisé par un processus SIS :

```
rcvGraphFrom: aTaskOrHost
   self sendCmd: 'write_blif -n' to: aTaskOrHost.
   ^BLIFParser fromText: (self rcvResultFrom: aTaskOrHost)
```

Comme le montrent ces exemples, les communications avec un processus de synthèse sont très simples à réaliser en utilisant l'interface de haut niveau. Celle-ci, vu les limitations sur le passage de messages entre processus externes est plutôt destinée à la mise en œuvre d'algorithmes de type maître–esclave, avec des communications uniquement entre le maître et les esclaves, sans communications transversales entre esclaves.

3.2.2 Interface vers SIS distant

Comme explicité dans la partie précédente, l'interface PVM fonctionne en créant des processus Unix, et en communiquant avec eux. Ces processus doivent exploiter la librairie PVM pour réaliser les échanges de messages.

Nous souhaitons réaliser une interface permettant d'exécuter SIS de manière transparente sur une machine distante, avec une interface similaire à celle mise en œuvre pour l'exécution locale de processus SIS (voir partie 3.1). Dans l'interface locale, les échanges de données sont réalisés *via* un pipe bidirectionnel, avec une exécution de SIS en mode interactif : cela permet de voir le processus SIS comme une *boite noire*, et de l'interfacer sans modification du code interne de l'outil d'optimisation[1].

Les composants de l'interface vers un processus SIS distant sont illustrés par la figure 3.4 :

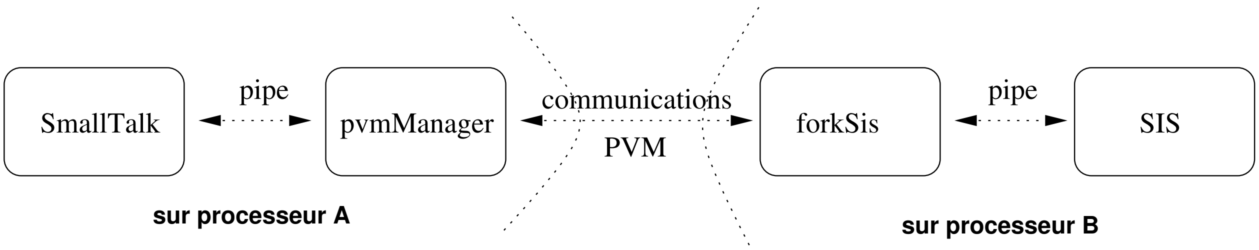

FIG. 3.4 – *Exécution d'un processus* SIS *en mode interactif sur un processeur distant à partir de* SMALLTALK

On retrouve les éléments de la figure 3.3 dans la figure 3.4 :
- le gestionnaire interne à SMALLTALK,
- le gestionnaire externe, exécuté sur le même processeur **A**,
- un processus distant (**forkSis** dans le cas présent) sur le processeur **B**,
- et s'ajoute le processus exécutant SIS en mode interactif, communiquant *via* un pipe sur les I/O standard avec le processus **forkSis**, situé sur le même processeur **B**.

forkSis est un programme écrit en C, chargé de transmettre, à l'aide de primitives PVM les entrées/sorties interactives du processus SIS auquel il est associé (et qu'il lance par un *fork()*). Il permet également la terminaison du processus SIS. Il reçoit répétivement les commandes SIS à exécuter, les transmet via l'entrée standard au processus SIS, puis renvoie les résultats associés (qui doivent d'ailleurs être consommés même si ils sont sans signification).

3.2.3 Synthèse en parallèle

Nous avons examiné dans les parties précédentes les différents points nécessaires à la mise en œuvre d'un outil de synthèse en parallèle de circuits hiérarchiques :
- la modélisation de la hiérarchie, les opérations de manipulation et de synthèse nœud par nœud de cette hiérarchie (pour l'instant localement au processeur exécutant SMALLTALK)
- l'exécution à distance d'un outil de synthèse

Le système OO-PPart, implanté dans la classe PPart associe un circuit hiérarchique avec une application distribuée de synthèse pour réaliser la synthèse distribuée de ce circuit. Nous présentons tout d'abord l'algorithme de synthèse distribuée, similaire à celui implanté dans PPart, puis les politiques de synthèse possibles pour un circuit hiérarchique.

1. c'est pas vrai : petite modification de source dûe à un problème de bufférisation des entrées/sorties dans SIS

Algorithme distribué

La classe PPart regroupe un gestionnaire de la classe PvmManager chargé du lancement de processus SIS en mode interactif (voir partie précédente) et un circuit hiérarchique. Une instance de la classe est obtenue soit à partir du circuit, soit en faisant appel à la classe elle-même :

```
unCircuit ppart
"ou :"
PPart on: unCircuit
```

Pour utiliser l'algorithme de manière distribuée, il faut initialiser correctement son gestionnaire PvmManager (ajout de machines, démarrage de la machine virtuelle), en lui envoyant directement les messages d'ajout et de démarrage. Ces messages, retransmis au gestionnaire, lance un processus SIS sur chaque machine exploitée. Ces processus peut être vus comme des démons, attendant des demandes de calcul à effectuer (SIS interactif). Une fois prête, la machine virtuelle peut être exploitée pour la synthèse des nœuds du circuit.

L'algorithme d'ordonnancement utilisé est un algorithme *round robin* classique : pour un ensemble de n circuits à synthétiser sur $m \leq n$ machines, les m premiers circuits sont envoyés sur un processeur différent, puis l'algorithme se met en attente des circuits optimisés. Dès qu'un circuit est reçu, un nouveau circuit est envoyé pour synthèse au processeur rendu disponible. L'algorithme termine lorsque tous les circuits ont été optimisés. Il renvoie ces versions optimisées.

La méthode implantant cet algorithme d'ordonnancement est située dans la catégorie de classe *parallel* de la classe PPart (optimize: aSetOfCircuits command: aCommand). Une version légèrement différente remplace de plus les circuits originaux par leur version optimisée dans la hiérarchie associée à l'instance de PPart.

Méthodes de haut niveau

On retrouve dans la version parallèle de l'algorithme de synthèse de circuits hiérarchique les mêmes méthodes de synthèse que celles développées dans l'interface locale vers SIS (partie 3.1).

En ce qui concerne les envois/réception de données, les messages correspondent au exemples de la partie 3.2.1. Pour ce qui est des algorithmes de synthèse globale, les méthodes de la partie 3.1 sont redéfinies (sans le préfixe local, les méthodes locales étant également directement accessibles dans la classe PPart pour la synthèse locale du circuit associé) :

- optimizeByLevel: cmd
- optimizeByLevel: cmd maxSize: size

voir la partie 1.2 pour un exemple complet de synthèse en parallèle.

3.2.4 Généralisation

L'utilisation d'un logiciel de synthèse en parallèle fait appel aux éléments suivants :

- Des procédures de lecture et d'écriture des circuits
- Un programme C d'interface entre le logiciel et PVM
- Des méthodes de lancement de tâches correspondant à l'exécution du programme C ci-dessus à partir de SMALLTALK

L'algorithme *round-robin* (section 3.2.3) de synthèse en parallèle peut être réécrit pour faire appel aux méthodes spécifiques à la synthèse à l'aide d'un outil donné, avec utilisation des méthodes d'entrée/sortie appropriée pour les circuits à synthétiser.

Nous présentons tout d'abord le mécanisme d'interfac cage à mettre en œuvre pour un outil quelconque, avant de détailler l'exemple de l'interface vers bop, l'outil de synthèse logique d'Alliance. En ce qui concerne, l'interface vers SIS, celle-ci a également été réécrite en respectant l'interface explicitée ci-dessous.

Spécialisation vers un outil (*framework*)

Nous avons choisi de spécialiser l'interface pour un outil donné en créant des sous-classes de la classe PvmTask pour chacun des outils possibles. Chaque sous-classe implante des messages d'instance suivants pour spécifier les méthodes propres à l'outil de synthèse auquel elle est destinée :

- Procédures de lecture des circuits :
  - rcvObjectCmd : le texte de la commande qui permet de provoquer l'écriture du circuit à partir du logiciel de synthèse.
  - rcvObjectParser : la classe de l'analyseur syntaxique à utiliser pour créer un objet SMALLTALK représentant le circuit (par exemple un BLIFModel) à partir de sa représentation textuelle. Le parser devra répondre au message fromText: aText pour créer l'objet SMALLTALK à partir de sa représentation textuelle.
- Procédure d'écriture des circuits : sendObjectCmd: anObject, qui renvoie le texte correspondant à la commande de lecture à exécuter par le logiciel de synthèse externe pour charger le circuit.
- Méthode de lancement de tâches à partir de SMALLTALK : nous avons choisi d'implanter pour chaque logiciel interfacé 2 méthodes dans la classe PPart (et non pas dans la sous-classe correspondant au logiciel) :
  - startXXX : le lancement d'une machine virtuelle spécialisée, avec un processus correspondant à l'exécution d'une tâche de classe XXX [2].
  - addXXX: aHost : ajout 'a une machine virtuelle d'une machine, avec un processus correspondant à l'exécution d'une tâche de classe XXX [3].
- Programme C d'interface entre le logiciel et PVM : celui-ci permet l'interprétation des commandes de lecture/écriture, et l'exécution du logiciel de synthèse sur le circuit courant.

Interface vers bop

bop est le logiciel de synthèse logique intégré à Alliance.

Modélisation de l'outil bop s'utilise basiquement de la manière suivante :
`bop -o fileIn.vhdl fileOut.vhdl` où `fileIn.vhdl` est le fichier à optimiser, au format VHDL, et `fileOut.vhdl` est le résultat produit après optimisation.

Contrairement à SIS, il ne s'utilise pas en mode interactif, et la seule commande exécutable est le lancement de bop tel que décrit ci-dessus. Pour respecter le modèle d'utilisation des outils décrit dans la section précédente, il est nécessaire d'introduire la notion de circuit courant. Cela est réalisé au niveau du programme C d'interface. L'écriture à partir de SMALLTALK d'un circuit correspond à la production dans un fichier fixe (local au processus de synthèse) d'une description VHDL du circuit. L'importation dans SmallTalk d'un circuit produit par bop correspond à la lecture du fichier résultat produit par bop dans un fichier par défaut. La seule commande de synthèse est bop, et elle s'applique au circuit courant, produisant sa sortie dans le fichier de sortie par défaut.

Modélisation des circuits Le format VHDL utilisé dans bop est légèrement différent du format VHDL de sortie VHDL de la classe BLIFModel (voir la section 4.1). Une sous-classe de la classe BLIFModel, AlcModel a donc été implantée, uniquement pour permettre l'écriture au format VHDL accepté par bop des circuits de classe BLIFModel. Cette classe se contente de redéfinir quelques messages de production VHDL pour se conformer au VHDL accepté par bop. L'analyseur lexical associé a également été implanté, dans la classe VBEParser, sous-classe de VHDLParser. VBEParser redéfinit simplement

2. actuellement: startSis, startAlc, startVpr
3. actuellement: addSis: h, addAlc: h, addVpr: h

l'extension associée aux fichiers VHDL en `.vbe`. Les outils de conversion standard de VHDL vers BLIFModel sont utilisables pour récupérer sous SMALLTALK un objet de la classe BLIFModel à partir d'une description VHDL (voir la section 4.1). Les conversions BLIFModel de/vers AlcModel d'un circuit se font à l'aide des messages asBLIFModel et asAlcModel implantés dans les deux classes.

Création de l'interface vers bop Nous rappelons que pour pouvoir utiliser l'algorithme distribué décrit dans la section 3.2.3, il faut redéfinir, dans une sous-classe (a) de PvmTask, les messages (b) rcvObjectCmd, (c) rcvObjectParser, (d) sendObjectCmd: anObject et (e) écrire le programme C d'interface correspondant à l'outil. Des messages spécialisés (f) sont ensuite à ajouter dans la classe PPart pour lancer les tâches correspondant à l'outil.

Les différentes étapes sont décrites ci-dessous.

(a) La classe est AlcTask, sous-classe de PvmTask.

(b)

```
rcvObjectCmd
   ^'write_vhdl'
```

La commande correspond à une demande d'écriture (`write_vhdl`) pour le programme C externe d'interface vers bop.

(c)

```
rcvObjectParser
   ^VBEParser
```

(d)

```
sendObjectCmd: aModel
| st cmd |

st := WriteStream on: ''.
aModel asAlcModel printVhdlOn: st.
st nextPutAll: '
'.
cmd := 'read_vhdl
' , st contents.
^cmd
```

La commande correspond à une demande de lecture (`read_vhdl`) pour le programme C externe d'interface vers bop.

(e) L'algorithme (simplifié) correspondant au programme C (`alcFork.c`) est le suivant :

```
Tant Que commande recue != 'FIN'
  Si commande == 'read_vhdl'
    copier dans le fichier par defaut AlcIN.vbe le VHDL recu ensuite
  Si commande == 'write_vhdl'
    renvoyer le contenu du fichier AlcOUT.vbe
  Si commande == 'bop'
    executer 'bop -o AlcIN.vbe AlcOUT.vbe'
Fin Tant Que
```

(f) Les messages d'instance de la classe PPart pour lancer une machine virtuelle avec une tâche de synthèse bop par nœud :

```
startAlc
  |t|
  self manager start.
  t := AlcTask new manager: self manager.
```

```
  self manager spawn: t.

addAlc: aHost
  self add: aHost with: AlcTask new.
```

Un exemple d'utilisation

```
| ppart design |

  design := BLIFModel fromFile: 'hierarchie.blif'.
  ppart := PPart on: design.
  ppart startAlc.
  ppart addAlc: 'anotherHost'.
  ppart addAlc: 'aThirdHost'.
  ppart optimizeByLevel: 'bop' maxSize: 1000.
  ppart stop.
  ppart circuit inspect.
```

Chapitre 4

Interfaces

Nous présentons dans cette partie la modélisation VHDL et EDIF d'un circuit et, les mécanisme de lecture et d'écriture associés. La dernière section est consacrée à l'interface de visualisation des circuits hiérarchique, qui fonctionne avec les différentes modélisations possibles pour le circuit.

4.1 Interface VHDL

Une interface de lecture de VHDL structurel a été développée pour permettre la lecture de circuits produit au niveau RTL à partir d'outils de haut niveau. Cette interface permet la lecture d'un sous-ensemble de VHDL correspondant à des descriptions structurelles ou à des descriptions comportementales au niveau logique d'équations booléennes. Le code VHDL peut être réparti dans plusieurs fichiers et être hiérarchiques, avec des appels à des composants externes. Les appels de composants peuvent comporter des paramètres génériques. Les descriptions peuvent également comporter des boucles et des instructions conditionnelles. L'ensemble de classes utilisées pour modéliser le sous-ensemble VHDL considéré permet également la production de fichiers hiérarchiques au format BLIF. Cela autorise les entrées à partir de descriptions VHDL dans l'outil de synthèse en parallèle OO-PPart.

Nous décrivons dans la première partie le sous-ensemble VHDL lisible par l'interface. Le mécanisme de traduction du VHDL structurel vers le format BLIF (et donc la modélisation SMALLTALK de OO-PPart) est abordé ensuite. Enfin nous listons les méthodes de lecture/écriture à partir de fichiers VHDL dans la dernière partie.

4.1.1 Sous-ensemble VHDL

Nous ne détaillons pas ici de manière exhaustive le sous-ensemble VHDL analysable à partir de l'interface SMALLTALK. Nous donnons plutôt des exemples significatifs, qui permettent d'appréhender les possibilités de l'analyseur VHDL. Celui-ci a été implanté à l'aide l'interface vers *Lex & Yacc*. Il lit un ensemble de fichiers VHDL (structurel et fonctionnel pour les descriptions au niveau RTL de fonctions booléennes) possédant l'extension `.vhd`. Le circuit complet est reconstitué en interne à l'aide de classes de stockage des informations VHDL. Le circuit doit être complètement spécifié par l'ensemble des fichiers.

• **Circuit**
Chaque fichier contient la description d'un composant, avec la définition de l'entité associée, les déclarations des composants externes utilisés, d'éventuelles librairies, des signaux locaux, et la description structurelle du circuit (ou fonctionnelle si il s'agit d'équations booléennes).
Exemple :

```
library ieee;
use ieee.std_logic_1164.all;
```

```
entity wal5 is
port(a : in std_logic_vector(4 downto 0);
     s : out std_logic_vector(2 downto 0));
end wal5;

architecture struct of wal5 is
component wal3
port (a : in std_logic_vector(2 downto 0);
      s : out std_logic_vector(1 downto 0));
end component;

signal p0 : std_logic_vector(1 downto 0);
signal p11 : std_logic;
signal zero : std_logic:='0';

begin
w0 : wal3 port map(a=>a(2 downto 0), s=>p0);
w1 : wal3 port map(a(0)=>p0(0),
                   a(2 downto 1)=>a(4 downto 3),
                   s(1)=>p11,
                   s(0)=>s(0));
w2 : wal3 port map(a(0)=>p0(1),
                   a(1)=>p11,
                   a(2)=>zero,
                   s(1)=>s(2),
                   s(0)=>s(1));
end struct;
```

• **Descriptions fonctionnelles au niveau RTL**
Les descriptions RTL lisibles par l'analyseur correspondent à des équations boolénnes.
Exemple :

```
entity fa is
port(a0,a1,a2   : in std_logic;
     s,c        : out std_logic
    );
end fa;

architecture beh of fa is
begin
  s <= a0 xor a1 xor a2;
  c <= (a0 and a1 and a2) or
       (a0 and a1) or (a0 and a2) or (a1 and a2);
end beh;
```

Les opérateurs booléens possibles sont : `and or xor not` .

• **Composants génériques**
L'analyseur VHDL peut prendre en compte des paramètres génériques lors de la lecture de composants et d'appels à des composants (uniquement des variables numériques de type `positive`). Les arguments numériques sont utilisables dans toute la définition structurelle du composant.
Exemple :

```
...
architecture struct of wal5 is
component wal3
port (generic vgen : positive;
```

```
                a     : in std_logic_vector(2 downto 0);
                s     : out std_logic_vector(1 downto 0));
end component;
...

begin
...
w1000 :   wal3 port map(vgen=>i+3, a=>p0, s(0)=>s(0), s(1)=>p0(1));
...
end struct;
```

• Boucles
Les instructions itératives imbriquées sont possibles. Elles peuvent être incluses – ou contenir – n'importe quelle instruction VHDL acceptée par l'analyseur. Les variables de boucle sont utilisables dans toute expression arithmétique. Les variables sont visibles dans le corps de la boucles, et peuvent être réutilisées dans d'autres contextes.
Exemple :

```
begin
...
bclExterne : for i in 200 to 202 generate
   bclInterne : for j in i+100 to 303 generate
    w1000 :   wal3 port map(a(10000)=>p(i+j),
                            a(i-100 downto j)=>a((j+4) downto 3));
   end generate bclInterne;
end generate bclExterne;
...
end struct;
```

Les expressions arithmétiques peuvent exploiter les fonctions suivantes (dans les boucles, mais également dans les autres contecxtes appropriés) : `+ - * / log **` .

• Instructions conditionnelles
Les instructions conditionnelles imbriquées sont possibles. Elles peuvent être incluses – ou contenir – n'importe quelle instruction VHDL acceptée par l'analyseur.
Exemple :

```
...
begin
...
testif : if (i=200) and (j>300) generate
    w100 :   wal3 port map(a(100)=>p0(i),
                          a(2 downto 1)=>a(4 downto 3),
                          s(i+1)=>p11, s(0)=>s(0));
end generate;
```

En plus des expressions arithmétiques et booléennes (voir la partie sur les descriptions RTL de fonctions booléennes), les opérateurs comparatifs suivants sont analysables : `= < > >= =<` .

4.1.2 Mécanisme de traduction vers le format BLIF

Le format BLIF possède une puissance d'expression beaucoup moins importante que le VHDL structurel. Les facteurs suivants doivent être pris en compte lors de la traduction vers un modèle BLIF :

- Il n'y a pas de composants génériques dans BLIF (et donc pas de variables numériques ni d'expressions arithmétiques).
- Les signaux correspondent à des variables booléennes.

Le sous-ensemble VHDL analysable est stocké dans un ensemble de classes de la catégorie *VHDL*, qui contient également l'analyseur lui-même, VHDLParser. Après analyse, un objet de classe VHDLComponant est obtenu. Celui-ci est ensuite modifié pour être directement transformable en un modèle BLIF.

La traduction vers le format BLIF procède de la manière suivante, en respectant la hiérarchie de composants décrite par un ensemble de fichiers VHDL :

- Chaque composant est traduit en un modèle BLIF.
- Toutes les expressions arithmétiques sont évaluées. Chaque appel à un composant possédant un jeu de valeurs distinct pour ses paramètres génériques donne lieu à la création d'un modèle BLIF différent.
- Les boucles sont complètement déroulées et les instructions conditionnelles sont évaluées.
- Chaque signal est traduit en un ensemble de variables booléennes.

Les appels (`port map` VHDL) étant traduits par des appels à des sous-modèles BLIF, la hiérarchie du circuit résultant est complètement préservée.

4.1.3 Méthodes d'entrées/sorties VHDL

Nous décrivons dans cette partie les principaux messages utilisables pour la lecture et l'écriture au format VHDL de circuits à partir de OO-PPart. La lecture permet de produire des descriptions BLIF hiérarchiques, intégrables ensuite en tant que modèle dans OO-PPart. L'écriture d'un nœud d'une hiérarchie, ou d'un circuit complet est possible à partir du système.

Lecture

La lecture d'un circuit hiérarchique décrit au format VHDL structurel par un ensemble de fichiers (extension `.vhd`) est réalisée à partir de la classe de l'analyseur, VHDLParser, en lui envoyant le message fromFiles: anArrayOfFilenames inDir: aDirectory.

anArrayOfFilenames est une liste de noms de fichiers (sans extension), situés dans le répertoire aDirectory. Le premier fichier doit contenir la description du composant racine de la hiérarchie du circuit. Le circuit doit être complètement spécifié par l'ensemble de fichiers. La méthode fromFiles: anArrayOfFilenames réalise la même tâche, mais dans le répertoire courant. L'objet retourné est de classe VHDLComponant, et peut être traduit au format BLIF avec le message printBLIFOn: aStream ou printBLIFToFile: aFilename. La conversion vers un BLIFModel est assurée (en passant par des fichiers BLIF) à l'aide du message asBLIFModel à un VHDLComponant.

Pour réaliser la traduction BLIF complète d'une hiérarchie, utiliser le message fromFiles: anArrayOfFilenames inDir: aDirectory toDir: aDirectory de la classe VHDLParser, qui génére un fichier VHDL par composant de la hiérarchie dont la racine est le composant décrit dans le premier fichier de la liste anArrayOfFilenames.

Tous ces messages permettent de générer des descriptions BLIF de circuits décrits au format VHDL. La lecture dans le système OO-PPart de circuits, modélisés par des objets de la classe BLIFModel est effectuée par la lecture de BLIF hiérarchique. La lecture de fichiers VHDL dans OO-PPart est donc exécutée en passant par le format intermédiaire BLIF.

nota : *Le message* fromFile: aFile dir: aDir *permet, à partir de la spécification d'un seul fichier, de récupérer la description* VHDL *complète d'un circuit à condition que chaque composant (*comp*) appelé dans* aFile *soit stocké dans* `aDir/comp.vhdl`.

Écriture

L'écriture au format VHDL de circuits stockés en mémoire dans des objets de la classe BLIFModel est réalisée à l'aide des méthodes suivantes. printVhdlOn: aStream (ou printVhdlToFile: aFilename) affiche la forme textuelle VHDL du receveur (un objet de la classe BLIFModel). Pour générer une hiérarchie complète de composants VHDL dans un ensemble de fichier, utiliser le message printVhdlToDir: aDirectory, qui génère, dans aDirectorory l'ensemble des composants de la hiérarchie dont le receveur est la racine.

4.2 Interface EDIF

4.3 Interface graphique

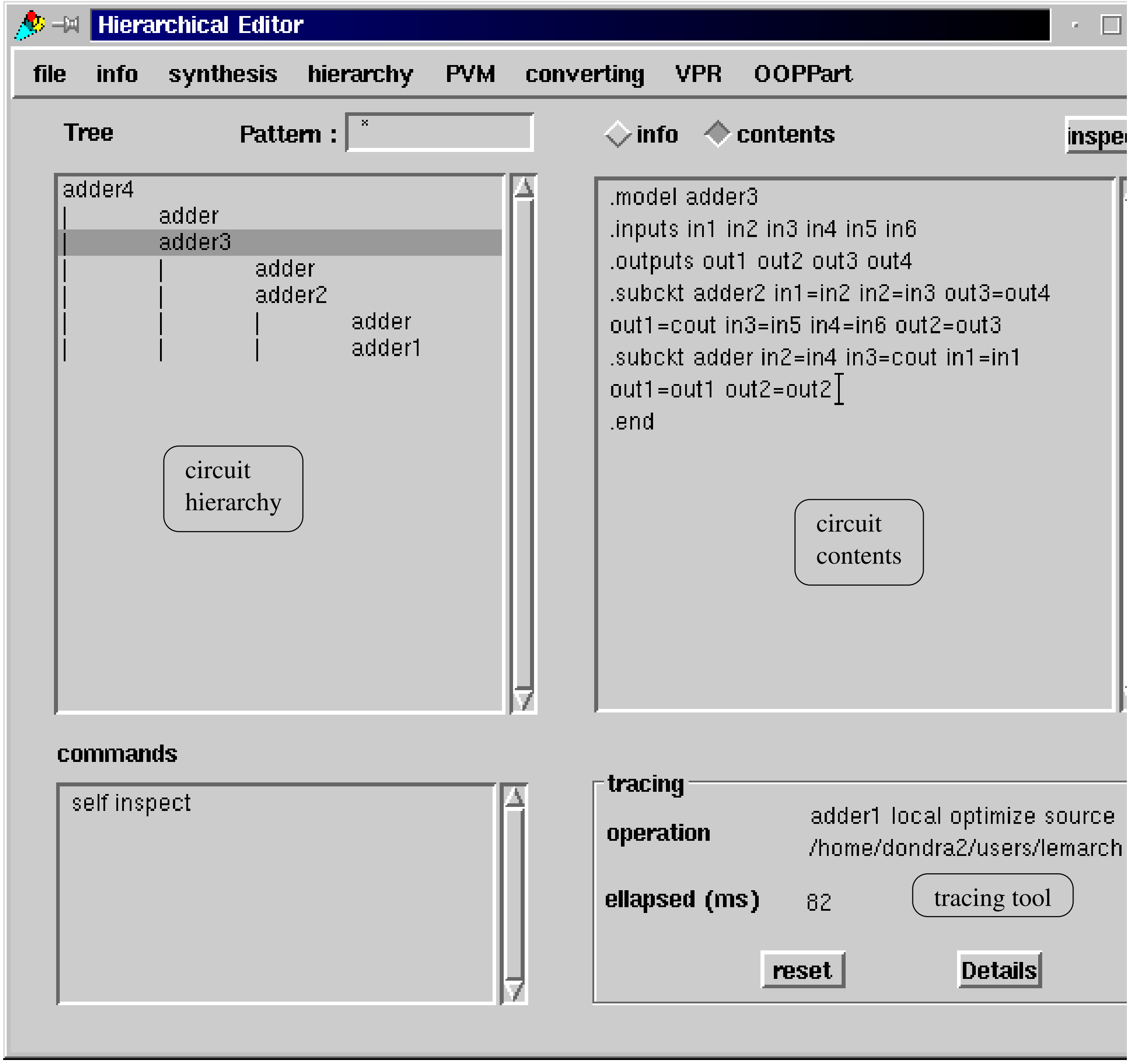

FIG. 4.1 – *L'interface graphique de* OO-PPart *pour la visualisation de circuits hiérarchiques*

Chapitre 5

Placement

5.1 Placement à l'aide de VPR

Chapitre 6

Disponibilité et installation

Le logiciel fonctionne sous VisualWorks 3.1, sur un OS Linux.

6.1 Distribution

L'ensemble des fichiers binaires est disponible pour Linux. La distribution se présente de la façon suivante :

- répertoire `Classes` : Les sources SmallTalk pour les différentes classes.
- répertoire `C` : Les interfaces C, avec un Makefile associé. La génération des binaires C créée les exécutables dans le répertoire `bin/${PVM_ARCH}` .
- répertoire `DLL/${PVM_ARCH}` : Les librairies dynamiques pour les différentes architectures.
- répertoire `tmp` : pour les fichiers temporaires créés par le logiciel.

Le binaire SIS compilé pour Linux est inclus dans le répertoire `bin/LINUX`.

6.2 Installation

L'installation nécessite un environnement PVM à jour.

Le fichier `ooppart.tar.gz` contient l'ensemble des répertoires ci-dessus. Pour l'installer :

- créer un répertoire d'installation, par exemple `OOPPart`.
- dé-zipper et dé-tarer `ooppart.tar.gz` dans `OOPPart`.
- mettre à jour la variable d'environnement `OOPPART` avec le chemin vers le répertoire d'installation. Par exemple, avec `csh` :
  `% setenv OOPPART /home/users/joe/.../OOPPart`
- aller dans le répertoire `C` et faire `make`
- démarrer VisualWorks 3.1 à partir du répertoire `OOPPart`.
- Charger le parcel DLL C connect, si celui-ci n'est pas déjà installé.
- File in le fichier `Classes/OOInstall.st`

6.3 Gestion des sources et de l'environnement

La classe OOPPart implante un mécanisme de gestion des sources SMALLTALK correspondant à l'application. Les diverses méthodes de classe permettent de sauvegarder les catégories de classe liées au logiciel, ou à l'inverse, de restaurer dans l'image SmallTalk les sources de ces catégories. La classe OOPPart contient également dans des variables de classe les noms des différents répertoires, librairies externes et exécutables utilisés dans les différentes classes de l'application.

Variables d'environnement Le fonctionnement des différentes méthodes est subordonné à la définition dans l'environnement des variables OOPPART et PVM_ARCH. La classe répond aux messages suivants :

- resetVars : mets à nil toutes les variables maintenues par la classe.
- ooPPart : la racine de l'arborescence de l'installation. Calculée à partir de la variable d'environnement OOPPART si elle vaut nil.
- architecture : le type d'architecture (normalement LINUX). Calculée à partir de la variable d'environnement PVM_ARCH si elle vaut nil.
- binDir, libraryDir, tmpDir : les différents sous-répertoires utilisés, reflétant l'installation standard si ils valent nil.
- alcFork, sisFork, vprFork, sisExec, vpr, pvmManager, tpack : les différents exécutables, avec leur valeur par défaut.
- eiMetis, eiHMetis : les différentes librairies.

Les exécutables et librairies sont stockées dans des sous-répertoires dont les noms correspondent à l'architecture de la machine. Il est donc possible avec une seule installation de OO-PPart d'exécuter le logiciel sur des plate-formes différentes.

Gestion des sources Les principales méthodes de classe de OOPPart pour la gestion des sources sont les suivantes :

- save : Sauvegarde de toutes les catégories de classes de l'application dans le répertoire `$OOPPART/Classes`.
- savegz : Sauvegarde avec save puis création de `ooppart.tar.gz` (la distribution complète de OO-PPart) ainsi que de `classes.tar.gz` contenant uniquement les sources, sans les binaires liés à l'application.
- update : Remplacement de toutes les catégories de classes de l'application par celles du répertoire `$OOPPART/Classes`.
- make : création des exécutables liés à l'application, à partir des sources C.

Bibliographie

[GBD+94] Geist (G.), Beguelin (A.), Dongarra (J.), Jiang (W.), Manchek (R.) et Sunderam (V.). – *PVM: Parallel Virtual Machine - A Users Guide and Tutorial for Network Parallel Computing.* – MIT Press, 1994.

[KAKS97] Karypis (G.), Aggarwal (R.), Kumar (V.) et Shekhar (S.). – Multilevel hypergraph partitioning : Application in VLSI domain. *In : Proc. ACM/IEEE Design Automation Conf.* – juin 1997.

[Lem99] Lemarchand (L.). – *Parallélisation par partitionnement pour la synthèse logique combinatoire sur* FPGA *à base de* LUT. – Rennes I, Thèse de PhD, IFSIC, 1999.

[Lem01] Lemarchand (L.). – *Une interface Lex & Yacc sous* SmallTalk. – Rapport technique, UBO, EA 2215, 2001.

[Sa92] Sentovich (E.) et al. – *SIS: a system for sequential circuit synthesis.* – memorandum n° UCB/ERL M92/41, University of California at Berkeley, mai 1992.