Compilation of Polychronous Data Flow Equations

Loïc Besnard, Thierry Gautier, Paul Le Guernic, Jean-Pierre Talpin

Compilation of polychronous data flow equations

Loïc Besnard, Thierry Gautier, Paul Le Guernic, and Jean-Pierre Talpin

1 Introduction

High-level embedded system design has gained prominence in the face of rising technological complexity, increasing performance requirements and shortening time to market demands for electronic equipments. Today, the installed base of intellectual property (IP) further stresses the requirements for adapting existing components with new services within complex integrated architectures, calling for appropriate mathematical models and methodological approaches to that purpose.

Over the past decade, numerous programming models, languages, tools and frameworks have been proposed to design, simulate and validate heterogeneous systems within abstract and rigorously defined mathematical models. Formal design frameworks provide well-defined mathematical models that yield a rigorous methodological support for the trusted design, automatic validation, and systematic test-case generation of systems. However, they are usually not amenable to direct engineering use nor seem to satisfy the present industrial demand.

Despite overwhelming advances in embedded systems design, existing techniques and tools merely provide *ad-hoc* solutions to the challenging issue of the so-called *productivity gap* [1]. The pressing demand for design tools has sometimes hidden the need to lay mathematical foundations below design languages. Many illustrating examples can be found, e.g. the variety of very different formal semantics found in state-diagram formalisms [2]. Even though these design languages benefit from decades of programming practice, they still give rise to some diverging interpretations of their semantics.

Loïc Besnard
CNRS/IRISA, Campus de Beaulieu, Rennes, France, e-mail: Loic.Besnard@irisa.fr
Thierry Gautier, Paul Le Guernic, Jean-Pierre Talpin
INRIA, centre de recherche Rennes - Bretagne Atlantique, Campus de Beaulieu, Rennes, France, e-mail: Thierry.Gautier@inria.fr, Paul.LeGuernic@inria.fr, Jean-Pierre.Talpin@inria.fr

The need for higher abstraction-levels and the rise of stronger market constraints now make the need for unambiguous design models more obvious. This challenge requires models and methods to translate a high-level system specification into (distributed) cooperating (sequential) processes and to implement high-level semantics-preserving transformations such as hierarchical code structuration, sequentialization or desynchronization (protocol synthesis).

Synchronous hypothesis, in this aim, has focused the attention of many academic and industrial actors. This synchronous paradigm consists of abstracting the non-functional implementation details of a system. In particular, latencies due to effective computing and communications depend on actual implementation architecture; thus they are handled when low level implementation contraints are considered; at higher level, time is abstracted as sequences of (multiple) events in a *logical time* model. Thus the designer can forget those details and focus his or her attention on the functionalities of the system, including logical synchronizations.

With this point of view, synchronous design models and languages provide intuitive models for embedded systems [3]. This affinity explains the ease of generating systems and architectures, and verifying their functionalities using compilers and related tools that implement this approach.

Synchronous languages rely on the synchronous hypothesis: computations and behaviors of a *synchronous process* are divided into a discrete sequence of atomic *computation steps* which are equivalently called *reactions* or *execution instants*. In itself this assumption is rather common in practical embedded system design.

But the synchronous hypothesis adds to this the fact that, *inside each instant*, the behavioral propagation is well-behaved (causal), so that the status of every signal or variable is established and defined prior to being tested or used. This criterion ensures strong semantic soundness by allowing universally recognized mathematical models to be used as supporting foundations. In turn, these models give access to a large corpus of efficient optimization, compilation, and formal verification techniques.

The *polychronous* model [4] extends the synchronous hypothesis to the context of multiple logical clocks: several synchronous processes can run asynchronously until some communication occurs; all communications satisfy the synchronous hypothesis. The resulting behaviors are then partial orders of reactions, which is obviously more general than simple sequences. This model goes beyond the domain of purely sequential systems and synchronous circuits; it embraces the context of complex architectures consisting of synchronous circuits and desynchronization protocols: globally asynchronous and locally synchronous architectures (GALS).

The SIGNAL **language** [5] supports the polychronous model. Based on data flow and equations, it goes beyond the usual scope of a programming language, allowing for specifications and properties to be described. It provides a mathematical foundation to a notion of *refinement*: the ability to model a system from the early stages of its requirement specifications (relations, properties) to the late stages of its syn-

thesis and deployment (functions, automata). The inherent flexibility of the abstract notion of *signal* handled in the SIGNAL language invites and favors the design of correct-by-construction systems by means of well-defined model transformations that preserve both the intended semantics and stated properties of the architecture under design.

The integrated development environment POLYCHRONY [6] provides SIGNAL program transformations that draw a continuum from synchrony to asynchrony, from specification to implementation, from abstraction to refinement, from interface to implementation. SIGNAL gives the opportunity to seamlessly model embedded systems at multiple levels of abstraction while reasoning within a simple and formally defined mathematical model. It is being extended by plugins to capture SystemC modules or real-time Java classes within the workbench. It allows to perform validation and verification tasks, e.g. with the integrated SIGALI model checker [7], or with the Coq theorem prover [8]. C, C++, multi-threaded and real-time Java and SYNDEX [9] code generators are provided.

This chapter focuses on formal transformations, based on the polychronous semantic model [4], that can be provided by a safe methodology to generate "correct-by-construction" executable code from SIGNAL processes. It gives a thorough presentation on program analysis and code generation techniques that can be implemented to transform synchronous multi-clocked equation systems into various execution schemes such as sequential and concurrent programs (C) or object-oriented programs (C++). Most of these techniques are available in the toolset POLYCHRONY to design embedded real-time applications.

This chapter is structured as follows: Section 2 presents the main features of the SIGNAL language and introduces some of the mathematical properties on which program transformations are based. In Section 3, some of the modularity features of SIGNAL are first introduced; then an example, used in the rest of this chapter, is presented. Section 4 is dedicated to Data Control Graph models that support program transformations; their use to guide various code generation schemes that are correct by construction is then considered. Section 5 introduces those various modes of code generation, illustrated on the considered example.

2 SIGNAL language

SIGNAL [10, 11, 12, 13, 14, 15] is a declarative language expressed within the polychronous model of computation. SIGNAL relies on a handful of primitive constructs, which can be combined using a composition operator. These core constructs are of sufficient expressive power to derive other constructs for comfort and structuring. In the following, we present the main features of the SIGNAL language and its associated concepts. We give a sketch of the primitive constructs and a few derived constructs often used. For each of them, the corresponding syntax and definition are mentioned. Since the semantics of SIGNAL is not the main topic of this chapter, we

give simplified definitions of operators. For further details, we refer the interested reader to [4, 5].

2.1 Synchronized data flow

Consider as an example the following expression in some conventional data flow formalism:

$$\textbf{if } a > 0 \textbf{ then } x = a \textbf{ endif};\ y = x + a$$

Considering the data flow semantics given by Kahn [16] as functions over flows, y is the greatest sequence of values $a'_t + a_t$ where a' is the subsequence of strictly positive values in a. Thus in an execution where the edges are considered as FIFO queues [17], if a is a sequence with infinitely many non-positive values, the queue associated with a grows forever, or (if a is a finite sequence) the queue associated with x remains eventually empty although a is non-empty. Now, suppose that each FIFO queue consists of a single cell [18]. Then as soon as a negative value appears on the input, the execution (of the $+$ operator) can no longer go on because its first operand (cell) does not hold a value (is absent): there is a deadlock. These results are not acceptable in the context of embedded systems where not only deadlocks but also uncontrolled time responses can be dramatic. Synchronized data flow introduces synchronizations between occurrences of flows to prevent such effects. In this context, the lack of value is usually represented by *nil* or *null*; we represent it by the symbol # (stating for "no event").

It would be somewhat significant if such deadlocks could be statically prevented. For that, it is necessary to be able to statically verify timing properties. Then the # should be handled when reasoning about time. In the framework of synchronized data flow, the # will correspond to the absence of value at a given logical instant for a given variable (or *signal*). In particular, to reach high level modularity allowing for instance internal clock rate increasing (*time refinement*), it must be possible to insert #'s between two defined values of a signal. Such an insertion corresponds to some resynchronization of the signal. However, the main purpose of synchronized data flow is to completely handle the whole synchronization at compile time, in such a way that the execution phase has nothing to do with #. This is assumed by a static representation of the timing relations expressed by each operator. Syntactically, the overall detailed timing are implicit in the language. SIGNAL describes processes which communicate through (possibly infinite) sequences of (typed) values with implicit timing: the *signals*.

2.2 Signal, execution, process in SIGNAL

More precisely, a *pure signal s* is a (total) function $T \rightarrow D$, where T, its *time domain*, is a chain in a partial order (for instance an increasing sequence of integers) and D is some data type; we name *pure flow* of such a pure signal s, the sequence of its values in D.

For all chains TT, $T \subset TT$, a pure signal $s : T \rightarrow D$ can be extended to a *synchronized signal* $ss : TT \rightarrow D_{\#}$ (where $D_{\#} = D \cup \{\#\}$) such that for all t in T, $ss(t) = s(t)$ and $ss(t) = \#$ when t is not in T; we name *synchronized flow of a synchronized signal* ss the sequence of its values in $D_{\#}$; conversely we name *pure signal of the synchronized flow* ss, the unique pure signal s from which it is extended and *pure flow of* ss the pure flow of s . The *pure time domain* of a synchronized signal is the time domain of its pure signal. When it is clear from the context, one may omit *pure* or *synchronized* qualifications. Given a pure signal s (respectively, a synchronized signal ss), s_t (respectively, ss_t) denotes the t^{th} value of its pure flow (respectively, its synchronized flow).

An *execution* is the assigment of a tuple of synchronized signals defined on the same time domain (as synchronized signals), to a tuple of variables. Let TT be the time domain of an execution, a *clock* in this execution is a "characteristic function" $clk : TT \longrightarrow \{\#, true\}$; notice that it is a synchronized signal. The clock of a signal x in an execution is the (unique) clock that has the same *pure time domain* as x; it is denoted by $\hat{x}$.

A *process* is a set of executions defined by a system of equations over signals that specifies relations between signal values and clocks. A *program* is a process.

Two signals are said to be *synchronous in an execution* iff they have the same clock (or equivalently the same pure time domain in this execution). They are said to be *synchronous in a process* (or simply *synchronous*), iff they are *synchronous* in all executions of this process.

Consider a given operator which has, for example, two input signals and one output signal all being synchronous. They are *logically related* in the following sense: for any t, the t^{th} token on the first input is evaluated with the t^{th} token on the second input, to produce the t^{th} token on the output. This is precisely the notion of *simultaneity*. However, for two occurrences of a given signal, we can say that one is before the other (*chronology*). Then, for the synchronous approach, an *event* is associated with a set of instantaneous calculations and communications.

2.3 SIGNAL *data types*

A flow is a sequence of values that belong to the same data type. Standard data types such as Boolean, integer... (or more specific ones such as *event*—see below) are provided in the SIGNAL language. One can also find more sophisticated data types such as *sliding window* on a signal, *bundles* (a structure the fields of which

are signals that are not necessarily synchronous), used to represent union types or signal multiplexing.

- The `event` type: to be able to compute on (or to check properties of) clocks, SIGNAL provides a particular type of signals called `event`. An `event` signal is *true* if and only if it is present (otherwise, it is #).
- Signal declaration: `tox x` declares a signal `x` whose common element type is `tox`. Such a declaration is a process that contains all executions that assign to `x` a signal the image of which is in the domain denoted by `tox`.

In the remainder of this chapter, when the type of a signal does not matter or when it is clear from the context, one may omit to mention it.

2.4 SIGNAL *elementary processes*

An elementary process is defined by an equation that associates with a signal variable an expression built on operators over signals; the arguments of operators can be expressions and variables.

• **Stepwise extensions.** Let `f` be a symbol denoting a n-ary function $[\![f]\!]$ on values (e.g., Boolean, arithmetic or array operation). Then, the SIGNAL expression

```
y := f(x1,...,xn)
```

defines the process equal to the set of executions that satisfy:

$$\begin{cases} - \text{ the signals } y,\ x1,\ \ldots,\ xn \text{ are synchronous,} \\ - \text{ their pure flows have same length } l \text{ and satisfy } \forall t \leq l, y_t = [\![f]\!](x1_t, \ldots, xn_t) \end{cases}$$

If f is a function, its stepwise extension is a *pure flow* function. Infix notation is used for usual operators.

Derived operator

- *Clock of a signal*: `^x` returns the *clock* of `x`; it is defined by `(^x)` $=_{\Delta}$ `(x = x)`, where = denotes the stepwise extension of usual equality operator.

• **Delay.** This operator defines the signal whose t^{th} element is the $(t-1)^{th}$ element of its (pure flow) input, at any instant but the first one, where it takes an initialization value. Then, the SIGNAL expression

```
y := x $ 1 init c
```

defines the process equal to the set of executions that satisfy:

$$\begin{cases} - \ y,\ x \text{ are synchronous,} \\ - \text{ pure flows have same length } l \text{ and satisfy } \forall t \leq l, \begin{cases} (t > 1) \Rightarrow y_t = x_{t-1} \\ (t = 1) \Rightarrow y_t = c \end{cases} \end{cases}$$

The delay operator is thus a *pure flow* function.

Derived operator

- *Constant*: `x := v`; when `x` is present its value is the constant value `v`;
  `x := v` is a derived equation equivalent to `x := x $ 1 init v`.
  Note that this equation does not have input: it is a pure flow function with arity 0.

• **Sampling.** This operator has one data input and one Boolean "control" input. When one of the inputs is absent, the output is also absent; at any logical instant where both input signals are defined, the output is present (and equal to the current data input value) if and only if the control input holds the value *true*. Then, the SIGNAL expression

`y := x when b`

defines the process equal to the set of executions that satisfy:

$$\begin{cases} - \ y,\ x,\ b \text{ are extended to the same infinite domain } T, \text{ respectively as } yy,\ xx,\ bb, \\ - \ \text{synchronized flows are infinite and satisfy } \forall t \in T \begin{cases} (bb_t = true) \Rightarrow yy_t = xx_t \\ (bb_t \neq true) \Rightarrow yy_t = \# \end{cases} \end{cases}$$

The `when` operator is thus a *synchronized flow* function.

Derived operators

- *Clock selection*: `when b` returns the clock that represents the (implicit) set of instants at which the signal `b` is *true*; in semantics, this clock is denoted by $[b]$.
  `(when b)` $=_{\Delta}$ `(b when b)` is a pure flow function.
- *Null clock*: the signal `when(b when (not b))` is never present: it is called *null clock* and is denoted by `^0` in the SIGNAL syntax, $\hat{0}$ as a semantic constant.
- *Clock product*: `x1^* x2` (denoted by $\hat{*}$ as a semantic operator) returns the clock that represents the intersection of pure time domains of the signals `x1` and `x2`. When their clock product is $\hat{0}$, `x1` and `x2` are said to be *exclusive*.
  `(x1^* x2)` $=_{\Delta}$ `((^x1) when (^x2))`

• **Deterministic merging.** The unique output provided by this operator is defined (i.e., with a value different from #) at any logical instant where at least one of its two inputs is defined (and non-defined otherwise); a priority makes it deterministic. Then, the SIGNAL expression

`z := x default y`

defines the process equal to the set of executions that satisfy:

$$\begin{cases} - \ \text{the time domain } T \text{ of } z \text{ is the union of the time domains of } x \text{ and } y, \\ - z,\ x,\ y \text{ are extended to the same infinite domain } TT \supseteq T, \text{ resp. as } zz,\ xx,\ yy, \\ - \ \text{synchronized flows satisfy } \forall t \in TT \begin{cases} (xx_t \neq \#) \Rightarrow zz_t = xx_t \\ (xx_t = \#) \Rightarrow zz_t = yy_t \end{cases} \end{cases}$$

The `default` operator is thus a *synchronized flow* function.

Derived operators

- *Clock union (or clock max)*: `x1^+ x2` (denoted by $\hat{+}$ as a semantic operator) returns an event signal that is present iff `x1` or `x2` is present.
  `(x1^+ x2)` $=_{\Delta}$ `((^x1) default (^x2))`
- *Clock difference*: `x1^- x2` returns an event signal that is present iff `x1` is present and `x2` is absent.
  `(x1^- x2)` $=_{\Delta}$ `(when ((not ^x2) default ^x1))`

Derived equations

- *Partial signal definition*: `y ::= x` is a partial definition for the signal `y` which is equal to `x`, when `x` is defined; when `x` is not defined its value is free.
  `(|y ::= x|)` $=_{\Delta}$ `(|y := x default y|)`
  This process is generally non deterministic. Nevertheless, it is very useful to define components such as transitions in automata, or modes in real-time processes, that contribute to the definition of the same signal. Moreover, it is heavily used in the process of code generation (communication of values via *shared variables*, see 3.1).
  The clock calculus can compute sufficient conditions to guarantee that the overall definition is consistent (different components cannot give different values at the same instant) and total (a value is given at all instants of the time domain of `y`).

2.5 SIGNAL *process operators*

A process is defined by composing elementary processes.

• **Restriction.** This operator allows one to consider as local signals a subset of the signals defined in a given process. If `x` is a signal with type `tox` defined in a process `P`,

`P where tox x` or `P where x`

defines a new process `Q` where communication ways (for composition) are those of `P`, except `x`. Let *A* the variables of `P` and *B* the variables of `Q`: we say that `P` is *restricted* to *B*, and executions of `P` are restricted in `Q` to variables of `B`. More precisely, the executions in `Q` are the executions in `P` from which `x` signal is removed (the projection of these executions on remaining variables). This has several consequences:

– the clock of each execution may be reduced,
– the ability to (directly) add new synchronization constraints to `x` is lost,
– if `P` has a single output signal named `x`, then `P where x` is a pure synchronization process. The generated code (if any) is mostly a synchronization code used to ensure signal occurrence consumptions.
– if `P` has a single signal named `x`, `P where x` denotes the neutral process: it cannot influence any other process. Hence no code is generated for it.

Derived equations

- `(| P where x, y |)` $=_{\Delta}$ `(| (| P where x |) where y |)`
- *Synchronization*: `x1 ^= x2` specifies that `x1` and `x2` are synchronous.
  `(| x1 ^= x2 |)` $=_{\Delta}$ `(| h := (^x1 = ^x2) |) where h`
- *Clock inclusion*: `x1 ^< x2` specifies that time domain of `x1` is included in time domain of `x2`.
  `(| x1 ^< x2 |)` $=_{\Delta}$ `(| x1 ^= (x1 ^* x2) |)`

• **Parallel composition**: Resynchronizations (by freely inserting #) have to take place when composing processes with common signals. However, this is only a formal manipulation. If `P` and `Q` denote two processes, the *composition* of `P` and `Q`, written `(| P | Q |)` defines a new process in which common names refer to common synchronized signals. Then, `P` and `Q` communicate (synchronously) through their common signals. More precisely, let `XPP` (resp., `XQQ`) be the variables of `P` (resp., `Q`); the executions in `(| P | Q |)` are the executions whose projections on `XPP` are executions of `P`, and projections on `XQQ` are executions of `Q`. In other words, `(| P | Q |)` defines the set of behaviors that satisfies both `P` and `Q` constraints (equations).

Derived operators

- `y := x cell c init x0` behaves as a synchronized memory cell: `y` is present with the most recent value of `x` when `x` is present or `c` is present and *true*. It is defined by the following program:
  `(| y := x default (y $1 init x0) | y ^= x ^+ when c |)`

- `y := var x init x0` behaves as a standard memory cell: when `y` is present, its value is the most recent value of `x` (including the current instant); the clock of `x` and the clock of `y` are mostly independent (the single constraint is that their time domains belong to a common chain). It is defined by the following program:
  `y := (x cell ^y init x0) when ^y`

Polychrony example: `(| x := a | y := b|)` defines a process that has two independent clocks. This process is a Kahn process (i.e., is a flow function); it can be executed as two independent threads, on the same processor (provided that the scheduler is fair) or on distinct processors; it can also be executed as a single reactive process, scanning its input and then executing none, one or two assignments depending on the input configuration.

2.6 Parallel semantics properties of SIGNAL

A SIGNAL specification close to the example given in Section 2.1

if $a > 0$ **then** $x = a$ **endif**; $y = x + a$

is the following "DeadLocking Specification":

```
DLS ≡ (| x := a when a > 0 | y := x + a |)
```

`DLS` denotes the set of executions in which $a_t > 0$ for all t. Then safe execution requires this program to be rejected if one cannot prove (or if it is not asserted) that the signal `a` remains strictly positive when it is present.
Embedding `DLS` without changing it with the following front-end process results in a safe program that accepts negative values for some occurrences of `a`; these negative values are not transmitted to `DLS`:

```
ap := a when a > 0 | (| (| DLS | a := ap |) where a |)
```

Process expression in normal form. The following properties of parallel composition are intensively used to compile processes:

- associativity: `(| P | Q |) | R ≡ P | (| Q | R |)`
- commutativity: `P | Q ≡ Q | P`
- idempotence: `P | P ≡ P` is satisfied by processes that do not produce side effects (for instance due to call to system functions). This property allows to replicate processes.
- externalization of restrictions: if `x` is not a signal of `P`,
  `P | (| Q where x |) ≡ (| P | Q |) where x`

Hence, a process expression can be normalized, modulo required variable substitution, as the composition of elementary processes, included in terminal restrictions.

Signal expression in normal form. Normalizations can be applied to expressions on signals thanks to properties of the operators, for instance:

- `when` is associative and right-commutative:
  `(a when b)when c ≡ a when(b when c) ≡ a when(c when b)`
- `default` can be written in exclusive normal form:
  `a default b ≡ a default (b when (b^-a))`
- `default` is assocative and `default` commutes in exclusive normal form:
  if `a` and `b` are exclusive then `a default b ≡ b default a`
- `when` is right-distributive over `default`:
  `(a default b) when c≡ (a when c) default (b when c)`
- Boolean normalization: logical operators can be written as expressions involving only `not`, `false` and operators on `event` type. For example:
  `AB := a or b ≡ (| AB := (when a) default b | a ^= b |)`

Process abstraction. A process *Pa* is, by definition, a process abstraction of a process *P* if $P|Pa = P$. This means that every execution of *P* restricted to variables of *Pa* is an execution of *Pa* and thus all safety properties satisfied by executions of *Pa* are also satisfied by executions of *P*.

3 Example

As a full example to illustrate the SIGNAL features and the compilation techniques (including code generation) we propose the description of a process that solves iteratively equations $aX^2 + bX + c = 0$. This process has three synchronous signals `a`, `b`, `c` as inputs. The output signals `x1`, `x2` are the computed solutions. When there is no solution at all or infinitely many solutions, their synchronized flows hold # and an output Boolean `x_st` is set. Before presenting this example let us introduce modularity features than can be found in SIGNAL.

3.1 SIGNAL *modularity features*

Process model: Given a process (a set of equations) `P_body`, a *process model* `MP` associates an interface with `P_body`, such that `P_body` can be expanded using this interface. A process model is a SIGNAL term

```
process MP ( ? t_I1 I1; ...; t_Im Im;
             ! t_O1 O1; ...; t_On On )
     P_body
```

that specifies its typed input signals after "`?`", and its typed output signals after "`!`". Assuming that there is no name conflict (such a conflict is solved by trivial renaming), the instantiation of `MP` is defined by:

```
(| (Y1, ..., Yn) := MP(E1, ..., Em) |)
                          =Δ
(| I1:=E1 |...| Im:=Em | P_body | Y1:=O1 |...| Yn:=On |)
   where t_I1 I1; ...; t_Im Im; t_O1 O1; ...; t_On On
```

Example When `a` is equal to 0, the second degree equation becomes $bX + c = 0$. This first degree equation is solved using the `FirstDegree` process model.

```
process FirstDegree = ( ? real b, c; ! boolean x_st; real x; )
  (| b ^= c
   | b1  := b when (b/=0.0)
   | c1  := c when (b/=0.0)
   | x   := -(c1/b1)
   | x_st := (c/=0.0) when (b=0.0)
   |) where real b1, c1; end
```

When the factor `b` is not 0, the output signal `x` holds the value of the solution (`-c/b`), the `x_st` Boolean signal is absent. Conversely, when `b` is 0, `x` is absent and `x_st` is either *true* when the equation has no solution (`c` is not 0) or *false* when the equation has infinitely many solutions (`c` is 0). This process is activated when the input parameter `a` equals 0. This is achieved by:

```
(x_st_1, x11) := FirstDegree (b when (a=0.0), c when (a=0.0))
```

The *interface* of a process model can begin with a list of static parameters given between "`{`" and "`}`"; see for instance `real epsilon` in the interface of `rac` (Example 3.2).

Local process model: A process model can be declared local to an other process model as `process rac` local to `process SecondDegree` in Example 3.2.

Shared variables: A variable `x` that has partial definitions (2.4) in a process model `MP`, or in process models local to `MP`, is declared as variable `shared real x`, local to `MP`. A shared signal `x` cannot appear in the interface of any process.

3.2 Full example with time refinement

In our example, the resolution of the equation $aX^2 + bX + c = 0$ uses the iterative Newton method: starting from $\Delta \geq 0$, the computation of $R = \sqrt{\Delta}$ is defined by the limit of the series $(R_n)_{n\geq 0}$:

$$\Delta = b^2 - 4ac \qquad R_0 = \frac{\Delta}{2} \qquad R_{n+1} = \frac{(R_n * R_n + \Delta)/R_n}{2} \ (n \geq 0)$$

The iterative method for computing the square root of the discriminant is implemented in SIGNAL using a time refinement of the clock of the discriminant.

The process model `rac` computes in `R` the sequence of roots R_t of the values of a signal S_t assigned to `S` (corresponding to the discrimant when it is not negative); `epsilon` is a static threshold parameter used to stop Newton iteration.

```
process rac = { real epsilon; }
  ( ? real S; ! boolean stable; real R; )
  (| (| S_n := var S
      | R_n := (S/2.0) default (next_R_n $1 init 1.0)
      | next_R_n := (((R_n+(S_n/R_n))/2.0) when loop) default R_n
      |) where real S_n; end
   | (| loop := (^S) default (not stable)
      | next_stable := abs(next_R_n-R_n)<epsilon
      | stable := next_stable $1 init true
      | R_n ^= stable
      | R := R_n when (next_stable and loop)
      |) where boolean next_stable; end
   |) where real R_n, next_R_n; boolean loop; end
```

The signal `S_n` holds the current value of `S` ($S_{t,n} = S_{t,0} = S_t$). The signal `R_n` is the current approximation of the current root to be computed ($R_{t,n}$, with $R_{t,0} = S_t/2$). The signal `next_R_n` is $R_{t,n+1}$. The signal `stable` is first *true*, then $false$ until the instant following the emission of the result in (`R`).

The clock that triggers steps is that of the signal `stable`: it represents a refinement of time, with respect to that of the input signal `S`.

The process model `SecondDegree` uses `rac` to compute the solutions of the second degree equation when the discriminant is positive.

```
process SecondDegree = { real epsilon; }
  ( ? real a, b, c; ! event x_st; real x21, x2; boolean stable)
  (| delta := (b*b)-(4.0*a*c)
   | x_st := when (delta<0.0)
   | x1_1 := (-b/(2.0*a)) when (delta=0.0)
   | (| (stable, delta_root) := rac{epsilon}(delta when (delta>0.0))
      | aa := var a | bb := var b
      | x1_2 := -((bb+delta_root)/(2.0*aa))
      | x2 := -((bb-delta_root)/(2.0*aa)) |) where aa, bb, delta_root; end
   | x21 := x1_1 default x1_2
   |) where delta, x1_1, x1_2;
            process rac ... end
```

When the discriminant `delta` is negative, the current equation does not have solution: the event `x_st` output signal is present, `x21` and `x2` are absent. When `delta` is 0 there is a single solution held by `x1_1` and then `x21`, `x_st` and `x2` are absent. When `delta` is strictly positive the two solutions are the current values of `x21` and `x2`, `x_st` is absent. The signal `stable` is *false* until the instant following the emission of the result in (`R`).

The process model `eqSolve` is the global solver: `a`, `b` and `c` are declared to be synchronous.

```
process eqSolve = { real epsilon; }
  ( ? real a, b, c; ! boolean x_st; real x1, x2; )
  (| a ^= b ^= c ^= when stable
   | (x_st_1, x11) := FirstDegree ( b when (a=0.0), c when (a=0.0))
   | (x_st_2, x21, x2, stable) :=
        SecondDegree{epsilon}(a when (a/=0.0), b when (a/=0.0), c when (a/=0.0))
   | x1 := x11 default x21
   | x_st := x_st_2 default x_st_1 |)
  where ... end
```

When the value of `a` is 0, `FirstDegree` input signals are present (and then `FirstDegree` is "activated"), `a`, `b` and `c` are not "transmitted" to `SecondDegree` which remains inactive. Conversely when `a` is not 0, `SecondDegree` is activated and `FirstDegree` is not. The results of the activated modes are merged to generate `x1`, `x2` and `x_st`.

4 Formal context for code generation

The equational nature of the SIGNAL language is a fundamental characteristic that makes it possible to consider the compilation of a process as a composition of endomorphisms over SIGNAL processes. We have given in Section 2.6 a few properties allowing to rewrite processes with rules such as commutativity and associativity of parallel composition. More generally, until the very final steps, the compilation process may be seen as a sequence of morphisms rewriting SIGNAL processes to SIGNAL processes. The final steps (C code generation for instance) are simple morphisms over the transformed SIGNAL processes.

In some way, because SIGNAL processes are systems of equations, compiling SIGNAL processes amounts to "solving" these equations. Among relevant questions arising when producing executable code, for instance, there are the following ones:

– Is the program deadlock free?
– Has it a deterministic execution?
– If so, can it be statically scheduled ?

To answer these questions, two basic tools are used in the compilation process. The first one is the modeling of the synchronization relation in $\mathscr{F}_3$ by polynomials with coefficients in the finite field $\mathbb{Z}/3\mathbb{Z}$ of integers modulo 3 [19]; the POLYCHRONY SIGNAL compiler manipulates a Boolean hierarchy instead of this field. The second one is the directed graph associated to data dependencies and explicit precedences. The synchronization and precedence relations are represented in a directed labeled graph structure called the *Data Control Graph* (DCG); it is composed of a *Clock Hierarchy* (CH, Section 4.3.1) and a *Conditioned Precedence Graph* (CPG, Section 4.4). A *node* of this CPG is an elementary process or, in a hierarchical organization, a composite process containing its own DCG.
This section introduces SIGNAL features used to state properties related to the Data Control Graph. Principles and algorithms applied to information carried out by this DCG are presented.

4.1 Endochronous acyclic process

When considering embedded systems specified in SIGNAL, the purpose of code generation is to synthesize an executable program that is able to deterministically compute the value of all signals defined in a process. Because these signals are timed by symbolic synchronization relation, one first needs to define a function from these relations to compute the clock of each signal. We say that a process is *endochronous* when there is a unique (deterministic) way to compute the clocks of its signals. Note that, for simulation purpose, one may wish to generate code for non deterministic processes (for example, partially defined ones) or even for processes that may contain deadlocks. Endochrony is a crucial property for processes to be executable: an endochronous process is a function over pure flows. It means that the pure flows resulting from its execution on an asynchronous architecture do not depend on propagation delays or operator latencies. It results from this property that **a network of endochronous processes is a Kahn Process Network (KPN) and thus is a function over pure flows**. But it is not necessarily a function over synchronized flows: synchronizations related to the number of #'s are lost because #'s are ignored.

Whereas synchronization relation determines which signals need to be computed at a given time, precedence relation tell us in which order these signals have to be

computed: `x` *precedes* `y` *at* `c`, represented as $c : x \rightarrow y$, means that, for all instants in the pure time domain of the clock signal c, the computation of y cannot be performed before the value of x is known. Therefore $c : x \rightarrow y$ is equivalent to $c \hat{*} x \hat{*} y : x \rightarrow y$. Hence, we say that $c : x \rightarrow y$ is in *normalized form* if and only if $c \hat{*} x \hat{*} y = c$. The *reduced form* $x \rightarrow y$ denotes the normalized form $x \hat{*} y : x \rightarrow y$.

An immediate cycle in the conditioned precedence graph denotes a deadlock. Figure 1 presents the main sources of deadlocks. Figure 1-a is a classical computation cycle. Figure 1-b is a "schizophrenic" cycle between clock and signal: to know if `zx` is present one must know if its value is 0, but to pick up its value it is required to know if it is present. Figure 1-c is a cyclic dependence due to the free definition of both `x1` and `x2` when neither `a` nor `b` are present; nevertheless if the clock of `x1` is `a ^+ b` then there is no deadlock during execution (either `a` is present and `x1 ^- a` is null or `b` is present and `x2 ^- b` is null).

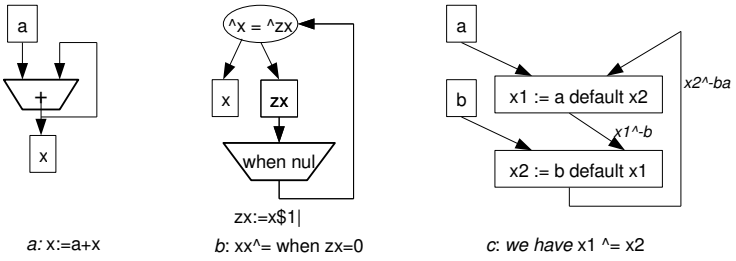


Fig. 1 Paradigms of deadlocks

4.2 SIGNAL *graph expressions*

The SIGNAL term `a --> b when c` is an elementary process in the SIGNAL syntax. It denotes the precedence relation $[c] : a \rightarrow b$. This constraint is usually implicit and related to data dependencies (for instance in `x := a+b` the constraints $a \rightarrow x$ and $b \rightarrow x$ hold), and clock/signal dependencies (the value of a signal x cannot be computed before $\hat{x}$, the clock of x, hence the implicit precedence relation $\hat{x} \rightarrow x$). Precedences can be freely added by the programmer. They can be computed by the compiler and made available in specifications associated with processes (Section 4.6).

Derived expressions

`a --> b` is a simplified term that means `a --> b when (a^*b)`.
Local precedence constraints can be combined as in `{b,c} --> {x_st,x}` meaning that for all pairs (u in `{b,c}`, v in `{x_st,x}`), u `-->` v holds.

The SIGNAL term `ll::P` is a derived process expression formally defined with SIGNAL primitive features. For the purpose of this chapter, it is enough to know that `ll::P` associates the *label* `ll` to the process `P`. A label `ll` is a special *event* signal that "activates" `P`. It may occur in synchronization and precedence constraints as any other signal, with a specific meaning for precedence: if a process `P` contains `ll1::P1` and `ll2::P2` then `ll1 -->ll2` in `P` means that every node in `P1` precedes all nodes of `P2`, while for a signal `x`, `x -->ll2` (resp. `ll1 --> x`) means that `x` precedes (resp. is preceded by) all nodes of `P2` (`P1`).

4.3 Synchronization relation

Table 1 gives the synchronization relation associated with a SIGNAL expression `P`, as a SIGNAL expression $\mathscr{C}$(P) (column 2), and as a time domain constraint $\mathscr{T}$(P) (column 3). The time domain constraints have to be satisfied by all executions of `P`. The synchronization relation associated with derived expressions (and normalizable ones) are deduced from this table. In this table "`[E]`" stands for "`when E`", "`^0`" is the "never present" clock, the pure time domain of a signal `x` is noted "$dom(x)$". Other notations are standard ones and SIGNAL notations.

construct P	clocks: $\mathscr{C}$(P)	pure time domains $\mathscr{T}$(P)
`boolean b`	`[b]^+[not b] ^= b | [b]^*[not b] ^= ^0`	$\mathscr{T}(\mathscr{C}(\mathrm{P}))$
`event b`	`[b] ^= b | [not b] ^= ^0`	$\mathscr{T}(\mathscr{C}(\mathrm{P}))$
`y := f(x1,...,xn)`	`y ^= x1^= ... ^= xn`	$\mathscr{T}(\mathscr{C}(\mathrm{P}))$
`y := x $1 init c`	`y ^= x`	$dom(y) = dom(x)$
`y := x when b`	`x^*[b] ^= y`	$dom(y) = dom(x) \cap dom([b])$
`y := x when not b`	`x^*[not b] ^= y`	$dom(y) = dom(x) \cap dom([\neg b])$
`z := x default y`	`x^+y ^= z`	$dom(z) = dom(x) \cup dom(y)$
$\mathrm{P}_1 \mid \mathrm{P}_2$	$\mathscr{C}(\mathrm{P}_1) \mid \mathscr{C}(\mathrm{P}_2)$	$\mathscr{T}(\mathscr{C}(\mathrm{P}_1)) \wedge \mathscr{T}(\mathscr{C}(\mathrm{P}_2))$
`P where x`	$\mathscr{C}$(P) `where x`	$\exists x \mathscr{T}(\mathscr{C}(\mathrm{P}))$

Table 1 Synchronization relation associated with a process

The transformation $\mathscr{C}$ defined in Table 1 satifies the following property making $\mathscr{C}$(P) a *process abstraction* of P:

$$(\mid \mathscr{C}(\mathrm{P}) \mid \mathrm{P} \mid) = \mathrm{P}$$

The clock of a Boolean signal `b` is partitioned into its exclusive sub-clocks $[b]$ and $[\neg b]$ which denote the instants at which the signal `b` is present and carries the values *true* and *false*, respectively, as shown in Figure 2.

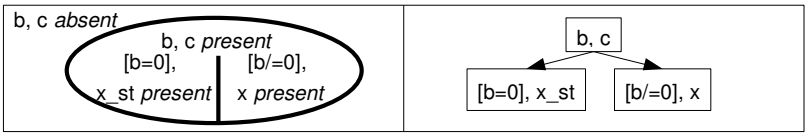


Fig. 2 Time subdomains and hierarchy for the FirstDegree process

4.3.1 Clock hierarchy

The synchronization relation provides the necessary information to determine how clocks can be computed. From the synchronization relation induced by a process, we build a so-called *clock hierarchy* [20]. A clock hierarchy is a relation $\hat{\searrow}$ (*dominates*) on the quotient set of signals by $\hat{=}$ (x and y are in the same class iff they are synchronous). Informally a class C dominates a class D (C *is higher than* D) or equivalently a class D *is dominated* by C (D *is lower than* C), written $D \hat{\swarrow} C$, if the clock of D is computed as a function of Boolean signals belonging to C and/or to classes recursively dominated by C.

To compute this relation, we consider a set V of free value Boolean signals (the *free variables* of the process); this set contains variables the definition of which cannot be rewritten using some implemented rewriting rule system: in the current POLYCHRONY version, input signals, delayed signals, results of most of the non-Boolean predicates are elements of this set V.

Depending on V, the construction of the relation $\hat{\searrow}$ is based on the following rules (for x a signal, C_x denotes its class; $\hat{\searrow}^*$ is the transitive closure of $\hat{\searrow}$):

1. If x_1 is a free variable such that the clock of x is defined by sampling that of x_1 ($\hat{x} = [x_1]$ or $\hat{x} = [\neg x_1]$) then $C_{x_1} \hat{\searrow} C_x$: the value of x_1 is required to compute the clock of x;
2. If $\hat{x} = f(\hat{x}_1 ..., \hat{x}_n)$, where f is a Boolean/event function, and there exists C such that $C \hat{\searrow}^* C_{x_i}$ for all x_i in $x_1, ..., x_n$ then $\hat{x}$ is written in canonical form $\hat{x} = cf(\hat{y}_1 ..., \hat{y}_m)$; $cf(\hat{y}_1 ..., \hat{y}_m)$ is either the null clock, or the clock of C, or is transitively dominated by C; in POLYCHRONY, BDDs are used to compute canonical forms;
3. If $\hat{x} = cf(\hat{y}_1 ..., \hat{y}_m)$ is a canonical form such that $m \geq 2$ and there exists C_z such that $C_z \hat{\searrow}^* C_{y_i}$ for all y_i in $y_1, ..., y_m$, then there exists a lowest class C that dominates those C_{y_i}, and $C \hat{\searrow} C_x$.

When the clock hierarchy has a unique highest class, like the classes of `Bx` in Figure 3-b, the process has a fastest rated clock and the status (presence/absence) of all signals is a pure flow function: this status depends neither on communication delays, nor on computing latencies.

The "clock calculus" provided by POLYCHRONY determines the clock hierarchy of a process. It has a triple purpose:

– it verifies that the process is *well-clocked*: synchronization relation of the process can be written as a set of acyclic definitions;
– it assigns to each clock a unique normalized definition when possible;
– it structures the control of the process according to its clock hierarchy.

4.3.2 "Endochronization"

A process that is not endochronous can be embedded in a *container* (Figure 3-c) such that the resulting process is endochronous. For instance, consider the clock hierarchy associated with the process `y := x when b`; it has two independent classes: the class of `^x` and the class of `^b` (Figure 3-a), the third one is defined by a clock product.

To get an endochronous process, one can introduce a new highest clock and two new input Boolean signals `Bx` and `Bb`, synchronous with that clock; the sampling of `Bx` and `Bb` defines respectively the clocks of `x` and `b` (Figure 3-b). This embedding can be made by the designer or heuristics can be applied to instrument the clock hierarchy with a default parameterization.

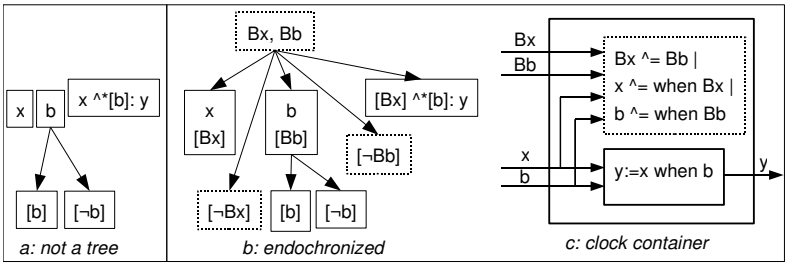


Fig. 3 Endochronization of `y := x when b`

Building such a container is useful not only to make a process endochronous but also to insure that the pure flow function associated with a KPN remains a synchronized flow function (i.e., synchronizations are preserved despite various communication delays).

4.4 Precedence relation

Table 2 gives the precedence relation associated with a SIGNAL expression `P`, as a SIGNAL expression $\mathscr{S}$(P) (column 2), and as a path algebra $\xrightarrow{\delta}$(P) (column 3). Notice that the delay equation (line 3) does not order the involved signals. The table is completed by relations coming from the clock hierarchy.

The transformation $\mathscr{S}$ defined in Table 2 satifies the following property making $\mathscr{S}$(P) a *process abstraction* of P:

$$(|\ \mathscr{S}(P)\ |\ P\ |) = P$$

construct P	precedence: $\mathscr{S}$(P)	path algebra: $\xrightarrow{\delta}$(P)
{a --> b when c}	{a --> b when c}	$[c] : \mathtt{a} \to \mathtt{b}$
y := f(x1,...,xn)	x1-->y |...| xn-->y	$\xrightarrow{\delta}(\mathscr{S}(\mathrm{P}))$
y := x $1 init c		
y:= x when b	x-->y	$\mathtt{x} \to \mathtt{y}$
z:= x default y	x-->z | y-->z when (^y^-^x)	$\xrightarrow{\delta}(\mathscr{S}(\mathrm{P}))$
P1 | P2	$\mathscr{S}(\mathrm{P1})\|\mathscr{S}(\mathrm{P2})$	$((\xrightarrow{\delta}(\mathrm{P1})) \cup (\xrightarrow{\delta}(\mathrm{P2})))^*$
P where x	$\mathscr{S}$(P) where x	$\xrightarrow{\delta}(\mathrm{P})$
clock construct P	**precedence: $\mathscr{S}$(P)**	**path algebra: $\xrightarrow{\delta}$(P)**
x, a signal	^x --> x	$\hat{\mathtt{x}} \to \mathtt{x}$
x ^= when b x ^= when not b	b --> ^x	$\mathtt{b} \to \hat{\mathtt{x}}$
y ^= cf(^x1,...,^xn)	^x1-->^y |...| ^xn-->^y	$\xrightarrow{\delta}(\mathscr{S}(\mathrm{P}))$

Table 2 Precedence relation associated with a process; the transitive closure $(P)^*$, is more precisely a path algebra presented in Section 4.4.1

4.4.1 Path Algebra $\xrightarrow{\delta}$

From basic precedence relation, a simple path algebra can be used to verify deadlock freeness, to refine precedence and to produce information for modularity. The path algebra is given by the following rules defining $\xrightarrow{\delta}^*$ for expressions $c : x \to y$ in normalized form:

- rule of series $\quad c : x \to y$ and $d : y \to z \Rightarrow c \hat{*} d : x \to z$
- rule of parallel $\quad \left.\begin{array}{l} c : x \to y \\ d : x \to y \end{array}\right\} \Rightarrow c \hat{+} d : x \to y$

A *pseudo cycle* in the precedence relation $\xrightarrow{\delta}$ associated with a process P is a sequence $c_1 : x_1 \to x_2$, $c_2 : x_2 \to x_3 \ldots c_{n-1} : x_{n-1} \to x_1$ in $\xrightarrow{\delta}^*$. P is deadlock free iff for all pseudo cycle $c_1 : x_1 \to x_2$, $c_2 : x_2 \to x_3 \ldots c_{n-1} : x_{n-1} \to x_1$ in its precedence relation, the product $c_1 \hat{*} c_2 \hat{*} \ldots \hat{*} c_{n-1}$ is null ($= \hat{0}$).

4.4.2 Precedence refinement

To generate sequential code for a (sub)process P it is usually necessary to add new dependencies $c_1 : x_1 \to x_2$ to P, getting so a new process PS which refines precedence of P. But it is wishable that if the composition of P with some process Q is deadlock free, then the composition of PS with the same process Q remains deadlock free; a refinement that satisfies this property is said to be *cycle consistent* by composition. In general, maximal cycle consistent refinement is not unique. The union of all maximal cycle consistent refinements is actually a preorder that can be computed using the path algebra [21]. The preorder associated with process First-Degree is shown in Figure 4-b (clocks are omitted): one can for instance compute b1 before c1 or conversely.

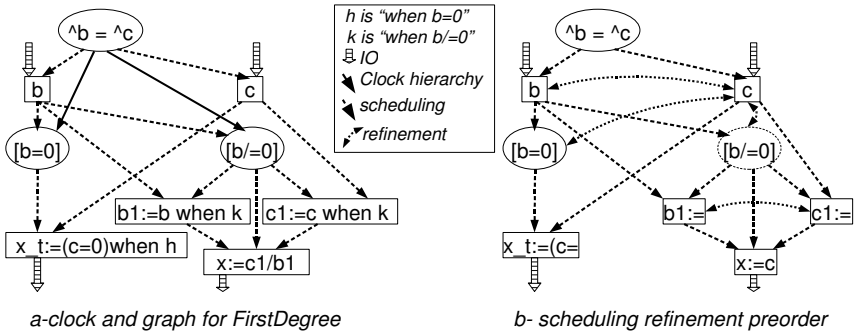


Fig. 4 FirstDegree process DCG and its precedence refinement preorder

4.5 Data Control Graph

The Data Control Graph, composed of a clock hierarchy and a Conditioned Precedence Graph as shown in Figure 4-a, not only associates with every clock node the set of signals that have this node as clock, but also produces a hierarchy of the CPG in which with every clock node is associated the subgraph of the calculus that must be processed when this clock is present (for instance computations of `b1`, `c1`, `x` are associated with the node `[b /= 0]` in Figure 4-a).

4.6 SIGNAL *process abstraction*

The interface of a process model `MP` can contain, in a process `P_abst`, a specific description of relations, typically, clock and precedence relations, applying on its inputs and outputs:

```
process MP(?I1, ...,Im; !O1,..., On) spec (|P_abst|) P_body
```

When the process body `P_body` of `MP` is a set of equations, the actual process associated with `MP` is not `P_body` but `(|P_abst|P_body|)`. Hence, `P_abst` is by construction an abstraction of `(|P_abst|P_body|)`.

When `MP` is an external process model, `P_abst` is assumed to be an abstraction of the process associated with `MP`. Supplementary properties are required to correctly use those external processes. Thus, a `process` is (can be) declared:

- *safe*: a *safe* process *P* is a single-state deterministic, endochronous automaton; such a process does not have occurrence of delay operator and cannot "call" external processes that are not safe; in practice the code generated to interact with *P* (to call *P*) can be freely replicated;

- *deterministic*: a *deterministic* process *P* is a deterministic, endochronous automaton (it is a pure flow function); such a process cannot "call" external processes that are not safe; in practice the code generated to interact with *P* (to step *P*) can be replicated only if the internal states of *P* are also replicated;
- *unsafe*: a process is unsafe by default.

As an example, the `FirstDegree` function (Example 3.1) `First Degree` is a safe process. Its synchronization and precedence relations are made explicit in its specification: all output signals depend on all input signals (`{b, c} --> {x, x_st}`), the signal `x_st` is synchronized with the clock at which `b` is zero, the signal `x` is present iff `b` is non-zero. The abstraction of its generated C code is then given by the `FirstDegree` SIGNAL external process presented below.

For a SIGNAL process `P` such an abstraction can be automatically computed by the POLYCHRONY SIGNAL compiler: it is the restriction to input/output signals (of `MP`) of the process `(|`$\mathscr{C}$`(P)|`$\mathscr{S}'$`(P)|)`, where $\mathscr{S}'$`(P)` is the precedence refinement of $\mathscr{S}$`(P)` taking into account the precedences introduced by code generation.

When the specifications are related to imported processes their source code may not be available (written in another language such as C++ or Esterel [22]). Legacy codes can be used in applications developed in SIGNAL, in which they are considered as external processes via specifications.

Besides, POLYCHRONY provides a translator from C code in SSA form to SIGNAL processes [23]. The translated process can be used to abstract the original C behavior. Each sequence of instructions in the SSA form is associated with a label. Each individual instruction is translated to an equation and each label is translated to a Boolean signal that guards its activation. As a result, the SIGNAL interpretation has an identical behavior as the original C program.

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

```
process FirstDegree =
      ( ? real b, c; ! boolean x_st; real x;)
      safe
      spec (| b --> c %precedence refinement%
            | {b, c} --> {x , x_st}
            | b  ^= c
            | x  ^= when (b/=0.0)
            | x_st ^= when (b=0.0) |)
      external "C" ;
```

Fig. 5 Legacy C code of the `FirstDegree` function and its SIGNAL abstraction

Syntactic sugar is made available in SIGNAL to easily handle usual functions. For instance the arithmetic C function `abs` used in `rac` can be declared as `function abs=(? real x; ! real s;)`; `function` means that the process is safe, its input and output signals are synchronous and every input precedes every output.

4.7 Clustering

The DCG (Section 4.5) with its CH (Section 4.3.1) ant its CPG (Section 4.4) is the basis of various techniques applied to structuring the generated code whilst preserving the semantic of the SIGNAL process thanks to formal properties of the SIGNAL language (Section 2.6). We give in this section a brief description of several important techniques that are illustrated in Section 5.

4.7.1 Data flow execution

The pure data flow code generation provides the maximal parallel execution. If we consider the `FirstDegree` example, we can see that some elementary equations are not pure flow functions, and then the result of their pure data flow execution is not deterministic. It is the case for `c1 := c when (b/=0.0)`: an occurrence of `c` being arrived and `b` not, `b` may be assumed absent. A simple solution to this problem is to consider only endochronous nodes as candidates for data flow execution. One can get this structure by duplicating the synchronization `b ^= c`. The resulting code for `FirstDegree`, in which each line is a pure flow function, is then:

```
(| b1 := b when (b/=0.0)
 | (| b ^= c | c1 := c when (b/=0.0) |)
 | x  := -(c1/b1)
 | (| b ^= c | x_st := (c/=0.0) when (b=0.0) |)
```

This example illustrates a general approach, using properties of the SIGNAL language (Section 2.6) to get and manage endochronous subprocesses as sources for future transformations. Building primitive endochronous nodes does not change the semantics of the initial process.

4.7.2 Data clustering

Data clustering is an operation that lets unchanged the semantics of a process. It consists in splitting processes on various criteria thanks to commutativity, associativity and other properties of process operators (Section 2.6). It is thus possible to isolate:

– the management of state variables (defined, directly or not, as delayed signals) in a “State variables” process (required synchronizations are associated with those variables),
– the management of local shared variables used instead of signal communications between subprocesses in “Local variables” blocks,
– the computation of specific data types towards multicore heterogeneous architectures.

4.7.3 Phylum

Atomic nodes can be defined w.r.t. the following criterion: when two nodes both depend on (are transitively preceded by) the same set of input variables, they can be both executed only after the inputs are achieved. This criterion defines an equivalence relation the class of which we name *phylums*: for a set of input variables `A` we name phylum of `A` the set of nodes that are preceded by `A` and only `A`. It results from this definitions that if a phylum `P1` is the phylum of `A1`, and `P2` that of `A2`, `P1` precedes `P2` iff `A1` $\subseteq$ `A2`.
As an example, the `FirstDegree` function (Example 4.7.1) might have four phylums: the phylum of the empty set of inputs which is empty, the phylum of `b` that contains `b1 := b when(b/=0.0)`, the phylum of `c` which is empty, and finally the phylum of {`b,c`} that contains the three remaining nodes. Due to its properties, an endochronous process can be splitted into a network of endochronous phylums that can be executed atomically (as function calls for instance).

4.7.4 From cluster to scheduling of actions

Using data clustering and phylums, a SIGNAL process can be transformed in an equivalent process the local communications of which are made invisible for a user context. This transformation is illustrated with the `FirstDegree` process; two phylums are created: `Phylum_B` is the phylum of `b` and `Phylum_BC` is the phylum of {`b,c`}; two labels `L_PH_B` and `L_PH_BC` are created to define the "activation clock" and the precedence related to these phylums. One can notice that this SIGNAL code is close to a low level imperative code.

```
process FirstDegree = ( ? real b, c; ! boolean x_st; real x; )
  (| b ^= c ^= b1 ^= bb ^= L_PH_B ^= L_PH_BC
   | L_PH_B :: Phylum_B(b)
   | L_PH_BC :: (x_st, x) := Phylum_BC(c)
   | L_PH_B --> L_PH_BC
   |) where shared real b1, bb;
         process Phylum_B = ( ? real b; )
            (| bb ::= b
             | b1 ::= b when (b/=0.0) |)
         process Phylum_BC = ( ? real c; ! boolean x_st; real x;)
            (| c1  := c when (b/=0.0)
             | x   := -(c1/b1)
             | x_st := (c/=0.0) when (bb=0.0) |) where real c1; end
  end
```

A *grey box* is an abstraction of such a clustered process: the grey box of a process contains a specification that describes synchronization and precedence over signals and labels. The phylums synchronized by these labels are declared as abstract processes. The grey box of a process `P` can be imported in another process `PP`. The global scheduling generated for `PP` includes the local scheduling of `P`. In the case of object-oriented code generation, this inclusion can be achieved by inheritance and redefinition of the scheduling, the methods associated with abstract processes remaining unchanged. The grey box associated with `FirstDegree` can be found below.

```
process FirstDegree = ( ? real b, c; ! boolean x_st; real x; ) %grey box%
  safe
  spec (| b ^= c ^= L_PH_B ^= L_PH_BC
        | L_PH_B :: Phylum_B(b)
        | L_PH_BC :: (x_st, x) := Phylum_BC(c)
        | b --> L_PH_B --> c --> L_PH_BC --> {x , x_st}
        | x  ^= when (b/=0.0)
        | x_st ^= when (b=0.0) |)
     where process Phylum_B = ( ? real b; ) external;
        process Phylum_BC = ( ? real c; ! boolean x_st; real x;) external;
     end
  external ;
```

4.8 Building containers

The construction of containers previously proposed to build endochronous processes (Section 4.3.2) is used to embed a given process in various execution contexts. The designer can for instance define input/output functions in some low level language and import their description in a SIGNAL process providing to the interfaced process an abstraction of the operating system; this is illustrated below for the `FirstDegree` process:

```
process EmbeddedFirstDegree = ( )
  (| (| L_SCAN :: (b,c) := scan() | b ^= c ^= L_SCAN |)
   | (x_st, x) := FirstDegree(b,c)
   | (| L_EA :: emitAlarm() | L_EA ^= x_st |)
   | (| L_PRINT :: print(x) | L_PRINT ^= x |)
   |) where real b, c, x; boolean x_st;
         process scan = ( ! real b, c; )
           spec (| b ^= c |)
         process emitAlarm()
         process print = ( ? real x )
         process FirstDegree = ( ? real b, c; ! boolean x_st; real x; )
            %description of FirstDegree%
      end
```

The statement `(b,c) := scan()`, synchronized to `L_SCAN`, delivers the new values of `b` and `c` each time it is activated. The statement `print(x)` prints the result when it is present. The statement `emitAlarm()` is synchronized with `x_st`, thus an alarm is emitted each time the equation has not a unique solution.

The construction of containers can be used to execute processes in various contexts, including resynchronization of asynchronous communications, thanks to the `var` operator (Section 2.5) or more generally to bounded fifos—that can be built in SIGNAL.

5 Code generation in POLYCHRONY toolset

Section 4 introduces the compilation of a SIGNAL process as a set of transformations. In this section, we describe how code can be generated, as final step of such

a sequence of transformations, following different schemes. When a process `P` is composed of interconnected endochronous subprocesses, free of clock constraints, it is a pure flow function (a KPN). One could then generate code for `P` following the Kahn semantics. Nevertheless, the execution of the generated code may deadlock. If this KPN process is also acyclic then deadlock cannot occur: the code generation functionalities of POLYCHRONY can be applied. The code is generated for different target languages (C, C++, Java) on different architectures, preserving various semantic properties (at least the defined pure flow function). However, it is possible to produce *reactive code* or *defensive code* when the graph is acyclic but there are remaining clock constraints. In these modes, all input configurations are accepted. For reactive code, inputs that satisfy the constraints are selected; for defensive code, alarms are emitted when a constraint is violated during the simulation.

5.1 Code generation principle

The code generation is based on formal transformations presented in the previous sections. It is strongly guided by the clock hierarchy resulting from the clock calculus to structure the target language program, and by the conditioned precedence graph not only to locally order elementary operations in sequences, but also to schedule component activations in a hierarchical target code. The code generation follows more or less the structure presented in Figure 6. The "step block" contains a step scheduler that drives the execution of the step component and updates the state variables (corresponding to delays). The step component may be hierarchically decomposed as a set of sub-components (clusters), scheduled by the step scheduler, and each sub-component has, in the same way, its own local step scheduler. The step block communicates with its environment through the IO container and it is controlled by a main program.

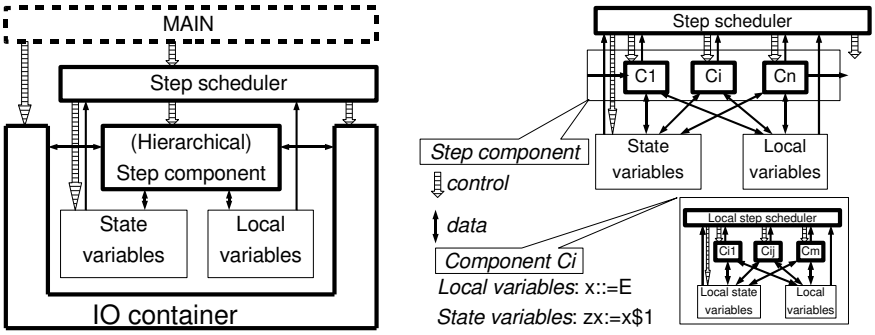


Fig. 6 Code generation general scheme

Target language code is generated in different files. For example, for C code generation, we have, for a process P, a main program `P_main.c`, a program body `P_body.c` (that contains the step block) and an input-output module `P_io.c` (the IO container). The *main* program calls the *initialization* function defined in the program body, then keeps calling the *step* function. The IO container defines the input and output communications of the program with the operating system.

Each component of the target code (except the main program, which is an explicit loop) may be seen as a SIGNAL process. Every such component, generated in C for instance, may be abtracted in SIGNAL for reuse in an embedding SIGNAL process. When target language is an object oriented language, then a class is generated for each component. This class can be specialized to fit new synchronizations resulting from embedding the original process in a new context process.

5.1.1 The step function

Once the program and its interface are initialized, the step function is responsible for 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 conditioned precedence graph produced by the front-end of the compiler. Various code generation schemes [21, 24, 25] are implemented in the POLYCHRONY toolset. They are detailed in the subsequent sections on the solver example:

- Global code generation
  - Sequential (Section 5.2)
  - Clustered with static scheduling (Section 5.3)
  - Clustered with dynamic scheduling (Section 5.4)
- Modular code generation
  - Monolithic (Section 5.5.1)
  - Clustered (Section 5.5.2)
- Distributed code generation (Section 5.6)

5.1.2 The IO container

If the process contains input and/or output signals (the designer did not build his or her IO container), the communication of the generated program with the execution environment is implemented in the IO container. 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 IO container (Figure 7) declares global functions for opening (`eqSolve_OpenIO`) and closing (`eqSolve_CloseIO`) all files, and for reading (`r_eqSolve_a`) and writing (`w_eqSolve_x1`) data along all input and output signals.

```
void eqSolve_OpenIO()
{ fra = fopen("Ra.dat","rt");
  if (!fra) {
  fprintf(stderr,
       "Can't open %s\n","Ra.dat");
   exit(1); }
  fwx1 = fopen("Wx1.dat","wt");
  if (!fwx1) {
    fprintf(stderr,
      "Can't open %s\n","Wx1.dat");
    exit(1); }
  /* ... idem for b, c, x2, x_st */}
```

```
void eqSolve_CloseIO()
{ fclose(fra);
  ...
  fclose(fwx1); }

int r_eqSolve_a(float *a)
{ return (fscanf(fra,"%f",a)!=EOF); }

void w_eqSolve_x1(float x1)
{ fprintf(fwx1,"%f ",x1);
  fprintf(fwx1,"\n"); fflush(fwx1); }
/* ... idem for b, c, x2, x_st */
```

Fig. 7 An extract of the C code generated for the solver: the IO container

The IO container is the place where the interface of generated code with an external visualization and simulation tool can be implemented. The POLYCHRONY toolset supports default communication functions for various operating systems and middlewares that can be modified or replaced by the user. The `r_xx_yy` functions return an error status that should be removed in embedded programs.

5.1.3 The main program

The *main* (see Figure 8) program initializes input/output files (`eqSolve_OpenIO`), state variables (`eqSolve_initialize`), and iterates call to the step function (`eqSolve_step`). In the case of simulation code, as in Figure 8, the infinite loop can be stopped if the step function returns error code 0 (meaning that input streams are empty) and the main program will close communication (`eqSolve_CloseIO`).

```
extern int main()
{ int code;
  eqSolve_OpenIO();                        /* input/output initializing */
  code = eqSolve_initialize();      /* initializing the state variables */
  while(code) code = eqSolve_step();                       /* the steps */
  eqSolve_CloseIO(); }                      /* input/output finalizing */
```

Fig. 8 Generated C code of the solver: the *main* program

5.2 Sequential code generation

This section describes the basic, sequential, inlining, code generation scheme that directly interprets the SIGNAL process obtained after clock hierarchization. This description is illustrated on the `eqSolve` process (Figure 9 and Figure 10). Figure 9 contains an extract of the clock hierarchy resulting from the clock calculus applied to the solver example. The code of the step block is structured according to the clock hierarchy as it can be observed in Figure 10. The precedence relation is the source of local deviations. In the `eqSolve_step` function, an original SIGNAL identifier `xxx` has a Boolean clock named `C_xxx`.

A few observations can be made on the step block that is generated for the `eqSolve` SIGNAL process (Figure 10). A first observation is that the master clock,

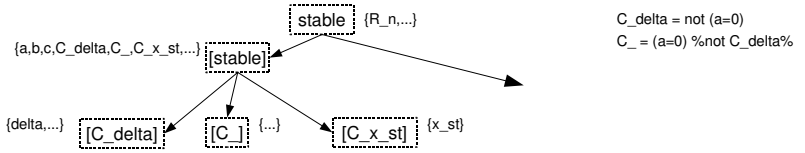


Fig. 9 Clock hierarchy of the solver process

which is the clock of the `stable` variable, ticks every time the step function is called.

Inputs are read as soon as their clock is evaluated and is *true* (see for example `r_eqSolve_a(&a)`, called when the signal `stable` is *true*). Outputs are sent as soon as they are evaluated (see for example `w_eqSolve_x_st(x_st)`, called when the signal `C_x_st` is *true*).

```
static float a, b, c;
static int x_st;
...
int eqSolve_initialize()
{ stable = 1;
  S_n = 0.0e0;
  next_R_n = 1.0;
  XZX_162 = 0.0e0;
  U = 0.0e0;
  eqSolve_step_initialize();
  return 1; }

void eqSolve_step_initialize()
{ C_ = 0;
  C_delta = 0;
  C_b1 = 0;
  C_x1_1 = 0;
  C__250 = 0; }

int eqSolve_step_finalize()
{
 stable = next_stable;
 C_x_st_1 = 0;
 C_231 = 0;
 eqSolve_step_initialize();
 return 1;
}
```

```
int eqSolve_step()
{
 if (stable) {
    if (!r_eqSolve_a(&a)) return 0;
    if (!r_eqSolve_b(&b)) return 0;
    if (!r_eqSolve_c(&c)) return 0;
    C_ = a == 0.0;
    C_delta = !(a == 0.0);
    if (C_delta) {
       delta = b * b - (4.0*a)*c;
       C_231 = delta < 0.0;
       C_x1_1 = delta == 0.0;
       C__250 = delta > 0.0;
       if (C_x1_1) x1_1 = -b/(2.0*a);}
    C_234 = (C_delta ? C_231 : 0);
    if (C_) {
       C_x_st_1 = b == 0.0;
       C_b1 = !(b == 0.0);
       if (C_x_st_1) x_st_1 = c != 0.0;}
    C_x_st_1_220 = (C_ ? C_x_st_1 : 0);
    C_x_st = C_x_st_1_220 || C_234;
    if (C_x_st) {
       if (C_234) x_st=1; else x_st=x_st_1;
       w_eqSolve_x_st(x_st); }
    }
  /* ... */
 eqSolve_step_finalize();
 return 1;}
```

Fig. 10 Generated C code of the solver: the step block

The state variables are updated at the end of the step (`eqSolve_step_finalize`). One can notice the tree structure of conditional if-then-else statements which directly translates the clock hierarchy. For instance, the computation of `x1_1` is executed only if `stable` is *true* and `a/=0` (`C_delta` is *true*) and `delta=0` (`C_x1_1` is *true*). One can also notice the precedence refinement as presented in 4.4.2: to generate sequential code, it is usually necessary to add serializations. To illustrate this, consider the abstraction, reduced to the precedence relations of the

solver (Figure 11-a). There is no precedence between the input signals. To generate the code, precedences are added from `a` to `b` and from `b` to `c`. This refinement is expressed in SIGNAL in the abstraction given in Figure 11-b. The same remarks applies for the local signals. So, the generated code is a specialization of the original process.

```
process eqSolve_ABSTRACT =
  ( ? real a, b, c;
    ! boolean x_st; real x1, x2; )
  spec (| {a,b,c} --> {x_st,x1,x2}
        | %clocks ... %
        |)
```

a-From the original process

```
spec (| (| a --> b | b --> c
         | x_st --> x2 | x2 --> x1
         |)
       | {a,b,c} --> {x_st,x1,x2}
       |  %clocks  ... %
       |)
```

b-Precedence refinement

Fig. 11 Abstractions (reduced to precedence relations) of the solver

Note that in this code generation scheme, the scheduling and the computations are merged.

5.3 Clustered code generation with static scheduling

The scheme presented here uses the result of clustering in phylums to generate code. This method is particularly relevant in code generation scenarios such as modular compilation and distribution.

Figure 12 displays the clusters obtained for the `eqSolve` process. Since there are three inputs a, b, c, the clustering in phylums can lead to, at most, $2^3 = 8$ clusters, plus one for state variables. Fortunately clustering is usually far from reaching worst combinatoric case. In the case of the solver, five clusters are non-empty. The clusters are subject to inter-cluster precedences that appear in the figure as solid arrows. Serializations, represented as dotted arrows, are added for static scheduling. The `Cluster_delays` is the "State variables" component (Figure 6), in charge of updating state variables. It is preceded by all other clusters.

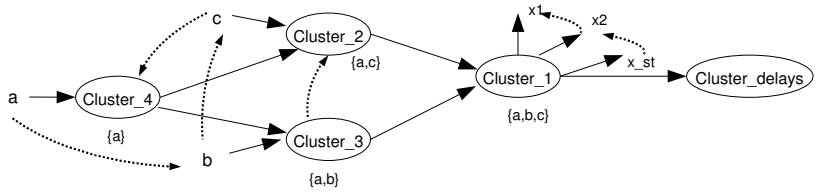


Fig. 12 The phylums of the solver

Figure 13 presents the code generated for this structuring of the solver into five phylums. The function `eqSolve_step` encodes a static scheduler for these clusters.

```
int eqSolve_step()
{
 if (stable) {
    if (!r_eqSolve_a(&a)) return 0;
    if (!r_eqSolve_b(&b)) return 0;
    if (!r_eqSolve_c(&c)) return 0;
 }
 eqSolve_Cluster_4();
 eqSolve_Cluster_3();
 if (stable) eqSolve_Cluster_2();
 eqSolve_Cluster_1();
 if (stable) if (C_x_st) w_eqSolve_x_st(x_st);
 if (C_x1_2) w_eqSolve_x2(x2);
 if (C_x1) w_eqSolve_x1(x1);
 eqSolve_Cluster_delays();
 return 1;
}
```

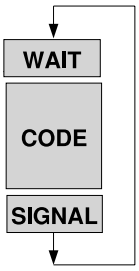
Fig. 13 Generated C code of the solver: statically scheduled clusters

In contrast to the previous code generation method, which globally relies on the clock hierarchy and locally relies (for each equivalence class of the clock hierarchy) on the conditioned precedence graph, clustering globally relies on the conditioned precedence graph and locally relies (for each cluster of the graph) on the clock hierarchy.

Note. In general any clustering is suitable with respect to some arbitrary criterion, provided that each cluster remains insensitive to communication delays or operator latencies. Monolithic clustering (one cluster for the whole process) minimizes scheduling overhead but has poor concurrency. The finest clustering (one cluster per signal) maximizes concurrency and unfortunately scheduling overhead. Heuristics are used in POLYCHRONY to limit the exploration cost.

5.4 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 of a cluster is encapsulated in a component implemented as a task. A component is structured as outlined in Figure 14-a: each component `Ti` has a semaphore `Si` used to manage the precedence relation between the components. 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)`. Moreover, one component is generated for each input-output function. The step scheduler (Figure 14-b) is implemented by a particular task `T0`: it starts execution by signaling all source tasks $j = 1,..N_0$ with `signal(S0j)` and ends by waiting on its sink tasks with `wait(S0)` statements. The semaphores and tasks are created in the `P_initialize` function of the generated code. When simulation code is generated, a `P_terminate` task is also added to kill all tasks of the application.




```
void * P_Ti() { /* task Ti */
  while (true) {
    pK_Isem_wait(Si);
    ...
    pK_Isem_wait(Si);
    P_Cluster_i();
    pK_Isem_signal(Si1);
    ...
    pK_Isem_signal(SiNi);
  }
}
```

a-A step component

```
void * P_step_Task()
{              /* Task T0 */
  while (true) {
  /*signal to the clusters
    without predecessors */
    pK_Isem_signal(S01);
    ...
    pK_Isem_signal(S0N0);
  /* wait the signal of the
     clusters without
     successors */
    pK_Isem_wait(S0);
    ...
    pK_Isem_wait(S0);
  }
}
```

b-The step scheduler

Fig. 14 Code template of a cluster task and of the step function

5.5 Modular code generation

Modular compilation consists of compiling a process, then exporting its model, and use it in another process. For the purpose of exporting and importing the model of a process whose code has been compiled separately, the POLYCHRONY SIGNAL compiler provides an annotation mechanism to associate a compiled process with a profile (Section 4.6).

This profile consists of an abstraction of the original process model consisting of the synchronization and precedence 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 process). The annotations also give the possibility to specify the language in which the process is compiled (C, C++, Java) since the function call conventions and variable binding may vary slightly from one language to another.

Starting from a SIGNAL process, modular compilation supports the code generation strategies, with or without clusters, outlined previously. Considering the specification of the solver, we detail possible compilation scenarios in which the sub-processes `FirstDegree` and `SecondDegree` are compiled separately. The process `SecondDegree` has been adapted to the context of modular compilation: it does not assume that the values of `a` are different from 0. The clock of the signal `stable` is the master clock of the `SecondDegree` process.

5.5.1 Sequential code generation for modular compilation

A naive compilation method consists in associating each of the sub-processes of the solver with a monolithic step function. Figure 15 gives the SIGNAL abstraction (black box abstraction) that is inferred—or could be user-provided—in order to use the `SecondDegree` process as external function.

```
process SecondDegree_ABSTRACT =
(? real a, b, c;
 ! boolean x_st; real x21, x2;
   boolean stable,C_x1_2,C_delta,C_x_st,C_x21;
)
spec (| (| stable --> {a,b,c}
      ... |)
       | (| stable ^= C_x1_2 ^= C_x21
       | ... |) |)
pragmas BlackBox "SecondDegree" end pragmas
external "C";
```

Fig. 15 Black box abstractions of the FirstDegree and SecondDegree process models

The interface of the process `SecondDegree` has been modified: it is necessary to export Boolean signals that are used to define clocks of output signals. Then the interface of `SecondDegree` is endochronous. The process `eqSolve_bb`, Figure 16, is the SIGNAL process of the solver in which the separately compiled processes `FirstDegree` (its abstraction and generated code can be found in Example 5) and `SecondDegree` are called.

```
process eqSolve_bb = {real epsilon}
     ( ? real a, b, c; ! boolean x_st; real x1, x2; )
  (| a ^= b ^= c ^= when stable
   | (x_st_1, x11) := FirstDegree_ABSTRACT(b when (a=0.0), c when (a=0.0))
   | (x_st_2, x21, x2, stable, C_x1_2, C_delta, C_x_st2, C_x21)
                := SecondDegree_ABSTRACT(a, b, c)
   | x1 := x11 default x21
   | x_st := when x_st_2 default x_st_1
   |) where ...
```

Fig. 16 Importing separately compiled processes in the specification of the solver

Unfortunately, the code from which `SecondDegree_ABSTRACT` is an abstraction is a sequential code. And it turns out that the added precedence relations, put in the context of process `eqSolve`, form a causality cycle that is reported by the compiler (Figure 17): the abstraction of `SecondDegree` exhibits the precedence `stable --> a`, the function call to `SecondDegree_ABSTRACT` implies that the input signal `a` precedes the output signal `stable`, hence the cycle. Code generation cannot proceed further.

```
process eqSolve_bb_CYC = ( )
(| (x_st_2, x21,x2, stable, C_x1_2, C_delta, C_x_st2, C_x21)
             := SecondDegree_ABSTRACT(a,b,c)
 |  stable --> a |)
```

Fig. 17 Trace of causality cycle in the specification of the solver

5.5.2 Clustered code generation for modular compilation

To avoid spurious causality cycles from being introduced, it is better suited to apply clustering code generation techniques. The profile of a clustered and separately compiled process (grey box abstraction), Figure 18, provides more information than a black box abstraction: it makes the clusters apparent and details the clock and precedence relations between them. In turn, this profile contains sufficient information to schedule the clusters, and hence, call them in the appropriate order dictated by the calling context.

```
process SecondDegree_ABSTRACT =
 ( ? real aa, bb, cc; ! boolean x_st; real x21, x2;
     boolean stable, C_x1_2, C_delta, C_x_st, C_x21; )
   pragmas
      GreyBox "SecondDegree"
   end pragmas
(| (| Tick := true
    | when Tick ^= stable ^= C_x1_2 ^= C_x21
    | when stable ^= aa ^= bb ^= cc ^= C_delta
    | when C_x1_2 ^= x2 | when C_delta ^= C_x_st
    | when C_x_st ^= x_st | when C_x21 ^= x21 |)
 | (| lab :: (x_st,x21,x2,C_x1_2,C_x_st,C_x21) := SecondDegree_Cluster_1()
    | lab ^= when Tick |)
 | (| lab_1 :: C_delta := SecondDegree_Cluster_2(aa)
    | lab_1 ^= when Tick |)
 | (| lab_2 :: SecondDegree_Cluster_3(bb) | lab_2 ^= when stable |)
 | (| lab_3 :: SecondDegree_Cluster_4(cc) | lab_3 ^= when stable |)
 | (| lab_4 :: stable := SecondDegree_Cluster_delays() | lab_4 ^= when Tick |)
 | (| cc --> lab | bb --> lab | aa --> lab | lab_1 --> lab_3 | lab_1 --> lab_2
    | lab_1 --> lab | aa --> lab_1 | lab_2 --> lab | bb --> lab_2
    | aa --> lab_2 | lab_3 --> lab | cc --> lab_3 | aa --> lab_3
    | lab_3 --> lab_4 | lab_2 --> lab_4 | lab_1 --> lab_4 | lab --> lab_4
    |) |) where %Declarations of the clusters% end;
```

Fig. 18 The grey box abstraction of the SecondDegree process model

The calling process schedules the generated clusters in the order best appropriate to its local context, as shown in Figure 19. The step associated with `FirstDegree` and the clusters of the separately compiled process `SecondDegree` are called in the very order dictated by scheduling constraints of the solver process, avoiding the introduction of any spurious cycle.

```
extern int eqSolve_gb_step()
{
 if (stable) {
   if (!r_eqSolve_gb_a(&a)) return 0;
   if (!r_eqSolve_gb_b(&b)) return 0;
   if (!r_eqSolve_gb_c(&c)) return 0;
   C_ = a == 0.0;
   if (C_) { FirstDegree_step(&FirstDegree1,b,c,&x_st_1,&x11);
             C_x11 = !(b == 0.0);  }
             C_x_st_1_227 = (C_ ? (b == 0.0) : 0); }
 SecondDegree_Cluster_2(&SecondDegree2,a,&C_delta);
 C_x11_236 = (C_ ? C_x11 : 0);
 if (stable) {
       SecondDegree_Cluster_3(&SecondDegree2,b);
       SecondDegree_Cluster_4(&SecondDegree2,c);  }
 SecondDegree_Cluster_1(&SecondDegree2,&x_st_2,&x21,&x2,
                        &C_x1_2,&C_x_st2,&C_x21);
  ...
 if (C_x1_2) w_eqSolve_gb_x2(x2);
 if (C_x1) { if (C_x11_236) x1 = x11; else x1 = x21; w_eqSolve_gb_x1(x1); }
 eqSolve_gb_step_finalize();
 return 1; }
```

Fig. 19 Generated C code for the first and second degree clusters steps

5.6 Distributed code generation

Distributed code generation in POLYCHRONY follows along the same principles as dynamically scheduled clustered code generation (Section 5.4). The final embedded application implemented on a distributed architecture can be represented by the Figure 20.

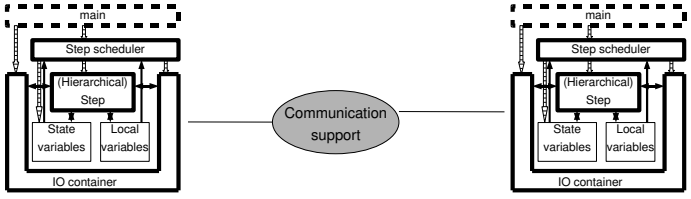


Fig. 20 Overview of a distributed code

While clustered code generation is automatic, distribution requires additional information provided by the user. Namely:

- A block-diagram and topological description of the target architecture.
- A mapping of software diagrams onto the target architecture blocks.

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

1. DCG computation for the main process structure (Section 4.5).
2. Booleanization: extension of event signals to Boolean signals according to the clock hierarchy.
3. Clustering of the main process according to the given target architecture and mapping (Section 5.6.1).
4. Endochronization of each cluster (Section 4.3.2); this is made by importing/exporting Boolean signals between clusters, according to the global DCG.
5. Addition of communication information (Section 5.6.2).
6. Individual compilation of all clusters.
7. Generation of the IO container (Section 5.6.3) and the global main program.

5.6.1 Topological annotations

Figure 21, the distribution methodology is applied on the example of Section 3. The user defines the mapping of the elementary components over the target architecture with a few pragmas. The pragma `RunOn` specifies the processors on which a set of components has to be located. For example, the expression `RunOn {e1} "1"` specifies that the component labelled with `e1` is mapped on the location `1` (`e1` is declared in the statement `e1 :: PROC1{}` as being the label of the component consisting of the subprocess `PROC1`).

```
process eqSolve =
  {real epsilon}
  ( ? real a, b ,c;
    ! boolean x_st;
      real x1, x2; )
  pragmas
   Topology {b,a} "1"
   Topology {c} "2"
   Topology {x1,x2} "2"
   Topology {x_st} "1"
   Target "MPI"
   RunOn {e1} "1"
   RunOn {e2} "2"
  end pragmas
  (|  e1 :: PROC1{}
   |  e2 :: PROC2{}
   |)
```

```
process PROC1 =
 ( ? real a, b, c, x11;
     boolean x_st_1;
   ! boolean stable;
     real x2, x1;
     boolean x_st; )
(| (x_st_2, x21, x2, stable)
      := SecondDegree{.}(...)
 | x1 := x11 default x21
 | x_st := x_st_2
        default x_st_1
 |)
 ;
```

```
process PROC2 =
 ( ? boolean stable;
     real b, c;
   ! real x11;
     boolean x_st_1;
     )
(| a ^= b ^= c
    ^= when stable
 | (x_st_1, x11)
    := FirstDegree(...)
 |)
```

Fig. 21 Functional clustering of the solver

The pragma `Topology` associates the input and output signals of the process with a location. For example, the pragma `Topology {a,b} "1"` tells that the input signals `a` and `b` must be read on location `1`. The pragma `Target` specifies the API used to generate code that implements communication: for instance, `Target "MPI"` tells that the MPI library is used for that purpose.

5.6.2 Communication annotations

Figure 22 displays the process transformation resulting of the functionClustering requested by the user. A few signals have been added to the interface of each com-

ponent: signals and clocks produced on one part of the system and used on the other one have to be communicated.

```
process eqSolve_EXTRACT_1_TRA=
 ( ? boolean x_st_1;
     real x11, c, b, a;
     boolean C_b1, C_x_st_1_490,
             C_, C_x_st_1;
   ! boolean x_st;
     real x1, x2;
     boolean stable;
  )
pragmas
   RunOn "1"
   Environment {c} "1"
   Environment {b} "3"
   Environment {a} "5"
   Environment {x_st} "7"
   Environment {x1} "8"
   Environment {x2} "9"
   Sending {stable} "10"
     "eqSolve_EXTRACT_2"
   Receiving {x_st_1} "11"
     "eqSolve_EXTRACT_2"
   Receiving {x11} "12"
     "eqSolve_EXTRACT_2"
   Receiving {C_} "13"
     "eqSolve_EXTRACT_2"
   Receiving {C_x_st_1} "14"
     "eqSolve_EXTRACT_2"
   Receiving {C_x_st_1_490} "15"
     "eqSolve_EXTRACT_2"
   Receiving {C_b1} "16"
     "eqSolve_EXTRACT_2"
end pragmas;
...
```

```
process eqSolve_EXTRACT_2_TRA=
  ( ? real c, b, a;
      boolean stable;
    ! boolean x_st_1;
      real x11;
      boolean C_, C_x_st_1,
              C_x_st_1_490, C_b1;
  )
pragmas
   RunOn "2"
   Environment {c} "2"
   Environment {b} "4"
   Environment {a} "6"
   Receiving {stable} "10"
    "eqSolve_EXTRACT_1"
   Sending {x_st_1} "11"
    "eqSolve_EXTRACT_1"
   Sending {x11} "12"
    "eqSolve_EXTRACT_1"
   Sending {C_} "13"
    "eqSolve_EXTRACT_1"
   Sending {C_x_st_1} "14"
    "eqSolve_EXTRACT_1"
   Sending {C_x_st_1_490} "15"
    "eqSolve_EXTRACT_1"
   Sending {C_b1} "16"
    "eqSolve_EXTRACT_1"
end pragmas
```

Fig. 22 The extracted subgraphs after the distribution

Required communications are automatically added using additional pragmas. The pragma `Environment` associates an input or output signal with the location of a communication channel. For instance, `Environment {c} "1"` means that signal `c` is communicated along channel `1`. The pragma `Receiving` associates an input signal with a channel location and its sender process. To send the signal `x11` from process `eqSolve_EXTRACT_1_TRA` along channel `11`, the following pragma is written: `Receiving {x11} "11" "eqSolve_EXTRACT_2"`. Similarly, the pragma `Sending` associates an output signal with a channel location and its receiving processes. For example, the output signal `stable` of process `eqSolve_EXTRACT_1_TRA` is sent along channel `10` to process `eqSolve_EXTRACT_2` by: `Sending {stable} "10" "eqSolve_EXTRACT_2"`.

5.6.3 IO Code generation

Multi-threaded, dynamically scheduled, code generation, as described in Section 5.4, is applied on the process resulting from the transformations performed for au-

tomated distribution. The information carried by the pragmas `Environment`, `Receiving` and `Sending` is used to generate communications. Figure 23 gives as illustration the code that implements communications of the signal `C_` (see Figure 22), using the MPI library, between the sender `eqSolve_EXTRACT_2` and the receiver `eqSolve_EXTRACT_1`.

From the file : eqSolve_EXTRACT_1_io.c

```
int r_eqSolve_EXTRACT_1_C_(int *C_) {
    MPI_Recv(C_,                    /* name */
             1,MPI_INT,                 /* type */
             eqSolve_EXTRACT_2, /* received from */
             13,                        /* the logical tag of the receiver */
             MPI_COMM_WORLD,            /* MPI specific parameter */
             MPI_STATUS_IGNORE);        /* MPI specific parameter */
    return 1;
}
```

From the file : eqSolve_EXTRACT_2_io.c

```
void w_eqSolve_EXTRACT_2_C_(int C_) {
    MPI_Send(&C_,                   /* name */
             1,MPI_INT,                 /* type */
             eqSolve_EXTRACT_1, /* sent to */
             13,                        /* the logical tag of the sender */
             MPI_COMM_WORLD);           /* MPI specific parameter */
}
```

Fig. 23 Example of communications

6 Conclusion

The POLYCHRONY workbench is an integrated development environment and technology demonstrator consisting of a compiler (set of services for, e.g., program transformations, optimizations, formal verification, abstraction, separate compilation, mapping, code generation, simulation, temporal profiling, etc.), a visual editor and a model checker. It provides a unified model-driven environment to perform embedded system design exploration by using top-down and bottom-up design methodologies formally supported by design model transformations from specification to implementation and from synchrony to asynchrony.

In order to bring the synchronous multi-clock technology in the context of model-driven environments, a metamodel of SIGNAL has been defined and an Eclipse plugin for POLYCHRONY is being integrated in the open-source platforms TopCased from Airbus [26] and OpenEmbeDD [27]. The POLYCHRONY workbench is now freely distributed [6].

In parallel with the POLYCHRONY academic set of tools, an industrial implementation of the SIGNAL language, called SILDEX, was developed by the TNI company, now included in Geensys. This commercial toolset, which is now called RT-BUILDER, is supplied by Geensys [28].

POLYCHRONY supports the polychronous data flow specification language SIGNAL. It is being extended by plugins to capture within the workbench specific modules written in usual software programming languages suchs as SystemC or Java. It provides a formal framework:

1. to validate a design at different levels,
2. to refine descriptions in a top-down approach,
3. to abstract properties needed for black-box composition,
4. to assemble predefined components (bottom-up with COTS).

To reach these objectives, POLYCHRONY offers services for modeling application programs and architectures starting from high-level and heterogeneous input notations and formalisms. These models are imported in POLYCHRONY using the data flow notation SIGNAL. POLYCHRONY operates these models by performing global transformations and optimizations on them (hierarchization of control, desynchronization protocol synthesis, separate compilation, clustering, abstraction) in order to deploy them on mission specific target architectures.

In this chapter, we meant to illustrate the application of a general principle: that of correct-by-construction design of systems, from the early stages of the design to the code generation phases on a given architecture. This is obtained by means of formally defined transformations, based on the mathematical polychrony model of computation, that may be expressed as source-to-source program transformations. This has several beneficial consequences for practical usability. In particular, scenarios of transformations can be fully controlled by an application designer. Among possible scenarios, a designer will have to his or her disposal predefined ones allowing for instance simulation of the application following different options, or safe code generation. This transformation-based mechanism makes it possible to formally validate the final result of a compilation. Moreover, it provides a suitable level of consideration for traceability purpose.

Source-to-source transformation of programs is used also for temporal analysis of SIGNAL processes on their implementation platform [29]. Basically, it consists of formal transformation of a process into another SIGNAL process that corresponds to a so-called *temporal interpretation* of the initial process. The temporal interpretation is characterized by quantitative characteristics of the implementation architecture. The new process can serve as an *observer* of the initial one.

Here, we focused more particularly on formal context for code generation (Section 4) and on code generation strategies available in POLYCHRONY (Section 5) by considering a SIGNAL process solving second degree equations (Section 3). While mathematically simple, this process exhibits non-trivial modes, synchronization relation, precedence relation, which we analyzed and transformed to illustrate several usage scenarios and for which we applied the following code generation strategies:

- Sequential code generation, Section 5.2, consists of producing a single step function for a complete SIGNAL process.
- Clustered code generation with static scheduling, Section 5.3, consists of partitioning the generated code into one cluster per set of input signals. The step function is a static scheduler of these clusters.

- Clustered code generation with dynamic scheduling, Section 5.4, consists of a dynamic scheduling of a set of clusters.
- Distributed code generation, Section 5.6, consists of physically partitioning a process across several locations and of installing point to point communications between them.
- Sequential code generation for separate compilation, Section 5.5.1, consists of associating the sequential generated code of a process with a profile describing its synthesized synchronization and precedence relations. The calling context of the process is passed to it as parameter.
- Clustered code generation for separate compilation, Section 5.5.2, consists of associating the clustered generated code of a process with a profile describing the synchronization and precedence relations of and between its clusters. The scheduler of the process is generated in each call context.

These code generation strategies are based on formal operations, such as abstractions, that make possible separate compilation, code substitutability and reuse of legacy code. The multiplicity of these strategies is a demonstration of the flexibility of the compilation and code generation tools provided in POLYCHRONY. It is also an indicator for the possibility offered to software developers to create new generators using the open-source version of POLYCHRONY. Such new generators will be needed for developing new execution schemes or to adapt the current ones in enlarged contexts providing for example a design-by-contract methodology [30].

References

1. S. K. Shukla, J.-P. Talpin, S. A. Edwards, and R. K. Gupta. High Level Modeling and Validation Methodologies for Embedded Systems: Bridging the Productivity Gap. In *VLSI Design 2003*, 9–14.
2. M. Crane and J. Dingel. UML vs. classical vs. rhapsody statecharts: not all models are created equal. In *Software and Systems Modeling*, 6(4):415–435, December 2007.
3. A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. In *Proceedings of the IEEE*, 91(1):64–83, January 2003.
4. P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for System Design. In *Journal for Circuits, Systems and Computers*, 12(3):261–304, April 2003.
5. L. Besnard, T. Gautier, and Paul Le Guernic. SIGNAL *V4-Inria Version: Reference manual*. http://www.irisa.fr/espresso/Polychrony.
6. The Polychrony platform `http://www.irisa.fr/espresso/Polychrony`.
7. M. Le Borgne, H. Marchand, E. Rutten, and M. Samaan. Formal verification of programs specified with Signal: application to a power transformer station controller. In *Science of Computer Programming*, 41:85–104, 2001.
8. M. Kerbœuf, D. Nowak, and J.-P. Talpin. Specification and Verification of a Steam-Boiler with Signal-Coq. In *Theorem Proving in Higher Order Logics (TPHOLs'2000)*, Lecture Notes in Computer Science, Springer, 2000.
9. T. Grandpierre and Y. Sorel. From algorithm and architecture specifications to automatic generation of distributed real-time executives: a seamless flow of graphs transformations. In *Formal Methods and Models for Codesign Conference*, Mont-Saint-Michel, France, June 2003.

10. P. Le Guernic. SIGNAL : Description algébrique des flots de signaux. In *Architecture des machines et systèmes informatiques*, 243–252. Hommes et Techniques, November 1982.
11. P. Le Guernic and A. Benveniste. *Real-time, synchronous, data-flow programming: the language* SIGNAL *and its mathematical semantics*. Technical Report 533 (revised version: 620), INRIA, June 1986.
12. P. Le Guernic and T. Gautier. Data-flow to von Neumann: the SIGNAL approach. In J. L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*, 413–438, 1991.
13. A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. In *Science of Computer Programming*, 16:103–149, 1991.
14. P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. In *Proceedings of the IEEE*, 79(9):1321–1336, Sep. 1991.
15. A. Gamatié. *Designing Embedded Systems with the SIGNAL Programming Language*. Springer, 2009.
16. G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing 74*, 471–475. North-Holland, 1974.
17. Arvind and K.P. Gostelow. *Some Relationships between Asynchronous Interpreters of a Dataflow Language*. North-Holland, 1978.
18. J. B. Dennis, J. B. Fossen, and J. P. Linderman. Data flow schemas. In A. Ershov and V. A. Nepomniaschy, editors, *International Symposium on Theoretical Programming*, 187–216. Lecture Notes in Computer Science, 5, Springer-Verlag, 1974.
19. M. Le Borgne. Dynamical systems over Galois fields: Applications to DES and to the Signal Language, In *Lecture Notes of the Belgian-French-Netherlands Summer School on Discrete Event Systems*, Spa, Belgium, June 1993.
20. T. Amagbegnon, L. Besnard, and P. Le Guernic. *Arborescent Canonical Form of Boolean Expressions*. Inria report n. 2290, 1994.
21. O. Maffeïs and P. 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. Lecture Notes in Computer Science vol. 863, Springer-Verlag, 547–566.
22. D. Potop-Butucaru, S.E. Edwards, and G. Berry. *Compiling Esterel*. Springer, 2007.
23. L. Besnard, T. Gautier, M. Moy, J.-P. Talpin, K. Johnson, and F. Maraninchi. Automatic translation of C/C++ parallel code into synchronous formalism using an SSA intermediate form, In *Ninth International Workshop on Automated Verification of Critical Systems (AVOCS'09)*, 2009, L. O'Reilly and M. Roggenbach, editors.
24. T. Gautier and P. Le Guernic. Code generation in the SACRES project. In *Towards System Safety, Proceedings of the Safety-critical Systems Symposium, SSS'99*, Huntingdon, UK, Springer, 1999, 127–149.
25. P. Aubry, P. Le Guernic, and S. 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.
26. The Topcased platform `http://www.topcased.org`.
27. The OpenEmbeDD platform `http://www.openembedd.org`.
28. Geensys' RT-Builder `http://www.geensys.com/?Outils/RTBuilder`.
29. A. Kountouris and P. Le Guernic. Profiling of SIGNAL programs and its application in the timing evaluation of design implementations. In *Proceedings of the IEE Colloq. on HW-SW Cosynthesis for Reconfigurable Systems*, pp. 6/1–6/9, Bristol, UK, February 1996. HP Labs.
30. Y. Glouche, T. Gautier, P. Le Guernic, and J.-P. Talpin. A Module Language for Typing SIGNAL programs by Contracts, In this book.