Electronic Notes in Theoretical Computer Science 88 (2003)
URL: `http://www.elsevier.nl/locate/entcs/volume88.html` 15 pages

An Overview of the SIGNAL Clock Calculus

Mirabelle Nebut [1]

L3I – *Univ. de La Rochelle – av M. Crépeau – 17042 La Rochelle Cédex 1 – France*

Abstract

The SIGNAL compilation process is based on a formal analysis called *clock calculus*. It constructively determines if a specification is endochronous by synthesizing a sequential control structure. The analysis applies to relations over clocks inferred from specifications and encoded into boolean equations. This paper first gives an overview of requisite fundamental notions related to clocks (control in data-flow specifications, link with operational semantics, boolean encoding) then uses this lighting to present technical aspects of the calculus [1,2].

Key words: SIGNAL, clock calculus, clocks, data-flow paradigm, control-flow synthesis, code generation

1 Introduction

The automatic and safe code generation from specifications is one of the attractive advantages offered by the synchronous development environments. While they share common concerns such as code efficiency and compactness, synchronous languages have developed proper and very different compilation techniques that yield to different code structures (automaton, imperative single loop, functional code, circuits, boolean equations, etc). The earlier LUSTRE and ESTEREL compilers generate an automaton, while the SIGNAL and SCADE compilers synthesize a sequential control structure. Nevertheless the ESTEREL compilation process is strongly evolving toward the generation of sequential code from a control-flow graph (ESUIF, SAXO-RT [12]), while SIGNAL might exploit the LUSTRE generation of automaton for verification purposes. This suggests that, in spite of paradigm differences, each language could take advantage of some others outstanding methods. It requires that their basic principles are clearly identified and as far as possible abstracted from technical, implementation-related and language-dependent details.

This paper focuses on the SIGNAL case. The SIGNAL compilation process, called *clock calculus*, is a very rich analysis based on original ideas which

[1] Email: `mnebut@univ-lr.fr`

involve both theoretical and implementation-related concepts, non standard objects (clocks), techniques (transformation of a system of equations into a set of definitions), data structures (clock trees) and specialized vocabulary. This variety of aspects makes it difficult to explain and it has been mainly presented from a technical implementation-related point of view [1,2]. Though an example-based presentation, or systematic translation schemes into imperative code should be explanatory, the present paper — that adds no research contribution — focuses on an overview of fundamental requisite notions that highlight the technical details presented in [1,2]. Section 2 recalls basic synchronous (mostly data-flow) models and operational semantics [2]. Section 3 focuses on clocks as they are defined (as set of instants) and used (for control purposes) in the data-flow paradigm, then presents their formalization in the Signal context, emphasizing their combinational nature and their boolean encoding. Section 4 presents the clock calculus itself, through both intuitive, theoretical and technical aspects. Finally Sect. 5 concludes.

2 General Synchronous Notions

The synchronous paradigm considers that the execution of a reactive system is an infinite sequence of *reactions* (a reaction is the process of inputs acquisition, computations and outputs emission). The synchronous hypothesis assumes that a reaction occurs inside a *logical instant.* Figure 1(a) represents such an execution, indexed by a discrete sequence of instants t_i. In Sect 2.1 we explain that synchronous variables can be absent inside a reaction. Then we focus on the data-flow paradigm and present in Sect. 2.2 a general operational model, instantiated on the Signal case.

2.1 Absence of Signals and Variables

The *imperative* paradigm is dedicated to control dominated applications. The control of imperative specifications is specified via emissions and receptions of events called signals (pure or valued) that have an inherent *status* of *absence* or *presence* in any reaction. In the case of Esterel the value of a signal can be modified only in case of presence but is persistent: It can be read when the signal is absent. Things are different in the *data-flow* paradigm, dedicated to intensive computations on data: Specifications are systems of equations that specify how the values of variables [3] are computed. Nevertheless variables also have a *status* of presence/absence [4]. Furthermore an absent variable has *no significant value.* We denote here absence by a special value [5] $\star$, like in [10]. A data-flow execution exhibits this special value, as shown on Fig. 1(b).

[2] [1] uses a flow semantics, but an operational one seems to be more intuitive.
[3] Lustre has *variables* but Signal has *signals* like Esterel: We choose to use the term "variable" for the whole data-flow paradigm.
[4] E.g. the Lustre equation `y = x when x>0` induces that y is absent if x is negative.
[5] Traditionally $\perp$ in the Signal context.

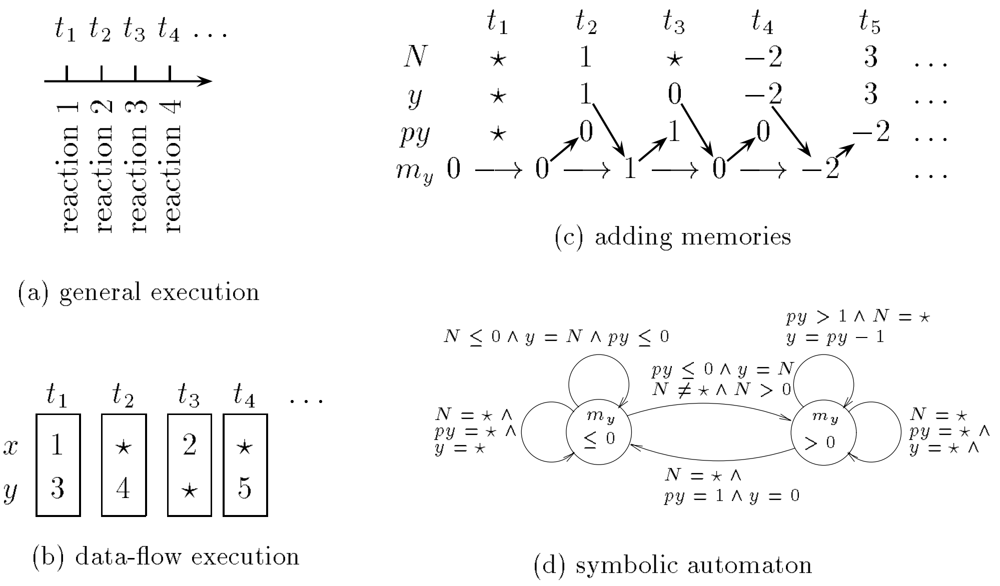


Fig. 1. synchronous models

2.2 Data-Flow Operational Semantics and Notations

Let us precise where does absence take place in data-flow operational models, using the very general Symbolic Labeled Transitions Systems (SLTS, see [10] for more formal details). A SLTS is built over a set of variables S of domain D and a set of persistent memories M (conventionally $m_x \in M$ is the memory associated to $x \in S$). Note that a variable has a status while a memory is by essence always present. A *valuation* $V : S \to D \cup \{\star\}$ represents a reaction of the system. A *state* is a valuation of memories $E : M \to D$. A SLTS contains an initialization predicate $\mathcal{I}$ (initial state), a memorization predicate $\mathcal{M}$ and a combinational (without state) predicate $\mathcal{C}$. $\mathcal{M}$ handles the state of the system, invisible from the outside. $\mathcal{C}$ interfaces the system with its environment and specifies what occurs inside a reaction. SLTS parallel composition is standard.

2.2.1 The Signal *Case*

Recall that the Signal kernel contains a parallel composition operator `|`, a delay operator `$`, plus operators `when`, `default` and functions (see their syntax on Fig. 2). Only the delay operator involves a memorization part. The equation `py := y $ 1 init v0` is represented by the SLTS:

$$(1)\, \mathcal{I} : (m_y = v_0) \quad , \quad \mathcal{C} : (py = \star \Leftrightarrow y = \star) \quad , \quad \mathcal{M} : \begin{cases} m'_y = \begin{cases} y \textbf{ if } y \neq \star \\ m_y \textbf{ else} \end{cases} \\ \text{if } py \neq \star \textbf{ then } py = m_y \end{cases}$$

Other *combinational* Signal operators induce a SLTS reduced to a predicate $\mathcal{C}$, as shown on Fig. 2. The Signal parallel composition corresponds to SLTS composition. Note that absence occurs only in sets of valuations models of the combinational part, that label transitions of the underlying symbolic automaton (see e.g. Fig. 1(d)). Such an automaton is a classical verification model: Its extension with $\star$ prevents from using standard tools. Accordingly the boolean verification tool SIGALI [9] dedicated to SIGNAL specifications uses an encoding of the three values $\{\star, \mathit{true}, \mathit{false}\}$ into {0,1,-1}, yielding computations in $\mathbb{Z}/3\mathbb{Z}$ by means of an extension of BDD to three values. Unfortunately this original technique does not scale to infinite domains.

Example 2.1 Let y be a counter initialized with an input variable N: while positive it decreases and is re-initialized with N after it has reached 0:

```
py := y $ 1 init 0 | y := N default py - 1 | N ^= when py <= 0
```

These equations are represented by the SLTS containing predicates $\mathcal{I}$ and $\mathcal{M}$ shown on Eq. (1) where $v_0 = 0$, and the following combinational part $\mathcal{C}$:

$$py = \star \Leftrightarrow y = \star \ \wedge \ y \neq \star \Leftrightarrow (N \neq \star \vee py \neq \star) \ \wedge \ N \neq \star \Rightarrow y = N$$
$$\wedge \ (N = \star \wedge py \neq \star) \Rightarrow y = py - 1 \ \wedge \ N \neq \star \Leftrightarrow (py \neq \star \wedge py \leq 0)$$

A possible execution is given Fig. 1(c) (arrows show links between variables and memories). The associated symbolic automaton is given Fig. 1(d). $\diamond$

$$\texttt{y := g(x1, ..., xn)} \rightsquigarrow \begin{array}{l} y = \star \Leftrightarrow x_1 = \star \Leftrightarrow \ldots \Leftrightarrow x_n = \star \\ \bigwedge y \neq \star \Rightarrow \ \forall i, x_i \neq \star \wedge y = g(x_1, \ldots, x_n) \end{array}$$

$$\texttt{y := x when c} \rightsquigarrow c \neq \mathit{true} \Rightarrow y = \star \ \bigwedge \ c = \mathit{true} \Rightarrow y = x$$

$$\texttt{y := x default z} \rightsquigarrow x \neq \star \Rightarrow y = x \ \bigwedge \ x = \star \Rightarrow y = z$$

Fig. 2. $\mathcal{C}$ for SIGNAL combinational operators

3 Clocks

We explain in Sect. 3.1 what are clocks and how they are used to specify the control of data-flow systems. Then we present in Sect. 3.2 their formalization traditionally used when dealing with SIGNAL.

3.1 Data-Flow Clocks

Clock of a System, of a Component The sequences of instants that appear in models of execution given Fig. 1 are called *time scale* in the imperative paradigm, and *clocks* in the data-flow one. More precisely, the *clock of a system* is the set of its instants of reaction (Fig. 1(a)). This notion is classical outside the synchronous paradigm (just consider circuits): A clock triggers the activation of a periodic system. Because there is no reason why concurrent components of a system should perform their computations at the same time, they must have distinct activations, so the system should contain several clocks: It is the base of the *multi-clock* approach, opposed to the monolithic mono-clock one.

Clock of a Variable Let us examine now how clocks are communicated to components. In the case of LUSTRE[6] the activation of a component is triggered by the *presence* of at least one of its input variable, say y. The clock of the component corresponds exactly to the set of instants where y is present: This set is by extension called the *clock of* y, denoted by $\widehat{y}$ in the SIGNAL context. Note that the potential absence of variables is now justified a posteriori: A variable has the double identity of a data-flow conveyer and (via its clock) of a sporadic event used to *control* behaviors like in the imperative paradigm.

Clocks and Control The SIGNAL philosophy emphasizes this last point: Clocks are fundamentally the main way the programmer has to specify the *control* of its specification[7], indicating the instants when some computations take place. Clocks are very widely used in SIGNAL: Any object related to a computation (e.g. an expression, a data dependency) is associated a clock which activates it. An intuitive case is the clock $\widehat{y}$ of a variable y, which triggers the computation of y. Following this principle, the executable code associated to a system contains *tests* over clocks; For instance "at instant t, `if` t belongs to clock h `then` activate computations associated to h".

3.2 The SIGNAL *Clocks Formalization*

Clock Algebra In the SIGNAL context relations over clocks are described using the *clock algebra* (denoted here by $\mathcal{H}$). Clocks were defined in Sect. 3.1 as sets of instants; The clock algebra accordingly uses set notations:

$$\mathcal{H} = \langle U, \cap, \cup, \setminus, \mathbb{O} \rangle$$

where U is a reference set of instants and $\mathbb{O}$ is the empty clock. Given a set of *clock variables* $\mathcal{K}$ interpreted as subsets of U (containing e.g. $\widehat{x}$), $\mathcal{H}$ can represent relations like $\widehat{x} = \widehat{y} \cup \widehat{z}$. We also use the set inclusion operator $\subseteq$.

Expressiveness Even if in their very first definition clocks are subsets of the time scale that indexes executions (see Fig. 1(a) and 1(b)), relations over clocks do not describe such executions (that are sequences of valuations) but only *sets of valuations/reactions*. Indeed, consider the equation $\widehat{x} = \widehat{y}$. It is true of an execution if, for any indexing instant t, x and y are both present or absent in the reaction that occurs at t. In other words, $\widehat{x} = \widehat{y}$ is true if in any reaction/valuation of the execution, x and y are both present or absent: The reference to time has disappeared and so did the order of valuations.

[6] The SIGNAL mechanism is more general, see [10] for a detailed discussion.
[7] Recall that data-flow equations offer no control structure like in the imperative paradigm; The plain execution of equations is very inefficient.

The expressiveness of relations over clocks is then purely *combinational.* The equation $\widehat{x} = \widehat{y}$ denotes the set of valuations V s.t. $V(x) = \star \Leftrightarrow V(y) = \star$.

Practical Encoding into the Propositional Calculus This set of valuations can be described equivalently by associating to clocks $\widehat{x}$ and $\widehat{y}$ the propositional variables b_x and b_y and by considering the boolean equation $b_x \Leftrightarrow b_y$, which describes the set of distributions $\delta : \{b_x, b_y\} \mapsto \{0,1\}$ where $\delta(b_x) = 0 \Leftrightarrow \delta(b_y) = 0$. We just have to interpret $\delta(b_x) = 0$ (resp. 1) as "x is absent (resp. present)". More generally [2] proposes a correspondence between $\mathcal{H}$ and boolean functions. We prefer the propositional calculus *PC* like [10]. The encoding is very simple: Each variable $k \in \mathcal{K}$ is associated a propositional variable b_k; Each set operator is associated the logical operator corresponding to its characteristic function. Informally consider the following example:

$\mathcal{H}$	U	$\mathbb{O}$	$\widehat{x} \cap \widehat{y} = \widehat{z} \cup \widehat{w}$	$\widehat{x} \setminus \widehat{z} = \widehat{y}$
PC	*true*	*false*	$b_x \wedge b_y \Leftrightarrow b_z \vee b_w$	$b_x \wedge \neg b_{\widehat{z}} \Leftrightarrow b_y$

Thanks to this encoding, the executable code handles clocks as propositional variables and not sets of instants[8]. A clock has a value *true* or *false* in a reaction, and tests over clocks mentioned in Sect. 3.1 are nothing but the test of a boolean variable: "`if` t belongs to h `then`" is encoded by "`if` $\mathtt{b_h}$ `then`".

4 Principles of the Signal Clock Calculus

The compilation process of synchronous languages is not limited to code generation: Some analyses are first applied to determine if the specification is indeed executable. Let us mention the LUSTRE [6] and ESTEREL [4] causality analyses, the LUSTRE [6] and LUCID SYNCHRONE [5] clock analyses and the ESTEREL constructive analysis [4]. The SIGNAL compilation process contains one major analysis called *clock calculus* [1,2] from which code generation *and* causality analysis [1] directly follow. As a consequence the clock calculus contains various aspects, which makes it very rich but difficult to explain.

The calculus applies to the *synchronizations* of a specification, presented in Sect. 4.1. It synthesizes a control structure from which single loop code directly follows (examples of control structures inferred from synchronizations are given in Sect 4.2). Its core is a constructive decision procedure which determines if a specification is *endochronous* (Sect. 4.3). We describe in Sect. 4.4 the data structures and algorithms, and their implementation in Sect. 4.5.

4.1 Synchronizations of a SIGNAL *Specification*

A SIGNAL equation specifies a relation 1. over the values of present variables 2. over the *status* of variables. For example `y := x default z` states that

[8] Note that LUSTRE clocks of variables are directly particular boolean variables of specifications, with the drawback that the notion of clock is recursive.

y is present iff x is present or z is present. This statement describes the *synchronizations* of the equation, or a *relation over clocks*. It can be described equivalently using the clock algebra (e.g. $\widehat{y} = \widehat{x} \cup \widehat{z}$) or Signal high-level operators (e.g. `y ^= x ^+ z`). But clocks of variables are not sufficient: The synchronizations induced by the under-sampling `when` operator involve the *value* of a boolean variable, `y := x when c` states that y is present iff x and c are present and c has value *true*.

It is therefore necessary to introduce a new kind of clock that deals with boolean values. Such a *condition-clock* is denoted by $[c] \in \mathcal{K}$ (`when c` in Signal), meaning the set of instants when c is present with value *true* ($[\neg c]$ corresponds to value *false*). $[c]$ is said to be obtained by *under-sampling* (or *extraction*) of $\widehat{c}$ by the condition c. Signal synchronizations are shown on Fig. 3. Expressed in the clock algebra they form a system of equations called *clock system/equations*. The braced equations are optional: They correlate $[c]$, $[\neg c]$ and $\widehat{c}$ by specifying that when c is present, c takes either the value *true* or *false* (in short that $([c], [\neg c])$ is a partition of $\widehat{c}$).

P	synchronizations of P in Signal	synchronizations of P in $\mathcal{H}$
`y := g(x`$_1$`,...,x`$_n$`)`	`y ^= x`$_1$ `| ... | y ^= x`$_n$	$\widehat{y} = \widehat{x_1} \ldots \widehat{y} = \widehat{x_n}$
`py := y $1 init v0`	`py ^= y`	$\widehat{py} = \widehat{y}$
`y := x when c`	`y ^= x ^* when c`	$\widehat{y} = \widehat{x} \cap [c]$ $\left.\begin{array}{l}[c] \cup [\neg c] = \widehat{c} \\ [c] \cap [\neg c] = \mathbb{O}\end{array}\right\}$
`y := x default z`	`y ^= x ^+ z`	$\widehat{y} = \widehat{x} \cup \widehat{z}$

Fig. 3. synchronizations

Let us precise now the link between the propositional encoding of synchronizations and boolean models extended with $\star$ like $\mathbb{Z}/3\mathbb{Z}$. The encoding of relations over clocks of variables into PC (Sect. 3.2) is very intuitive because the status of a variable is clearly boolean. A condition-clock $[c]$ is similarly encoded into a propositional variable $b_{[c]}$ but distributions indicate both the status and the value of c, e.g. $\delta(b_{[c]}) = 0$ means that c is either absent or present with value *false*. As explained in Sect. 2.1 the semantical computation domain is $\{\star, \mathit{true}, \mathit{false}\}$ but the introduction of condition-clocks makes possible an encoding into the only *two values* of the propositional calculus. This trick is nothing but the classical encoding of a tri-values logic into a boolean algebra using auxiliary variables[9]:

[9] Note that one of the variables $b_{[c]}$, $b_{[\neg c]}$ and $b_{\widehat{c}}$ is redundant with others: The clock calculus uses only $b_{[c]}$ and $b_{[\neg c]}$.

$$b_c = b_{[c]} = b_{[\neg c]} = 0 \rightsquigarrow c = \star \qquad b_c = 1, \begin{cases} b_{[c]} = 0,\ b_{[\neg c]} = 1 \rightsquigarrow c = \mathit{false} \\ b_{[c]} = 1,\ b_{[\neg c]} = 0 \rightsquigarrow c = \mathit{true} \end{cases}$$

In other words thanks to condition-clocks the clock algebra can describe the whole boolean combinational part of Signal specifications[10] and similarly any boolean combinational predicate $\mathcal{C}$ (Sect. 2.2). It means that, by applying such an encoding to $\mathcal{C}$, Sigali could be implemented using standard stuff.

From now, we assimilate clock variables (resp. relations over clocks) and their correspondent propositional variables (resp. boolean equations).

4.2 From Synchronizations to Control Structure: Main Ideas

As explained in Sect. 3.1 the Signal philosophy strongly emphasizes that clocks indicate the control of data-flow specifications. Accordingly the control-flow of the target executable code is synthesized from relations over clocks, or synchronizations. Since clocks can describe only what occurs inside a reaction, we address only the control-flow corresponding to combinational instructions (see Sect. 5 for a complete example). We give here an intuition of the main ideas (they will be detailed in the following sections), illustrated by examples. The first example concerns the equation `y := x default z`, the second one the counter of Ex. 2.1, the third one equations `y := x + 2 | z := y + 2 when y <= 0 | w := z when z < 0`. Their synchronizations are given respectively by Eq. (2), (3) and (4):

(2) $\widehat{y} = \widehat{x} \cup \widehat{z}$

(3) $\widehat{y} = \widehat{py} \quad \widehat{y} = \widehat{N} \cup \widehat{py} \quad \widehat{N} = [c]$

(4) $\widehat{y} = \widehat{x} \quad \widehat{z} = \widehat{y} \cap [c'] \quad \widehat{w} = \widehat{z} \cap [c'']$

where c, c' and c'' abstract respectively conditions $py \leq 0$, $y \leq 0$ and $z < 0$. The corresponding code is given on Fig. 4(a), 4(b), and 4(c) respectively. It appears clearly that the control-flow is materialized by tests over clocks[11].

Clocks are fundamentally used as *r/w guards* for the *value* of variables. Any access to the value of a variable y is embedded into a test over $\widehat{y}$: On Fig. 4(c) `z := y + 2` (line 5) is guarded by a test on $\widehat{y}$ (y is read) and on $\widehat{z}$ (z must be computed), lines 1 and 4. Testing the value of clocks implies that clocks must be chosen a *definition*: It is the main goal of the clock calculus[12]. Some definitions are quite intuitive: The choice for the definition of `b`$_{\mathtt{y}}$ line 1

[10] For example if x, y and z are boolean variables the equation `y := x default z` can be encoded into the boolean system $\widehat{y} = \widehat{x} \cup \widehat{z}$, $[y] = [x] \cup ([z] \setminus \widehat{x})$.

[11] The connection between control-flow (clocks) and data-flow (computations of values of variables) is determined syntactically. For example the parsing of equation `y := x default z` attaches to $\widehat{y}$ the need of computing y, to $\widehat{x}$ the definition of y by x, and to $\widehat{z} \setminus \widehat{x}$ the definition of y by z. Hence the code of Fig. 4(a) from line 2. So, once the control-flow has been inferred from synchronizations, the code directly follows.

[12] The definition of a condition-clock is not dealt with by the calculus, since not specified by synchronizations (see also Sect. 4.3). See for example the definition of `b`$_{\mathtt{[c]}}$ Fig. 4(b) line 2 (in fact the Signal compiler suppresses this useless variable).

```
(1) b_y := b_x or b_z;
(2) if b_y
    then // compute y
(3)   if b_x then y := x
      end if
(4)   if (b_z and not b_x)
      then y := z
      end if
    end if
```

(a)

```
(1) if b_y
    then
(2)   b_[c] := py <= 0
(3)   b_N := b_[c] ;
      if b_N
      then y := N
      else y := py - 1
      end if
(4) end if
```

(b)

```
(1) if b_y
    then
(2)   y := x + 2
(3)   b_z := y <= 0
(4)   if b_z
(5)   then z := y + 2
           b_w := z < 0
           if b_w
           then w := z
           end if
      end if
    end if
```

(c)

Fig. 4. control structures inside reactions

Fig. 4(a) directly follows from transforming the *equation* (2) into an *oriented* definition. A glance at the counter shows that in the general case extracting definitions from *relations* over clocks is more complex: Eq. (3) implies that $\widehat{y}$ is constrained by the recursive equation $\widehat{y} = \widehat{N} \cup \widehat{y}$; Moreover $\widehat{y}$ is in fact not given a definition since considered as an *input* clock, line 1 Fig. 4(b).

Finally the knowledge of some clock *inclusions* and *equivalences* is used to optimize the control structure by avoiding useless tests at execution time. Synchronizations of Eq. (4) imply inclusions $\widehat{z} \subseteq \widehat{y}$ and $\widehat{w} \subseteq \widehat{z}$ (for instance $\mathtt{b_z}$ cannot be true if $\mathtt{b_y}$ is not also true). Consequently on Fig. 4(c) tests over $\mathtt{b_y}$, $\mathtt{b_z}$ and $\mathtt{b_w}$ are *nested.* Tests are also *factorized*: $\mathtt{b_y}$ is tested only once while y is written (line 2) and read (line 5). Additionally the *number of clock variables* is optimized: Since synchronizations imply $\widehat{y} = \widehat{x}$ there is no need for some variable $\mathtt{b_x}$.

4.3 Endochrony: From Equations to Definitions

A component is *endochronous* if it can be executed in an asynchronous environment which provides only values of inputs, with no information about their status (see [3] for more details). The component has no way to test deterministically the status of its inputs (intuitively because such a test is blocking): It cannot test deterministically more than one input clock. So an endochronous component must own an identified *master clock* (which is nothing but its activation clock), which is the only input clock of the executable code. Hence all other (necessarily slower) clocks must be recursively defined from the master.

LUSTRE components are endochronous by construction: A reference master clock is given and any other clock is either an already existing clock, or a clock defined functionally by the *under-sampling* of an existing clock. SIGNAL is more general: Endochrony is not ensured. Moreover as explained in Sect. 4.1 Clocks are linked together not only through under-sampling but also by any combination of operators $\cup$, $\cap$ and $\setminus$. Finally some fundamentally *relational* synchronizations cannot be transformed into functions.

Example 4.1 The code of Fig. 4(a) cannot be executed deterministically and does not represent an endochronous component: The greatest clock $\widehat{y}$ is not a master, since computed as a function of the input clocks $\widehat{x}$ and $\widehat{z}$. The master clock on Fig. 4(b) and 4(c) is $\widehat{y}$. Consider the equation `y ^< x`. It specifies the clock inclusion $\widehat{y} \subseteq \widehat{x}$ but does not indicate how $\widehat{y}$ is computed from $\widehat{x}$. $\diamond$

To check that a specification is endochronous, the calculus must infer a master clock (if it exists) and compute for any other clock a definition which is a *function* of other clocks (if possible): If it succeeds the specification is declared endochronous. To do so it makes the clock system *triangular*[13] (this process is called *resolution*). It uses a particular strategy that focuses on under-sampling, whose importance appears clearly in LUSTRE. A condition-clock like $[c]$ depends on the status and value of c, but this value is unknown since not defined by synchronizations. So $[c]$ cannot be given a definition and may carry any value: Condition-clocks play the role of special *parameters* present in the initial system. All other clock variables play the role of *variables*: From now only these ones are called clock variables. So the strategy consists in defining clock variables by a *function of condition-clocks. Inclusions* $[c] \subseteq \widehat{c}$ and $[\neg c] \subseteq \widehat{c}$ are particularly meaningful[14] and widely exploited.

4.4 Data Structure and General Algorithms

From now we enter into technical details taken from [1,2,10]. Because the resolution exploits under-sampling inclusions, the used data-structure is based on *trees* whose nodes are clock variables and s.t. for two nodes n_1 and n_2, "n_1 is a descendant of n_2" means that "the clock n_1 is included into the clock n_2". Its goal is to reduce the size of definitions and to represent both the shape of a triangular system and the control structure of the specification. We distinguish two types of definitions for clocks: *Syntactical* definitions appear in code (e.g. definition of $\widehat{y}$ by $\widehat{x} \cup \widehat{z}$ on Fig. 4(a)) while *semantical* definitions characterize a clock h by a function $def(h)$ of condition-clocks (e.g. assume given syntactical definitions $h_3 := h_1 \cup h_2$ and $h_4 := h_3 \cap h_1$ where $def(h_1) = [c_1]$ and $def(h_2) = [c_2]$, then $def(h_4) = [c_1]$). We denote by $var(h)$ the set of condition-clocks on

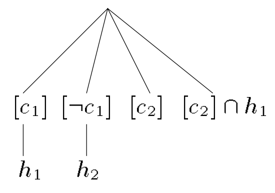
[13] The notion of triangular system is very generally defined in the theory of boolean equations. Solving a consistent boolean system amounts to compute all solutions of $f(x_1, \ldots, x_n) = 0$. Instead, one can give to this equation a *general solution* under *parametric form*, given by a family of boolean functions $\{\psi_i(p_1, \ldots, p_n)\}_{i=1\ldots n}$ where p_i are *parameters* that can take any value in $\{0, 1\}$. The *triangular form* of the general solution is: $x_1 = \psi_1(p_1)$ $x_2 = \psi_2(p_1, p_2)$ $\ldots$ $x_n = \psi_n(p_1, \ldots, p_n)$ where ψ_is are *definitions*. An *irredundant solution* is obtained by constraining parameters by a *constraint system* C which ensures an injection between vectors $p_1, \ldots, p_n$ and solutions of the equation.

[14] For instance adding $[c] \subseteq \widehat{py}$ to Eq. (3) suppresses the recursive constraint on $\widehat{y}$ since $\widehat{N} \cup \widehat{py}$ is now trivially equivalent to $\widehat{y}$.

which *def*(h) depends (e.g. $\{[c_1], [c_2]\}$ for h_3). To simplify we consider only syntactical definitions of the kind h_1 *op* h_2 where $op \in \{\cup, \cap, \setminus\}$. Semantical definitions will be of fundamental importance in Sect. 4.5.

4.4.1 Clocks Layout in Trees

An inclusion induced by under-sampling is represented by an intuitive basic tree (called *partition tree*) represented on Fig. 6(a). Lustre synchronizations can be directly represented by a tree containing only under-samplings and which root is the master clock. Signal trees must also represent inclusions induced for instance by defining a clock h by $[c_2] \cap h_1$ (see Fig. 5(a)). The principle is not to represent all inclusions (here $h \subseteq [c_2]$ and $h \subseteq h_1$) but only those which indicate the corresponding code structure. A *depth first traversal* (dft) of the tree exhibits the order of computations induced by definitions. An example of control structure directly inferred from a tree is given Fig. 5. Note that nested tests follow exactly the tree structure and that the partition $([c_1], [\neg c_1])$ corresponds to an `if then else` statement.



```
compute c1;
if c1 then compute h1; if h1 then...  endif
     else compute h2; if h2 then...  endif
endif /* if c1 */
compute c2; if c2 then...  endif;
if c2 and h1 then...  endif
```

(a) (b)

Fig. 5. control structure inferred from a tree

4.4.2 Construction of Trees

The initial step is to build all partition trees: Any clock variable is root of such a tree or of a tree reduced to a root. Then the algorithm iterates the following process: 1. choice from synchronizations of a syntactical definition for some clock variable h_3 of the type h_1 *op* h_2 s.t. h_1 and h_2 belong to a tree a'; 2. computation of *def*(h_3) and insertion of the sub-tree a whose root is h_3 into a' (it is a *fusion*, see Fig 6(b)). Hence, the root excluded, each node corresponds either to a condition-clock (never defined) or to a clock variable which has been given a definition. The root r is a temporary "local" master clock: *def*(r) = *true*. All the problem is to find a convenient placement for h_3. Two ideas are important here.

Sub-tree Properties A sub-tree a contains a set of condition-clocks from which other clocks can be defined (we call it its *context*) and must respect two principles: 1. any clock h in a is such that *var*(h) belongs to this context (*locality criteria*); 2. a dft finds all variables of *var*(h) before h (this ensures a triangular shape for the clock system, i.e. the respect of dependencies in computations).

Branching of Nodes Since h_1 and h_2 belong to the same sub-tree a' they share the same context and have a common ancestor h (called their *branching*), which is of particular interest: If $def(h_3) = def(h_1)\ op\ def(h_2)$ it is easy to check that the insertion of h_3 as the *right-most child* of h respects the above two principles (whatever *op* be, the inclusion $h_1\ op\ h_2 \subseteq h$ holds), see Fig 6(b). In fact, some deeper insertions may be correct, leading to more nested control structure and more inclusions made explicit: We refer to Sect. 4.5.

If a single tree is obtained the specification is declared endochronous, its master clock being the root. The algorithm progresses by computing definitions $h_1\ op\ h_2$ s.t. h_1 and h_2 belong to the same tree. If no such expression can be found in synchronizations, another compilation module applies them some rewriting rules. The used rewriting system is ad-hoc and in particular incomplete. If no convenient expression appears after rewriting, the calculus stops and the specification is declared not to be endochronous (while it may be so).

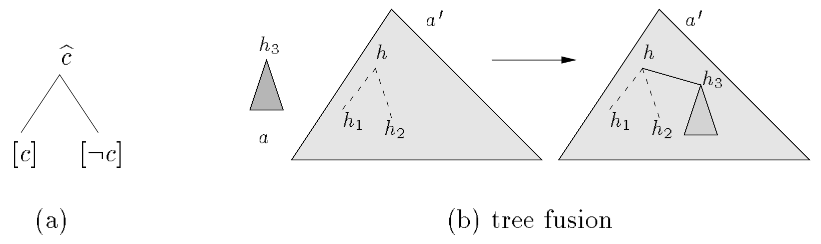


Fig. 6. construction of trees

4.4.3 Proof of Boolean Properties

Synchronizations are represented by a set of equivalence classes. Assume that two clocks of the same class h_1 and h_2 have been given a definition. Since h_1 and h_2 are equivalent, the equivalence $def(h_1) = def(h_2)$ must be verified for synchronizations to be coherent. The calculus can only check that this equivalence is a logical consequence of the set of already computed definitions $def(h)$ [15]. If the proof fails, the equivalence is reported to the user as a *clock constraint* which makes the specification not endochronous.

4.5 Implementation of Definitions and Insertions

Definitions $def(h)$ are implemented by BDD. Condition-clocks $[c]$ and $[\neg c]$ are represented by complementary elementary BDD (Fig. 7(a)), which implements their partition of $\widehat{c}$ provided $\widehat{c}$ is represented by the BDD **1** (**1** "defines" any root r of a tree). The set $var(def(h))$ is the corresponding BDD support.

[15] The still relational part of synchronizations cannot be exploited since the algorithm only considers semantical definitions, thus the proof is necessarily incomplete.

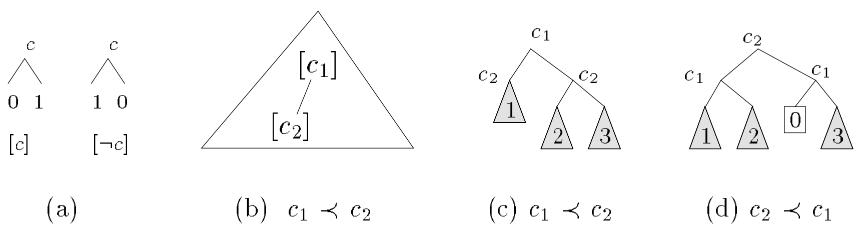


Fig. 7. BDD and trees

The tree and BDD properties are exploited to uniquely characterize $def(h)$ from its syntactical definition. For any clock k in a tree a of root r, let $f(k)$ denote the father clock of k. We define the *enlarged definition* of k as [2]:

$$def_e(k) = def(h) \wedge def(f(h)) \wedge \ldots \wedge def(f^i(h)) \wedge def(r)$$

Because of inclusion $k \subseteq f(k)$, $def_e(k)$ is equivalent to $def(k)$ but takes into account inclusions such as $[c] \subseteq \widehat{c}$. Consider again a clock h_3 defined by $h_1 \ op \ h_2$ where the branching of h_1 and h_2 is h. We have:

$$def_e(h_1) = \overbrace{def(h_1) \wedge def(f(h_1)) \wedge \ldots \wedge def(f^i(h_1))}^{F_1} \wedge def_e(h)$$

If we define similarly F_2 for h_2 then we can indeed insert h_3 as a child of h and give it the *factorized* definition $def(h_3) = F_1 \ op \ F_2$. In fact [2] shows that there exists a unique *deepest* clock h' descendant of h and a unique expression F s.t. h_3 can be defined by F and inserted under h': 1. $def_e(h_1) \ op \ def_e(h_2) = F \wedge def_e(h')$ and 2. the two sub-tree properties of Sect. 4.4.2 are verified. To find h', one just need to consider the greatest node n (for a dft) that appears in $var(def_e(h_1) \ op \ def_e(h_2))$ and to follow the path from n to h while condition 1. is verified.

Since BDD have canonical forms and the pair (h', F) is unique, trees are a canonical representation of synchronizations, very efficient for two reasons. Firstly thanks to the tree structure only *parts* of the boolean system are represented (definitions $def(h)$) and considered at each operation (enlarged definitions $def_e(h)$). Moreover the order $\prec$ of variables in BDD supports is not chosen randomly but incrementally determined during trees construction by the dft order. As a consequence *BDD are naturally small.* For instance the tree of Fig. 7(b) induces the order $c_1 \prec c_2$, meaning "compute first c_1 then c_2". This order leads to a smaller BDD (Fig. 7(c)) than the reverse order (Fig. 7(d)).

5 Conclusion

This paper gives an overview of the Signal compilation process under its main aspects. Before entering into technical details it focuses on fundamental notions that must be understood to fully appreciate principles of the clock calculus. It explains in particular what are clocks, how they are formalized as sets of instants but used as propositional variables to encode the combinational boolean part of specifications, and what is the difference with the encoding into $\mathbb{Z}/3\mathbb{Z}$ (traditionally misunderstood).

The calculus has remained stable since 1995. It could be intrinsically improved: LUSTRE-like *assertions* should be introduced and algorithmically considered not as constraints to prove but as hypotheses for proofs mentioned in Sect. 4.4.3; These proofs could take into account relational aspects and not only functional ones. The other synchronous compilation process must also be examined: A deep comparison with the LUSTRE [6] and LUCID SYNCHRONE [5] approaches must be done (for example consequence of simpler synchronizations on the complexity and modularity of analyses); New ESTEREL techniques that infer a control-flow from an event-graph must also be considered. We make here a few remarks about the LUSTRE compilation into automata.

Contrary to SIGNAL, LUSTRE synthesizes the control-flow of the executable code from the evolution of the specification *boolean memories* [7]. So code optimization consists in testing these values only when necessary: Such tests are suppressed by statically computing these values. The generated code is structured like an *automaton*, whose states are values of boolean memories and transitions represent combinational reactions specific to the source state. It implies that the reachable boolean state space is explored. On the contrary, the SIGNAL process is bounded to the purely combinational analysis of relations over clocks, so can only structure the control flow *inside reactions*. States are not distinguished and the generated code is an *single loop* whose body represents all possible reactions that can occur in any state of the system. For instance the complete code for the counter contains an initialization of m_y by 0, followed by a single loop which embeds the code of Fig. 4(b) where memories have been inserted: Addition of `py := m`$_y$ before line 2 and of `m`$_y$ `:= y` before line 4.

It is well known that the single loop code is very compact, while the size of an automaton is exponential (LUSTRE algorithms essentially aim at reducing this size [7]). But the automaton structure is very well adapted to verification purposes (for which state explosion is a common problem). LUSTRE verification tools take as input the *interpreted automaton* synthesized by the compiler. Such an automaton looks like the one of Fig. 1(d) with a major great particularity: Transitions are labeled by *formal imperative-like expressions*. These expressions are similar in nature to the code encountered in the SIGNAL loop body: Absence is also compiled through *tests* over *clocks* [11]. For instance the left to right transition could be labelled by `if b`$_y$ `then if py <= 0 then y := N`. In this way LUSTRE avoids the use of models extended with $\star$ and commonly addresses numerical infinite domains (e.g. NBAC tool [8]). So the LUSTRE approach could bring to SIGNAL a standard model for verification hence a basis for numerical tools, following the work of [10].

Acknowledgements: Thanks are due to anonymous referees for their useful comments, some of them were not taken into consideration for lack of space.

References

[1] P. Amagbégnon, L. Besnard, and P. Le Guernic. Implementation of the data-flow synchronous language SIGNAL. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 163–173, La Jolla, California, 18–21 June 1995.

[2] Tochéou Pascalin Amagbegnon. *Forme canonique arborescente des horloges de Signal*. PhD thesis, Université de Rennes I, IFSIC, December 1995. In french.

[3] A. Benveniste, B. Caillaud, and P. Le Guernic. From synchrony to asynchrony. In *CONCUR'99*, volume 1664 of *Lecture Notes in Computer Science*, pages 162–177. Springer, 1999.

[4] G. Berry. *The Constructive Semantics of Esterel*. Draft book, July 1999. current version 3.0.

[5] P. Caspi and M. Pouzet. Synchronous Kahn networks. In *International Conference on Functional Programming*, pages 226–238, 1996.

[6] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[7] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming, Passau, August 1991, PLIP'91*, volume 528 of *LNCS*, pages 207–218, Passau, Germany, 1991. Springer Verlag.

[8] B. Jeannet. Dynamic partitioning in linear relation analysis. Application to the verification of synchronous programs. Technical Report RS-00-38, BRICS, 2000. To appear in Formal Methods in System Design.

[9] M. Le Borgne, H. Marchand, E. Rutten, and M. Samaan. Formal verification of signal programs: Application to a power transformer station controller. In *Proceedings of the Fifth International Conference on Algebraic Methodology and Software Technology AMAST'96*, pages 271–285, Munich, Germany, July 1996. Springer-Verlag, LNCS 1101.

[10] M. Nebut. *Réactions synchrones : spécification et analyse*. PhD thesis, Université de Rennes 1, IFSIC, 2002. order 2741, In french.

[11] P. Raymond. *Compilation efficace d'un langage déclaratif synchrone : le générateur de code* LUSTRE-*V3*. PhD thesis, Institut National Polytechnique de Grenoble, 1991. In french.

[12] *Synchronous Languages, Applications, and Programming*, volume 65, Grenoble, France, april 2002. Electronic Notes in Theoretical Computer Science.