# Programming and controller synthesis with Heptagon/BZR and ReaX

Nicolas Berthier    Gwenaël Delaval

## 1 Introduction

Heptagon/BZR[1][2] is a reactive language, belonging to the synchronous languages family, whose main feature is to include discrete controller synthesis within its compilation.

It is equipped with a behavioral contract mechanisms, where assumptions can be described, as well as an "enforce" property part: the semantics of this latter is that the property should be enforced by controlling the behaviour of the node equipped with the contract. This property will be enforced by an automatically built controller, which will act on free controllable variables given by the programmer.

ReaX[2][1] is a controller synthesis tool, dedicated to the control of logico-numerical programs.

This report documents the integration of these two tools.

## 2 Heptagon/BZR in a (small) nutshell

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.

Heptagon programs are structured in *nodes*, which are provided with *inputs* and *outputs*, and possibly local variables. A node is defined by mean of a set of *parallel equations*, defining the local variables and outputs' current values, as functions of current inputs, other variables' values, and current state.

Figure 1 shows a short Heptagon example. It consists of a two-mode automaton, with a mode *Up* in which the output is increased by the current input value, and a mode *Down* in which the output is decreased. The output `y`'s current value is defined as the current value of the local variable `x`, whose last value is recorded sp as tp be used in the computation performed on the next step. In the `Up` (resp. `Down`) mode, `x` is defined as the value of `x` at the previous instant, increased (resp. decreased) by the current value of the input `v`. The automaton stays in the `Up` mode until `x` is greater or equal 10; meaning that when this condition is true, the next mode will be the mode `Down`.

The Heptagon/BZR language provides a *contract* construct, allowing the expression of assumptions on the environment (inputs) (`assume` keyword), and guaranteed or enforced properties (`enforce` keyword) on the outputs.

For example, the `twomodes` node given above can be enriched with a contract, saying that if for every instant, $0 \leq \texttt{v} \leq 1$, then the property $0 \leq \texttt{twomodes(v)} \leq 10$ will always hold (Fig. 2).

Contracts can also be provided with a declaration of *controllable variables*: these variables are local to the node, and their value can be used into it. However, these variables are not defined by the programmer. Their value will be given, during execution, by a *controller*, which will be computed offline by a *controller synthesis tool*.

Figure 3 shows how the `twomodes` node can be equipped with a controllable variable `c`, which will *control* the transition between the two modes.

We will see in the following sections how these contracts can be handled by the ReaX verification and synthesis tool.

[1] `http://bzr.inria.fr`

[2] `http://reatk.gforge.inria.fr`

```
node twomodes(v:int) returns (y:int)
var last x : int = 0;
let
  y = x;
  automaton
    state Up
      do x = last x + v
      until x >= 10 then Down
    state Down
      do x = last x - v
      until x <= 0 then Up
  end
tel
```

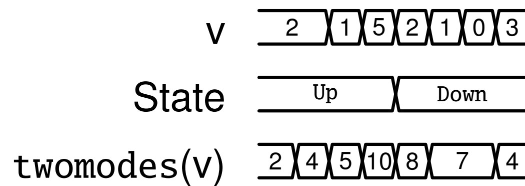


Figure 1: Heptagon short example

```
node twomodes (v:int) = (y:int)

contract
  assume (v <= 1) & (v >= 0)
  enforce (o <= 10) & (o >= 0)

var last x : int = 0;
let
  y = x;
  automaton
    state Up
      do x = last x + v
      until x >= 10 then Down
    state Down
      do x = last x - v
      until x <= 0 then Up
  end
tel
```

Figure 2: Example of contract in Heptagon/BZR

```
node twomodes (v:int) = (y:int)

contract
  assume (v <= 1) & (v >= 0)
  enforce (o <= 10) & (o >= 0)
  with (c:bool)

var last x : int = 0;
let
  y = x;
  automaton
    state Up
      do x = last x + v
      until c then Down
    state Down
      do x = last x - v
      until c then Up
  end
tel
```

Figure 3: Contract with one controllable variable

# 3 Integration of Reax and Heptagon/BZR

## 3.1 Compilation chain

Figure 4 describes the full compilation process, involving the Heptagon/BZR compiler (`heptc`) and the ReaX controller synthesis tool.

The Heptagon compiler is usually used to generate target code in a general-purpose language, like C (option `-target c`) or Java (option `-target java`). This generated code is composed of two functions for each Heptagon:

- a *reset* function, used to reset the node's state ;
- a *step* function, which takes as input the current inputs of the node, update the current state, and computes the current outputs.

If a node is provided with a contract, then the ReaX verification/synthesis tool can be used. The Heptagon backend towards Ctrl-n equations (input format for ReaX) is activated with the `-target ctrln` option.

The ReaX tool can then be used to synthesize (generate automatically) a controller which, composed with the initial program, will give values to the controllable variables so that the "enforce" property stated by the contract is verified.

This controller is initially a Boolean predicate, over the values of inputs, current state and controllable variable. The ReaX option `-triang` allows the obtention of a function (in the same Ctrl-n input format), giving values to the controllable variables, functions of values of current inputs and state.

This controller function can then be translated into an Heptagon node by the `ctrl2ept` tool. This Heptagon node is composed in parallel with the initial one, by the technical mean of a node instanciation. Thus, the generated code of this controller can be compiled and linked with the code generated from the initial Heptagon program.

# 4 Verification of logico-numerical Heptagon programs with ReaX

Heptagon/BZR contracts can be used, with the ReaX tool, to verify logico-numerical programs.

Let us look back at the example given in Figure 2, and name this program Modes (in a file named `modes.ept`).

```
node twomodes (v:int) = (y:int)

contract
  assume (v <= 1) & (v >= 0)
  enforce (o <= 10) & (o >= 0)
```

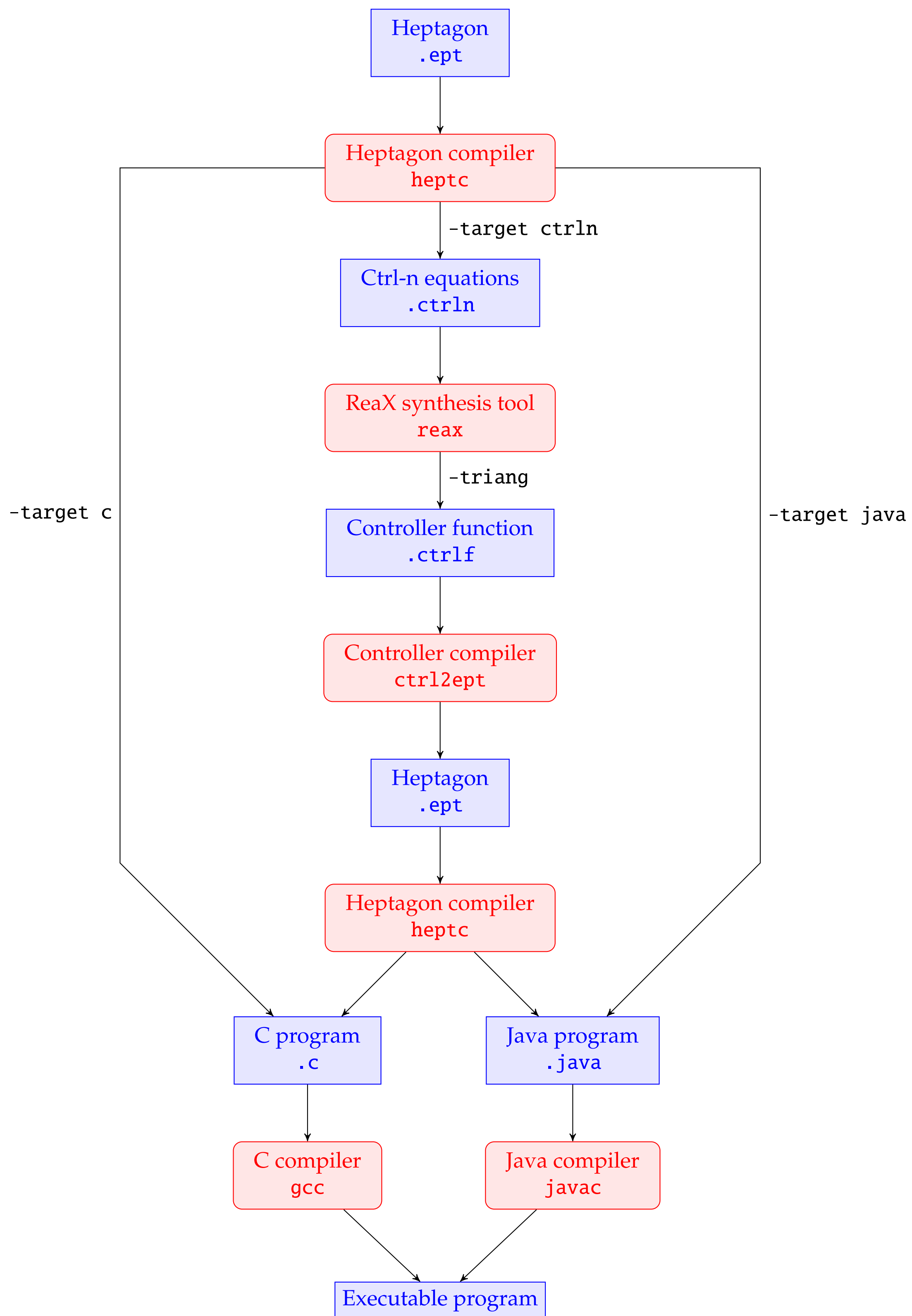


Figure 4: BZReaX full compilation chain

```
var last x : int = 0;
let
  y = x;
  automaton
    state Up
      do x = last x + v
      until x >= 10 then Down
    state Down
      do x = last x - v
      until x <= 0 then Up
  end
tel
```

We now want to check that the property stated by the contract is guaranteed by the program. We begin then by compiling this program towards the Ctrl-n input format :

```
> heptc -target ctrln modes.ept
```

We obtain then a Ctrl-n program placed in a file named `modes_ctrln/twomodes.ctrln`. This file can be given as input to the ReaX tool:

```
> reax -a 'sS:d={P:D}' modes_ctrln/twomodes.ctrln
[0.008 I Main] Reading node from ‘modes_ctrln/twomodes.ctrln’...
[0.024 I Supra] Variables(bool/num): state=(4/2), i=(0/1), u=(0/1), c=(0/0)
[0.024 I Df2cf] Preprocessing: discrete program
[0.024 I Verif] Forcing selection of power domain.
[0.024 I Synth] logico-numerical synthesis with powerset extension of power
                domain over strict convex polyhedra with BDDs:
[0.068 I sB] Building controller...
[0.072 I sB] Computing boundary transtions...
[0.072 I sB] Simplifying controller...
[0.072 I Synth] logico-numerical synthesis with powerset extension of power
                domain over strict convex polyhedra with BDDs succeeded.
[0.072 I Main] Extracting generated controller...
[0.072 I Main] Checking generated controller...
[0.072 I Main] Outputting into ‘modes_ctrln/twomodes.ctrlr’...
```

The option `-a` followed by the string `'sS:d={P:D}'` defines the algorithme used for the verification. The algorithms available are:

- `'sB'` for Boolean verification/synthesis (for Boolean programs, or whose Boolean abstraction is meaningful)
- `'sS'` for verification/synthesis using abstract interpretation (over-approximation). The option `'d={...}'` allows the selection of abstract domains :
  - `I` selects the domain of *intervals*, for programs with comparisons or operations between variables and constants ;
  - `P` selects the domain of *convex polyhedra*, suitable for programs with comparisons or simple operations (sum or difference) between variables.

  The second part (`...:D`) of this string selects the *powerset extension* of the power domain over the abstract domain, suitable for programs with mixed Boolean, modes and numerical operations on modes.

Our program involves operation between a state variable and an input (`last x + v`); thus we need to use the convex polyhedra domain. Using a less precise domain such as intervals would lead to a failed verification:

```
> reax -a 'sS:d=I:D' modes_ctrln/twomodes.ctrln
[0.012 I Main] Reading node from ‘modes_ctrln/twomodes.ctrln’...
[0.024 I Supra] Variables(bool/num): state=(4/2), i=(0/1), u=(0/1), c=(0/0)
[0.024 I Df2cf] Preprocessing: discrete program
[0.024 I Verif] Forcing selection of power domain.
[0.025 I Synth] logico-numerical synthesis with powerset extension of power
                domain over intervals with BDDs:
[0.033 I Synth] logico-numerical synthesis with powerset extension of power
                domain over intervals with BDDs failed.
```

## 5 Controller synthesis on logico-numerical Heptagon programs with ReaX

We add now a controllable variable to our `twomodes` node. This controllable variable `c` will be used to control the transition between the two modes:

```
node twomodes (v:int) = (y:int)

contract
  assume (v <= 1) & (v >= 0)
  enforce (o <= 10) & (o >= 0)
  with (c:bool)

var last x : int = 0;
let
  y = x;
  automaton
    state Up
      do x = last x + v
      until not c then Down
    state Down
      do x = last x - v
      until not c then Up
  end
tel
```

The transition between `Up` and `Down` is no longer defined as a condition on the current value of `x`, but controlled by the controller, via the given value of `c`.

The full compilation of this program consists in:

1. Heptagon compilation towards C code and Ctrl-n equations; the `-s` option selects the simulated node (`twomodes`)

   ```
   > heptc -hepts -s twomodes -target c -target ctrln modes.ept
   ```

2. Controller synthesis with ReaX (`-triang` option to obtain a function)

   ```
   > reax -a 'sS:d={P:D}' --triang modes_ctrln/twomodes.ctrln
   [0.008 I Main] Reading node from 'modes_ctrln/twomodes.ctrln'...
   [0.024 I Supra] Variables(bool/num): state=(4/2), i=(1/1), u=(0/1), c=(1/0)
   [0.024 I Df2cf] Preprocessing: discrete program
   [0.024 I Verif] Forcing selection of power domain.
   [0.024 I Synth] logico-numerical synthesis with powerset extension of power
                   domain over strict convex polyhedra with BDDs:
   [0.072 I sB] Building controller...
   [0.072 I sB] Computing boundary transtions...
   [0.072 I sB] Simplifying controller...
   [0.072 I Synth] logico-numerical synthesis with powerset extension of power
                   domain over strict convex polyhedra with BDDs succeeded.
   [0.092 I t.] Triangulation...ng...
   [0.092 I Main] Extracting triangularized controller...
   [0.092 I Main] Checking triangularized controller...
   [0.092 I Main] Splitting triangularized controller...
   [0.092 I Main] Extracting split triangularized controller...
   [0.092 I Main] Checking split triangularized controller...
   [0.092 I Main] Outputting into 'modes_ctrln/twomodes.0.ctrls'...
   ```

3. Translation of the controller towards Heptagon

   ```
   > ctrl2ept -n Modes.twomodes
   ```

4. Compilation of the controller towards C

   ```
   > heptc -target c modes_controller.ept
   ```

5. Compilation and linking of all C files, obtaining an executable file for the simulation

```
> gcc -o sim -I/usr/local/lib/heptagon/c -Imodes_c -Imodes_controller_c
             modes_c/_main.c modes_c/modes.c modes_c/modes_types.c
             modes_controller_c/modes_controller.c
             modes_controller_c/modes_controller_types.c
```

The program can then be simulated:

v 0 1 0 1 0 1

twomodes(v) 0 1 2 3 4 5 6 7 8 9 10 9 8 7 6 5 4 3 2 1 0 1 2

# 6 Encapsulation: the BZReaX script

The full compilation chain described in the previous section has been encapsulated into a script named `bzreax`, whose scope is described in Figure 5.

Thus, the full compilation of the program described in pthe previous section can be obtained by:

```
> bzreax modes.ept twomodes -a 'sS:d={P:D}' -s
[0.008 I Main] Reading node from ‘modes_ctrln/twomodes.ctrln’...
[0.020 I Supra] Variables(bool/num): state=(4/2), i=(1/1), u=(0/1), c=(1/0)
[0.020 I Df2cf] Preprocessing: discrete program
[0.020 I Verif] Forcing selection of power domain.
[0.020 I Synth] logico-numerical synthesis with powerset extension of power
                domain over strict convex polyhedra with BDDs:
[0.056 I sB] Building controller...
[0.060 I sB] Computing boundary transtions...
[0.060 I sB] Simplifying controller...
[0.060 I Synth] logico-numerical synthesis with powerset extension of power
                domain over strict convex polyhedra with BDDs succeeded.
[0.072 I t.] Triangulation...ng...
[0.072 I Main] Extracting triangularized controller...
[0.072 I Main] Checking triangularized controller...
[0.072 I Main] Splitting triangularized controller...
[0.072 I Main] Extracting split triangularized controller...
[0.072 I Main] Checking split triangularized controller...
[0.072 I Main] Outputting into ‘modes_ctrln/twomodes.0.ctrls’...
Info: Loading module of controllers for node Modes.twomodes...
Info: Reading function from ‘modes_ctrln/twomodes.0.ctrls’...
Info: Outputting into ‘modes_controller.ept’...
Info: To launch the simulator, run: ‘hepts -mod Modes -node twomodes -exec ./sim’
```

# References


[1] Nicolas Berthier and Hervé Marchand. Discrete controller synthesis for infinite state systems with ReaX. In *12th International Workshop on Discrete Event Systems*, pages 46–53, Cachan, France, 2014.

[2] Gwenaël Delaval, Éric Rutten, and Hervé Marchand. Integrating Discrete Controller Synthesis into a Reactive Programming Language Compiler. *Discrete Event Dynamic Systems*, 23(4):385–418, 2013.

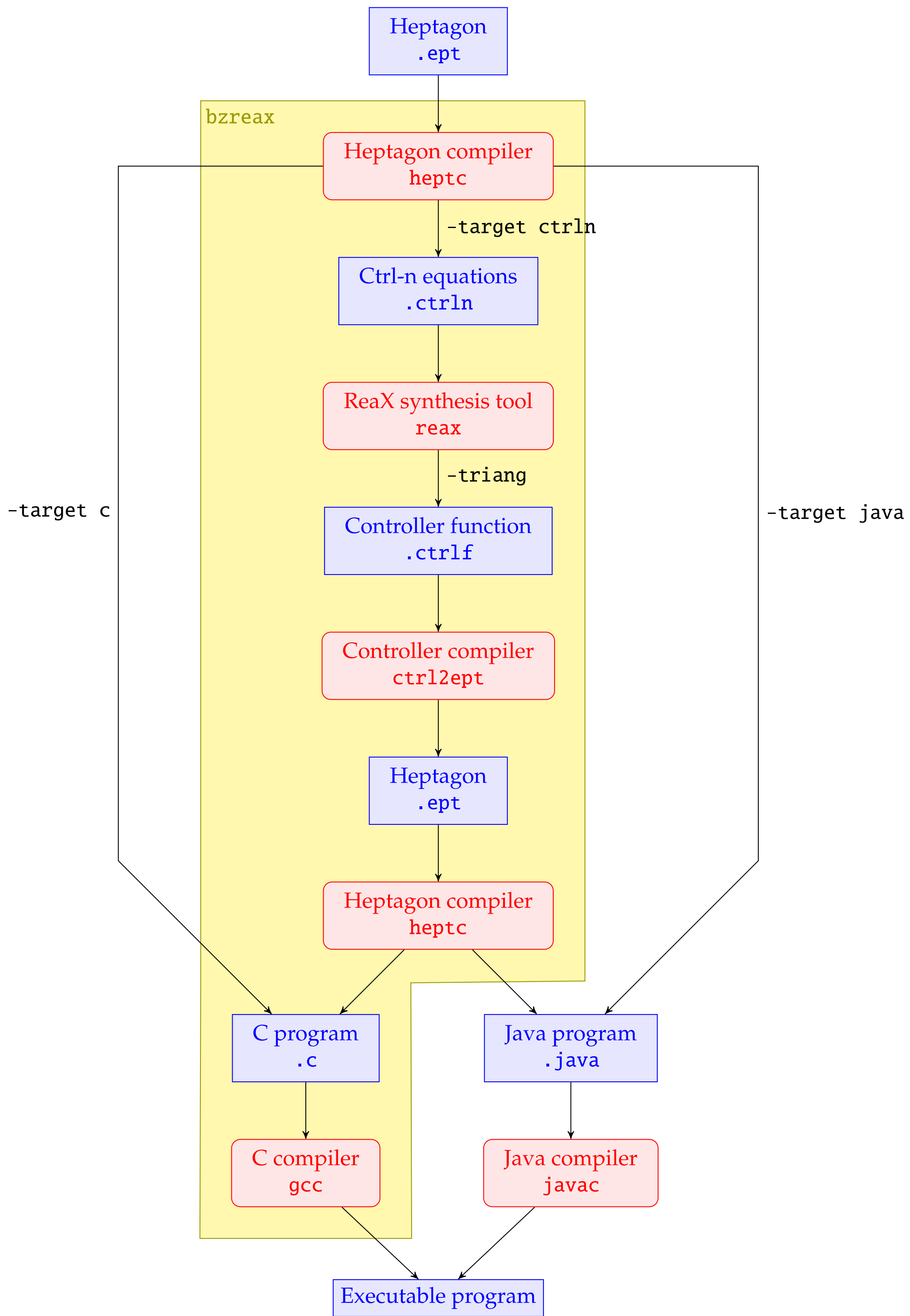


Figure 5: BZReaX script