Polychronous Design of Real-Time Applications with Signal

Thierry Gautier, Paul Le Guernic, Jean-Pierre Talpin

Polychronous Design of Real-Time Applications with Signal

Thierry Gautier, Paul Le Guernic and Jean-Pierre Talpin [*]

June 30, 2008

Abstract. This paper provides an introduction to the synchronous, multi-clocked, data-flow specification language Signal. The main operators are described and their use is illustrated through a few simple examples. Basic techniques for compiling Signal programs are outlined.

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*. 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. Even though these design languages benefit from decades of programming practice, they still give rise to some diverging interpretations of their semantics.

[*]INRIA Rennes – Bretagne Atlantique, Campus de Beaulieu, 35042 Rennes Cedex, France. {*thierry.gautier, paul.le_guernic, jean-pierre.talpin*}*@irisa.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 a distribution of purely sequential programs and to implement semantics-preserving transformations and high-level optimizations such as hierarchization (sequentialization) or desynchronization (protocol synthesis).

In this aim, system design based on the so-called "synchronous hypothesis" has focused the attention of many academic and industrial actors. The synchronous paradigm consists of abstracting the non-functional implementation details of a system and lets one benefit from a focused reasoning on the logics behind the instants at which the system functionalities should be secured.

With this point of view, synchronous design models and languages provide intuitive models for embedded systems [7]. This affinity explains the ease of generating systems and architectures and verify their functionalities using compilers and related tools that implement this approach.

In the relational mathematical model behind the design language SIGNAL, the supportive data-flow notation of the integrated development environment POLYCHRONY, this affinity goes beyond the domain of purely sequential systems and synchronous circuits and embraces the context of complex architectures consisting of synchronous circuits and desynchronization protocols: globally asynchronous and locally synchronous architectures (GALS).

This unique feature is obtained thanks to the fundamental notion of *polychrony* [31]: the capability to describe systems in which components obey to multiple clock rates. 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 synthesis and deployment (functions, automata).

The notion of polychrony goes beyond the usual scope of a programming language, allowing for specifications and properties to be described. As a result, the SIGNAL design methodology draws 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.

The inherent flexibility of the abstract notion of signal handled in SIGNAL invites and favors the design of correct-by-construction systems by means of well-defined model transformations that preserve the intended semantics and stated properties of the architecture under design.

Synchronous languages rely on the synchronous hypothesis, which lets computations and behaviors be divided into a discrete sequence of *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.

In this article, we consider the Signal language, which is based on the polychronous semantic model [31] and its associated toolset Polychrony to design embedded real-time applications.

Outline. We present the main operators of the Signal language in Section 2, discussing first the principles of synchronized data-flow. We describe a few simple examples allowing to illustrate constraint programming with Signal and the specific feature of oversampling. Then in Section 3, we present the basic tools for compilation: clock calculus and graph calculus. Sequential code generation is illustrated and notions for partitioning programs toward separate compilation or distributed code generation are introduced. Finally, a technique for temporal analysis of Signal programs is briefly presented. Concluding remarks in Section 4 refer to the Polychrony workbench.

Historical notes. First of all, before introducing the language, we draw a few steps of its "history". The first studies for a new language started in INRIA-Rennes in 1981 in a project for the design of software for signal processing machines (the application domain was later widened). Le Guernic and Benveniste were in charge of the design of the language, with Gautier. The whole project was a cooperation between teams from INRIA-Rennes and INRIA-Rocquencourt, and CNET (French telecommunication organism). The first paper on Signal, viewed as an algebraic description of networks of flows, was published by Le Guernic in 1982 [27]. It was recognized later that several teams in France worked in parallel with similar ideas: this gave rise to the so-called "synchronous school", around the synchronous languages Esterel, Lustre and Signal, with many contacts and fruitful scientific exchanges. The first complete description of the Signal language (version 1) was provided by Gautier in his PhD [18]. The encoding of clocks using $\mathbb{Z}/_{3\mathbb{Z}}$ was proposed by Le Guernic and Benveniste in 1986 [28]. A full compiler, including clock calculus (with hierarchies of Boolean clocks), was described by Besnard [11]. The clock calculus was later improved by Amagbegnon [1], who defined arborescent canonical forms. The semantics of the language has been described using different models: operational semantics [9], denotational semantics [10], trace semantics [29, 39] (used in the current reference manual for Signal V4 [12]), tagged model [31] (now considered as a reference paper for the polychronous model). Nowak [36] proposed a co-inductice semantics that was used for modeling Signal in the proof assistant Coq. A number of PhD's have been devoted to different aspects of Signal implementation, many of these works were conducted in the context of cooperative projects including European ones such as Synchron, Syrf, Sacres, SafeAir, etc. To mention only some of them that are in phase with the mainstream of the current Signal version: Chéron [13] proposed optimization methods; Le Goff [26] defined clustering models for Signal programs; Maffeïs [33] formalized the required notions for abstraction and separate compilation; Aubry [3] described distributed implementation (the PhD's mentioned here are written in French but corresponding articles written in English may be found on the POLYCHRONY site). Many other studies, not detailed here, concerned extensions of SIGNAL, translations to or from SIGNAL, specific applications, etc. We also mention the definition of an affine clock calculus for affine clocks by Smarandache [39] and the polychronous modeling of real-time executive services of the ARINC avionic standard by Gamatié [17]. Belhadj [4], Kountouris [23] and Le Lann [32], also with Wolinski, used SIGNAL for hardware description and synthesis ([23] describes also a method for temporal interpretation of SIGNAL programs). Dutertre [16], Le Borgne [25] and Marchand [34] developed the theory of polynomial dynamical systems on $\mathbb{Z}/_{3\mathbb{Z}}$, implemented it in the SIGALI tool and applied it for verification and controller synthesis on SIGNAL programs. Le Guernic and others had characterized specific classes of polychronous programs such as endochronous ones. In [6], Benveniste, Caillaud and Le Guernic analyzed the links between synchrony and asynchrony and introduced the property of isochrony in the context of synchronous transition systems. In [31], Le Guernic, Talpin and Le Lann expressed the notion of endo-isochrony in the tagged model of polychrony. A property of weak endochrony was described by Potop-Butucaru, Caillaud and Benveniste [38]. In his PhD, Ouy [37] introduced polyendochrony and showed that it is possible to test it in a polynomial way.

The POLYCHRONY workbench, which is now freely distributed from http://www.irisa.fr/espresso/Polychrony, was built progressively during these years and includes a lot of the previously mentioned works. It is regularly updated. In parallel with the POLYCHRONY academic set of tools, an industrial implementation, 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 (http://www.geensys.com/).

2 The SIGNAL language

SIGNAL [9] is a declarative design language expressed within the polychronous model of computation. In the following, we present the SIGNAL language and its associated concepts.

2.1 Synchronized data-flow

Consider as an example the following program expressed in some conventional data-flow formalism [22]:

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

What is the meaning of this program? In an interpretation where the edges are considered as FIFO queues [2], if a is a sequence with non-positive values, the queue associated with a will grow forever, or (if a is a finite sequence) the queue associated with x will eventually be empty although a is non-empty. It is not clear that the meaning of this program is the meaning that the author had in mind! Now, suppose that each FIFO queue consists of a single cell [15]. Then as soon as a negative value appears on the input, the execution can no longer go on: there is a deadlock. This is usually represented by the special undefined value $\perp$ (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 $\perp$ should be handled when reasoning about time, but it has to be considered with a non standard meaning. In the framework of synchronized data-flow, the $\perp$ will correspond to the absence of value at a given logical instant for a given variable (or *signal*). In particular, it must be possible to insert $\perp$'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 that the whole synchronization should be completely handled at compile time, in such a way that the execution phase has nothing to do with $\perp$. This will be assumed by a static representation of the timing relations expressed by each operator. Syntactically, the timing will be implicit in the language. SIGNAL describes processes which communicate through (possibly infinite) sequences of (typed) values with implicit timing: the *signals.* For example, `x` denotes the infinite sequence $\{x_t\}_{t\geq 0}$ where t denotes a logical time index. At any instant, a signal may be present, at which point it holds a value; or absent and denoted by $\perp$ in the semantic notation. There is a particular type of signals called `event`. A signal of this type is always *true* when it is present (otherwise, it is $\perp$). Signals defined with the same time index are said to have the same *clock*, so that clocks are equivalence classes of simultaneous signals. The *clock* of a signal `x`, noted `^x` in the language, represents the set of instants at which the signal `x` is present. A *process* is a system of equations over signals that specifies relations between values and clocks of the signals. A *program* is a process.

Consider a given operator which has, for example, two input signals and one output signal. We shall speak of *synchronous* signals if 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 tokens on a given signal, we can say that one is before the other (*chronology*). Then, for the synchronous approach, an *event* is a set of instantaneous calculations, or equivalently, of instantaneous communications.

2.2 SIGNAL constructs

SIGNAL [30] 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. Here, we give a sketch of the primitive constructs (bold-faced) and a few derived constructs (italics) often used. For each of them, the corresponding syntax and definition are mentioned.

Functions/Relations. Let `f` be a symbol denoting a n-ary function $[\![f]\!]$ on instantaneous values (e.g., arithmetic or Boolean operation). Then, the Signal expression

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

defines an elementary process such that:

$$y_t \neq \perp \Leftrightarrow x1_t \neq \perp \Leftrightarrow ... \Leftrightarrow xn_t \neq \perp, \forall t : y_t = f(x1_t, ..., xn_t),$$

where xi_k denotes the k^{th} element of the sequence denoted by $\{xi_t\}_{t\geq 0}$.

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

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

defines an elementary process such that:

$$x_t \neq \perp \Leftrightarrow y_t \neq \perp,\ \forall t > 0 : y_t = x_{t-1}, y_0 = c.$$

At the first instant, the signal `y` takes the initialization value `c`. Then, at any instant, `y` takes the previous value of `x`.

Under-sampling. This operator has one data input and one Boolean "control" input, but it has a different meaning when one of the inputs holds $\perp$. In this case, the output is also $\perp$; at any logical instant where both input signals are defined, the output will be different from $\perp$ if and only if the control input holds the value *true*. Then, the Signal expression

```
y:= x when b
```

defines an elementary process such that:

$$y_t = x_t \text{ if } b_t = true,\ \text{else } y_t = \perp.$$

The derived statement `y:= when b` is equivalent to `y:= b when b`. In this case, `y` has the type `event` (it is always *true* when present).

Deterministic merging. The unique output provided by this operator is defined (i.e., with a value different from $\perp$) 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 an elementary process such that:

$$z_t = x_t \text{ if } x_t \neq \perp,\ \text{else } z_t = y_t.$$

Parallel composition: Resynchronizations (that is to say, possible insertions of $\perp$) 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` defines a new process, denoted by

```
(| P | Q |)
```

where common names refer to common signals. Then, `P` and `Q` communicate through their common signals.

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 defined in a process `P`,

```
P where x
```

defines a new process where communication ways (for composition) are those of `P`, except `x`.

Derived operators are defined from the kernel of primitive operators. In particular:
Clock extraction: `h := ^x` specifies the clock `h` of `x` as a signal of type `event`, and can be defined as: `h := (x = x)`.
Synchronization: `x1 ^= x2` specifies that `x1` and `x2` have the same clock, and is defined as: `(| h := (^x1 = ^x2) |) where h`.
Clock union: `h := x1 ^+ x2` specifies the clock union of `x1` and `x2`, which is also defined as: `h := ^x1 default ^x2`.
Clock intersection: `h := x1 ^* x2` specifies the clock intersection of `x1` and `x2`, which is also defined as: `h := ^x1 when ^x2`.
Memory: `y := x cell b init y0` allows to memorize in `y` the latest value carried by `x` when `x` is present or when `b` is *true*. It is defined as:
`(| y := x default (y $ 1 init y0) | y ^= x ^+ (when b) |)`.

2.3 A simple example

The purpose of the following process is to define a signal `v` which counts in the reverse order the number of occurrences of the events at which a Boolean signal `reset` holds the value *false*; `v` is reinitialized (with a value `v0`) each time `reset` is *true*.

```
(| zv := v $ 1 init 0
 | vreset := v0 when reset
 | zvdec := zv when (not reset)
 | vdec := zvdec - 1
 | v := vreset default vdec
 | reach0 := when (zv = 1)
 |) where integer zv, vreset, zvdec, vdec;
```

Comments: `v` is defined with `v0` each time `reset` is present and has the value *true* (operator `when`); otherwise (operator `default`), it takes the value of `zvdec-1`, `zvdec` being defined as the previous value of `v` (delay), `zv`, when this value is present and moreover, when `reset` is present and has the value *false* (operator `when`). The Boolean signal `reach0` is defined (with the value *true*) when the previous value of `v` was equal to `1`. Notice that `v` is decremented when `reset` has a value *false*.

Model of process: The above process can be abstracted and declared as a *model of process*, with its ways of communication, stated explicitly (some intermediate variables have also been removed):

```
process RCOUNT =
  { integer v0; }
  ( ? boolean reset;
    ! event reach0;
      integer v; )
  (| zv := v $ 1 init 0
   | v := (v0 when reset) default ((zv when (not reset)) - 1)
   | reach0 := when (zv = 1)
   |)
  where
    integer zv;
  end;
```

It may be referred to as, for example, `RCOUNT(10)` (`v0` is a formal parameter of the process; "`?`" stands as a tag for the input signals and "`!`" for the output ones). Here, there are one input signal, `reset`, and two output signals, `reach0` and `v`.

2.4 Constraint programming with Signal

We demonstrate definition of programs by property specification and addition of constraints, that illustrates the style of programming that Signal leads to: programming by composition of systems of equations. We consider the example of a "mailbox", defined in an incremental way.

In a first step, we define a simple memory, that may be represented by a cell `M_In` with one input `In` and one output `Out`: each time a new value arrives on the input `In`, it replaces the previously memorized one and is itself memorized in `M_In`; each time a value is required on the output `Out`, the current value of `M_In` is delivered on this output.

In Signal, this is specified by the following equations:

```
(a)      M_In := In default (M_In $ 1 init V0)
(b)    | Out := M_In when ^Out                        (1)
```

(`V0` represents a constant value, used as initial value of the memory). Memorization is through the delay operator: memorized signals are the state variables of a Signal program.

The notation `^Out` is the syntax that represents the clock of the signal `Out`, considered as an event-type signal (Boolean which is *true* when it is present).

Let us comment both equations of (1). To simplify, we name `zM_In` the signal `(M_In $ 1 init V0)`. The equation

```
M_In := In default zM_In
```

defines the signal `M_In` as made up from:

- the values of the signal `In` when `In` is defined (i.e., at the instants of the clock of `In`),
- the values of the signal `zM_In` when `In` is not present, but `zM_In` is.

When neither `In` nor `zM_In` are present, `M_In` is also not defined. The instants of the clock of the signal `M_In` are the union of the instants of `In` and of those of `zM_In`. Since at each one of its instants, the signal `zM_In` has the value that `M_In` had at the previous instant, equation (a) expresses indeed that `M_In` is defined by `In` when a new value arrives on the input `In` and keeps its previous value at the other instants.

In Signal, every signal is characterized by its clock (the set of instants at which it is defined), including signals representing state variables. A signal `(X $ 1)` always has the same clock as the corresponding signal `X`. Here, `zM_In` and `M_In` have the same clock, which is written: `zM_In ^= M_In`. Thus, considering clocks, equation (a) states simply that the clock of `M_In` is at least as frequent as the clock of `In`, which may be written: `M_In ^> In`.

Consider now equation (b) of (1). The signal `Out` is defined by the value of the signal `M_In` when `M_In` is defined *and* the second argument of the `when` operator (here, the signal `^Out`), is also defined and has the value *true* (here, the signal `^Out`, which represents a clock, has the value *true* whenever it is defined). Thus the instants of the clock of the signal `Out` are the intersection of the instants of `M_In` and the instants at which the condition that forms the second argument of the `when` is *true*. Here, the signal `Out` is defined by the value of the memory at the instants of its own clock (the instants at which some value is required on the output `Out`). Considering clocks, equation (b) states that the clock of `M_In` is at least as frequent as the clock of `Out`: `M_In ^> Out`.

The process (1) does not fix the clock of the memory `M_In`. If one wants to specify that this clock must be exactly the union of the instants of `In` and the instants of `Out`, it is sufficient to add the following synchronization equation:

```
M_In ^= In ^+ Out
```

Let us declare and name this small program:

```
process MEM =
  { type T; T V0; }
```

```
( ? T In;
  ! T Out; )
(| M_In := In default (M_In $ 1 init V0)
 | Out := M_In when (^Out)
 | M_In ^= In ^+ Out
 |)
where
  T M_In;
end;
```

Comments: There are two static parameters for this program: the first one is the generic type, T, of memorized values; the second one is the initial value V0.

In a first step, we have defined a program MEM that works asynchronously, according to arrivals and requests of messages, without particular constraints. If $\perp$ denotes the absence of value at a given instant, we have, for any instant t:

$$\begin{array}{rcl} \mathtt{In}_t \neq \perp & \Rightarrow & \mathtt{M\_In}_t = \mathtt{In}_t \\ \mathtt{In}_t = \perp & \Rightarrow & \mathtt{M\_In}_t = \mathtt{M\_In}_{t-1} \\ \mathtt{Out}_t \neq \perp & \Rightarrow & \mathtt{Out}_t = \mathtt{M\_In}_t \end{array} \tag{2}$$

Suppose now we want to add a first constraint to the program, that there is no loss of messages: every data that arrives in the memory has to be read. For any instant t:

$$\mathtt{In}_t \neq \perp \quad \Rightarrow \quad \exists s \geq 0\ \mathtt{Out}_{t+s} = \mathtt{In}_t \tag{3}$$

For that purpose, we define a Boolean signal, accept, which is *true* when some value is emitted to the environment (instants of ^Out) and *false* when some value is received on the input of the memory (instants of ^In):

```
(| accept := (^Out) default (not ^In) default z_accept
 | z_accept := accept $ 1 init true                          (4)
 |)
```

(note that (not ^In) means (false when ^In)). At the instants which are not instants of In nor instants of Out, the Boolean accept keeps its previous value.

The property (3) is easily translated as a constraint on the instants at which $\mathtt{In}_t \neq \perp$: a new input In can be accepted only when the previous value has been emitted, i.e., when the Boolean accept was *true* at the previous instant. In Signal, if we rename here WRITE_ACCEPT the previous value of the Boolean accept:

```
(| WRITE_ACCEPT := z_accept
 | In ^= In when WRITE_ACCEPT                                (5)
 |)
```

(the clock of the Boolean WRITE_ACCEPT could be fixed, for instance at the clock of all instants at which a new input may be read). If we compose the program

MEM with the equations (4) and (5), we get a new program the behavior of which is the intersection of the behaviors of its components. Thus it is a memory that accepts a new input only when the previous value has been emitted.

Suppose now we accept loss of messages, but we want to avoid their possible duplication on the output: a given message cannot be emitted several times on the output `Out`. Here, we forget the equations (5), but we keep the definition of the Boolean `accept` since the problem is the dual of the previous one: a new value can be emitted on `Out` only from the input of a new value in the memory, i.e., when the previous value of the Boolean `accept` was the value *false*. However, keeping the same definition of `accept`, we must add as possible instants of `Out` the instants at which there is, at the same time, some input on `In` (if this is not forbidden: in that case, a value that arrives on `In` is immediately emitted on `Out`):

```
(| READ_ACCEPT := not z_accept
 | DIRECT_READ_ACCEPT := (^In) default READ_ACCEPT
 | Out ^= Out when DIRECT_READ_ACCEPT                    (6)
 |)
```

Then, the composition of the program `MEM` with the equations (4) and (6) specifies a memory that emits at most once the memorized values.

If we want to add both constraints, in other words, to specify a mailbox, for which every received message will be emitted once and only once, then it is sufficient to compose the program `MEM` with the equations (4), (5) and (6).

2.5 Oversampling in SIGNAL

We describe in this section, again with a small example, a characteristic feature of the SIGNAL language: the ability to specify *oversampling*, i.e. programs for which outputs may be more frequent than inputs.

We consider a communication protocol for which FDMA accesses (*frequency division multiple access*) are transformed into TDMA ones (*time division multiple access*). In addition, we suppose that the number of simultaneous users varies along time. Part of the specification consists in receiving packets containing some variable number `u` of information, and re-emitting these information as a sequence of `u` successive information. This mechanism of variable rate oversampling can be expressed as follows in SIGNAL (we concentrate here on the mechanism itself, forgetting the content of carried information):

```
process OVERSAMPLE =
  ( ? integer u;
    ! boolean b; )
  (| z := u default v         (i)
   | v := (z $ 1 init 1) - 1  (ii)
   | b := v <= 0              (iii)
   | u ^= when b              (iv)
   |)
```

```
where
  integer z, v;
end;
```

A trace for this program is given below:

`u` :	3			2		5	...
`z` :	3	2	1	2	1	5	...
`v` :	0	2	1	0	1	0	...
`b` :	*T*	*F*	*F*	*T*	*F*	*T*	...

Equation (iv) expresses that the clock of the input `u` is defined by the set of instants at which the Boolean `b` is *true* Thus the input `u` is read when the Boolean `b` is *true*. From equation (iii), the Boolean `b` is *true* at the instants at which `v` is negative or null, and *false* at the other instants of `v` (from this equation, `b` and `v` are also defined at the same instants, as it is always the case for signals appearing in arithmetic or Boolean functions/relations). From equation (ii), `v` is defined as the delayed value of `z`, decremented by 1 (from (ii), `v` and `z` have also the same clock). Finally, equation (i) expresses that `z` is equal to `u` as a priority, or by default to `v` when `u` is absent.

The clock of the output `b` is more frequent than the clock of the input `u`.

3 Compiling Signal programs

Among relevant questions when compiling Signal programs, there are the following ones:

- Is the program deadlock free?
- Has it an effective execution?
- If so, what scheduling may be statically calculated (for a multiprocessor implementation)?

To be able to answer these questions, two basic tools are used before execution on a given architecture. The first one is the modeling of the synchronization relations in $\mathcal{F}_3$ by polynomials with coefficients in the finite field $\mathbb{Z}/_{3\mathbb{Z}}$ of integers modulo 3. The second one is the directed graph of data dependencies. These basic tools are used for all compiling services: program transformations, optimizations, abstraction, code generation, temporal profiling, etc.

3.1 The synchronization space

First, let us consider Signal processes restricted to the single domain of Boolean values. The equation

```
x3 := x1 when x2
```

expresses the following assertions:

- if `x1` is defined, and `x2` is defined and *true*, then `x3` is defined and `x3 = x1`,
- if `x1` is not defined, or `x2` is not defined, or `x2` is defined and *false*, then `x3` is not defined.

It appears that useful information are (if `x` is a signal):

- `x` is defined and *false*,
- `x` is defined and *true*,
- `x` is not defined.

They can be respectively encoded in the finite field $\mathbb{Z}/_{3\mathbb{Z}}$ of integers modulo 3 as the following values: $-1, 1$ and 0. Then, if v is the encoding value associated with the signal `x`, the presence of the signal `x` may be clearly represented by v^2. This representation of an indeterminate value of `x` (*true* or *false*) leads to an immediate generalization to non-Boolean values: their presence is encoded as 1 and their absence as 0. In this way, v^2 may be considered as the proper clock of the signal `x`.

This principle is used to represent synchronization relations expressed through Signal programs. In the following, each signal and its encoding value are denoted by the same variable. The coding of the elementary operators is deduced from their definition. This coding is introduced below:

- The equations

  $$\mathtt{y}^2 = \mathtt{x}_1^2 = \ldots = \mathtt{x}_n^2$$

  denoting the equality of the respective clocks of signals $\mathtt{y}$, $\mathtt{x}_1$, $\ldots$, $\mathtt{x}_n$ are associated with $\mathtt{y} := \mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n)$ (all the synchronous processes are encoded in this way, however, "dynamical systems" in $\mathcal{F}_3$ must be used to encode Boolean delays—this is not detailed here [8]).

- Boolean relations may be completely encoded in $\mathcal{F}_3$. For instance, $\mathtt{x}_2 = -\mathtt{x}_1$ corresponds to $\mathtt{x}_2 := \mathtt{not}\ \mathtt{x}_1$:
  if $\mathtt{x}_1 = \mathit{true}$, then $\mathtt{x}_1 = 1$ and $-(\mathtt{x}_1) = -1$, which is associated with *false*.

- The equation

  $$\mathtt{x}_3 = \mathtt{x}_1(-\mathtt{x}_2 - \mathtt{x}_2^2)$$

  is associated with $\mathtt{x}_3 := \mathtt{x}_1\ \mathtt{when}\ \mathtt{x}_2$ ($\mathtt{x}_1$, $\mathtt{x}_2$, $\mathtt{x}_3$ Boolean signals); it may be interpreted as follows: $\mathtt{x}_3$ holds the same value as $\mathtt{x}_1$ ($\mathtt{x}_3 = \mathtt{x}_1$) when $\mathtt{x}_2$ is *true* (when $-\mathtt{x}_2 - \mathtt{x}_2^2 = 1$).

  The equation

  $$\mathtt{x}_3^2 = \mathtt{x}_1^2(-\mathtt{x}_2 - \mathtt{x}_2^2)$$

  is associated with $\mathtt{x}_3 := \mathtt{x}_1\ \mathtt{when}\ \mathtt{x}_2$ when $\mathtt{x}_1$, $\mathtt{x}_3$ are non-Boolean signals.

- The equation

  $$\mathtt{x}_3 = \mathtt{x}_1 + (1 - \mathtt{x}_1^2)\mathtt{x}_2$$

  is associated with $\mathtt{x}_3 := \mathtt{x}_1$ **`default`** $\mathtt{x}_2$ ($\mathtt{x}_1$, $\mathtt{x}_2$, $\mathtt{x}_3$ Boolean signals); it is interpreted as follows: $\mathtt{x}_3$ has a value when $\mathtt{x}_1$ is defined, i.e., when $\mathtt{x}_1^2 = 1$ (then $\mathtt{x}_3$ holds the same value as $\mathtt{x}_1$: $\mathtt{x}_3 = \mathtt{x}_1^2\mathtt{x}_1 = \mathtt{x}_1$), or when $\mathtt{x}_2$ is defined but not $\mathtt{x}_1$, i.e., when $(1-\mathtt{x}_1^2)\mathtt{x}_2^2 = 1$ (then $\mathtt{x}_3$ holds the same value as $\mathtt{x}_2$: $\mathtt{x}_3 = (1 - \mathtt{x}_1^2)\mathtt{x}_2^2\mathtt{x}_2 = (1 - \mathtt{x}_1^2)\mathtt{x}_2$).

  The equation

  $$\mathtt{x}_3^2 = \mathtt{x}_1^2 + (1 - \mathtt{x}_1^2)\mathtt{x}_2^2$$

  is associated with $\mathtt{x}_3 := \mathtt{x}_1$ **`default`** $\mathtt{x}_2$ when $\mathtt{x}_1$, $\mathtt{x}_2$, $\mathtt{x}_3$ are non-Boolean signals.

Then the composition of Signal processes collects the clock expressions of every composing process.

3.2 The clock calculus

The algebraic coding of the synchronization relations has a double function. First, it is the way to detect synchronization constraints. Consider for example the following program (which is that of section 2.1):

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

The meaning of this program is "add `a` to (`a` **`when`** `a` > 0)"; remember that it must be "rejected" if `a` can take any value since the clocks are then inconsistent. More exactly, this program constrains the possible values of `a`. Its algebraic encoding is

$$\begin{gathered}\mathtt{c}^2 = \mathtt{a}^2\\ \mathtt{x}^2 = \mathtt{a}^2(-\mathtt{c} - \mathtt{c}^2)\\ \mathtt{y}^2 = \mathtt{x}^2 = \mathtt{a}^2\end{gathered}$$

which results in $\mathtt{c}^2 = \mathtt{a}^2 = \mathtt{y}^2 = \mathtt{x}^2 = \mathtt{a}^2(-\mathtt{c} - \mathtt{c}^2)$
and by substitution $\mathtt{c}^2 = \mathtt{c}^2(-\mathtt{c} - \mathtt{c}^2)$
and then $\mathtt{c} = 1$ or $\mathtt{c} = 0$.

But c is the result of the evaluation of the non-Boolean signal a. However the coding in $\mathcal{F}_3$ does not allow reasoning about non-Boolean values, therefore the actual value (*true* or *false*) of c cannot be predicted.

The other function of this coding is to organize the control of the program. An order relation may be defined on the set of clocks: a clock h^2 is said to be greater than a clock k^2, which is denoted by $h^2 \geq k^2$, if the set of instants of k is included in the set of instants of h (k is an undersampling of h). The set of clocks with this relation is a lattice. The purpose of the clock calculus is to synthesize the upper bound of the lattice, which is called the *master clock*, and to define each clock by some computation expression, i.e., an undersampling of the master clock according to values of Boolean signals. However, for a given SIGNAL process, the master clock may not be the clock of a signal of the process. In this case, several maxima (local master clocks) will be found.

For a program to be "correct", the partial order induced by the inclusion of instants, restricted to the undersamplings by a *free* Boolean condition (input Boolean signal or Boolean expression on non-Boolean signals), must be a tree, the root of which is the more frequent clock. Then such a program, also referred to as *endochronous*[1], can be run in an autonomous way (master mode). Otherwise, there are several local master clocks, and the process needs extra information from its environment to be run in a *deterministic* way. So, an endochronous program is deterministic [31].

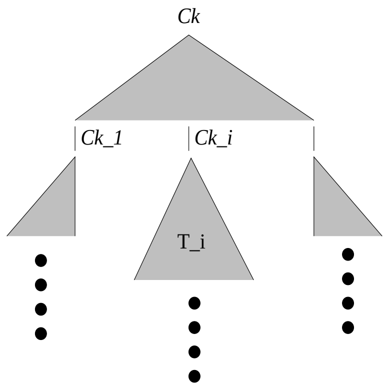

Figure 1: Clock hierarchy of an endochronous program.

FIG. 1 illustrates the clock hierarchy of an endochronous program. It is described by a unique tree where the root node represents the master clock (*Ck*). We can notice that from this global tree, one can derive several "endochronous" sub-trees (for example T_i).

Clock expressions can be rewritten as Boolean expressions of a SIGNAL program. The operator ^+ represents the sum of clocks (upper bound) and the operator ^$*$ represents the product (lower bound). Then, any clock expression may be recursively reduced to a sum of monomials, where each monomial is a product of undersamplings (otherwise, the clock is a root).

[1]A more formal characterization of *endochrony* can be found in [31].

3.3 An example

Consider again the process RCOUNT of section 2.3 (in the version written with intermediate signals). The clock calculus finds the following clocks:

$$\begin{gathered}
\texttt{reset}^2 \\
\texttt{vreset}^2 = -\texttt{reset} - \texttt{reset}^2 \\
\texttt{v}^2 = \texttt{zv}^2 = \alpha^2 = (-\texttt{reset} - \texttt{reset}^2) + (\texttt{reset} - \texttt{reset}^2)\texttt{v}^2 \\
\texttt{vdec}^2 = \texttt{zvdec}^2 = \texttt{v}^2(\texttt{reset} - \texttt{reset}^2) \\
\texttt{reach0}^2 = -\alpha - \texttt{v}^2
\end{gathered}$$

where α is the coding of $\texttt{zv} = 1$.

The clock calculus does not synthesize a master clock for this process. In fact, it is not endochronous (and it is non-deterministic): when reset is *false*, then zvdec is defined if zv is defined, i.e., if v is defined; but v is defined (when reset is *false*) if vdec is defined, i.e., if zvdec is defined, and then, when reset is *false*, an occurrence of v may occur, but does not necessarily occur.

The hierarchy of clocks is represented by the following SIGNAL process, which defines several trees (the roots of which are clk_reset, clk_vdec and clk_v):

```
(| (| clk_reset ^= reset
    | (| clk_vreset := when reset
       | clk_vreset ^= vreset
       | clk_1_2 := when (not reset)
       |)
    |)
 | (| clk_vdec := clk_1_2 ^* ck_v
    | clk_vdec ^= vdec ^= zvdec
    |)
 | (| ck_v := ck_vreset ^+ clk_vdec
    | ck_v ^= v ^= zv}
    | (| reach0 := when (zv=1)
       |)
    |)
 |)
```

The hierarchy is represented by the composition embeddings; the clk_i's represent the names of the clocks considered as signals (the suffixes i are given by the compiler), or they keep their own name if they are event-type signals (like reach0).

Now, we consider the following process, where RCOUNT is used in some context:

```
process USE_RCOUNT =
  { integer v0; }
  ( ? boolean h;
    ! event reach0;
      integer v; )
  (| h ^= v
   | reset := (^reach0 when (^h)) default (not (^h))
   | (reach0, v) := RCOUNT {v0} (reset)
   |)
  where
    boolean reset;
  end;
```

An external Boolean clock `h` defines the instants at which `v` has a value. The `reset` signal is also synchronous with `h` and it has the value *true* exactly when `reach0` is present. There is a master clock ($\mathtt{h}^2 = \mathtt{v}^2 = \mathtt{reset}^2$) and a tree may be built by the compiler. Therefore, the program becomes endochronous.

3.4 The graph of conditional dependencies

The second tool necessary to implement a Signal program on a given architecture is the graph of dependencies. Then, according to criteria to be developed, it will be possible to define subgraphs that may be distributed on different processors. However, a classical data-flow graph would not really represent the dependencies of a Signal program. Since the language handles signals the clocks of which may be different, the dependencies are not constant in time. For that reason, the graph has to express *conditional* dependencies, where the conditions are nothing but the clocks at which dependencies are effective. Moreover, in addition to dependencies between signals, the following relation has to be considered: for any signal `x`, the values of `x` cannot be known before its clock; in other words, `x` depends on $\mathtt{x}^2$. This relation will be implicit below.

The *Graph of Conditional Dependencies* (GCD) calculated by the Signal compiler for a given program is a labeled directed graph where:

- the vertices are the signals, plus clock variables,
- the edges represent dependence relations,
- the labels are polynomials on $\mathcal{F}_3$ which represent the clocks at which the relations are valid.

The following describes the dependencies associated with elementary processes. The notation $c^2 : x_1 \rightarrow x_2$ means that x_2 depends on x_1 (or more exactly, x_1 cannot depend on x_2) when $c^2 = 1$. It has to be noticed that the processes which involve only Boolean signals do not generate data dependencies.

Then, we consider only processes defining non-Boolean signals:

$\mathtt{y} := \mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n)$	$\mathtt{y}^2 : \mathtt{x}_1 \to \mathtt{y}, \ldots, \quad \mathtt{y}^2 : \mathtt{x}_n \to \mathtt{y}$
$\mathtt{y} := \mathtt{x}$ `when` $\mathtt{b}$	$\mathtt{y}^2 : \mathtt{x} \to \mathtt{y}, \quad \mathtt{y}^2 : \mathtt{b} \to \mathtt{y}^2$
$\mathtt{z} := \mathtt{x}$ `default` $\mathtt{y}$	$\mathtt{x}^2 : \mathtt{x} \to \mathtt{z}, \quad \mathtt{y}^2 - \mathtt{x}^2\mathtt{y}^2 : \mathtt{y} \to \mathtt{z}$

Notice that the delay does not produce data dependencies (nevertheless, remember that any signal is preceded by its clock).

The graph, together with the clock hierarchy, represents all the necessary control-flow and data-flow information. It is used to detect incorrect dependencies. Such a bad dependency will appear as a circuit in the graph. However, since dependencies are labeled by clocks, some circuits may not occur at any time. An effective circuit is such that the product of the labels of its arcs is not null. This may be compared with the cycle sum test of [40], to detect deadlock on the dependence graph of a data-flow program.

All the above properties checked by the SIGNAL compiler during the clock calculus are mainly static. Properties such as reachability or liveness, which are dynamic, cannot be addressed with the compiler. For that, the SIGALI tool, which implements a symbolic model checking technique, can be used [35]. Basically, a SIGNAL program denotes an automaton in which states are described by the so-called "state variables" that are defined by the *delay* operator. At each logical instant, the current state of a program is given by the current values of its state variables. The technique adopted in SIGALI consists in manipulating the system of equations resulting from the modeling of SIGNAL programs in $\mathcal{F}_3$ instead of the sets of its states. This avoids the enumeration of the state space, which can potentially explode. So, each set of states is uniquely characterized by a predicate and operations on sets can be equivalently performed on the associated predicates. A few experiments showed that the symbolic model-checking technique adopted by SIGALI enables to check properties on automata with several millions of states within a reasonable delay. More details on SIGALI can be found in [35].

3.5 Sequential code generation

Automatic sequential code generation for endochronous SIGNAL programs is based on the clock hierarchy obtained from the clock calculus and on the graph of conditional dependencies.

To illustrate code generation, we consider the following alternative specification of a one-place buffer in SIGNAL. It uses two sub-processes, one is the process `alternate` which desynchronizes the signals `i` and `o` by synchronizing them to the *true* and *false* values of an alternating Boolean signal `b`. The other one is the process `current`. It defines a `cell` in which values are stored at the input clock `^i` and loaded at the output clock `^o`.

```
process buffer = (? i; ! o;)
  (| alternate (i, o)
   | o := current (i)
   |)
where
    process alternate = (? i, o; ! )
      (| zb := b$1 init true
       | b := not zb
       | o ^= when (not b)
       | i ^= when b
       |) where boolean b, zb;
    end;
    process current = (? i; ! o;)
      (| zo := i cell ^o init false
       | o  := zo when ^o
       |) where zo;
    end;
end;
```

The clock calculus determines three synchronization classes. We observe that `clk_b`, `b`, `zb`, `zo` are synchronous and define the master clock synchronization class of `buffer`; `clk_i` and `clk_o` are sub-clocks of `clk_b`, that correspond to the *true* and *false* values of the Boolean flip-flop variable `b`, respectively. We represent also the dependencies (scheduling relations) calculated by the compiler (this may be written in the Signal syntax):

```
(| clk_b ^= b ^= zb ^= zo
 | (| clk_i := when b
    | clk_i ^= i
    | clk_o := when (not b)
    | clk_o ^= o
    | (| {zo -> o} when clk_o |)
    |)
 | (| zb -> b
    | {i -> zo} when clk_i
    |)
 |)
```

The compiler uses the hierarchization algorithm to find a sequential execution path starting from a system of clock relations. At the main clock `clk_b`, `b` and `clk_o` are calculated from `zb`. At the sub-clock `clk_i`, the input signal `i` is read. At the sub-clock `clk_o` the output signal `o` is written. Finally, `zb` is calculated. Notice that the sequence of instructions follows the scheduling relations determined during clock inference.

```
  buffer_iterate () {
      b = !zb;
      c_o = !b;
      if (b) {
        if (!r_buffer_i(&i)) return FALSE;
      };
      if (c_o) {
        o = i;
        w_buffer_o(o);
      };
      zb = b;
      return TRUE;
 }
```

Such a piece of code is executed within an infinite loop, representing the infinite sequence of reactions of the specified system. Each iteration step corresponds to an instant of the master clock of the system.

3.6 Partitioning programs

The notions presented below are used for partitioning SIGNAL programs into clusters, so as to get abstractions for separate compilation, and from which it is possible to generate code, either with static scheduling of the clusters, or multi-threaded code with dynamic scheduling. It is also the base for generating distributed code. Further technical details on this topic can be found in [19, 5]. In the following, an application is represented by a SIGNAL program $P = P_1 \mid P_2 \mid ... \mid P_n$, where each sub-program P_i can be itself recursively composed of other sub-programs (i.e., $P_i = P_{i1} \mid P_{i2} \mid ... \mid P_{im}$). The following hypothesis are assumed:

1. considered programs P are *endochronous* (see Section 3.2), hence deterministic;
2. they do not contain any definition leading to cycles;
3. there is a set of processors $q = \{q_1, q_2, ..., q_m\}$; and
4. a function $locate : \{P_i\} \longrightarrow \mathcal{P}(q)$, which associates with each subpart of an application $P = P_1 \mid P_2 \mid ... \mid P_n$ a non-empty set of processors (the allocation can be done either manually or automatically).

First transformation. Let us consider a SIGNAL program $P = P_1 \mid P_2$, as illustrated in FIG. 2. Each sub-program P_i (represented by a circle) is itself composed of four sub-programs P_{i1}, P_{i2}, P_{i3} and P_{i4}. The program P is distributed on two processors q_1 and q_2 as follows:

$$\begin{array}{l} \forall i \in \{1,2\}\ \forall k \in \{1,2\}, \quad locate(P_{ik}) = \{q_1\}, \quad and \\ \forall i \in \{1,2\}\ \forall k \in \{3,4\}, \quad locate(P_{ik}) = \{q_2\} \end{array}$$

Hence, P can be rewritten into $P = Q_1 \mid Q_2$, where $Q_1 = P_{11} \mid P_{12} \mid P_{21} \mid P_{22}$ and $Q_2 = P_{13} \mid P_{14} \mid P_{23} \mid P_{24}$:

$$\begin{array}{lcll}
P & = & P_1 \mid P_2 & \\
 & = & (P_{11} \mid P_{12} \mid P_{13} \mid P_{14}) \mid (P_{21} \mid P_{22} \mid P_{23} \mid P_{24}) & \\
 & = & (P_{11} \mid P_{12}) \mid (P_{13} \mid P_{14}) \mid (P_{21} \mid P_{22}) \mid (P_{23} \mid P_{24}) & \\
 & = & (P_{11} \mid P_{12}) \mid (P_{21} \mid P_{22}) \mid (P_{13} \mid P_{14}) \mid (P_{23} \mid P_{24}) & \text{(commutativity of } \mid) \\
 & = & (P_{11} \mid P_{12} \mid P_{21} \mid P_{22}) \mid (P_{13} \mid P_{14} \mid P_{23} \mid P_{24}) & \\
 & = & Q_1 \mid Q_2 &
\end{array}$$

The sub-programs Q_1 and Q_2 resulting from the partitioning of P are called *s-tasks* [19]. This transformation yields a new form of the program P that reflects a multi-processor architecture. It also preserves the semantics of the transformed program (since it simply consists of program rewriting).

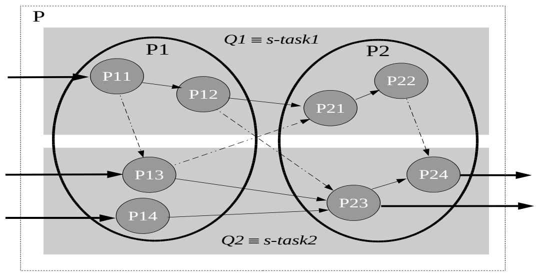

Figure 2: Decomposition of a SIGNAL process into two s-tasks Q_1 and Q_2.

The above transformation remains valid even if $locate(P_{ik})$ is not a singleton. In that case, P_{ik} is split into new sub-programs which are considered at the same level as P_{jl}'s where $locate(P_{jl})$ is a singleton. For instance, let us consider the program P, it can be rewritten as:

$$P \;=\; P_{11\_13} \mid P_{12} \mid P_{14} \mid P_{21} \mid P_{22} \mid P_{23} \mid P_{24}$$

where $locate(P_{11\_13}) = \{q_1, q_2\}$. Then it follows that

$$\begin{array}{lcll}
P & = & P_{11} \mid P_{13} \mid P_{12} \mid P_{14} \mid P_{21} \mid P_{22} \mid P_{23} \mid P_{24} & (P_{11\_13} \text{ is split}) \\
 & = & P_{11} \mid P_{12} \mid P_{13} \mid P_{14} \mid P_{21} \mid P_{22} \mid P_{23} \mid P_{24} & \text{(commutativity of } \mid) \\
 & = & P_1 \mid P_2 &
\end{array}$$

Second transformation. We want to refine the level of granularity resulting from the above transformation. For that, let us consider descriptions at processor level (in other words, s-tasks). We are now interested in how to decompose s-tasks into fine grain entities. An s-task can be seen as a set of *nodes* (e.g. P_{11}, P_{12}, P_{21} and P_{22} in Q_1). In order to have an optimized execution at the s-task level, nodes are gathered in such a way that they can be executed atomically. By atomic execution, we mean that nodes execution completes without interruption. So, we distinguish two possible ways to define such subsets of nodes, also referred to as *clusters*: either they are composed of a single SIGNAL primitive construct, or they contain more than one primitive construct. The former yields a finer granularity than the latter. However, from the execution point of view, the latter is more efficient since more actions can be achieved at a same time (i.e. atomically).

The definition of atomic nodes uses the following criterion: all the expressions present in such a node depend on the same set of inputs. This relies on a *sensitivity analysis* of programs. We say that a causality path exists between a node N_1 (resp. an input i) and a node N_2 if there is at least one situation where the execution of N_2 depends on the execution of N_1 (resp. on the occurrence of i). In that case, all the possible intermediate nodes are also executed.

Definition 3.1 *Two nodes N_1 and N_2 are sensitively equivalent iff for each input i: there is a causality path from i to N_1 $\Leftrightarrow$ there is a causality path from i to N_2.*

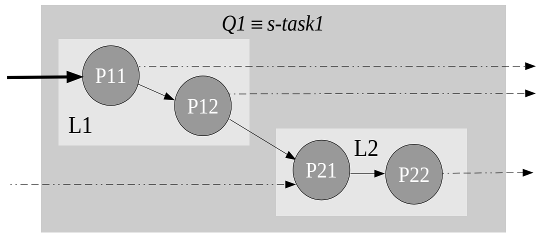

Figure 3: Decomposition of an s-task into two clusters L_1 and L_2.

Sensitively equivalent nodes belong to the same cluster. Inputs always precede outputs within a cluster. Also, if a transformed program is endochronous, the resulting clusters are also endochronous. As a matter of fact, the clock hierarchy associated with each cluster is an endochronous sub-tree of the global clock tree characterizing the program. Hence, this ensures a deterministic execution of each cluster. FIG. 3 shows a decomposition of the s-task Q_1 into two clusters $L1$ and L_2. The input of the sub-program $P11$ (bold-faced arrow) is originally an input of P. The other arrows represent communications between s-tasks (these message exchanges are local to P). We can notice that after this second transformation, the semantic equivalence of the initial program and the resulting one is strictly preserved.

The two transformations presented above describe a partitioning of SIGNAL programs following a multi-task multi-processor architecture.

3.7 Temporal analysis of Signal programs

A technique has been defined in order to address timing issues of Signal programs on different implementation platforms [24]. Basically, it consists of formal transformations of a program into another Signal program that corresponds to a so-called *temporal interpretation* of the initial one. The new program serves as an *observer* of the initial program. An observer of a program `P` is an *abstraction* $\mathcal{O}(\texttt{P})$ of `P` in which we only specify the properties we want to check. The term "abstraction" means here that $\mathcal{O}(\texttt{P})$ does not constrain the original behavior of `P` when the two programs are composed. As shown in Figure 4, the observer receives from the observed program the signals required for analysis and indicates whether or not the considered properties have been satisfied (this can be expressed, e.g., through Boolean output signals like in Lustre programs [21]). The use of observers for verification is very practical because they can be easily described in the same formalism as the observed program. Thus, there is no need to combine different formalisms as in other analysis techniques such as some model-checking techniques, which associate temporal logics with automata [14].

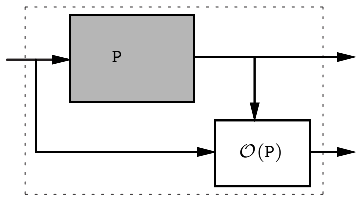

Figure 4: Composition of a program `P` together with its observer $\mathcal{O}(\texttt{P})$.

The Polychrony environment associated with the Signal language provides functionalities including those mentioned in the above sections.

4 Conclusions

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.

Polychrony supports the synchronous, multi-clocked, data-flow specification language Signal. 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, or with the Coq theorem prover. It is freely distributed from http://www.irisa.fr/espresso/Polychrony. Based on the SIGNAL language, 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).

Many documents, reference publications and examples are also available on the POLYCHRONY site.

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. C, C++, multi-threaded and real-time Java and SYNDEX [20] code generators are provided.

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 (http://www.topcased.org/) and OpenEmbeDD (http://www.openembedd.org/).

The Geensys company supplies a commercial implementation of Polychrony, called RT-BUILDER, used for industrial scale projects by Snecma/Hispano-Suiza and Airbus Industries (see http://www.geensys.com/).

References

[1] T. Amagbegnon. *Forme canonique arborescente des horloges de* SIGNAL. PhD thesis, Université de Rennes 1, November 1995.

[2] Arvind and K.P. Gostelow. *Some Relationships between Asynchronous Interpreters of a Dataflow Language*. North-Holland, 1978.

[3] P. Aubry. *Mises en œuvre distribuées de programmes synchrones*. PhD thesis, Université de Rennes 1, IFSIC, October 1997.

[4] M. Belhadj. *Conception d'architectures en utilisant* SIGNAL *et VHDL*. PhD thesis, Université de Rennes I, IFSIC, December 1994.

[5] A. Benveniste. Safety critical embedded systems design: the SACRES approach. In *Formal Techniques in Real-Time and Fault Tolerant systems, FTRTFT'98 school*, Lyngby, Denmark, September 1998.

[6] A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in dataflow synchronous languages: specification and distributed code generation. *Information and Computation*, 163(1):125–171, 2000.

[7] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1):64–83, January 2003.

[8] A. Benveniste and P. Le Guernic. Hybrid dynamical systems theory and the SIGNAL language. *IEEE transactions on Automatic Control*, 35(5):535–546, May 1990.

[9] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the Signal language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.

[10] A. Benveniste, P. Le Guernic, Y. Sorel, and M. Sorine. A denotational theory of synchronous reactive systems. *Information and Computation*, 99(2):192–230, August 1992.

[11] L. Besnard. *Compilation de* Signal *: horloges, dépendances, environnements*. PhD thesis, Université de Rennes I, IFSIC, September 1992.

[12] L. Besnard, T. Gautier, and P. Le Guernic. SIGNAL V4-INRIA version: Reference Manual (working version), May 2008.

[13] B. Chéron. *Transformations syntaxiques de Programmes* Signal. PhD thesis, Université de Rennes I, IFSIC, September 1991.

[14] C. Daws and S. Yovine. Two Examples of Verification of Multirate Timed Automata with Kronos. In *Proceedings of the 16th IEEE Real Time Systems Symposium (RTSS'95)*, Pisa, Italy, December 1995. IEEE Press.

[15] 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*, pages 187–216. Lecture Notes in Computer Science, 5, Springer-Verlag, 1974.

[16] B. Dutertre. *Spécification et preuve de systèmes dynamiques*. PhD thesis, Université de Rennes I, IFSIC, December 1992.

[17] A. Gamatié. *Modélisation polychrone et évaluation de systèmes temps réel*. PhD thesis, Université de Rennes I, Rennes, France, May 2004.

[18] T. Gautier. *Conception d'un langage flot de données pour le temps réel*. PhD thesis, Université de Rennes I, December 1984.

[19] T. Gautier and P. Le Guernic. Code generation in the Sacres project. In *Safety-critical Systems Symposium, SSS'99, Springer*, Huntingdon, UK, February 1999.

[20] 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.

[21] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In *Algebraic Methodology and Software Technology*, pages 83–96, Enschede, The Netherlands, 1993. Springer-Verlag 1994.

[22] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing 74*, pages 471–475. North-Holland, 1974.

[23] A. Kountouris. *Outils pour la validation temporelle et l'optimisation de programmes synchrones.* PhD thesis, Université de Rennes I, Rennes, France, October 1998.

[24] 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*, pages 6/1–6/9, Bristol, UK, February 1996. HP Labs.

[25] M. Le Borgne. *Systèmes dynamiques sur des corps finis.* PhD thesis, Université de Rennes I, IFSIC, September 1993.

[26] B. Le Goff. *Inférence de contrôle hiérarchique : application au temps réel.* PhD thesis, Université de Rennes I, IFSIC, 1989.

[27] P. Le Guernic. SIGNAL : Description algébrique des flots de signaux. In *Architecture des machines et systèmes informatiques*, pages 243–252. Hommes et Techniques, November 1982.

[28] 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.

[29] 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*, pages 413–438, 1991.

[30] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, Sep. 1991.

[31] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for System Design. *Journal for Circuits, Systems and Computers*, 12(3):261–304, April 2003.

[32] J.-C. Le Lann. *Simulation et synthèse de circuits s'appuyant sur le modèle synchrone.* PhD thesis, Université de Rennes 1, IFSIC, March 2002.

[33] O. Maffeïs. *Ordonnancements de graphes de flots synchrones ; application à la mise en œuvre de* SIGNAL. PhD thesis, Université de Rennes I, IFSIC, January 1993.

[34] H. Marchand. *Méthodes de synthèse d'automatismes décrits par des systèmes à événements discrets finis.* PhD thesis, Université de Rennes 1, IFSIC, October 1997.

[35] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic. Synthesis of discrete-event controllers based on the SIGNAL environment. *Discrete Event Dynamic System: Theory and Applications*, 10(4):325–346, October 2000.

[36] D. Nowak. *Spécification et preuve de systèmes réactifs.* PhD thesis, Université de Rennes 1, IFSIC, October 1999.

[37] J. Ouy. *Génération de code asynchrone dans un environnement polychrone pour la production de systèmes GALS.* PhD thesis, Université de Rennes 1, IFSIC, January 2008.

[38] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. *Formal Methods in System Design*, 28(2):111–130, 2006.

[39] I. Smarandache. *Transformations affines d'horloges : application au codesign de systèmes temps-réel en utilisant les langages* SIGNAL *et Alpha.* PhD thesis, Université de Rennes 1, IFSIC, October 1998.

[40] W. W. Wadge. An extensional treatment of dataflow deadlock. In G. Kahn, editor, *Semantics of Concurrent Computation*, pages 285–299. Lecture Notes in Computer Science, 70, Springer-Verlag, 1979.