Traduction SCADE/Lustre vers VHDL *

Adrien Guatto et Marc Pouzet
LRI †

31 aout 2010

1 Introduction

Ce document présente le problème de la compilation d'un langage synchrone tel que SCADE vers un langage pour le hardware tel que VHDL. Nous décrivons le problème posé, les principales solutions envisagées et la solution proposée par le LRI.

Ce travail s'est appuyé sur la réalisation d'un prototype d'étude, appelé Heptagon. Il s'agit d'un compilateur produisant à la fois du code séquentiel (ici, principalement C et Java) et du code VHDL à partir d'un programme synchrone. Le langage d'entrée est un sous-ensemble de Scade 6 et qui en reprend les principales constructions : équations data-flow, automates hiérarchiques et tableaux. Son compilateur est organisé de manière similaire au compilateur KCG de Scade développé par Esterel-Technologies [1]. Le prototype Heptagon a été mis a la disposition des partenaires du projet GENCOD.

1.1 Compilation de SCADE/Lustre vers VHDL

Rapellons l'organisation générale du compilateur KCG (Figure 1).

La compilation d'un programme se déroule en quatre grandes étapes : 1/ une phase d'analyse statique (typage [**?**], calcul d'horloges [**?**], analyse de causalité et analyse d'initialisation [**?**]) ; 2/

*Rapport d'étude dans le cadre du projet GENCOD.

†Marc Pouzet est maintenant professeur à l'Université Pierre et Marie Curie et rattaché à l'École normale supérieure. Adrien Guatto, étudiant de l'Université Pierre et Marie Curie, a effectué son stage dans le cadre du projet.

1. Pour être complet, Heptagon est un sous-ensemble du langage Lucid Synchrone [3] dont les principaux traits sont intégrés à Scade 6. Nous aurions donc pu tout aussi bien réaliser un prototype d'étude à partir de Lucid Synchrone. Le langage étant plus riche (ordre supérieur, inférence de type, polymorphisme, etc.), cela nécessitait de résoudre des problèmes peu pertinents pour le projet GENCOD. D'où le choix de réaliser un prototype simplifié, plus proche de la structure interne du compilateur KCG pour un langage proche de Scade 6.

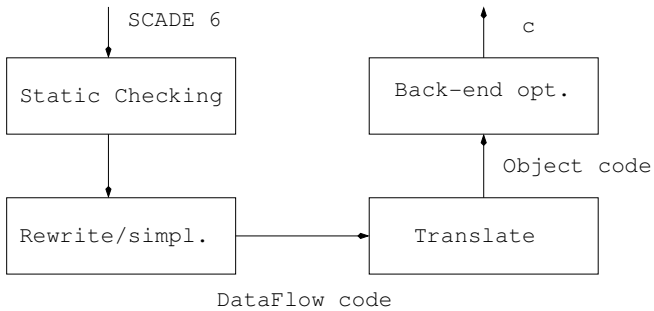

Figure 1 – Organisation générale du compilateur de SCADE

une phase comportant une succession de réécritures produisant à la fin un code data-flow avec horloges ; 3/ une phase de compilation vers du code impératif séquentiel (object code) où chaque noeud Scade est représenté par une fonction de transition ; 4/ une phase d'optimisation appliquée au code séquentiel (élimination des copies, propagation de constantes, etc.). Au préalable à cette phase, les modules sont expansés et le code polymorphe est spécialisé. Ces deux étapes ne sont pas décrites ici.

Au regard de cette organisation, il y a deux points d'entrées naturels pour produire du code VHDL :

- à partir du code intermédiaire dataflow avec horloges, après élimination des structures de contrôle (e.g., automates, conditions d'activation) ;
- à partir du code final généré par le compilateur (dans le cas de KCG, le code C, par exemple).

Notons que le code data-flow est déjà un point d'entrée pour les outils de vérification formelle utilisés dans le compilateur KCG (le Prover Plug-in). La vérification d'une propriété (écrite en Scade) se fait en traduisant le code et la propriété vers le format data-flow qui sert alors de passerelle vers l'outil Prover.

1.2 Génération de VHDL à partir d'équations data-flow gardées

Ce sont principalement les deux approches qui ont été retenues dans le projet GENCOD. La société GeenSoft a réalisé un compilateur à partir du code C généré par KCG. Dans ce document, nous décrivons l'autre approche dite "directe" à partir du code intermédiaire data-flow vers un sous-ensemble "synthétisable" de VHDL (nous reviendrons plus loin sur ce qualificatif). L'objectif est ici de définir les fonctions de traduction le plus formellement possible afin de pouvoir les intégrer à un environnement certifié tel que SCADE.

1.2.1 Un mot sur la traduction de C vers VHDL

La compilation de SCADE est modulaire : un noeud `counter` est traduit vers deux fonctions C dont l'interface est schématiquement :

```
/* counter.c */
int counter_step(int Counter_res, int Counter_tick,
    counter_mem* self) { ... }

counter_init(counter_mem* self)
  { self->counter_t1 = 0;
    self->counter_t2 = 0; }
)
```

où :

```
typedef struct {
  int counter_t1;
  int counter_t2; }
counter_mem;
```

`counter_step` est la fonction de transition qui prend en entrée un argument supplémentaire représentant son état interne. L'exécution d'un pas produit une sortie et met à jour l'état interne (par effet de bord). La fonction `counter_init` permet d'initialiser l'état interne.

Le corps de ces deux fonctions est formé d'affectations de variables locales où de l'état (ici `self`) ainsi que de structures de contrôle (conditionnelles, construction "switch" et boucles "for" où d'appel à d'autres fonctions de transition). La génération de code VHDL suit le schéma suivant :

- Une affectation de variable locale est traduite par une équation VHDL sur une variable locale. E.g., x `=` e est traduit schématiquement en une equation VHDL x `:=` e. Il faut cependant être très attentif à ce que `x` ne génère pas de registre. Ce point est assez délicat puisque,

en particulier lors de la traduction de `if` *cond* { x `=` *exp*`;` }. Il correspond à la définition d'un flot dont l'horloge est *cond*. Parce que sa définition est partielle (la valeur de x est indéfinie lorsque *cond* est faux, sa traduction en VHDL va conduire le synthétiseur à allouer un registre pour x, ce qui est à la fois inutile et inefficace. Dans le cas où le programme en entrée n'a pas d'effets de bord, la sémantique de SCADE garantit que ce programme est équivalent à l'affectation simple $x = exp$.
- Une affectation de variable d'état $self$ `->` t `=` *exp* devra, elle, générer un registre. Il faut cependant retrouver la condition booléenne d'activation de cette équations.

L'intérêt d'une traduction du code C produit par KCG vers VHDL est d'abord de ne pas toucher au compilateur existant (déjà qualifié). Nous voyons cependant plusieurs difficultés dans cette approche :

- La nécessité de certification demande d'instrumenter le compilateur KCG avec des informations donnant la traçabilité (lien entre les noms de variables produites et les noms dans le code source, en particulier).
- Certaines optimisations pertinentes lorsque l'on génère du code séquentiel, peuvent ne pas être utiles, voire pénalisantes, pour une compilation vers VHDL. C'est le cas de l'optimisation des structures de contrôle ou de la compilation des boucles (cf. discussion ci-dessus, conduisant à générer trop de registres).
- Il est nécessaire de restreindre le périmètre du compilateur C vers VHDL : on ne réalise pas un compilateur capable de traduire tout code C vers VHDL mais plutôt un compilateur adapté au code C produit par KCG et prenant en compte la technique de compilation sous-jacente. Comment décrire ce périmètre ?
- Enfin, c'est une approche assez peu naturelle puisqu'elle consiste à compiler un langage parallèle (SCADE) vers un autre langage parallèle (VHDL) en passant par un langage séquentiel.

Puisque le compilateur KCG produit un code intermédiaire data-flow, nous avons étudié sa traduction directe vers VHDL.

. Bien qu'il n'existe pas de définition précise de ce sous-ensemble, nous avons identifié un sous-ensemble appelé MiniVHDL (Section **??**). dans la littérature, nous avons identifié unles divers échanges avec les partenaires du projet nous ont cont du Il est apparu au cours du projet qu'il n'existait pas de définition précise d'un sous-ensemble synthétisable de VHDL. Après échange avec les partenaires, nous avons identifié un sous-ensemb. Le contexte du projet GENCOD étant la génération de code L'objectif était de proposer avec l'objectif de produire du code "synthétisable".L'une des difficulté a été d'identifier un sous-ensemble de VHDL "synthétisable".Plusieurs difficultés sont apparues, notamment l'ambiguité de l'expression "VHDL synthétisable"dL'objectif est de propoduire du code et les résultats obtenus sont les suivants :

Le code VHDL est obtenu à partir du code Le A la différence de la solution proposée par GeenSoft où la génération de code VHDL est obtenue à partir du code C généré par le compilateur KCG de SCADEA6. Ce prototypune technique de traduction de programmes synchronesprésente une méthode de traduction présente une approche On détaille ici la traduction de Heptagon, un langage synchrone combinant des équations à flots de données, dans l'esprit de Lustre, et des automates hiérarchiques, tels qu'introduits par Lucid Synchrone et SCADE 6, vers le langage de description de circuits VHDL. Cette traduction prend le parti de ne pas passer par un langage impératif mais de directement utiliser la représentation interne à flots de données du compilateur.

Après avoir rappellé brièvement la forme des langages d'entrée et de sortie, on va expliciter la procédure de traduction retenue.

2 Heptagon

Heptagon est un langage synchrone académique conceptuellement proche de SCADE, mais un peu moins expressif : pas de signaux, peu de constructions riches (émissions sur transitions, etc.). Par rapport aux itérations précédente, il dispose désormais de tableaux de taille statique et des itérateurs associés.

On va décrire rapidement quelques exemples illustrant les fonctionnalités du langage, avant de détailler le processus de compilation et l'architecture du compilateur.

2.1 Quelques exemples

Compteur d'événements simple

```
node simple_count(e : bool) returns (o : int)
let
  o = (if e then 1 else 0) + (0 fby o);
tel
```

Le nœud *count* compte le nombre d'évènements e reçus depuis le premier instant du programme.

Compteur multi-événements réinitialisable

```
node count<<n : int>>(event : bool^n; rst : bool)
     returns (count : int)
var pres : int;
let
  pres = fold (+)<<n>>(map int_of_bool<<n>>(event), 0);
  reset count = (0 fby count) + pres every rst;
tel
```

Ce programme implante un compteur d'événements qui comptabilise le nombre de booléens valant `true` sur son entrée *event*, celle-ci étant un tableau de booléens de taille n, où n est un paramètre statiquement connu du noeud. On utilise les itérateurs `map` et `fold` pour calculer le nombre d'événements observés dans l'instant ; le premier permet de traduire les booléens en entiers, et le second d'additioner ceux-ci. Notons également l'utilisation de la construction de réinitialisation `reset`, actionnée simplement lorsque l'entrée *rst* est vraie.

Allocateur de ressource

```
(* Allocateur de ressource pour deux demandeurs. *)
(* Sûreté : non (g0 et g1) *)
node alloc(r0 : bool; r1 : bool) returns (g0 : bool; g1 : bool)
let
  automaton
    state IDLE0
      do g0 = false;
         g1 = false;
      until r0 then ALLOC0 | (r1 & not r0) then ALLOC1
    state IDLE1
      do g0 = false;
         g1 = false;
      until r1 then ALLOC1 | (r0 & not r1) then ALLOC0
    state ALLOC0
      do g0 = true;
         g1 = false;
      until (not r0) then IDLE1
    state ALLOC1
      do g0 = false;
         g1 = true;
      until (not r1) then IDLE0
```

```
    end;
tel
```

Cet exemple présente un automate réalisant l'allocation d'une ressource quelconque à deux demandeurs, avec priorité *round-robin* (en cas de demande simultannée, le processus qui vera sa requêtre satisfaite sera celui ayant obtenu la ressource il y a le plus longtemps).

2.2 Architecture du compilateur

On décrit brièvement l'architecture du compilateur : après des phases initiales d'analyse lexicale, syntaxique et de typage, le programme Heptagon est soumis aux vérifications traditionnelles des langages synchrones (causalité, etc.). Ensuite, la compilation progresse par étapes successives en réécrivant le programme jusqu'à arriver à une forme simple qui peut être traduite simplement en MiniLS. Le code résultant, après avoir été soumis à d'éventuelles optimisations, est mis dans une certaine forme dite "normale" et ordonnancé avant d'être finalement traduit en code séquentiel. L'architecture du compilateur Heptagon est présenté à la figure 2. Le processus de compilation de MiniLS vers un langage impératif séquentiel est décrit en détails dans l'article [1].

Le passage au code séquentiel est court-circuité lors d'une compilation vers VHDL, qui traduit MiniLS directement vers celui-ci. Pour faciliter cette étape, on pratiquera en amont trois transformations simplificatrices sur MiniLS.

A Suppression de la réinitialisation logique.

B Suppression des itérateurs sur tableaux via mise à plat.

C Introduction d'une variable intermédiaire pour chaque argument d'un appel de noeud.

Le code obtenu sera finalement traduit vers un sous ensemble de VHDL baptisé MiniVHDL (étape D), prêt à être traité par les outil dédiés (étape E).

3 De Heptagon à VHDL

3.1 Langages internes

3.1.1 MiniLS

MiniLS est équivalent au langage synchrone à flots de données Lustre [2], auquel on adjoint une construction de réinitialisation modulaire d'un noeud [2]. Un programme MiniLS passera successivement par trois formes distinctes lors du processus de compilation :

- La forme originale 3 telle qu'obtenue à partir du code Heptagon original. Cette forme est décrite ici par souci de cohérence et ne nous intéresse pas directement.
- La forme normale 4. Des transformations décrites ci-dessous seront effectuées sur celle-ci pour faciliter la traduction vers VHDL.
- La forme finale 5, normalisée, sans réinitialisations (*every*) ni itérateurs, et où tous les paramètres effectifs sont des identifiants.

MiniLS est donc un langage à flots de données proche de Lustre dont les programmes sont d'équations mutuellement récursives. Ces équations calculent des suites infinies de valeurs progressant selon un temps découpé en instants discrets. Les paquets d'équations sont regroupés dans des unités lexicales nommées "noeuds" à la granularité dans l'esprit d'une fonction d'un langage impératif tel que C.

On associe à chaque expression du langage un type, qui précise la nature des valeurs calculées, ainsi qu'une horloge, garde booléenne qui spécifie les instants durant lesquels l'expression calcule une valeur définie.

Les expressions simples du langage s'exécutent uniquement dans l'instant, et n'ont pas d'effet en dehors de celui-ci.

2. $f^{ck}(e_1, \ldots, e_n)$ `every` z appelle le noeud f avec les arguments $e_1, \ldots, e_n$, et réinitialise la mémoire de celui-ci lorsque z est vraie.

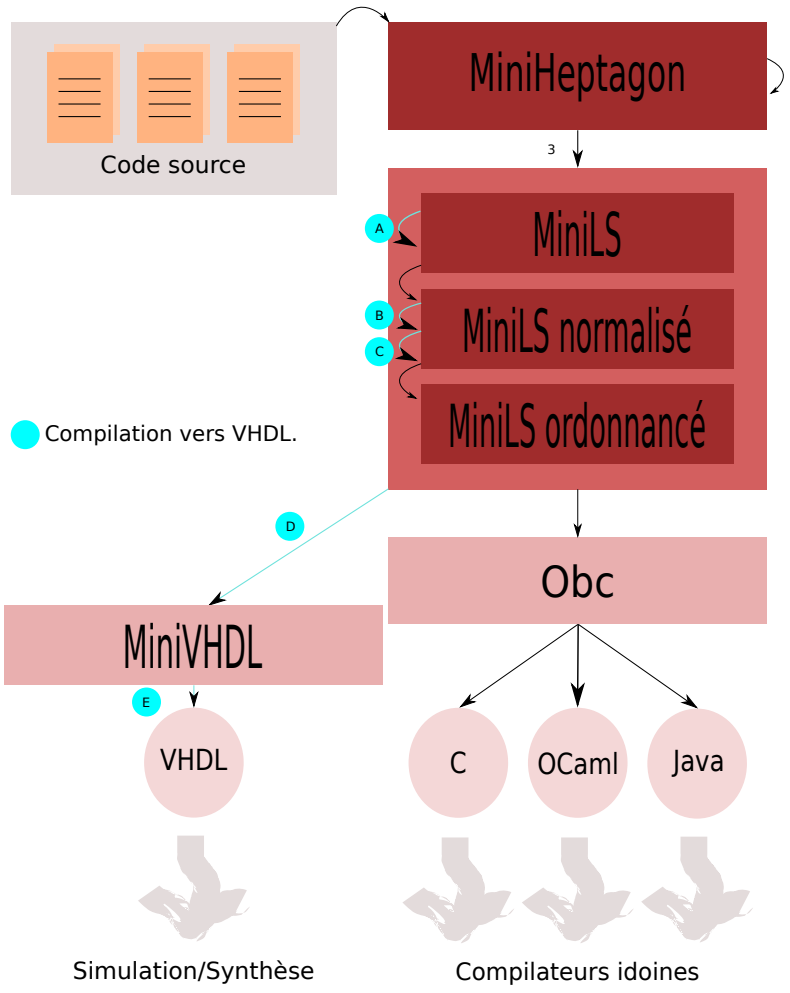

FIGURE 2 – Architecture du compilateur Heptagon

$$
\begin{array}{rcl}
td & ::= & \texttt{type}\ \ bt = C + \cdots + C \\
d & ::= & \texttt{node}\ f(p) = p\ \texttt{with var}\ p\ \texttt{in}\ D \\
p & ::= & x : bt; \ldots ; x : bt \\
D & ::= & pat = e; \ldots ; pat = e \\
pat & ::= & x \mid (pat, \ldots, pat) \\
e & ::= & x \mid v \mid \mathbf{op}(e, \ldots, e) \mid v\ \texttt{fby}^{ck}\ e \mid \texttt{pre}^{ck}\ e \\
 & \mid & f^{ck}(e, \ldots, e)\ \texttt{every}\ x \mid e\ \texttt{when}\ C(x) \\
 & \mid & \texttt{merge}\ x\ (C \Rightarrow e)\ \ldots\ (C \Rightarrow e) \\
 & \mid & \texttt{map}\ fn(e_1, \ldots, e_n) \\
 & \mid & \texttt{fold}\ fn(e_1, \ldots, e_n) \\
 & \mid & \texttt{mapfold}\ fn(e_1, \ldots, e_n) \\
v & ::= & i \mid C \\
ck & ::= & \texttt{base} \mid ck\ \texttt{on}\ C(x)
\end{array}
$$

FIGURE 3 – MiniLS

$$
\begin{array}{rcl}
e & ::= & x \mid v \mid \mathbf{op}(e, \ldots, e) \mid e\ \texttt{when}\ C(x) \\
ce & ::= & e \mid \texttt{merge}\ x\ (C \Rightarrow ce)\ \ldots\ (C \Rightarrow ce) \\
eq & ::= & x = ce \mid x = v\ \texttt{fby}^{ck}\ e \mid x = \texttt{pre}^{ck}\ e \\
 & \mid & (x, \ldots, x) = f^{ck}(e, \ldots, e)\ \texttt{every}\ x \\
 & \mid & (x, \ldots, x) = f^{ck}(e, \ldots, e) \\
 & \mid & \texttt{map}\ fn(e_1, \ldots, e_n) \\
 & \mid & \texttt{fold}\ fn(e_1, \ldots, e_n) \\
 & \mid & \texttt{mapfold}\ fn(e_1, \ldots, e_n)
\end{array}
$$

FIGURE 4 – MiniLS normalisé

1	1	1	1	1	...
1 + 2	3	3	3	3	...

À l'inverse, les expressions pre et fby permettent de retarder leur argument d'un instant, avec une valeur d'initialisation dans le cas de fby.

1 fby 2	1	2	2	2	...
pre 3		3	3	3	...

Les expressions `when` et `merge` permettent respectivement de sous-échantilloner et sur-échantilloner des expression. L'emploi de ces constructions a un effet sur les horloges : l'horloge de e `when` $C(x)$ est ck `on` $C(x)$, où ck est celle de e, et l'horloge de `merge` $x\ (C_1 \Rightarrow e_1)\ \ldots\ (C_n \Rightarrow e_n)$ est ck quand celle des e_i est ck `on` $C(x)$.

$$
\begin{array}{rcl}
e & ::= & x \mid v \mid \mathbf{op}(e, \ldots, e) \mid e \texttt{ when } C(x) \\
ce & ::= & e \mid \texttt{merge } x \ (C \Rightarrow ce) \ \ldots \ (C \Rightarrow ce) \\
eq & ::= & x = \texttt{pre}^{ck}\, e \\
 & \mid & (x, \ldots, x) = f^{ck}(x, \ldots, x)
\end{array}
$$

FIGURE 5 – MiniLS simplifié (et normalisé)

x	1	2	3	4	...
h	$false$	$true$	$false$	$true$	...
x `when` $true(h)$		2		4	...
u		1		4	...
v	0		3		...
`merge` h $(true \Rightarrow u)$ $(false \Rightarrow v)$	0	1	3	4	

Les itérateurs `map`, `fold`, et `mapfold` prennent en argument une fonction f, une taille de tableau et un ou plusieurs tableaux. Ces constructions ont le même sens que dans les langages de programmation fonctionnels habituels.

- `map` applique en parallèle f à tous les éléments des tableaux pour former un nouveau tableau.
- `fold` applique en série f sur tous les éléments d'un tableau en passant un accumulateur et renvoie ce dernier une fois le parcours terminé.
- `mapfold`, qui suppose que f renvoie au moins deux valeurs, combine les deux en construisant un nouveau tableau avec le premier résultat de f (à la manière de `map`) et en passant un accumulateur durant le parcours (tout comme `fold`).

Enfin, la construction $f(e, \ldots, e)$ `every` x réinitialise l'appel à f dès que x est vrai.

3.1.2 MiniVHDL

MiniVHDL 6 est un fragment très restreint de VHDL, suffisant pour décrire l'essence de notre processus de traduction. Un composant MiniVHDL `component` f `port` P `with sig` $sigs$ `and var` $lvars$ `and subcomponents` $ports$ `in` I correspondra concrètement à un composant VHDL formé des instantiations $ports$, signaux internes $sigs$ et d'un processus avec les variables locales $lvars$ et de corps I.

Le langage est structuré sous forme de composants. Chacun d'eux possède des signaux d'entrée/sortie, des signaux locaux, des variables locales, d'éventuels sous-composants, et enfin un corps formé d'une suite d'instructions.

La sémantique informelle du langage est la suivante : dès que la valeur d'un signal d'entrée ou local change, le corps du composant est exécuté. Celui-ci peut modifier la valeur d'autres signaux via l'instruction $x \Leftarrow e$ (x étant par définition un signal), entraînant ainsi l'activation de sous-composants, ou même la réactivation du composant courant. Les variables locales sont semblables aux variables des langages de programmation traditionnels : elles n'ont de valeur que pendant l'exécution du corps d'un composant. L'exécution s'arrête une fois tous les signaux stables.

Notons que les paramètres effectifs des sous-composants sont des noms de signaux ; cela justifie la forme simplifiée MiniLS décrite au paragraphe précédent.

3.2 Simplification de MiniLS normalisé

Nous effectuons donc trois passes pour simplifier le code MiniLS.

$$
\begin{array}{rcl}
\mathit{component} & ::= & \texttt{component}\ f\ \texttt{port}\ \mathit{sm}; \ldots;\ \mathit{sm}\ \texttt{with sig}\ d; \ldots; d \\
 & & \texttt{and var}\ d; \ldots; d\ \texttt{and subcomponents}\ p; \ldots; p\ \texttt{in}\ I \\
\mathit{sm} & ::= & x : \mathit{mode}\ \mathit{ty} \\
\mathit{sd} & ::= & x : \mathit{mode}\ \mathit{ty} := e \\
\mathit{mode} & ::= & \texttt{in} \mid \texttt{out} \\
d & ::= & x : \mathit{ty} \\
p & ::= & \texttt{port map}\ x(\mathit{bd}; \ldots; \mathit{bd}) \\
\mathit{bd} & ::= & x \Rightarrow x \\
I & ::= & i; \ldots; i \\
i & ::= & x \Leftarrow e \mid x ::= e \mid \texttt{case}\ e\ \texttt{of}\ (v \Rightarrow I)\ \ldots (v \Rightarrow I) \\
e & ::= & \mathit{id} \mid v \mid \mathbf{op}(e, \ldots, e) \\
v & ::= & i \mid 'bitp' \\
\mathit{bitp} & ::= & \mathit{bit} \mid \mathit{bit}\ \mathit{bitp} \\
\mathit{bit} & ::= & \mathbf{0} \mid \mathbf{1} \\
\mathit{bt} & ::= & \texttt{natural} \mid \texttt{std\_logic} \mid \texttt{bit} \mid \ldots
\end{array}
$$

FIGURE 6 – MiniVHDL

3.2.1 Élimination de la réinitialisation logique

Comme expliqué plus haut, certaines constructions de MiniLS proposent au programmeur une forme de réinitialisation modulaire : les équations de la forme $v\ \texttt{fby}^{ck}\ e$ d'un noeud f instancié par la construction $f^{ck}(e_1, \ldots, e_n)\ \texttt{every}\ z$ doivent-être réinitialisées dès lors que z est vrai.

La première passe de simplification, qui s'exécute sur le code MiniLS obtenu à la troisième étape du processus décrit plus haut, va exprimer la réinitialisation en fonction du reste du langage, réduisant ainsi la complexité du langage à traduire vers VHDL.

On va supposer dans ce qui suit que l'identifiant **rst** est un identifiant utilisé nulle part dans le programme. En pratique, le compilateur s'assure de l'absence de conflit avec les variables de l'utilisateur.

L'idée est d'ajouter à chaque noeud un argument supplémentaire nommé **rst** qui vaudra *true* lorsqu'une réinitialisation de la mémoire est nécessaire, et de modifier les expressions contenant des constructions `fby` ou `every` pour prendre en compte **rst**. L'exemple suivant reprend le compteur simple présenté plus haut pour illustrer cette simplification.

```
node simple_count(e : bool) returns (o : int)
let
  o = (if e then 1 else 0) + (0 fby o);
tel

node main() returns (o : int)
let
  o = simple_count(true fby false fby true);
tel
```

Une fois le reset éliminé, le code est le suivant :

```
node simple_count(rst : bool; e : bool) returns (o : int)
let
  o = (if e then 1 else 0) + (if rst then 0 else (0 fby o));
```

```
tel

node main(rst : bool) returns (o : int)
let
  o = simple_count(rst,
                   if rst
                   then true
                   else (true fby (if rst
                                   then false
                                   else (false fby true))));
tel
```

On va détailler les fonctions effectuant ces deux tâches : $RstE(e)$ prend une expression MiniLS e et renvoie une nouvelle expression où la réinitialisation a été éliminée, et $RstNode(nd)$ prend un nœud nd et renvoie un nouveau nœud prenant un `rst` à un noeud et transforme ses équations.

$$
\begin{array}{lcl}
RstE(\mathbf{op}(e_1,\ldots,e_n)) & = & \mathbf{op}(RstE(e_1),\ldots,RstE(e_n)) \\
RstE(v\ \texttt{fby}^{ck}\ e) & = & \texttt{if}\ rst\ \texttt{then}\ v\ \texttt{else}\ (v\ \texttt{fby}^{ck}\ RstE(e)) \\
RstE(\texttt{pre}^{ck}\ e) & = & \texttt{pre}^{ck}\ RstE(e) \\
RstE(f^{ck}(e_1,\ldots,e_n)\ \texttt{every}\ x) & = & f^{ck}(\mathbf{rst}\ \texttt{or}\ x, RstE(e_1)\ldots,RstE(e_n)) \\
RstE(f^{ck}(e_1,\ldots,e_n)) & = & f^{ck}(\mathbf{rst}, RstE(e_1),\ldots,RstE(e_n)) \\
RstE(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) & = & \texttt{if}\ RstE(e_1)\ \texttt{then}\ RstE(e_2) \\
 & & \qquad\texttt{else}\ RstE(e_3) \\
RstE(e\ \texttt{when}\ C(x)) & = & RstE(e)\ \texttt{when}\ C(x) \\
RstE(\texttt{merge}\ e\ (C_1 \Rightarrow e_1)\ \ldots\ (C_n \Rightarrow e_n)) & = & \texttt{merge}\ RstE(e)\ (C_1 \Rightarrow RstE(e_1)) \\
 & & \qquad\ldots \\
 & & \qquad(C_n \Rightarrow RstE(e_n))
\end{array}
$$

$$
\begin{array}{l}
RstEqs(pat_1 = e_1;\ldots;pat_n = e_n) = \\
\quad pat_1 = RstE(e_1);\ldots;pat_n = RstE(e_n) \\
RstNode(\texttt{node}\ f(f) = x_1,\ldots,x_n\ \texttt{with var}\ y_1,\ldots,y_n\ \texttt{in}\ eqs) = \\
\quad \texttt{node}\ f(f) = rst, x_1,\ldots,x_n\ \texttt{with var}\ y_1,\ldots,y_n\ \texttt{in}\ ResetEqs(eqs)
\end{array}
$$

3.2.2 Suppression des itérateurs

La version actuelle du compilateur et de son générateur de code VHDL supporte les tableaux de dimension arbitraire [3] et constructions associées ; il nous faut donc compiler les itérateurs `map`, `fold` et `mapfold` vers VHDL.

Par souci de simplicité et uniformité, nous avons fait le choix de les éliminer par *inlining* lors d'une transformation source-à-source sur MiniLS. On ne détaillera pas cette opération qui consiste simplement à remplacer les itérateurs par plusieurs équations. L'équation $x = \texttt{map}\ fn(t_1,\ldots,t_m)$ lorsque les tableaux $t_1,\ldots,t_m$ sont de taille n sera ainsi remplacée par $n+1$ équations dont les n premières effectuent l'application de f pour chaque indice et la dernière affecte à x le tableau en résultant. On applique des transformations similaires aux opérateurs `fold` et `mapfold`.

Le programme suivant présente un exemple effectuant un ET logique sur tous les éléments d'un tableau via l'itérateur `fold`.

```
node main() returns (o : bool)
var tab : bool^3;
let
  tab = [true, true, true];
  o = fold (&)<<3>>(tab, true);
tel
```

3. En pratique, les outils de synthèse de circuits à partir de code VHDL imposent une dimension maximale.

Son pendant avec itérateur mis à plat se contente de passer l'accumulateur d'élément en élément.

```
node main(rst_4 : bool) returns (o : bool)
var z_10 : bool; z_9 : bool; z_8 : bool; tab : bool^3;
let
  tab = [true; true; true];
  z_8 = tab[0] & true;
  z_9 = tab[1] & z_8;
  z_10 = tab[2] & z_9;
  o = z_10;
tel
```

3.2.3 Simplification des appels

Comme nous le verrons plus bas, les appels de nœuds seront compilés en instantiations de composants ; or, les arguments d'une construction VHDL *port map* sont forcément des identifiants. Pour simplifier la génération de VHDL, on va donc modifier en amont chaque appel de noeud en introduisant une variable intermédiaire pour chaque argument. La fonction $Simpl(eq, eqs)$ simplifie l'équation eq en ajoutant la ou les équations produits à la liste des équations déjà traitées eqs ; $SimplNode(nd)$ prend un noeud nd et simplifie les appels présents dans ses équations.

$$
\begin{array}{l}
Simpl(x = ce, eqs) \quad = \\
\quad (x = ce) :: eqs \\
Simpl(x = \texttt{pre}^{ck}\, e, eqs) \quad = \\
\quad (x = \texttt{pre}^{ck}\, e) :: eqs \\
Simpl((x_1, \ldots, x_n) = f^{ck}(e_1, \ldots, e_n), eqs) \quad = \\
\quad (y_1 = e_1) :: \ldots :: (y_n = e_n) :: ((x_1, \ldots, x_n) = f^{ck}(rst, y_1, \ldots, y_n)) :: eqs \\
\quad \text{où } y_1, \ldots, y_n \text{ sont des noms de variables frais}
\end{array}
$$

$$
\begin{array}{l}
SimplNode(\texttt{node}\ f(x_1, \ldots, x_n) = y_1, \ldots, y_n\ \texttt{with var}\ p\ \texttt{in}\ D) \quad = \\
\quad \texttt{node} f(x_1, \ldots, x_n) = y_1, \ldots, y_n \\
\quad \texttt{with var}\ p'\ \texttt{in}\ fold\_right\ Simpl\ D\ [] \\
\quad \text{en supposant que que } p' \text{ correspondaux variables définies par} \\
\quad \text{les nouvelles équations.}
\end{array}
$$

Ces simplifications effectuées, on va s'atteler à la traduction de MiniLS vers MiniVHDL.

3.3 MiniLS simplifé vers MiniVHDL

Les idées générales de la traduction de MiniLS simplifié vers MiniVHDL sont les suivantes :

- Chaque noeud MiniLS correspondra à un composant (Mini)VHDL.
- Chaque équation à mémoire (i.e. contenant fby ou pre) va correspondre à un signal VHDL local, chaque équation combinatoire à une variable locale.
- Les horloges importent uniquement pour les équations de la forme $x = v\,\texttt{fby}^{ck}\,e$ et $(x_1, \ldots, x_n) = f^{ck}(e_1, \ldots, e_n)$: il faudra alors générer la garde booléenne correspondant à ck. Les autres sont combinatoires et ne nécessitent pas de traitement particulier.
- La réinitialisation logique est gérée en amont comme expliquée ci-dessus, elle est donc implicitement asynchrone (indépendante des fronts montants de l'horloge).
- Les partenaires ont exprimé le désir de pouvoir réinitialiser physiquement toute la mémoire lors du bascument d'un signal précis nommé **hwrst** : on ajoute donc ce signal supplémentaire invisible dans le code MiniLS et on génère le code de réinitialisation correspondant lors du traitement du fby. Tout comme pour l'identifiant rst plus haut, on suppose que l'identifiant **hwrst** n'est pas utilisé dans le programme.
- Pour respecter la sémantique à Δ-cycles de VHDL, il importe de faire évoluer la mémoire par un pas du calcul uniquement sur front montant de l'horloge.

– En suivant le modèle synchrone, les valeurs calculées par le circuit à d'autres moments que le front montant n'ont pas de sens bien défini ; on les ignorera donc.
– Chaque appel de noeud correspondra à une instantiation. Comme spécifié plus haut, les paramètres effectifs d'un signal VHDL sont obligatoirement des signaux auxquels il faudra assigner la valeur correcte.

3.3.1 Traduction des types

Les déclarations de types de données ont été laissées implicites aussi bien dans la syntaxe de MiniLS que de MiniVHDL ; les possibilités étant exactement les mêmes (énumérations et enregistrements), on choisit de ne pas s'attarder sur leur traduction qui reste une traduction mot-à-mot d'une syntaxe concrète à l'autre.

3.3.2 Traduction des constantes et fonction auxiliaires sur les horloges

La fonction $TradConst(c)$ traduit une constante MiniLS c en constante MiniVHDL.

$$\begin{array}{lcl} TradConst(i) & = & i \\ TradConst(true) & = & '1' \\ TradConst(false) & = & '0' \\ TradConst(C) & = & C \end{array}$$

La fonction auxiliaire *GuardClock* permet de traduire une horloge MiniLS en expression MiniVHDL de type booléen. Elle sera utilisée pour contrôler la mise à jour des registres, s'assurant que cette dernière n'est effectuée qu'aux instants où l'horloge est effective.

$$\begin{array}{lcl} GuardClock(\texttt{base}) & = & rising\_edge(clk) \\ GuardClock(ck\ \texttt{on}\ C(x)) & = & x = TradConst(C)\ \texttt{and}\ GuardClock(ck) \end{array}$$

Tout comme les mises à jour des mémoires, les appels à d'autres noeuds sont dirigés par les horloges qui en donnent la cadence. Il nous faudra donc une fonction voisine de *GuardClock* pour calculer l'expression MiniVHDL correspondant à l'horloge utilisée dans l'appel d'un noeud.

$$\begin{array}{lcl} ExpClock(\texttt{base}) & = & clk \\ ExpClock(ck\ \texttt{on}\ C(x)) & = & x = TradConst(C)\ \texttt{and}\ ExpClock(ck) \end{array}$$

3.3.3 Traduction des expressions et équations

La fonction $TradExp(e)$ traduit l'expression MiniLS normalisée et simplifié e en expression MiniVHDL.

$$\begin{array}{lcl} TradExp(v) & = & TradConst(v) \\ TradExp(x) & = & x \\ TradExp(\mathbf{op}(e_1, \ldots, e_n)) & = & \mathbf{op}(TradExp(e_1), \ldots, TradExp(e_n)) \\ TradExp(e\ \texttt{when}\ C(x)) & = & TradExp(e) \end{array}$$

La construction `when` n'a pas de sens calculatoire, et peut n'être vue que comme une annotation d'horloge. Elle sera ignorée lors du processus de traduction.

La fonction $TradCExp(ce)$ traduit les expressions ce de contrôle formées d'expressions simples ou de `merge` imbriqués en instructions MiniVHDL.

$$\begin{array}{l} TradCExp(x, \texttt{merge}\ y\ (C_1 \Rightarrow ce_1)\ \ldots\ (C_n \Rightarrow ce_n)) \quad = \\ \quad \texttt{case}\ y\ \texttt{of}\ (TradConst(C_1) \Rightarrow TradCExp(x, ce_1)) \\ \qquad\qquad \ldots \\ \qquad\qquad (TradConst(C_n) \Rightarrow TradCExp(x, ce_n)) \\ TradCExp(x, e) \quad = \\ \quad x ::= TradExp(e) \end{array}$$

Enfin, la fonction $TradEq$ permet de passer des équations aux instructions MiniVHDL. Elle prend un argument supplémentaire permettant de compter le nombre d'appels de noeuds afin de générer des arguments supplémentaires, et on se donne une fonction supplémentaire $MakeArg(x, i)$ qui génère un nom de variable frais à partir du nom de variable x et de l'entier i.

La traduction des équations appelant un noeud nécessite des explications concernant la façon de compiler les appels d'un noeud MiniLS qui seront données à la section suivante.

$$
\begin{array}{lcl}
TradEq(x = ce, i) & = & TradCExp(x, ce), i \\
TradEq(x = \texttt{pre}^{ck}\, e, i) & = & \texttt{if } GuardClock(ck) \texttt{ then} \\
 & & \quad x \Leftarrow TradExp(e) \\
 & & \texttt{end if}, i \\
TradEq(x = y\, \texttt{fby}^{ck}\, e, i) & = & \texttt{if hwrst then} \\
 & & \quad x \Leftarrow y \\
 & & \texttt{elsif } GuardClock(ck) \texttt{ then} \\
 & & \quad x \Leftarrow TradExp(e) \\
 & & \texttt{end if}, i \\
TradEq((x_1, \ldots, x_n) = f^{ck}(y_1, \ldots, y_n), n) & = & MakeArg(\text{"}ck\text{"}, i) \Leftarrow ExpClock(ck) \\
 & & MakeArg(y_1, i) \Leftarrow y_1 \\
 & & \ldots \\
 & & MakeArg(y_n, i) \Leftarrow y_n, i + 1
\end{array}
$$

Précisons que par construction, l'interface d'un composant est toujours de la forme $(clk, in_1, \ldots, in_n, out_1, \ldots, out_n$

3.3.4 Gestion des tableaux

Hormis le cas des itérateurs traités précédemment, la gestion des tableaux n'appelle pas de commentaire particulier, à l'exception de la nécessité de déclarer à l'avance les types tableaux (bornes exclues) en VHDL. Deux solutions sont envisageables :

- Calculer la dimension maximale des tableaux rencontrés dans le programme, et utiliser cette information pour pré-déclarer les tableaux VHDL idoines.
- Déclarer quoi qu'il advienne les types de tableaux utiles et refuser les programmes comprenant des dimensions supérieures à une limite fixée à l'avance .

Le compilateur emploie pour l'instant la première méthode, mais la seconde ne nous semble pas dérangeante pour des raisons pragmatiques [4].

3.3.5 Compilation modulaire et appels de noeuds

MiniVHDL offre une forme de modularité basée sur une hiérarchie de composants. Chacun de ces derniers spécifie une liste de composants fils dont les ports (au sens de la figure 6) sont instanciés avec des signaux. Nous prenons donc soin d'utiliser des signaux comme résultats mais aussi arguments ; par souci de simplicité, on introduit des signaux locaux pour chaque argument, signaux qui seront affectés lors de la traduction de l'équation correspondant à l'appel de noeud original.

La fonction $GatherPortMaps(D, i)$ rassemble cette liste de sous-noeuds à partir des appels de noeud présents dans le paquet d'équations D et créé les instantiations de composants correspondantes. L'entier i nous servira à distinguer les appels de noeuds et de créer de nouveaux noms de signaux frais grâce à la fonction $MakeArg$ décrite plus haut. On se donne également une fonction $GetArgName(f, n)$ qui renvoie le nom du n-ème argument du noeud de nom f.

4. Notons que notre version de l'outil Xilinx ISE refuse par exemple tout tableau de dimension supérieure à trois.

$$
\begin{array}{lcl}
GatherPortMaps([], i) & = & [] \\
GatherPortMaps(((x_1, \ldots, x_n) = f^{ck}(y_1, \ldots, y_n)) :: eqs, i) & = & \\
\quad \texttt{port map}\ f(clk \Rightarrow MakeArg(\text{"}clk\text{"}, i) & & \\
\qquad GetArgName(f, 1) \Rightarrow MakeArg(y_1, i) & & \\
\qquad \ldots & & \\
\qquad GetArgName(f, n) \Rightarrow MakeArg(y_n, i)) & & \\
\quad :: GatherPortMaps(eqs, i + 1) & &
\end{array}
$$

On définit ensuite les fonctions auxiliaires $NeedVar$, $Vars$, $ParamSigs$ et $SignalOfVarDec$ respectivement chargées de déterminer si une équation introduit des déclarations de variables locales ou non, de calculer la liste des variables définies par une équation, de calculer les signaux à passer en arguments aux appels de noeuds présents dans un paquet d'équations et enfin de traduire simplement une déclaration de variable MiniLS en déclaration de signal MiniVHDL avec mode d'utilisation (entrée ou sortie).

$$
\begin{array}{lcl}
NeedVar(x = v\ \texttt{fby}^{ck}\ e) & = & false \\
NeedVar((x_1, \ldots, x_n) = f^{ck}(y_1, \ldots, y_n)) & = & false \\
NeedVar(x = ce) & = & true
\end{array}
$$

$$
\begin{array}{lcl}
Vars(x = v\ \texttt{fby}^{ck}\ e) & = & [x] \\
Vars((x_1, \ldots, x_n) = f^{ck}(y_1, \ldots, y_n)) & = & x_1 :: \ldots :: x_n \\
Vars(x = ce) & = & [x]
\end{array}
$$

$$
\begin{array}{l}
ParamSigs(x = v\ \texttt{fby}^{ck}\ e :: eqs, i) = \\
\quad ParamSigs(eqs, i) \\
ParamSigs((x_1, \ldots, x_n) = f^{ck}(y_1, \ldots, y_n) :: eqs, i) = \\
\quad MakeArg(\text{"}clk\text{"}, i) :: MakeArg(y_1, i) :: \ldots :: MakeArg(y_n, i) \\
\quad :: ParamSigs(eqs, i + 1) \\
ParamSigs(x = ce :: eqs, i) = \\
\quad ParamSigs(eqs, i)
\end{array}
$$

$$
SignalOfVarDec(x : bt, mode) \quad = \quad \texttt{signal}\ x : mode\ TransBaseType(bt)
$$

3.3.6 Traduction des noeuds

Enfin, $TradNode(nd)$ se charge de traduire un noeud nd en composant MiniVHDL, en créant un signal local pour chaque équation retardée et argument d'appel de noeud, une variable pour chaque équation combinatoire, et la liste de sous-composants requise.

$$
\begin{array}{l}
TradNode(\texttt{node}\ f(in) = out\ \texttt{with var}\ p\ \texttt{in}\ D) = \\
\quad \text{soit } port = \\
\qquad clk : \texttt{in std\_logic}; \\
\qquad \{SignalOfVarDec(x : bt, \texttt{in}) \mid (x : bt) \in in\}; \\
\qquad \{SignalOfVarDec(x : bt, \texttt{out}) \mid (x : bt) \in out\}, \\
\quad \text{soit } signals = \{Vars(eq) \mid eq \in D, \neg NeedVar(eq)\}, \\
\quad \text{soit } sig\_args = ParamSigs(D, 1), \\
\quad \text{soit } variables = \{Vars(eq) \mid eq \in D, NeedVar(eq)\}, \\
\quad \text{soit } ports = GatherPortMaps(D, 1), \\
\quad \text{soit } body = fold\_rightTradEqD([], 1) \text{ dans} \\
\texttt{component}\ f\ \texttt{port}\ port\ \texttt{with sig}\ signals \cup sig\_args \\
\texttt{and var}\ variables\ \texttt{and subcomponents}\ ports\ \texttt{in}\ body
\end{array}
$$

4 Discussion

5 Conclusion

On a traduit Heptagon vers VHDL en profitant des forces du modèle synchrone : sa sémantique est bien adaptée à une description sous forme de circuits. Le processus de traduction reste simple et compréhensible grâce aux garanties offertes par le synchrone : nul besoin d'écrire plusieurs fois dans le même élément mémorisant par cycle.

Table des matières

Références

[1] Darek Biernacki, Jean-Louis Colaco, Grégoire Hamon, and Marc Pouzet. Clock-directed Modular Code Generation of Synchronous Data-flow Languages. In *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Tucson, Arizona, June 2008.

[2] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre : a declarative language for real-time programming. In *POPL '87 : Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 178–188, New York, NY, USA, 1987. ACM.

[3] Marc Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, April 2006. Distribution available at : `www.lri.fr/~pouzet/lucid-synchrone`.

A Exemples de code généré

On présente ici quelques codes générés par notre prototype. En l'état actuel de ce dernier, beaucoup de variables intermédiaires inutiles sont générées ; cela s'explique pour deux raisons :

- Ces codes représentent des sorties brutes n'ayant subies aucune optimisation.
- Certaines passes du compilateur ont naturellement tendance à introduire des variables intermédiaire de façon préventive, simplifiant ainsi leur fonctionnement.

A.1 Compteur

MiniLS initial

MiniLS normalisé, ordonnancé et simplifié Notons que le noeud *compteur* est ici instancié avec son paramètre n égal à 4, ceci afin de pouvoir déplier les itérateurs.

VHDL

```
use work.types.all;

library ieee;
use ieee.std_logic_1164.all;

entity count_23 is
  port (signal clk_1 : in std_logic; signal hw_rst_3 : in std_logic;
        signal rst_2 : in std_logic;
        signal event : in std_logic_vector (0 to 3);
        signal rst : in std_logic; signal o_count : out integer);
end entity count_23;

architecture rtl of count_23 is
  signal z_24 : integer;
  signal z_27 : integer;
  signal z_26 : integer;
  signal z_25 : integer;
  signal h_v_15 : integer;
  signal arg_ck_4 : std_logic;
  signal arg_v_42 : integer;
  signal arg_v_43 : integer;
  signal arg_ck_3 : std_logic;
  signal arg_v_36 : integer;
  signal arg_v_37 : integer;
  signal arg_ck_2 : std_logic;
  signal arg_v_38 : integer;
  signal arg_v_39 : integer;
  signal arg_ck_1 : std_logic;
  signal arg_v_40 : integer;
  signal arg_v_41 : integer;
  component int_of_bool
    port (signal clk_1 : in std_logic; signal hw_rst_3 : in std_logic;
          signal rst_2 : in std_logic; signal b : in std_logic;
          signal o_o : out integer);
  end component;
  for int_of_bool4: int_of_bool use entity work.int_of_bool;
  for int_of_bool3: int_of_bool use entity work.int_of_bool;
  for int_of_bool2: int_of_bool use entity work.int_of_bool;
  for int_of_bool1: int_of_bool use entity work.int_of_bool;
begin
  update : process (clk_1, hw_rst_3, z_25, z_26, z_27, z_24, rst_2, event,
```

```
                rst)
  variable h_v_17 : std_logic;
  variable h_v_37 : integer;
  variable h_v_39 : integer;
  variable h_v_43 : integer;
  variable h_v_41 : integer;
  variable h_v_18 : std_logic_vector (0 to 3);
  variable h_v_42 : integer;
  variable h_v_36 : integer;
  variable h_v_38 : integer;
  variable h_v_40 : integer;
  variable h_v_19 : integer_vector (0 to 3);
  variable h_v_35 : integer;
  variable h_v_32 : integer;
  variable h_v_33 : integer;
  variable h_v_34 : integer;
  variable z_28 : integer;
  variable z_29 : integer;
  variable z_30 : integer;
  variable z_31 : integer;
  variable h_v_16 : integer;
  variable pres : integer;
  variable count : integer;
begin
  h_v_17 := (rst_2 or rst);
  h_v_37 := event(3);
  h_v_39 := event(2);
  h_v_43 := event(0);
  h_v_41 := event(1);
  h_v_18 := (others => h_v_17);
  h_v_42 := h_v_18(0);
  h_v_36 := h_v_18(3);
  arg_ck_4 <= clk_1;
  arg_v_42 <= h_v_42;
  arg_v_43 <= h_v_43;
  h_v_38 := h_v_18(2);
  arg_ck_3 <= clk_1;
  arg_v_36 <= h_v_36;
  arg_v_37 <= h_v_37;
  h_v_40 := h_v_18(1);
  arg_ck_2 <= clk_1;
  arg_v_38 <= h_v_38;
  arg_v_39 <= h_v_39;
  arg_ck_1 <= clk_1;
  arg_v_40 <= h_v_40;
  arg_v_41 <= h_v_41;
  h_v_19 := (0 => z_27, 1 => z_26, 2 => z_25, 3 => z_24);
  h_v_35 := h_v_19(0);
  h_v_32 := h_v_19(3);
  h_v_33 := h_v_19(2);
  h_v_34 := h_v_19(1);
  z_28 := (h_v_35 + 0);
  z_29 := (h_v_34 + z_28);
  z_30 := (h_v_33 + z_29);
  z_31 := (h_v_32 + z_30);
  case rst is
    when '1' => h_v_16 := 0;
    when '0' => case rst_2 is
```

```
                  when '1' => h_v_16 := 0;
                  when '0' => h_v_16 := h_v_15;
                  when others => null;
                end case;
      when others => null;
    end case;
    pres := z_31;
    count := (h_v_16 + pres);
    if (hw_rst_3 = '1') then
      h_v_15 <= 0;
    elsif rising_edge(clk_1) then
      h_v_15 <= count;
    end if;
    o_count <= count;
  end process update;
  int_of_bool4: int_of_bool port map (clk_1 => arg_ck_4,
                                      hw_rst_3 => hw_rst_3,
                                      rst_2 => arg_v_42, b => arg_v_43,
                                      o_o => z_24);
  int_of_bool3: int_of_bool port map (clk_1 => arg_ck_3,
                                      hw_rst_3 => hw_rst_3,
                                      rst_2 => arg_v_36, b => arg_v_37,
                                      o_o => z_27);
  int_of_bool2: int_of_bool port map (clk_1 => arg_ck_2,
                                      hw_rst_3 => hw_rst_3,
                                      rst_2 => arg_v_38, b => arg_v_39,
                                      o_o => z_26);
  int_of_bool1: int_of_bool port map (clk_1 => arg_ck_1,
                                      hw_rst_3 => hw_rst_3,
                                      rst_2 => arg_v_40, b => arg_v_41,
                                      o_o => z_25);
end architecture rtl;
```

A.2 Allocateur de ressource

MiniLS initial

MiniLS normalisé, ordonnancé et simplifié

VHDL

```
use work.types.all;

library ieee;
use ieee.std_logic_1164.all;

entity alloc is
  port (signal clk_1 : in std_logic; signal hw_rst_3 : in std_logic;
        signal rst_2 : in std_logic; signal r0 : in std_logic;
        signal r1 : in std_logic; signal o_g0 : out std_logic;
        signal o_g1 : out std_logic);
end entity alloc;

architecture rtl of alloc is
  signal h_v_35 : std_logic;
  signal h_v_42 : st_2;
  signal h_v_43 : std_logic;
```

```
  signal h_v_32 : std_logic;
  signal h_v_33 : std_logic;
  signal h_v_34 : std_logic;
begin
  update : process (clk_1, hw_rst_3, rst_2, r0, r1)
    variable ck_23 : st_2;
    variable r_21 : std_logic;
    variable r_20 : std_logic;
    variable pnr_14 : std_logic;
    variable r_19 : std_logic;
    variable r_22 : std_logic;
    variable h_v_41 : std_logic;
    variable g0 : std_logic;
    variable r_18 : std_logic;
    variable r_17 : std_logic;
    variable r_16 : std_logic;
    variable r_15 : std_logic;
    variable g1 : std_logic;
    variable h_v_36 : std_logic;
    variable h_v_37 : std_logic;
    variable ns_31 : st_2;
    variable nr_30 : std_logic;
    variable h_v_39 : std_logic;
    variable ns_29 : st_2;
    variable nr_28 : std_logic;
    variable h_v_38 : std_logic;
    variable h_v_40 : std_logic;
    variable ns_27 : st_2;
    variable nr_26 : std_logic;
    variable nr_24 : std_logic;
    variable ns_25 : st_2;
    variable nr_13 : std_logic;
    variable ns_11 : st_2;
    variable r_12 : std_logic;
  begin
    case rst_2 is
      when '1' => ck_23 := IDLE0_1;
      when '0' => ck_23 := h_v_42;
      when others => null;
    end case;
    case rst_2 is
      when '1' => r_21 := '1';
      when '0' => r_21 := h_v_33;
      when others => null;
    end case;
    case rst_2 is
      when '1' => r_20 := '1';
      when '0' => r_20 := h_v_34;
      when others => null;
    end case;
    case rst_2 is
      when '1' => pnr_14 := '0';
      when '0' => pnr_14 := h_v_43;
      when others => null;
    end case;
    case rst_2 is
      when '1' => r_19 := '1';
      when '0' => r_19 := h_v_35;
```

```
  when others => null;
end case;
case rst_2 is
  when '1' => r_22 := '1';
  when '0' => r_22 := h_v_32;
  when others => null;
end case;
h_v_41 := r0;
case ck_23 is
  when ALLOC1_1 => g0 := '0';
  when ALLOC0_1 => g0 := '1';
  when IDLE1_1 => g0 := '0';
  when IDLE0_1 => g0 := '0';
  when others => null;
end case;
case ck_23 is
  when ALLOC1_1 => r_18 := '0';
  when ALLOC0_1 => r_18 := r_22;
  when IDLE1_1 => r_18 := r_22;
  when IDLE0_1 => r_18 := r_22;
  when others => null;
end case;
case ck_23 is
  when ALLOC1_1 => r_17 := r_21;
  when ALLOC0_1 => r_17 := '0';
  when IDLE1_1 => r_17 := r_21;
  when IDLE0_1 => r_17 := r_21;
  when others => null;
end case;
case ck_23 is
  when ALLOC1_1 => r_16 := r_20;
  when ALLOC0_1 => r_16 := r_20;
  when IDLE1_1 => r_16 := '0';
  when IDLE0_1 => r_16 := r_20;
  when others => null;
end case;
case ck_23 is
  when ALLOC1_1 => r_15 := r_19;
  when ALLOC0_1 => r_15 := r_19;
  when IDLE1_1 => r_15 := r_19;
  when IDLE0_1 => r_15 := '0';
  when others => null;
end case;
case ck_23 is
  when ALLOC1_1 => g1 := '1';
  when ALLOC0_1 => g1 := '0';
  when IDLE1_1 => g1 := '0';
  when IDLE0_1 => g1 := '0';
  when others => null;
end case;
h_v_36 := (not r1);
h_v_37 := (not r0);
case h_v_36 is
  when '1' => ns_31 := IDLE0_1;
  when '0' => ns_31 := ALLOC1_1;
  when others => null;
end case;
case h_v_36 is
```

```
  when '1' => nr_30 := '1';
  when '0' => nr_30 := '0';
  when others => null;
end case;
h_v_39 := r1;
case h_v_37 is
  when '1' => ns_29 := IDLE1_1;
  when '0' => ns_29 := ALLOC0_1;
  when others => null;
end case;
case h_v_37 is
  when '1' => nr_28 := '1';
  when '0' => nr_28 := '0';
  when others => null;
end case;
h_v_38 := (r0 and (not r1));
h_v_40 := (r1 and (not r0));
case h_v_39 is
  when '1' => ns_27 := ALLOC1_1;
  when '0' => case h_v_38 is
                when '1' => ns_27 := ALLOC0_1;
                when '0' => ns_27 := IDLE1_1;
                when others => null;
              end case;
  when others => null;
end case;
case h_v_39 is
  when '1' => nr_26 := '1';
  when '0' => case h_v_38 is
                when '1' => nr_26 := '1';
                when '0' => nr_26 := '0';
                when others => null;
              end case;
  when others => null;
end case;
case h_v_41 is
  when '1' => nr_24 := '1';
  when '0' => case h_v_40 is
                when '1' => nr_24 := '1';
                when '0' => nr_24 := '0';
                when others => null;
              end case;
  when others => null;
end case;
case h_v_41 is
  when '1' => ns_25 := ALLOC0_1;
  when '0' => case h_v_40 is
                when '1' => ns_25 := ALLOC1_1;
                when '0' => ns_25 := IDLE0_1;
                when others => null;
              end case;
  when others => null;
end case;
case ck_23 is
  when ALLOC1_1 => nr_13 := nr_30;
  when ALLOC0_1 => nr_13 := nr_28;
  when IDLE1_1 => nr_13 := nr_26;
  when IDLE0_1 => nr_13 := nr_24;
```

```
      when others => null;
    end case;
    case ck_23 is
      when ALLOC1_1 => ns_11 := ns_31;
      when ALLOC0_1 => ns_11 := ns_29;
      when IDLE1_1 => ns_11 := ns_27;
      when IDLE0_1 => ns_11 := ns_25;
      when others => null;
    end case;
    if (hw_rst_3 = '1') then
      h_v_35 <= '1';
    elsif rising_edge(clk_1) then
      h_v_35 <= r_15;
    end if;
    if (hw_rst_3 = '1') then
      h_v_42 <= IDLE0_1;
    elsif rising_edge(clk_1) then
      h_v_42 <= ns_11;
    end if;
    if (hw_rst_3 = '1') then
      h_v_43 <= '0';
    elsif rising_edge(clk_1) then
      h_v_43 <= nr_13;
    end if;
    r_12 := pnr_14;
    if (hw_rst_3 = '1') then
      h_v_32 <= '1';
    elsif rising_edge(clk_1) then
      h_v_32 <= r_18;
    end if;
    if (hw_rst_3 = '1') then
      h_v_33 <= '1';
    elsif rising_edge(clk_1) then
      h_v_33 <= r_17;
    end if;
    if (hw_rst_3 = '1') then
      h_v_34 <= '1';
    elsif rising_edge(clk_1) then
      h_v_34 <= r_16;
    end if;
    o_g0 <= g0;
    o_g1 <= g1;
  end process update;
end architecture rtl;
```

B Utilisation du compilateur

Nous supposerons dans ce manuel que l'archive du compilateur a été décompressée dans le répertoire `$HEPTDIR`. Cette archive contient le présent rapport, un répertoire `exs/` avec une batterie d'exemples Heptagon compilés vers VHDL, et un binaire en code-octet pour la machine abstraite OCaml. Ce dernier peut-être exécuté via `ocamlrun heptc`, ou bien directement lorsque votre `ocamlrun` est présent dans `/usr/bin`. Nous supposerons par la suite que c'est le cas et que votre variable d'environnement `$PATH` contient `$HEPTDIR`/bin.

Pour utiliser le compilateur, il faut tout d'abord renseigner la variable d'environnement `$HEPTLIB` spécifiant au compilateur où trouver la bibliothèque standard.

```
$ export HEPTLIB=$HEPTDIR/lib
```

Vous pouvez ensuite vérifier que le compilateur est disponible et fonctionnel via la commande suivante :

```
$ heptc -version
The Heptagon compiler, version 0.4 (wed. aug. 18 11:17:42 CET 2010)
Standard library in [...]
```

Pour compiler un fichier Heptagon, le compilateur doit être invoqué avec l'option `-target`. Les arguments possibles pour cette option sont :

- `vhdl` : génère du code VHDL.
- `mls` : génère le code à flots de données MiniLS intermédiaire.
- `obc` : génère un code impératif simple dans le langage idéalisé Obc.
- `c` : génère du code C.

Les cibles VHDL et C invoquées sur un fichier `source.ept` produisent respectivement un dossier `source_vhdl` et `source_c` qui contiennent les fichiers sources générés. Par exemple :

```
$ cat source.ept
node main() returns (o : int)
let
  o = 0 fby (o + 1);
tel
$ heptc -target vhdl source.ept
$ ls source_vhdl
main.vhd  types.vhd
```

L'option `-s noeud` permet de génèrer le code nécessaire à un exécutable autonome à partir d'un fichier source Heptagon, c'est à dire un *test-bench* dans le cas de VHDL et une fonction `main()` en ce qui concerne C. Voici un exemple d'utilisation de la sortie C :

```
$ cat source.ept
node noeud() returns (o : int)
let
  o = 0 fby (o + 1);
tel

node main() returns (o : int)
let
  o = noeud() + 1;
tel
$ heptc -target c -s main source.ept
$ ls source_c
_main.c  _main.h  source.c  source.h  source_types.c  source_types.h
$ cc -Isource_c source_c/*.c -o source
$ ./source 5 # Option indiquant a l'executable genere de s'arreter apres 5 pas
=> 1
=> 2
=> 3
=> 4
=> 5
```

C Grammaire de Heptagon

Cette grammaire de référence explicite la syntaxe du langage Heptagon.

type	::=	`int` | `bool` | `float` | **type-name**
	|	*type*`^`*constant*
constant	::=	**constant-name**
	|	**integer**
	|	**string**
	|	*constant* + *constant*
	|	*constant* − *constant*
	|	...
expression	::=	*constant*
	|	**variable-name**
	|	*expression* + *expression* | *expression* − *expression* | ...
	|	*iterator* `(`*iterator-argument*`)` << *constant* >> `(`*expression*$^+$`)`
	|	**function-name** *static-parameters* `(`*expression*, ..., *expression*`)`
	|	`last` **variable-name**
	|	`pre` *expression*
	|	*constant* `fby` *expression*
	|	*expression* `->` *expression*
	|	`(`*expression*, ..., *expression*`)`
	|	*expression* = *expression*
	|	`if` *expression* `then` *expression* `else` *expression*
	|	{*field-def*$^+$}
	|	*expression*.**field-name**
	|	{*expression* `with` *field-def*$^+$}
	|	[*expression*, ..., *expression*]
	|	*expression*`^`*constant*
	|	*expression*.[*constant*$^+$]
	|	*expression*.[*constant* .. *constant*]
	|	*expression*.[*expression*$^+$] `default` *expression*
	|	[*expression* `with` *expression*$^+$ = *expression*]
	|	*expression* @ *expression*
static-parameters	::=	ϵ
	|	<< *constant*, ..., *constant* >>
iterator	::=	`map` | `fold` | `mapfold`
iterator-argument	::=	**function-name** *static-parameters* | + | − | ...
field-def	::=	**field-name** = *constant*

equation	::=	`automaton` *state-handler*$^+$ `end`
	|	`switch` *expression* `of` *switch-handler*$^+$
	|	`present` *present-handler*$^+$*optional-present-handler*
	|	`reset` *block* every*expression*
	|	*pattern* = *expression*;
pattern	::=	**variable-name**
	|	(*pattern*, . . . , *pattern*)
switch-handler	::=	| **constructor-name** *block*
present-handler	::=	| *expression block*
optional-present-handler	::=	ϵ
	|	`default` *present-handler*
state-handler	::=	`state` **state-name** *block until-transitions*? *unless-transitions*?
until-transition	::=	`until` *escapes*$^+$
unless-transition	::=	`unless` *escapes*$^+$
escape	::=	*expression* `then` **constructor-name**
	|	*expression* `continue` **constructor-name**
block	::=	*variable-declarations* `do` *equation**
variable-declarations	::=	ϵ
	|	`var` *variable-declaration*$^+$
variable-declaration	::=	**variable-name** : *type*;
fun-or-node	::=	*fun-or-node-kind* **fun-or-node-name**(*variable-declarations*) `returns` (*variable-declarations*) `var` *variable-declarations* `let` *equation** `tel`
fun-or-node-kind	::=	`node` | `fun`
const-decl	::=	`const` *ident* : *type* = *constant*
type-decl	::=	`type` **type-name** = *type-decl-desc*
type-decl-desc	::=	**type-name**
	|	{**tag-name**, . . . , **tag-name**}
	|	{**field-name** : **type-name**, . . . , **field-name** : **type-name**}
program	::=	*const-decl** *type-decl** *fun-or-node**