Author manuscript, published in "29th Hawaii International Conference on System Sciences (HICSS-29), Maui, Hawaii : United States (1996)"
DOI : 10.1109/HICSS.1996.495517

In proceedings of the 29th Hawaii International Conference on System Sciences HICSS-29, Hawaii, January 1996, to appear, avaible at http://www.irisa.fr/prive/aubry/hicss/

hal-00544057, version 1 - 7 Dec 2010

Synchronous distribution of SIGNAL programs

P. Aubry P. Le Guernic S. Machard

{Pascal.Aubry,Paul.LeGuernic,Sylvain.Machard}@irisa.fr
IRISA/INRIA, Campus de Beaulieu, F-35042 Rennes Cedex

Abstract

SIGNAL, a synchronous and data-flow oriented langage, allows the user to design safe real-time applications. Its compiler uses a single formalism called "Synchronized Data-Flow Graphs[1]" all along the conception chain from specification to proof and verification. We show how this formalism can be kept on until distributed code generation. The implementation described here, called synchronous distribution, respects the semantics of SIGNAL. We finally show the limits of SDFGs and conclude on the necessity of another model describing dynamic behaviours of distributed executions.

1 Introduction

Based on the hypothesis of a discrete logical time, the synchronous languages [1] (SIGNAL, LUSTRE [2], ESTEREL [3]) have proved their efficiency for the design of critical and safe real-time applications. They are characterized by strong semantics that allow the programmer to use verification and proof techniques. SIGNAL [4], one of the synchronous languages, is declarative and data-flow oriented. In this language, a signal is a sequence of values; the instants of presence of the signal are called clocks. The SIGNAL compiler [5] is a formal system able to solve equations and to reason upon logic. All along the compilation process, it manipulates only equations and dependencies finally structured in an internal representation called *Synchronized Data-Flow Graph*.

SDFGs can be seen as a generalization of *Directed Acyclic Graphs*[2]. Indeed, the signals (vertices of SDFGs) are present only at some instants (their clock) and the dependencies (edges of SDFGs) are clock-labelled (i.e. take effect only at the instants of a given clock). Each object of SDFGs is located in a hierarchy of clocks, resulting of what is called the *clock calculus*. Those graphs are used in the final phases of the compilation to produce textual outputs, sequential code generation [6] and hardware synthesis [7].

For different reasons (delocalization of sensors, fault tolerance, frequency increasement, ...), many real-time applications require code distribution. As translations between different representations are error-prone, safety requirements of critical applications are insured in particular by the perservation of a single formalism all along the design process. From specification to final implementation, simulation, proof and verification, the SIGNAL compiler uses a single formalism: SIGNAL equations and dependencies.

[1] Called thereafter SDFGs.
[2] also called DAGs.

We intend to prove that this formalism can be kept on even until the final phase of distributed code generation, once equations have been assigned on processors, and that this approach allows the preservation of semantical properties in a distributed system. The implementation uses mixed static/dynamic scheduling.

Firstly, we give a short overview of SIGNAL (section 2) to present the SDFGs. Then we show how communications can be introduced in SDFGs (section 3), how the control part of SIGNAL programs can be distributed (section 4) to extract sub-graphs corresponding to different processors (section 5). Finally, we show possible implementations (section 6) and give some perspectives of this work.

2 SIGNAL

2.1 The SIGNAL language

As SIGNAL is a dataflow-oriented language, it describes processes which communicate through sequences of (typed) values with an implicit timing: *signals*. For instance, a signal `X` denotes the sequence $(\mathtt{x}_t)_{t\in I\!N\setminus\{0\}}$ of data indexed by time.

$\leadsto$ **Kernel of SIGNAL** The kernel of the SIGNAL language includes the operators on signals and the process operators. Four kinds of operators act on signals:

- **instantaneous functions** is a class of operators which encompasses all the usual functions (`and`, `<=`, `+`, `fft`, ...) extended to act on signals. Let f a symbol which denotes an n-ary function acting on signals and $[\![f]\!]$ the corresponding function acting on values, the SIGNAL process `Y:=`f`{X1,...,XN}` specifies that $\forall t \geq 1 \quad \mathtt{y}_t = [\![f]\!](\mathtt{x1}_t, \ldots, \mathtt{xn}_t)$. In the specified behavior, one may notice that the value $\mathtt{y}_t$ carried by `Y` at instant t is equal to the function $[\![f]\!]$ applied to the values held by `X1`, ..., `XN` at the same instant. This fact is the result of a special specification approach: the *(strong) synchronous approach* (see [8] for an overview). In the dataflow synchronous approach [1], the execution of the operators is assumed of zero duration[3], only the logical precedence of values on a signal represents passing time. Therefore, firing waits and implicit queueing of data are suppressed at the specification level.
- **shift register** makes explicit the memorization of data; it enables the reference to a previous value of a signal. For instance, the process `Y:=X $ 1` defines a basic process such that $\mathtt{y}_1 = \mathtt{v0}$ and $\forall t > 1 \quad \mathtt{y}_t = \mathtt{x}_{t-1}$ where `v0` denotes an initial and constant value

[3] In fact this theorical point of view means that durations are not taken into account.

associated with the declaration of Y. In contrast to the last two operators, the signals referred to in instantaneous functions or in the shift register must be bound to the same time index, the same *clock*.

- the **selection** operator allows us to draw some data of a signal through some boolean condition. The process `Y:=X when B` specifies that Y carries the same value as X each time X carries a data and B carries the value *true* (B must be a boolean signal). Otherwise, Y is absent, i.e. Y carries no value.
- the **merge** operator combines flows of data. The process `Y:=X1 default X2` defines Y by merging the values carried by X1 and X2 and giving priority to X1's data when both signals are simultaneously present.

The four previous operators specify basic processes. The specification of complex processes is achieved with the **parallel composition operator**: the composition of two processes P1 and P2 is denoted `(| P1 | P2 |)`. In the composed process, the common names between P1 and P2 refer to common signals; they stand for the communication links between P1 and P2. This parallel composition is an associative, commutative and idempotent operator (`(| P | P |)` ~ P).

The last feature of the kernel is the possibility to reduce the **scope of a signal**: in the process `P()/X`, the signal X is set local to the process P.

⇝ **Extended operators** Built on the previous primitive operators, some built-in features often used by programmer have been added :

- **clock extraction** makes explicit the clocks of signals. `C := event X` means that C is the clock of X and is equivalent to `C := (X=X)`.
- **synchronization** between signals induces new constraints in programs. `synchro{X,Y}` means that the two signals X and Y have the same clock (i.e. must be present at the same instants). `synchro{X,Y}` is a process equivalent to
  `(| C := (event X) = (event Y) |) /C`.
- the **memory cell** allows the programmer to keep the previous values of a signal at the *true* occurences of a boolean. `Y := X cell B` is present when X is present or B is present and true. In the first case, it is equal to X; otherwise, it is equal to the last occurence of X. `Y := X cell B` is equivalent to

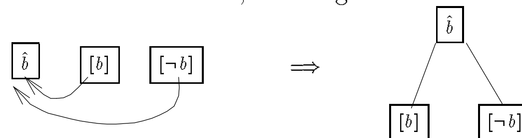
```
(| synchro{Y, (event X) default (when B)}
 | ZX := X $ 1
 | Y := X default ZX |) / ZX
```

The specification of SIGNAL programs is architecture-independent. This independence comes from the synchronous specification approach and the dataflow/equational style of the SIGNAL language. Therefore, the inference of reliable and efficient implementations is achieved in two steps. Firstly, we intend to *validate the specification independently from any target architectures*. The next subsection describes the compilation process and describes the final representation of the compiled programs: Synchronous Data-Flow Graphs. Then we show how transformations of these graphs can lead to distributed implementations.

2.2 Synchronous Data-Flow Graphs

As SIGNAL is equational, the SIGNAL compiler is not a simple translator from high-level specifications to executable code. It is a formal system able to reason upon logic and clocks.

- The first step of the compilation is the **reduction into the kernel language** of the input source.
- Clock-nodes are then created, gathering all the synchronous signals of the program. A clock c can be:
  - the **clock of a signal** X; it is thereafter noted $\hat{X}$ and is present when X, which must be an input signal of the program, is present.
  - the positive (respectively negative) **sampling** of a boolean condition b; it is thereafter noted $[b]$ (resp. $[\neg b]$) and is present when b is present and true (resp. false).
  - the **upper bound** of two other clocks c_1 and c_2; it is thereafter noted $c_1 \vee c_2$ and is present when at least one of c_1 and c_2 is present.
  - the **lower bound** of two other clocks c_1 and c_2; it is thereafter noted $c_1 \wedge c_2$ and is present when c_1 and c_2 are both present.
  - the **complementary** of a clock c_1 in a clock c_2; it is thereafter noted $c_2 \ominus c_1$ and is present when c_2 is present and c_1 is absent.
- The main phase of the compilation is called the **clock calculus**. By analysing all the clocks of the program, it builds *clock trees* by placing clocks under their father[4]. For instance, the clocks $[b]$ and $[\neg b]$ are placed under the clock $\hat{b}$, with regard to one another:



The result of the clock calculus is a forest of trees of which all the roots are free one another. If one single root is present, the forest is reduced to a single tree; this means that the compiler has found a *main-clock*, quicker than any other one in the program. Such programs are called *endochronous*. During this analysis, circuits in the definitions of clocks are detected, and clock constraints are also established.

- Each clock c of the hierarchy owns *instructions* (SIGNAL equations) explaining the computations of the signals present at the instants of c. An instruction can be:
  - a **definition of signal**. It defines a signal X by an equation like `X :=` *exp*.
  - an **external call**. It specifies that an external function P has to be called when c is present.
  - a **delay**, which is a special definition of signal. It defines a signal ZX by an equation like
    `ZX := X $` n `window` m
    where X is a signal, n and m are integer values. ZX is an m-array containing at any instant t the values $X_{t-(m+n-1)}, \dots, X_{t-n}$.

[4] A clock c in a tree can not be present if its father is absent.

hal-00544057, version 1 - 7 Dec 2010

These instructions induce dependencies between signals. As signals are not always present (as in DAGs), dependencies are clock-labelled: a dependency between two signals X and Y is effective only at the instant of their *dependency-clock* c: $\texttt{X} \xrightarrow{c} \texttt{Y}$. The clock c is always included in $\hat{\texttt{X}}$ and $\hat{\texttt{Y}}$, which means that the dependency can be effective only if X and Y are both present. Two additionnal rules apply to dependencies, caracterizing serialization:

$$\texttt{X} \xrightarrow{c_1} \texttt{Y} \xrightarrow{c_2} \texttt{Z} \implies \texttt{X} \xrightarrow{c_1 \wedge c_2} \texttt{Z},$$

and parallelism:

$$\left.\begin{array}{l} \texttt{X} \xrightarrow{c_1} \texttt{Y} \\ \texttt{X} \xrightarrow{c_2} \texttt{Y} \end{array}\right\} \implies \texttt{X} \xrightarrow{c_1 \vee c_2} \texttt{Y}.$$

A transitive closure on the dependencies of the graph reveals circuits like:

$$\texttt{X}_1 \xrightarrow{c_1} \texttt{X}_2 \xrightarrow{c_2} \ldots \xrightarrow{c_{n-1}} \texttt{X}_n \xrightarrow{c_n} \texttt{X}_1.$$

Thanks to the rule of serialization, the clock of this circuit is $c = \bigwedge_i c_i$; if $c = \emptyset$ (never present clock), the circuit is never effective and the scheduling of the computation depends on the clocks c_i. Such a circuit is rejected at compile-time, because of the cost of the analysis needed to figure out *sometimes* and *never-effective* circuits[5].

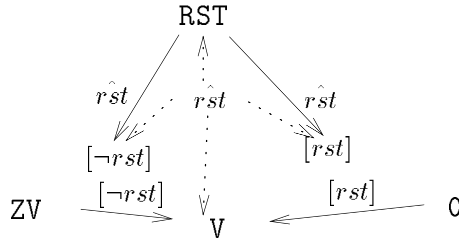
We see in this short description of SDFGs their two main aspects. A clock-hierarchy can be used to reduce control computations, and a data-flow graph in which dependencies (edges) between signals (vertices) are clock-labelled. In the sequel of this paper, we consider the dual graph: nodes compute signals and clocks.

2.3 An example

Let us consider the following example, specifying in SIGNAL a counter V synchronous with a boolean RST and reset to zero when RST is true:

```
process P =
    { ? boolean RST ! integer V }
    (| synchro{ RST, V }
     | V := (0 when RST) default (ZV + 1)
     | ZV := V $ 1
     |) where integer ZV init -1
end
```

With such an input, the compiler finds three different clocks in the program: $\hat{rst}$, $[rst]$ and $[\neg rst]$. The signals V, ZV and RST are synchronous and their clock is $\hat{rst}$. The clocks $[rst]$ and $[\neg rst]$ are sub-clocks of $\hat{rst}$. The final SDFG of this program is:



[5] Those circuits are rejected also for historical reasons: the compiler generates, at the moment, only monoprocessor sequential executable code, statically scheduled at compile-time.

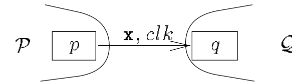
On this SDFG, signal names are bold-faced, clocks are emphasized. Solid and dotted arrows represent respectively clock and data dependencies; the clocks labelling the dependencies are located just next to the arrows.

3 Communications in SDFGs

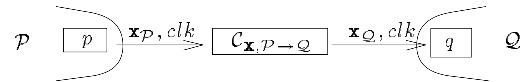
We assume in this section that all the nodes (signals and clocks) of the graph have been located on a set of processors. This point of view fits well to some requirements of real-time applications: though speed is an important criterian to appreciate reactive systems, distribution is also needed for specific reasons such as the delocalization of sensors. Readers interested in the distribution of SIGNAL programs on quantitative criteria may refer the SIGNAL/SYNDEX interface [9]. The way the nodes of the SDFG are assigned to the processors is not shown here. One can think of directives set by the user at the source level (pragmas for instance) or after the creation of the SDFG with an interactive tool taking place in the design process just before the distribution itself.

3.1 Data communication between two nodes

Let us consider a signal **x** produced by a node p on a processor $\mathcal{P}$ and consumed by a node q on a processor $\mathcal{Q}$ at a clock clk included in $\hat{\mathbf{x}}$:

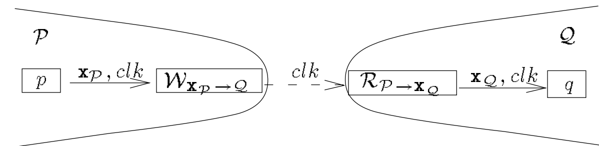


If $\mathcal{P} = \mathcal{Q}$, the graph is left unchanged. Otherwise, a communication is needed between $\mathcal{P}$ and $\mathcal{Q}$. We introduce a *communication node* $\mathcal{C}_{\mathbf{x},\mathcal{P}\to\mathcal{Q}}$:



The signal x, produced by the node p and consumed by the node $\mathcal{C}_{\mathbf{x},\mathcal{P}\to\mathcal{Q}}$ at the clock clk, is renamed $\mathbf{x}_\mathcal{P}$ and a new signal $\mathbf{x}_\mathcal{Q}$, produced by the node $\mathcal{C}_{\mathbf{x},\mathcal{P}\to\mathcal{Q}}$ and consumed by the node q at the clock clk, is introduced. This simple operation can be seen in SIGNAL as a flow-renaming, as the synchronous hypothesis says that computation durations are null (at least ignored from a practical point of view). Thus, communication nodes do not change the semantics of a program.

The communication node introduced before can be cut in two, a write node $\mathcal{W}_{\mathbf{x}_\mathcal{P}\to\mathcal{Q}}$ on $\mathcal{P}$ and a read node $\mathcal{R}_{\mathcal{P}\to\mathbf{x}_\mathcal{Q}}$ on $\mathcal{Q}$:



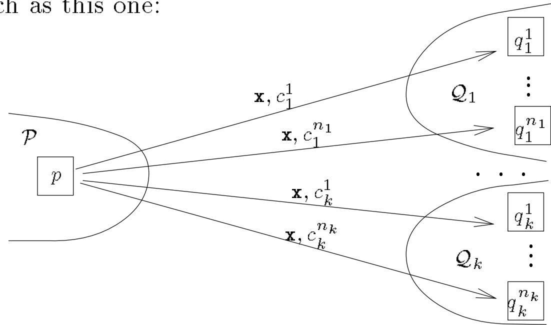
The reader should note that a dependency (at the clock clk) is left between the two nodes $\mathcal{W}_{\mathbf{x}_\mathcal{P}\to\mathcal{Q}}$ and $\mathcal{R}_{\mathcal{P}\to\mathbf{x}_\mathcal{Q}}$, insuring that the dependency between the nodes p and q is left unchanged. We can then affirm that the new graph obtained by the introduction of read/write nodes

hal-00544057, version 1 - 7 Dec 2010

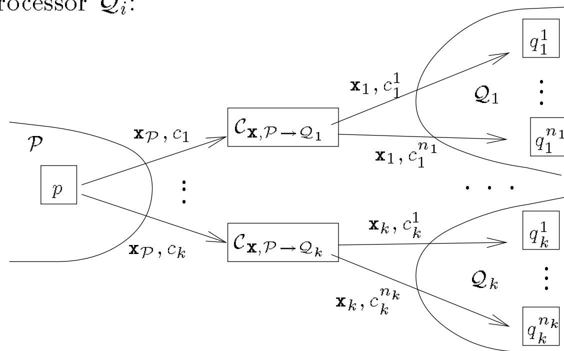
is dead-lock free, i.e. it introduces no circuit. It is obvious that the memory is kept bounded by all the transformations explained above. Finally, from the SIGNAL point of view, read/write nodes are seen as external functions; as the synchronous hypothesis tells that such processes have a null computation duration, the response time theoritically is also kept bounded. In practice, we have to ensure that computation durations are bounded, to get a global bounded response time for the program. As quantitative aspects of distribution are not studied here, we assume that communication durations are all bounded. We have shown that the introduction of communication-nodes[6] in SDFGs does not change the semantics of the initial program.

3.2 Data communications in a complete graph

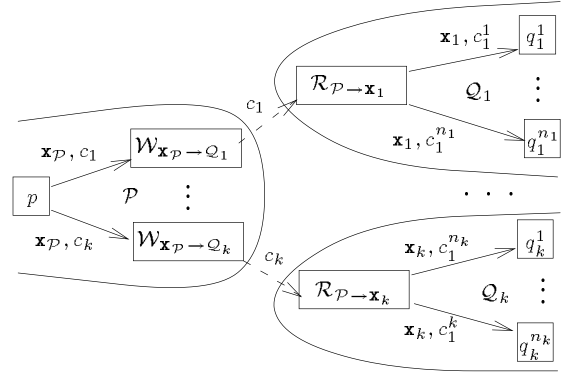
In this sub-section, we show how the principle described above can be extended to a complete graph, such as this one:



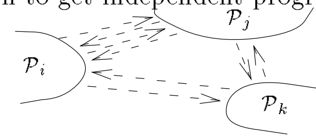
For this, let us consider the more general case of a node p of a processor $\mathcal{P}$ producing a signal $\mathbf{x}$ consumed by different nodes located on a set of processors $\{\mathcal{Q}_i\}_{i\in\{1..k\}}$. The difficulty introduced for the transformation of such a sub-graph is induced by the presence of many clocks. The choice made to preserve the dependencies at the right clocks is to introduce one communication for each processor $\mathcal{Q}_i$:



The clock dependence between the node p and any node $\mathcal{C}_{\mathbf{x},\mathcal{P}\rightarrow\mathcal{Q}_i}$ is set to $c_i = \bigvee_{j=1}^{n_i} c_i^j$. The communication nodes can still be seen as simple renamings (of $\mathbf{x}_\mathcal{P}$ into $\mathbf{x}_{\mathcal{Q}_i}$). The introduction of read/write nodes is then problemfree thanks to the dependence between two corresponding read/write nodes:



Of course, such transformations can be performed for every edge of a SIGNAL graph. The result is a new graph, in which the dependencies between two nodes located on different processors are no more data-dependencies but (simple) clock-labelled dependencies[7]: Each sub-graph is located on a single processor. The independant sub-graphs can now be extracted from the complete graph to get independent programs:



Presented this way, the problem of the distribution seems very simple. In fact, we have intentionally hidden two main issues: the distribution of the control and the consequences of the extraction of sub-graphs into independent programs [8] on their semantics. They are discussed in the next sections.

3.3 Discussion

To be efficient, the algorithm must not unmark any already-marked node. This could happen when introducing a new clock-node depending on a signal $\mathbf{x}$ of which the production node p has already been treated (marked). As the communications between p and the nodes consuming x may be affected by the new consumption of $\mathbf{x}$, communication nodes should then be changed. To prevent this, only nodes without unmarked successors are treated. The proof shows below that it is always possible.

In a first approximation, let us say that the communication clock chosen for a signal is $c_\mathcal{Q} = \bigvee_{i\in\{1..k\}} c_i$, which corresponds to the lower bound of the possible clocks. Other possible clocks are $\hat{\mathbf{x}}$ (the production clock of $\mathbf{x}$) and all the clocks between $c_\mathcal{Q}$ and $\hat{\mathbf{x}}$ in the clock-hierarchy. In these cases, no supplementary node is introduced (because the communication clock is always already in the graph) but this way communications are fired more often than necessary.

3.4 Proof

To insure the correctness of this algorithm, we have to prove that it never locks, that it always ends, that

[6] Called thereafter comm-nodes.

[7] the only meaning is a temporal precedence between the two nodes which has to be respected by any scheduler.

[8] to generate executable code.

hal-00544057, version 1 - 7 Dec 2010

hal-00544057, version 1 - 7 Dec 2010

the initial dependencies between the initial nodes of the graph are left unchanged and that the new graph is a SDFG according to the subsection 2.2.

Dead-locks may happen if the choice (third line of the algorithm) of a node p is impossible. If no node was added to the graph, this would be obvious. As some new clock-nodes may be introduced (as unmarked), we have to prove that this can not lead to the following dead-lock situation: all the unmarked nodes have unmarked successors. This is strictly equivalent to say that there is a cycle in the graph. We have then to prove that the new nodes do not induce cycles. As this is the crucial point of the proof, it is detailed in 3.5.

The only solution for the algorithm to infinitly loop is that it generates more nodes than it consumes: the new nodes are introduced unmarked, which means that the number of unmarrked nodes may not reduce at each step. In fact, as the initial number of clocks of the graph is fixed, and as the algorithm only introduce upper bounds of already existent clocks, the number of new clocks is also bounded[9], which means that the algorithm will always end.

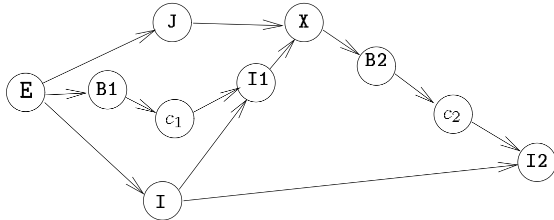
The last point to verify is the corectness of the new graph regarding to the properties of the first one. All the objects introduced (the clocks $c_{\mathcal{Q}}$ and the signals ($\mathtt{x}_{\mathcal{Q}}$) are placed exactly as in the clock calculus of the SIGNAL compilation. Let us note that the signals `x` do not need to be moved after the introduction of communication nodes. Finally, thanks to the dependencies between two corresponding read/write nodes, the dependencies between the initial nodes are left unchanged. Finally, this algorithm is correct if the communication clock does not introduce circuits in the graph.

3.5 Communication clocks

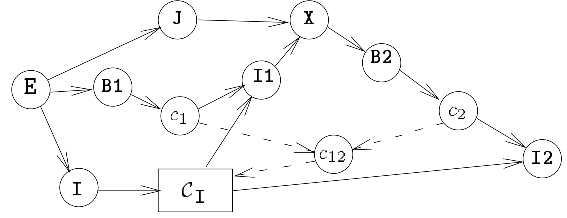
Let us consider this SIGNAL program:

```
(|      { I, B1, J } := f{ E }
 |      I1 := I when B1
 |      I2 := I when B2
 |      X := I1 default J
 |      B2 := g{ X }              |)
```

The compilation process introduces the clocks c_1 and c_2 denoting respectively the instants the booleans `B1` and `B2` are true. To make the associated SDFG as clear as possible, we did not mention the main clock of the program, synchronous with `E`, `I`, `B1`, `J`, `X` and `B2`:



If we assume that the programmer wants to produce `I` on a processor $\mathcal{P}$ and `I1` and `I2` and a processor $\mathcal{Q}$, a comm-node $\mathcal{C}_{\mathtt{I}}$ from $\mathcal{P}$ to $\mathcal{Q}$ is needed. The upper bound of the consumption clocks is $c_1 \vee c_2$ and is noted c_{12}. The introduction of c_{12} obviously entails a circuit in the graph:



After analysis, the circuit is never effective because its dependency-clock is null. This problem, firstly encountered in [10] is left unresolved. Indeed, we did neither find examples where communications clocks induce effective circuits nor managed to prove that those circuits are always never-effective.

As the correctness of the previous algorithm depends on the choice of the communication clock, we choose $c_{\mathcal{Q}}$ in such way that no circuit is introduced. If $\bigvee_{i\in\{1..k\}} c_i$ is not present in the graph, it is introduced, produced on the processor $\mathcal{Q}$[10]. If firing the comm-nodes at this clock induces circuits, we choose the slowest clock ancestor up to $\hat{\mathtt{X}}$ that can fire the comm-nodes without introducing any new circuit. In the worst case, the clock chosen is $\hat{\mathtt{X}}$ and `X` is communicated as often as it is produced.

4 Control distribution

Two corresponding read/write nodes are activated at the same clock. This clock should then be present on each processor. With our hypothesis saying that no signal is duplicated, the clock is produced on one processor and should then be communicated to the other one: *data communications imply control communications.*

Let us assume that clocks are implemented as booleans[11] (true/false values assumed as present/absent states). So they can be treated exactly like the other signals. The only difference is that they can not be communicated at their own clock but at a clock quicker than it (its communication clock). This implies the implementation of this communication clock on the sending and the receiving processor: *control communications imply other control communications.* Of course, the phenomenon is bounded because the number of clocks of any program is fixed by the compiler while creating the SDFG and thus the depth of any clock is also bounded.

A possible choice is to distribute all the clocks of the program on each processor. This trivial and systematical method is very expensive and may generate useless communications in most of the programs. To improve the distribution, a reduction of the clock-hierarchy on each processor is needed.

4.1 Extraction of useless clocks

The first step of the simplification of the hierarchy is the elimination of *useless* clocks. To see which clocks

[9] because the upper bound is associative and commutative, and the clocks are stored in the graph in a single way.

[10] Because all the clocks c_i are already on this processor.

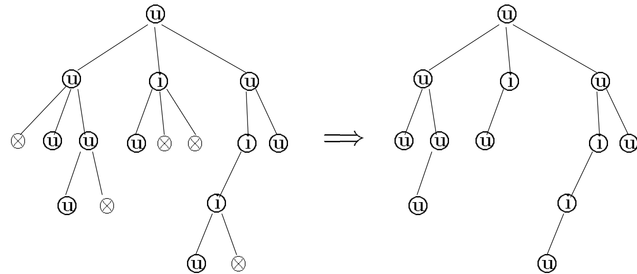
[11] This hypothesis is justified in [6].

hal-00544057, version 1 - 7 Dec 2010

can be immediatly suppressed, we introduce a classification.

- A clock c is said *necessary* on a processor $\mathcal{P}$ if and only if at least one of these assertions is true:
  — an instruction has to be fired on $\mathcal{P}$ when c is present (a signal has c as its clock).
  — a signal computed on $\mathcal{P}$ depends on c.
  — another clock, different from any sub-clock, computed on $\mathcal{P}$ depends on c.
- To keep the richness of the initial hierarchy, we say that a clock c is *useful* if it is necessary or if at least two sub-hierarchies of c own necessary clocks. This way the evaluation of sub-clocks is not necessary when their ancestor is not present, reducing the computation of the control part of a program (see the production of sequential mono-processor code in [6]). These useful clocks have to be implemented on $\mathcal{P}$.
- At the opposite, a clock c is said useless on a processor $\mathcal{P}$ if and only if:
  — no instruction has to be fired when c is present (no signal has c as its clock).
  — no signal on $\mathcal{P}$ depends on c.
  — no other useful clocks produced on $\mathcal{P}$ depends on c.
  — all the sub-clocks of c are useless (on $\mathcal{P}$).
- The other clocks of the program, that are neither useless nor necessary, are clocks with no instruction, no successor (signal or clock) but their useful sub-clocks. Those clocks are said *intermediate*.

As a useless clock c does not precede any signal or useful clock computed on $\mathcal{P}$, no need to implement it on $\mathcal{P}$. Useless clocks can be extracted from the clock-hierarchy without any problem. The resulting hierarchy is said *purified*. The following figure shows such a transformation on a clock-hierarchy projected on a processor $\mathcal{P}$:



In the hierarchies, useful, useless and intermediate clocks on $\mathcal{P}$ have been represented respectively with the symbols ⓤ, ⊗ and ⓘ. As all the useful clocks have to be implemented on $\mathcal{P}$, reducing the purified hierarchy means extracting intermediate clocks.

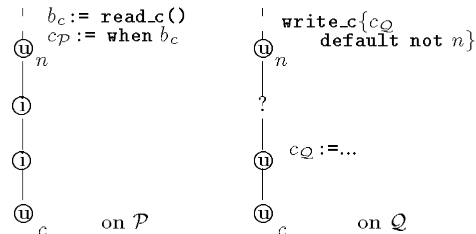
4.2 Extraction of intermediate clocks

As said before, a possible choice is to communicate any clock c when its upper clock in the clock-hierarchy is present (when c is the main clock of the program, there is no need to communicate it if absent because no computations are needed). This leads to the duplication of almost all the clocks on each process, keeping all the richness of the hierarchy but obviously generating useless communications: the communication of a clock may need the communication of all its ancestors in the clock-hierarchy.

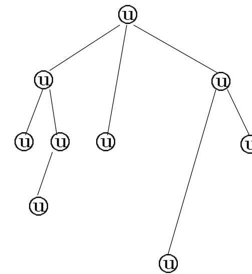
An extremal solution consists in moving all the useful clocks of the hierarchy up to the main clock of the program, thus present on all the processors. But as the reduction is an improvement of the distributed program, thus theoritically leading to better implementations, it must preserve the richness of the hierarchy. What we propose here is an intermediate solution.

Let us consider a clock c useful on $\mathcal{P}$. If it is not produced on $\mathcal{P}$, as it must be implemented (because used on $\mathcal{P}$), it must be read from the processor $\mathcal{Q}$ producing it. As clocks are read as booleans, the problem is to determine at which clock c will be read (let us note this clock r). This clock must obviously be an ancestor of c in the hierarchy (quicker than c). If we want to extract from the graph a maximum number of intermediate clocks while preserving the hierarchy, the best choice for r is the nearest useful ancestor of c, noted n (this way, all the intermediate clocks between n and c become useless). As the boolean used to communicate the clock c must be written and read at the same clock, n must be present on $\mathcal{Q}$, which is not always verified in practical cases.

$\rightsquigarrow$ **If n is useful on $\mathcal{Q}$,** c can be communicated when n is present:



On $\mathcal{P}$, b_c is read and $c_\mathcal{P}$ is produced when n is present; on $\mathcal{Q}$, $c_\mathcal{Q}$ is still produced when its father in the clock-hierarchy is present and $c_\mathcal{Q}$ is written to $\mathcal{P}$ when n is present. The two intermediate clocks between c and n can be extracted from the processor $\mathcal{P}$. If all the clocks can be communicated this way, the gain is optimal and all the intermediate clocks are suppressed. With our example, this leads to the following hierarchy on $\mathcal{P}$:

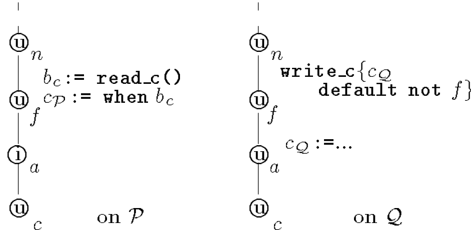


$\rightsquigarrow$ **If n is not useful on $\mathcal{Q}$,**

- One possibility is to communicate c at a clock faster than n. In the worst cases, this would lead to make all the communications at the fastest clock of the program. This way, a part of the hierarchy would be lost, so this solution can not be taken into account.

hal-00544057, version 1 - 7 Dec 2010

- Another possibility is to force n to be implemented on $\mathcal{Q}$. That would result in increasing the computation frequency on $\mathcal{Q}$ if n is faster than the quickest useful clock of $\mathcal{Q}$. This solution is rejected.
- As c is produced on $\mathcal{Q}$, its immediate ancestor a is useful on $\mathcal{Q}$. This shows that there is at least one clock between n and c useful on $\mathcal{Q}$. We choose to communicate c at the fastest clock f between n and c (in the worst case, this clock is a). This intermediate clock f on $\mathcal{P}$ becomes useful. In our previous example, if a is different from f, we see that a can be extracted from $\mathcal{P}$:



On P, b_c is read and $c_{\mathcal{P}}$ is produced when f is present; on $\mathcal{Q}$, $c_{\mathcal{Q}}$ is still produced when its father in the clock-hierarchy is present and $c\mathcal{Q}$ is wrote to $\mathcal{P}$ when f is present. All the (intermediate) clocks between c and f can be extracted from the clock hierarchy.

Let us remark that in the worst cases, this method sets new clocks as useful in the hierarchy only among the ancestors of c. As control communications must be introduced on the whole hierarchy, the traversal of the clock-tree must be made from the leaves up to the root. Moreover the clocks must be treated one by one on each processor: if we apply this method processor by processor on the hierarchy we may not benefit from the clocks introduced by the communications needed on the others processors.

4.3 Root clocks

Let q be the quickest useful clock on a processor $\mathcal{P}$ after purification of the hierarchy (thanks to the previous algorithm, q is the latest usefull clock encountered). If q is not the main clock of the program, it is produced by another processor and sent to $\mathcal{P}$. When q is absent, no computations are needed on $\mathcal{P}$ and thus there is no need to communicate q to $\mathcal{P}$: only the present (true) values of the clock are interesting. We can then communicate q only when it is present. Moreover, the communication clock does not need any more to be faster than q:

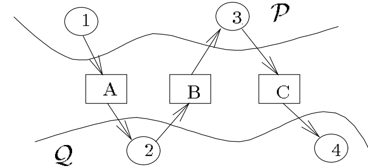
Finally, when q is the quickest clock of a process $\mathcal{P}$, only the present values of q need to be sent to $\mathcal{P}$ and the communication clock can be q itself[12].

The roots on each processor are detected during the purification of the hierarchy and stored in an array called `root`. When introducing communication nodes, the algorithm checks whether the current clock is root on the current processor or not and sets up correct communications.

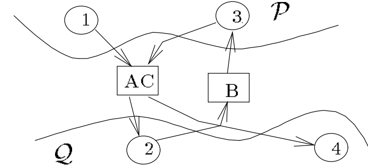
[12] In an implementation, the reception of a false value can be the signal for the processor $\mathcal{P}$ to end its execution.

4.4 Gathering communications

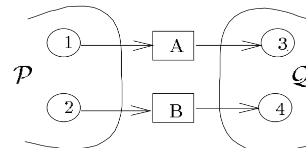
As the number of communications is one of the quality criteria of distribution, we want to gather communications if possible. First of all, because of the formalism of the SDFG, two comm-nodes can be grouped if fired at the same clock. Secondly, it is only interesting if the source and destination processors are the same. But these conditions are not sufficient; indeed, grouping nodes this way may introduce circuits in the SDFG. Let us consider the following SDFG, where all the nodes are synchronous (fired at the same clock c):



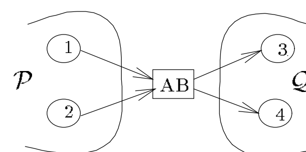
If we agregate the comm-nodes A and C in a single comm-node AC, we introduce a circuit:



Of course, circuits have to be rejected. When the agregation of many comm-nodes into a single one does not introduce circuits, it is possible but not always wanted because it can reduce concurrency by forcing a new synchronization (in one instant) in the program. Let us now consider this new example:



All the nodes are assumed to be synchronous again. If we gather the comm-nodes A and B (see below), we make the nodes 3 and 4 to be ready to be executed at the same moment when one can be executed independently from the other one:



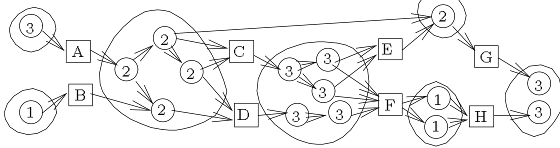
In order to prevent useless reduction of the concurrency, we gather comm-nodes if and only if 1) their source and destination processors are the same, 2) they are fired at the same clock, 3) their successors on the destination processor are the same, and 4) their predecessors on the sourse processor are the same[13]. The assertion 2) tells us that the optimization should be done by any traversal of the clock-hierarchy while the third one insures that no new circuit is introduced in the graph.

[13] In fact, this condition can be extended later by "their predecessors belong to the same cluster" when considering non-atomic nodes.

hal-00544057, version 1 - 7 Dec 2010

5 Extraction of sub-graphs

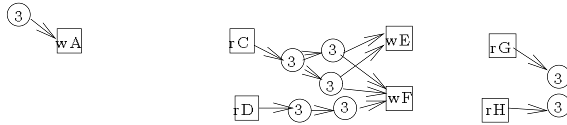
We said before that the graph obtained after transformation keeps all the properties of the initial graph. By simply extracting the sub-graphs from the complete graph, we lose the temporal dependencies between the newly introduced read/write nodes and it could lead to incorrect implementations. To convice the reader of this, let us consider the following graph:



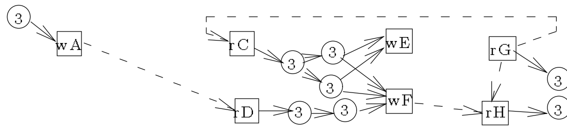
Computation and comm-nodes are represented respectively by circle and squares, each comm-node standing for one write-node and its corresponding read-node.

5.1 Direct extraction

If we simply extract from this graph the nodes located on the processor 3, we get the sub-graph



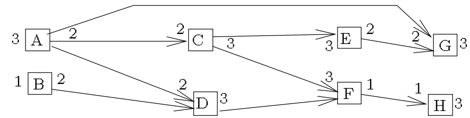
which can, in the absence of additionnal information lead to incorrect implementations; here is a possible reinforcement of the dependencies made by a static scheduler, for instance:



Obviously, though this reinforcement does not introduce any cycle regarding to the sub-graph, as there exists some dependence between the couples of read/write nodes, it would lead to an easilly predictable dead-lock. This matter is fully described in [6] and solved by the "abtract graph method". What we propose here is just another algorithm to get correct sub-graphs: firstly, we make explicit the dependencies between the read/write-nodes of the whole SDFG by a transitive closure applied on each processor; secondly, we show a fast algorithm (applying to these nodes only) to get the minimal set of dependencies to add to the read/write-nodes of each processor to extract independant sub-graphs.

5.2 Reinforcement of sub-graphs

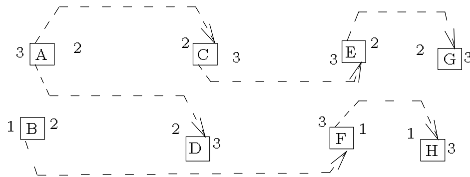
To be sure that any correct reinforcement[14] of the dependencies by a scheduler leads to a correct execution, we must add some dependencies between the input/output nodes of the graph. Moreover the dependencies should be minimal: a too strong reinforcement prevents from getting some possible executions. A possiblity consists in a transitive closure on the complete graph. Already implemented in the SIGNAL compiler, this easy solution is quite expensive because the only dependencies we want to add are dependencies from write-nodes to read-nodes on the same processor. Firstly, we make explicit the dependencies from comm-nodes located on the same processor. Applied to the previous graph, we get the following dependencies:



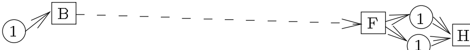
After this *selective* transitive closure on each processor, a global transitive closure applied to the very limited subset of communication nodes reveals the necessary dependencies that should be added. This abstraction of the subgraphs to their interface results in faster and correct extractions. The algorithms leading to the extractions will be explained in a complete version of this paper.

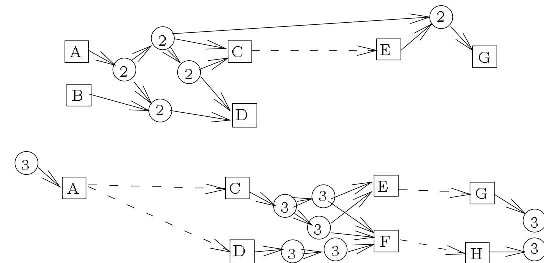
5.3 Example

Applied to our previous example, The new dependencies added by the algorithm are:



Projected on each processor, we see on this example that necessary and sufficient dependencies have been added:





6 Simulation

All the previous modifications of the SDFG lead to one sub-graph per processor. We have proved in the previous sections that the composition of these sub-graphs is equivalent to the initial graph but a new issue, specific to distributed systems, appears for simulation. On a single processor, the execution of consecutive instants is exclusive because all the instructions of an instant must be terminated before the following instant starts. On a distributed system, without any additionnal information, the processor producing the main clock of the program (predecessor of any node) may be ready to execute an instant while the execution of the previous one is not ended on other processors. Obviously, this is in contradiction with the semantics of SIGNAL that tells that instants are successive, but may

[14] Briefly, a reinforcement is said correct if it does not introduce circuits in the SDFG.

hal-00544057, version 1 - 7 Dec 2010

be wanted by the user to get data-flow-like simulation. In this section, we see some possible implementations leading to:

- synchronous executions, where the execution of an instant can not start while the previous instant is not ended;
- asynchronous executions, where overlays between instants are allowed but bounded, controlled by FIFO-queued communications and validated at compile-time;

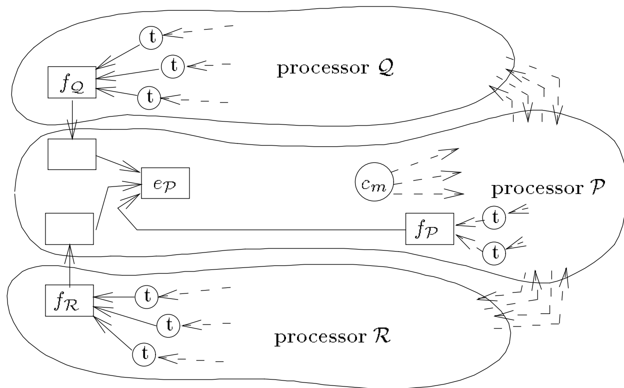
After those different macro-behaviours, we show the implementations of the sub-graphs and finally the communications between processors.

6.1 Synchronous executions

We want here to infer a synchronous execution preserving the semantics of SIGNAL: the execution of an instant can not start while the previous instant is not ended. This can be done if and only if the processor $\mathcal{P}$ producing the main clock c_m of the program is informed of the end of the execution of the current instant on all the other processors[15]. As we want to keep on with the SIGNAL formalism, this will be done in three steps.

- We note $\mathcal{T}_\mathcal{Q}$ the set of computation and clock nodes without any successor[16]. For each processor, if $\mathcal{T}_\mathcal{Q}$ is not empty, we introduce a virtual node $e_\mathcal{Q}$ succeeding to all the elements of $\mathcal{T}_\mathcal{Q}$. This way, the execution of an instant is ended when $e_\mathcal{Q}$ is executed. As the execution of $e_\mathcal{Q}$ must be transmited to $\mathcal{P}$, if $\mathcal{P} \neq \mathcal{Q}$, $e_\mathcal{Q}$ is a write-node sending a dummy value to the processor $\mathcal{P}$.
- We introduce on $\mathcal{P}$ the corresponding read-nodes $e_{\mathcal{Q}_\mathcal{P}}$, all of preceeding another new node called $e_\mathcal{P}$.

The nodes added by this transformation are shown on the following graph:



where the nodes belonging to $\mathcal{T}_i$ are represented by the symbol ⓣ. Each node e_i is set to the main clock of the process i. It is not the slowest one; indeed, the upper bound of all the clocks of the ⓣ-nodes of the processor i is the best one but it may not be present in the SDFG on i and its introduction may introduce circuits as seen in 3.5. It is easy to see that the three steps do not introduce circuits because the ⓣ-nodes had initially no successor. The reader may note that the node $e_\mathcal{P}$ is not necessary: the only presence of the read-nodes $e_{i_\mathcal{P}}$ insures that there will be no overlay between instants if c_m is not fired before the previous executions of these nodes is ended.

As well as the desynchronizations introduced by the oversampling in the communications of clocks and the reduction of the main clock on each processor, the transformations above preserve the observationnal semantics of the initial program. The next execution overview below does not.

6.2 Asynchronous executions

Synchronous executions can be interpreted seen from the previous processor $\mathcal{P}$ (producing the main clock) as if the other processes were acting on its authority: the only desynchronizations observed are *in the instant*. In other words and with a temporal point of view, as overlays between instants are not allowed, the gain of time is minimal because the processors are kept idle in particular since the moment they achieve their execution until the beginning of the next instant when they could start its execution.

What we describe here is another desynchronization of the program: if we want to make preemption possible by idle parts of the program, it is easy to see that the SIGNAL formalism is not sufficient because SDFG only deal with static properties. Asynchronous executions require some deep transformations that can not be performed without a fine knowledge of dynamic behaviours: another modelization, describing the execution of synchronous programs is needed.

Indeed, let us assume that we do not introduce the previous nodes e_i and $e_{i_\mathcal{P}}$. As some processors may execute their sub-graphs faster than other processors, without any acknowlegment, such asynchronous executions can lead to the accumulation of tokens on the communication media between the processors. A simple way to resolve this problem is to bound the desynchronisation between two processors, by having FIFO-queued communications. This also means a dynamic scheduler refusing the emission of values by write-nodes if the corresponding FIFO is full. Obviously, these asynchronous implementations require some long preliminary studies that we can not develop here.

6.3 Scheduling

During the extraction of sub-graphs onto the processors, dependencies between comm-nodes located on the same processor have been added to allow any static scheduler to rule the execution of sub-graphs. Thus, a possible choice for the implementation of sub-graphs is to generate statically sequenced code; all the techniques described in [6] can be applied to the sub-graphs. On a single processor, executable code statically sequenced at compile-time always makes efficient programs. Choices made for the scheduling do not have any effect on losses due to idle periods. On many processors, an "at least partially dynamic" scheduler is needed because idle periods in a statically sequenced code can be reduced only with a fine knowledge of execution costs on the different processors. As SIGNAL specifications are architecture-independent, those informations are missing and scheduling must be made at

[15] because c_m is preceeding all the nodes of the SDFG.

[16] The read-nodes always have succesors on $\mathcal{Q}$ and the write-nodes on another processor.

run-time. Of course, a dynamic scheduler is much more expensive than a static one. The final implementation should then statically sequence a maximum of nodes of the sub-graphs while leaving enough freedom between statically sequenced parts to benefit from concurrency for the best.

To achieve this goal, we perform clustering techniques shown in [11] on each sub-graphs to get clusters that can be implemented in a procedural way.

6.4 Communications

The first implementations of the distribution of SIGNAL programs has been made on UNIX; processors are UNIX processes (possibly on many machines of different sites). As one of our goals is to finally provide separate compilation and hardware/software implementations, we wanted an abstraction of communications. The first implementations are currently made on the P.O.M. (Parallel Observable Machine [12]) developped at I.R.I.S.A. in the PAMPA team.

In a near future, we plan to use CORBA [13]. Indeed, CORBA provides a standardization of object-oriented communications between applications. The abstraction of physical communications is made through an IDL[17] object-oriented and commonly defined by many hardware designers and software developpers. Final implementations should use the support of a run-time kernel and libraries. The programs generated will shift the responsability of communications on to CORBA's Broker, used as a "black box". Of course, the use of such a standard reduces the performances of the final implementation, but allows more portability.

7 Perspectives

We have shown a complete method to generate distributed implementations from a SIGNAL program. What we did not mention here is the way the nodes of the SDFG were assigned to processors. From our point of view saying that the distribution motivations are only qualitative, the user's directives should be set in the SIGNAL source program to make his work easier. Moreover, this way the directives can be kept at the source level through successive improvements of the specification while the graph level is just a temporary representation, invisible by programmers. These feature is currently implemented in version V4 of SIGNAL.

A big improvement of the control distribution would be the definition of cost functions with quantitative informations on durations. As this is not part of SIGNAL scope because of the architure-independence, the method presented in section 4 to reduce the clock-hierarchies on the processors could be adaptated to quantitative tools.

Finally, and this is the main part of our future work, SIGNAL specifications should allow asynchronous executions. Our present studies have shown the limits of the SDFGs, ruling only static properties. This leads us to define a new model for the execution of synchronous programs able to represent dynamic behaviours of applications.

[17] Interface Description Langage, described in [13].

References

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

[2] 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.

[3] F. Boussinot and R. De Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9):1293–1304, September 1991.

[4] Paul Le Guernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, septembre 1991.

[5] Loïc Besnard. *Compilation de* SIGNAL*: horloges, dépendances, environnement*. PhD thesis, Université de Rennes 1, France, September 1992. in french.

[6] O. Maffeïs. *Ordonnancements de graphes de flots synchrones; Application à* SIGNAL. PhD thesis, Université de Rennes 1, France, January 1993. in french.

[7] Mohammed Belhadj. *Conception d'architectures en utilisant* SIGNAL *et VHDL*. PhD thesis, Université de Rennes 1, France, December 1994. in french.

[8] Albert Benveniste, Paul Caspi, Paul Le Guernic, and Nicolas Halbwachs. Data-Flow Synchronous Languages. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Lecture Notes in Computer Science 803, Proc. of the REX School/Symposium, Noordwijkerhout, Netherlands*, pages 1–45, Springer–Verlag, June 1993.

[9] C. Lavarenne, O. Segrouchni, Y. Sorel, and M. Sorine. The SYNDEX software environment for real-time distributed systems design and implementation. In *European Control Conference*, pages 1684–1689, June 1991.

[10] Bruno Chéron. *Transformations syntaxiques de programmes* SIGNAL. PhD thesis, Université de Rennes 1, France, September 1991. in french.

[11] Bernard Le Goff and Paul Le Guernic. The granules, glutton: an idea, an algorithm to implement on multiprocessor. In R. Cori M. Wirsing, editor, *STACS 88, Lectures Notes in Computer Science*, AFCET, Springer-Verlag, Bordeaux France, February 1988. Volume 294.

[12] F. Guidec and Y. Mahéo. *POM: a virtual parallel machine featuring observation machanisms*. Research report 902, IRISA, January 1995.

[13] OMG, editor. *The Common Object Request Broker: Architecture and Specification*. Object Management Group, 1992.

hal-00544057, version 1 - 7 Dec 2010