1 Introduction and tutorial

1.1 Heptagon: short presentation

Heptagon is a synchronous dataflow language, with a syntax allowing the expression of control structures (e.g., switch or mode automata).

A typical Heptagon program will take as input a sequence of values, and will output a sequence of values. Then, variables (inputs, outputs or locals) as well as constants are actually variable or constant streams. The usual operators (e.g., arithmetic or Boolean operators) are applied pointwise on these sequences of values.

For example, the Heptagon program below is composed of one node plus, performing the pointwise sum of its two integer inputs:

```
node plus(x:int;y:int) returns (z:int)
let
  z = x + y;
 tel
```

x and y are the inputs of the node plus; z is the output. x, y and z are of type int, denoting integer streams. z is defined by the equation \( z = x + y \).

An execution of the node plus can then be:

```
  x  1  2  3  4  ...
  y  1  2  1  2  ...
plus(x,y)  2  4  4  6  ...
```

1.2 Required tools

The use of the Heptagon compiler by itself does not require any additional tools. However, the usual use involves a compiler for the generated code (e.g., gcc for C generated code, or javac for Java).

The tools below are optional or are related to some subparts of Heptagon:

- The graphical display tool sim2chro can be obtained from Verimag. It can be used together with the graphical simulator hepts (see Section 1.6).
- Alternatively to sim2chro, the GTKWave wave/chronograms viewer can be used with hepts.

1

\[\text{http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lustre-v4/distrib/}\]
\[\text{http://gtkwave.sourceforge.net}\]
The Sigali tool for model-checking and discrete controller synthesis \[2\] is mandatory for the use of contracts (see Section \[2.8\]). Sigali can be downloaded on the BZR website: \url{http://bzr.inria.fr}.

1.3 Compilation and installation

1.3.1 Required libraries and tools for the compilation

The compilation of the Heptagon compiler requires:

- OCaml \[4\] (version $\geq 3.11$), and the ocamlbuild and ocamlfind tools
- the Menhir \[5\] parser generator
- the ocamlgraph \[6\] library.

The compilation of the Heptagon simulator (optional) requires the LablGTK2 \[7\] library.

1.3.2 Compilation

Once the compiler is downloaded and untarred, go into the \texttt{heptagon/} directory and type:

> ./configure && make

This command will build the compiler, optionally the simulator (if the LablGTK2 library is found), and the standard library.

1.3.3 Installation

After the compilation, or the download and untar of the binary distribution, go into the \texttt{heptagon/} directory and type:

> make install

By default, this will install the binaries (\texttt{heptc} and \texttt{hepts}) into \texttt{/usr/local/bin} and the standard library into \texttt{/usr/local/lib}. Consider the configure script options (\texttt{./configure --help}) for other specific needs.

1.4 Heptagon programs compilation

The Heptagon compiler is named \texttt{heptc}. Its list of options is available by:

> heptc -help

Every options described below are cumulable.

Assuming that the program to compile is in a file named \texttt{example.ept}, then one can compile it by typing:

\url{http://www.irisa.fr/vertecs/Logiciels/sigali.html}
\url{http://caml.inria.fr}
\url{http://gallium.inria.fr/~fpottier/menhir/}
\url{http://ocamlgraph.lri.fr}
\url{http://wwwfun.kurims.kyoto-u.ac.jp/soft/lsl/lablgtk.html}
However, such compilation will only perform standard analysis (such as typing, causality, scheduling) and output intermediate object code, but not any final or executable code.

The Heptagon compiler can thus generate code in some general languages, in order to obtain either a standalone executable, or a linkable library. The target language must then be given by the -target option:

```bash
> heptc -target <language> example.ept
```

Where `<language>` is the name of the target language. For now, available languages are C (c option) and Java (java or java14 option).

### 1.5 Generated code

The generic generated code consists, for each node, of two imperative functions:

- one “reset” function, used to reset the internal memory of the node;
- one “step” function, taking as input the nodes inputs, and whose call performs one step of the node, updates the memory, and outputs the nodes outputs.

A standard way to execute Heptagon program is to compile the generated files together with a main program of the following scheme:

```
call the reset function
for each instant
    get the inputs values
    outputs ← step(inputs)
    do something with outputs values
```

Appendix A give specific technical details for each target language.

### 1.6 Simulation

A graphical simulator is available: hepts. It allows the user to simulate one node by providing a graphical window, where simulation steps can be performed by providing inputs of the simulated node.

This simulator tool interacts with an executable, typically issued of Heptagon programs compilation, and which await on the standard input the list of the simulated node’s inputs, and prints its outputs on the standard output. Such executable, for the simulation of the node $f$, can be obtained by the -s <node> option:

```bash
> heptc -target c -s f -hepts example.ept
```

We can then directly compile the generated C program (whose main function stand in the _main.c file):

```bash
> cd example_c
> gcc -Wall -c -I <path_to_lib>/c example.c
> gcc -Wall -c -I <path_to_lib>/c _main.c
> gcc -o f_sim _main.o example.o # executable creation
```
Where `<path_to_lib>` is the path to the Heptagon library (e.g., `/usr/local/lib/heptagon`), and can be obtained with the command `heptc -where`.

This executable `f_sim` can then be used with the graphical simulator `hepts`, which takes as argument:

- The name of the module (capitalized name of the program without the `.ept` extension),
- the name of the simulated node,
- the path to the executable `f_sim`.

```bash
> hepts -mod Example -node f -exec example_c/f_sim
```

## 2 Syntax and informal semantics

Heptagon programs are synchronous Moore machines, with parallel and hierarchical composition. The states of such machines define dataflow equations.

### 2.1 Nodes

Heptagon programs are structured in nodes: a program is a sequence of nodes. A node is a subprogram with a name `f`, inputs `x_1, ..., x_n`, outputs `y_1, ..., y_p`, local variables `z_1, ..., z_q` and declarations `D`. `y_i` and `z_i` variables are to be defined in `D`, using operations between values of `x_j`, `y_j`, `z_j`. The declaration of one variable comes with its type (`t_i`, `t'_i` and `t''_i` being the type of respectively `x_i`, `y_i` and `z_i`).

```
node f(x_1:t_1;...;x_n:t_n) returns (y_1:t'_1, ..., y_p:t'_p)
var z_1:t''_1, ..., z_q:t''_q;
let D
tel
```

Heptagon allows to distinguish, by mean of clocks and control structures (switch, automata), for declarations and expressions, the discrete instants of activation, when the declarations and expressions are computed and progress toward further states, and other instants when neither computation nor progression are performed.

### 2.2 Expressions

#### 2.2.1 Values and combinatorial operations

Heptagon is a dataflow language, i.e., every value, variable or constant, is actually a stream of value. The usual operators (e.g., arithmetic or Boolean operators) are applied pointwise on these sequences of values, as combinatorial operations (as opposed to sequential operations, taking into account the current state of the program: see delays in Section 2.2.2).

Thus, `x` denotes the stream `x_1, x_2, ...,` and `x + y` is the stream defined by `(x + y)_i = x_i + y_i`.

|   | x1 | x2 | x3 | x4 | ...
|---|----|----|----|----|----
| x |   |    |    |    |    
| y |   |    |    |    |    
| x + y | x1 + y1 | x2 + y2 | x3 + y3 | x4 + y4 | ...
2.2.2 Delays

Delays are the way to introduce some state in a Heptagon program.

- **pre** \(x\) gives the value of \(x\) at the preceding instant. The value at the first instant is undefined.
- \(x \rightarrow y\) takes the value of \(x\) at the first instant, and then the value of \(y\):
- \(x \text{ fby } y\) is equivalent to \(x \rightarrow \text{pre } y\).

\[
\begin{array}{|c|c|c|c|}
\hline
x & x_1 & x_2 & x_3 \\
\hline
y & y_1 & y_2 & y_3 \\
\hline
\text{pre } x & \bot & x_1 & x_2 \\
\hline
x \rightarrow y & x_1 & y_2 & y_3 \\
\hline
x \text{ fby } y & x_1 & y_1 & y_2 \\
\hline
\end{array}
\]

Here is a small example of a node that sums its inputs:

\[
\begin{align*}
\text{node } & \text{sum}(i:\text{int}) \text{ returns } (o:\text{int}) \\
& \text{let} \\
& o = 0 \text{ fby } (o + i) \\
& \text{tel}
\end{align*}
\]

2.2.3 Clocks

It is possible to mix streams with different rates by having streams that are not present at each instant. This can be done using the **when** operator, that samples a stream according to a condition, either a boolean or an enumerated value. If \(x\) is a stream that is always present, then \(x \text{ when } c\) (resp. \(x \text{ when } C(y)\)) is a stream equal to \(x\) but only present when \(c\) is true (resp. \(y = C\)). \(x \text{ when not } c\) is a shortcut for \(x \text{ when } (\neg c)\). The **split** operator allows to sample a stream according to all possible values, i.e. \(\text{split } c \ (x) = x \text{ when } c, x \text{ when not } c \) if \(x\) is a boolean and \(\text{split } y \ (x) = x \text{ when } C_1(y), \ldots, x \text{ when } C_p(y)\) if \(y\) has an enumerated type defined by \(\text{type } t = C_1 \mid \ldots \mid C_p\).

\[
\begin{array}{|c|c|c|c|}
\hline
x \text{ when } c & x_1 & \cdot & x_3 \\
\hline
y & C^c & C & C^c \\
\hline
x \text{ when } C(y) & \cdot & x_2 & \cdot \\
\hline
\end{array}
\]

The clock of an expression (resp. a stream) is a boolean stream that defines the instants when it is present. The clock of the streams that are always present is called base or \(\cdot\). If \(x\) has clock \(ck\), denoted \(x :: ck\), then \(x \text{ when } c\) has clock \(ck \text{ on } c\).

The **merge** operator joins slow streams to create a faster stream.

\[
\begin{array}{|c|c|c|c|}
\hline
x & x_1 & \cdot & x_3 \\
\hline
y & \cdot & y_2 & \cdot \\
\hline
\text{merge } c \ x \ y & x_1 & y_2 & x_3 \\
\hline
\end{array}
\]

**merge** expects complementary streams. If \(z = \text{merge } c \ x \ y\) and \(z :: ck\), then we must have \(x :: ck \text{ on } c\) and \(y :: ck \text{ onot } c\). It is thus different from **if c then x else y** that expects all its arguments to have the same clock. An analysis pass called **clock calculus** checks that these conditions are met.

Here is a first example of a bidirectional counter:

\[
\begin{align*}
\text{node } & \text{sum}(i:\text{int}) \text{ returns } (o:\text{int}) \\
& \text{let} \\
& o = 0 \text{ fby } (o + i) \\
& \text{tel}
\end{align*}
\]
type modes = Up | Down

node two(m: modes; v: int) returns (o: int)
  var x: int;
  x_up: int :: . on Up(m);
  x_down: int :: . on Down(m);
  let
    o = 0 fby x;
    x_up = o when Up(m) + 1;
    x_down = o when Down(m) + 1;
    x = merge m (Up -> x_up) (Down -> x_down)
  tel

Note that clocks are inferred by the compiler, so the clocking annotations are optional.

It is important to understand the interaction between sampling and delays. The value of a delay only changes when it is activated, that is, when its clock is true. As soon as a function \( f \) contains some delay operator, sampling its inputs is not equivalent to sampling its outputs, that is, \( f(x \text{ when } c) \neq (f x) \text{ when } c \).

<table>
<thead>
<tr>
<th>c</th>
<th>true</th>
<th>false</th>
<th>true</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>counter(1)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>counter(1) \text{ when } c</td>
<td>0</td>
<td>.</td>
<td>2</td>
<td>.</td>
</tr>
<tr>
<td>counter(1 \text{ when } c)</td>
<td>0</td>
<td>.</td>
<td>1</td>
<td>.</td>
</tr>
</tbody>
</table>

2.3 Declarations

A declaration \( D \) can be either:

- an equation \( x = e \), defining variable \( x \) by the expression \( e \) at each activation instants;
- a node application \( (y_1, \ldots, y_p) = f(e_1, \ldots, e_n) \), defining variables \( y_1, \ldots, y_p \) by application of the node \( f \) with values \( e_1, \ldots, e_n \) at each activation instants;
- parallel declarations of \( D_1 \) and \( D_2 \), denoted \( D_1; D_2 \). Variables defined in \( D_1 \) and \( D_2 \) must be exclusive. The activation of this parallel declaration activate both \( D_1 \) and \( D_2 \), which are both computed and both progress;
- a switch control structure;
- a present control structure;
- a reset control structure;
- an automaton.

2.3.1 Switch control structures

The switch control structure allows to controls which equations are evaluated, on the basis of the value of one conditional expression:
The conditional expression has to be of type Boolean or enumerated. For Boolean conditions, an alternative syntax is available:

The last keyword defines a memory which is shared by the different modes. Thus, \texttt{last } x \texttt{is the value of the variable } x \texttt{in the previous instant, whichever was the activated mode.}

### 2.3.2 Present control structure

The \texttt{present} control structure allows to control the activation of equations, based on several conditional expressions:

To determine which block is activated, the conditional expressions are evaluated sequentially. The block activated at each instant is the one associated to the first expression evaluated to true.

A \texttt{default} block can also be defined, and will be activated if no conditional expression is true:
var last x:int = 0;
let
  o = x;
present
  | v > 1 do x = last x + v
  | v = 1 do x = last x * 2
  | default do x = last x - v
eend
tel

2.3.3 Reset control structure

The reset/every structure allows to reset a block of declarations to its initial state, when a condition is reached.

node example_reset(v:int) returns (o:int)
  var t : int;
  let
    reset
      t = 0 fby t + v;
      o = 0 fby o + t;
    every o > 42
  tel

definition
  var last x:int = 0;
  let
    o = x;
present
    | v > 1 do x = last x + v
    | v = 1 do x = last x * 2
    | default do x = last x - v
eend
tel

2.3.4 Automata

An automaton is a set of states (one of which being the initial one), and transitions between these states, triggered by Boolean expressions. A declaration is associated to each state. The set of variables defined by the automaton is the union, not necessarily disjoint (variables can have different definitions in different states, and can be partially defined : in this case, when the variable is not defined in an active state, the previous value of this variable is taken.

At each automaton activation instant, one and only one state of this automaton is active (the initial one at the first activation instant). The declaration associated to this active state is itself activated and progress in this activation instant.

Example  The following example gives the node updown. This node is defined by the automaton given in Figure 1 composed of two states:

- the state Up gives to x its previous value augmented of 1
- the state Down gives to x its previous value diminued of 1

This automaton comprises two transitions:

- it goes from Up (the initial state) to Down when x becomes greater or equal than 10;
- it goes from Down to Up when x becomes less or equal 0.
node updown() returns (y:int)
  var last x:int = 0;
  let
  y = x;
  automaton
  state Up
  do x = last x + 1
  until x >= 10 then Down
  state Down
  do x = last x - 1
  until x <= 0 then Up
  end
tel

<table>
<thead>
<tr>
<th>current state</th>
<th>Up</th>
<th>Up</th>
<th>...</th>
<th>Up</th>
<th>Up</th>
<th>Down</th>
<th>Down</th>
<th>Down</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>1</td>
<td>2</td>
<td>...</td>
<td>9</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>...</td>
</tr>
</tbody>
</table>

Expressions on outgoing transitions of this active state are evaluated, so as to compute the next active state: these are weak transitions. Transitions are evaluated in declaration order, in the textual syntax. If no transition can be triggered, then the current state is the next active state.

**Weak and strong transitions.** Transitions between states can be of two types: weak transitions, and strong ones.

**Weak transitions** are denoted `until e then S`. The meaning of such transition is that when `e` becomes true, the current state is executed before leaving it. The target state is only executed at next instant.

**Strong transitions** are denoted `unless e then S`. When `e` becomes true, the automaton instantly leaves the current state, and the target state is executed.

For causality reasons, the condition on strong transitions cannot depend instantly on the variables defined in the current state (as this state is not known and can change because of these transitions). Therefore, the example of the Figure 1 can be written as shown below:
\[ y = x; \]

**automaton**

**state** Up
  \[ \text{do } x = \text{last } x + 1 \]
  unless (last \( x \geq 10 \)) then Down
**state** Down
  \[ \text{do } x = \text{last } x - 1 \]
  unless (last \( x \leq 0 \)) then Up
**end**
*tel*

On this example, the sequence of states is exactly the same as in the previous version; however, the transition from *Up* to *Down* is taken one instant after (i.e., the instant next to the one where \( x = 10 \)). The resulting stream \( y \) is the same:

<table>
<thead>
<tr>
<th>current state</th>
<th>( Up )</th>
<th>( Up )</th>
<th>...</th>
<th>( Up )</th>
<th>( Down )</th>
<th>( Down )</th>
<th>( Down )</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y )</td>
<td>1</td>
<td>2</td>
<td>...</td>
<td>9</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
</tr>
</tbody>
</table>

**Transition and memory.** When a transition (either weak or strong) denoted with *then* is taken, the target state is reset in its initial state: such transitions are said *memoryless*. Conversely, transitions with memory are denoted *until* e continue \( S \) (resp. *unless* e continue \( S \)): in that case, the target state \( S \) is executed without reset.

### 2.4 Structured types

#### 2.4.1 Arrays

Arrays are denoted using the `^` symbol. For instance, `int^10` is the type of arrays of size 10 containing integers. Arrays can be multidimensional, like `int^3^2`. One should note that indices appear in reverse order compared to C: `int^3^2` should be understood as `(int^3)^2`, that is an array of size 2 containing arrays of size 3 (this would have been written `int t[2][3]` in C).

The following operators are defined for arrays:

- Accessing an element at a constant index: \( t[4] \)
- Accessing an element at a dynamic index with a default value: \( z = t[\text{x}] \text{ default } v \) is a stream defined by \( z_i = t_i[x_i] \) if \( 0 \leq x_i < n \) and \( z_i = v_i \) otherwise (where \( n \) is the size of \( t \)).
- Accessing an element at a truncated index: \( z = t[>x<] \) is defined by \( z_i = t_i[\min(\max(0,x_i),n-1)] \) (where \( n \) is the size of \( t \)).
- Modifying an element: \( t' = [ t \text{ with } [x] = v ] \) is a new array equal to \( t \), except for the element at index \( x \) which is equal to \( v \), that is for all \( j \) in \([0,n-1]\) such that \( j \neq x_i, t'_i[j] = t_i[j] \) and \( t'_i[x_i] = v_i \).
- Defining an array by copying one value: \( x^n \) is the array of size \( n \) whose elements are all equal to \( x \).
- Defining an array explicitly: [1, x, 3, y, 5].
**Extracting a slice:** \( t[n..m] \) returns the sub array of \( t \) of size \( m-n+1 \) starting at index \( n \) and ending at index \( m \).

**Concatening arrays:** \( t1@t2 \).

Il est également possible de utiliser les itérateurs pour opérer sur des tableaux. Par exemple, on peut ajouter deux tableaux ponctuellement en faisant \( t = map(<n>>)(+(t1, t2)) \) (où \( n \) est la taille de \( t1, t2 \) et \( t \)). Les itérateurs suivants sont définis :

- **map** applique un nœud ponctuellement aux entrées de tableau et renvoie une ou plusieurs entrées. Si \( f \) a le type \( t1 * t2 \rightarrow t'1 * t'2 \) alors \( map<<n>>f \) a le type \( t1^n * t2^n \rightarrow t'1^n * t'2^n \).

- **mapi** est le même que **map** mais la fonction itérée doit attendre un autre argument entier, qui va être égal à l'indice des entrées de tableau. Si \( f \) a le type \( t1 * t2 * int \rightarrow t'1 * t'2 \) alors \( map<<n>>f \) a le type \( t1^n * t2^n \rightarrow t'1^n * t'2^n \).

- **fold** itère un nœud sur un tableau accumulant des valeurs. Si \( f \) a le type \( t1 * t \rightarrow t \) alors \( fold<<n>>f \) a le type \( t1^n * t \rightarrow t \). Pour l'instance, \( fold<<2>>f(t, 0) \) est équivalent à \( f(t[1], f(t[0], 0)) \).

- **foldi** est à **fold** comme **mapi** est à **map**.

- **mapfold** combine le résultat de **map** et **fold**, en积累ant la valeur d'entrée et renvoyant un nouvel tableau. Si \( f \) a le type \( t1 * t \rightarrow t'1 * t \) alors \( mapfold<<n>>f \) a le type \( t1^n * t \rightarrow t'1^n * t \).

![Figure 2: Itérateurs in HEPTAGON](image)

Itérateurs peuvent également être utilisés pour des tableaux multidimensionnels. Dans le cas de **mapi** et **foldi**, la fonction itérée doit alors attendre un argument pour chaque dimension.

Il est également possible de mimer l'application partielle avec les itérateurs, en donnant une liste d'arguments, pas nécessairement des tableaux, qui deviennent les premiers arguments de chaque application de la fonction itérée. Pour l'instance, \( o = map<<n>>f <(a)>(t) \) est équivalent à \( o[i] = f(a, t[i]) \) pour chaque élément de l'output array \( o \).

Les tableaux sont sémantiquement fonctionnels: chaque modification sur un tableau crée un nouveau tableau. L'implémentation de ces tableaux peut être séparée en arrays de plusieurs façons. Cela peut conduire à des code très inefficace. C'est pourquoi l'optimisation pass **memory allocation** met en place des opérations de partage d'arrays et évite les copies inutiles dans chaque nœud. Il peut être activé en utilisant la commande **memalloc** ou **-O** option.
2.4.2 Records

The syntax for declaring record types is the following:

```
type t = { f1: t1; f2: t2 }
```

Note that two different record types cannot have fields with the same name. It is possible to declare a new record (\( \{ f1 = x; f2 = 9 \} \)), read a field of a record (\( x.f1 \)) and modify one field (\( r' = \{ r \text{ with } .f1 = v \} \) returns a new record where all fields are equal to the ones in \( r \) except for \( f1 \) which is equal to \( v \)).

2.4.3 Types alias

It is possible to declare an alias for any type (like a typedef in C), for instance:

```
type meter = int
type matrix = intˆ10ˆ10
```

This alias can then be used anywhere a type is expected.

2.5 Parametricity

The size of arrays can be parametrized by so-called static expressions, that is expressions that are reduced to constants at compile-time. They are either a literal, a global constant, a static parameter of a node or a static operation on two static expressions (+, -, *, /, ., <=, or, etc). Global constants are declared by:

```
const n : int = 100
const t0 : floatˆn = 1.0ˆn
const r0 = { f1 = 0; f2 = t0 }
```

The parameters of a node are given between << and >> at the declaration of the node and are instantiated with the same syntax:

```
node f<<m:int; t1: intˆn>>(a:intˆm) = (o:intˆm)
let
  o = map<<m>>(+) (a, t1);
tel

node g(a:intˆn) = (o:intˆn)
let
  o = f<<n, t0>>(a);
tel
```

If the backend support parametricity (like in Java), static parameters are kept in the generated code. Otherwise, a pass of the compiler generates all the versions of the node that are needed. If a parametrized node defined in a file \( f1.ept \) is used in \( f2.ept \), it is necessary to first compile \( f1.ept \) with the -c option (and without any -target option), in order to generate a binary file \( f1.epo \). The compilation of the second file, this time with the -target option, will generate code for all the necessary nodes, from \( f1.ept \) and \( f2.ept \).

2.6 Location annotations

Memory allocation avoids unnecessary array copies within the nodes automatically (it can be enable with the -memalloc or -O options). In order to avoid copies when calling nodes, the user must add
location annotations. We will give here only a short introduction, the interested reader can refer to [1] for more details. They express the fact that a function modifies in-place its argument. For instance, consider this node that swaps two elements in an array:

```
node swap(i, j:int; t_in:float^100 at r) returns (t_out:float^100 at r)
var t_tmp:float^100 at r;
let
  t_tmp = [ t_in with [i] = t_in[j] ];
  t_out = [ t_tmp with [j] = t_in[i] ];
tel
```

The location annotations are introduced by the keyword at followed by a location name. All the streams with the same location name are guaranteed to be stored together in the generated code, so the function generated for swap will directly modify its argument in-place.

Located variables are called semilinear: they can only be updated once, but they can be read many times. An update is a function that modify its argument in-place, for instance a node that inputs and outputs located variables, and a read is any other function. For instance, modifying one element in an array is an update, but accessing one element is a read.

Only located variables can be given to a function that expects located arguments. A located variable can only be obtained by updating another located variable or by explicitly initializing a new location with the init construction:

```
node shuffle(i_arr, j_arr : int^m; q : int) = (v : float)
var t, t_prev : float^n at r;
let
  init<<r>> t_prev = t_0 fby t;
  t = fold<<m>> swap(i_arr, j_arr, t_prev);
  v = t[q];
tel
```

2.7 Interfaces

Interface files, with the extension .epi, do not contain any definition but only the declaration (or signatures) of nodes, types and constants. In particular, they are useful to import external functions. For instance:

```
external fun sort(a:int^100) = (o:int^100)
```

The imported function should respect the calling convention given in appendix A. See the directory examples/extern.C for a complete example.

Interface files can be compiled with heptc, in order to obtain a compiled interface (.epci file), necessary for the subsequent use of the declared types and values in other Heptagon programs.

2.8 Contracts for controller synthesis

Contracts are an extension of the Heptagon language, so as to allow to perform discrete controller synthesis (DCS) on Heptagon programs. The extended language is named BZR.

We associate to each node a contract, which is a program associated with two outputs :

- an output $e_A$ representing the environment model ;
an invariance objective $e_G$;

• a set $\{c_1, \ldots, c_n\}$ of controllable variables used for ensuring this objective.

This contract means that the node will be controlled, i.e., that values will be given to $c_1, \ldots, c_n$ such that, given any input trace yielding $e_A$, the output trace will yield the true value for $e_G$.

Contracts are denoted:

\[
\text{node } f(x_1:t_1; \ldots; x_n:t_n) \text{ returns } (y_1:t'_1; \ldots; y_p:t'_p)
\]

\[
\text{contract}
\]

\[
\text{var } \ldots
\]

\[
\text{let } \ldots
\]

\[
\text{tel}
\]

\[
\text{assume } e_A
\]

\[
\text{enforce } e_G
\]

\[
\text{with } (c_1:t''_1; \ldots; c_q:t''_n)
\]

\[
\text{var } \ldots
\]

\[
\text{let}
\]

\[
y_1 = f_1(x_1, \ldots, x_n, c_1, \ldots, c_q);
\]

\[
\vdots
\]

\[
y_p = f_p(x_1, \ldots, x_n, c_1, \ldots, c_q);
\]

\[
\text{tel}
\]

3 Example with Contracts: Multi-task System

3.1 Delayable Tasks

We consider a multi-task system composed of $n$ delayable tasks. Figure 3 shows a delayable task. A delayable task takes three inputs $r$, $c$ and $e$: $r$ is the task launch request from the environment, $e$ is the end request, and $c$ is meant to be a controllable input controlling whether, on request, the task is actually launched (and therefore goes in the active state), or delayed (and then forced by the controller to go in the waiting state by stating the false value to $c$). This node outputs a unique boolean $\text{act}$ which is true when the task is in the active state.

The Figure 4 shows then a node $\text{ntasks}$ where $n$ delayable tasks have been put in parallel. The tasks are inlined so as to be able to perform DCS on this node, taking into account the tasks’ states. Until now, the only interest of modularity is, from the programmer’s point of view, to be able to give once the delayable task code.

This $\text{ntasks}$ node is provided with a contract, stating that its composing tasks are exclusive, i.e., that there are no two tasks in the active state at the same instant. This contract is enforced with the help of the controllable inputs $c_i$.

3.2 Contract composition

We want to reuse the $\text{ntasks}$ node, in order to build modularly a system composed of $2n$ tasks. The Figure 5 shows the parallel composition of two $\text{ntasks}$ nodes. We associate to this composition a new contract, which role is to enforce the exclusivity of the $2n$ tasks.
node delayable(r,c,e:bool) returns (act:bool)
let
    automaton
    state Idle
    do act = false
    until r & c then Active
    | a & not c then Wait
    state Wait
    do act = false
    until c then Active
    state Active
    do act = true
    until e then Idle
end
tel

Figure 3: Delayable task

It is easy to see that the contract of ntasks is not precise enough to be able to compose several of these nodes. Therefore, we need to refine this contract by adding some way to externally control the activity of the tasks.

3.3 Contract refinement

We first add an input c, meant to be controllable. The refined contract will enforce that:

1. the tasks are exclusive,
2. one task is active only at instants when the input c is true. This property, appearing in the contract, allow a node instantiating ntasks to forbid any activity of the n tasks instantiated.

The Figure 6 contains this new ntasks node.

However, the controllability introduced here is know too strong. The synthesis will succeed, but the computed controller, without knowing how c will be instantiated, will actually block every tasks in their idle state. Indeed, if the controller allows one task to go in its active state, the input c can become false at the next instant, violating the property to enforce.

Thus, we propose to add an assumption to this contract: the input c will not become false if a task was active an instant before. This new contract is visible in Figure 7.

We can then use this new ntasks version for the parallel composition, by instantiating the c input by a controllable variable and its negation. This composition can be found in Figure 8.

4 Known issues

Heptagon is a research prototype. Users are kindly invited to submit bugs to the Heptagon developers via heptagon-developers@lists.gforge.inria.fr

Yet, the following subject are known to be issues for the current Heptagon version:

- Discrete controller synthesis (DCS) can be applied only on full Boolean programs (with static integer expressions). If applied on nodes containing non-Boolean expressions or values, an abstraction
node ntasks(r₁,...,rₙ,e₁,...,eₙ:bool)
    returns (a₁,...,aₙ:bool)
contract
let
    ca₁ = a₁ & (a₂ or ... or aₙ);
    ...
    caₙ₋₁ = aₙ₋₁ & aₙ;
    enforce not (ca₁ or \ldots or caₙ₋₁)
with (c₁,...,cₙ:bool)
let
    a₁ = inlined delayable(r₁,c₁,e₁);
    ...
    aₙ = inlined delayable(rₙ,cₙ,eₙ);
    tel

Figure 4: ntasks node: n delayable tasks in parallel

will be computed, removing every equation containing untranslatable value or expression. This abstraction is conservative w.r.t. invariance properties, and does not jeopardize the use of the computed controller, but this controller itself can be less efficient due to the abstraction.

• For the above reason, DCS does not mix well with array types, which will be abstracted.

• Translation to Sigali code (-target z3z option) for DCS application activates the -bool option, for translation of enumerated values into Boolean vectors. This translation cannot be applied on nodes whose signature comprise enumerated clocks, e.g., the node filter below:

```plaintext
type t = A | B

node filter(x:int; c:t) returns (y:int on A(c))
let
    y = x when A(c)
    tel
```

• Nodes whose signature comprise sampled streams cannot be inlined. However, inlining of nodes comprising clocks without appearing in signature is possible.
node main(r₁,...,r₂ₙ,e₁,...,e₂ₙ:bool)  
returns (a₁,...,a₂ₙ:bool)  
contract  
let  
ca₁ = a₁ & (a₂ or ... or a₂ₙ);  
:  
ca₂₂ₙ₋₁ = a₂₂ₙ₋₁ & a₂ₙ;  
tel  
enforce not (ca₁ or ... or ca₂₂ₙ₋₁)  
let  
(a₁,...,aₙ) = ntasks(r₁,...,rₙ,e₁,...,eₙ);  
(aₙ₊₁,...,a₂ₙ) = ntasks(rₙ₊₁,...,r₂ₙ,eₙ₊₁,...,e₂ₙ);  
tel

Figure 5: Composition of two ntasks nodes

node ntasks(c,r₁,...,rₙ,e₁,...,eₙ:bool) returns (a₁,...,aₙ:bool)  
contract  
let  
ca₁ = a₁ & (a₂ or ... or aₙ);...  
caₙ₋₁ = aₙ₋₁ & aₙ;  
one = a₁ or ... or aₙ;  
tel  
enforce not (ca₁ or ... or caₙ₋₁) & (c or not one)  
with (c₁,...,cₙ:bool)  
let  
a₁ = inlined delayable(r₁,c₁,e₁);  
:  
aₙ = inlined delayable(rₙ,cₙ,eₙ);  
tel

Figure 6: First contract refinement for the ntasks node
node ntasks(c, r₁, ..., rₙ, e₁, ..., eₙ:bool) returns (a₁, ..., aₙ:bool)
contract
let
  ca₁ = a₁ & (a₂ or ... or aₙ);
  caₙ₋₁ = aₙ₋₁ & aₙ;
  one = a₁ or ... or aₙ;
  pone = false fby one;
et
assume (not pone or c)
enforce not (ca₁ or ... or caₙ₋₁) & (c or not one)
with (c₁, ..., cₙ)
let
  a₁ = inlined delayable(r₁, c₁, e₁);
  ...
  aₙ = inlined delayable(rₙ, cₙ, eₙ);
et

Figure 7: Second contract refinement for the ntasks node

node main(r₁, ..., r₂ₙ, e₁, ..., e₂ₙ:bool) returns (a₁, ..., a₂ₙ:bool)
contract
let
  ca₁ = a₁ & (a₂ or ... or a₂ₙ);
  ...
  ca₂ₙ₋₁ = a₂ₙ₋₁ & a₂ₙ;
et
enforce not (ca₁ or ... or ca₂ₙ₋₁)
with (c:bool)
let
  (a₁, ..., aₙ) = ntasks(c, r₁, ..., rₙ, e₁, ..., eₙ);
  (aₙ₊₁, ..., a₂ₙ) = ntasks(\Not c, rₙ₊₁, ..., r₂ₙ, eₙ₊₁, ..., e₂ₙ);
et

Figure 8: Two ntasks parallel composition
A Generated code

A.1 C generated code

C generated files from an Heptagon program example.ept are placed in a directory named example.c. This directory contains one file example.c. For each node $f$ of the source program, assuming that $f$ has inputs $(x_1: t_1, \ldots, x_n: t_n)$ and outputs $(y_1: t'_1, \ldots, y_p: t'_p)$, $t_i$ and $t'_i$ being the data types of these inputs and outputs, then the example.c file contains, for each node $f$:

- A `Example__f_reset` function, with an argument `self` being a memory structure instance:
  ```c
  void Example__f_reset(Example__f_mem* self);
  ```

- A `Example__f_step` function, with as arguments the nodes inputs, a structure `out` where the output will be put, and a memory structure instance `self`:
  ```c
  void Example__f_step(t_1 x_1, ..., t_n x_n,
                       Example__f_out* _out,
                       Example__f_mem* self);
  ```

After the call of this function, the structure `out` contains the outputs of the node:

```c
typedef struct {
    t'_1 y_1;
    ...
    t'_p y_p;
} Example__f_out;
```

An example of main C code for the execution of this node would be then:

```c
#include "example.h"

int main(int argc, char * argv[]) {
    Example__f_m mem;
    t_1 x_1;
    ...
    t_n x_n;
    Example__f_out ans;

    /* initialize memory instance */
    f_reset(&mem);

    while(1) {
        /* read inputs */
        scanf("...", &x_1, ..., &x_n);

        /* perform step */
        Example__f_step(x_1, ..., x_n, &ans, &mem);

        /* write outputs */
    }
}
```
printf("...", ans.y1, ..., ans.yp);
}
}

The above code is nearly what is produce for the simulator with the -s option (see Section 1.6).

A.2 Java generated code (Java 1.5 and above)

Java generated files from an Heptagon program example.ept are placed in a directory named java/Example, constituted in a Java package named Example. This directory contains one Java class F (in the file F.java) for each node f of the source program. Assuming that f has inputs (x1 : t1, ..., xn : tn) and outputs (y1 : t′1, ..., yp : t′p), t1 and t′1 being the data types of these inputs and outputs, then this f class implements the following interface:

```java
public interface f {
    public void reset();
    public jeptagon.Pervasives.Tuple n step(t1 x1, ..., tn xn);
}
```

The Tuple class is a n-uple (as a composite of n Object instances). The output y1...yn are accessible through the properties c0...c(n-1) of the Tuple’s instance.

A.3 Java generated code (Java 1.4)

The option -target java14 generated Java 1.4 compliant code (typically, without enumerated types and genericity).

Java generated files from an Heptagon program example.ept are placed in a directory named java/Example, constituted in a Java package named Example. This directory contains one Java class F (in the file F.java) for each node f of the source program. Assuming that f has inputs (x1 : t1, ..., xn : tn) and outputs (y1 : t′1, ..., yp : t′p), t1 and t′1 being the data types of these inputs and outputs, then this f class implements the following interface:

```java
public interface f {
    public void reset();
    public void step(t1 x1, ..., tn xn);
    public t′1 getOutput0();
    ...
}
```

References