Projet GENCOD

Rapport d'étude sur la traduction de SCADE/Lustre vers VHDL [1]

SP2

[1] Version du document : 31 aout 2010.

Table des matières

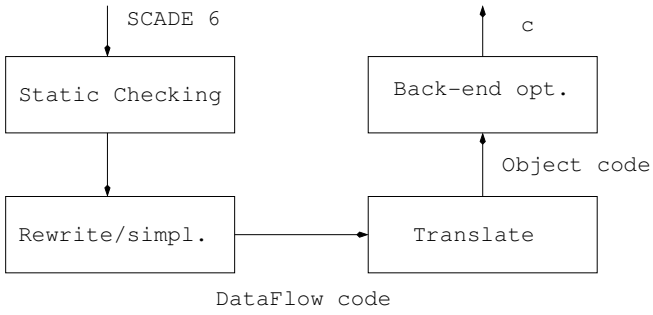

FIG. 1 – Organisation générale du compilateur de SCADE

1 Introduction

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

Le travail présenté ici a été réalisé par Marc Pouzet et Adrien Guatto. Il s'est déroulé au Laboratoire de Recherche en Informatique (LRI) de l'Université Paris-Sud, à Orsay puis à l'École normale supérieure de Paris [2].

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 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 [3]. Le prototype HEPTAGON a été mis a la disposition des partenaires du projet GENCOD.

1.1 Compilation de SCADE/Lustre vers VHDL

Rappelons 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 [4], calcul d'horloges [3], analyse de causalité et analyse d'initialisation [5]) ; 2/ une phase comportant une succession de réécritures produisant à la fin un code data-flow avec horloges ; cette étape élimine les principales structures de contrôle (automates, conditions d'activations) en produisant des équations "gardées" ; 3/ une phase de compilation du code data-flow vers du code impératif séquentiel (object code) ; 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 préliminaires ne sont pas décrites ici.

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

1. à partir du code final généré par le compilateur (dans le cas de KCG, le code C).

[2] Marc Pouzet a été nommé en 2010 à l'Université Pierre et Marie Curie et est 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.

[3] Pour être complet, HEPTAGON est aussi un sous-ensemble du langage LUCID SYNCHRONE [9] que nous avions développé précédemment et qui a servi de cadre d'étude à la conception de SCADE 6. Nous aurions donc pu réaliser un prototype d'étude de la compilation vers VHDL 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 et nous éloignait de ce qui est réalisé dans le compilateur KCG. D'où le choix de considérer un langage simplifié, plus proche de SCADE 6 pour prototyper une éventuelle évolution de KCG.

2. à partir du code intermédiaire data-flow avec horloges, après élimination des structures de contrôle (e.g., automates, conditions d'activation) ;

Remarquons que le code intermédiaire data-flow est déjà un point d'entrée pour les outils de vérification formelle utilisés dans le compilateur KCG (tels que le plug-in de Prover Technology) : la vérification d'une propriété (programmée en SCADE) est obtenue par traduction préalable vers le format data-flow, format qui sert de passerelle vers l'outil Prover.

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

Les deux solutions décrites ci-dessus ont été retenues dans le projet GENCOD. La société GeenSoft a réalisé un compilateur à partir du code C généré par KCG. Nous décrivons ici l'autre solution où le code VHDL est produit directement à partir des équations data-flow. Pour cela, nous décrivons formellement chacune des étapes de traduction, dans une perspective d'intégration dans une chaîne de compilation certifiée.

L'utilisation de Lustre pour la génération de matériel a été envisagée très tôt (thèse de Frédéric Rocheteau [10]). Signalons également que LUSTRE est utilisé dans plusieurs enseignements de matériel [1].

1.2.1 Un mot sur la traduction de C vers VHDL

L'intérêt principal de produire du code VHDL à partir du code C généré par KCG est de ne pas toucher au compilateur existant, déjà qualifié. À condition de qualifier le générateur de code C vers VHDL et de fournir les informations de retour au source (traçabilité), on peut envisager de disposer d'une chaîne complète qualifiée. Discutons ici des points délicats de cette solution.

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) { ... }

void 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->counter_t1` et `self-counter_t2`) 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 à partir du code séquentiel suit le schéma suivant :

1. Une affectation de variable locale est traduite par une équation VHDL sur une variable locale. E.g., x `=` e est traduit en une instruction VHDL x `:=` e. Il faut cependant être attentif à ce que `x` ne génère pas de registre. Ce point est plus délicat qu'il n'y parait, en particulier lors de la traduction d'un programme de la forme `if` *cond* { x `=` *exp* `;` }. Il correspond à la définition d'un flot dont l'horloge est *cond*, c'est-à-dire que x n'est définit qu'aux instant où *cond* est vrai. 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 $x = exp$. Par ailleurs, la sémantique de SCADE garantit qu'aucun calcul de x n'utilise la valeur de x en dehors des instants où *cond* est vrai.

2. Une affectation de variable d'état `self->`t `=` exp doit être traduite vers une instruction de la forme t `<=` exp. Cette affectation doit cependant être exécutée conditionnellement. Il faut donc retrouver, dans le code C, la condition booléenne d'activation de la mise à jour de l'état `self->`t.

3. La compilation des itérateurs est également délicate, en particulier lorsqu'ils sont appliqués à des tableaux de bits. Une équation de la forme :

   ```
   t1 = map not <<10>>(t0)
   ```

   où `t1` et `t0` sont deux tableaux de taille 10 de valeurs booléennes. Le code impératif produit par le compilateur de SCADE a la forme suivante [4].

   ```
   for (i = 0; i < 10; i++) {
     t1[i] = !t0[i]; }
   ```

   Or, le langage VHDL dispose d'opérateurs agissant directement sur les tableaux de bits (e.g., non logique, et logique). La bonne traduction de la première équation est donc :

   ```
   t1 := not t0;
   ```

   La génération de ce code, si elle est triviale à partir du code format data-flow — il suffit de traduire specifiquement l'opération `map not` — est plus délicate à partir du code impératif traduit par KCG. Ceci d'autant plus que KCG applique plusieurs optimisations sur les tableaux afin d'éliminer les copies successives et de fusionner les boucles.

Retenons ici qu'une information précieuse pour la génération de code VHDL a été perdue durant la compilation vers du code séquentiel et doit donc être reconstruite. Nous identifions trois autres difficultés.

- 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 alors décrire ce périmètre ? Dans quel mesure le développement logiciel réalisé s'appliquerait à un code C produit par un autre compilateur (e.g., issu de Simulink) ?

On peut enfin trouver peu naturel de passer par un langage intermédiaire séquentiel (ici C) pour compiler un langage parallèle tel que SCADE vers un langage parallèle tel que VHDL. Ces divers points ont motivé la définition d'une méthode de compilation plus directe, à partir d'un des formats intermédiaire d'un compilateur synchrone. Le format que nous avons choisi est un langage data-flow où les équations sont gardées par des conditions booléennes.

2 Prototypage

Nous avons réalisé un compilateur de référence, appelé HEPTAGON, dans le cadre du projet GENCOD. Le langage d'entrée est un sous-ensemble de SCADE 6 : il permet de combiner des équations de suite telles qu'écrites en LUSTRE avec des structures de contrôle (e.g., automates,

[4] Nous donnons ici la sortie produite par le compilateur de HEPTAGON.

conditions d'activation) et des itérateurs de tableaux (dans l'esprit de [6, 8]). Le langage se veut minimal avant tout afin de pouvoir expérimenter et valider la technique de compilation vers VHDL. Il dispose des principales constructions de SCADE 6. Certaines constructions n'ont cependant pas été intégrées (e.g., signaux, émission sur les transitions). Sa base théorique est décrite dans l'article [2].

2.1 Exemples

Nous donnons ici quelques exemples de programmes écrits dans HEPTAGON. La syntaxe est celle de LUSTRE.

2.1.1 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 noeud `count` compte le nombre d'événements `e` reçus depuis le premier instant du programme.

2.1.2 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'additionner ceux-ci. Notons également l'utilisation de la construction de réinitialisation `reset`, actionnée simplement lorsque l'entrée `rst` est vraie.

2.1.3 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 simultanée, le processus qui verra 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 (typage, analyse de causalité, etc.). Ensuite, la compilation progresse par étapes successives en réécrivant le programme jusqu'à arriver à une forme simplifiée data-flow écrite dans un langage intermédiaire appelé MiniLS. Le processus de compilation de MiniLS vers du code séquentiel est décrit en détails dans l'article [2]. L'architecture du compilateur Heptagon est présentée dans 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, trois simplifications sont appliquées sur le code MiniLS.

A. suppression de la réinitialisation logique ;

B. suppression des itérateurs par expansion de code ;

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, appelé MiniVHDL(à l'étape D), prêt à être traité par les outil dédiés (étape E). Remarquons qu'il n'existe pas de définition précise identifiant un sous-ensemble "synthétisable" du langage Vhdl, géré par les principaux outils industriels. Après échange avec les partenaires du projet GENCOD, MiniVHDL nous parait maintenant correspondre à un sous-ensemble "raisonnable" de Vhdl.

3 De Heptagon à VHDL

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

3.1 Langages internes

3.1.1 MiniLS

MiniLS est un langage data-flow synchrone dans l'esprit de Lustre [7], auquel on adjoint une construction de réinitialisation modulaire et d'écrire des équations gardées par une horloge. La compilation de MiniLS vers Vhdl passe successivement par trois formes distinctes.

1. La forme originale (dont la syntaxe est définie dans la figure 3) telle qu'obtenue à partir du code Heptagon original.
2. Le code est traduit vers une forme normale (figure 4).

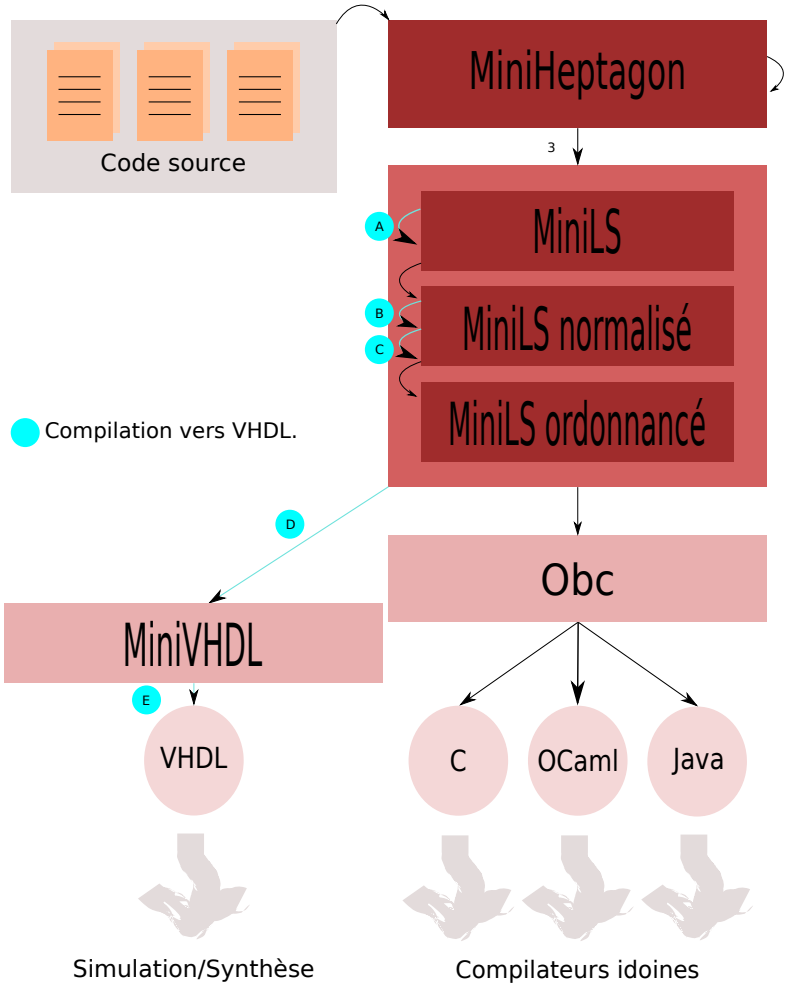

FIG. 2 – Architecture du compilateur HEPTAGON

3. La forme finale (voir figure 5) est une forme normalisée et dans laquelle les réinitialisations et les itérateurs de tableaux ont été éliminés. De plus, les paramètres effectifs des appels de fonctions sont nécessairement des noms de variables.

3.2 Sémantique intuitive

La sémantique de MINILS est celle de LUSTRE : un noeud est composé d'un ensemble d'équations de suites mutuellement récursives. Discutons des points qui distinguent le langage de LUSTRE :

1. Chaque expression e est annotée par une expression d'horloge ck. e doit être calculée lorsque ck est vrai. Cette annotation est produite automatiquement par le compilateur au cours du calcul d'horloges.
2. $f^{ck}(e_1, \ldots, e_n)$ `every` x permet de réinitialiser l'appel de noeud $f^{ck}(e_1, ..., e_n)$ lorsque e est vraie.

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

FIG. 3 – MINILS

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

FIG. 4 – MINILS normalisé

3. Les constructions `when` et `merge` permettent, appliquées à des expressions, de sous-échantillonner et sur-échantillonner ces dernières. 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` $C(x)$. L'expression ck `on` $C(x)$ indique que le résultat est présent lorsque l'horloge ck est vraie et que x est égal à C.

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	

4. Les itérateurs `map`, `fold`, et `mapfold` prennent en argument une fonction f, une taille de tableau et un ou plusieurs tableaux. La sémantique de ces constructions est celle donnée dans [6, 8].
   – `map` applique en parallèle f à tous les éléments des tableaux pour former un nouveau tableau.

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

FIG. 5 – MINILS simplifié (et normalisé)

- `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`).

3.2.1 MiniVHDL

MINIVHDL (défini dans la figure 6) est un fragment de VHDL suffisant pour décrire l'essence du processus de traduction. Un composant MINIVHDL :

$$\texttt{component } f \texttt{ port } P \texttt{ with sig } sigs \texttt{ and var } lvars \texttt{and subcomponents } ports \texttt{ in } I$$

correspond 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 et de 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 précédemment.

3.3 Simplification de MiniLS normalisé

Nous effectuons donc trois passes pour simplifier le code MINILS.

3.3.1 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'identificateur **rst** est un identificateur 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

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

FIG. 6 – MiniVHDL

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 noeud nd et renvoie un nouveau noeud prenant un `rst` à un noeud et transforme ses équations.

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

$$
RstEqs(pat_1 = e_1;\ldots;pat_n = e_n) = pat_1 = RstE(e_1);\ldots;pat_n = RstE(e_n)
$$

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

3.3.2 Suppression des itérateurs

La version actuelle du compilateur et de son générateur de code VHDL supporte les tableaux de dimension arbitraire [5] 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 expansion 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 $x = \texttt{map}\ f\ n\ (t_1,\ldots,t_m)$ 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
```

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
```

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

Un traitement spécial est adopté pour l'utilisation de `map` avec certains opérateurs. En effet, VHDL permet d'utiliser certains opérateurs sur des tableaux de bits, sans avoir à déconstruire ces derniers ; par exemple, il est possible d'effectuer un ET logique bit-à-bit sur deux tableaux t_1 et t_2 via t_1 `and` t_2. La passe effectue la transformation de `map` **op** $<< n >> (t_1, \ldots, t_n)$ à t_1 **op** $\ldots$ **op** t_n lorsque cela est possible.

3.3.3 Simplification des appels

Comme nous le verrons plus loin, les appels de noeuds seront compilés en instantiations de composants. Or, les arguments d'une construction VHDL *port map* sont forcément des identificateurs. Afin de simplifier la génération de VHDL, nous modifiont 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) équation(s) produite(s) à la liste des équations déjà traitées eqs ; $SimplNode(nd)$ prend un noeud nd et simplifie les appels présents dans ses équations. L'opération (::) désigne la concaténation en tête de liste.

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

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

Ces simplifications effectuées, le programme résultant est traduit vers MINIVHDL.

3.4 MiniLS simplifié vers MiniVHDL

Le principe général de la traduction de MINILS simplifié vers MINIVHDL est le suivant :

1. La traduction est appliquée au corps d'un noeud, c'est-à-dire un système d'équation ordonnancé.
2. Chaque noeud MINILS correspond à un composant MINIVHDL.
3. Chaque équation à mémoire (i.e. contenant fby ou pre) correspond à un signal VHDL local et chaque équation combinatoire à la défnition d'une variable locale (une affectation en VHDL).
4. Les horloges importent uniquement pour les équations de la forme :

   $$x = v\, \texttt{fby}^{ck}\, e \quad \text{et} \quad (x_1, \ldots, x_n) = f^{ck}(e_1, \ldots, e_n)$$

   Il faut cependant générer la garde booléenne correspondant à l'horloge logique ck. Les autres équations, même si elles sont gardées par une horloge sont combinatoires et ne nécessitent pas de traitement particulier.
5. La réinitialisation logique est gérée en amont comme expliquée ci-dessus, elle est donc implicitement asynchrone (indépendante des fronts montants de l'horloge).
6. Les partenaires ont exprimé le désir de pouvoir réinitialiser physiquement toute la mémoire lors du basculement d'un signal précis nommé **hwrst** (pour "hardware reset"). Nous ajoutons donc un signal supplémentaire dans dans le code MINILS et on génère le code de réinitialisation correspondant lors du traitement du fby. Tout comme pour l'identificateur **rst** plus haut, on suppose que l'identificateur **hwrst** n'est pas utilisé dans le programme (un renommage préalable permet de l'assurer).

7. 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.
8. 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.
9. 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.

3.4.1 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.

3.4.2 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.

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

3.4.3 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.

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

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.

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

Précisons que par construction, l'interface d'un composant suit toujours la forme suivante :

$$(clk, hwrst, in_1, \ldots, in_n, out_1, \ldots, out_n)$$

Rappelons que le reset "logique" (celui utilisé dans les automates, par exemple ou la construction `every`) est maintenant une entrée supplémentaire (ici in_1).

3.4.4 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 :

1. Calculer la dimension maximale des tableaux rencontrés dans le programme, et utiliser cette information pour pré-déclarer les tableaux Vhdl idoines.
2. 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 [6].

3.4.5 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 instantiés avec des signaux. Nous prenons donc soin d'utiliser des signaux comme résultats mais

[6] Notons que notre version de l'outil Xilinx ISE refuse tout tableau de dimension supérieure à trois.

aussi comme 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.

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

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).

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

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

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

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

3.4.6 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.

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

L'implémentation suit fidèlement les étapes décrites ici. Ces fonctions ont été implémentées dans le compilateur HEPTAGON. Le code chargé de la traduction de MINILS vers MINIVHDL est court (de l'ordre de 600 lignes de code OCaml). Notons que le compilateur réalisé est capable d'assurer le retour au source (traçabilité) depuis le code HEPTAGON (conservation des identificateurs et de leur renommage au cours des étapes de compilation). L'ensemble du compilateur fait de l'ordre de 15000 lignes de code Ocaml.

4 Conclusion

Ce document a décrit une méthode de compilation d'un langage synchrone proche de SCADE à partir d'une représentation intermédiaire data-flow avec horloges. Le résultat est une implémentation décrite formellement et particulièrement petite (en nombre de lignes de code). Nous pensons que la technique proposée ici pourrait être intégrée assez facilement dans un compilateur tel que KCG à partir de son format interne intermédiaire data-flow.

Plusieurs exemples fournis par les membres du projet ont été expérimentés. L'efficacité du code obtenu semble comparable au code produit depuis le code C généré par le compilateur KCG.

Références

[1] Paul Amblard. Using Lustre in Practical Educational Activities : Digital Circuits Design, Formal Languages. In *ETAPS Workshop : Synchronous Language Applications Programming, SLAP 05*, Edinburgh, April 2005.

[2] 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.

[3] Jean-Louis Colaço, Alain Girault, Grégoire Hamon, and Marc Pouzet. Towards a Higher-order Synchronous Data-flow Language. In *ACM Fourth International Conference on Embedded Software (EMSOFT'04)*, Pisa, Italy, september 2004.

[4] Jean-Louis Colaço and Marc Pouzet. Clocks as First Class Abstract Types. In *Third International Conference on Embedded Software (EMSOFT'03)*, Philadelphia, Pennsylvania, USA, october 2003.

[5] Jean-Louis Colaço and Marc Pouzet. Type-based Initialization Analysis of a Synchronous Data-flow Language. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(3) :245–255, August 2004.

[6] Jean-Louis Colaço et Marc Pouzet. Prototypages. Rapport final du projet GENIE II, Verilog SA, Janvier 2000.

[7] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9) :1305–1320, September 1991.

[8] L. Morel. Array iterators in lustre : From a language extension to its exploitation in validation. *EURASIP Journal on Embedded Systems*, 2007.

[9] 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`.

[10] Frédéric Rocheteau and Nicolas Halbwachs. POLLUS : A LUSTRE based hardware design environment. In *Algorithms and Parallel VLSI Architectures*, pages 335–346, 1991.

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

```
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|ALLOC0_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)))
      (ALLOC0_1 -> (r_22 when ALLOC0_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)))
      (ALLOC0_1 -> (false when ALLOC0_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)))
      (ALLOC0_1 -> (r_20 when ALLOC0_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)))
      (ALLOC0_1 -> (r_19 when ALLOC0_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)(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);
  g1 =
    merge ck_23
      (ALLOC1_1 -> (true when ALLOC1_1(ck_23)))
      (ALLOC0_1 -> (false when ALLOC0_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)))
      (ALLOC0_1 -> (true when ALLOC0_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)))
      (ALLOC0_1 -> (true when ALLOC0_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)))
      (ALLOC0_1 -> (r_22 when ALLOC0_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)))
      (ALLOC0_1 -> (false when ALLOC0_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)))
      (ALLOC0_1 -> (r_20 when ALLOC0_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)))
      (ALLOC0_1 -> (r_19 when ALLOC0_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)))
    (ALLOC0_1 -> (false when ALLOC0_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 ALLOC0_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 ALLOC0_1(ck_23)) when true(_v_37)))
    (false -> ((ALLOC0_1 when ALLOC0_1(ck_23)) when false(_v_37)));
nr_28 =
  merge _v_37
    (true -> ((true when ALLOC0_1(ck_23)) when true(_v_37)))
    (false -> ((false when ALLOC0_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 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 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'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**