

Implementation of the data-flow synchronous language SIGNAL

Tochéou Pascalin Amagbegnon, Loïc Besnard, Paul Le Guernic

▶ **To cite this version:**

Tochéou Pascalin Amagbegnon, Loïc Besnard, Paul Le Guernic. Implementation of the data-flow synchronous language SIGNAL. ACM SIGPLAN Notices, 1995, 30 (6), pp.163-173. 10.1145/223428.207134 . hal-00544128

HAL Id: hal-00544128
https://hal.science/hal-00544128

Submitted on 7 Dec 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Implementation of the Data-flow Synchronous Language SIGNAL

Pascalin Amagbégnon Loïc Besnard Paul Le Guernic
e-mail: {Pascalin.Amagbegnon,Loic.Besnard,Paul.Leguernic}@irisa.fr
phone: (33)-99-84-74-36
IRISA-INRIA
Campus de Beaulieu
35042 Rennes CEDEX
FRANCE

Abstract

This paper presents the techniques used for the compilation of the data-flow, synchronous language SIGNAL. The key feature of the compiler is that it performs formal calculus on systems of boolean equations. The originality of the implementation of the compiler lies in the use of a tree structure to solve the equations.

1 Introduction

Traditionally, real-time systems have been programmed in imperative asynchronous languages like ADA, OCCAM or C together with some operating system facilities. But these tools are not satisfactory as there is considerable need of provably correct software and as systems become more and more complex.

To remedy the insufficiencies of the current tools, the *synchronous paradigm* has been proposed and developed in[4]. Its main hypothesis is that a) operators react instantaneously with their inputs (computations have zero duration), b) time is just a succession of events (no explicit reference to a notion of physical time). The validity of the synchrony assumption is thoroughly discussed in [5]. Let us point out briefly, some advantages a programmer can get from this simplifying hypothesis.

The assumption that an operator (e.g an adder) computes its outputs simultaneously with the occurrences of its inputs is a very useful approximation in many fields. As an example in the field of hardware synthesis, logic gates are supposed to compute their outputs synchronously in the first approximation. It makes the design of a circuit simple. Then, propagation time is taken into account and the maximum clock frequency allowed is calculated. Such a separation between pure functionality and execution time is now possible in software with synchronous languages: the programmer specifies a functionality of the program, and the compiler of a synchronous language handles execution time on a particular target processor.

The second assumption of the synchronous paradigm (no physical time) provides the programmer with a framework in which he/she can handle uniformly real-time constraints. As a matter of fact, real-time constraints are not always expressed in terms of milli- or micro-seconds. The statement "the train must stop within 30 meters" is a real-time constraint just as "the train must stop within 30 seconds". In a framework like ADA which possesses a notion of time only in terms of seconds (and not in terms of meters), those constraints will not be handled with similar programs. Hence the usefulness of the second synchrony hypothesis.

Four languages are built upon this synchronous paradigm: they differ mostly in the programming style they offer. ESTEREL[7] is an imperative synchronous language. SIGNAL[6] and LUSTRE[11] are data-flow languages. Finally in ARGOS[21], systems are specified with parallel and hierarchical automata. A summary of the synchronous approach can be found in [14].

This paper presents the techniques used in the compilation of the SIGNAL language which is a data-flow language. In this language, systems are specified with equations over *synchronized streams*. The declarative style of SIGNAL provides the programmer with the nice high-level constructs needed to describe a real-time system in terms of operators network, differential/difference equations. But, the higher the level of the language, the bigger the challenge to construct a compiler able to generate efficient executable code by using a "reasonable" amount of computing resources (memory, cpu-time).

SIGNAL's compilation is based on an abstract interpretation of each statement as a system of boolean equations. These equations express the synchronizations in the program. The compiler solves a system of boolean equations for each program in order to a) check the consistency of the synchronizations,

and, b) generate efficient code (silicon[3], sequential[20], parallel[9]).
In this paper, we mainly report the techniques used to analyze the boolean equations. In Section 2 we present the language SIGNAL: the basic objects, the boolean equations and the dependency graph associated to each SIGNAL program. Section 3 puts the emphasis on the system of boolean equations and introduces a hierarchichal representation for it. Finally we conclude with some experimental results which demonstrate the effectiveness of our approach.

2 The SIGNAL language

In this section, we present the basic objects of the language. We use sequences to describe the semantics of the statements. See [6] for more details on formal semantics of SIGNAL.

2.1 Signals and clocks

Signals. A signal X is a sequence $(X_t)_{t\in I}$ of values chosen in a domain D (the type of the signal). Integer, boolean, real are examples of signal types. The time index I is a *totally ordered* set of *instants*. We are interested in a discrete time model. So, instants are taken in a denumerable set. At any given instant t, a signal may be *present* or *absent* depending on whether or not, the instant under consideration belongs to I; a signal carries a value only when it is present.
Clocks. The set of instants at which a signal is present is its clock. So, the clock of a signal $(X_t)_{t\in I}$ is its time index I. Two signals always present at the same instants are said to be *synchronous*: they have the same clock. Thus, the clock of a signal X is the equivalence class of X for the *synchrony relation*; in that sense it is denoted $\widehat{X}$.
Notation. Following [8], the set theoretic operators for clocks, which are sets, are denoted $\wedge$ (intersection), $\vee$ (union) and $\setminus$ (set difference). We use $<op>$ to denote one of the three operators.

2.2 The kernel of SIGNAL

A statement in SIGNAL is an equation on signals; it is called a process. We give here the kernel operators; the full language features other operators which can be rewritten in terms of these kernel operators.
Functional expressions. The operators (e.g $+, -, *, and$) defined on basic data types (e.g booleans, integers) are canonically extended to sequences and consequently to signals. Let f be such an operator of arity n and let $(X_{1t})_{t\in I}, \ldots, (X_{nt})_{t\in I}$ be n sequences with the same time index I. The equation

$$\forall t \in I,\ Y_t = f(X_{1t}, \ldots, X_{nt})$$

is written in SIGNAL (see Figure 1)

```
Y := f(X1 ,... ,Xn)
```

The signals involved in that equation are required to have the same time index I: they must be synchronous. Thus, the definition of the signal Y implies the following equation on the clocks:

$$\widehat{Y} = \widehat{X_1} = \ldots = \widehat{X_n}$$

Reference to past values. We reference past values of a — discrete — signal with the "$" operator. The SIGNAL process

```
ZX := X $ 1 init v0
```

is the representation of the following equation on the sequences $(X_t)_{t\in I}$ and $(ZX_t)_{t\in I}$ defined on the same index I:

$$\forall t \in I,\ ZX_t = X_{t-1} \text{ and } ZX_1 = v_0$$

Here again, ZX is by definition synchronous with X: $\widehat{ZX} = \widehat{X}$. A timing diagram for this operator is depicted on Figure 2.
Downsampling. Given a signal U and a boolean-valued signal C (sometimes termed *condition*), the process

```
X := U whenC
```

defines the signal X which carries the same value as U when both U and C are present, and, C carries the value $true$. Before giving a formal definition of X, let us introduce some notations we will use frequently. We denote:

$$[C] = \text{the set of instants at which C carries the value } true$$

$$[\neg C] = \text{the set of instants at which C carries the value } false$$

When a boolean-valued signal occurs, it carries either the value $true$ or the value $false$. So the pair $([C], [\neg C])$ defines a partition of $\widehat{C}$ (the clock of C). This can be represented by:

$$\begin{cases} [C] \vee [\neg C] = \widehat{C} \\ [C] \wedge [\neg C] = \mathbb{0} \end{cases} \quad (1)$$

where $\mathbb{0}$ denotes the null clock, the empty set of instants.
With this notation, the signal `X := U whenC` is the sequence such that:

$$\begin{cases} \widehat{X} = \widehat{U} \wedge [C] \text{ the time index, the clock} \\ \forall t \in \widehat{X},\ X_t = U_t \end{cases}$$

$(X_t)_{t\in\widehat{X}}$ can be viewed as a subsequence of $(U_t)_{t\in\widehat{U}}$. See Figure 3 for a timing diagram.
Deterministic merge. Given two signals U and V, the process

```
X := U defaultV
```

defines the signal which carries the same value as U when U is present, or the same value as V when U is absent. It is

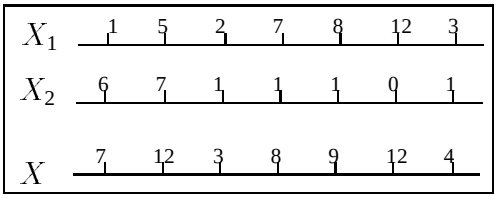


Figure 1: X := X_1 + X_2

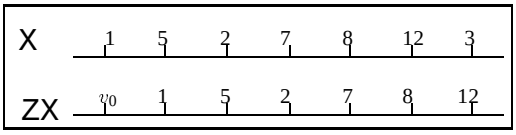


Figure 2: ZX := X $ 1 **init** v_0

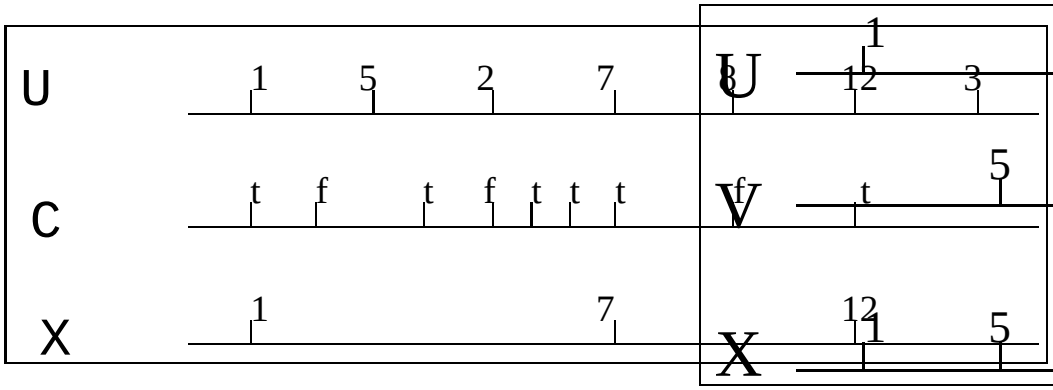


Figure 3: X := U whenC

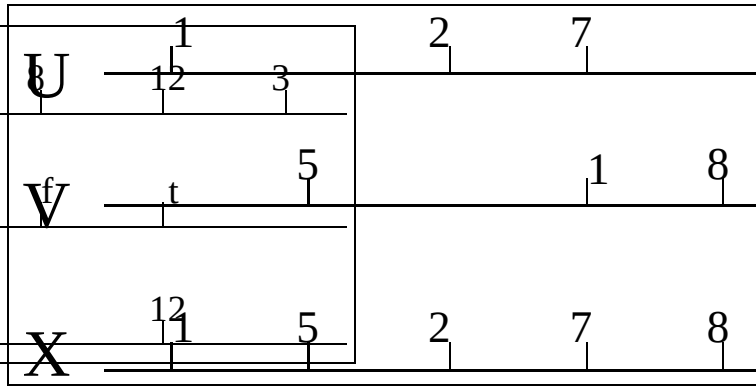


Figure 4: X := U defaultV

absent when both U and V are absent. It merges the flows U and V and puts a priority on U. A timing diagram is depicted on Figure 4. More formally:

$$\begin{cases} \widehat{X} = \widehat{U} \vee \widehat{V} & \\ \forall t \in \widehat{U} & X_t = U_t \\ \forall t \in \widehat{V} \setminus \widehat{U} & X_t = V_t \end{cases}$$

X_t is well defined for all $t \in \widehat{X}$, since any instant in $\widehat{X}$ belongs either to $\widehat{U}$ or to $\widehat{V} \setminus \widehat{U}$.

Composition of processes. The elementary processes we have presented till now may be composed with the commutative and associative operator "|". From an equational point of view, this operator is the union of two systems of equations. For example the SIGNAL process of the system:

$$\forall t \in I, \begin{cases} ZX_t &= X_{t-1} \\ X_t &= ZX_t + Y_t \end{cases}$$

is

```
(|  ZX  :=  X $ 1
 |  X   :=  ZX + Y
 |)
```

2.3 Extended language

For convenience, the full language SIGNAL offers many derived operators; they can be expressed in terms of the kernel operators. Here are some of them:

- the operator **event** : syntactically, the clock $\widehat{X}$ of a signal X is written **event** X; in fact, it is an abbreviation for the boolean signal defined by **event** X := (X = X); it is synchronous with X and carries the value $true$ each time it occurs; thus it can be identified with $\widehat{X}$ the clock of X;
- the unary when: for a condition C, the signal whenC is an abbreviation for C whenC; this signal carries the value $true$ each time it is present, and is synchronous with the clock $[C]$ which is the set of instants when the signal C is present and carries the value $true$; so the signal whenC and the clock $[C]$ can be unified;
- similarly, the signal when(**not** C) can be identified with the clock $[\neg C]$;
- the process **synchro** $\{X_1, \ldots, X_n\}$ in SIGNAL's syntax specifies the equality of the clocks of its operands i.e the equation $\widehat{X_1} = \widehat{X_2} = \ldots = \widehat{X_n}$.

2.4 System of boolean equations

It appears clearly that a system of boolean equations lies under each SIGNAL process. We hinted that during the presentation of the kernel of SIGNAL. We recapitulate these equations in Table 1.

At this stage, the main difference between SIGNAL and the classical data-flow languages [16] [25] is that in SIGNAL we manipulate *synchronized* data-flow by means of *clocks*. The main purpose of synchronized data-flow is that all the synchronizations (expressed in terms of equations over clocks) should be completely handled at compile time. For more details, see [17].

signal process	clock calculus	additional equations
Y:= f(X$_1$,...,X$_n$)	$\widehat{Y} = \widehat{X_1} = \ldots = \widehat{X_n}$	
ZX:= X$1	$\widehat{ZX} = \widehat{X}$	
X:= U defaultV	$\widehat{X} = \widehat{U} \vee \widehat{V}$	
Y:= X whenC	$\widehat{Y} = \widehat{X} \wedge [C]$	$[C] \vee [\neg C] = \widehat{C}$ $[C] \wedge [\neg C] = \mathbb{0}$

Table 1: From SIGNAL operators to boolean equations

2.5 Conditional dependency graph

A data dependency is associated with each process. A process is compiled into a graph representing the dependencies between signals. The edge $X \xrightarrow{k} Y$ connecting X and Y means that at each instant of the clock k, Y's value depends on X's value. Such a graph is constructed from the elementary processes as shown in Table 2.

process	dependency
X:= f(X$_1$,...,X$_n$)	$\forall i = 1 \ldots n,\ X_i \xrightarrow{\widehat{X}} X$
ZX:= X$1	no dependency
X:= U whenC	$U \xrightarrow{\widehat{X}} X$
X:= U defaultV	$U \xrightarrow{\widehat{U}} X \xleftarrow{\widehat{V}\setminus\widehat{U}} V$
For each condition C	$C \xrightarrow{\widehat{C}} [C]$ $C \xrightarrow{\widehat{C}} [\neg C]$
For each signal X	$\widehat{X} \xrightarrow{\widehat{X}} X$

Table 2: From SIGNAL operators to a conditional dependency graph

2.6 Description of the generated code

Sequential code generation from the conditional dependency graph follows a very simple scheme thoroughly described in [19]. Each signal is implemented by a variable. Since a signal carries a value only when it is present (i.e it's clock is present), in the generated code, access (read or write) to the variable that implements a signal is guarded by a test on the presence of the signal. Moreover, the assignment of the value of a variable to another variable is guarded by the clock that labels the dependency between the two variables.
As an example, consider the process `X := U defaultV`. The signal X merges the signals U and V with a priority to U. The dependency graph of that process is

$$U \xrightarrow{\widehat{U}} X \xleftarrow{\widehat{V}\setminus\widehat{U}} V$$

and the code generated is:

```
if present(X̂) then
      if present(Û) then
            X := U
      endif
      if present(V̂ \ Û) then
            X := V
      endif
endif
```

In that piece of code, we write `if` $present(\widehat{U})$ to test if the instant under consideration belongs to the clock $\widehat{U}$. In the following paragraphs we will detail how the compiler implements the test of a clock's presence.

3 Solving the system of boolean equations

3.1 Resolution: the needs

The goal of the compiler is to check the consistency of the synchronizations expressed by the system of equations and to generate executable code for various architectures [3, 20]. From the conditional dependency graph and the code generation scheme, we can figure out what the needs are in terms of resolution.

The dependency, $\widehat{X} \xrightarrow{\widehat{X}} X$ requires that, at any given instant, before the value of a signal X is computed, a test be made on the presence/absence of X; that is, the presence/absence of its clock $\widehat{X}$. So there is a need for a resolution method that will allow to *efficiently* check at run-time the presence of all the clocks which are related by a system of equations. The choice made in the SIGNAL compiler is to transform the system of equations into a list of *explicit definitions*; that is, the compiler identifies in the system a set of clock variables (the *free variables*) in terms of which the other clocks are expressed. This explicitization is achieved by means of *triangularization*; that is, a transformation of the system of equations into a set of equalities of the form $k_i = k_{i1} <op> k_{i2}$ such that the clock-to-clock dependency graph (the edges $k_{i1} \longrightarrow k_i \longleftarrow k_{i2}$) be an acyclic graph (a partial order). So, the presence/absence of k_i at run-time, can be quickly deduced from the presence/absence

of k_{i1} and k_{i2} where k_{i1} and k_{i2} are subterms which the compiler tries to share with other clocks in order to yield efficient code.

Not only does a triangularization compute the order in which clocks must be evaluated, but it also exhibits the free variables of the system. It is important that the free variables be determined because they are the ones that the environment of the real-time system must provide as inputs to the program. So, if they were not statically computed, there would be a need for tests at run-time to check the consistency of the inputs; that overhead would make efficiency more difficult to achieve.

The problem of transforming a system of boolean equations into an explicit system is NP-hard [12]. So the algorithm implemented in the compiler does not seek completeness. It is rather a heuristic aimed at fast compilation of commonly encountered systems of equations. Hence some correct SIGNAL programs may be rejected because the compiler fails to produce the explicit form of their clock equations. Currently, the triangularization is carried out in the compiler through an arborescent representation of the equations. Before going into details on the representation, we give the main ideas of the strategy that lies under it.

3.2 Strategy of resolution

A triangular system of equations is progressively constructed from the original system (see Table 1). An equation of the form $k = k_1 <op> k_2$ is oriented (we note $k := k_1 <op> k_2$) in order to consider the clock formula $k_1 <op> k_2$ as the definition of the clock variable k. During this process, an orientation of some equations may not be trivially possible. There are two reasons for that.

1. The equation under consideration is of the form $k = k_1 <op> k_2$ but there is already a definition of the variable k. In this case, a rewriting can be performed to verify that the formula $k_1 <op> k_2$ is equivalent to the previous definition of k.

2. It is an equation of the form $k = k_1 <op> k_2$ but an orientation would induce a cycle in the clock-to-clock dependency graph. In this case, an attempt can be made to rewrite the formula $k_1 <op> k_2$ and break the cycle.

Note that an equation of the form $k_1 <op> k_2 = h_1 <op> h_2$ can be brought to the two previous cases by the insertion of a new variable h and by writing $h = k_1 <op> k_2$ and $h = h_1 <op> h_2$: first orient the equations and then prove the equivalence.

At the end of this process the program is said to be temporally incorrect if there are some equations whose orientation induces a cycle or if there are some non-proved equalities. Hence a canonical rewriting system is needed to check the equivalence of two clock formulas (which are boolean formulas).

Although this strategy cannot triangularize an arbitrary system of equations, it is very efficient in compiling common SIGNAL programs that implement realistic systems.

3.3 Example

We give in this section, an example of SIGNAL program. The purpose of this example is to illustrate the kind of rewriting the compiler has to perform. The reader interested in realistic programs written in SIGNAL is referred to [2] for the programming of a production cell controller, and to [18] for a speech processing system.

Consider a SIGNAL program called `PROCESS_ALARM` which must compute a boolean-valued signal `ALARM` from 3 boolean-valued signals (say sensors) `BRAKE, STOP_OK, LIMIT_REACHED`. Here is an informal specification of the behavior:

- the sensor `BRAKE` is true if the brakes of the train are activated;
- `STOP_OK` is true if the train is stopped;
- `LIMIT_REACHED` is true if the train goes beyond some limit it should not normally surpass;
- `ALARM` must be true if the train has not stopped before the limit.

A possible implementation of `PROCESS_ALARM` is the following SIGNAL equation

```
ALARM := BRAKE and LIMIT_REACHED and
         ( not STOP_OK)
```

In that equation, all the signals are required to have the same clock ($\widehat{ALARM} = \widehat{BRAKE} = \widehat{LIMIT_REACHED} = \widehat{STOP_OK}$). This means that at each instant (a *reaction* of the program), all the sensors are sampled and the value of `ALARM` is computed.

Let us imagine a more sophisticated version of that program: a sensor is sampled only when its value is necessary. We can think of this as an improvement made by a programmer in order to reduce the communications with the execution environment of the program. The sensors `LIMIT_REACHED` and `STOP_OK` need to be sampled only during a braking action. And the sensor `BRAKE` needs to be sampled only when no braking is going on. So, we introduce a state variable `BRAKING_STATE` as shown of figure 5.

Let us now focus on the clock equations that lies under this program. If we denote for short `BRAKING_STATE` by `C`, `BRAKING_NEXT_STATE` by `C'`, `BRAKE` by `D`, `STOP_OK` by `C1`, `LIMIT_REACHED` by `C2` and `ALARM` by `C3` we have:

$$\begin{cases} \widehat{C} = \widehat{C'} & (1) \\ \widehat{C'} = [D] \vee [C_1] \vee \widehat{C} & (2) \\ [C] = \widehat{C_1} = \widehat{C_2} & (3) \\ [\neg C] = \widehat{D} & (4) \\ \widehat{C_3} = \widehat{C_1} = \widehat{C_2} & (5) \end{cases}$$

```
(|  BRAKING_STATE := BRAKING_NEXT_STATE $ 1            % memorize the next state
 |  BRAKING_NEXT_STATE := ( true whenBRAKE) default    % enter the braking state
                        ( false whenSTOP_OK) default   % leave the braking state
                         BRAKING_STATE                 % stay in the previous state
 |  synchro {whenBRAKING_STATE,STOP_OK, LIMIT_REACHED} % sample in braking state
 |  synchro {when( not BRAKING_STATE), BRAKE}          % sample when not in braking state
 |  ALARM := LIMIT_REACHED  and ( not STOP_OK)         % the brake need no longer be checked
 |)
```

Figure 5: Source code of process Alarm

Lines (1), (3), (4) and (5) specify equalities of variables. For such equations, we choose one variable which will replace the others when they are referenced. By replacing $\widehat{C'}$ by $\widehat{C}$ in (2) we have:

$$\begin{cases} \widehat{C'} = \widehat{C} \\ \widehat{C_1} = [C] \\ \widehat{C_2} = [C] \\ \widehat{C_3} = [C] \\ \widehat{D} = [\neg C] \\ \widehat{C} = [D] \vee [C_1] \vee \widehat{C} \quad (6) \end{cases}$$

All the equations of the system can be trivially oriented from right to left except for equation (6). Indeed, the variable $\widehat{C}$ appears in both sides of the equation; so, an orientation would induce a cycle. To break that cycle, let us use the extra knowledge (not apparent in the system) we have about boolean valued signals: the clock $[C_1]$ is included in $\widehat{C_1}$; this is mainly due to the fact that $([C_1], [\neg C_1])$ is a partition of the clock $\widehat{C_1}$. Similarly $[C] \subseteq \widehat{C}$. Since $\widehat{C_1} = [C]$ we have $[C_1] \subseteq \widehat{C}$ and consequently $[C_1] \vee \widehat{C}$ can be rewritten into $\widehat{C}$. A similar argument shows that $[D] \subseteq \widehat{D} = [\neg C] \subseteq \widehat{C}$ and $[D] \vee \widehat{C} = \widehat{C}$. Then the formula $[D] \vee [C_1] \vee \widehat{C}$ can be rewritten into $\widehat{C}$. Finally the equation (6) becomes $\widehat{C} = \widehat{C}$ which is trivially true and deleted from the system.
In order to obtain the triangular form of the system of equations, the compiler must perform rewriting in respect to the inclusion relation among clocks. That is why a special tree representation has been introduced in the compiler to represent efficiently part of this inclusion relation.
Note that in the previous example, the variable $\widehat{C}$ cannot be computed by an expression in the program: it is a free variable exhibited by the compilation. This means that the execution environment must provide it as an input to the program. This can be rephrased as: "the specification does not determine the pace at which the sensors must be sampled". Should they be sampled every meter or every milli-second, it is a choice of the environment; it is not a real feature of the alarm functionality.

3.4 Hierarchical representation of the equations

To meet the requirements presented above, an arborescent organization of the formulas has been defined in [8]. It speeds up the rewriting and it captures the triangularity of the system of equations. We present here the main ideas.

Partition tree.
Consider the boolean-valued signal C of the previous example and its clock $\widehat{C}$. According to the properties described in the previous sections, the pair $([C], [\neg C])$ is a partition of $\widehat{C}$. Such a partition is represented by the tree in Figure 6. Since any clock can be partitioned by a condition, this basic tree can grow as its nodes are partitioned. Figure 7 represents the partition tree of the example PROCESS_ALARM of the previous section. In such a tree, an edge between a parent node and a child node captures the inclusion of the child in its parent. The root may be an arbitrary formula but the internal nodes are partitions.

Forest of clocks. As any clock formula can be partitioned, the formulas originating from a SIGNAL program can be grouped into partition trees; this set of trees is called a *forest of clocks*. Within this forest, some trees may be one-node trees; these are clocks that have not been partitioned in the original SIGNAL program.

Fusion of clock trees. In the forest of clocks, let T and T' be 2 trees with roots r and k such that a) k be defined by a formula k_1<op>k_2, b) k_1 and k_2 belong to the tree T; which means that the operands of the root k are in the tree T. We carry out a *fusion* of T' into T by inserting k into the tree T as depicted in Figure 8. In Figure 8 we point out a particular node h of the tree T. h is the $branching$ of the nodes k_1 and k_2; it is the first common ancestor of the 2 nodes. T' is now a subtree of the merge tree T''. The fusion of 2 partition trees yields a more general tree we call *clock tree*.
The main idea of the insertion of the formula k_1<op>k_2 is that it is inserted under the branching of its operands, at the "right hand side". This insertion procedure has two interesting features: a) it preserves the triangularity of the system of equations, b)it optimizes the code generated by nesting *if-then-else* control structures.

Triangularity preservation. During a depth first search (dfs) of the tree T'' "from left to right", the nodes k_1 and k_2 are visited before the node $k = k_1$<op>k_2; it means that the

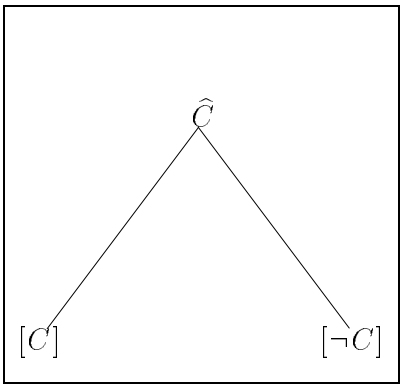


Figure 6: Basic partition tree

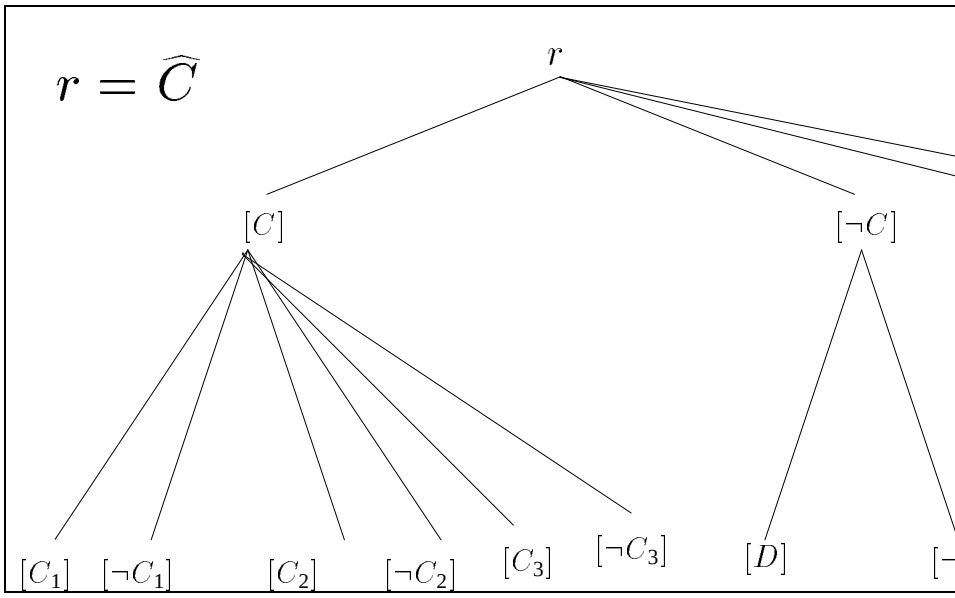


Figure 7: A hierarchical partitioning

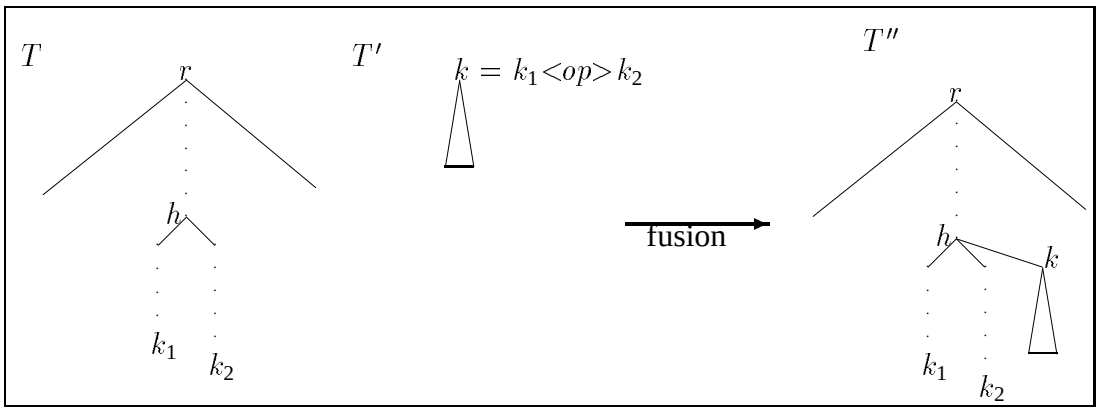


Figure 8: Fusion of trees

ordering that makes the system be triangular is embodied in the tree.

Code optimization. A partition tree can be viewed as the representation of an inclusion relation. In a partition tree, a node is included in its parent. And more generally, a node is included in its ancestors.

In Figure 8 the clock $k = k_1 <op> k_2$ is included in the clock h. As a matter of fact, h being an ancestor of both k_1 and k_2, we have $k_1 \subseteq h$ and $k_2 \subseteq h$. Consequently all of the 3 formulas $k_1 \vee k_2$, $k_1 \wedge k_2$ and $k_1 \setminus k_2$ — denoted $k_1 <op> k_2$ — are included in h. That is, the clock tree resulting from a fusion of 2 trees represents an inclusion relation. On the example PROCESS_ALARM, we have shown the usefulness of the inclusion relation for the rewriting. Now let us show how it can help in optimizing the code generated.

The nesting of *if-then-else* structures for code optimization is based on the remark that, if h and k are 2 clocks such that $h \subseteq k$, then for an instant t, the following implication holds: $t \notin k \implies t \notin h$. In other words, if the test $t \in k$ fails, there is no need to test if $t \in h$. Thus, code generation can take advantage of the inclusion relation between clocks. For example in Figure 9, code a is more efficient than code b. As reported in [19] this kind of improvement can yield a code which runs 300% faster for some SIGNAL programs.

Arborescent resolution

We give here in three steps the algorithm of resolution.

1. Take a tree T' in the forest and attempt to rewrite its root k in a way that will make the operands of k belong to the same tree T. If this succeeds, the root formula of T' can be inserted into T as described in 3.4 without disturbing the triangularity of T.

code a.
if $present(k)$ *then*
    do-something-k
    if $present(h)$ *then*
        do-something-h
    endif
endif

code b.
if $present(k)$ *then*
    do-something-k
endif
if $present(h)$ *then*
    do-something-h
endif

Figure 9: Nesting *if-then-else* control structures

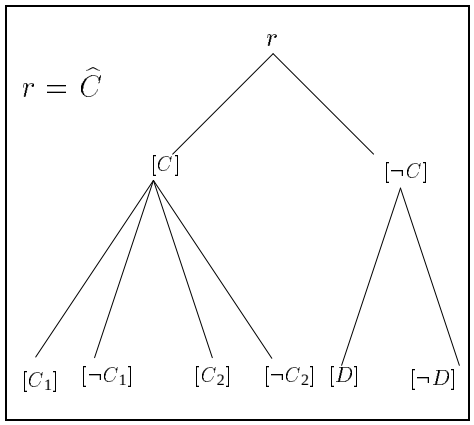


Figure 10: A hierarchical partitioning

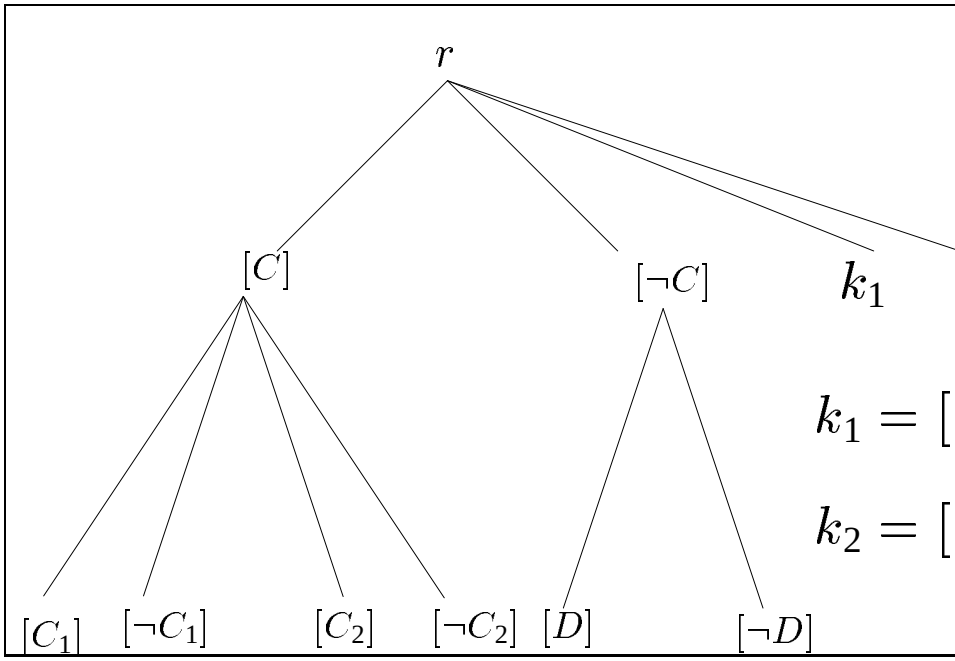


Figure 11: insertion of formulas

2. Realize the fusion of T and T' to yield a tree T'' as described above.

   After the fusion of T and T', a formula which had one operand in T and the other one in T' now has its 2 operands in the tree T''. So, the fusion of T and T' may lead to more fusions; that is the purpose of step 3.

3. Do step 1 and step 2 till the rewriting rules of step 1 no longer apply.

Step 1 is implemented using a notion of *p-depth* resolution thoroughly presented in [8]. To put it roughly, the user of the compiler can set an integer parameter p; this parameter is the maximum depth of the syntactic trees of the formulas manipulated during the rewriting. Setting a limit to the formulas, solves the duration and termination problems commonly encountered in rewriting systems. Step 2 is a simple tree manipulation. It raises though a question of canonicity that we illustrate on the following example.

Example. Consider again the tree in Figure 7 (redrawn in Figure 10 without some nodes which are not relevant for the current context) and consider the formulas k_1 and k_2 which are roots of some trees (not drawn). k_1 and k_2 are defined respectively by $[C_1] \vee [D]$ and $[C_2] \vee [\neg D]$. Recall that the main idea of the fusion of a tree T' into a tree T is that root r' of T' is inserted into T under the branching of its operands. Following that idea, the insertion of k_1 and k_2 yields the tree depicted in Figure 11. Now consider the formula $k = k_1 \wedge k_2$. The same argument would trivially place k as a child of r (see Figure 12). But k can be rewritten into another expression. Applying axioms of boolean algebra and using the inclusion relations embodied in clock trees allow to rewrite k as $[C_1] \wedge [C_2]$. The branching of $[C_1]$ and $[C_2]$ being $[C]$, k can be placed under $[C]$ (see Figure 12). As the code generation is based on *if-then-else* nesting, the insertion of k under $[C]$ yields much more efficient code.

Canonical factorization. The previous example shows that it is important to insert a formula under the deepest possible parent. So we developed an insertion algorithm which optimizes the depth. Our algorithm has an important feature: among the potential parents of the formula, it chooses the one with the greatest depth; and we show in [1] that such

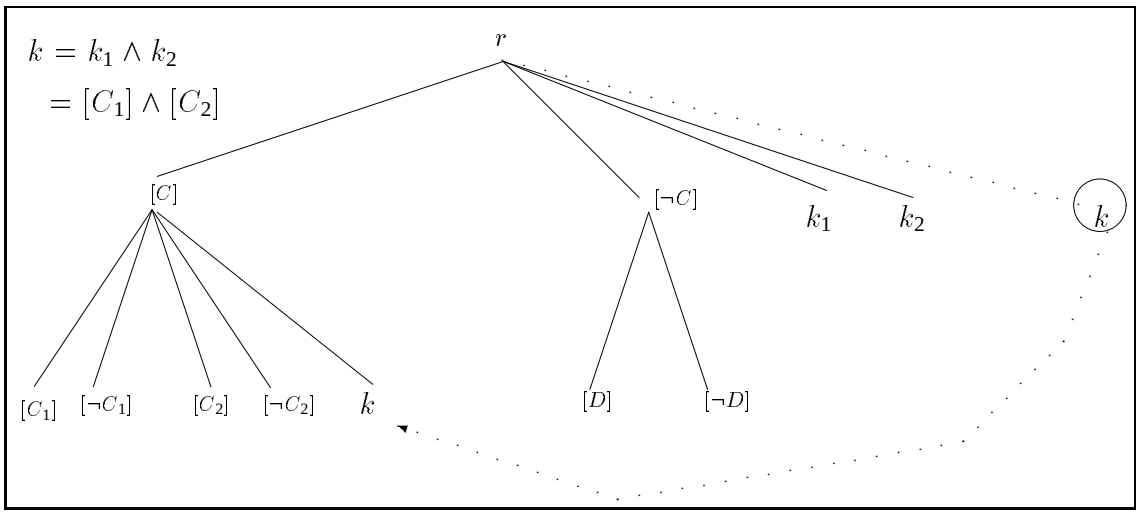


Figure 12: Best insertion

a parent is unique. That feature makes our tree structure a canonical form. The algorithm is not detailed here but the main concepts are given below:

- a BDD (Binary Decision Diagram [10]) is associated to each clock; thus the tree of clocks is transformed into a tree of BDDs;
- the problem of finding a parent for a formula is reformulated as factorizing a boolean function;
- factorizations are carried out by taking advantage of the specific properties of our tree.

4 Related work

The effort to generate code from a data-flow synchronous language has also been undertaken for the LUSTRE language. The compilation of LUSTRE produces an automaton; it offers as an option, a trade-off between response time and size of generated code.
The automaton may be a one-state and one-transition automaton. The transition is fired at each reaction of the program. It is labelled with a set of equations to be evaluated in order to compute the outputs. This style is termed *single-loop code generation scheme*. SIGNAL's compilation produces the same kind of automaton. The major difference is that in SIGNAL, the code generated is improved by the nesting of *if-then-else* control structures which has been made possible by our clock inclusion tree. To our knowledge no such hierarchical inclusion information has been used in LUSTRE to improve the code generated.

As reported in [15] for LUSTRE and in [22] for the ESTEREL synchronous language, the efficiency of the code generated can be improved by the production of a partially explored automaton: that is, the compiler may pick some boolean variables and simulate *statically* their evolution. This static simulation yields a bigger automaton than the one of the single-loop style. But in this case, the set of equations evaluated at each reaction is smaller since the simulated boolean variables need not be computed any longer. So the code is bigger and runs faster. The problem with this style of generation is that, in the worst case, the automaton grows exponentially with the number of simulated boolean variables. Hence the need of heuristics to cleverly select the variables to be simulated.

5 Conclusion

In this paper we have presented the data-flow oriented language SIGNAL and we have given an overview of the boolean techniques used for its compilation. These techniques have been successfully implemented and we give here some experimental results.
Figure 13 shows the amount of computing resources required for the compilation of sample SIGNAL programs. To show the effectiveness of our arborescent representation, we compare three representations of boolean systems of equations.

- *Tree and* BDD (*T&BDD*): a tree structure together with a BDD canonical form as presented earlier in this paper.
- BDD *characteristic function*: the whole system of equations is represented by a single BDD; a system of equations over n boolean variables can be viewed as a subset of $\{0,1\}^n$. Hence it can be given a representation in

sample SIGNAL programs	number of variables	*T&BDD*		BDD characteristic function		BDD charac. func. after *T&BDD*	
		nodes	time	nodes	time	nodes	time
STOPWATCH	1318	61893	27.07s	unable-cpu		unable-cpu	
WATCH	785	34753	14.67s	unable-cpu		unable-cpu	
ALARM	465	3428	2.19s	unable-mem		unable-cpu	
CHRONO	282	1548	0.92s	unable-mem		422975	409.09s
SUPERVISOR	202	425	0.45s	unable-cpu		226472	146.32s
PACE MAKER	96	50	0.10s	53610	160.50s	582	0.36s
ROBOT	99	36	0.27s	unable-cpu		415	0.31s

unable-cpu: computation was unable to terminate within the 40mn time limit.
unable-mem: computation was unable to terminate within the 200MB memory limit.

Figure 13: Comparisons

the form of a characteristic function. This representation of subsets of $\{0,1\}^n$ is very common in the field of hardware verification and silicon compilation [13, 24]. To solve exactly boolean equations, there is a complete algorithm which runs polynomially in the size of this BDD (see [12]). In order to justify our non-complete algorithm, we show that very often in practical cases, this BDD is too big to be computed.

- BDD *characteristic function after T&BDD*: the original system of equations is transformed by a *T&BDD* into a tree (which is still a system of equations); then a BDD characteristic function is constructed. The difference between this system and the original one, is that some variables may be (and very often are) eliminated due to their equivalence with other variables. So, the triangularized system has less variables.

The representations are compared in terms of memory (number of BDD nodes) and time (Unix user-time). The measures are conducted on a SUN4/Sparc10 with 64MB main memory. Manipulations of BDDs use a UC Berkeley BDD package [23].

For the experimentation we set a 200MB virtual memory limit and a 40mn cpu time limit.

As it is shown on the table 13, most of the measures that involve a characteristic function were unable to compute within the resource limits. It appears clearly that characteristic functions are impractical.

6 Acknowledgements

The authors would like to thank the Committee members for their careful review and helpful comments.

References

[1] T. Amagbegnon, L. Besnard, and P. Le Guernic. *Arborescent Canonical Form of Boolean Expressions*. Internal publication 826, IRISA, May 1994.

[2] T. P. Amagbegnon, P. L. Guernic, H. Marchand, and E. Rutten. SIGNAL – the specification of a generic, verified production cell controller. In T. L. In C. Lewerentz, editor, *Case Study Production Cell – A Comparative Study in Formal Software Development*, Lecture Notes in Computer Science, Springer-Verlag, 1995. 891.

[3] M. Belhadj. Using VHDL for link to synthesis tools. In *North Atlantic Test Workshop*, Nîmes, France, June 1994.

[4] A. Benveniste and G. Berry. Special section on another look at real-time programming. *Proceedings of the IEEE*, 79(9):1268–1336, September 1991.

[5] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.

[6] A. Benveniste and P. Le Guernic. *A denotational theory of synchronous communicating systems*. Research Report 685, INRIA, Rocquencourt, June 1987.

[7] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 87–152, 1992.

[8] L. Besnard. *Compilation de* SIGNAL*: horloges, dépendances, environnement*. PhD thesis, Université de Rennes 1, France, Sep. 1992. in french.

[9] P. Bournai, C. Lavarenne, P. Le Guernic, O. Maffeïs, and Y. Sorel. *Interface SIGNAL-SynDEx*. Research report 2206, INRIA France, Rennes, March 1994.

[10] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on computers*, C-35(8):677–691, August 1986.

[11] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: a declarative language for programming synchronous systems. In *14th ACM Symposium on Principles of Programming Languages*, pages 178–188, Munich, 1987.

[12] O. Coudert. *SIAM: Une Boîte à Outils Pour la Preuve Formelle de Systèmes Séquentiels*. PhD thesis, Ecole Nationale Supérieure des Télécommunications, France, 1991.

[13] B. Dutertre. *Spécification et preuve de systèmes dynamiques*. PhD thesis, Université de Rennes 1, France, Décembre 1992.

[14] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer, 1993.

[15] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In J. Maluszynski and M. Wirsing, editors, , page , Springer Verlag, August 1991. LNCS 528.

[16] 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.

[17] P. Le Guernic and T. Gautier. Data-flow to von neumann: the SIGNAL approach. In J. Gaudiot and L. Bic, editors, *Advanced topics in data-flow computing*, pages 413–438, Prentice Hall, 1991.

[18] C. Le Maire, R. Andre-Obrecht, and P. Le Guernic. A new real-time synchronous programming approach to continuous speech recognition. *IEEE transactions on Automatic Control*, 1990.

[19] O. Maffeïs. *Ordonnancements de graphes de flots synchrones; Application à* SIGNAL. PhD thesis, Université de Rennes 1, France, Jan. 1993.

[20] O. Maffeïs, B. Chéron, and P. Le Guernic. *Transformations du Graphe des programmes* SIGNAL. Research report 1574, INRIA France, Rennes, January 1992.

[21] F. Maraninchi. Argonaute: graphical description, semantics and verification of reactive systems by using a process algebra. In J. Sifakis, editor, *Automatic Verification Methods for Finite-state Systems*, pages 38–53, Springer-Verlag, 1989. LNCS 407.

[22] F. Mignard. *Compilation du langage Esterel en système d'équations boolénnes*. PhD thesis, Ecole des Mines de Paris, France, 1994.

[23] E. M. Sentovich, S. K. J., L. L., M. C., M. R., A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vicentelli. *SIS: A System form Sequential Circuit Synthesis*. Research report UCB/ERL M92/41, UCB, 1992.

[24] H. J. Touati, H. Savoy, R. Brayton, B. Lin, and A. Sangiovanni-Vicentelli. Implicit state enumeration of finite state machines using bdd's. In *IEEE conference on Computer-Aided Design*, pages 130–133, 1990.

[25] W. W. Wadge and E. A. Ashcroft. LUCID, *the Dataflow Programming Language*. Academic Press, 1985.