Proceedings of the 40th IEEE
Conference on Decision and Control
Orlando, Florida USA, December 2001

The SIGNAL declarative synchronous language : controller synthesis & systems/architecture design [1]

Albert Benveniste, Patricia Bournai, Thierry Gautier,
Michel Le Borgne, Paul Le Guernic, Hervé Marchand [2]

Abstract

Dynamical systems working have been recognized as essential in the area of computer science, under the name of *reactive systems* by David Harel. *Synchronous languages* have been proposed as a paradigm to deal with reactive systems and develop tools for them. In this paper we introduce synchronous programming paradigm via the notion of *multiclock dynamical systems* and illustrate it via the SIGNAL language. We give an outline of controller synthesis in SIGNAL, and system/architecture design.

Keywords : discrete event systems, reactive systems, distributed architectures, embedded code generation.

1 Introduction

Reactive systems and synchronous languages. Dynamical systems working on-line and in closed loop with their environment are the central concept of control science. More recently, the same concept has been recognized as essential in the area of computer science, under the name of *reactive systems* by David Harel [1]. *Synchronous languages* have been proposed as a paradigm to deal with reactive systems and develop tools for them [2, 3]. The french community has been active in this area [4] [5] [6], but other formalisms are also considered synchronous [7]. In this paper we introduce synchronous programming paradigm via the notion of *multiclock dynamical systems* and illustrate it via the SIGNAL language.

Multiclock dynamical systems. Discrete time dynamical systems of the generic form

$$\begin{array}{rcl} x_k & = & f(x_{k-1}, u_{k-1}) \\ y_k & = & g(x_k, u_k) \end{array} \qquad (1)$$

are familiar to control engineers. Modeling larger systems requires combining systems of equations of the form (1). When regarded globally, the result is generally an implicit or descriptor dynamical system. Therefore, the generic form (1) should be replaced by considering systems of equations of the generic implicit form :

$$\mathcal{C}(x_k^j, x_{k-1}^j, \ldots, x_{k-p^j}^j, j = 1, \ldots, J) \qquad (2)$$

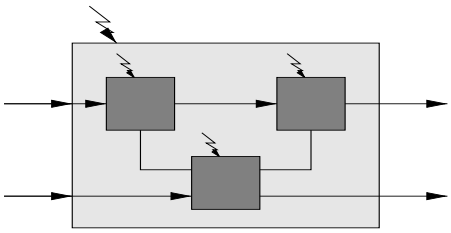
where $\mathcal{C}$ denotes a set of relations or constraints, and x^j denote the variables involved — we do not distinguish between input, state, and output variables. Such a *behavioral* approach has been advocated and extensively studied in the linear system case in particular by Jan C. Willems [8]. Note that a discrete event system can also be modeled in this way, by resorting to discrete state variables to encode states.

When dealing with complex systems, however, it is not acceptable to be bound to a single, global, time index "k". Different subsystems or components of the system may have their own, local, natural pace. Sensors and actuators have their own sampling rates, and sampling is sometimes even irregular. System monitoring involves event detection and event based reconfiguration. Therefore software components in charge of the reconfiguration are typically triggered by the detected events.

Based on this remark, we need to allow, in dynamical systems of the form (2), the use of *multiple* time indices $k_1, \ldots, k_L$, and the number L of these different "clocks" may be very large indeed. Of course, some constraints may involve signals having different time indices, e.g., when possibly event-based downsampling (i.e., filtering events) or upsampling (i.e., inserting new events) occurs. Handling time indices explicitly becomes rapidly cumbersome, and instead provision for manipulating multiple clocks easily needs to be provided. A simple idea consists in introducing a special value, written $\perp$ (pronounce "absent") to refer to absence. Therefore, for a signal $(x_k)_{k\geq 0}$ in the usual sense, two successive occurrences x_k and x_{k+1} can be separated by an arbitrary but finite number of $\perp$'s. These $\perp$'s should be regarded as wildcards to indicate that other signals may be present at a given instant while the considered one is not. Domains of variables are extended with this special value, and so are relations involving variables. By doing so, relations involving signals with different clocks are easily considered. The following picture illustrates this concept.

[1] This work is or has been supported in part by the following projects : Eureka-SYNCHRON, Esprit R&D -SACRES (Esprit project EP 20897), Esprit LTR-SYRF (Esprit project EP 22703).

[2] Irisa/Inria, Campus de Beaulieu, 35042 Rennes cedex, France ; corresp. author email : Albert.Benveniste@irisa.fr, etc. See http://www.irisa.fr/sigma2/benveniste/home.html and http://www.irisa.fr/ep-atr/welcome.english.html



Each subsystem has its own activation clock (for instance, a clock representing the greatest set of instants at which there is at least one event in the subsystem), depicted by the different lightnings, and each different signal has its own clock. This clearly requires mechanisms for data dependent up/down-sampling. Some branches of the graph are not directed, revealing the relational nature of the connections. system structure applies inductively, in a hierarchical way.

The essence of synchronous programming. Based on the above discussion, we feel the following features are indeed essential for characterizing this paradigm:

1. Programs progress via an infinite sequence of *reactions*: $P = R^{\omega}$, where R denotes the family of possible reactions, and superscript $.^{\omega}$ denotes infinite concatenation of reactions.
2. Within a reaction, decisions can be taken on the basis of the *absence* of some events.
3. When it is defined, parallel composition is always given by taking the conjunction of associated reactions: $P_1 \| P_2 = (R_1 \wedge R_2)^{\omega}$.

The SIGNAL **environment.** Based on this paradigm, the SIGNAL language has been developed, with associated set of tools: the academic SIGNAL/SIGALI toolset (graphical editor, compiler, code generator, model checker, simulator) [9], and the commercial SILDEX tool [10]. Before giving formally the semantics of SIGNAL let us just say that a SIGNAL program describes relations between flows of data and events. The compiler transforms the program into a system of equations and then calculates the solutions of the system. The compilation of SIGNAL code provides a dependence graph on which static correctness proofs can be derived: it automatically checks the network of dependencies between data flows, and detects causal cycles, temporal inconsistencies from the point of view of time indexes. SIGNAL automatically synthesizes the scheduling of operation involved inside a control loop (note that this work is often an error-prone task when done by hand in classical C-like language), and this scheduling is proved to be correct from the point of view of data dependencies. Further, the compiler synthesizes automatically *global optimizations* of the dependence graph, following different criteria. Then, according to some formal transformations of the graph, the user can choose to generate either *Embedded code* or code dedicated to simulation, or performance evaluation. At the same time, SIGALI, the model checker (also used for controller synthesis purposes) allows us to prove dynamical properties. All these functionalities are integrated in the SIGNAL environment, which is organized around the hierarchical synchronized data-flow graph. In that sense, the whole design process requires no manual transformation of models from one tool to another. SIGNAL can then be seen as a fully integrated environment. In this overview paper, we have decided to focus on two topics to which our group has contributed significantly, namely *controller synthesis,* and *systems/architecture design and distributed code generation.*

2 The SIGNAL language

The SIGNAL language allows to specify multiclock dynamical systems following a block-diagram style. Blocks represent dynamical systems, and can be connected together, to form higher blocks, and so on. Multiclock dynamical systems involve *signals* and relate them together via operators. Signals are typed sequences (boolean, integer, real,...) whose domain is augmented with the special symbol $\perp$ to denote absence. In SIGNAL, symbol $\perp$ is not handled explicitly by the user, this prevents the user from manipulating explicitly time indices. SIGNAL has a small number of primitive constructs, listed below [1].

syntax	**description**
`Z := X op Y`	$Z_\tau \neq \perp \Leftrightarrow X_\tau \neq \perp \Leftrightarrow Y_\tau \neq \perp$ $\forall k : Z_k = \mathrm{op}(X_k, Y_k)$
`Y := X$1` (delay)	$X_\tau \neq \perp \Leftrightarrow Y_\tau \neq \perp$ $\forall k : Y_k = X_{k-1}$
`X := U when B`	$X_\tau = U_\tau$ when $B_\tau =$ *true* otherwise $X_\tau = \perp$
`X := U default V`	$X_\tau = U_\tau$ when $U_\tau \neq \perp$ otherwise $X_\tau = V_\tau$
`(|P|Q|)`	compose `P` and `Q`

In this table, subscript X_τ denotes the occurrence of signal X at an arbitrary instant τ. Note that the first two statements are *single-clocked*, as all signals involved must have the same clock. In these first two statements, integer k indexes the instants at which the above mentioned signals are *present*, and "op" denotes a generic operation $+, \times, \ldots$ pointwisely extended to sequences. Note that index "k" is not mentioned explicitly in the syntax but is rather handled implicitly. In the third statement, `B` is a boolean signal and *true* denotes the value "true". In the last statement, `P,Q` denote programs (i.e., blocks, subsystems) and `(|P|Q|)` is their composition.

The first two statements set constraints on the clocks of the involved signals. Using this feature, arbitrary constraints can be expressed as follows: pick a boolean-valued expression `C(X,Y,Z)` involving three signals with identical clocks. Set `B := C(X,Y,Z)` and write statement `B := B when B`. The last statement expresses that `B` on the one hand, and `B when B` on the other hand, should be equal. But the latter selects those occurrences of `B` where `B` takes the value *true*.

[1] We list only primitive statements. The actual syntax involves derived operators, in particular for handling constraints on clocks.

Therefore the program

```
(| B := C(X,Y,Z) | B := B when B |)
```

states that either `X,Y,Z` are all absent, or condition `C(X,Y,Z)` is satisfied.

3 Controller synthesis

Controller synthesis consists in automatically synthesizing a controller from a partial specification (the "plant") together with control objectives. Running the controller in parallel with the plant yields the desired reactive system. In this section we discuss controller synthesis in the framework of SIGNAL. We here restrict ourselves to multiclock dynamical systems involving only boolean and *clocks* (i.e., pure signals with domain $\{true, \perp\}$). Such SIGNAL programs are equivalent to arbitrary finite state machines. Controller synthesis is conveniently performed by embedding SIGNAL into dynamical systems over the finite field $\mathbb{Z}/3\mathbb{Z} = \{\Leftrightarrow 1, 0, +1\}$ with special rules $1+1 = \Leftrightarrow 1$, and $\Leftrightarrow 1 \Leftrightarrow 1 = +1$. The following coding is used *true* $\mapsto +1$, *false* $\mapsto \Leftrightarrow 1$, $\perp \mapsto 0$. Using this coding, SIGNAL primitives translate as follows:

`B := not A`	$b = -a$
`C := A and B`	$c = ab(ab-a-b-1)$ $a^2 = b^2$
`C := A or B`	$c = ab(1-a-b-ab)$ $a^2 = b^2$
`B := A $1 (init` b_0`)`	$x' = a + (1-a^2)x$ $b = a^2 x$ $x_0 = \mathtt{b}_0$
`C := A when B`	$c = a(-b-b^2)$
`C := A default B`	$c = a + (1-a^2)b$

In this table, x, x' denote auxiliary current and next state variables, these are needed to encode the delay operator. Any SIGNAL specification can then be translated into a set of equations called polynomial dynamical system (PDS) of the form:

$$S = \begin{cases} X' = P(X,Y,U) \\ 0 = Q(X,Y,U) \\ 0 = Q_0(X) \end{cases} \quad (3)$$

where X, Y, U, X' are vectors of variables in $\mathbb{Z}/3\mathbb{Z}$ and $dim(X) = dim(X') = n$. The components of the vectors X and X' represent the current and next states of the system and are called *state variables*. They originate from the translation of the delay operator. Y is a vector of variables in $\mathbb{Z}/3\mathbb{Z}$, called *uncontrollable event variables*, whereas U is a vector of *controllable event variables*. The first equation is the *state transition equation*; the second equation is called the *constraint equation* and specifies which event may occur in a given state; the last equation gives the initial states. The behavior of such a PDS is the following: at each instant t, given a state x_t and an admissible y_t, we can choose some u_t which is admissible, i.e., such that $Q(x_t, y_t, u_t) = 0$. In this case, the system evolves into state $x_{t+1} = P(x_t, y_t, u_t)$.

3.1 Control synthesis problem

Given a PDS S, as defined by (3) a controller is defined by a system of two equations $C(X,Y,U) = 0$ and $C_0(X) = 0$, where the latter equation $C_0(X) = 0$ determines initial states satisfying the control objectives and the former describes how to choose the instantaneous controls; when the controlled system is in state x, and an event y occurs, any value u such that $Q(x,y,u) = 0$ and $C(x,y,u) = 0$ can be chosen. The behavior of the system S composed with the controller is then modeled by:

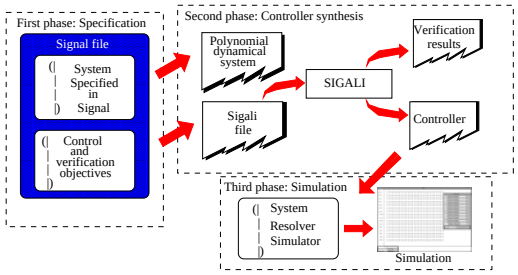
$$S_c = \begin{cases} X' = P(X,Y,U) \\ 0 = Q(X,Y,U) = C(X,Y,U) \\ 0 = Q_0(X) = C_0(X) \end{cases} \quad (4)$$

Using algebraic methods, avoiding state space enumeration, we can compute controllers (C, C_0) which ensure:

- the *invariance* of a set of states (`S_Invariance()`), the *reachability* of a set of states from the initial states of the system (`S_Reachability()`), the *attractivity* of a set of states E from a set of states F (`S_Attractivity()`) [12],
- the *minimally restrictive control* (choice of a control such that the system evolves, at the next instant, into a state where the maximum number of uncontrollable events is admissible (`S_Free_Max()`)) [12], as well as the *stabilization of a system* (`S_Stab()`)) [11].

3.2 Integration in the SIGNAL environment

We sketch how the controller synthesis methodology has been integrated in the SIGNAL environment. To simplify the use of the tool, the same language is now used to specify the physical model of the system and the control objectives (as well as the verification objectives). We have developed a tool allowing the controller synthesis as well as the visualization of the result by interactive simulation of the controlled system, see next figure. In the first stage, the user specifies the physical model and the control objectives in SIGNAL. The second stage is performed by the SIGNAL compiler which translates the initial SIGNAL program into a PDS and the control objectives in terms of polynomial relations and operations. The controller is then synthesized, using SIGALI. In the third stage, the obtained controller is included in the original SIGNAL program in order to perform simulation.



First phase: Specification of the model. The physical model is first specified in the language SIGNAL. It describes the global behavior of the system. In the same stage we specify a process, that describes all the properties that must be

enforced on the system. Properties to be checked as well as the control objectives to be synthesized, can be expressed directly in the SIGNAL program. The syntax is shown below:

```
(| Sigali(Control_Objective(PROP)) |)
```

The keyword `Sigali` means that the subexpression has to be evaluated by SIGALI. The function `Control_Objective` means that SIGALI has to compute a controller according to the boolean `PROP`, which can be seen as a set of states in the corresponding PDSin order to ensure the control objective for the controlled system (it could be one of the control objectives presented in section 3.1). The overall SIGNAL program is obtained by composing the two processes.

Second phase: Controller Synthesis. To perform the computation of the controller with regard to the different control objectives, the SIGNAL compiler produces a file which contains the PDS resulting from the abstraction of the complete SIGNAL program and the algebraic control (as well as verification) objectives. We thus obtain a file that can be read by SIGALI. Suppose that we must enforce, in a SIGNAL program named "system.SIG" the invariance of the set of states where the boolean `PROP` is *true*. The corresponding SIGNAL program is:

```
(| (| system{} (physical specified in Signal) |)
 | PROP : definition of the boolean PROP in Signal
 | Sigali(S_Invariance(True(PROP)) |)
```

The corresponding SIGALI file, obtained after the compilation of the global SIGNAL program, is the following:

```
read(''system.z3z'');  => loading of the PDS ''S''
Set_States: True(PROP) => states where PROP is true
S_c: S_Invariance(S,Set_States) => Synthesize the
controller ensuring  the invariance of  Set_States
```

The file "system.z3z" is the PDS that represents the initial system. The `PROP` signal becomes a polynomial `Set_States` expressed by state variables and events, which is equal to 0 when `PROP` is *true*. The last line of the file consists in synthesizing a controller which ensures the invariance of the set of states where the polynomial `Set_States` takes the value 0. This file is then interpreted by SIGALI that checks the verification objective and computes the controller. The result of the controller synthesis is a polynomial which is represented by a BDD (Binary Decision Diagram) which is saved in a file and used to perform simulation.

Third phase: simulating the result. To obtain a simulation that allows to visualize the new behavior of the controlled system, the controller is automatically integrated in the initial SIGNAL program through an algebraic equation resolver written both in SIGNAL and C^{++} (some generic processes for simulation can be added at this stage). The reader is referred to [12] for examples and additional details on the implementation.

4 Systems/architecture design

When moving from the SIGNAL specification based on the ideal model of synchrony, to the real embedded code for execution, we are faced with the following difficulties:

1. Referring to the table of section 2, we see that SIGNAL handles implicit (or descriptor) type of systems. Indeed, generating executable code from descriptor type of specification is a difficult task. We shall outline our approach for this problem. For details, the reader is referred to [13].
2. The ideal model of synchrony, as summarized at the end of section 1, does not comply with the actual behavior of distributed control systems, in which some kind of asynchrony is expected. We shall outline a *theory of desynchronization* as a possible solution for this issue. The reader is referred to [13] for details.

4.1 From implicit specifications to executable code: causality analysis

Unlike behavioral theory for *linear* dynamical systems as extensively studied by Jan C. Willems, moving from implicit specifications to an executable equivalent form is undecidable for multiclock dynamical systems in general. We shall therefore follow an approach akin to so-called "abstract interpretation", in which an *approximate* solution is searched for, that applies to all cases. The idea is that

1. we shall fully handle equations of multiclock dynamical systems involving booleans or clocks, whereas,
2. we shall replace statements of the form `Z := X op Y` by their following "approximation"

   ```
   (| Z ^= X ^= Y %clocks must be equal
    | (X,Y) --> Z %causality constraint |)
   ```

   The first statement is just the SIGNAL syntax to express the equality of the clocks of `X,Y,Z`. The second statement is a new SIGNAL primitive. Its general form is `(X,Y) --> Z when B` which states that, *when* `Z` *is present and* `B` *is true,* `Z` *cannot be produced prior to the pair* `X,Y`.

This abstraction is systematically applied using the causality rules of table 1, until a fixpoint is reached (this requires at most two steps).

statement: `P`	**causality**: `caus(P)`
`Z := X op Y`	`(X,Y) --> Z`
`Y := X$n` (n-delay)	
`X := U when B`	`(|B --> X` `|U --> X when B|)`
`X := U default V`	`(|U --> X when ^U|)` `|V --> X when ^V ^- ^U |)`
`X --> Y when B`	`B --> Y`
`(|P|Q|)`	`(|caus(P)|caus(Q)|)`

Table 1: *Causality analysis of* SIGNAL *programs*

In this table, keyword `^U` denotes the clock of `U`, i.e., the pure signal which is present exactly when `U` is present. `^V^-^U` denotes the clock representing the instants at which `V` is present and `U` is absent. Note that no causality results from the delay operator, since computing the next state is always

explicit, see (3). The causality rule ^X-->X is also applied for every signal X Using this technique, any SIGNAL program is abstracted into the following generic form, compare with (3):

$$S_{\text{abst}} \quad = \quad \left\{ \begin{array}{lcl} X' & = & P(X,Y,U) \\ 0 & = & Q_b(X,Y,U)\ ,\ \mathcal{G}(X,Y,U) \\ 0 & = & Q_{b,0}(X)\ ,\ \mathcal{G}_0(X) \end{array} \right. \quad (5)$$

In (5), constraint $0 = Q(X,Y,U)$ has been decomposed into its boolean, solvable, part $0 = Q_b(X,Y,U)$, plus a system of causality constraints $\mathcal{G}(X,Y,U)$ involving statements of the form `X --> Y when B`. And There is a similar decomposition for the initial condition. The key remark is that system (5) is now of *finite* nature, and therefore it can be transformed into an input/output form for execution. Note that "solving" (5) is more involved than just handling finite state machines, due to its hybrid nature, mixing together automata, and graphs labelled by predicates. Solving for the graph part consists of constructing a partial order compatible with the set of causality constraints of the form `X --> Y when B`.

A variation of this technique is implemented in the SIGNAL compiler. Note that, besides serving for causality analysis, statement `X --> Y when B` can serve for other purposes. In particular, it can be used simply to enforce a scheduling constraint between X and Y. Therefore, *schedulers can also be formally handled using* SIGNAL: they can be specified, and further composed. This is a key advantage in architecture modeling [13].

4.2 From synchronous specification to asynchronous, distributed architectures

Referring to the ideal model of synchrony, as summarized at the end of section 1, we can use token based dataflow networks to model asynchronous, distributed executions, see [2]. In this model, no global clock is available, and absence cannot be sensed. Therefore it is tempting to substitute SIGNAL primitive synchronous statements by corresponding dataflow actors. This is shown in table 2.

The diagrams read as follows: the diagrams on the left show the enabling condition, and the corresponding diagrams on the right depict the result of the firing. When several lines are shown for the same statement, they correspond to different cases, depending on the enabling condition. The idea is that each SIGNAL statement would be replaced by its associated dataflow actor, and thus a SIGNAL program would result in a token based dataflow network. This does not work properly as we discuss next.

The first two statements translate exactly into the actors shown in the right column. Each actor consumes one token on each input and produces one token on its outputs. The delay is modeled by the presence of an initial token inside the actor.

For the `when` statement, we propose the corresponding actor on the right column. For the boolean guard B, the black patch indicates a *true* value for the token, while a white patch indicates a *false*. When the boolean guard has a *true* token,

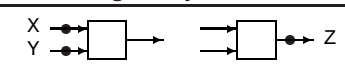
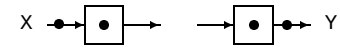
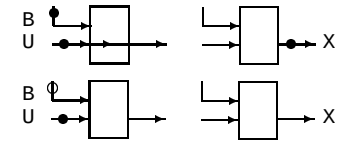
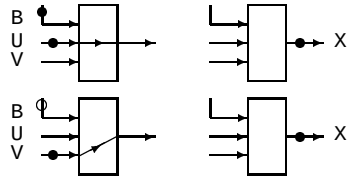
SIGNAL **statement**	**analogous asynch. actor**
`Z := X op Y`	
`Y := X$1` (1-delay)	
`X := U when B`	
`X := U default V`	

Table 2: *From* SIGNAL *statements to dataflow actors.*

the U token passes the "gate", whereas it is lost when the boolean guard has a *false* token. This performs data dependent downsampling. This actor is *not* equivalent to `when` statement, however. The `when` statement does not set any clock constraint on its inputs, while the shown actor requires equality of its input clocks (there are as many tokens on both input channels). This is why we name it an "analogous" actor, not an equivalent one. The same problem arises with the `default` statement.

However, we can emulate the `when` statement by the following means, for the particular case where we know that U is less frequent than B, see figure 1.

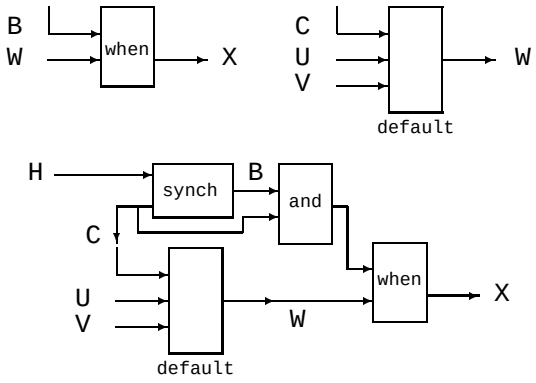


Figure 1: *A correct translation of the* `when`.

Here we use four actors. The `when` and `default` actors are specified in the above table. The `synch` actor receives one token at its input H, and delivers one token at each of its two boolean outputs B,C: this models that B,C have the same clock. The `and` actor is the boolean "and". Both `synch` and `and` actors are single-clocked and memoryless, i.e., they are of the type listed in the first row of table 2. The `when` actor requires B,W to have the same clock, and the `default` actor requires that U is less frequent than W. This models that U is

less frequent than B. Then it is easily verified that U passes the network exactly when 1/ it is present (i.e., C is *true*), and 2/ B is *true*. A similar but more complex solution can be found for the when statement in the general case.

This work can be systematically performed, for each SIGNAL primitive statement. Note that this translation requires introducing additional signalling and operators. This overhead is required to maintain the semantics when no global clock is available to define the successive reactions. Clearly, this naive method would result in an unacceptable overhead and is not applicable. However, this method can be applied instead to the overall program, using a powerful symbolic analysis of the clocking of this program. This results in a minimum overhead in terms of additional signalling and operators needed. For a formal theory supporting this technique, the reader is referred to [13, 14]. This approach can be seen as a way to systematically synthesize the needed protocols to maintain the program semantics when a distributed implementation is performed, using an asynchronous communication architecture. It is implemented in the SIGNAL Inria compiler [9] as well as in the commercial SILDEX tool [10].

4.3 Code profiling

Finally, we briefly mention our approach to code profiling, to evaluate, e.g., performance. Given an implementation Q of a program and a model of time consumption for each of the atomic actions in Q, we automatically generate a program $T(Q)$ homomorphic to Q; $T(Q)$ is the parallel composition of the images $T(Q_i)$ of the subcomponents Q_i (including communications) of Q. $T(Q_i)$ are given by the user as SIGNAL components whose interfaces are composed of integer flows $T(x)$ instead of the original flows x. For example, $T(x)$ can be used to represent the sequence of the availability dates for the occurrences of the original flow x. $T(Q)$ is thus a model of real time consumption of the application (functional specification and architectural support), that can be simulated. This can give directly access to the maximum time necessary to perform a computation cycle [15]. Some other real time properties to be satisfied can also be described as predicates in SIGNAL. Then some of these properties can be checked by using verification tools.

5 Usage

In this section we summarize the tools available and their use. For further information, the reader is referred to http://www.irisa.fr/sigma2/benveniste/home.html and http://www.irisa.fr/ep-atr/welcome.english.html.

The academic SIGNAL/SIGALI **environment.** A new release of the SIGNAL/SIGALI toolset, named POLYCHRONY, is to be ftp available by autumn 2001. For the current version, contact leguernic@irisa.fr. The SIGNAL team has tight cooperations with other french groups working on synchronous languages (ESTEREL, LUSTRE). It has a long ongoing cooperation with the company TNI, Brest, France, which markets the SILDEX tool. It has or had significant cooperations with industrials in the framework of direct contracts or european projects (SACRES, and currently SAFEAIR). Major cooperations were with EdF (Electricité de France) and Snecma (an aircraft engine manufacturer).

The commercial SILDEX **tool from TNI.** The company TNI, Brest, France, markets the SILDEX tool [10] for reactive systems specification and validation, and sequential (C, Ada) and distributed code generation. SILDEX has a powerful GUI allowing a mixed state-machine/dataflow style of modelling.

References

[1] D. Harel and A. Pnueli. On the development of reactive systems. *Logics and models of concurrent systems*, K.R. Apt Ed., NATO ASI series, vol F-13, 1985, 477-498.

[2] A. Benveniste, G. Berry, "Real-Time systems design and programming", *Another look at real-time programming, Proc. of the IEEE*, vol. 9 n^o 9, September 1991, 1270–1282.

[3] N. Halbwachs, *Synchronous programming of reactive systems,*. Kluwer Academic Pub., 1993.

[4] G. Berry, *The Constructive Semantics of Pure Esterel*, draft book 3, July 2, 1999.

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

[6] P. Le Guernic, T. Gautier, M. Le Borgne, C. Le Maire, "Programming real-time applications with SIGNAL", *Another look at real-time programming, Proc. of the IEEE*, vol. 9 n^o 9, September 1991, 1321–1336.

[7] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts.* McGraw Hill, 1998.

[8] J. C. Willems. Paradigms and Puzzles in the Theory of Dynamical Systems. *IEEE Transactions on Automatic Control*, 36(3), 258–294, 1991.

[9] Inria/Irisa. SIGNAL tool. Contact leguernic@irisa.fr for the current version. SIGNALV4/SIGALI ftp available by Autumn 2001.

[10] TNI. SILDEX tool, http://www.tni.fr/frame-sommaire.eng.html

[11] H. Marchand and M. Samaan. On the Incremental Design of a Power Transformer Station Controller using Controller Synthesis Methodology. IEEE Transaction on Software Engineering, Vol 26(8), August 2000.

[12] H. Marchand, P. Bournai, M. Leborgne, and P. Le Guernic. Synthesis of Discrete-Event Controllers based on the Signal Environment Discrete Event Dynamical System: Theory and Applications, Vol 10:325-346, 2000.

[13] A. Benveniste, B. Caillaud and P. Le Guernic. Compositionality in dataflow synchronous languages: specification & distributed code generation. *Information and Computation*, 163, 125-171 (2000).
http://www.irisa.fr/sigma2/benveniste/pub/BCLg99a.html

[14] A. Benveniste, B. Caillaud and P. Le Guernic. From synchrony to asynchrony. In J.C.M. Baeten and S. Mauw, editors, CONCUR'99, Concurrency Theory, 10th International Conference, vol. 1664 of LNCS, 162-177. Springer V., 1999. See also
http://www.irisa.fr/sigma2/benveniste/pub/BCLg99b.html

[15] A. Kountouris, P. Le Guernic. Profiling of SIGNAL Programs and its application in the timing evaluation of design implementations. Proceedings of the IEE Colloq. on HW-SW Cosynthesis for Reconfigurable Systems, IEE, HP Labs, Bristol, UK, 1996.