

# Traduction SCADE/Lustre vers VHDL \*

Adrien Guatto et Marc Pouzet  
LRI <sup>†</sup>

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 <sup>1</sup>. Le prototype Heptagon a été mis à 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.

<sup>†</sup>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.

<sup>1</sup>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.

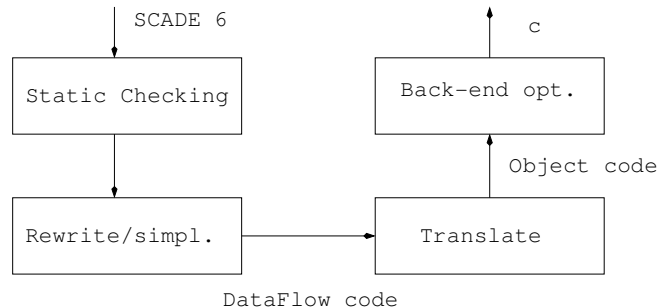


Figure 1: Organisation générale du compilateur de SCADE

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 équation 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-ensem. Le contexte du projet GEN-COD 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'ambiguïté 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 rappelé 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 count(e : bool) returns (o : int)
let
  o = (if e then 1 else 0) + (0 fby o);
tel
```

Le nud *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*. 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'additionner ceux-ci. Notons également l'utilisation de la construction de réinitialisation *reset*, actionnée simplement lorsque l'entrée *rst*<sup>2</sup> est vraie.

#### Allocateur de ressource

```
(* Allocateur de ressource pour deux demandeurs. *)
(* Sret : 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;
```

---

<sup>2</sup>On suppose que le nom *rst* est frais

```

    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 simultanée, le processus qui vera sa requête 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, on le dépouille progressivement de ses constructions de haut niveau par des réécritures successives, 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 présentée à la figure 2.

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.
- D Traduction vers un sous ensemble de VHDL baptisé MiniVHDL.

Le code MiniVHDL obtenu peut ensuite être traité par les outil dédiés.

## 3 De Heptagon à VHDL

### 3.0.1 Syntaxes

**MiniLS** MiniLS est équivalent au langage synchrone à flots de données bien connu qu'est Lustre [2], auquel on adjoint une construction de réinitialisation modulaire d'un noeud<sup>3</sup>.

Les programmes MiniLS sont présents dans le compilateur sous trois formes distinctes ; la première est donnée par souci de cohérence et ne nous intéresse pas directement, les précédentes passes du compilateur nous fournissant directement la deuxième. Ce processus de compilation est décrit en détails dans l'article [1].

- Forme originale 3 telle qu'obtenue à partir du code Heptagon original.
- Forme normale 4.
- Forme finale 5, normalisée, sans réinitialisations (*every*) ni itérateurs, et où tous les paramètres effectifs sont des noms d'identifiants.

On va décrire les constructions du langage :

---

<sup>3</sup>  $f^{ck}(e_1, \dots, e_n)$  **every**  $z$  appelle le noeud  $f$  avec les arguments  $e_1, \dots, e_n$ , et réinitialise la mémoire de celui-ci lorsque  $z$  est vraie.

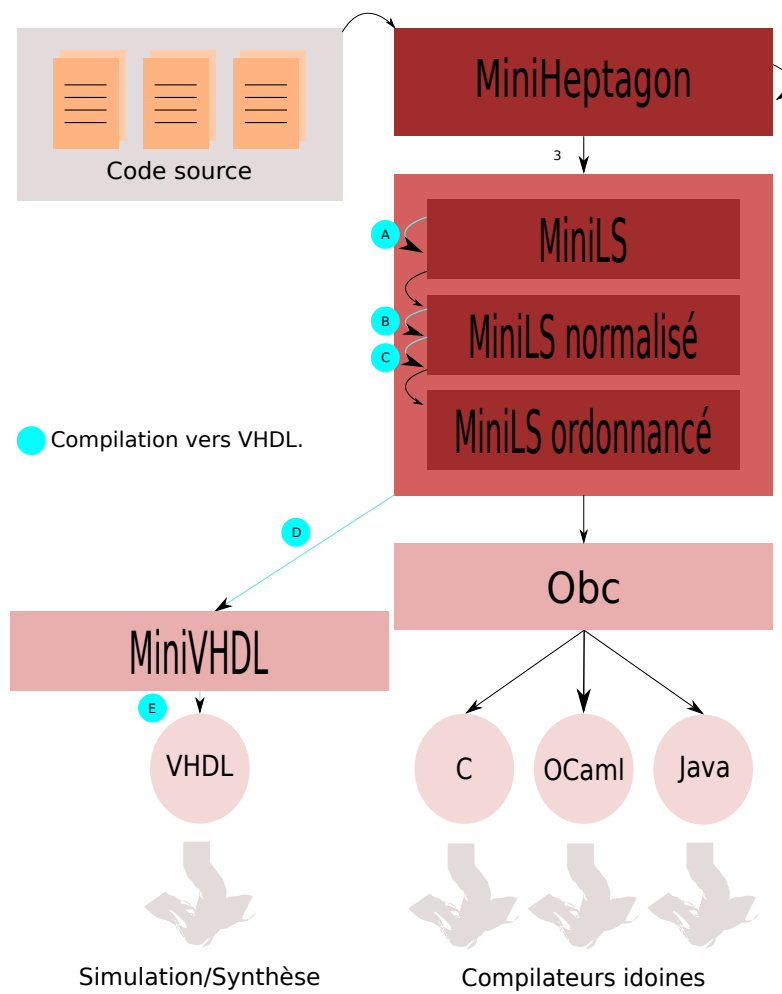


Figure 2: Architecture du compilateur Heptagon

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

Figure 3: MiniLS

$$\begin{aligned}
e &::= x \mid v \mid \text{op}(e, \dots, e) \mid e \text{ when } C(x) \\
ce &::= e \mid \text{merge } x (C \Rightarrow ce) \dots (C \Rightarrow ce) \\
eq &::= x = ce \mid x = v \text{fby}^{ck} e \mid x = \text{pre}^{ck} e \\
&\quad \mid (x, \dots, x) = f^{ck}(e, \dots, e) \text{every } x \\
&\quad \mid (x, \dots, x) = f^{ck}(e, \dots, e) \\
&\quad \mid \text{map } fn(e_1, \dots, e_n) \\
&\quad \mid \text{fold } fn(e_1, \dots, e_n) \\
&\quad \mid \text{mapfold } fn(e_1, \dots, e_n)
\end{aligned}$$

Figure 4: MiniLS normalisé

**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*.

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

### 3.0.2 Simplification de MiniLS normalisé

Nous effectuons donc trois passes pour simplifier le code MiniLS.

**Élimination de la réinitialisation logique** Certaines constructions de MiniLS proposent au programmeur une forme de réinitialisation modulaire : les équations de la forme  $v \text{fby}^{ck} e$  d'un noeud  $f$  instancié par la construction  $f^{ck}(e_1, \dots, e_n) \text{every } z$  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, permet d'éliminer ces constructions *every* dans le but de

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

Figure 5: MiniLS simplifié (et normalisé)

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

Figure 6: MiniVHDL



compiler plus uniformément vers VHDL, autrement dit sans nécessiter un traitement ad-hoc de la réinitialisation.

L'idée est d'ajouter à chaque noeud un argument supplémentaire nommé *rst* correspondant à un appel nécessitant une réinitialisation de la mémoire. La passe se charge de modifier les appels de noeuds pour synthétiser l'expression correcte, en particulier dans le cas d'une équation de type  $(x_1, \dots, x_n) = f^{ck}(e, \dots, e)$  **every**  $x$  où  $x$  doit être capable de réinitialiser  $f$ . Les mémoires réinitialisables  $v$  **fby**<sup>ck</sup>  $e$  doivent bien-entendu être réinitialisées lorsque l'argument **rst** est vrai ; pour cela, il suffit de remplacer ces dernières par **if** *rst* **then**  $v$  **else** **pre**<sup>ck</sup>  $e$ .

Les fonctions suivantes effectuent ces deux tâches : *RstE* traite une expression MiniLS, et *RstNode* ajoute l'argument **rst** à un noeud et transforme ses équations.

$$\begin{aligned}
RstE(\mathbf{op}(e_1, \dots, e_n)) &= \mathbf{op}(RstE(e_1), \dots, RstE(e_n)) \\
RstE(v \mathbf{fby}^{ck} e) &= \mathbf{if} \text{ } rst \text{ then } v \text{ else } v \mathbf{fby}^{ck} RstE(e) \\
RstE(\mathbf{pre}^{ck} e) &= \mathbf{pre}^{ck} RstE(e) \\
RstE(f^{ck}(e_1, \dots, e_n) \mathbf{every} \ x) &= f^{ck}(\mathbf{rst} \ \mathbf{or} \ x, RstE(e_1), \dots, RstE(e_n)) \\
RstE(f^{ck}(e_1, \dots, e_n)) &= f^{ck}(\mathbf{rst}, RstE(e_1), \dots, RstE(e_n)) \\
RstE(\mathbf{if} \ e_1 \ \mathbf{then} \ e_2 \ \mathbf{else} \ e_3) &= \mathbf{if} \ RstE(e_1) \ \mathbf{then} \ RstE(e_2) \\
&\quad \mathbf{else} \ RstE(e_3) \\
RstE(e \ \mathbf{when} \ C(x)) &= RstE(e) \ \mathbf{when} \ C(x) \\
RstE(\mathbf{merge} \ e \ (C_1 \Rightarrow e_1) \ \dots \ (C_n \Rightarrow e_n)) &= \mathbf{merge} \ RstE(e) \ (C_1 \Rightarrow RstE(e_1)) \\
&\quad \dots \\
&\quad (C_n \Rightarrow RstE(e_n)) \\
RstEqs(pat_1 = e_1; \dots; pat_n = e_n) &= \\
\quad pat_1 = RstE(e_1); \dots; pat_n = RstE(e_n) &= \\
RstNode(\mathbf{node} \ f(f) = x_1, \dots, x_n \ \mathbf{with} \ \mathbf{var} \ y_1, \dots, y_n \ \mathbf{in} \ eqs) &= \\
\quad \mathbf{node} \ f(f) = rst, x_1, \dots, x_n \ \mathbf{with} \ \mathbf{var} \ y_1, \dots, y_n \ \mathbf{in} \ ResetEqs(eqs) &=
\end{aligned}$$

**Suppression des itérateurs** La version actuelle du compilateur et de son générateur de code VHDL supporte les tableaux de dimension arbitraire <sup>4</sup> 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  $x = \mathbf{map} \ f n(t_1, \dots, t_m)$  lorsque les tableaux  $t_1, \dots, t_m$  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

```

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

<sup>4</sup>En pratique, les outils de synthèse de circuits à partir de code VHDL imposent une dimension maximale.

```

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

```

**Simplification des appels** Comme nous le verrons plus bas, les appels de nuds seront compilés en instantiations de composants ; or, les arguments d’une construction VHDL *port map* sont forcément des identifiants. Autant se simplifier la tâche en amont : on va donc transformer tout appel de noeud en introduisant une variable intermédiaire pour chaque argument.

$$\begin{aligned}
& \text{Simpl}(x = ce, eqs) &= \\
& \quad (x = ce) :: eqs \\
& \text{Simpl}(x = \text{pre}^{ck} e, eqs) &= \\
& \quad (x = \text{pre}^{ck} e) :: eqs \\
& \text{Simpl}((x_1, \dots, x_n) = f^{ck}(e_1, \dots, e_n), eqs) &= \\
& \quad (y_1 = e_1) :: \dots :: (y_n = e_n) :: ((x_1, \dots, x_n) = f^{ck}(rst, y_1, \dots, y_n)) :: eqs \\
& \quad \text{où } y_1, \dots, y_n \text{ sont des noms de variables frais} \\
& \text{SimplNode}(\text{node } f(x_1, \dots, x_n) = y_1, \dots, y_n \text{ with var } p \text{ in } D) = \\
& \quad \text{node } f(x_1, \dots, x_n) = y_1, \dots, y_n \\
& \quad \text{with var } p' \text{ in fold\_right Simpl } D [] \\
& \quad \text{en supposant que } p' \text{ correspond} \\
& \quad \text{aux variables définies par les nouvelles équations.}
\end{aligned}$$

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

### 3.0.3 MiniLS simplifié 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 *fb* ou *pre*) va correspondre à un signal, chaque équation combinatoire à une variable locale.
- 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 basculement d’un signal précis nommé *hwrst*<sup>5</sup> : on ajoute donc ce signal supplémentaire invisible dans le code MiniLS et on génère le code de réinitialisation correspondant lors du traitement du *fb*.
- Pour respecter la sémantique à  $\Delta$ -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.

---

<sup>5</sup>Tout comme *rst*, *hwrst* est frais.

**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.

**Traduction des constantes et fonction auxiliaires sur les horloges** La fonction *TradConst* traduit une constante MiniLS en constante MiniVHDL.

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

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.

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

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.

$$\begin{aligned} \text{ExpClock}(\text{base}) &= \text{clk} \\ \text{ExpClock}(\text{ck on } C(x)) &= x = \text{TradConst}(C) \text{ and } \text{ExpClock}(\text{ck}) \end{aligned}$$

**Traduction des expressions et équations** La fonction *TradExp* traduit les expressions simples MiniLS normalisés et simplifiés en expressions MiniVHDL.

$$\begin{aligned} \text{TradExp}(v) &= \text{TradConst}(v) \\ \text{TradExp}(x) &= x \\ \text{TradExp}(\text{op}(e_1, \dots, e_n)) &= \text{op}(\text{TradExp}(e_1), \dots, \text{TradExp}(e_n)) \\ \text{TradExp}(e \text{ when } C(x)) &= \text{TradExp}(e) \end{aligned}$$

La fonction *TradCExp* traduit les expressions de contrôle formées d'expressions simples ou de *merge* imbriqués en instructions MiniVHDL.

$$\begin{aligned} \text{TradCExp}(x, \text{merge } y \ (C_1 \Rightarrow ce_1) \ \dots \ (C_n \Rightarrow ce_n)) &= \\ \text{case } y \text{ of } (\text{TradConst}(C_1) \Rightarrow \text{TradCExp}(x, ce_1)) & \\ \dots & \\ (\text{TradConst}(C_n) \Rightarrow \text{TradCExp}(x, ce_n)) & \\ \text{TradCExp}(x, e) &= \\ x ::= \text{TradExp}(e) & \end{aligned}$$

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.

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

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

**Gestion des tableaux** Hormis le cas des itérateurs traités précédemment, la gestion des tableaux n'appelle pas de commentaire particulier, à l'exception de l'anecdotique mais gênante 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 adienne 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<sup>6</sup>.

**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  $\text{GatherPortMaps}(D, i)$  rassemble cette liste de sous-noeuds à partir des appels de noeud présents dans le paquet d'équations  $D$  et crée 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  $\text{MakeArg}$  décrite plus haut. On se donne également une fonction  $\text{GetArgName}(f, n)$  qui renvoie le nom du  $n$ -ème argument du noeud de nom  $f$ .

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

On définit ensuite les fonctions auxiliaires  $\text{NeedVar}$ ,  $\text{Vars}$ ,  $\text{ParamSigs}$  et  $\text{SignalOfVarDec}$  respectivement chargées de déterminer si une équation introduit des déclarations de variables

<sup>6</sup>Notons que notre version de l'outil Xilinx ISE refuse par exemple tout tableau de dimension supérieure à trois.

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).

$$\begin{aligned}
\text{NeedVar}(x = v \mathbf{fby}^{ck} e) &= \text{false} \\
\text{NeedVar}((x_1, \dots, x_n) = f^{ck}(y_1, \dots, y_n)) &= \text{false} \\
\text{NeedVar}(x = ce) &= \text{true} \\
\\
\text{Vars}(x = v \mathbf{fby}^{ck} e) &= [x] \\
\text{Vars}((x_1, \dots, x_n) = f^{ck}(y_1, \dots, y_n)) &= x_1 :: \dots :: x_n \\
\text{Vars}(x = ce) &= [x] \\
\\
\text{ParamSigs}(x = v \mathbf{fby}^{ck} e :: eqs, i) &= \\
\text{ParamSigs}(eqs, i) & \\
\text{ParamSigs}((x_1, \dots, x_n) = f^{ck}(y_1, \dots, y_n) :: eqs, i) &= \\
\text{MakeArg}("clk", i) :: \text{MakeArg}(y_1, i) :: \dots :: \text{MakeArg}(y_n, i) & \\
:: \text{ParamSigs}(eqs, i + 1) & \\
\text{ParamSigs}(x = ce :: eqs, i) &= \\
\text{ParamSigs}(eqs, i) & \\
\\
\text{SignalOfVarDec}(x : bt, mode) &= \text{signal } x : mode \text{ TransBaseType}(bt)
\end{aligned}$$

**Traduction des noeuds** Enfin, *TradNode* se charge de traduire un noeud 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.

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

## 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.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Compilation de SCADE/Lustre vers VHDL . . . . .	1
1.2	Génération de VHDL à partir d'équations data-flow gardées . . . . .	2
1.2.1	Un mot sur la traduction de C vers VHDL . . . . .	2
<b>2</b>	<b>Heptagon</b>	<b>3</b>
2.1	Quelques exemples . . . . .	4
2.2	Architecture du compilateur . . . . .	5
<b>3</b>	<b>De Heptagon à VHDL</b>	<b>6</b>
3.0.1	Syntaxes . . . . .	6
3.0.2	Simplification de MiniLS normalisé . . . . .	8
3.0.3	MiniLS simplifié vers MiniVHDL . . . . .	9
<b>4</b>	<b>Discussion</b>	<b>12</b>
<b>5</b>	<b>Conclusion</b>	<b>12</b>
<b>A</b>	<b>Exemples de code généré</b>	<b>13</b>
A.1	Compteur . . . . .	13
A.2	Allocateur de ressource . . . . .	16
<b>B</b>	<b>Utilisation du compilateur</b>	<b>24</b>

## 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 sont générées ; une passe d'optimisation puissante est en développement.

On liste successivement le code MiniLS obtenu à partir de Heptagon, puis le code MiniLS normalisé et ordonnancé, et enfin le code VHDL.

### A.1 Compteur

#### MiniLS initial

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

**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.

```
node count_23(rst_2 : bool; event : bool^4; rst : bool) returns (count : int)
var _v_43 : int; _v_42 : int; _v_41 : int; _v_40 : int; _v_39 : int;
_v_38 : int; _v_37 : int; _v_36 : int; _v_35 : int; _v_34 : int;
_v_33 : int; _v_32 : int; z_31 : int; z_30 : int; z_29 : int; z_28 : int;
z_27 : int; z_26 : int; z_25 : int; z_24 : int; _v_19 : int^4;
_v_18 : bool^4; _v_17 : bool; _v_16 : int; _v_15 : int; pres : int;
```

```

let
  _v_17 = ( or )(rst_2, rst);
  _v_37 = event[3];
  _v_39 = event[2];
  _v_43 = event[0];
  _v_41 = event[1];
  _v_18 = _v_17^4;
  _v_42 = _v_18[0];
  _v_36 = _v_18[3];
  z_24 = int_of_bool(_v_42, _v_43);
  _v_38 = _v_18[2];
  z_27 = int_of_bool(_v_36, _v_37);
  _v_40 = _v_18[1];
  z_26 = Compteur2.int_of_bool(_v_38, _v_39);
  z_25 = Compteur2.int_of_bool(_v_40, _v_41);
  _v_19 = [z_27; z_26; z_25; z_24];
  _v_35 = _v_19[0];
  _v_32 = _v_19[3];
  _v_33 = _v_19[2];
  _v_34 = _v_19[1];
  z_28 = ( + )(_v_35, 0);
  z_29 = ( + )(_v_34, z_28);
  z_30 = ( + )(_v_33, z_29);
  z_31 = ( + )(_v_32, z_30);
  _v_16 =
    merge rst
      (true -> (0 when true(rst)))
      (false ->
        (merge rst_2 (true -> (0 when true(rst_2)))
          (false -> (_v_15 when false(rst_2)))
          when false(rst)));
  pres = z_31;
  count = ( + )(_v_16, pres);
  _v_15 = 0 fby count;
tel

```

## 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

```
type st_2 = IDLE1_1|IDLE0_1|ALLOC1_1|ALLOCO_1
```

```

node alloc(r0 : bool; r1 : bool) returns (g0 : bool; g1 : bool)
var ns_31 : st_2; nr_30 : bool; ns_29 : st_2; nr_28 : bool; ns_27 : st_2;
  nr_26 : bool; ns_25 : st_2; nr_24 : bool; ck_23 : st_2; pnr_14 : bool;
  nr_13 : bool; r_12 : bool; ns_11 : st_2; r_22 : bool; r_21 : bool;
  r_20 : bool; r_19 : bool; r_18 : bool; r_17 : bool; r_16 : bool;
  r_15 : bool;
let
  r_22 = true fby r_18;
  r_21 = true fby r_17;
  r_20 = true fby r_16;
  r_19 = true fby r_15;
  r_18 =
    merge ck_23
      (ALLOC1_1 -> (false when ALLOC1_1(ck_23)))
      (ALLOCO_1 -> (r_22 when ALLOCO_1(ck_23)))
      (IDLE1_1 -> (r_22 when IDLE1_1(ck_23)))
      (IDLE0_1 -> (r_22 when IDLE0_1(ck_23)));
  r_17 =
    merge ck_23
      (ALLOC1_1 -> (r_21 when ALLOC1_1(ck_23)))
      (ALLOCO_1 -> (false when ALLOCO_1(ck_23)))
      (IDLE1_1 -> (r_21 when IDLE1_1(ck_23)))
      (IDLE0_1 -> (r_21 when IDLE0_1(ck_23)));
  r_16 =
    merge ck_23
      (ALLOC1_1 -> (r_20 when ALLOC1_1(ck_23)))
      (ALLOCO_1 -> (r_20 when ALLOCO_1(ck_23)))
      (IDLE1_1 -> (false when IDLE1_1(ck_23)))
      (IDLE0_1 -> (r_20 when IDLE0_1(ck_23)));
  r_15 =
    merge ck_23
      (ALLOC1_1 -> (r_19 when ALLOC1_1(ck_23)))
      (ALLOCO_1 -> (r_19 when ALLOCO_1(ck_23)))
      (IDLE1_1 -> (r_19 when IDLE1_1(ck_23)))
      (IDLE0_1 -> (false when IDLE0_1(ck_23)));
  nr_13 =
    merge ck_23
      (ALLOC1_1 -> nr_30)(ALLOCO_1 -> nr_28)(IDLE1_1 -> nr_26)
      (IDLE0_1 -> nr_24);
  ns_11 =
    merge ck_23
      (ALLOC1_1 -> ns_31)(ALLOCO_1 -> ns_29)(IDLE1_1 -> ns_27)
      (IDLE0_1 -> ns_25);
  g1 =
    merge ck_23
      (ALLOC1_1 -> (true when ALLOC1_1(ck_23)))
      (ALLOCO_1 -> (false when ALLOCO_1(ck_23)))
      (IDLE1_1 -> (false when IDLE1_1(ck_23)))
      (IDLE0_1 -> (false when IDLE0_1(ck_23)));
  g0 =
    merge ck_23
      (ALLOC1_1 -> (false when ALLOC1_1(ck_23)))
      (ALLOCO_1 -> (true when ALLOCO_1(ck_23)))
      (IDLE1_1 -> (false when IDLE1_1(ck_23)))
      (IDLE0_1 -> (false when IDLE0_1(ck_23)));
  (ns_31, nr_30) =
    if not((r1 when ALLOC1_1(ck_23)))
    then ((IDLE0_1 when ALLOC1_1(ck_23)), (true when ALLOC1_1(ck_23)))

```

```

    else ((ALLOC1_1 when ALLOC1_1(ck_23)), (false when ALLOC1_1(ck_23)));
(ns_29, nr_28) =
  if not((r0 when ALLOC0_1(ck_23)))
  then ((IDLE1_1 when ALLOC0_1(ck_23)), (true when ALLOC0_1(ck_23)))
  else ((ALLOC0_1 when ALLOC0_1(ck_23)), (false when ALLOC0_1(ck_23)));
(ns_27, nr_26) =
  if (r1 when IDLE1_1(ck_23))
  then ((ALLOC1_1 when IDLE1_1(ck_23)), (true when IDLE1_1(ck_23)))
  else if ( & )((r0 when IDLE1_1(ck_23)), not((r1 when IDLE1_1(ck_23))))
  then ((ALLOC0_1 when IDLE1_1(ck_23)), (true when IDLE1_1(ck_23)))
  else ((IDLE1_1 when IDLE1_1(ck_23)), (false when IDLE1_1(ck_23)))
;
(ns_25, nr_24) =
  if (r0 when IDLE0_1(ck_23))
  then ((ALLOC0_1 when IDLE0_1(ck_23)), (true when IDLE0_1(ck_23)))
  else if ( & )((r1 when IDLE0_1(ck_23)), not((r0 when IDLE0_1(ck_23))))
  then ((ALLOC1_1 when IDLE0_1(ck_23)), (true when IDLE0_1(ck_23)))
  else ((IDLE0_1 when IDLE0_1(ck_23)), (false when IDLE0_1(ck_23)))
;
ck_23 = IDLE0_1 fby ns_11;
pnr_14 = false fby nr_13;
r_12 = pnr_14;
tel

```

### MiniLS normalisé, ordonnancé et simplifié

```

type st_2 = IDLE1_1|IDLE0_1|ALLOC1_1|ALLOC0_1
node alloc(rst_2 : bool; r0 : bool; r1 : bool) returns (g0 : bool; g1 : bool)
var _v_43 : bool; _v_42 : st_2; _v_41 : bool; _v_40 : bool; _v_39 : bool;
_v_38 : bool; _v_37 : bool; _v_36 : bool; _v_35 : bool; _v_34 : bool;
_v_33 : bool; _v_32 : bool; ns_31 : st_2; nr_30 : bool; ns_29 : st_2;
nr_28 : bool; ns_27 : st_2; nr_26 : bool; ns_25 : st_2; nr_24 : bool;
ck_23 : st_2; pnr_14 : bool; nr_13 : bool; r_12 : bool; ns_11 : st_2;
r_22 : bool; r_21 : bool; r_20 : bool; r_19 : bool; r_18 : bool;
r_17 : bool; r_16 : bool; r_15 : bool;
let
  ck_23 =
    merge rst_2
      (true -> (IDLE0_1 when true(rst_2)))
      (false -> (_v_42 when false(rst_2)));
  r_21 =
    merge rst_2
      (true -> (true when true(rst_2)))(false -> (_v_33 when false(rst_2)));
  r_20 =
    merge rst_2
      (true -> (true when true(rst_2)))(false -> (_v_34 when false(rst_2)));
  pnr_14 =
    merge rst_2
      (true -> (false when true(rst_2)))(false -> (_v_43 when false(rst_2)));
  r_19 =
    merge rst_2
      (true -> (true when true(rst_2)))(false -> (_v_35 when false(rst_2)));
  r_22 =
    merge rst_2
      (true -> (true when true(rst_2)))(false -> (_v_32 when false(rst_2)));
  _v_41 = (r0 when IDLE0_1(ck_23));
  g0 =
    merge ck_23

```

```

    (ALLOC1_1 -> (false when ALLOC1_1(ck_23)))
    (ALLOCO_1 -> (true when ALLOCO_1(ck_23)))
    (IDLE1_1 -> (false when IDLE1_1(ck_23)))
    (IDLE0_1 -> (false when IDLE0_1(ck_23)));
r_18 =
  merge ck_23
    (ALLOC1_1 -> (false when ALLOC1_1(ck_23)))
    (ALLOCO_1 -> (r_22 when ALLOCO_1(ck_23)))
    (IDLE1_1 -> (r_22 when IDLE1_1(ck_23)))
    (IDLE0_1 -> (r_22 when IDLE0_1(ck_23)));
r_17 =
  merge ck_23
    (ALLOC1_1 -> (r_21 when ALLOC1_1(ck_23)))
    (ALLOCO_1 -> (false when ALLOCO_1(ck_23)))
    (IDLE1_1 -> (r_21 when IDLE1_1(ck_23)))
    (IDLE0_1 -> (r_21 when IDLE0_1(ck_23)));
r_16 =
  merge ck_23
    (ALLOC1_1 -> (r_20 when ALLOC1_1(ck_23)))
    (ALLOCO_1 -> (r_20 when ALLOCO_1(ck_23)))
    (IDLE1_1 -> (false when IDLE1_1(ck_23)))
    (IDLE0_1 -> (r_20 when IDLE0_1(ck_23)));
r_15 =
  merge ck_23
    (ALLOC1_1 -> (r_19 when ALLOC1_1(ck_23)))
    (ALLOCO_1 -> (r_19 when ALLOCO_1(ck_23)))
    (IDLE1_1 -> (r_19 when IDLE1_1(ck_23)))
    (IDLE0_1 -> (false when IDLE0_1(ck_23)));
g1 =
  merge ck_23
    (ALLOC1_1 -> (true when ALLOC1_1(ck_23)))
    (ALLOCO_1 -> (false when ALLOCO_1(ck_23)))
    (IDLE1_1 -> (false when IDLE1_1(ck_23)))
    (IDLE0_1 -> (false when IDLE0_1(ck_23)));
_v_36 = not((r1 when ALLOC1_1(ck_23)));
_v_37 = not((r0 when ALLOCO_1(ck_23)));
ns_31 =
  merge _v_36
    (true -> ((IDLE0_1 when ALLOC1_1(ck_23)) when true(_v_36)))
    (false -> ((ALLOC1_1 when ALLOC1_1(ck_23)) when false(_v_36)));
nr_30 =
  merge _v_36
    (true -> ((true when ALLOC1_1(ck_23)) when true(_v_36)))
    (false -> ((false when ALLOC1_1(ck_23)) when false(_v_36)));
_v_39 = (r1 when IDLE1_1(ck_23));
ns_29 =
  merge _v_37
    (true -> ((IDLE1_1 when ALLOCO_1(ck_23)) when true(_v_37)))
    (false -> ((ALLOCO_1 when ALLOCO_1(ck_23)) when false(_v_37)));
nr_28 =
  merge _v_37
    (true -> ((true when ALLOCO_1(ck_23)) when true(_v_37)))
    (false -> ((false when ALLOCO_1(ck_23)) when false(_v_37)));
_v_38 = ( & )((r0 when IDLE1_1(ck_23)), not((r1 when IDLE1_1(ck_23))));
_v_40 = ( & )((r1 when IDLE0_1(ck_23)), not((r0 when IDLE0_1(ck_23))));
ns_27 =
  merge _v_39
    (true -> ((ALLOC1_1 when IDLE1_1(ck_23)) when true(_v_39)))

```

```

        (false ->
            (merge _v_38
                (true -> ((ALLOC0_1 when IDLE1_1(ck_23)) when true(_v_38)))
                (false -> ((IDLE1_1 when IDLE1_1(ck_23)) when false(_v_38)))
                when false(_v_39)));
nr_26 =
    merge _v_39
        (true -> ((true when IDLE1_1(ck_23)) when true(_v_39)))
        (false ->
            (merge _v_38
                (true -> ((true when IDLE1_1(ck_23)) when true(_v_38)))
                (false -> ((false when IDLE1_1(ck_23)) when false(_v_38)))
                when false(_v_39)));
nr_24 =
    merge _v_41
        (true -> ((true when IDLE0_1(ck_23)) when true(_v_41)))
        (false ->
            (merge _v_40
                (true -> ((true when IDLE0_1(ck_23)) when true(_v_40)))
                (false -> ((false when IDLE0_1(ck_23)) when false(_v_40)))
                when false(_v_41)));
ns_25 =
    merge _v_41
        (true -> ((ALLOC0_1 when IDLE0_1(ck_23)) when true(_v_41)))
        (false ->
            (merge _v_40
                (true -> ((ALLOC1_1 when IDLE0_1(ck_23)) when true(_v_40)))
                (false -> ((IDLE0_1 when IDLE0_1(ck_23)) when false(_v_40)))
                when false(_v_41)));
nr_13 =
    merge ck_23
        (ALLOC1_1 -> nr_30)(ALLOC0_1 -> nr_28)(IDLE1_1 -> nr_26)
        (IDLE0_1 -> nr_24);
ns_11 =
    merge ck_23
        (ALLOC1_1 -> ns_31)(ALLOC0_1 -> ns_29)(IDLE1_1 -> ns_27)
        (IDLE0_1 -> ns_25);
_v_35 = true fby r_15;
_v_42 = IDLE0_1 fby ns_11;
_v_43 = false fby nr_13;
r_12 = pnr_14;
_v_32 = true fby r_18;
_v_33 = true fby r_17;
_v_34 = true fby r_16;
tel

```

## 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 supposons 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 supposons par la suite que c'est le cas et que votre variable d'environnement `$PATH` contient `$HEPTDIRbin/`.

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 générer 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'ex'ecutable g\ 'en\ 'er\ 'e de s'arr\ ^eter apr\ 'es 5 pas
=> 1
```

=> 2  
=> 3  
=> 4  
=> 5

## References

- [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](http://www.lri.fr/~pouzet/lucid-synchrone).