Synchronous Languages and Reactive System Design

C²A

Abstract **Synchronous languages are devoted to programming reactive systems. They are based on the abstract point of view that programs instantaneously and deterministically react to input events coming from their environment. This paper presents three such languages, ESTEREL, LUSTRE, and SIGNAL, developed in France since the early 1980s. These languages are now provided with industrial environments, and are being used in large industrial applications.**

I. Issues in Reactive Systems Design

A. Reactive Systems

Reactive systems are computer systems that continuously react to their environment at a speed determined by this environment. Most industrial "real-time" systems are reactive — control, supervision and signal-processing systems. Such systems become critical players in our everyday environment. Embedded control systems for aircraft, railways, or even cars, are well-known examples of reactive systems. Most plants now involve complex reactive systems for monitoring and supervision. Communication protocols or man-machine interfaces are also reactive systems. The main features of these systems are the following:

They involve concurrency. By construction, a reactive system acts concurrently with its environment. Furthermore, it is often convenient and natural to view a reactive system as made of a set of concurrent components that cooperate to achieve the intended behavior. Finally, reactive systems are sometimes implemented on parallel or distributed architectures in order to increase their performance or their reliability. Let us note at once that there are two distinct levels of concurrency: the logical level where concurrency is a major tool for functional decomposition and the physical level where concurrency reflects actual machine or networking structures. Design tools should concentrate on the logical level, while compilers or synthesis systems should bridge the gap between both levels.

They may be submitted to strict timing requirements. In this case, one often speaks of hard real-time systems. Timing constraints belong to the system specifications, they must be taken into account at design time, and their satisfaction must be checked on the final implementation. Their fulfillment obviously requires efficient implementation, but it especially requires precise evaluation of execution time. Even for systems that are not subject to hard constraints, it is useful to precisely predict the speed of reactions.

They are generally deterministic. Generally speaking, the outputs of a reactive system are completely determined by the sequence and timing of its inputs. This inherent determinism distinguishes reactive systems from interactive ones (e.g., operating systems, data-base query and management), which are intrinsically nondeterministic as they involve processes which are scheduled according to varying, unknown, and unessential parameters concerning resource availability. Determinism makes the design, analysis, and debugging of systems much easier. This is a major reason to preserve determinism all the way from specification to implementation.

Their reliability is critical. This may be the most important point. It is commonplace to say that errors in reactive systems can have dramatic consequences, involving human lives and huge amounts of money. Therefore, these systems require extremely rigorous design methods and constitute a field where formal verification must be used whenever possible.

They are often made partly of software and partly of hardware. The choice of implementing a given system or a part of it on hardware or software depends on cost / performance analyses based on trade-offs that may strongly vary over time. This is a major reason to postpone implementation choices until late in the design and to make tools flexible w.r.t. architectural choices.

*C²A is the Cooperative Control research Action, an academic-industrial joint research group supported by INRIA and CNRS. Authors of this article are all active members of this group. The actual list of contributors is given in the appendix. Corresponding author is Albert Benveniste, IRISA/Inria, Campus de Beaulieu, 35042 Rennes Cedex, France, email: benveniste@irisa.fr, tel: +33 99 84 72 35, fax: +33 99 84 71 71.

If we review the available, classical design tools, we find on one hand multitasking systems driven by a real-time operating system (OS), and on the other hand general-purpose concurrent languages such as ADA. The former do not supply an acceptable level of reliability because they do not provide a global and formal view of the system, which is separated into tasks and calls for services provided by the OS; furthermore, they do not preserve application determinism. The latter are intrinsically non-deterministic and do not provide the necessary accurate real-time capabilities. For a more detailed discussion of these points, see [4] [3].

B. *The Synchronous Approach*

Synchronous languages have been specifically designed to ease the programmer's task when dealing with reactive systems. Their basic choice is to provide idealized primitives for concurrency and communication: a reaction to an input event is considered to be instantaneous, thus making output strictly synchronous with input. Interaction between concurrent components within the program is conceptually performed by broadcasting events in no time and in a fully deterministic way.

In practice, synchrony is the assumption that the program reacts rapidly enough to perceive all the external events in suitable order and to produce all the outputs for an input before reacting to a new input. For instance, automatic controllers are commonly programmed as an infinite loop over a body consisting of acquiring inputs, performing internal computations and returning outputs. The execution of the body is considered atomic. This is readily abstracted into a perfectly synchronous reaction. If practical synchrony is satisfied — and, more importantly, if its satisfaction can be checked — the conceptual synchrony hypothesis is in fact a more realistic abstraction than that which assumes that computers deal with unbounded or real numbers.

Although the synchrony hypothesis may look relatively new for software, it is well-known in other fields: it is precise the zero-delay model commonly used in digital hardware design; it is routinely used in control theory, where one always neglects the cost of numerical calculations when designing control laws.

Before describing the synchronous model further, let us compare it briefly with the more traditional model of CSP [2] and ADA. These languages are meant to be implemented on top of asynchronously distributed platforms. They use point-to-point communication and assume that all computing and communication actions take an unspecified positive amount of time. This is well tailored to interactive systems design, but largely inappropriate for reactive systems design: unspecified-time communication makes subsystem composition non-deterministic and unpredictable w.r.t. reaction time, which is just the opposite of what we want. Furthermore, non-deterministic point-to-point communication makes it difficult to handle distributed information compared to broadcasting. Conceptually speaking, synchrony is just the simplest way of handling deterministic concurrency and exact distributed information handling. Furthermore, the basic mathematics of synchrony turns out to be simpler than those of asynchrony; this will lead us to more powerful optimization, verification, and actual timing analysis techniques. On the other hand, large scale computerized systems cannot be considered as fully synchronous and cooperation between synchronous and asynchronous techniques will be required. This is the intent of mixed models such as CRP (Communicating Reactive Processes) [20]. Here we shall concentrate simply on synchronous programming.

Notice a slight vocabulary difficulty: rendez-vous in CSP and ADA is often called synchronous since its final handshake act involves a synchronization. This may create a conflict with our much stronger notion of synchrony, which consists in tightly synchronizing the paces of the components. In case of possible confusion, we shall call our hypothesis "perfect synchrony".

Events and Reactions

The underlying model common to all synchronous languages is very simple and reminiscent of basic automata and discrete control theories. A perfectly synchronous reactive system reacts repeatedly to a sequence of input events by producing a matching sequence of output events. For example, an input sequence to a speedometer may be "`Meter, Meter, Second, Meter, Second, ...`", and the corresponding output sequence "`-, -, 2, -, 1`" if speed is defined as the number of meters per second during the previous second. Here '`-`' is used for an empty output. The flow of inputs provides us with a logical timing system, and the behavior of the system is deterministic w.r.t. logical timing. The notion of physical (chronometric) time is replaced by a simple notion of order among events: the only relevant notions are simultaneity and precedence between events. Synchrony simply states that the indices in the output sequence match those of the input sequence in time; in other words, the speedometer outputs the actual speed at any second.

In this logical view of time, any individual signal name can count as a time unit. One can say "*the train must stop within 60 meters*" as well as "*the train must stop within 60 seconds*", both statements acting as real-time constraints of the very same nature. This is what we call multiform time. Of course, physical time is needed in many applications. At the programmer's level, it is considered as an external event, similar to any other event coming from the environment — `Second` in our example; at implementation level, particular care must often be taken to handle timing interrupts.

Simultaneity makes perfect sense in synchronous formalisms. At any logical instant, the input to a system or subsystem may be made of several elementary events that are considered to be simultaneous. For example, a possible input sequence to the speedometer is "`Meter, Meter, Meter & Second`" where "`&`" indicates simultaneity (in this case, one has to specify whether the last meter adds to the current speed or not). The paradigm of perfect synchrony is summarized and illustrated in Figure 1.

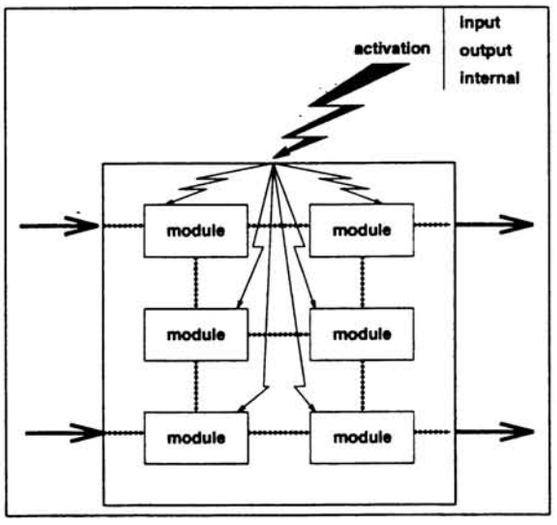


Fig. 1: This is an idealized picture of a reactive system following the synchronous paradigm. In reaction to some external stimulus, or upon self-request of the system, an "instantaneous" reaction occurs, i.e., an ideal "zero-time-tasking" is assumed. This zero-time tasking is shown by the lightning bolt, which propagates activation instantaneously to all modules at all levels of the hierarchy. External stimuli can be sensor update, interrupt or alarm handling, etc., and need not to be bound to an overall clock. Synchronous languages are formalisms in which the user can define functions based on zero-time tasking. This clearly makes specification and programming independent from hardware and firmware target architecture. Also, since "zero-time tasking" neither involves any code for execution, nor causes overhead, the architecture can be arbitrarily modified, and equivalence is still guaranteed. This helps the designer to move from the functional architecture down to the hardware or firmware supporting architecture with proven equivalence. After such modifications, "zero-time tasking" can be easily converted into a combination of sequencing and real-life real-time tasking, possibly asynchronous and distributed.

Synchronous Languages

The rest of the paper is devoted to the presentation of several synchronous languages or visual formalisms that have been built on top of the perfect synchrony hypothesis and of their support tools. Owing to the well-definedness of the synchronous model, all languages share clean formal semantics, elaborate static consistency checks, automatic generation of efficient hardware or software targets, and availability of formal verification tools.

The languages fall into two classes.

- *Imperative languages* like ESTEREL are appropriate when control handling is predominant, e.g., in mode-based man-machine interface systems, interface logic, protocols, or alarm-handling systems. Another example of a synchronous imperative language is the ARGOS graphical formalism [19], a perfectly synchronous variant of STATECHARTS [12] that was developed to overcome some semantic difficulties in the original STATECHARTS formalism.
- *Dataflow languages* like LUSTRE and SIGNAL are better suited when data-handling dominates, as in continuous process control or signal processing.

The two styles of languages match two quite distinct styles of technical cultures. ESTEREL and ARGOS are based on state machine ideas familiar to computer science-oriented engineers, while LUSTRE and SIGNAL are synchronous versions of block-diagram schemes familiar to automatic control-oriented engineers. For the time being, it is clearly important for engineers to be able to specify and program applications using ways of thinking and notations close to those of their native culture: this is the only way for them to be sure that their description correctly captures their intuition and to discover design errors as early as possible. There is yet no single solution to cover all existing cultures. In the future, a unification of the existing styles and languages will certainly become mandatory; this is an active subject of research.

Tools and Environments

As far as tools and programming environments are concerned, the requirements from a synchronous design environment are similar to those of all other software and hardware design areas. Namely, users want:

- to use editors that help them write, comment, and debug specifications and programs;
- to perform extensive simulations of the system and of its environment using a variety of tools ranging from graphical simulators to full system prototypes;
- to derive automatically the embedded executable code from the high-level description;

- to improve their productivity by reusing code instead of rewriting it, and by detecting inconsistencies as early as possible;
- to improve the software reliability either by simple means (target code readability, reuse, extensive testing) or in more sophisticated ways (formal verification, tool certification);
- to automatically maintain complete documentation;
- to convince safety authorities that the developed software is zero-default;
- to be sure of the durability of the tools they use.

Such facilities are offered by industrial environments developed around the languages. Available environments will be briefly described in turn, together with main industrial applications.

II. The Synchronous Imperative Approach

The textual synchronous language ESTEREL has been developed at the Ecole des Mines and INRIA Sophia-Antipolis since 1982. Its compilers and environments have been available since 1989.

A. *The* ESTEREL *Language [1]*

The main unit in ESTEREL programs is the module. A module has a data interface, which describes the data types used in the program, a signal interface, which declares the signals used in input and output events, and a body, which is an executable statement. We illustrate the main features of the language with an example. Here is the signal interface of a speedometer module:

```
module Speedometer :
input Second, Meter;
output Speed : integer;
```

The body consists of a free combination of imperative statements based on explicit instantaneous signal emission, "`emit S`", instantaneous signal testing, "`present S`", sequencing, '`;`', explicit concurrency, '`||`', watchdog operators "`watching S`", "`every S`", etc., and a "`trap-exit`" exception mechanism fully compatible with concurrency. Here is the body of the speedometer:

```
loop
   var Distance := 0 : integer in
      do
         every Meter do
            Distance := Distance + 1
         end every
      watching Second;
         emit Speed(Distance)
      end var
end loop
```

Here, the main statement is the watchdog "`do ... watching Se`" This construct preempts its body and terminates instantaneously as soon as `Second` occurs, disregarding the internal state of the body.

To illustrate further the ESTEREL style, we program the controller of a runner. The specification is to run slowly for 15 seconds, then jump and breathe at every step for 50 meters, and finally run fast for the remainder of two laps. The input signals are `Second`, `Meter`, `Step`, and `Lap`.

```
do
   do
      <run slowly>
   watching 15 Second;
   do
      every Step do
         <jump> || <breathe>
      end every
   watching 50 Meter;
   <full speed>
watching 2 Lap
```

Here `<run slowly>` etc. denote unspecified pieces of code, perhaps submodule calls. Programming in ESTEREL mostly consists in controlling the life and death of statements using watchdogs that monitor the multiform time units.

To illustrate the use of exceptions, we will modify the jumping phase. Since jumping is strenuous, we would like to ensure that the heart keeps beating normally. For this, we enclose the whole program in a statically scoped exception construct

```
trap HeartAttack in
   ...
handle HeartAttack do
   <go to hospital>
end trap
```

and we add a heart monitoring statement in parallel with `<jump>` and `<breathe>`. In case of heart failure, the monitoring part executes the "`exit HeartAttack`" statement that provokes immediate termination of the `trap` body, regardless of its internal state and of how many concurrent statements it contains, and immediately transfers control to the trap handler.

ESTEREL programs have a formal semantics based on the synchrony hypothesis that is given in [7].

B. *The* ESTEREL *Tools*

The tool suite includes a compiler, a graphical simulator, and verification tools.

The Compiler

The ESTEREL compiler translates ESTEREL programs into C or into hardware netlists, curently with some restrictions on the latter. Five successive generations of compatible compiling tools have been developed, ESTEREL v3 being the first industrial one. The current generation ESTEREL v5 will be distributed very soon.

An ESTEREL program is split into a data path and a control state machine that drives the data path. With ESTEREL v3, the control machine is represented as a fully explicit transition graph. The run-time code is very fast but may suffer size explosion. The ESTEREL v4 and ESTEREL v5 compiler avoid size explosion by using an implicit representation of the control state machine as a system of boolean equations on variables and registers (elementary delays), just like a synchronous hardware circuit. In the ESTEREL v4 compiler, only acyclic circuits are handled. The new ESTEREL v5 compiler also handles cyclic circuits. In the interpretation mode, it directly translates a cyclic circuit into a table-based C program. In compiling mode, it transforms a cyclic circuit into an equivalent acyclic one using algorithms based on Binary Decision Diagrams [22].

Simulation and Symbolic Debugging

The ESTEREL environment offers efficient (linear in the size of the Esterel code) simulators through which programs can be exercised from graphical control panels. Source code symbolic debugging is available during simulations, and breakpoints can be placed on the source code.

Verification

Program verification can be performed using several tools. The AUTO tool performs proofs by *reduction modulo bisimulation*, i.e., by constructing reduced images of the finite state machine according to user-definable observation criteria [8]. For example, to show that an elevator never travels with the door open, one hides all events but those related to door and engine control. Since the resulting reduced automaton is very small, it can be drawn on the screen and analyzed visually. The TEMPEST [15] tool developed at AT&T translates a temporal logic formula into ESTEREL and uses the ESTEREL compiler to prove or disprove the formula. Finally, the control circuit can be printed in circuit format and passed to various existing circuit verifiers.

III. THE SYNCHRONOUS DATA-FLOW APPROACH

A. Synchronous Data-Flow Languages

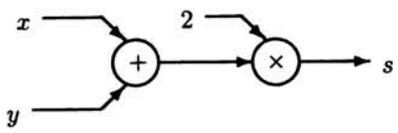
Formalisms based on block-diagrams are heavily used in safety-critical control systems (nuclear power plant control and supervision, avionics, automotive,...), in which software is strongly involved.

Designers of real-time safety-critical systems have seen their job progressively evolving because of the microprocessor revolution. Most of them were obliged to change their way of thinking, initially inherited from electronics and analog technology, to adopt those of computer scientists proposing supposedly miracle languages. However, these proposed programming languages appeared unsuitable to the needs of control engineers: the sequential paradigm did not fit the intrinsic parallelism of the addressed problems. Moreover, computer science generally considers a physical concurrency (concerning actual execution) which does not match the logical concurrency (concerning the high-level description of system behavior) needed by the designers.

Synchronous data-flow languages are a satisfactory answer to the needs of control system designers. The data-flow model is directly based on a description by means of block-diagrams. Moreover, the synchronous model perfectly fits the execution scheme of the target systems: an infinite loop successively performing input sampling, internal computations, and output writing.

The data-flow model is based on a block diagram-based description. A block diagram may be viewed as a network of operators (or as a system of equations) running in parallel at the rate of their inputs. An example of such a network is drawn below, which corresponds to the following equation (textual view):

$$s = 2 * (x + y)$$



The synchronous data-flow approach consists in the addition of a temporal behavior to the data-flow model. A natural way is to relate the discrete time with the rate of values traveling along the "wires" of the network. For instance, the description given above expresses the following temporal relation: At any cycle n, $s_n = 2 * (x_n + y_n)$. That way, the temporal dimension naturally underlies any data-flow description. Moreover, synchronous data-flow languages allow some operators to be activated at different rates, through the notion of *clock*. Each "wire" can be provided with a clock, which defines the logical instants where it carries values. This way, and in accordance with the data-flow principle, each operator is activated at the rate of its inputs. Our notion of "synchronous data-flow" must be distinguished from the one used in token-based models like Ptolemy [17], where nothing is said about the conceptual simultaneity of tokens traveling on different wires. A more mathematical presentation of the seman-

tics of data-flow synchronous languages can be found in [9].

B. *The Data-Flow Language* LUSTRE

The LUSTRE language [13]

The LUSTRE language, which is briefly described below, has enjoyed some industrial success because of its simplicity and its adequacy for the design of control systems. The SAO+/SAGA environment, which has been developed around LUSTRE by the company Verilog, provides solutions for the other needs expressed by users.

LUSTRE [13, 11] is a synchronous data-flow language developed at VERIMAG laboratory. A LUSTRE program defines its output variables as functions of its input variables. Each variable or expression E denotes a function of discrete time, giving its value E_n at each "instant" n. Variables are defined by means of equations: an equation `X=E`, specifies that the variable `X` is always equal to expression `E`.

Expressions are made of variable identifiers, constants (considered as constant functions), the usual arithmetic, boolean and conditional operators (applied pointwise to functions) and only two specific operators: the "previous" operator (`pre`) which refers to the previous value of its argument, and the "followed-by" operator (`->`) which is used to define initial values: if `E` and `F` are LUSTRE expressions, so are `pre(E)` and `E -> F`, and we have at any instant $n > 0$:

- $(\texttt{pre(E)})_n = E_{n-1}$, while $(\texttt{pre(E)})_0$ has the undefined value *nil*.
- $(\texttt{E -> F})_n = F_n$, while $(\texttt{E -> F})_0 = E_0$.

For instance, let `E = 0->(pre(x) + y)`, and x_n, y_n be the respective values of `x` and `y` at "instant" n. This means that the initial value E_0 of `E` is 0, and at any non-initial instant n, $E_n = x_{n-1} + y_n$.

```
node Counter (reset:  bool; init, incr:  int)
   returns (count:  int);
let
   count =init -> if reset then init
                  else pre(count)+incr;
tel
```

Fig. 2: Example of LUSTRE program: A counter

A LUSTRE program is structured into *nodes*: a node is a subprogram defining its output parameters as functions of its input parameters. This definition is given an unordered set of equations, possibly involving local variables. Once declared, a node may be freely instantiated in any expression, just as a basic operator.

As an illustration, Figure 2 shows an extremely simple node describing a counter: it receives a boolean input, `reset`, and two integer inputs, `init` and `incr`. It returns an integer output, `count`, which behaves as follows: at the initial instant and whenever the input `reset` is true, the output is equal to the current value of the input `init`. At any other instant, the value of `count` is equal to its previous value incremented by the current value of `incr`.

Figure 3 illustrates the behavior of the program: it shows the sequence of values of the expressions of the program in response to a particular sequence of the input parameters. Vertical reading of this table gives the value of each expression at each execution cycle.

So, through the notion of node, LUSTRE naturally offers hierarchical description and component reuse. Data traveling along the "wires" of an operator network can be complex, structured information.

From a temporal point of view, industrial applications show that several processing chains, evolving at different rates, can appear in a single system. LUSTRE offers a notion of boolean clock, allowing the activation of nodes at different rates (i.e., only when their clock is true).

Several compiling techniques have been studied. A straightforward approach to code generation from a LUSTRE program is a single loop that waits for input values, computes all the variables in a suitable order, writes the output values and updates the memory (to implement "`pre`" operators). More sophisticated techniques [13] have been developed to generate more efficient control structures: they consist in an exhaustive compile-time evaluation of the behavior of some selected boolean variables. The result is a more or less detailed interpreted automaton.

Apart from code optimization, such an automaton can also be used for verification. The general approach [14] that has been proposed for verifying critical properties about LUSTRE programs can be used with all synchronous languages. Experience shows that most critical properties required about a reactive program are "safety properties" (i.e., expressing that "something bad never happens"). Any such property can be expressed by means of an auxiliary program, called an *observer*, which receives as inputs the input/output variables of the program to be verified, and emits a boolean output, say `alarm`, as soon as the observed behavior does not satisfy the property. Now, one only has to prove that the behavior of the program composed of the system program and its observer never emits the `alarm` signal (see Fig. 4). For logical properties, this can be shown by an exhaustive simulation of the control automaton of the composed program. Extremely efficient implicit techniques are available for exploring all states of an automaton, making this sort of verification feasible for

very large programs.

Research work about LUSTRE at Verimag Laboratory also concern distributed code generation [10]: from a single program and distribution directives given by the user that assign processors to variables, the distribution tool is able to generate the code for each processor, so that, thanks to a very simple communication protocol, the cooperation of these distributed processes has the same functional behavior as the centralized program. This distribution tool can be used with other synchronous languages.

In conclusion, LUSTRE meets many of the needs of critical reactive system designers.

- It is very close to the formalisms they are familiar with.
- It allows hierarchical description and component reuse.
- It is deterministic.
- It has unique, clear and precise semantics.
- It offers a natural a graphical representation, which is the best way for communication with humans.

LUSTRE tools

Based on LUSTRE, Verilog's SAO+/SAGA environment offers a graphical editor for block-diagrams and a C code generator. From the data-flow nature of the language, programs can be expressed both in textual and in graphic form. The available commercial editor allows both syntaxes to be freely mixed.

The C code generator produces ANSI-C, which can be optimized. Available optimizations concern the code size and the performance in memory and time. The generated code is also readable: its structure follows the hierarchy of the LUSTRE description, and the generated variables are named in order to ensure traceability with respect to the data involved in the LUSTRE description.

The SAO+/SAGA code generator is also interfaced with the public-domain verification tool LESAR [14], developed at Verimag laboratory.

C. *The Data-Flow Relational Language* SIGNAL

Particular Features of the SIGNAL Language Model

The SIGNAL language [16] extends slightly the standard principles of synchronous dataflow as described in the introduction of subsection A.. In this subsection, synchronous dataflow programs are modeled as infinite activation loops, successively performing input reading, internal computations, and output writing. Call such synchronous programs Single Clock Activated (SCA) programs. Basically, SIGNAL extends this model to that of Multiple Clock Activated (MCA) programs. MCA programs are just networks of interconnected SCA programs. Each SCA program is activated by its own infinite loop, which we call an *activation clock*. Each activation step of a clock is called an *instant* of this clock. Activation clocks can run independently, or they can be given constraints relating them. Different clocks can share instants, during which their respective SCAs can communicate. Thus different SCAs can be activated *simultaneously*, and this is why SIGNAL is a synchronous language. In addition to relaxing the strict synchrony hypothesis, this facility allows SIGNAL to provide *mixed data/demand driven* activation modes for MCA programs.

To specify SIGNAL MCA programs, relations on their various clocks can be stated. For instance, assume we want to model two tasks S, T running asynchronously under clocks `H,K` respectively, and communicating at the instants of some subclock `L` of both `H` and `K` . The corresponding synchronization specification is simply expressed by the following SIGNAL statement

```
L ^= H when K
```

In this statement, `^=` specifies equality of clocks of both sides, and the expression `H when K` denotes the clock composed of the common instants of `H` and `K`.

This model of "relaxed synchrony" is implemented in SIGNAL in the following way. Objects handled by SIGNAL are *signals*, a signal is defined as a sequence of values available at the instants of some *clock*. SIGNAL allows the definition of relations (i.e., constraints) involving signals and clocks. Relations on signals are *equations* involving current and past values of signals. Equations can also be used to specify relations on clocks. Equations involving clocks and booleans are sufficient to specify automata. Other equations are used to specify computations or, more generally, to handle other data types. Through the notion of "*process*", SIGNAL offers hierarchical description and component reuse.

SIGNAL programming of synchronization and control entails the specification of synchronization constraints, each involving a small number of different signals. Synthesis of a global and coherent synchronization of the whole program, which matches each specified synchronization constraint, is deferred to the SIGNAL compiler.

A Sample of the Language

To illustrate SIGNAL and its style of programming, we present some examples. Although graphical programming is also available, for toy examples with simple architectures, textual programming is preferred.

A second order digital filter corresponds to the following formula:

$$y_n = a_1 y_{n-1} + a_2 y_{n-2} + b_0 u_n + b_1 u_{n-1} \quad (1)$$

Its SIGNAL program is simply

```
(| y := a1*y$1 + a2*y$2 + b0*u + b1*u$1 |)
```

In this program, `x := y$k` is the k-delay or k-shift register, defined by $\forall n > 0 : x_n = y_{n-k}$, thus " `$1` " is equivalent to the "z^{-1}" shift operator used by control or signal processing engineers.

A push-button alternatively starts and stops some device. The status of the device is encoded with a boolean signal `ON_OFF`, and is assumed to be "off" at the initialization. The SIGNAL program is

```
(| ON_OFF := not (ON_OFF$1 init false)
 | ON_OFF ^= BUTTON |)
```

where " | " is the composition operator. The first equation, considered as a single SIGNAL program, specifies the infinite sequence of alternating *true* and *false*: *f t f t f...* This equation has no input. In particular, its clock is not bound to the environment, which is a kind of nondeterminism. The second equation specifies that `ON_OFF` and `BUTTON` have the same clock (equality of clocks is written " ^= "). It is used to bound the clock of the first equation to that of the external pure signal `BUTTON`. Combining both equations yields the desired result, which is deterministic. Generally, combining possibly nondeterministic specifications to get a desired executable program is typical of SIGNAL programming style.

Combining automata (this example) and computations (the previous example) is easily performed using this unified framework. We illustrate this next.

An FDMA/TDMA transmultiplexer: upsampling in SIGNAL. Consider a communication protocol, in which frequency division multiple access (FDMA) is translated into time division multiple access (TDMA). We shall also assume that the number of simultaneous users is variable over time. Part of the specification involves receiving a packet containing a variable number `FB` of frequency bins, and reemitting the `FB` bins in the form of a sequence of successive bins. This mechanism of variable rate upsampling is expressed in SIGNAL as follows (we illustrate here the notion of "process"):

```
process MUX =                          % declaring a process
   ( ? integer FB;                     % ? : declaring inputs
     ! integer N; )                    % ! : declaring outputs
   (| N  := FB default (ZN - 1)        % body
    | ZN := N$1 init 1
    | FB ^= when (ZN <= 1) |)
   end
```

The behavior of this program is depicted below:

```
FB:  3        2     5  ...
ZN:  1  3  2  1  2  1  ...
 N:  3  2  1  2  1  5  ...
```

The third equation expresses that input `FB` is read `when` boolean signal `(ZN <= 1)` is true; `ZN` is the delayed value of `N` ; finally, the first equation expresses that `N` equals `FB` with priority, or `(ZN - 1)` by `default` if `FB` is absent. Note that the clock of output `N` is more frequent than that of input `FB`. It is a particular and powerful feature of the SIGNAL formalism that programs with upsampling can be specified.

Principles of SIGNAL Compilation

To reason about synchronization and logic, each signal is summarized by the following status: *absent, true, false* for boolean signals, and *absent, present* for other signals, where the label "present" is substituted for any value. Performing the substitution {any non boolean value} $\leftarrow$ {present} transforms any SIGNAL program into a program involving only clocks and booleans, i.e., an automaton. For example, applying this to the `MUX` program yields the program shown on bottom, also called the *clock calculus*:

```
% original program
(| N  := FB default (ZN-1)
 | ZN := N$1 init 1
 | FB ^= when (ZN <= 1) |)

% associated clock calculus
(| N  ^= FB default (ZN-1)
 | ZN ^= N
 | FB ^= when (ZN <= 1) |)
```

The third equation is unchanged since it involves only clocks. The first equation involves integers, thus only their status present/absent is kept, and this equation is just encoded into the equality of clocks of " `N` " and " `FB default (ZN-1)` ". The second equation involves integers and a delay, thus only presence/absence is kept (equality of clocks) and information regarding memory is lost.

The clock calculus of a program encodes the state of its control — in the `MUX` example, there are two states, according to whether `(ZN <= 1)` is true or not. Then, to each state, computations can be attached. The resulting form of the program is structured as a tree of nested modules, it cleanly separates control from computations, and is suitable to code generation. In this way the compiler delivers a state-dependent scheduling for the computations.

Since clock calculi are just finite state machines, they are amenable to model checking or even synthesis techniques: consistency can be verified and determinism can be checked [16].

SIGNAL tools

Based on the above principles, C, ADA, VHDL code is generated[1]. Generated C code can be used for animation. Debugging tools allow the user to check the consistency of the specified synchronizations.

Properties of the control skeleton of the program can be handled in two different ways. First, a safety property can be written in SIGNAL and included as part of the program specification: this property will be enforced as a constraint at execution time. Alternatively, properties can be checked a posteriori on a given program, a model checker is provided for this purpose in the SILDEX tool.

IV. INDUSTRIAL TOOLS AND USERS EXPERIENCES

Figures 5, 6, 7, summarize the commercially available tools and corresponding users reports.

V. THE COMMON FORMATS OF SYNCHRONOUS LANGUAGES

The groups working on synchronous languages decided to join their efforts by designing a suite of common formats. These formats are to be used as internal representations of programs written in synchronous languages in order to allow tools to be shared (e.g., code generators, distributed code generators, verification tools, silicon compilers, simulators) and to make easier the integration of programs written in different languages.

Two formats are currently under design. The *declarative code* DC is a high-level format dedicated to both the representation of declarative or data-flow synchronous programs, and to the equational representation of imperative programs. It is a parallel, structured format, where programs are considered as operator networks. It is the privileged source format for silicon compiling, symbolic verification and optimization, and distributed code generation. The *extended declarative code* DC_+ is an extension of DC. It retains the full power of the data-flow synchronous model. It offers powerful modular features and clocking mechanisms. DC_+ is to be used instead of DC when separate compilation is wanted or for clock-based optimizations. In addition, it allows the use of different techniques for services that are also accessible via DC[2].

In addition, two formats are available for interested users. The *object code* OC is a low-level format. It is a sequential, unstructured format, representing programs as interpreted automata. It is intended to be the source format for tools specific to automata (simulation, animation, verification). Finally, the *imperative code* IC is a high level format dedicated to the representation of imperative synchronous programs; it is an internal format for ESTEREL tools.

[1] BLIF code is also generated from the Inria SIGNAL environment.

[2] Referring to Section C., DC and DC_+ implement the SCA and MCA models respectively.

REFERENCES

[1] F. BOUSSINOT, R. DE SIMONE, "The ESTEREL language", *Another look at real-time programming*, special section of *Proc. of the IEEE*, vol. 9 n° 9, September 1991, 1293–1304.

[2] C.A.R. Hoare. Communicating sequential processes. *Communication of the ACM*, 21(8):666–676, 1978.

[3] A. Benveniste and G. Berry, "The Synchronous Approach to Reactive and Real-Time Systems", *Proceedings of the IEEE*, vol. 79, N° 9, September 1991, 1270–1282.

[4] G. Berry. Real time programming: Special purpose or general purpose languages. In *IFIP World Computer Congress*, San Francisco, 1989.

[5] G. Berry. ESTEREL on hardware. *Philosophical Transactions Royal Society of London A*, 339:87–104, 1992.

[6] G. Berry. The semantics of pure esterel. In M. Broy, editor, *Program Design Calculi*, volume 118 of *Series F: Computer and System Sciences*, pages 361–409. NATO ASI Series, 1993.

[7] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[8] G. Boudol, V. Roy, R. de Simone, and D. Vergamini. Process calculi, from theory to practice: Verification tools. In *International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble*. LNCS 407, Springer Verlag, 1990.

[9] A. Benveniste, P. Caspi, P. Le Guernic, N. Halbwachs. Data-Flow Synchronous Languages, *Proc. of the 1993 REX School/Symposium, A Decade of Concurrency*, 1–46, LNCS vol. 803, Springer Verlag, 1994.

[10] P. Caspi, A. Girault, and D. Pilaud. Distributing reactive systems. In *Seventh International Conference on Parallel and Distributed Computing Systems, PDCS'94*. ISCA, October 1994.

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

[12] D. Harel. Statecharts: A visual approach to complex systems. *Science of Computer Programming*, 8(3), 1987.

[13] 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.

[14] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous dataflow programming language LUSTRE. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, September 1992.

[15] L. Jagadeesan and C. Puchol and J. von Olnhausen. A Formal Approach to Reactive Systems Software: a Telecommunications Application in ESTEREL. *Formal Methods in System Design*, 6, 1–29 (1995).

[16] P. LeGuernic, T. Gautier, M. LeBorgne, and C. LeMaire. Programming real time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, September 1991.

[17] Ed. A. Lee., T.M. Parks. Dataflow Process Networks. *Proceedings of the IEEE*, may 1995, to appear. (http://ptolemy.eecs.berkeley.edu/papers/processNets).

[18] O. Maffeïs, P. Le Guernic. Distributed Implementation of SIGNAL: Scheduling & Graph Clustering, 3rd International School and Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, Lecture Notes in Computer Science 863, Springer-Verlag, 547–566, 1994.

[19] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR'92*, Stony Brook, August 1992. LNCS 630, Springer Verlag.

[20] S. Ramesh, G. Berry, R. K. Shyamasundar. Communicating reactive processes. In *Proc. 20th ACM Conf. on Principles of Programming Languages, Charleston, Virginia*, 1993.

[21] H. Touati and G. Berry. Optimized controller synthesis using ESTEREL. In *Proc. International Workshop on Logic Synthesis IWLS'93, Lake Tahoe*, 1993.

[22] T. Shiple and G. Berry and H. Touati. Constructive Analysis of Cyclic Circuits In *Proc European Design and Test Conf.*, Paris, 1996.

cycle nr.	0	1	2	3	4	5	6	7
reset	*f*	*f*	*f*	*f*	*t*	*f*	*f*	*f*
init	0	0	0	0	10	0	0	0
incr	1	1	1	1	1	1	2	2
count	0	1	2	3	10	11	13	15
pre(count)	*nil*	0	1	2	3	10	11	13

Fig. 3: Behavior of the Counter program

A APPENDIX : LIST OF CONTRIBUTORS

Authors are grouped according to their affiliation, ordering has no particular meaning.

- Christine Bodennec, Claude Lemaire, Patrick Munier, VERIMAG/ Verilog, Miniparc - ZIRST, 38330 Montbonnot St Martin, France, email: name@verilog.fr, fax: +33 76 41 36 15.
- François Dupont, TNI, Technopôle Brest-Iroise, CP1, 29608 Brest Cedex, France, fax: +33 98 49 45 33.
- Gérard Berry, Robert De Simone, Jean-Marc Tanzi, CMA, Ecole des Mines, BP 207, 06904 Sophia Antipolis Cedex, France, emails berry@cma.cma.fr, rs@cma.cma.fr, jmt@cma.cma.fr.
- Paul Caspi, Nicolas Halbwachs, VERIMAG/CNRS, Miniparc - ZIRST, 38330 Montbonnot St Martin, France, email: Surname.Name@imag.fr.
- Albert Benveniste, Thierry Gautier, Paul Le Guernic, INRIA-IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France, email Surname.Name@inria.fr
- Yves Auffray, Thierry Bravier, Emmanuel Ledinot, Dassault-Aviation, 78 Quai Marcel Dassault, F92215 Saint Cloud Cedex, email: `ledinot@dassault-avion.fr`.
- Jean-Louis Bergerand, Christian Dubois, Schneider Electric, F38050 Grenoble Cedex.
- Philippe Baufreton, Alain Janvier, SNECMA, Site de Villaroche, F77550 Moissy Cramayel.

ACKNOWLEDGEMENT : *Thanks are due to Ellen Sentovich for improvements on an earlier version of this manuscript.*

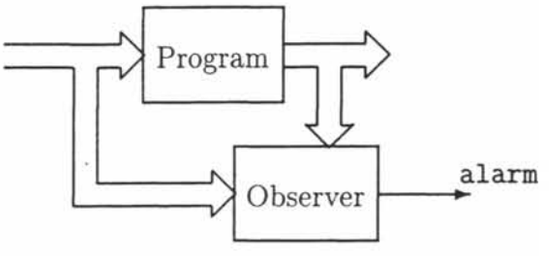


Fig. 4: Verification program

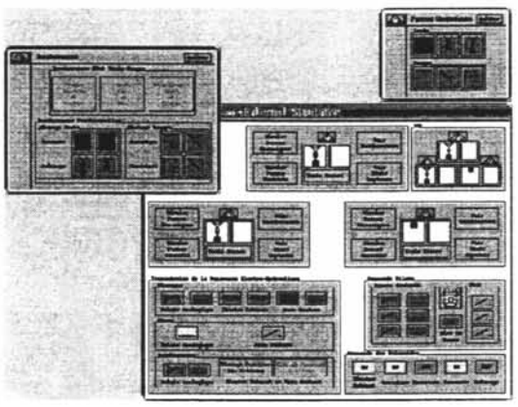


Fig. 5: *Using the* ESTEREL *environment.* "We at Dassault-Aviation have used ESTEREL for both rapid system prototyping and embedded software component generation. In particular, we have designed a high performance task sequencer for the failure-monitoring module in a military aircraft control system. By using ESTEREL, we were able to replace our previous static scheduler by a dynamic one. In addition, safety and correctness were guaranteed, thanks to the very fine capabilities provided by ESTEREL to describe preemption and various interrupts as well as task concurrency. Final C code was automatically generated, and was nearly as good as hand-optimized code. Verifications via formal proof checking were performed, and all critical properties have been *proved.* For such applications, safe design means saved money. The screen snapshot given above is the graphical simulation panel used to run the ESTEREL automaton controlling the traps and landing gear of the Rafale aircraft. The completed program involves about 300 input/output signals and 80 ESTEREL modules. 20 safety properties were checked using symbolic techniques. The finite state machine with counters produced by the ESTEREL compiler had more than 10^{14} reachable states." *Yves Auffray, Dassault-Aviation, St Cloud, France.*

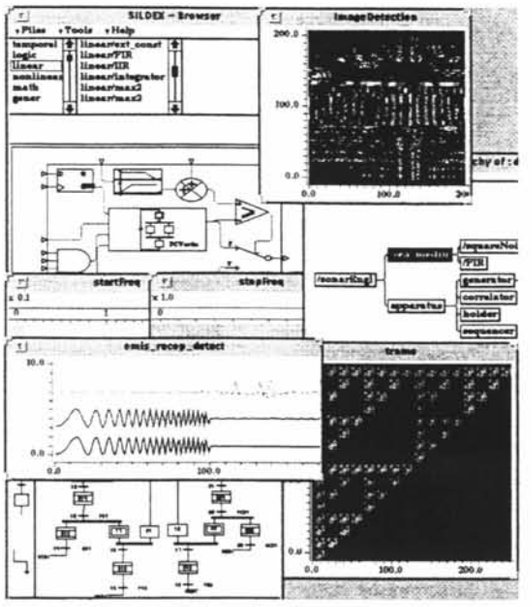


Fig. 6: *The* SILDEX *environment for the language* SIGNAL. The SILDEX real-time application development tool provides a framework for specifying, coding, testing, and verifying an application based on a single graphical view. Programs are edited as block-diagrams. The tool manages process libraries; realizing a project involves enriching one or several librairies with new processes. For applications involving complex sequencing, a Sequential Function Chart editor is provided in combination with the SIGNAL editor. For safety-critical applications, a formal prover is included, which returns counter-examples in case of violation of a safety property. Code generators deliver efficient C and ADA code, and documentation generators target industry standard formats. A simulator allows immediate interactive execution of the generated C code. SILDEX is commercially available from TNI, Brest, France. *Alain Janvier, SNECMA Villaroche, said:* "At Snecma, we specify, design, manufacture, and support Full Authorized Digital Engine Control systems for both aircraft and helicopter engines. The first produced computer for the M88-2 engine (a new generation engine for the Rafale fighter) features the latest technological advances. It involves digital engine control based on a new generation of digital architecture and software design methods. Corresponding computer controlled systems are maximally critical, so maximally safe design is requested. Today we use the SILDEX environment for the SIGNAL language in order to describe and validate such systems. In particular, the correctness of the system against certain critical properties can be proved, which is a dramatic improvement with respect to conventional available techniques, whether visual or textual, in which simulation is the only basis for testing. Then, we also use SILDEX and SIGNAL for automatic code generation. The increased confidence in the system and software specifications, as well as in the code generation method, is expected to enable a drastic reduction in the effort spent in software and system testing on the final product. "

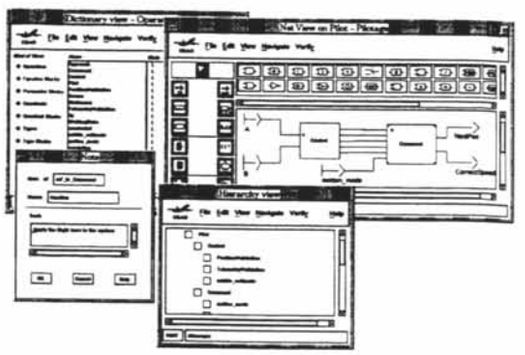


Fig. 7: *The* SAO+/SAGA *environment for the langage* LUSTRE. SAO+/SAGA is an open environment developed by Verilog and based on the experience acquired by Schneider Electric and Aerospatiale in designing critical real-time systems. The SAO+/SAGA environment consists of two main tools: a multiview graphical editor and an automatic code generator. The editor handles LUSTRE descriptions and proposes a hierarchical view of the description: a network view to edit block diagrams graphically and a text view to edit textual equations in the LUSTRE language. The automatic code generator produces 100% of the C code that implements a LUSTRE description. The code generated is ANSI-compatible and can be optimized on request, in terms of memory size and processing speed. The C code is readable and traceable, and can be used for simulation. This code generator can also be connected to a formal property verifier developed by the Verimag research laboratory. "At Schneider-Electric we have developed the SAGA block-diagram environment based on the LUSTRE language for the design of safety critical instrumentation and control software. In particular, we have used SAGA for the microprocessor-based protection system for the core of the new French 1,450MW nuclear power plants. 200,000 lines of C code were produced, 87% of them automatically. The synchronous approach, supported by SAGA, has allowed software to be efficiently designed while meeting safety goals. We were able to meet the functional and response time requirements using a simple deterministic software architecture, without the need for interrupts or multitasking support. Formal verification provided in conjunction with LUSTRE was a significant step toward "proven" software. The case study design has 80 inputs, 100 outputs, and 4,000 lines of C code automatically generated by SAGA. Nearly 60 safety properties involving the setting of emergency shutdown or protection actions have been formally proved. Today, Schneider-Electric, in cooperation with Aerospatiale (from the Airbus Industry consortium) are partners of Verilog in the marketing of the SAGA environment." *Christian Dubois, Schneider-Electric, Grenoble.*