Code generation strategies in the Polychrony environment

Loïc Besnard, Thierry Gautier, Jean-Pierre Talpin

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Code generation strategies in the Polychrony environment

Loïc Besnard — Thierry Gautier — Jean-Pierre Talpin

N° 6894

Avril 2009

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--6894--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
RENNES - BRETAGNE ATLANTIQUE

Code generation strategies in the Polychrony environment

Loïc Besnard , Thierry Gautier , Jean-Pierre Talpin

Thème COM — Systèmes communicants
Équipes-Projets Espresso

Rapport de recherche n° 6894 — Avril 2009 — 31 pages

Abstract: This report describes all code generation strategies available in the Polychrony toolset [1]. The data structure manipulated by the SIGNAL compiler is outlined and put to work presenting different ways to transform it in order to implement specific code generation schemes. Each compilation strategy is briefly presented.

Key-words: synchronous programming, compilation, program transformation, code generation

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Strategies de génération de code de l'environnement Polychrony

Résumé : Ce rapport décrit les stratégies de génération de code disponibles dans l'environnement Polychrony. Nous présentons les structures de données manipulées par le compilateur SIGNAL puis les différentes stratégies mises en oeuvre pour la manipuler de manire générer du code.

Mots-clés : programmation synchrone, compilation, transformation de programmes, génération de code

Contents

1 Introduction

This report describes all code generation strategies available in the Polychrony toolset [1]. It starts with a gentle example and its implementation. Then, the data structure of the SIGNAL compiler is outlined and put to work presenting different ways to transform it in order to implement specific code generation schemes. Each compilation strategy is briefly presented, the last one being that selected for the Spacify project.

1.1 An example

Our example is the resolution of the equation $aX^2 + bX + c = 0$ using the iterative Newton method (when $a > 0$). The parameters a, b, c are the inputs of the algorithm. The variable X is the output. Starting from $\Delta \geq 0$, the computation of $\sqrt{\Delta}$ is defined by the limit of the series $(X_n)_{n\geq 0}$:

$$\Delta = b^2 - 4ac \qquad X_0 = \frac{\Delta}{2} \qquad X_{n+1} = \frac{(X_n * X_n + \Delta)/X_n}{2} \ (n \geq 0)$$

An implementation of this resolution method features two modes depicted by the first degree and second degree of figure 1. The first degree mode is reactive and immediately outputs $-b/c$ when $a = 0$. The second degree mode implements Newton's iteration method and outputs the solutions of the equation upon stabilization.

The clock that triggers an iteration is left implicit in this specification. It will be made explicit in the process of generating code.

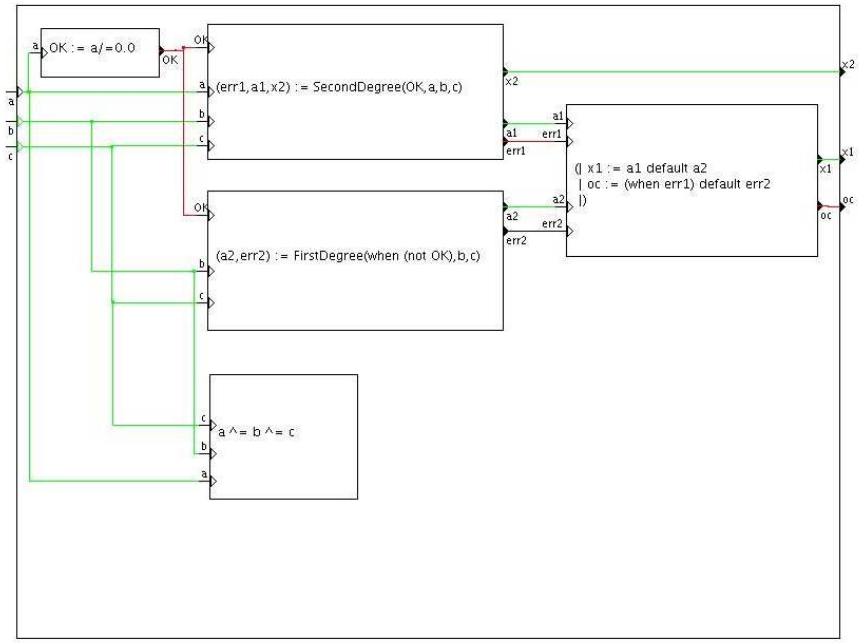


Figure 1: Block diagrammatic rendering of the solver

A close-up on the textual rendering of the solver, figure 2, shows that mode management is performed by three equations to select and activate them, merge their solutions or report errors.

- The interface `( ? real a, b, c; ! real x2, x1; boolean oc; )` consists of three real input signals a, b, c that are the parameters of the equation. It defines two real output signals

```
process equationSolving = ( ? real a, b, c; ! real x2, x1; boolean oc; )
  (|     (a2, err2) := FirstDegree (when (not OK), b, c)
   | (err1, a1, x2) := SecondDegree (OK, a, b, c)
   |              OK := a/=0.0
   |              x1 := a1 default a2
   |              oc := (when err1) default err2
   |               a ^= b ^= c
   |)
```

Figure 2: Equational rendering of the solver

that hold the solutions of the equation when present and, otherwise, the boolean output signal *oc*, that is true iff the equation has no solution, that is false iff the equation has infinitely many solutions.

- Equation `(a2, err2) := FirstDegree (when (not OK), b, c)` calls the first degree mode when the input parameter a equals 0. The input boolean signal Compute is present and true iff the mode is active. The output signal x samples the value of the solution $-c/b$ when the mode is active (the signal Compute is true) and when the factor b is not 0. Otherwise, the boolean signal err2 indicates that the equation has no solution (if a and b are 0 and not c) or inifinitely many (if c is 0 as well).

```
process FirstDegree = ( ? boolean Compute; real b, c; ! real x; boolean err2; )
  (| bb1  := bb when (bb/=0.0)
   | cc1  := cc when (bb/=0.0)
   | x    := -(cc1/bb1)
   | err2 := (cc/=0.0) when (bb=0.0)
   | bb   := b when Compute
   | cc   := c when Compute
   |)
```

- Equation `(err1, a1, x2) := SecondDegree (OK, a, b, c)` calls the second degree mode. As above, the mode is active when its boolean input signal OK is present and true. Its two real output signals $x21$ and $x2$ hold the solution of the equation for the input parameters a, b, c. If no solution exists then the boolean output signal is instead set to true.

```
process SecondDegree = ( ? boolean OK; real a, b, c; ! boolean err; real x21, x2; )
  (| (nul,err,d) := Discriminant{}(bb,cc)
   | aa  := a when OK
   | bb  := (b when OK)/aa
   | cc  := (c when OK)/aa
   | x21 := x1 default z
   | z   := (-(bb when nul))/2.0
   | (stable, x1, x2) := twoRoots{epsilon}(d, bb)
   | c ^= b ^= a
   | c ^= when stable
   |)
```

- Equation `OK := a /= 0.0` is present and false when the first degree mode is active, present and true when the second degree mode is active, and absent when the solver is inactive.
- Equation `x1:= a1 default a2` merges the first solution of the equation provided by either of the modes

- Equation `oc := (when err1) default err2` merges the error signals of either of the active modes
- Equation `a ^= b ^= c` synchronizes the input parameters.

The complete SIGNAL specification of the solver is given in Figure 3. The complete definition of the SIGNAL language can be found in [2].

```
process equationSolving =
     ( ? real a, b, c; ! real x2, x1; boolean oc; )
  (| OK := a/=0.0
   | (err1,a1,x2) := SecondDegree(OK,a,b,c)
   | (a2,err2) := FirstDegree(not OK,b,c)
   | (| x1 := a1 default a2
      | oc := (when err1) default err2
      |)
   | a ^= b ^= c
   |)
  where
    boolean OK, err1;
    real a1, a2;
    err2;
    process FirstDegree =
      ( ? boolean Compute;
          real b;
          real c;
        ! real x;
          boolean err2;
          )
      (| bb1 := bb when (bb/=0.0)
       | cc1 := cc when (bb/=0.0)
       | x := -(cc1/bb1)
       | err2 := (cc/=0.0) when (bb=0.0)
       | bb := b when Compute
       | cc := c when Compute
       |)
      where
      bb1, cc1;
      real bb, cc;
      end;
 process SecondDegree =
      ( ? boolean OK; real a, b, c;
        ! boolean err; real x21, x2; )
   (| (nul,err,d) := Discriminant{}(bb,cc)
    | aa := a when OK
    | bb := (b when OK)/aa
    | cc := (c when OK)/aa
    | c ^= b ^= a
    | z := (-(bb when nul))/2.0
    | (stable,x1,x2) := twoRoots{epsilon}(d,bb)
    | x21 := x1 default z
    | c ^= when stable
    |)
   where
     constant real epsilon = 0.00001;
     process Discriminant =
       ( ? real b, c; ! event nul; boolean err; real d; )
       (| dd := (b*b)-(4.0*c)
        | d := dd when (dd>0.0)
        | nul := when (dd=0.0)
        | err := when (dd<0.0)
        |)
       where
         real dd;
       end;
    process twoRoots =
         { real eps; }
         ( ? real discr, b; ! boolean stable; real x1, x2; )
      (| (stable,d) := rac{eps}(discr)
       | bb := (b cell (^d)) when (^d)
       | x2 := -((bb-d)/2.0)
       | x1 := -((bb+d)/2.0)
       |)
      where
        process rac =
          { real epsilon; }
          ( ? real x; ! boolean stable; real y; )
          (| (| mx := x cell (^yy)
              | next_yy := (((yy+(mx/yy))/2.0) when biterate)
                           default yy
              | yy := (x/2.0) default (next_yy$1 init 1.0)
              |)
           | (| biterate := (^x) default (not stable)
              | next_stable := abs(next_yy-yy)<epsilon
              | stable := next_stable$1 init true
              | yy ^= stable
              | y := yy when (next_stable and biterate)
              |)
           |)
          where
            real yy, next_yy, mx;
            boolean next_stable, biterate;
            process abs = ( ? x; ! s; )
              (| s := (x when (x>=0.0)) default (-x) |)
              ;
            end;
       real d;
       bb;
       end;
    event nul;
    real d, aa, bb, cc;
    z, x1, stable;
    end;
end;
```

Figure 3: A specification of the solver in SIGNAL

2 Data structures

Because SIGNAL programs are systems of equations, producing executable code amounts to "solving" these equations. The code generation strategy that achieves this goal uses the synchronization and scheduling relations that are induced by the program. This yields an intermediate representation consisting of a directed acyclic graph structure called the *Hierarchical Conditional Dependence Graph* (HCDG). This graph is composed of a clock hierarchy and of a conditioned scheduling graph. This section explains how the information carried by the HCDG is computed.

2.1 Clock relations

Table 1 gives the clock relations (or synchronization relations) associated with SIGNAL equations. The clock of a signal `x` is noted $\hat{x}$. It symbolically represents the periods in time during which the signal x is present.

The clock of a Boolean signal `b` is partitioned into its exclusive sub-clocks $[b]$ and $[\neg b]$ which denote the times when the signal `b` is present and carries the values *true* and *false*, respectively. The composition of equations induces the union of clock relations.

constructs	clock relations
`y := f(x1,...,xn)`	$\hat{y} = \hat{x}_1 = ... = \hat{x}_n$
`y := x $1 init c`	$\hat{y} = \hat{x}$
`y := x when b`	$\hat{y} = \hat{x} \cap [b]$, $[b] \cup [\neg b] = \hat{b}$ and $[b] \cap [\neg b] = \emptyset$
`z := x default y`	$\hat{z} = \hat{x} \cup \hat{y}$

Table 1: Clock relations for equations

In the example of the solver, the inference of clock relations yields the following profile of the solver process. In this profile, the internal clocks triggering the iterative algorithm have been lifted to the interface of the solver in order to modularly operate it.

```
process equationSolving_ABSTRACT =
     ( ? real a, b, c; ! real x2, x1;
         boolean oc, stable, C_y, C_err, C_x1;
     )
     spec (| stable ^= C_y ^= C_x1
           | when stable ^= a ^= b ^= c ^= C_err
           | when C_y ^= x2
           | when C_err ^= oc
           | when C_x1 ^= x1
           | ...
```

- Equations `when C_x1 ^= x1`, `when C_y ^= x2` and `when C_err ^= oc` define the boolean signals `C_x1, C_y, C_err` to carry the clocks of the outputs x1, x2, err. They are true iff the signal they are synchronized to is present. They are false otherwise.
- Equation `stable ^= C_y ^= C_x1` synchronizes the boolean signal stable, that tells when the iteration is either progressing (false) or finished (true), to the clocks `C_x1, C_y` of the solutions x1 and x2.
- Equation `when stable ^= a ^= b ^= c ^= C_err` tells that the error code (true or false) and the input signals a, b and c can be read and synchronized when the iteration has finished (signal stable is true)

2.2 Scheduling relations

Whereas clock relations allow us to determine which signals need to be computed at a given time, scheduling relations tell us in which order these signals have to be computed. Therefore, each vertex in the graph resulting of the scheduling analysis expresses a conditional scheduling relation: $c : x \to y$ means that the computation of y cannot be performed before x at the clock c (it is written $x \to y$ when conditioned by $\hat{x}$. For instance, and for any signal x, we have that the value of x cannot be computed before $\hat{x}$, the clock of x, hence the implicit scheduling relation $\hat{x} \to x$. Table 2 summarizes the scheduling relations associated with each primitive SIGNAL equation. Notice, in particular, that the delay does not have scheduling relations between its input and output. The composition of equations induces the union of scheduling relations.

constructs	scheduling relations
`y := f(x1,...,xn)`	$\hat{y} : \mathtt{x}_1 \to \mathtt{y}, \ldots, \quad \hat{y} : \mathtt{x}_n \to \mathtt{y}$
`y := x $1 init c`	
`y := x when b`	$\mathtt{x} \to \mathtt{y}, \quad \hat{y} : \mathtt{b} \to \hat{y}$
`z := x default y`	$\mathtt{x} \to \mathtt{z}, \quad \hat{y} - \hat{x} : \mathtt{y} \to \mathtt{z}$

Table 2: Scheduling relations for equations

In the example of the solver, the inference of scheduling relations according to the rules of Table 2 yields the following scheduling profile, when reduced to the interface of the process. It simply says that none of the outputs x1, x2 or oc can be computed before the input parameter a, b, c are available.

```
process equationSolving_ABSTRACT =
     ( ? real a, b, c; ! real x2, x1;
         boolean oc, stable, C_y, C_err, C_x1;
     )
     spec (| ...
           | a --> x2
           | b --> x2
           | c --> x2
           | a --> x1
           | b --> x1
           | c --> x1
           | a --> oc
           | b --> oc
           | c --> oc
           |)
```

It is worth mentioning that the complete scheduling graph of the solver should also contain relations `C_x1 --> x1`, `C_y --> x2`, `C_err --> oc`, between the outputs and their clocks.

2.3 Clock hierarchization

To generate an executable program that is able to compute the value of each signal in a process, one first needs to define a way to compute the clock of each signal. We say that a process is *endochronous* when there is a unique (deterministic) way to compute the clock of all its signal.

The clock relations of a process provide the necessary information to determine how clocks can be computed. From the clock relations induced by a process, we build a so-called hierarchy [3]. A hierarchy is a relation $\leq$ that groups synchronous signals x, y in an equivalence class C_x and orders one equivalence class D above another, with $C \leq D$, when the status of its signals (absent or present) can be determined from the clock and value of signals of its predecessor C.

The construction of a hierarchy $\leq$ is based on three simple rules

1. if $\hat{x} = \hat{y}$ then $C_x = C_y$: two synchronous signals x and y are placed in the same clock equivalence class.

2. if $\hat{x} = [y]$ or $\hat{x} = [\neg y]$ then $C_y \leq C_x$: if the clock of x is defined by a sample of that of y then it can only be computed when the value of y is known. Hence its clock equivalence class resides below that of y.

3. if $\hat{x} = \hat{y}\, f\, \hat{z}$ and there exists C such that $C \leq C_y$ and $C \leq C_Z$ then $C \leq C_x$: if the clocks of y and z can be computed from the same set C of clocks and if the clock of x is defined by a function of them $f \in \{+, *, -\}$, then the clock of x can also be computed once C is known.

A functionality of the Polychrony environment, called the "clock calculus", determines the hierarchy of a process. The clock calculus has actually two functions:

1. it verifies that the synchronization relations of the program have a solution (they are said well-clocked);

2. it structures the control of the program according to its clock hierarchy.

For the example of Section 1.1, the complete result of the clock calculus is given in Figure 4 as a transformed SIGNAL program where the clocks and their definitions are made explicit (the syntactic embedding reflects the hierarchy of clocks). The sub-processes in Figure 4 correspond to the (non-singleton) clock equivalence classes of the original process. A graphical rendering of the hierarchy, reduced to some of the most significant signals, would be as follows.

C_1 $C_1 =$ {`stable, yy, mx, s`}
- $\leq C_2$ $C_3 =$ {`CLK_x2`}
- $\leq C_3$ $C_2 =$ {`when stable, a, b, c, OK`}
  - $\leq C_4$ $C_4 =$ {`when OK, CLK_aa, cc, dd, XZX`}
    - $\leq$ {`CLK_d`}
    - $\leq$ {`CLK`}
    - $\leq$ {`nul`}
    - $\leq$ {`err1`}
  - $\leq C_5$ $C_5 =$ {`when (not OK), bb_177, cc_178`}
    - $\leq$ {`CLK_a2`}
    - $\leq$ {`CLK_err2`}
  - $\leq C_6$ $C_6 =$ {`CLK_oc, oc`}
- $\leq C_7$ $C_7 =$ {`CLK_a1`}
- $\leq C_8$ $C_8 =$ {`CLK_x1`}

Let us pick, for instance, clock `CLK_d`. It is defined `when dd>0.0`, and sits below "`when OK`", which is itself below "`when stable`", under the master clock. This means that, to compute the clock `CLK_d` of d, one has to wait for the process to be triggered by the `stable` signal, and then test the value of `stable` and `OK`.

Notice that the master clock of the process, called `stable`, has been synthesized, and, from the larger Figure 4, that the clock of all input and output signals are computed. Therefore the solver process is well-clocked and endochronous.

```
process equationSolving_TRA =
     ( ? real a, b, c;
       ! real x2, x1;
         boolean oc;
     )
   pragmas
      Main
   end pragmas
   (| (| CLK_stable := ^CLK_stable
       | ACT_CLK_stable{}
       |) |)
   where
   constant real epsilon = 0.00001;
   event CLK_stable;
   process ACT_CLK_stable = ( )
      (|     CLK_stable ^= stable ^= yy ^= next_yy
          ^= next_stable ^= biterate ^= pre_next_yy
          ^= mx ^= s ^= XZX_131
       | (| CLK_53 := when biterate
          | CLK_54 := when (not biterate)
          |)
       | (| CLK_70 := when (XZX_131>=0.0)
          | CLK_71 := when (not (XZX_131>=0.0))
          |)
       | (| CLK_x2 := when (next_stable
       | (|            and biterate)
          | CLK_x2 ^= x2 ^= x1_47
          | ACT_CLK_x2{}
          | CLK_78 := when (not (next_stable
          |            and biterate))
          |)
       | (| XZX_163 := when stable
          | XZX_163 ^= a ^= b ^= c
          | ACT_XZX_163{}
          | CLK_94 := when (not stable)
          |)
       | (| CLK_151 := CLK_x2 ^+ XZX_163 |)
       | (| CLK_XZX_148 := CLK_x2 ^+ CLK_aa
          | CLK_XZX_148 ^= XZX_148
          | XZX_148 :=   (bb when CLK_aa)
          |         cell CLK_XZX_148 init 0.0e0
          |)
       | (| CLK_a1 := CLK_x2 ^+ nul
          | CLK_a1 ^= a1
          | a1 :=      (x1_47 when CLK_x2)
          |    default (z when CLK_171)
          |)
       | (| CLK_159 := CLK_a1 ^+ err1 |)
       | (| CLK_x1 := CLK_a1 ^+ CLK_a2
          | CLK_x1 ^= x1
          | x1 :=      (a1 when CLK_a1)
          |    default (a2 when CLK_173)
          |)
       | (| CLK_163 := CLK_x1 ^+ CLK_oc |)
       | (| CLK_168 := CLK_stable ^- CLK_d |)
       | (| CLK_171 := nul ^- CLK_x2 |)
       | (| CLK_173 := CLK_a2 ^- CLK_a1 |)
       | (| stable := next_stable$1 init true
          | yy :=      ((d/2.0) when CLK_d)
          |    default (pre_next_yy when CLK_168)
          | next_yy := (((yy+(mx/yy))/2.0) when CLK_53)
          |    default (yy when CLK_54)
          | mx := (d when CLK_d)
          |    cell CLK_stable init 0.0e0
          | next_stable := s<epsilon
          | biterate := CLK_d
          |    default ((not stable) when CLK_168)
          | pre_next_yy := next_yy$1 init 1.0
          | s := (XZX_131 when CLK_70)
          |   default ((-XZX_131) when CLK_71)
          | XZX_131 := next_yy-yy
          |)
       |)
      where
      ...
      process ACT_CLK_x2 = ( )
         (| CLK_x2 ^= d_97 ^= bb_98
          | (| x2 := -((bb_98-d_97)/2.0)
             | x1_47 := -((bb_98+d_97)/2.0)
             | d_97 := yy when CLK_x2
             | bb_98 := XZX_148 when CLK_x2
             |)
          |)
         where
         real d_97, bb_98;
         end %ACT_CLK_x2%;
      process ACT_XZX_163 = ( )
         (| XZX_163 ^= OK
          | (| CLK_aa := when OK
             | CLK_aa ^= bb
             | ACT_CLK_aa{}
             | XZX_169 := when (not OK)
             | ACT_XZX_169{}
             |)
          | (| CLK_oc := err1 ^+ CLK_err2
             | CLK_oc ^= oc
             | oc := err1
             |    default (err2 when CLK_err2)
             |)
          | (| OK := a/=0.0 |)
          |)
         where
         event CLK_err2, XZX_169;
         boolean OK, err2;
         process ACT_CLK_aa = ( )
            (| CLK_aa ^= aa ^= cc ^= dd ^= XZX ^= XZX_77
             | (| CLK_d := when (dd>0.0)
                | CLK_d ^= d
                | (| d := dd when CLK_d |)
                | CLK := when (not (dd>0.0))
                |)
             | (| nul := when (dd=0.0)
                | nul ^= z
                | ACT_nul{}
                | CLK_25 := when (not (dd=0.0))
                |)
             | (| err1 := when (dd<0.0)
                | CLK_27 := when (not (dd<0.0))
                |)
             | (| CLK_144 := nul ^+ err1 |)
             | (| CLK_146 := CLK_d ^+ CLK_144 |)
             | (| CLK_148 := CLK_d ^+ err1 |)
             | (| aa := a when CLK_aa
                | bb := XZX/aa
                | cc := XZX_77/aa
                | dd := (bb*bb)-(4.0*cc)
                | XZX := b when CLK_aa
                | XZX_77 := c when CLK_aa
                |)
             |)
            where
            event CLK_148, CLK_146, CLK_144, CLK_27, CLK_25, CLK;
            real aa, cc, dd, XZX, XZX_77;
            process ACT_nul = ( )
               ...
               end %ACT_nul%;
            end %ACT_CLK_aa%;
         process ACT_XZX_169 = ( )
             ...
             |)
            where
            event CLK_142, CLK_118, CLK_110;
            real bb_177, cc_178;
            process ACT_CLK_a2 = ( )
                ...
               where
               real bb1, cc1;
               end %ACT_CLK_a2%;
            end %ACT_XZX_169%;
         end %ACT_XZX_163%;
      end %ACT_CLK_stable%;
   end %equationSolving_TRA%;
```

Figure 4: Rendering of the solver transformed by the clock calculus

2.4 Clock-driven graph scheduling

In the process of creating the hierarchical representation of a process, Figure 4, the clock calculus also associates a sub-process with each clock of the hierarchy. This sub-process represents the set of the computations that occur at this clock. It is in fact the syntactic rendering of the scheduling graph attached to that clock.

For the example of Section 1.1, for instance, the computations associated with the master clock is given in Figure 5 (left, and its corresponding graph, right).

```
| stable := next_stable$1 init true
| yy := ((d/2.0) when CLK_d) default (pre_next_yy when CLK_168)
| next_yy := (((yy+(mx/yy))/2.0) when CLK_53) default (yy when CLK_54)
| mx := (d when CLK_d) cell CLK_stable init 0.0e0
| next_stable := s<epsilon
| biterate := CLK_d default ((not stable) when CLK_168)
| pre_next_yy := next_yy$1 init 1.0
| s := (XZX_131 when CLK_70) default ((-XZX_131) when CLK_71)
| XZX_131 := next_yy-yy
```

```
       d --> yy
      yy --> next_yy
      yy --> XZX_131
 next_yy --> XZX_131
 XZX_131 --> s
       d --> mx
       s --> next_stable
   CLK_d --> biterate
  stable --> biterate
```

Figure 5: Equation solving example: the set of computations of the master clock.

2.5 Refinement heuristics

When a process is deemed endochonous because its clock hierarchy is a tree, and when its scheduling graph is *acyclic*, then the code generation functionalities of Polychrony can be applied. Generated code can be obtained for different target languages (C, C++, Java).

If the analyzed process is not endochronous, then heuristics can be applied to instrument each root of its hierarchy with a default parameterization. This parameterization can be used to simulate an otherwise non-deterministic process. This parameterization can also be used as a way to trigger the different threads of a, so-called, weakly-endochronous process (a process consisting of independent or confluent endochronous sub-processes). For instance, one would refine the hierarchy with two independent master classes C_1 and D_1, below left, with an added boolean clock B, below right, to select one or the other at all times.

$$\begin{array}{r} C_1 \leq C_2 \\ \leq C_3 \\ D_1 \leq D_2 \\ \leq D_3 \end{array} \Rightarrow \begin{array}{r} B \leq C_1 \leq C_2 \\ \leq C_3 \\ \leq D_1 \leq D_2 \\ \leq D_3 \end{array}$$

Heuristics may also be employed to refine the synthesized scheduling graph. For instance, and for a given graph G, if the targeted execution scheme is sequential (monolithic code generation, code generation inside a cluster), then one needs to reinforce or saturate the graph G. A graph G can be saturated with the graph H as $G \cup H$ iff, for any call context represented by a graph F, if $F \cup G$ is acyclic then so is $F \cup G \cup H$: (1). Reinforcement is not optimal, since there can possibly be several incomparable (not included in one another) maximal pre-orders saturating the same graph F according to the property (1).

3 Code generation principle

Code generation strongly relies on the clock hierarchy resulting from the clock calculus. Each clock is represented by a Boolean variable which is *true* when the clock is present. Code generation also takes into account the scheduling graph to order elementary operations in sequence.

The code generated by the compiler is an infinite loop of elementary iterations. Each iteration is a reaction to input signals in which the embedding of `if-then-else` statements corresponds to the hierarchy of clocks and in which sequencing corresponds to the structure of the scheduling graph.

Generated code is produced in different files. For example, for C code generation, we have a main program suffixed _main.c, a program body suffixed _body.c, and an input-output module suffixed _io.c. The *main* program calls the *initialization* function defined in the program body, then keeps calling the *iterate* function (see Figure 6). The file _io.c defines the inputs and outputs of the program with the operating system.

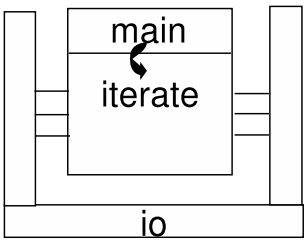


Figure 6: Single loop code

3.1 The main program

The *main* (see Figure 7) is responsible for initiating communication with the operating system, for initializing the state variables of the program, and for launching the infinite loop that iterates the execution step. In the case of simulation code, as in Figure 7, the infinite loop can be stopped if the iterate function returns error code 0 (meaning that input streams are empty) and the main program will close communication.

```
EXTERN int main()
{
    logical code;
    equationSolving_OpenIO(); /* input/output initialising */
    code = equationSolving_initialize(); /* initializing the application */
    while(code) code = equationSolving_iterate(); /* the steps */
    equationSolving_CloseIO(); /* input/output finalizing. */
}
```

Figure 7: Generated C code of the solver : the *main* program

3.2 The input-output interface

Communication of the program with the execution environment is implemented in the i/o file. In the simulation code generator, each input or output signal is interfaced with the operating system by a stream connected to a file containing input data and collecting output data. The i/o files declares global functions for opening and closing all files, and for reading and writing events along all input and output signals.

For instance, Figure 8, function `equationSolving_OpenIO` opens a stream `fra` for input signal a uploading data from the file `Ra.dat`, function `equationSolving_CloseIO` closes it, function `r_equationSolving_a` allows to read from it (as long as `EOF` is not readched, meaning the end of the simulation).

```
EXTERN void equationSolving_OpenIO()
{
  fra = fopen("Ra.dat","rt");
  if (!fra)
    {
      fprintf(stderr,"Can not open file %s\n","Ra.dat");
      exit(1);
    }
  /* ... idem for b, c */

  fwx1 = fopen("Wx1.dat","wt");
  if (!fwx1)
    {
      fprintf(stderr,"Can not open file %s\n","Wx1.dat");
      exit(1);
    }
  /* ... idem for x2, oc */
}
```

```
EXTERN void equationSolving_CloseIO()
{
  fclose(fra);
  ...
  fclose(fwx1);
  ...
}

EXTERN logical r_equationSolving_a(float *a)
{
  if (fscanf(fra,"%f",a)==EOF)return FALSE;
  return TRUE;
}
/* ... idem for b, c */

EXTERN void w_equationSolving_x1(float x1)
{
  fprintf(fwx1,"%f ",x1);
  fprintf(fwx1,"\n");
  fflush(fwx1);
}
/* ... idem for x2, oc */
```

Figure 8: Generated C code of the solver : the i/o part

The generated code in the i/o file is highly configurable. In can connect the program to any operating system or any middleware interface supporting lossless FIFO communication media. Naturally the infinite loop break can be omitted for embedded execution or possibly configured as a reset function, etc. The i/o file is the place where the interface of generated code with an external visualization and simulation tool can be implemented.

3.3 The iterate function

Once the program and its interface are initialized, the iterate function is responsible for iteratively performing the execution steps that read data from input streams, calculate and write results along output streams. There are many ways to implement this function starting from the clock hierarchy and scheduling graph produced by the front end of the compiler. These code generation schemes [6, 5, 4] are all implemented in the Polychrony toolset and are detailed in the subsequent sections of the report.

- Global compilation
  - Sequential code generation (section 4)
  - Clustered code generation with static scheduling (section 5)
  - Clustered code generation with dynamic scheduling (section 6)
  - Distributed code generation (section 7)
- Modular compilation
  - Monolithic code generation (section 8.1)
  - Legacy code encapsulation (section 8.3)
  - Clustered code generation (section 8.2)

4 Sequential code generation

This section describes the basic, sequential, inlining, code generation scheme that directly interpret the Signal program obtained after clock hierarchization (Figure 4) in order to produce sequential code.

```
/* input signals     */
static float a, b, c;
/* output signals    */
static float x2, x1;
static logical oc;
/* local signals     */
...

static void equationSolving_STEP_initialize()
{
    OK = FALSE;
    C_cc = FALSE;
    C_d = FALSE;
    C_ = FALSE;
    C_cc_422 = FALSE;
}
EXTERN logical equationSolving_STEP_finalize()
{
    stable = next_stable;
    C_411 = FALSE;
    C_error = FALSE;
    equationSolving_STEP_initialize();
    return TRUE;
}
EXTERN logical equationSolving_iterate()
{
 if (stable) {
     if (!r_equationSolving_a(&a)) return FALSE;
     if (!r_equationSolving_b(&b)) return FALSE;
     if (!r_equationSolving_c(&c)) return FALSE;
     OK = a != 0.0;
     C_cc = !OK;
     if (OK)
         {
           bb = b / a;
           dd = bb * bb - 4.0 * (c / a);
           C_d = dd > 0.0;
           C_ = dd == 0.0;
           C_411 = dd < 0.0;
         }
     C_414 = (OK ? C_411 : FALSE);
     if (C_cc) {
           C_cc_422 = b != 0.0;
           C_error = b == 0.0;
           if (C_error) error = c != 0.0;
         }
     C_error_436 = (C_cc ? C_error : FALSE);
     C_err = C_414 || C_error_436;
     if (C_err) {
           if (C_414) err_262 = TRUE; else err_262 = error;
           oc = err_262;
           w_equationSolving_oc(oc);
         }
     }
 OK_385 = (stable ? OK : FALSE);
 C_d_395 = (OK ? C_d : FALSE);
 if (C_d_395) mx = dd;
 C__406 = (OK ? C_ : FALSE);
 C_cc_428 = (C_cc ? C_cc_422 : FALSE);
 if (C_d_395) yy = dd / 2.0; else yy = next_yy;
 if (C_d_395) biterate = TRUE; else biterate = !stable;
 if (biterate) next_yy = (yy + mx/yy)/2.0; else next_yy = yy;
 XZX_168 = next_yy - yy;
 if (XZX_168 >= 0.0) s = XZX_168; else s = -XZX_168;
 next_stable = s < (0.00001);
 C_y = next_stable && biterate;
 C_x = C_y || C__406;
 C_x1 = C_x || C_cc_428;
 if (C_y) {
       x2 = -((bb - yy) / 2.0);
       w_equationSolving_x2(x2);
     }
 if (C_x)
    if (C_y) x21 = -((bb + yy)/2.0); else x21 = -bb/2.0;
 if (C_x1) {
       if (C_x) x1 = x21; else x1 = -(c / b);
       w_equationSolving_x1(x1);
     }
 equationSolving_STEP_finalize();
 return TRUE;
}
```

Figure 9: Generated C code of the solver : the body part

A few observations can be made on the *iterate* block. A first observation is that the internal clocks that were lifted in Figure 4 have disappeared. They have been unified with that of the the `stable` variable, that ticks every time the iterate function is called.

Inputs are read as soon as their clock is evaluated and is *true*. For example the reading of the signal `a` (statement `r_equationSolving_a(&a)`) is called when the signal `stable` is *true*. Outputs are sent as soon as they are evaluated. For example the writing of the signal `oc` (statement `w_equationSolving_oc(oc)`) is called when the signal `C_err` is *true*.

The state variables are updated at the end of the step (`equationSolving_STEP_finalize` block). Note that in some cases, `X$1` and `X` can be merged.

Finally, one can also notice the tree structure of conditional if-then-else statements which directly translates the clock hierarchy. For instance, the statements associated with `C_err` are executed only if `stable` is satisfied. In this code generation scheme, the scheduling and the computations are merged.

5 Clustered code generation with static scheduling

Instead of being flattened as a clock hierarchy, generated code can be partitioned into clusters mimicking the structure of the scheduling graph, in order to exploit some of its concurrency. This method is particularly relevant in code generation scenarios such as separate compilation and distribution. Figure 10 presents generated code for a partition of the solver into four clusters. The function `equationSolving_iterate` encodes a static scheduler for the clusters.

```
static void equationSolving_Cluster_1()
{
  C_d = FALSE;
  C_ = FALSE;
  C_411 = FALSE;
  C_414 = FALSE;
  C_err = FALSE;
  if (stable)
    {
      if (C_cc)
        {
          if (C_cc_422) x = -(c / b);
          if (C_error) error = c != 0.0;
        }
      if (OK)
        {
          dd = bb * bb - 4.0 * cc;
          C_d = dd > 0.0;
          C_ = dd == 0.0;
          C_411 = dd < 0.0;
          if (C_) x_1 = -bb / 2.0;
        }
      C_414 = (OK ? C_411 : FALSE);
      C_err = C_414 || C_error_436;
      if (C_err)
        {
          if (C_414) err_262 = TRUE; else err_262 = error;
          oc = err_262;
        }
    }
  C_d_395 = (OK ? C_d : FALSE);
  if (C_d_395) mx = dd;
  C__406 = (OK ? C_ : FALSE);
  C_475 = !C_d_395;
  if (C_d_395) yy = dd / 2.0; else yy = next_yy;
  if (C_d_395) biterate = TRUE; else biterate = !stable;
  C = !biterate;
  if (biterate) next_yy = (yy + mx/yy)/2.0; else next_yy = yy;
  XZX_168 = next_yy - yy;
  C_369 = XZX_168 >= 0.0;
  C_372 = !(XZX_168 >= 0.0);
  if (C_369) s = XZX_168; else s = -XZX_168;
  next_stable = s < (0.00001);
  C_y = next_stable && biterate;
  C__454 = C_y || OK_385;
  C_x = C_y || C__406;
  C_x1 = C_x || C_cc_428;
  C_479 = !C_y && C__406;
  C_483 = !C_x && C_cc_428;
  if (C_y)
      {
        x_191 = -((bb - yy) / 2.0);
        x2 = x_191;
        x_200 = -((bb + yy) / 2.0);
      }
  if (C_x) if (C_y) x21 = x_200; else x21 = x_1;
  if (C_x1) if (C_x) x1 = x21; else x1 = x;
}
```

```
static void equationSolving_Cluster_2()
{
  OK = FALSE;
  C_cc = FALSE;
  if (stable)
    {
      OK = a != 0.0;
      C_cc = !OK;
    }
  OK_385 = (stable ? OK : FALSE);
}
static void equationSolving_Cluster_3()
{
  C_cc_422 = FALSE;
  C_error = FALSE;
  C_error_436 = FALSE;
  if (stable)
    {
      if (OK) bb = b / a;
      if (C_cc)
        {
          C_cc_422 = b != 0.0;
          C_error = b == 0.0;
        }
      C_error_436 = (C_cc ? C_error : FALSE);
    }
  C_cc_428 = (C_cc ? C_cc_422 : FALSE);
}
static void equationSolving_Cluster_4()
{
  if (OK)  cc = c / a;
}
static void equationSolving_Cluster_delays()
{
  stable = next_stable;
  equationSolving_STEP_initialize();
}
EXTERN logical equationSolving_iterate()
{
  if (stable)
      {
        if (!r_equationSolving_a(&a)) return FALSE;
        if (!r_equationSolving_b(&b)) return FALSE;
        if (!r_equationSolving_c(&c)) return FALSE;
      }
  equationSolving_Cluster_2();
  equationSolving_Cluster_3();
  if (stable)  equationSolving_Cluster_4();
  equationSolving_Cluster_1();
  if (stable)
    if (C_err) w_equationSolving_oc(oc);
  if (C_y) w_equationSolving_x2(x2);
  if (C_x1) w_equationSolving_x1(x1);
  equationSolving_Cluster_delays();
  return TRUE;
}
```

Figure 10: Generated C code of the solver : statically scheduled clusters

The code comprises one main computation cluster `equationSolving_Cluster_1`, that performs an iteration of the resolution, and three auxiliary clusters. By contrast to the previous code generation method, which globally relies on the clock hierarchy and locally relies (for each equivalence class of the hierarchy) on the scheduling graph, clustering globally relies on the scheduling graph and locally relies (for each cluster of the graph) on the clock hierarchy.

The principle of clustering is to partition the scheduling graph into computations that rely on the exact same set of inputs. Conversely, one can also cluster the scheduling graph of a program according to its output signals. A cluster can be executed atomically as soon as its inputs are available. The clock of a cluster is given by the common ancestor of its nodes in the clock hierarchy. Let $G \subseteq X^2$ be a scheduling graph on the set of signals X, $I \subset X$ the input signals in this graph. Then, for any $x \in X$ and $J \subseteq I$, x belongs to the cluster J, written $x \in C_J$, iff $pred^*_G(x) \cap I = J$.

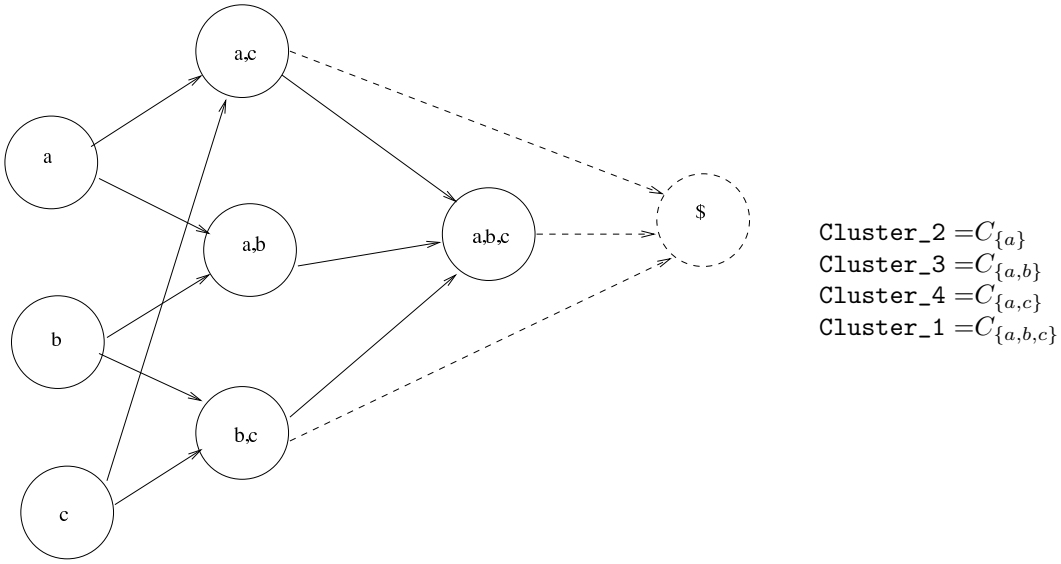


Figure 11: A cluster of G depends on the same set of inputs J

In the case of the solver, Figure 11, three inputs a, b, c can lead to, at most, $2^3 = 8$ clusters, plus one for delayed signals (as explained below). Clustering is, fortunately, never combinatoric. In the case of the solver, only four clusters are non-empty. Furthermore, the static scheduling of the main iteration function shows that these clusters are subject to inter-cluster data-dependencies. For instance, variable `OK` is defined in `Cluster_2` and then used in `Cluster_3`, meaning that these two clusters could be merged.

$$\texttt{Cluster\_2} \rightarrow \texttt{Cluster\_3} \rightarrow \texttt{Cluster\_4} \rightarrow \texttt{Cluster\_1}$$

A feature of our clustering technique maximizes the possibility to adapt it to the structure of the scheduling graph. It is to make all combinatoric clusters of the process precede one that is in charge of updating variables : $\forall \texttt{i}, \texttt{Cluster\_i} \rightarrow \texttt{Cluster\_delays}$.

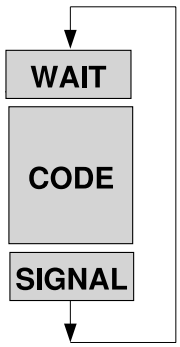
Note In general, there is a great flexibility in choosing a good partition of the scheduling graph. Any clustering is optimal with respect to some arbitrary criterion. Monolithic clustering (one cluster for the whole process) minimizes scheduling overhead and concurrency. Extreme clustering (one cluster per signal) maximizes scheduling overhead and concurrency. To avoid a combinatoric exploration of all partitions, Polychrony relies on heuristics to combine neighbor clusters together (those who always execute one after the other).

6 Clustered code generation with dynamic scheduling

Clustered code generation can be used for multi-threaded simulation by equipping it with *dynamic* scheduling artifacts. The code generation method implements clusters by tasks and generates synchronizations between tasks. Its principles are outlined in Figures 12 and 13.

- One task `Ti` is generated for each cluster `Cluster_i`, for each input-output function, plus one called `T0` for the iterate function `iterate`.
- The code of clusters `Cluster_i` is unchanged.
- Each task `Ti` has a semaphore `Si`.

- Each task `Ti` starts by waiting on its M_i predecessors with `wait(Si)`statements.
- Each task `Ti` ends by signaling all its successors $j = 1, ..N_i$ with `signal(Sij)`.



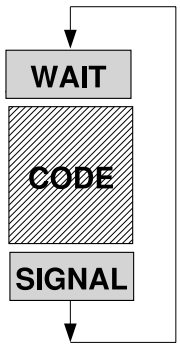

```
void * X_Ti() { /* task i */
  while (true) {
        pK_Isem_wait(Si);
        ... /* for all predecessors of i*/
        pK_Isem_wait(Si);

        X_Cluster_i();

        pK_Isem_signal(Si1);
        ... /* for all successors of i */
        pK_Isem_signal(SiNi);
      }
}
```

Figure 12: Code template of a cluster task

- The iterate function starts execution by signaling all source tasks $j = 1, ..N_0$ with `signal(S0j)`
- The iterate function ends by waiting on its sink tasks with `wait(S0)` statements.




```
void * X_iterate_Task() /* Task 0 */
{
  while (true) {
        /* signals the semaphores of
           all source tasks */
        pK_Isem_signal(S01); ?
        ...
        pK_Isem_signal(S0N0); ?
        /* wait on own semaphore */
           for all sink tasks */
        pK_Isem_wait(S0);
        ...
        pK_Isem_wait(S0);
     }
}
```

Figure 13: Code template of the iterate task

The semaphores and tasks are created in the `_initialize` function of the generated code. When simulation code is generated, a `_terminate` task is also added to kill all tasks of the application. The complete generated code for the clustered and dynamically scheduled solver is given in Figure 14.

```
EXTERN logical equationSolving_initialize() {
    bb = 0.0e0;
    stable = TRUE;
    mx = 0.0e0;
    next_yy = 1.0;
    equationSolving_STEP_initialize();
    pK_Isem_ini(0,0);
    pK_Isem_ini(1,0);
     ...
    pK_Isem_ini(11,0);
    pK_Isem_ini(12,0);
    pK_Task_create(0,_a_Task);
    pK_Task_create(1,_b_Task);
    pK_Task_create(2,_c_Task);
    pK_Task_create(3,_equationSolving_Cluster_2_Task);
    pK_Task_create(4,_equationSolving_Cluster_3_Task);
    pK_Task_create(5,_equationSolving_Cluster_4_Task);
    pK_Task_create(6,_equationSolving_Cluster_1_Task);
    pK_Task_create(10,_equationSolving_Cluster_delays_Task);
    pK_Task_create(7,_oc_Task);
    pK_Task_create(8,_x2_Task);
    pK_Task_create(9,_x1_Task);
    pK_Task_create(11,_equationSolving_iterate_Task);
    pK_Task_create(12,_equationSolving_terminate_Task);
    return TRUE;
}
pK_decl_Task(_equationSolving_iterate_Task) {
  while(TRUE)
    {
      pK_Isem_signal(0);
      pK_Isem_signal(1);
      pK_Isem_signal(2);
      pK_Isem_wait(11);
    }
}
static void equationSolving_Cluster_1()
{ ... }
pK_decl_Task(_equationSolving_Cluster_1_Task) {
    while(TRUE)
      {
        pK_Isem_wait(6);
        pK_Isem_wait(6);
        pK_Isem_wait(6);
        equationSolving_Cluster_1();
        pK_Isem_signal(7);
        pK_Isem_signal(9);
        pK_Isem_signal(8);
      }
}
static void equationSolving_Cluster_2()
{ ... }
pK_decl_Task(_equationSolving_Cluster_2_Task) {
    while(TRUE)
      {
        pK_Isem_wait(3);
        equationSolving_Cluster_2();
        pK_Isem_signal(5);
        pK_Isem_signal(4);
        pK_Isem_signal(6);
      }
}
static void equationSolving_Cluster_3()
{ ...}
pK_decl_Task(_equationSolving_Cluster_3_Task) {
    while(TRUE)
      {
        pK_Isem_wait(4);
        pK_Isem_wait(4);
        equationSolving_Cluster_3();
        pK_Isem_signal(6);
      }
}
static void equationSolving_Cluster_4()
{ ...}
pK_decl_Task(_equationSolving_Cluster_4_Task) {
    while(TRUE)
      {
        pK_Isem_wait(5);
        pK_Isem_wait(5);
        if (stable) equationSolving_Cluster_4();
        pK_Isem_signal(6);
      }
}
static void equationSolving_Cluster_delays()
{ ... }
pK_decl_Task(_equationSolving_Cluster_delays_Task) {
    while(TRUE)
      {
        pK_Isem_wait(10);
        pK_Isem_wait(10);
        pK_Isem_wait(10);
        equationSolving_Cluster_delays();
        pK_Isem_signal(11);
      }
}
pK_decl_Task(_a_Task) {
  while(TRUE)
    {
      pK_Isem_wait(0);
      if (stable)
         if (!r_equationSolving_a(&a))
            {
               pK_Isem_signal(12);
               return NULL;
            }
      pK_Isem_signal(3);
    }
}
pK_decl_Task(_x1_Task) {
    while(TRUE)
      {
        pK_Isem_wait(9);
        if (C_x1) w_equationSolving_x1(x1);
        pK_Isem_signal(10);
      }
}
pK_decl_Task(_equationSolving_terminate_Task) {
    pK_Isem_wait(12);
    pK_Task_Cancel(0,_a_Task);
    pK_Task_Cancel(1,_b_Task);
    pK_Task_Cancel(2,_c_Task);
    pK_Task_Cancel(3,_equationSolving_Cluster_2_Task);
    pK_Task_Cancel(4,_equationSolving_Cluster_3_Task);
    pK_Task_Cancel(5,_equationSolving_Cluster_4_Task);
    pK_Task_Cancel(6,_equationSolving_Cluster_1_Task);
    pK_Task_Cancel(7,_oc_Task);
    pK_Task_Cancel(8,_x2_Task);
    pK_Task_Cancel(9,_x1_Task);
    pK_Task_Cancel(10,_equationSolving_Cluster_delays_Task);
    pK_Task_Cancel(11,_equationSolving_iterate_Task);
}
```

Figure 14: Generated C code of the solver : dynamically scheduled clusters

7 Distributed code generation

Distributed code generation in Polychrony follows along the same principles as for dynamically scheduled clustered code generation. While clustered code generation is automatic, distribution requires additional information to be provided by the user. Namely:

- A block-diagramming description of the architecture software.
- A topological description of the target architecture.
- A morphism to map software diagrams onto the target architecture blocks.

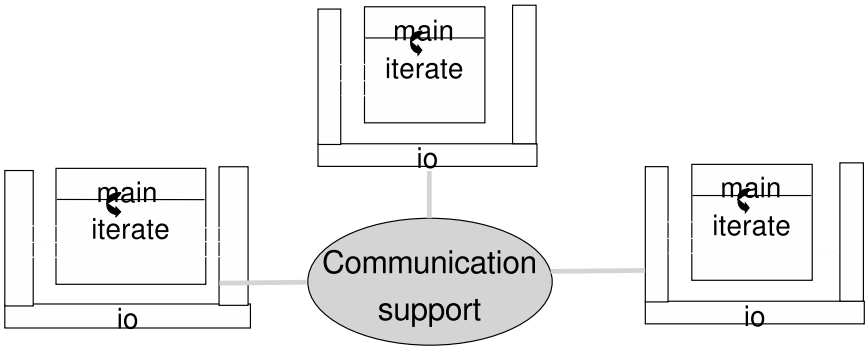


Figure 15: Overview of a distributed code

Automated distribution consists of a global compilation script that proceeds with the following steps:

1. Hierarchization of the main program structure (sub-programs may still be compiled separately)
2. "Booleanization" signals clocks are represented by booleans.
3. Partition of the main program according to the given software architecture.
4. Extractions of sub-graphs induced by each partition.
5. Synthesis of interfaces for each partition (projection, i/o dependences).
6. Addition of communication informations.
7. Compilation of each partition into clusters.
8. Communications are generated in the i/o module (the simulation code generator uses the MPI "Message-Passing Interface").
9. The *main* program is responsible for initiating/finalizing communications.

7.1 Topological annotations

The distribution methodology is applied on the example of Section 1.1 in Figure 16. The user partition is specified by associating the program with a few pragmas.

The pragma `RunOn` specifies the location on which a set of partitions is computed. For example, the expression `RunOn {e1} "1"` specifies that the partition labelled with `e1` is mapped on the location `1`. The statement `e1::PROC1{}` additionally declares `e1` as being the partition consisting of the subprocess `PROC1`.

The pragma `Topology` associates the input or output signals of the program with a location. For example, the pragma `Topology {a,b} "1"` tells that the input signals `a` and `b` must be read from location `1`.

The pragma `Target` specifies the API used to generate code that implements communication. for instance, `Target "MPI"` says that the MPI library is used.

```
process equationSolving =
     ( ? real a,b,c;
       ! boolean error;
         real x1, x2;
         )
  pragmas
    Topology {b,a} "1"
    Topology {c} "2"
    Topology {x1, x2} "2"
    Topology {error} "1"
    Target "MPI"
    RunOn {e1} "1"
    RunOn {e2} "2"
  end pragmas
  (|  e1:: PROC1{}
   |  e2:: PROC2{}
   |)
   where
     label e1,e2;
     boolean OK;
     real x;
     err2;
     process PROC1 =
        ( ? real a, b,c,x;
            err2;
            ! boolean OK;
              real x2;
              x1;
              boolean error;
         )
        (| (| err1 := err
            | a1 := x21
            | a2 := x
            | (| x1 := a1 default a2
               | error := err1 default err2
               |)
            |) where real a2, a1;
                 event err1; end
         | SecondDegree{0.0001}(when OK,a,b,c)
         | OK := a/=0.0
         |)
        where
          real x21;
          event err;
        end
      ;
      process PROC2 =
        ( ? boolean OK;
            real b, c;
            ! real x;
              err2;
              )
       (| b ^= c
        | FirstDegree(when (not OK),b,c)
        |)
     ;
    process SecondDegree =
         { real epsilon; }
         ( ? event OK;
             real a, b, c;
           ! event err;
             real x21, x2;
         )
      ...
      ;
    process FirstDegree =
      ( ? boolean Compute;
          real b, c;
        ! real x;
          boolean err2;
          )
      ...
      where
      end ;
    end ;
```

Figure 16: Functional partition of the solver

7.2 Communication annotations

Figure 17 displays the program transformation resulting of the functional partition requested by the user. Some signals have been added to the interface of the partitions : signals and clocks produced on one end of the system and used on the other must be communicated.

```
process equationSolving_EXTRACT_1_TRA =
     ( ? boolean err2;
         real x, a, b, c;
         boolean C_cc_1692, C_error;
       ! boolean error;
         real x1, x2;
         boolean stable, C_cc;
     )
 pragmas
    RunOn "1"
    DefinedClockHierarchy
    Environment {c} "1"
    Environment {b} "3"
    Environment {a} "5"
    Environment {error} "6"
    Environment {x1} "7"
    Environment {x2} "8"
    Sending {stable} "9" "equationSolving_EXTRACT_2"
    Sending {C_cc} "10" "equationSolving_EXTRACT_2"
    Receiving {x} "11" "equationSolving_EXTRACT_2"
    Receiving {err2} "12" "equationSolving_EXTRACT_2"
    Receiving {C_cc_1692} "13" "equationSolving_EXTRACT_2"
    Receiving {C_error} "14" "equationSolving_EXTRACT_2"
```

```
process equationSolving_EXTRACT_2_TRA =
 end pragmas
      ...
end %equationSolving_EXTRACT_1_TRA%;
     ( ? real b, c;
         boolean stable, C_cc;
       ! real x;
         boolean err2, C_cc_1692, C_error;
     )
 pragmas
    RunOn "2"
    DefinedClockHierarchy
    Environment {c} "2"
    Environment {b} "4"
    Receiving {stable} "9" "equationSolving_EXTRACT_1"
    Receiving {C_cc} "10" "equationSolving_EXTRACT_1"
    Sending {x} "11" "equationSolving_EXTRACT_1"
    Sending {err2} "12" "equationSolving_EXTRACT_1"
    Sending {C_cc_1692} "13" "equationSolving_EXTRACT_1"
    Sending {C_error} "14" "equationSolving_EXTRACT_1"
 end pragmas
  ....
 end %equationSolving_EXTRACT_2_TRA%;
```

Figure 17: The extracted sub-graphs after the partition

Communications are specified using additional pragmas. Pragma `Environment` associates an input or output signal to the location of a communication channel. For instance, `Environment c "1"` means that signal c is communicated along channel 1.

Pragma `Receiving` associates an input signal with a channel location and its sending process. For instance, the pragma `Receiving {x} "11" "equationSolving_EXTRACT_2"` tells that signal x is sent from process `equationSolving_EXTRACT_1_TRA` along channel `11`.

Similarly, pragma `Sending` associates an output signal with a channel location and its receiving processes. For example, `Sending {C_cc} "10" "equationSolving_EXTRACT_2"` tells that the output signal `C_cc` of process `equationSolving_EXTRACT_1_TRA` is sent to process `equationSolving_EXTRACT_2` along channel `10`.

7.3 Code generation

Multi-threaded, dynamically scheduled, code generation, as described in Section 6, can be applied on the program resulting of the transformations performed for automated distribution.

The information carried by the pragmas `Environment`, `Receiving` and `Sending` is used to generate communications. For instance, Figure 18 gives the generated code that implements communications for the signal `C_error` sent by the process `equationSolving_EXTRACT_2` to the process `equationSolving_EXTRACT_1` using the MPI library.

From the file : equationSolving_EXTRACT_1_io.c

```
EXTERN logical r_equationSolving_EXTRACT_1_C_error(logical *C_error)

    MPI_Recv(C_error,                   /* name */
             1,MPI_INT,                 /* type */
             equationSolving_EXTRACT_2, /* received from */
             14,                        /* the logical tag of the pragma receiving */
             MPI_COMM_WORLD,            /* MPI specific parameter */
             MPI_STATUS_IGNORE);        /* MPI specific parameter */
    return TRUE;
```

From the file : equationSolving_EXTRACT_2_io.c

```
EXTERN void w_equationSolving_EXTRACT_2_C_error(logical C_error)

    MPI_Send(&C_error,                      /* name */
             1,MPI_INT,                     /* type */
             equationSolving_EXTRACT_1,     /* sent to */
             14,                            /* the logical tag of the pragma sending */
             MPI_COMM_WORLD);               /* MPI specific parameter */
```

Figure 18: Example of communications

8 Modular code generation

Polychrony provides all needed services to implement a separate compilation methodology [4]. Separate compilation consists of compiling a process, of exporting its model, and use this model in another process. For the purpose of exporting and importing the model of a process whose code has been compiled separately, the SIGNAL compiler provides an annotation mechanism to associate a profile to a compiled program.

This profile consists of an abstraction of the original process' model consisting of the synchronization and scheduling relations of the input and output signals of the process. These properties may be provided by the designer (in the case of legacy C code, for instance) or calculated by the compiler (in the case of a compiled Signal program). The annotations also gives the possibility to specify the language in which the program is compiled (C, C++, Java) since the function call convention and variable binding may vary slightly from one to another.

Starting from a Signal program, separate compilation supports both of the code generation strategies we previously outlined : with or without clusters. Considering the specification of the

solver, we detail all possible compilation scenarios in which the sub-processes `FirstDegree` and `SecondDegree` are compiled separately.

8.1 Sequential code generation for separate compilation

The first compilation method consists of associating each of the sub-processes of the solver with a monolithic iterate function. Figure 19 gives the SIGNAL abstractions that are inferred (or could be user-provided) in order to use the `FirstDegree` and `SecondDegree` processes as external functions.

```
process FirstDegree_ABSTRACT =
     ( ? boolean Compute;
         real b, c;
       ! real x;
         boolean err2, C_bb, C_error;
     )
     spec (| (| Compute --> x
              | b --> x
              | c --> x
              | Compute --> err2
              | b --> err2
              | c --> err2
              |)
           | (| Compute ^= b ^= c
              | when Compute ^= C_bb ^= C_error
              | when C_bb ^= x
              | when C_error ^= err2
              |)
           |)
   pragmas
      BlackBox "FirstDegree"
   end pragmas
   external "C"
%FirstDegree_ABSTRACT%;
process SecondDegree_ABSTRACT =
     ( ? boolean OK;
         real a, b, c;
       ! boolean err;
         real x21, x2;
         boolean stable, C_y, C_err, C_x;
     )
     spec (| (| OK --> err
              | a --> err
              | b --> err
              | c --> err
              | OK --> x21
              | a --> x21
              | b --> x21
              | c --> x21
              | OK --> x2
              | a --> x2
              | b --> x2
              | c --> x2
              |)
           | (| stable ^= C_y ^= C_x
              | when OK ^= C_err
              | when stable ^= OK ^= a ^= b ^= c
              | when C_y ^= x2
              | when C_err ^= err
              | when C_x ^= x21
              |)
           |)
   pragmas
      BlackBox "SecondDegree"
   end pragmas
   external "C"
%SecondDegree_ABSTRACT%;
```

Figure 19: Black boxes abstraction of the FirstDegree and SecondDegree processes

One can observe that the interfaces of the processes `FirstDegree` and `SecondDegree` have been modified. The reason is that it is necessary to export some signals to rebuild the clock hierarchy of the original programs in the calling program. For a black box abstraction, the profile of a process, listed affter the keyword `spec`, is composed of its input/output synchronization and scheduling relations.

The C code corresponding to process `SecondDegree` is given in Figure 20. The difference with that of Section 4 is that its parameters are carried by a structure `data` in order to, of course, have multiple calls of the process in different call contexts.

```
typedef struct
{
 /*  ==> input signals     */
 logical OK;
 float a, b, c;
 /*  ==> output signals    */
 logical err;
 float x21, x2;
 logical stable, C_y, C_err, C_x;
 /*  ==> local signals     */
 float bb, dd, yy, next_yy;
 logical biterate;
 float mx;
 logical next_stable;
 float s, XZX_135, x_176;
 logical OK_263,C_d,C_d_277,C__282,C__288;
} SecondDegree;
EXTERN logical SecondDegree_initialize
       (SecondDegree *data) {
  data->stable = TRUE;
  data->bb = 0.0e0;
  data->mx = 0.0e0;
  data->next_yy = 1.0;
  data->err = TRUE;
  SecondDegree_STEP_initialize(data);
  return TRUE;
}
static void SecondDegree_STEP_initialize
       (SecondDegree *data) {
  data->C_d = FALSE;
  data->C__282 = FALSE;
}
EXTERN void SecondDegree_iterate(SecondDegree *data,
       logical _OK_,float _a_,float _b_, float _c_,
       logical *_err_,float *_x21_,float *_x2_,
       logical *_stable_,logical _C_y_,
       logical *_C_err_,logical *_C_x_) {
 *_stable_ = data->stable;
 data->OK = _OK_;
 data->a = _a_;
 data->b = _b_;
 data->c = _c_;
 if (data->stable)
   if (data->OK) {
       data->bb = data->b / data->a;
       data->dd = data->bb * data->bb - 4.0 * (data->c / data->a);
       data->C_d = data->dd > 0.0;
       data->C__282 = data->dd == 0.0;
       data->C_err = data->dd < 0.0;
       *_C_err_ = data->C_err;
       if (data->C_err) *_err_ = TRUE;
     }
 data->OK_263 = (data->stable ? data->OK : FALSE);
 data->C_d_277 = (data->OK ? data->C_d : FALSE);
 if (data->C_d_277) data->mx = data->dd;
 data->C__288 = (data->OK ? data->C__282 : FALSE);
 if (data->C_d_277) data->yy = data->dd/2.0;
 else data->yy = data->next_yy;
 if (data->C_d_277) data->biterate = TRUE;
 else data->biterate = !data->stable;
 if (data->biterate)
   data->next_yy = (data->yy + data->mx / data->yy)/2.0;
 else data->next_yy = data->yy;
 data->XZX_135 = data->next_yy - data->yy;
 if (data->XZX_135 >= 0.0) data->s = data->XZX_135;
 else data->s = -data->XZX_135;
 data->next_stable = data->s < (0.00001);
 data->C_y = data->next_stable && data->biterate;
 data->C_x = data->C_y || data->C__288;
 *_C_y_ = data->C_y;
 *_C_x_ = data->C_x;
 if (data->C_y) {
     data->x2 = -((data->bb - data->yy) / 2.0);
     *_x2_ = data->x2;
   }
 if (data->C_x) {
     if (data->C_y) data->x_176 = -((data->bb + data->yy) / 2.0);
     else data->x_176 = -data->bb / 2.0;
     data->x21 = data->x_176;
     *_x21_ = data->x21;
   }
 SecondDegree_STEP_finalize(data);
}
EXTERN logical SecondDegree_STEP_finalize(SecondDegree *data) {
 data->stable = data->next_stable;
 SecondDegree_STEP_initialize(data);
 return TRUE;
}
```

Figure 20: C code for the SecondDegree model: the body part for separated compilation

Figure 21 defines the SIGNAL program in which the separately compiled processes `FirstDegree` and `SecondDegree` are called. The modular code generation scheme proposed so far appears best suited to integrate small and mostly sequential specification.

```
process equationSolving = ( ? real a, b, c; ! real x2, x1; boolean oc; )
 (| OK := a/=0.0
  | (err1,a1,x2,stable,C_y,C_err,C_x) := SecondDegree_ABSTRACT(OK,a,b,c)
  | (a2,err2,C_bb,C_error) := FirstDegree_ABSTRACT(not OK,b,c)
  | x1 := a1 default a2
  | oc := (when err1) default err2
  | a ^= b ^= c
  |) where ...
end;
```

Figure 21: Importing separately compiled processes in the specification of the solver

In the case of the solver, however, the second degree function is one sophisticated process. In particular, it contains an internal loop controlled by the `stable` signal. However, during the separate compilation of `SecondDegree_ABSTRACT`, we applied the compiler heuristics of section 2.5 in order to produce a sequential C function from this process. In doing so, its interface was slightly modified with additional clock and scheduling relations (FIgure 19). It turns out that these additional

scheduling relations, put in the context of process `equationSolving`, form a causality cycle that is reported by the compiler, Figure 22. Code generation cannot proceed further.

```
process equationSolving_CYC = ( )
   (| (err1,a1,x2,stable,C_y,C_err,C_x) := SecondDegree_ABSTRACT(OK,a,b,c)
    |  stable --> when stable | when stable --> a | a --> x2 | x2 --> stable
    |) %equationSolving_CYC% ;
```

Figure 22: Cycle induced by the imported processes as reported by the compiler

8.2 Clustered code generation for separate compilation

To prevent for the scenario of Figure 22 from happening, it is better suited to apply clustering code generation techniques in order to avoid spurious causality cycles from being introduced. The profile of a clustered and separately compiled process, Figure 23, is called a "grey box".

```
process SecondDegree_ABSTRACT =
     ( ? boolean OK;
         real a, b, c;
       ! boolean err;
         real x21, x2;
         boolean stable, C_y, C_err, C_x;
     )
   pragmas GreyBox "SecondDegree"
   end pragmas
   (| (| Tick := true
       | when Tick ^= stable ^= C_y ^= C_x
       | when OK ^= C_err
       | when stable ^= OK ^= a ^= b ^= c
       | when C_y ^= x2
       | when C_err ^= err
       | when C_x ^= x21
       |)
    | (| nlabel :: (err,x21,x2,C_y,C_err,C_x) :=
                     SecondDegree_Cluster_1(OK)
       | nlabel ^= when Tick
       |)
    | (| nlabel_328 :: SecondDegree_Cluster_2(OK)
       | nlabel_328 ^= when OK
       |)
    | (| nlabel_329 :: SecondDegree_Cluster_3(OK)
       | nlabel_329 ^= when OK
       |)
    | (| nlabel_330 :: SecondDegree_Cluster_4(OK,a)
       | nlabel_330 ^= when stable
       |)
    | (| nlabel_331 :: SecondDegree_Cluster_5(OK,b)
       | nlabel_331 ^= when stable
       |)
    | (| nlabel_332 :: SecondDegree_Cluster_6(OK,c)
       | nlabel_332 ^= when stable
       |)
    | (| nlabel_333 :: SecondDegree_Cluster_7(OK)
       | nlabel_333 ^= when Tick
       |)
    | (| nlabel_368 :: stable := SecondDegree_Cluster_delays()
       | nlabel_368 ^= when Tick
       |)
    | (| c --> nlabel
       | b --> nlabel
       | a --> nlabel
       | OK --> nlabel
       | nlabel_328 --> nlabel
       | OK --> nlabel_328
       | nlabel_329 --> nlabel
       | OK --> nlabel_329
       | nlabel_330 --> nlabel_329
       | nlabel_330 --> nlabel_328
       | a --> nlabel_330
       | OK --> nlabel_330
       | nlabel_331 --> nlabel_328
       | b --> nlabel_331
       | OK --> nlabel_331
       | nlabel_332 --> nlabel_329
       | c --> nlabel_332
       | OK --> nlabel_332
       | nlabel_333 --> nlabel
       | OK --> nlabel_333
       | nlabel_333 --> nlabel_368
       | nlabel_332 --> nlabel_368
       | nlabel_331 --> nlabel_368
       | nlabel_330 --> nlabel_368
       | nlabel_329 --> nlabel_368
       | nlabel_328 --> nlabel_368
       | nlabel --> nlabel_368
       |)
    |)
   where
   boolean Tick;
   label nlabel, nlabel_328, nlabel_329, nlabel_330,
   nlabel_331, nlabel_332, nlabel_333, nlabel_368;
   action SecondDegree_Cluster_1 =
        ( ? boolean OK;
          ! boolean err;
            real x21, x2;
            boolean C_y, C_err, C_x;
        )
      pragmas
         DefinedClockHierarchy
         Cluster
         BlackBox ""
      end pragmas
      external "C"
   %SecondDegree_Cluster_1%;
   action SecondDegree_Cluster_2 =
        ( ? boolean OK; )
      pragmas
         DefinedClockHierarchy
         Cluster
         BlackBox ""
      end pragmas
      external "C"
   %SecondDegree_Cluster_2%;
   action SecondDegree_Cluster_3 =
        ( ? boolean OK; )
      pragmas
         DefinedClockHierarchy
         Cluster
         BlackBox ""
      end pragmas
      external "C"
   %SecondDegree_Cluster_3%;
   action SecondDegree_Cluster_4 =
        ( ? boolean OK; real a; )
      pragmas
         DefinedClockHierarchy
         Cluster
         BlackBox ""
      end pragmas
      external "C"
   %SecondDegree_Cluster_4%;
   action SecondDegree_Cluster_5 =
        ( ? boolean OK; real b; )
      pragmas
         DefinedClockHierarchy
         Cluster
         BlackBox ""
      end pragmas
      external "C"
   %SecondDegree_Cluster_5%;
   action SecondDegree_Cluster_6 =
        ( ? boolean OK; real c; )
      pragmas
         DefinedClockHierarchy
         Cluster
         BlackBox ""
      end pragmas
      external "C"
   %SecondDegree_Cluster_6%;
   action SecondDegree_Cluster_7 =
        ( ? boolean OK; )
      pragmas
         DefinedClockHierarchy
         Cluster
         BlackBox ""
      end pragmas
      external "C"
   %SecondDegree_Cluster_7%;
   action SecondDegree_Cluster_delays =
        ( ! boolean stable init true; )
      pragmas
         DelayCluster
         Cluster
         BlackBox ""
      end pragmas
      external "C"
   %SecondDegree_Cluster_delays%;
   end
%SecondDegree_ABSTRACT%;
```

Figure 23: The "grey box" abstraction of the SecondDegree model

A grey box provides the same information as a "black box", but it partitions it into the clusters that are generated from the original process, and details the clocks and the scheduling relations between the clusters. In turn, its profile contains sufficient information to schedule the clusters, and hence, call them in the appropriate order dictated by the calling context.

```
EXTERN logical SecondDegree_initialize(SecondDegree *data) {
  data->stable = TRUE;
  data->bb = 0.0e0;
  data->mx = 0.0e0;
  data->next_yy = 1.0;
  data->err = TRUE;
  SecondDegree_STEP_initialize(data);
  return TRUE;
}
EXTERN void SecondDegree_Cluster_1 (
  SecondDegree *data,logical _OK_,logical *_err_,
  float *_x21_,float *_x2_,logical *_C_y_,
  logical *_C_err_,logical *_C_x_)
{
  data->OK = _OK_;
  data->C_d = FALSE;
  data->C__282 = FALSE;
  if (data->stable) {
      if (data->OK) {
          data->dd = data->bb * data->bb - 4.0 * data->cc;
          data->C_d = data->dd > 0.0;
          data->C__282 = data->dd == 0.0;
          data->C_err = data->dd < 0.0;
          if (data->C__282)
              data->x = -data->bb / 2.0;
        }
    }
  data->C_d_277 = (data->OK ? data->C_d : FALSE);
  if (data->C_d_277) data->mx = data->dd;
  data->C__288 = (data->OK ? data->C__282 : FALSE);
  data->C_311 = !data->C_d_277;
  if (data->C_d_277) data->yy = data->dd / 2.0;
  else data->yy = data->next_yy;
  if (data->C_d_277) data->biterate = TRUE;
  else data->biterate = !data->stable;
  data->C = !data->biterate;
  if (data->biterate)
    data->next_yy = (data->yy + data->mx / data->yy) / 2.0;
  else data->next_yy = data->yy;
  data->XZX_135 = data->next_yy - data->yy;
  data->C_244 = data->XZX_135 >= 0.0;
  data->C_247 = !(data->XZX_135 >= 0.0);
  if (data->C_244) data->s = data->XZX_135;
  else data->s = -data->XZX_135;
  data->next_stable = data->s < (0.00001);
  data->C_y = data->next_stable && data->biterate;
  data->C_260 = data->C_y && data->stable;
  data->C_ = data->C_y || data->OK_263;
  data->C_x = data->C_y || data->C__288;
  data->C_315 = !data->C_y && data->C__288;
  data->C_321 = data->stable && data->C_x;
  if (data->C_y) {
      data->x_158 = -((data->bb - data->yy) / 2.0);
      data->x2 = data->x_158;
      data->x1 = -((data->bb + data->yy) / 2.0);
    }
  if (data->C_x) {
     if (data->C_y) data->x_176 = data->x1;
     else data->x_176 = data->x;
     data->x21 = data->x_176;
    }
  if (data->stable)
    if (data->OK) if (data->C_err) *_err_ = TRUE;
  if (data->C_x) *_x21_ = data->x21;
  if (data->C_y) *_x2_ = data->x2;
  *_C_y_ = data->C_y;
  if (data->stable)
    if (data->OK) *_C_err_ = data->C_err;
  *_C_x_ = data->C_x;
}
EXTERN void SecondDegree_Cluster_2
  (SecondDegree *data,logical _OK_) {
  data->OK = _OK_;
  data->bb = data->b / data->a;
}
EXTERN void SecondDegree_Cluster_3
  (SecondDegree *data,logical _OK_) {
  data->OK = _OK_;
  data->cc = data->c / data->a;
}

EXTERN void SecondDegree_Cluster_4
  (SecondDegree *data,logical _OK_,float _a_) {
  data->OK = _OK_;
  data->a = _a_;
}
EXTERN void SecondDegree_Cluster_5
  (SecondDegree *data,logical _OK_,float _b_) {
  data->OK = _OK_;
  data->b = _b_;
}
EXTERN void SecondDegree_Cluster_6
  (SecondDegree *data,logical _OK_,float _c_) {
  data->OK = _OK_;
  data->c = _c_;
}
EXTERN void SecondDegree_Cluster_7
  (SecondDegree *data,logical _OK_) {
  data->OK = _OK_;
  data->OK_263 = (data->stable ? data->OK : FALSE);
}
EXTERN void SecondDegree_Cluster_delays
  (SecondDegree *data,logical *_stable_) {
  data->stable = data->next_stable;
  *_stable_ = data->stable;
  SecondDegree_STEP_initialize(data);
}
static void SecondDegree_STEP_initialize
  (SecondDegree *data) {}
```

Figure 24: Generated modular C code for the second degree process

The clustered C code of process `SecondDegree` is given in Figure 24. It differs from that of Section 5 in the parameter `data` that carries its calling context (its input, output and local signals). Another noticeable change is the disappearance of the iterate function. Instead, the calling program schedules the generated clusters in the order best appropriate to its local context, as shown in the next Figure 25.

In the generated code of the solver, Figure 25, the clusters of the separately compiled processes `FirstDegree` and `SecondDegree`, are called in the very order dictated by scheduling constraints of the solver process, avoiding the introduction of any spurious cycle.

```
EXTERN logical equationSolving_initialize() {
 stable = TRUE;
 SecondDegree_initialize(&SecondDegree1);
 FirstDegree_initialize(&FirstDegree2);
 equationSolving_STEP_initialize();
 return TRUE;
}
static void equationSolving_STEP_initialize() {
 C_err = FALSE;
 C_bb = FALSE;
 OK = FALSE;
 err1 = FALSE;
 XZX_116 = FALSE;
}
EXTERN logical equationSolving_iterate() {
 if (stable) {
     if (!r_equationSolving_a(&a)) return FALSE;
     if (!r_equationSolving_b(&b)) return FALSE;
     if (!r_equationSolving_c(&c)) return FALSE;
     OK = a != 0.0;
     SecondDegree_Cluster_4(&SecondDegree1,OK,a);
     SecondDegree_Cluster_5(&SecondDegree1,OK,b);
     SecondDegree_Cluster_6(&SecondDegree1,OK,c);
     XZX_116 = !OK;
     FirstDegree_Cluster_2
       (&FirstDegree2,XZX_116,b,&C_bb,&C_error);
     FirstDegree_Cluster_3(&FirstDegree2,XZX_116,c);
     C_error_183 = (XZX_116 ? C_error : FALSE);
     if (XZX_116) FirstDegree_Cluster_1
                    (&FirstDegree2,XZX_116,&a2,&err2);
     if (OK) {
         SecondDegree_Cluster_2(&SecondDegree1,OK);
         SecondDegree_Cluster_3(&SecondDegree1,OK);
     }}
 SecondDegree_Cluster_7(&SecondDegree1,OK);
 SecondDegree_Cluster_1
   (&SecondDegree1,OK,&err1,&a1,&x2,&C_y,&C_err,&C_x);
 C_error = FALSE;
 C_bb_177 = (XZX_116 ? C_bb : FALSE);
 C_x1 = C_x || C_bb_177;
 if (stable) {
     err1_211 = (C_err ? err1 : FALSE);
     C_oc = C_error_183 || err1_211;
     if (C_oc)
      {
       if (err1_211) oc = TRUE; else oc = err2;
       w_equationSolving_oc(oc);
      }
   }
 if (C_y) w_equationSolving_x2(x2);
 if (C_x1) {
     if (C_x) x1 = a1; else x1 = a2;
     w_equationSolving_x1(x1);
   }
 equationSolving_STEP_finalize();
 return TRUE;
}
EXTERN logical equationSolving_STEP_finalize() {
 SecondDegree_Cluster_delays(&SecondDegree1,&stable);
 C_error = FALSE;
 equationSolving_STEP_initialize();
 return TRUE;
}
```

Figure 25: Generated C code of the solver importing the first and second degree clusters

8.3 Legacy code encapsulation for separate compilation

Just as with the separate compilation functionalities we just described, foreign programs, in C or Java code, can be encapsulated with a grey-box or black-box profile and used in a Signal program.

Such an imported program module can be declared as a `process` (like the second degree function), an `action` (a cluster), `node` (a Scade-like program) or `function` (it is combinatoric). We consider here only the `process` and the `function` models. A `process` may be declared *safe* (if it doesn't have any side-effect), *deterministic*, or *unsafe* (by default).

For instance, the arithmetic C function `abs` could be declared as `function abs=(? real x; ! real s;)`. Implicitly, this would mean that it has no side-effect, that its input and output signals are synchronous, i.e. `x ^= s` and that its output depends on its input, i.e. `x --> s`.

Let us consider the slightly bigger example of the first degree function `FirstDegree`. Figure 26 displays its C code (left) and necessary profile (right) before import in the solver program. One notice that `FirstDegree` has no side effect. However, output variables `oc` and `x` are not always produced. One is produced when `cond` is false, the other when it is true. Therefore, its profile cannot be declared as a `function`. It must be declared as a *safe* `process`.

As a consequence, its synchronization and scheduling relation must be made explicit. They are that the output signals depend on all input signals, written `{cond , b, c} --> {x , oc}`. The signal x is synchronized to the condition that `cond` is true and that b is non-zero. The signal oc is present iff `cond` is false and b is zero.

```
extern void FirstDegree(int cond, float b, float c,
                        float *x, int *oc) {
  if (cond) {
      if (b != 0.0)
        *x = -(c/b);
      else
        *oc = (c !=0.0);
    }
}
```

```
process FirstDegree =
      ( ? boolean cond; real b, c;
        ! real x; boolean oc; )
      safe
      spec (| {cond , b, c} --> {x , oc}
            | x  ^= when cond when (b/=0.0)
            | oc ^= when not cond when (b=0.0)
            |)
      external "C" ;
```

Figure 26: Legacy C code and profile of the first degree function

The profile of Figure 26, right, can actually be automatically inferred from its source C code, left, by using Polychrony's functionality for interpreting C code in SSA form. Figure 27 displays the result of this interpretation, left, and the abstraction of the generated process, right. In the encoding of the SSA form of the first degree function, each sequence of instruction is associated with a label (here L1 to L3). Each individual instruction is translated by an equation and each label is translated by a boolean signal that guards its activation. As a result, the Signal interpretation has an identical behavior as the original C program.

```
process FirstDegree =
    ( ? boolean cond;
        real b;
        real c;
      ! real x;
        boolean oc;
    )
  pragmas
     I_Names {cond,b,c}
     O_Names {x,oc}
  end pragmas
  (| (| __pK_1 := b_2 ^+ c_3 |)
   | cond_1 ^= b_2 ^= c_3 ^= bb_0
   | (| D_1739_6 := ((c_3 cell __pK_1)
                     /(b_2 cell __pK_1)) when L1
      | D_1740_7 := (-D_1739_6) when L1
      | x_8 := D_1740_7 when L1
      |)
   | (| D_1741_4 := (c_3/=0.0) when L2
      | oc_5 := D_1741_4 when L2
      |)
   | when bb_0 ^= cond ^= b ^= c
   | (| cond_1 := cond cell (^bb_0)
      | b_2 := b cell (^bb_0)
      | c_3 := c cell (^bb_0)
      |)
   | (| x := x_8 when L3
      | oc := oc_5 when L3
      |)
   | (| bb_0 := (not (^bb_0))$1 init true
      | L0 := (cond_1) when bb_0
      | L1 := (b_2/=0.0) when L0
      | L2 := (not (b_2/=0.0)) when L0
      | L3 := (true when L2) default (true when L1)
              default (not (cond_1) when bb_0
      |)
  |)
  where
  event __pK_1;
  boolean bb_0,L0,L1,L2,L3;
  real D_1739_6, b_2, c_3, x_8;
  real D_1740_7;
  boolean D_1741_4, cond_1, oc_5;
  end
%FirstDegree%;
process FirstDegree_ABSTRACT =
    ( ? boolean cond;
        real b, c;
      ! real x;
        boolean oc, bb_0, C_x, C_oc;
    )
    spec (| (| cond --> x
             | b --> x
             | c --> x
             | cond --> oc
             | b --> oc
             | c --> oc
             |)
          | (| bb_0 ^=  bb_0
             | when bb_0 ^= cond ^= b
                ^= c ^= C_x ^= C_oc
             | when C_x ^= x
             | when C_oc ^= oc
             |)
          |)
  pragmas
     BlackBox "FirstDegree"
  end pragmas
  external "C"
%FirstDegree_ABSTRACT%;
```

Figure 27: Model and abstraction of the legacy first degree function

Moreover, the abstraction of that process by its synchronization and scheduling relations (Figure 27, right), once projected on the input-output signals of the process, displays similar relations as that given above, Figure 26.

The last figure, Figure 28 displays the generated code of the solver in which the `FirstDegree` process is an external C program written and when `SecondDegree` has been compiled separately into clusters (section 8.2).

```
EXTERN logical equationSolving1_initialize () {
  stable = TRUE;
  SecondDegree_initialize (&SecondDegree1);
  equationSolving1_STEP_initialize ();
  return TRUE;
}
static void equationSolving1_STEP_initialize () {
  C_err = FALSE;
  OK = FALSE;
  err1 = FALSE;
  XZX_110 = FALSE;
  C_err2 = FALSE;
}
EXTERN logical equationSolving1_iterate () {
  if (stable) {
      if (!r_equationSolving1_a (&a)) return FALSE;
      if (!r_equationSolving1_b (&b)) return FALSE;
      if (!r_equationSolving1_c (&c)) return FALSE;
      OK = a != 0.0;
      SecondDegree_Cluster_4 (&SecondDegree1, OK, a);
      SecondDegree_Cluster_5 (&SecondDegree1, OK, b);
      SecondDegree_Cluster_6 (&SecondDegree1, OK, c);
      XZX_110 = !OK;
      FirstDegree (XZX_110, b, c, &a2, &err2);
      C_err2 = XZX_110 && (b == 0.0);
      if (OK)
      {
       SecondDegree_Cluster_2 (&SecondDegree1, OK);
       SecondDegree_Cluster_3 (&SecondDegree1, OK);
    }}
  SecondDegree_Cluster_7 (&SecondDegree1, OK);
  SecondDegree_Cluster_1 (&SecondDegree1, OK, &err1, &a1,
                          &x2, &C_y, &C_err, &C_x);
  C_a2_179 = (stable ? ((b != 0.0) && XZX_110) : FALSE);
  C_x1 = C_x || C_a2_179;
  if (stable) {
      err1_207 = (C_err ? err1 : FALSE);
      C_oc = C_err2 || err1_207;
      if (C_oc)
        {
         if (err1_207) oc = TRUE; else oc = err2;
         w_equationSolving1_oc (oc);
    }}
  if (C_y) w_equationSolving1_x2 (x2);
  if (C_x1) {
      if (C_x) x1 = a1; else x1 = a2;
      w_equationSolving1_x1 (x1);
    }
  equationSolving1_STEP_finalize ();
  return TRUE;
}
EXTERN logical equationSolving1_STEP_finalize () {
  SecondDegree_Cluster_delays (&SecondDegree1, &stable);
  equationSolving1_STEP_initialize ();
  return TRUE;
}
```

Figure 28: Generated code of the solver calling external C and clustered Signal modules

9 Conclusion

In this report, we have informally presented all the code generation strategies available in the Polychrony toolset [1] by considering a program solving second degree equations. While mathematically simple, this program exhibits non-trivial modes, synchronization relations, scheduling relations, which we analyzed and transformed to illustrate several usage scenarios and apply one of the following code generation strategies :

- Sequential code generation, Section 4, consists of producing one iterate function for a complete Signal program.
- Clustered code generation with static scheduling, Section 5, consists of partitioning the generated code into one cluster per set of input signals. The iterate function is a static scheduler of these clusters
- Clustered code generation with dynamic scheduling, Section 6, consists of a dynamic scheduling a set of program clusters.
- Distributed code generation, Section 7, consists of physically partitioning a program across several locations and of installing point to point communications between them.
- Sequential code generation for separate compilation, Section 8.1, consists of associating the sequential generated code of a process with a profile describing its synthesized synchronization and scheduling relations. The calling context of the process is passed to it as parameter.

- Clustered code generation for separate compilation, Section 8.2, consists of associating the clustered generated code of a process with a profile describing the synchronization and scheduling relations of and between its clusters. The scheduler of the process is generated in each call context.
- Legacy code encapsulation for separate compilation, Section 8.3, consists of associating a foreign function or procedure with a profile which declares its synchronization and scheduling relations, for the purpose of properly calling it from within a Signal program.

List of Figures

References

[1] Espresso Team, IRISA. Polychrony tool. http://www.irisa.fr/espresso/Polychrony.

[2] Loïc Besnard, Thierry Gautier, and Paul Le Guernic. Signal V4-Inria Version: Reference manual. http://www.irisa.fr/espresso/Polychrony.

[3] Tochéou Amagbegnon, Loïc Besnard,and Paul Le Guernic Inria report n. 2290, 1994. Arborescent Canonical Form of Boolean Expressions.

[4] Thierry Gautier and Paul Le Guernic. Code generation in the SACRES project. Towards System Safety, Proceedings of the Safety-critical Systems Symposium, SSS'99, Huntingdon, UK, Springer, 1999, 127-149.

[5] Pascal Aubry, Paul Le Guernic, and Sylvain Machard. Synchronous distribution of Signal programs. In Proc. of the 29th Hawaii International Conference on System Sciences, vol. 1. 1996, IEEE Computer Society Press, 656–665.

[6] Olivier Maffeïs and Paul Le Guernic. Distributed Implementation of SIGNAL: Scheduling & Graph Clustering. In 3rd International School and Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems,1994. LNCS vol. 863, Springer-Verlag, p547-566.

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399