Heptagon/BZR manual

June 14, 2012

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 Compilation

The Heptagon compiler is named `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 `example.ept`, then one can compile it by typing :

```
> heptc example.ept
```

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:

```
> 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` option).

1.3 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.

1.4 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:

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

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

```
> cd example_c
> gcc -Wall -c example.c
> gcc -Wall -c _main.c
> gcc -o f_sim _main.o example.o       # executable creation
```

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`.

```
> 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. The Figure 1 gives an example of such program.

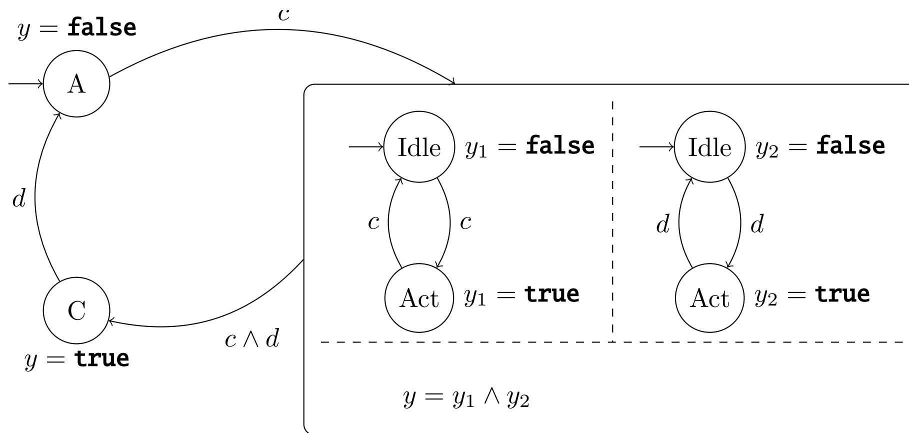


Figure 1: Mixed state and dataflow example

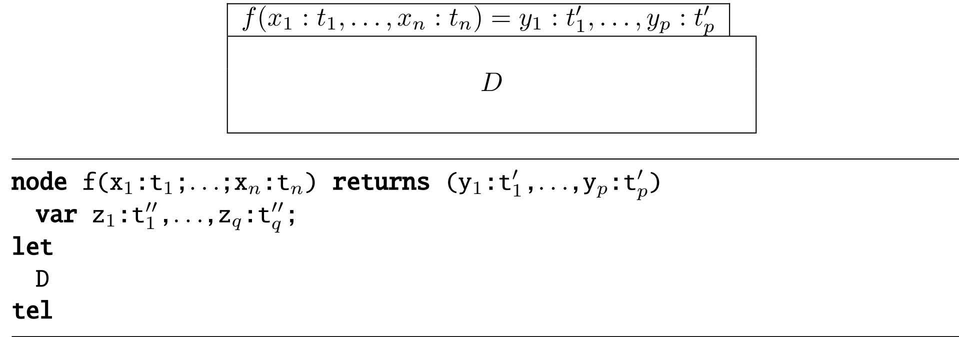
2.1 Nodes

Heptagon programs are structured in *nodes*: a program is a sequence of nodes. A node is a subprogram with a name f, inputs $x_1, \ldots, x_n$, outputs $y_1, \ldots, y_p$, local variables $z_1, \ldots, z_q$ 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. Figure 2 gives the syntax of node definitions, together with a graphical syntax used in this manuel[1]. 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).

The program of the Figure 1 can thus be structured as the semantically equivalent program of the Figure 3. The Figure 4 gives the textual syntax of this program.

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.

[1] declaration of local variables are mandatory for the compiler in the textual syntax, however we will sometimes omit it in the graphical syntax for the sake of brevity




```
node f(x₁:t₁;...;xₙ:tₙ) returns (y₁:t'₁,...,yₚ:t'ₚ)
  var z₁:t''₁,...,z_q:t''_q;
let
  D
tel
```

Figure 2: Graphical and textual syntax of node definition

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 $x_1.x_2\ldots$, and `x + y` is the stream defined by $(\texttt{x + y})_i = x_i + y_i$.

x	x_1	x_2	x_3	x_4	...
y	y_1	y_2	y_3	y_4	...
x + y	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$x_4 + y_4$	...

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 ->y` takes the value of `x` at the first instant, and then the value of `y`;
- `x` **`fby`** `y` is equivalent to `x ->`**`pre`** `y`.

x	x_1	x_2	x_3
y	y_1	y_2	y_3
pre x	$\perp$	x_1	x_2
x ->y	x_1	y_2	y_3
x **fby** y	x_1	y_1	y_2

2.3 Declarations

A declaration D can be either :

- an equation $x = e$, defining variable x by the expression e at each activation instants ;

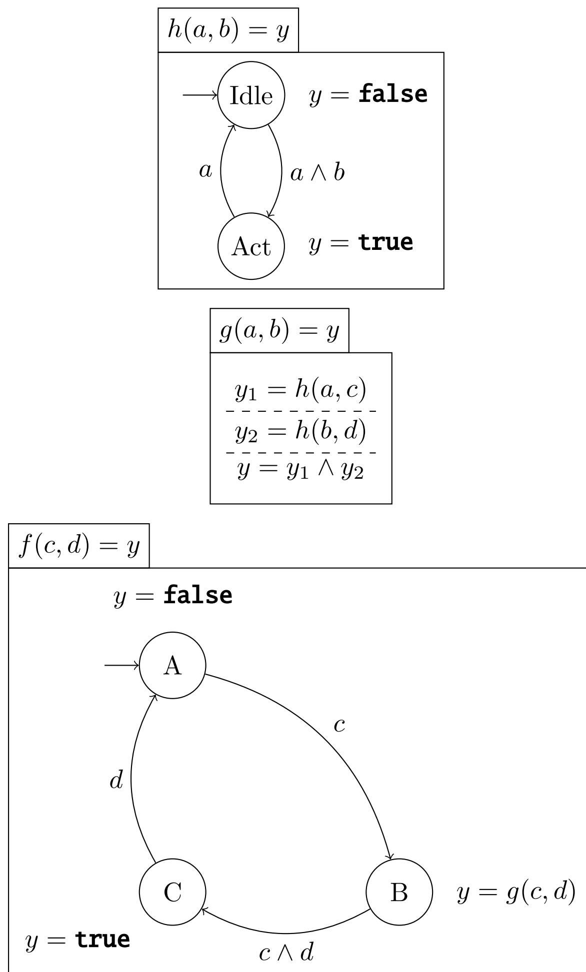


Figure 3: Structured program example

```
node h(a:bool) returns (y:bool)
  let
    automaton
      state Idle
        do y = false
        until a then Active
      state Active
        do y = true
        until a then Idle
    end
  tel

node g (a,b:bool) returns (y:bool)
  var y1,y2 : bool;
  let
    y = y1 & y2;
    y1 = h(a);
    y2 = h(b);
  tel

node f (c,d:bool) returns (y:bool)
  let
    automaton
      state A
        do y = false
        until c then B
      state B
        do y = g(c,d)
        until c & d then C
      state C
        do y = true
        until d then A
    end
  tel
```

Figure 4: Textual syntax

- 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, noted graphically $D_1 \vdots D_2$ and textually 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 ;

- an automaton.

2.3.1 Switch control structures

The `switch` control structure allows to controls which equations are evaluated:

```
type modes = Up | Down

node two(m:modes;v:int) returns (o:int)
  var last x:int = 0;
let
  o = x;
  switch m
  | Up do x = last x + v
  | Down do x = last x - v
  end
tel
```

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

2.3.2 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 an automaton 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
```

current state	*Up*	*Up*	*Up*	*Up*	*Up*	*Up*	*Up*	*Up*	*Up*	*Up*	*Down*	*Down*	*Down*
y	1	2	3	4	5	6	7	8	9	10	9	8	7

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.

3 BZR: Contracts for controller synthesis

Contracts are an extension of the Heptagon language, so as to allow to perform discrete controller synthesis 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.

$f(x_1, \ldots, x_n) = (y_1, \ldots, y_n)$
assume e_A
guarantee e_G **with** $(c_1, \ldots, c_n)$
$y_1 = f_1(x_1, \ldots, x_n, c_1, \ldots, c_n)$ $\ldots$ $y_n = f_n(x_1, \ldots, x_n, c_1, \ldots, c_n)$

In the textual syntax, the contracts are noted :

```
node f(x1:t1;...;xn:tn) returns (y1:t'1;...;yp:t'p)
contract
  var ...
  let
     ...
  tel
  assume eA
  enforce eG
  with (c1:t''1;...;cq:t''n)

var ...
let
  y1 = f1(x1,...,xn,c1,...,cq);
  ⋮
  yp = fp(x1,...,xn,c1,...,cq);
tel
```

4 BZR Running Example: Multi-task System

4.1 Delayable Tasks

We consider a multi-task system composed of n delayable tasks. Figure 5 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 `act` which is true when the task is in the active state.

```
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 5: Delayable task

The Figure 6 shows then a node ntasks where n delayable tasks have been put in parallel. The tasks are inlined so as to be able to perform DSC 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.

```
node ntasks(r_1,...,r_n,e_1,...,e_n:bool)
     returns (a_1,...,a_n:bool)
  contract
  let
    ca_1 = a_1 & (a_2 or ... or a_n);
    ⋮
    ca_{n-1} = a_{n-1} & a_n;
  tel
  enforce not (ca_1 or \ldots or ca_{n-1})
  with (c_1,...,c_n:bool)
let
  a_1 = inlined delayable(r_1,c_1,e_1);
  ⋮
  a_n = inlined delayable(r_n,c_n,e_n);
tel
```

Figure 6: ntasks node: n delayable tasks in parallel

This 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.

4.2 Contract composition

We want know to reuse the ntasks node, in order to build modularly a system composed of $2n$ tasks. The Figure 7 shows the parallel composition of two ntasks nodes. We associate to this composition a new contract, which role is to enforce the exclusivity of the $2n$ tasks.

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.

4.3 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 8 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

```
node main(r_1,...,r_2n,e_1,...,e_2n:bool)
      returns (a_1,...,a_2n:bool)
  contract
  let
    ca_1 = a_1 & (a_2 or ... or a_2n);
    ⋮
    ca_2n-1 = a_2n-1 & a_2n;
  tel
  enforce not (ca_1 or ... or ca_2n-1)
let
  (a_1,...,a_n) = ntasks(r_1,...,r_n,e_1,...,e_n);
  (a_n+1,...,a_2n) = ntasks(r_n+1,...,r_2n,e_n+1,...,e_2n);
tel
```

Figure 7: Composition of two `ntasks` nodes

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 9.

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 10.

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:

```
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`:

```
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:

```
typedef struct \{
  t'_1 y_1;
  ...
```

```
node ntasks(c,r_1,...,r_n,e_1,...,e_n:bool) returns (a_1,...,a_n:bool)
  contract
  let
    ca_1 = a_1 & (a_2 or ... or a_n);...
    ca_{n-1} = a_{n-1} & a_n;
    one = a_1 or ... or a_n;
  tel
  enforce not (ca_1 or ... or ca_{n-1}) & (c or not one)
  with (c_1,...,c_n:bool)
let
  a_1 = inlined delayable(r_1,c_1,e_1);
  ⋮
  a_n = inlined delayable(r_n,c_n,e_n);
tel
```

Figure 8: First contract refinement for the ntasks node

```
    t'_p y_p;
  \} Example__f_ans;
```

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

```
#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.y_1, ..., ans.y_p);
  \}
\}
```

The above code is nearly what is produce for the simulator with the `-s` option (see Section 1.4).

```
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;
  tel
  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);
tel
```

Figure 9: Second contract refinement for the `ntasks` node

A.2 Java generated code

Java generated files from an Heptagon program `example.ept` are placed in a directory named `example_java`. 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 $(x_1 : t_1, \dots, x_n : t_n)$ and outputs $(y_1 : t'_1, \dots, y_p : t'_p)$, t_i and t'_i being the data types of these inputs and outputs, then this `f` class implements the following interface:

```
public interface f {

   public void reset();

   public fAnswer step(t1 x1, ..., tn xn);
}
```

The `fAnswer` class being a structure containing the outputs:

```
public class fAnswer {
  t'1 y1;
  ...
  t'p yp;
}
```

```
node main(r_1,...,r_2n,e_1,...,e_2n:bool) returns (a_1,...,a_2n:bool)
  contract
  let
    ca_1 = a_1 & (a_2 or ... or a_2n);
    ⋮
    ca_2n-1 = a_2n-1 & a_2n;
  tel
  enforce not (ca_1 or ... or ca_2n-1)
  with (c:bool)
let
  (a_1,...,a_n) = ntasks(c,r_1,...,r_n,e_1,...,e_n);
  (a_n+1,...,a_2n) = ntasks(\Not c,r_n+1,...,r_2n,e_n+1,...,e_2n);
tel
```

Figure 10: Two ntasks parallel composition