The Synchronous Languages Twelve Years Later

Albert Benveniste, *Fellow, IEEE*, Paul Caspi, Stephen A. Edwards, *Member, IEEE*, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone

Abstract— **Twelve years ago, Proceedings of the IEEE devoted a special section to the synchronous languages. This article discusses the improvements, difficulties, and successes that have occured with the synchronous languages since then. Today, synchronous languages have been established as a technology of choice for modeling, specifying, validating, and implementing real-time embedded applications. The paradigm of synchrony has emerged as an engineer-friendly design method based on mathematically-sound tools.**

Keywords: **Embedded systems, Esterel, formal methods, Lustre, real-time systems, Signal, synchronous languages.**

In 1991, Proceedings of the IEEE devoted a special section to the synchronous languages [1], [2]. It included papers describing the three French synchronous languages Esterel [3], Lustre [4], and Signal [5], which are also the subject of this paper. At the time, the three languages were well-defined and had seen some industrial use, but were still under development. In the intervening years, the languages have been improved, gained a much larger user community, and have been successfully commercialized. Today, synchronous languages have been established as a technology of choice for modeling, specifying, validating, and implementing real-time embedded applications. The paradigm of synchrony has emerged as an engineer-friendly design method based on mathematically sound tools.

This paper discusses the improvements, difficulties, and successes that have occured with the synchronous languages since 1991. It begins with a discussion of the synchronous philosophy and the challenge of maintaining functional, deterministic system behavior when combining the synchronous notion of instantaneous communication with deterministic concurrency. Section II describes successful uses of the languages in industry and how they have been commercialized. Section III discusses new technology that has been developed for compiling these languages, which has been substantially more difficult than first thought. Section IV describes some of the major lessons learned over the last twelve years. Section V discusses some future challenges, including the limitations of synchrony. Finally, section VI concludes the paper with some discussion of where the synchronous languages will be in the future.

Throughout this paper, we take the area of embedded control systems as the central target area of discussion, since this has been the area in which synchronous languages have best found their way today. These systems are typically safety-critical, such as flight control systems in flight-by-wire avionics and antiskidding or anticollision equipment on automobiles.

Albert Benveniste and Paul Le Guernic are with Irisa/Inria, Campus de Beaulieu, 35042 Rennes cedex, France; email: Albert.Benveniste@inria.fr, http://www.irisa.fr/sigma2/benveniste/

Paul Caspi and Nicolas Halbwachs are with Verimag/CNRS, Centre Equation, 2, rue de Vignate, F-38610 Gières, France; email: Paul.Caspi@imag.fr, http://www-verimag.imag.fr/∼caspi/

Stephen A. Edwards is with Columbia University, New York, NY 10027 USA; email: sedwards@cs.columbia.edu, http://www.cs.columbia.edu/∼sedwards/

Robert de Simone is with INRIA 2004 route des Lucioles, BP 93, 06902 Sophia Antipolis, France; email: Robert.De-Simone@inria.fr, http://www.inria.fr/personnel/Robert.de_Simone.fr.html

I. The Synchronous Approach

The three synchronous languages Signal, Esterel, and Lustre are built on a common mathematical framework that combines synchrony (i.e., time advances in lockstep with one or more clocks) with deterministic concurrency. This section explores the reasons for choosing such an approach and its ramifications.

A. Fundamentals of Synchrony

The primary goal of a designer of safety-critical embedded systems is convincing him- or herself, the customer, and certification authorities that the design and its implementation is correct. At the same time, he or she must keep development and maintenance costs under control and meet nonfunctional constraints on the design of the system, such as cost, power, weight, or the system architecture by itself (e.g., a physically distributed system comprising intelligent sensors and actuators, supervised by a central computer). Meeting these objectives demands design methods and tools that integrate seamlessly with existing design flows and are built on solid mathematical foundations.

The need for integration is obvious: confidence in a design is paramount, and anything outside the designers' experience will almost certainly reduce that confidence.

The key advantage of using a solid mathematical foundation is the ability to reason formally about the operation of the system. This facilitates certification because it reduces ambiguity and makes it possible to construct proofs about the operation of the system. This also improves the implementation process because it enables the program manipulations needed to automatically construct different implementations, useful, for example, for meeting nonfunctional constraints.

In the 1980s, these observations lead to the following decisions for the synchronous languages:

1. Concurrency—The languages must support functional concurrency, and they must rely on notations that express concurrency in a user-friendly manner. Therefore, depending on the targeted application area, the languages should offer as a notation block diagrams (also called dataflow diagrams), or hierachical automata, or some imperative type of syntax, familiar to the targeted engineering communities. Later, in the early nineties, the need appeared for mixing these different styles of notations. This obviously required that they all have the same mathematical semantics.
2. Simplicity—The languages must have the simplest formal model possible to make formal reasoning tractable. In particular, the semantics for the parallel composition of two processes must be the cleanest possible.
3. Synchrony—The languages must support the simple and frequently-used implementation models in Fig. 1, where all mentioned actions are assumed to take finite memory and time.

```
Initialize Memory
for each input event do
    Compute Outputs
    Update Memory
end
```

```
Initialize Memory
for each clock tick do
    Read Inputs
    Compute Outputs
    Update Memory
end
```

Fig. 1. Two common synchronous execution schemes: event driven (left) and sample driven (right).

B. Synchrony and Concurrency

Combining synchrony and concurrency while maintaining a simple mathematical model is not so straightforward. Here, we discuss the approach taken by the synchronous languages.

Synchrony divides time into discrete instants. This model is pervasive in mathematics and engineering. It appears in automata, in the discrete-time dynamical systems familiar to control engineers, and in synchronous digital logic familiar to hardware designers. Hence it was natural to decide that a synchronous program would progress according to successive *atomic reactions.* We write this for convenience using the "pseudomathematical" statement $P \equiv R^{\omega}$, where R denotes the set of all possible reactions and the superscript ω indicates nonterminating iterations.

In the block diagrams of control engineering, the nth reaction of the whole system is the combination of the individual nth reactions for each constitutive block. For block i,

$$\begin{aligned} X_n^i &= f(X_{n-1}^i, U_n^i) \\ Y_n^i &= g(X_{n-1}^i, U_n^i) \end{aligned} \tag{1}$$

where U, X, Y are the (vector) input, state, and output, and combination means that some input or output of block i is connected to some input of block j, say

$$Y_n^j(k) = U_n^i(l) \text{ or } Y_n^i(l), \tag{2}$$

where $Y_n^j(k)$ denotes the k-th coordinate of vector output of block j at instant n. Hence the whole reaction is simply the conjunction of the reactions (1) for each block, and the connections (2) between blocks.

Connecting two finite-state machines (FSM) hardware is similar. Fig. 2(a) shows how a finite-state system is typically implemented in synchronous digital logic: a block of acyclic (and hence functional) logic computes outputs and the next state as a function of inputs and the current state. Fig. 2(b) shows the most natural way to run two such FSMs concurrently and have them communicate, i.e., by connecting some of the outputs of one FSM to the inputs of the other and vice versa.

Therefore, the following natural definition for parallel composition in synchronous languages was chosen, namely: $P_1 \parallel P_2 \equiv (R_1 \wedge R_2)^{\omega}$, where $\wedge$ denotes conjunction. Note that this definition for parallel composition also fits several variants of the synchronous product of automata. Hence the model of synchrony can be summarized by the following two pseudoequations:

$$P \equiv R^{\omega}, \tag{3}$$

$$P_1 \parallel P_2 \equiv (R_1 \wedge R_2)^{\omega}. \tag{4}$$

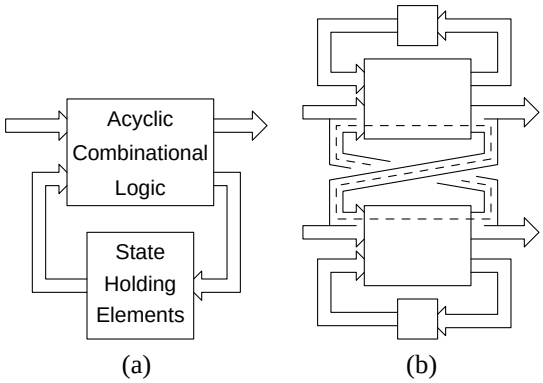


Fig. 2. (a) The usual structure of a finite-state machine implemented in hardware. (b) Connecting two FSMs. The dashed line shows a path with instantaneous feedback that arises from connecting these two otherwise functional FSMs.

However, there is no free lunch. Definition (4) for parallel composition requires forming the conjunction of the reactions for each component. It is well known that such a conjunction will not in general be a function but rather a *relation* or, equivalently, a *constraint*.

The product of automata is a familiar example: two automata must synchronize when performing a shared transition; this is a constraint. Similarly, it is well known that allowing arbitrary connections among blocks in a block diagram yields *implicit* dynamical systems in which a reaction relates "inputs, states, and outputs" (we put quotes to indicate that the distinction between input and output becomes subtle). Such implicit models (also called "descriptor" or "behavioral" in the control community) are in fact extremely convenient for modeling systems from first principles, where balance equations are naturally encountered. The same occurs in hardware: composing two systems as in Fig. 2(b) does not always produce a system of the form of Fig. 2(a) because it is possible to create a delay-free (i.e., cyclic) path such as the one drawn with a dashed line.

This problem of functional systems not being closed under synchronous, concurrent composition can be addressed in at least four different ways.

1. Microsteps—One can insist that a reaction remains operational by definining it to be a sequence of elementary *microsteps*. In this approach, primitive system components are assumed to be such a sequence of elementary microsteps, and the parallel composition of primitive components is performed by interleaving their evaluation in some appropriate way. This is a move to an operational type of semantics that violates the beautiful and simple mathematical operation of conjunction. Nevertheless, this is the approach taken in the Very High Speed Integrated Circuit Hardware Description Language (VHDL) [6] and Verilog [7] modeling languages, in Harel's Statecharts [8], and in the older formalism of Grafcet [9], [10] used to program PLCs in control systems. It also conforms to the computer science and engineering view of program execution as a sequence of guarded actions. Nevertheless, microstep semantics are confusing and prone to many conflicting interpretations [11], [12].
2. Acyclic—Usually, control engineers simply insist that their block diagrams contain no zero-delay loops. In this case, the

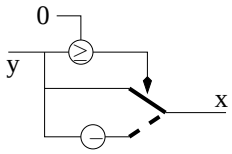


Fig. 3. A simple data-flow network

system is guaranteed to behave functionally. This approach turns out to be well-suited to the sample-driven approach of Fig. 1. The Lustre language adopts this view.

3. Unique fixpoint—This approach accepts that each reaction is the solution of a fixpoint equation, but insists that the system always behave functionally, i.e., that each reaction is a deterministic function of the form

$$\{\text{state, input}\} \quad \longmapsto \quad \{\text{next state, output}\}. \tag{5}$$

Compiling a program with these semantics becomes difficult because it is necessary to prove the relations implied by the program always have a unique solution of the form (5). Despite these difficulties, the Esterel language has adopted this approach.

4. Relation or constraint—This approach accepts reactions as constraints and allows each reaction to have zero solutions ("the program is blocked and has no reaction"), one solution of the form (5), or multiple consistent solutions that are interpreted as nondeterministic behavior. While implementations usually demand a unique solution, the other cases may be of interest for partial designs or high-level specifications. In this approach all programs have a semantics. But the issue of checking whether a program is a deterministic function of the form (5) still remains a prior to generating executable code. Signal has adopted this approach.

C. How The Fundamentals of Synchrony Were Instantiated in the Synchronous Languages

In the following sections, we give further details about how the above discussed fundamentals of synchrony have been instantiated in the synchronous languages Lustre, Esterel, and Signal.

C.1 Lustre

The goal of the designers of Lustre (pronnouced LOOSE-truh in the French) [13], [4] was to propose a programming language based on the very simple data-flow model used by most control engineers: their usual formalisms are either systems of equations (differential, finite-difference, Boolean equations) or data-flow networks (analog diagrams, block-diagrams, gates and flip-flops). In such formalisms, each variable that is not an input, is defined exactly once in terms of other variables. One writes "$x = y + z$", meaning that *at each instant k* (or, *at each step k*) $x_k = y_k + z_k$. In other words, each variable is a function of time, which is supposed to be discrete in a digital implementation: in basic Lustre, any variable or expression denotes a *flow*, i.e., an infinite sequence of values of its type. A basic Lustre program is effectively an infinite loop, and each variable or expression takes the *k*-th value of its sequence at the *k*-th step in the loop. All the usual operators—Boolean, arithmetic, comparison, conditional—are implicitly extended to operate pointwise on flows. For example, one writes

```
x = if y >= 0 then y else -y
```

to express that *x* is equal to the absolute value of *y* in each step. In the Lustre-based commercial Scade tool, this equation has a graphical counterpart like the block diagram of Fig. 3. Notice that constants (like `0`, `true`) represent constant flows. Any combinational function on flows can be described in this way. Two additional temporal operators make it possible to describe sequential functions:

1. For any flow `x`, `pre(x)` is the flow whose value at each step is the *previous value* of `x`. At the very first step, `pre(x)` takes the undefined value *nil* that corresponds to an uninitialized variable in traditional languages.
2. The "`->`" operator defines initial values: if `x` and `y` are flows of the same type, `x ->y` is the flow that is equal to `x` in the first step and equal to `y` thereafter.

More formally:

$$\texttt{pre(x)}_n = \begin{cases} \textit{nil} & \text{for } n = 0 \\ x_{n-1} & \text{for } n > 0 \end{cases} \qquad (\texttt{x ->y})_n = \begin{cases} x_0 & \text{for } n = 0 \\ y_n & \text{for } n > 0 \end{cases}$$

These operators provide the essential descriptive power of the language. For instance,

```
edge = false -> (c and not pre(c));
nat = 0 -> pre(nat) + 1;
edgecount = 0 -> if edge then pre(edgecount) + 1
                 else pre(edgecount);
```

defines `edge` to be true whenever the Boolean flow `c` has a rising edge, `nat` to be the step counter ($\texttt{nat}_n = n$), and `edgecount` to count the number of rising edges in `c`.

Lustre definitions can be recursive (e.g., `nat` depends on `pre(nat)`), but the language requires that a variable can only depend a past values of itself.

Lustre provides the notion of a *node* to help structure program. A *node* is a function of flows: a node takes a number of typed input flows and defines a number of output flows by means of a system of equations that can also use local flows. Each output or local flow must be defined by exactly one equation. The order of equations is irrelevant. For instance, a resettable event counter could be written

```
node COUNT (event, reset: bool)
returns (count: int);
let
  count = if (true->reset) then 0
                  else if event then pre(count)+1
                               else pre(count);
tel
```

Using this node, our previous definition of `edgecount` could be replaced by

```
edgecount = COUNT (edge,false);
```

This concludes the presentation of the base language. Several other features depend on the tools and compiler used:

1. Structured types—Real applications make use of structured data. This can be delegated to the host language (imported types and functions, written in C). Lustre V4 and the commercial Scade tool offer record and array types within the language.

2. Clocks and activation conditions—It is often useful to activate some parts of a program at different rates. Data-flow synchronous languages provide this facility using *clocks*. In Lustre, a program has a *basic clock*, which is the finest notion of time (i.e., the external activation cycle). Some flows can follow slower clocks, i.e., have values only at certain steps of the basic clock: if x is a flow, and c is a Boolean flow, x when c is the flow whose sequence of values is the one of x when c is true. This new flow is on the clock c, meaning that it does not have value when c is false.

The notion of clock has two components:

(a) A static aspect, similar to a type mechanism: each flow has a clock, and there are constraints about the way flows can be combined. Most operators are required to operate on flows *with the same clock*; e.g., adding two flows with different clocks does not make sense and is flagged by the compiler.

(b) A dynamic aspect: according to the data-flow philosophy, operators are only activated when their operands are present, i.e., when their (common) clock is true. Clocks are the only way to control the activation of different parts of the program.

While the whole clock mechanism was offered in the original language, it appeared that actual users mainly need its dynamic aspect and consider the clock consistency constraints tedious. This is why the Scade tool offers another mechanism called "*activation conditions*": any operator or node can be associated an activation condition that specifies when the operator is activated. The outputs of such a node, however, are always available; when the condition is false, their values are either frozen or take default values before the first activation.

By definition, a Lustre program may not contain syntactically cyclic definitions. The commercial Scade tool provides an option for an even stronger constraint that forbids an output of a node to be fed back as an input without an intervening "pre" operator. Users generally accept these constraints. The first constraint ensures that there is a static dependence order between flows and allows a very simple generation of sequential code (the scheduling is just topological sorting). The second, stronger constraint enables separate compilation: when enforced, the execution order of equations in a node cannot be affected by how it is called and therefore the compiler can schedule each node individually before scheduling a system at the node level. Otherwise, the compiler would have to topologically sort every equation in the program.

C.2 Esterel

While Lustre is declarative and focuses primarily on specifying data flow, the Esterel language is imperative and suited for describing control. Intuitively, an Esterel program consists of a collection of nested, concurrently-running threads described using a traditional imperative syntax (Table I is a sampling of the complete language, whose description has appeared elsewhere [3], [14], [15]) whose execution is synchronized to a single, global clock. At the beginning of each reaction, each thread resumes its execution from where it paused (e.g., at a pause statement) in the last reaction, executes traditional imperative code (e.g., assigning the value of expressions to variables and making control decisions), and finally either terminates or pauses in preparation for the next reaction.

TABLE I
Some basic Esterel statements.

emit S	Make signal S present immediately
present S then p else q end	If signal S is present, perform p otherwise q
pause	Stop this thread of control until the next reaction
p ; q	Run p then q
loop p end	Run p; restart when it terminates
await S	Pause until the next reaction in which S is present
p ‖ q	Start p and q together; terminate when both have terminated
abort p when S	Run p up to, but not including, a reaction in which S is present
suspend p when S	Run p except when S is present
sustain S	Means loop emit S; pause end
run M	Expands to code for module M

Threads communicate exclusively through signals: Esterel's main data type that represents a globally-broadcast event. A coherence rule guarantees an Esterel program behaves deterministically: all threads that check a signal in a particular reaction see the signal as either present (i.e., the event has occurred) or absent, but never both.

Because it is intended to represent an event, the presence of a signal does not persist across reactions. Precisely, a signal is present in a reaction if and only if it is emitted by the program or is made present by the environment. Thus, signals behave more like wires in a synchronous digital circuit than variables in an imperative program.

Preemption statements, which allow a clean, hierarchical description of state-machine-like behavior, are one of Esterel's novel features. Unlike a traditional if-then-else statement, which tests its predicate once before the statement is executed, an Esterel preemption statement (e.g., abort) tests its predicate each reaction in which its body runs. Various preemption statements provide a choice of whether the predicate is checked immediately, whether the body is allowed to run when the predicate is true, whether the predicate is checked before or after the body runs, and so forth.

To illustrate how Esterel can be used to describe control behavior, consider the program fragment in Fig. 4 describing the user interface of a portable compact disc player. It has input signals for play and stop and a lock signal that causes these signals to be ignored until an unlock signal is received, to prevent the player from accidentally starting while stuffed in a bag.

Note how the first process ignores the Play signal when it is already playing, and how the suspend statement is used to ignore Stop and Play signals.

This example uses Esterel's instantaneous execution and communication semantics to ensure that the code for the play operation, for example, starts exactly when the Play signal arrives. Say the code for Stop is running. In that reaction, the await Play

```
loop
   suspend
      await Play; emit Change
   when Locked;
   abort
      run CodeForPlay
   when Change
end
||
loop
   suspend
      await Stop; emit Change
   when Locked;
   abort
      run CodeForStop
   when Change
end
||
every Lock do
   abort
      sustain Locked
   when Unlock
end
```

Fig. 4. An Esterel program fragment describing the user interface of a portable CD player. Play and Stop inputs represent the usual pushbutton controls. The presence of the Lock input causes these commands to be ignored.

statement terminates and the Change signal is emitted. This prevents the code for Stop from running, and immediately starts the code for Play. Because the abort statement does not start checking its condition until the next reaction, the code for Play starts even though the Change signal is present.

As mentioned earlier, Esterel regards a reaction as a fixpoint equation, but only permits programs that behave functionally in every possible reaction. This is a subtle point: Esterel's semantics do allow programs such as cyclic arbiters where a static analysis would suggest a deadlock but dynamically the program can never reach a state where the cycle is active.

Checking whether a Esterel program is deadlock-free is termed *causality analysis*, and involves ensuring causality constraints are never contradictory in any reachable state. These constraints arise from control dependencies (e.g, the ";" sequencing operator requires its second instruction to start after the first has terminated, the present statment requires its predicate to be tested before either branch may run) and data dependencies (e.g., emit S must run before present S *even if they are running in parallel*). Thus, a statement like present S else emit S end has contradictory constraints and is considered illegal, but present S else present T then emit S end end may be legal if signal T is never present in a reaction in which this statement runs.

Formally, Esterel's semantics are based on *constructive causality* [15], which is discussed more in a later section on Esterel compilation techniques. The understanding of this problem and its clean mathematical treatment is one of Esterel's most significant contributions to the semantic foundations of reactive synchronous systems. Incomplete understanding of this problem is exactly what prevented the designers of the Verilog [7], [16], VHDL [6], and Statecharts [8], [12] languages from having truly synchronous semantics. Instead, they have semantics based on awkward delta-cycle microsteps or delayed relation notions with an unclear semantic relation to the natural models of synchronous circuits and Mealy machines. This produces a gap between simulation behavior and hardware implementations that Esterel avoids.

From Esterel's beginnings, optimization, analysis, and verification were considered central to the compilation process. Again, this was only possible because of formal semantics. The automata-based V3 compiler came with model checkers (Auto/Graph and later Fc2Tools) based on explicit state space reduction and bisimulation minimization. The netlist-based V4 and V5 compilers used the logic optimization techniques in Berkeley's SIS environment [17] to optimize its intermediate representation. In the V5 compiler, constructive causality analysis as well as symbolic binary decision diagram (BDD)-based model-checking (implemented in the Xeve verification tool [18]) rely on an implicit state space representation implemented with the efficient TiGeR BDD library.

C.3 Signal

Signal is designed for specifying systems [5], [19], [20]. In contrast with programs, systems must be considered *open*: a system may be upgraded by adding, deleting or changing components. How can we avoid modifying the rest of the specification while doing this?

It is tempting to say any synchronous program P does something at each reaction. In this case we say the *activation clock* of P coincides with its successive reactions. This is the viewpoint taken for the execution schemes of Fig. 1: in an event-driven system, at least one input event is required to produce a reaction; in a sample-driven system, reactions are triggered by the clock ticks.

However, if P is an open system, it may be embedded in some environment that gets activated when P does not. So we must distinguish the activation clock of P from the "ambient" sequence of reactions and take the viewpoint that each program possesses its own, local, activation clock. Parallel composition relates the activation clocks of the different components and the "ambient" sequence of reactions is implicit (not visible to the designer). A language that supports this approach is called a *multiclock* language. Signal is a multiclock language.

Signal allows systems to be specified as block diagrams. Blocks represent components or subsystems and can be connected together hierarchically to form larger blocks. Blocks involve *signals* and relate them via operators. In Signal each signal has an associated clock[1] and the activation clock of a program is the supremum of the clocks of its involved signals. Signals are typed sequences (boolean, integer, real, ...) whose domain is augmented with the extra value $\perp$ that denotes the absence of the signal in a particular reaction. The user is not allowed to manipulate the $\perp$ symbol to prevent him from manipulating the ambient sequence of reactions.

Signal has a small number of primitive constructs[2], listed in table II. In this table, X_τ denotes the status (absence, or actual carried value) of signal X in an arbitrary reaction τ.

[1]Lustre also has clocks, but they have been largely supplanted by activation conditions. Unlike clocks, activation conditions are operators, not types, and do not impose a consistency check.

[2]In addition to these primitive statements, the actual syntax also includes derived operators that can, for example, handle constraints on clocks.

TABLE II
Signal operators (left), and their meaning (right).

Z := X op Y	$Z_\tau \neq \bot \Leftrightarrow X_\tau \neq \bot \Leftrightarrow Y_\tau \neq \bot$ $\forall k : Z_k = \mathrm{op}(X_k, Y_k)$
Y := X$1	$X_\tau \neq \bot \Leftrightarrow Y_\tau \neq \bot$ $\forall k : Y_k = X_{k-1}$ (delay)
X := U when B	$X_\tau = U_\tau$ when $B_\tau = \mathit{true}$ otherwise $X_\tau = \bot$
X := U default V	$X_\tau = U_\tau$ when $U_\tau \neq \bot$ otherwise $X_\tau = V_\tau$
P | Q	compose P and Q following (4)

TABLE III
Signal operators (left) and their abstraction (right).

Z := X op Y	$h_Z = h_X = h_Y$ $(X, Y) \rightarrow Z$
Y := X$1	$h_X = h_Y$
X := U when B	$h_X = h_U$ when B B $\rightarrow$ X U $\rightarrow$ X when B
X := U default V	$h_X = h_U \vee h_V$ U $\rightarrow$ X V $\rightarrow$ X when $(h_V - h_U)$
P | Q	abstract(P) $\cup$ abstract(Q)

In the first two statements, the involved signals are either all present (they are $\neq \bot$) or all absent, in the considered reaction. We say that all signals involved have *the same clock*, and the corresponding statements are called *single-clocked*. In these first two statements, integer k represents instants at which signals are *present*, and "op" denotes a generic operation $+, \times, \ldots$ extended pointwise to sequences. Note that index k is implicit and does not appear in the syntax. The third and fourth statements are multi-clock ones. In the third statement, B is a boolean signal and *true* denotes the value "true." With this set of primitives, Signal supports the two synchronous execution schemes of Fig. 1. The default statement is an example of the first scheme, it corresponds to the Esterel statement

```
present U then emit X(?U) else present V then emit X(?V)
```

The generic op statement is an example of the first scheme, it coincides with the Lustre statement

```
Z = op(X,Y)
```

More generally, Signal allows the designer to mix *reactive* communication (offered by the environment) and *proactive* communication (demanded by the program).

The first two statements impose constraints on signals' clocks, which can be used to impose arbitrary constraints. Say you wish to assert the Boolean-valued expression C(X,Y,Z) always holds. Define B := C(X,Y,Z) and write B := B when B. This statement says B must be equal to B when B, which says B is either *true* or absent; B *false* is inconsistent. 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. Thus, invariant properties can be included in the specification and are guaranteed to hold by construction. This also allows incomplete or partial designs to be specified.

In Signal, reactions are therefore transition *relations* in general, so executing a reaction involves solving the corresponding fixpoint equation, cf. the discussion at the end of subsection I-B. In the current version of the compiler, this is achieved by applying the abstraction technique shown in table III. In this abstraction, h_X denotes the clock of signal X, defined by

$$h_X \equiv \text{if X} = \bot \text{ then } \bot \text{ else } \mathit{true},$$

and statement "U $\rightarrow$ X when B" is to be interpreted as "X causally depends on U when X and U are present and B is true"; U $\rightarrow$ X stands for U $\rightarrow$ X when *true*. Each Signal statement is abstracted as a set of equations, and symbol $\cup$ in this table refers to the union of such sets. The resulting abstract domain involves Boolean and clock types plus directed graphs, and equations can be solved in this abstract domain. Benveniste et al. [21] prove that if the abstraction of the program has a unique, functional solution then the original program does as well. This clock and causality calculus is the basis of Signal compilation. For convenience, "U $\rightarrow$ X when B" has been added as an actual primitive statement of Signal, since 1995, so the clock and causality calculus can be expressed using Signal itself.

Signal has an equational style. Hence, similarly to Lustre, its primary visualization is in the form of hierarchical block-diagrams (sometimes called also dataflow diagrams).

C.4 Variants

Many researchers have proposed extensions and variants of the synchronous language model. We only mention a few.

Some work has been done on developing common intermediate representations (OC, DC and DC_+ [22]) for the synchronous languages, allowing parts of different compilation chains to be combined.

The SyncCharts formalism [23] originated in work by Charles André at the University of Nice (France) and has since been incorporated in the commercial system from Esterel Techologies. SyncCharts attempts to provide a StateCharts-like graphical syntax with additional constructs that allow a fully synchronous semantics.

Also closely related to StateCharts, Maranichi's Argos [24] and its follow-up Mode-Automata formalism [25] were developed by the Lustre team to add states and control modes to declarative data-flow style of synchronous reactive programming.

Imperative formalisms such as state diagrams, automata, and Statecharts, can and have been expanded into Signal [26], [27].

ECL (Esterel C language, designed at Cadence Berkeley Labs) [28] and Jester (java-Esterel, designed at the italian PARADES consortium) [29] aim at providing Esterel constructs in a C or Java syntax. These appear to be increasingly successful at attracting a wider body of programmers: ECL has been integrated into the system released from Esterel Technologies.

The Synchronous Eiffel language, part of the Synchronie Workbench developed at the German National Research Center for Computer Science (aka GMD) [30], is an attempt to recon-

cile imperative Esterel features with the Lustre declarative style in a single object oriented framework. Lucid Synchrone [31], [32][3] takes advantage of the (first order) functional aspect of Lustre so as to generalise it to higher order constructs in a ML-like style. In particular the Lustre clock calculus is extended and inferred as an ML type system. Recently, a Scade compiler has been designed based on these principles.

Schneider has proposed the Quartz language [33], [34] as a verifiable alternative to Esterel that adds assertions and very precisely prescribed forms of nondeterminisim. Designed within the Higher Order Logic (aka HOL) theorem prover, Quartz has a mechanically proven translation to circuits.

II. Highlights of the Last Twelve Years

The last twelve years have seen a number of successful industrial uses of the synchronous languages. Here, we describe some of these engagements.

A. Getting Tools to Market

The Esterel compilers from Berry's group at INRIA/CMA have long been distributed freely in binary form[4]. The commercial version of Esterel was first marketed in 1998 by the French software company Simulog. In 1999, this division was spun off to form the independent company Esterel Technologies[5]. Esterel Technologies recently acquired the commercial Scade environment for Lustre programs to combine two complementary synchronous approaches.

In the early nineties, Signal was licensed to Techniques Nouvelles pour l'Informatique TNI[6], a French software company located in Brest, in western France. From this license, TNI developed and marketed the Sildex tool in 1993. Several versions have been issued since then, most recently Sildex-V6. Sildex supports hierarchical dataflow diagrams and state diagrams within an integrated GUI, and allows Simulink and Stateflow discrete-time models from Matlab to be imported. Globally Asynchronous Locally Synchronous (GALS) modeling is supported, see subsection V-B for details. Model checking is a built-in service. The RTBuilder add-on package is dedicated to real-time and timeliness assessments. TNI recently merged with Valiosys, a startup operating in the area of validation techniques, and Arexys, a startup developing tools for system-on-chip design. Together, they will offer tools for embedded system design.

B. The Cooperation with Airbus and Schneider Electric

Lustre's users pressed for its commercialization. In France in the 1980s, two big industrial projects including safety-critical software were launched independently: the N4 series of nuclear power plants, and the Airbus A320 (the first commercial fly-by-wire aircraft). Consequently, two companies, Aerospatiale (now Airbus Industries) and Merlin-Gerin (now Schneider Electric) were faced with the challenge of designing highly safety-critical software, and unsuccessfully looked for suitable existing tools. Both decided to build their own tools: SAO at Aerospatiale, and SAGA at Schneider Electric. Both of these proprietary tools were based on synchronous data-flow formalisms. SAGA used Lustre because of an ongoing cooperation between Schneider Electric and the Lustre research group. After some years of successfully using these tools, both companies were faced with the problem of maintaining and improving them, and admitted that it was not their job. Eventually, the software company Verilog undertook the development of a commercial version of SAGA that would also subsume the capabilities of SAO. This is the Scade environment, and it currently offers an editor that manipulates both graphical and textual descriptions; two code generators, one of which is accepted by certification authorities for qualified software production; a simulator; and an interface to verification tools such as Prover plug-in[7]. 2001 has brought further convergence: Esterel Technologies has purchased the Scade business unit.

Scade has been used in many industrial projects, including the integrated nuclear protection system of the new French nuclear plants (Schneider Electric), part of the flight control software of the Airbus A340-600, and re-engineered track control system of the Hong Kong subway (CS Transport).

C. The Cooperation with Dassault Aviation

Dassault Aviation was one of the earliest supporters of the Esterel project, and has long been one of its major users. They have conducted many large case studies, including the full specification of the control logic for an airplane landing gear system and a fuel management system that handles complex transfers between various internal and external tanks.

These large-scale programs, provided in Esterel's early days, were instrumental in identifying new issues for further research, to which Dassault engineers often brought their own preliminary solutions. Here are three stories that illustrate the pioneering role of this company on Esterel [35].

Some of the Dassault examples did exhibit legitimate combinational loops when parts were assembled. Compiling the full program therefore required constructive causality analysis. This was strong motivation for tackling the causality issue seriously from the start; it was not a purely theoretical question easily sidestepped.

Engineers found a strong need for modular or separate compilation of submodules, again for simply practical reasons. Indeed, some of the optimization techniques for reducing program size would not converge on global designs because they were simply too big. Some partial solutions have been found [36], but a unified treatment remains a research topic.

Object-oriented extensions were found to be desirable for "modeling in the large" and software reuse. A proposal for an Esterel/UML coupling was drafted by Dassault [37], which has since been adopted by and extended in the commercial Esterel tools.

D. The Cooperation with Snecma

During the 1980s, Signal was developed with the continuing support of CNET[8]. This origin explains the name. The origi-

[3] http://www-spi.lip6.fr/softs/lucid-synchrone.html
[4] http://www.esterel.org/
[5] http://www.esterel-technologies.com/
[6] http://www.tni-valiosys.com
[7] http://www.prover.com
[8] Centre National d'Etudes des Télécommunications, the former national laboratory for research in telecommunications, now part of France Telecom

nal aim of Signal was to serve as a development language for signal processing applications running on digital signal processors (DSPs). This led to its dataflow and graphical style and its handling of arrays and sliding windows, features common to signal processing algorithms. Signal became a joint trademark of CNET and INRIA in 1988.

Signal was licensed to TNI in the early nineties, and the cooperation between Snecma and TNI started soon thereafter. Snecma was searching for modeling tools and methods for embedded systems that closely mixed dataflow and automaton styles with a formally sound basis. Aircraft engine control was the target application. As an early adopter of Sildex, Snecma largely contributed to the philosophy and style of the toolset and associated method.

E. The Cooperation with Texas Instruments

Engineers at Texas Intruments design large DSP chips targeted at wireless communication systems. The circuitry is typically synthesized from a specification written in VHDL and the validation process consists of confronting the VHDL with a *reference* model written in C. Currently, validation is performed by simulating test sequences on both models and verifying their behavior is identical. The quality of the test suite, therefore, determines the quality of the validation.

Esterel was introduced in the design flow as part of a collaboration with Texas Instruments' Wireless Terminals Business Center, located at Villeneuve-Loubet (France) [38]. Parts of the C specification were rewritten in Esterel, mostly at specific safety-critical locations. The language provided formal semantics and an FSM interpretation so that tests could be automatically synthesized with the goal of attaining full coverage of the control state space. An early result from these experiments showed that the usual test suites based on functional coverage only exercised about 30% of the full state space of these components. With the help of dedicated algorithms on BDD structures full state coverage was achieved for the specifications considered, and further work was conducted on *transition* coverage, i.e., making sure the test suite exercised all program behaviors. Of course the coverage checks only apply to the components represented in Esterel, which are only a small part of a much larger C specification.

The work conducted in this collaboration proved highly beneficial, since such simulation-based validation seems to be prevalent in industry. With luck, the methodology established here can easily be applied elsewhere. Automatic test suite generation with guaranteed state/transition coverage is a current marketing lead for Esterel.

III. Some New Technology

Modeling and code generation are often required in the design flow of embedded systems. In addition, verification and test generation are of increasing importance due to the skyrocketing complexity of embedded systems. In this section we describe synchronous-language-related efforts in these areas.

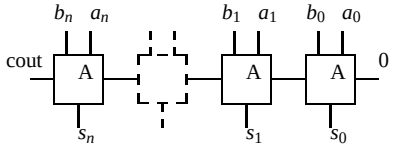
(FTR&D).

A. Handling arrays

In a dataflow language, arrays are much more than a data structure; they are a very powerful way of structuring programs and defining parameterized regular networks. For example, it is often useful to apply the same operator to all elements of an array (the "map" operation). Avoiding run-time array-index-out-of-bounds errors is another issue in the context of synchronous languages. Consequently, only restricted primitives must be provided to manipulate arrays.

Arrays were first introduced in Lustre [39] for describing circuits (arrays are virtually mandatory when manipulating bits). The Lustre-V4 compiles arrays by expansion: in both the circuit and the generated code an array is expanded into one variable per element. This is suitable for hardware, but can be very inefficient in software. Because instantaneous dependencies among array elements are allowed, compiling this mechanism into code with arrays and loops is problematic.

The Lustre-V4 experience showed that arrays are manipulated in a few common ways in most applications, suggesting the existence of a small set of "iterators." These iterators, which are well known in the functional language community, are convenient for most applications and can be easily compiled into small, efficient sequential code [40]. With these iterators, arrays can be compiled into arrays, operations on arrays can be compiled into loops, and many implicit intermediate arrays can be replaced with scalar variables ("accumulators") in the generated code. For example, consider a very simple sequential adder that obeys a classical iterator structure:



where A is a standard full adder that sums two input bits and a carry and produces a sum and carry out. If we expand this structure, all the array elements a[i], b[i], s[i] become separate variables in the code along with all the carries c[i], which are described as an auxiliary array in the source program. Using the iterator "map-red" (map followed by reduction) to describe the structure, the need for the auxiliary array vanishes and the generated code becomes a loop:

```
c = 0;
for (i = 0; i <= n; i++) {
   s[i] = A1(a[i], b[i], c);
   c = A2(a[i], b[i], c);
}
```

where A1 and A2 represent code for computing s and c in A.

The Scade tool implements these techniques and similar technology is provided in the Sildex tool. One typical operator has the form A[B], where B is an array of integers and A is an array of any type. This is treated as a composition of maps where B represents a (possibly multidimensional) iteration over the elements of A. Extensions of this basic mechanism are provided.

B. New Techniques for Compiling Esterel

The first Esterel compilers were based on the literal interpretation of its semantics written in Plotkin's structural operational

style [41]. The V1 and V2 compilers [42] built automata for Esterel programs using Brzozowski's algorithm for taking derivatives of regular expressions [43]. Later work by Gonthier [44] and others produced the V3 compiler [45], which drastically accelerated the automata-building process by simulating the elegant intermediate code (IC) format—a concurrent control-flow graph hanging from a reconstruction tree representing the hierarchical nesting of preemption statements. The automata techniques of the first three compilers work well for small programs, and produce very fast code, but they do not scale well to industrial-sized examples because of the state explosion problem. Some authors have tried to improve the quality of automata code by merging common elements to reduce its size [46] or performing other optimizations [47]. Nevertheless, none of these techniques are able to compile concurrent programs longer than about 1000 lines.

The successors to the automata compilers are based on translating Esterel into digital logic [48]. This translation is natural because of Esterel's synchronous, finite-state semantics. In particular, unlike automata, it is nearly one-to-one (each source statement becomes a few logic gates), so it scales very well to large programs. Although the executables generated by these compilers can be much smaller than those from the automata compilers, there is still much room for improvement. For example, there are techniques for reducing the number of latches in the generated circuit and improving both the size and speed of the generated code [49]. Executables generated by these compilers simply simulate the netlist after topologically sorting its gates.

The V4 compiler, the earliest based on the digital logic approach, met with resistance from users because it considered incorrect many programs that compiled under the V3 compiler. The problem stemmed from the V4 compiler insisting that the generated netlist be acyclic, effectively requiring control and data dependencies to be consistent in every possible state, even those which the program cannot possibly enter. The V3 compiler, by contrast, considers control and data dependencies on a state-by-state basis, allowing them to vary, and only considers states that the program appears to be able to reach.

The solution to the problem of the overly restrictive acyclic constraint came from recasting Esterel's semantics in a constructive framework. Initial inspiration arose from Malik's work on cyclic combinational circuits [50]. Malik considers these circuits correct if three-valued simulation produces a two-valued result in every reachable state of the circuit. In Esterel, this translates into allowing a cyclic dependency when it is impossible for the program to enter a state where all the statements in the cycle can execute in a single reaction. These semantics turn out to be a strict superset of the V3 compiler's.

Checking whether a program is constructive—that no static cycle is ever activated—is costly. It appears to require knowledge of the program's reachable states, and the implementation of the V5 compiler relies on symbolic state space traversal, as described by Shiple et al. [51]. Slow generated code is the main drawback of the logic network-based compilers, primarly because logic networks are a poor match to imperative languages such as C or processor assembly code. The program generated by these compilers wastes time evaluating idle portions of the program since it is resigned to evaluating each gate in the network in every clock cycle. This inefficiency can produce code a hundred times slower than that from an automata-based compiler [52].

Two recently-developed techniques for compiling Esterel attempt to combine the capacity of the netlist-based compilers with the efficiency of code generated by the automata-based compilers. Both represent an Esterel program as a graph of basic blocks related by control and data dependencies. A scheduling step then orders these blocks and code is generated from the resulting scheduled graph.

The compiler of Edwards [52] uses a concurrent control-flow graph with explicit data dependencies as its intermediate representation. In addition to traditional assignment and decision nodes, the graph includes fork and join nodes that start and collect parallel threads of execution. Data dependencies are used to establish the relative execution order of statements in concurrently-running threads. These are computed by adding an arc from each emission of a signal to each *present* statement that tests it.

An executable C program is generated by stepping through the control-flow graph in scheduled order and tracking which threads are currently running. Fork and join nodes define thread boundaries in the intermediate representation. Normally, the code for each node is simply appended to the program being generated, but when the node is in a different thread, the compiler inserts code that simulates a context switch with a statement that writes a constant into a variable that holds the program counter for the thread being suspended followed by a multiway branch that restores the program counter of the thread being resumed.

This approach produces fast executables (consistently about half the speed of automata code and as many as a hundred times faster than the netlist-based compilers) for large programs, but is restricted to compiling the same statically acyclic class of programs as the V4 compiler. Because the approach relies heavily on the assumption the existence of a static schedule, it is not clear how it can be extended to compile the full class of constructive programs accepted by the V5 compilers.

Another, related compilation technique due to Weil et al. [53] compiles Esterel programs using a technique resembling that used by compiled-code discrete-event simulators. French *et al.* [54] describe the basic technique: the program is divided into short segments broken at communication boundaries and each segment becomes a separate C function invoked by a centralized scheduler. Weil *et al.*'s SAXO-RT compiler works on a similar principle. It represents an Esterel program as an event graph: each node is a small sequence of instructions, and the arcs between them indicate activation in the current reaction or the next. The compiler schedules the nodes in this graph according to control and data dependencies and generates a scheduler that dispatches them in that order. Instead of a traditional event queue, it uses a pair of bit arrays, one representing the events to run in the current reaction, the other representing those that will run in the next. The scheduler steps through these arrays in order, executing each pending event, which may add events to either array.

Because the SAXO-RT compiler uses a fixed schedule, like

Edwards' approach and the V4 compiler it is limited to compiling statically acyclic programs and therefore does not implement Esterel's full constructive semantics. However, a more flexible, dynamic scheduler might allow the SAXO-RT compiler to handle all constructively valid Esterel programs.

An advantage of these two latter approaches, which only the SAXO-RT compiler has exploited [55], is the ability to impose additional constraints on the order in which statements execute in each cycle, allowing some control over the order in which inputs may arrive and outputs produced within a cycle. While this view goes beyond the idealized, zero-delay model of the language, it is useful from a modeling or implementation standpoint where system timing does play a role.

C. Observers for Verification and Testing

Synchronous observers [56], [57] are a way of specifying properties of programs, or, more generally, to describe non deterministic behaviors. A well-known technique [58] for verifying programs consists of specifying the desired property by means of a language acceptor (generally a Büchi automaton) describing the *unwanted* traces of the program and showing that the synchronous product of this automaton with the program has no behaviors, meaning that no trace of the program is accepted by the automaton.

We adopt a similar approach, restricting ourselves to *safety properties* for the following reasons. First, for the applications we address, almost all the critical properties are safety properties: nobody cares that a train *eventually* stops, it must stop before some time or distance to avoid an obstacle. Second, safety properties are easier to specify since a simple finite automaton is sufficient (no need for Büchi acceptance criterion). Finally, safety properties are generally easier to verify. In particular, they are preserved by abstraction: if an abstraction of a program is safe, it follows that the concrete program is, too.

Verification by observers is natural for synchronous languages because *the synchronous product used to compute trace intersection is precisely the parallel composition provided by the languages.* This means safety properties can be specified with a special program called an observer that observes the variables or signals of interest and at each step decides if the property is fulfilled up to this step, emitting a signal that indicates whether it was. A program satisfies the property if and only if the observer never complains during any execution. This technique for specifying properties has several advantages:

1. The property can be written in the same language than the program. It is surely an argument to convince users to write formal specifications: they don't need to learn another formalism.
2. The observer can be executed; so testing it is a way to get convinced that it correctly expresses the user's intention. It can also be run during the actual execution of the program, to perform autotest.

Several verification tools use this technique [59], [18], [60], [61], [62]: in general, such a tool takes a program, an observer of the desired property, and an observer of the assumptions on the environment under which the property is intended to hold. Then, it explores the set of reachable states of the synchronous product of these three components, checking that if the property observer complains in a reachable state then another state was reached before where the assumption observer complained. In other words, it is not possible to violate the property without first violating the assumption. Such an "assume-garantee" approach allows also compositional verification [63], [64].

Adding observers has other applications, such as automatic testing [65], [66]. Here, the property observer is used as an oracle that decides whether a test passes. Even more interesting is the use of an assumption observer, which may be used to automatically generate *realistic* test sequences that satisfy some assumptions. This is often critical for the programs we consider since they often *control* their environments, at least in part. Consequently one cannot generate interesting test sequences or check that the control is correct without assuming that the environment obeys the program's commands.

IV. Major Lessons

Now that time has passed, research has produced results, and usage has provided feedback, some lessons can be drawn. We summarize them in this section.

A. The Strength of the Mathematical Model

The main lesson from the last decade has been that the fundamental model [time as a sequence of discrete instants (3) and parallel composition as a conjunction of behaviors (4)] has remained valid and was never questioned. We believe this will continue, but it is interesting to see how the model resisted gradual shifts in requirements.

In the 1980s, the major requirement was to have a clean abstract notion of time in which "delay 1; delay 2" exactly equals "delay 3" (due to G. Berry), something not guaranteed by real-time languages (e.g., Ada) or operating systems at that time. Similarly, deterministic parallel composition was considered essential. The concept of reaction (3) answered the first requirement and parallel composition as a conjunction (4) answered the second. These design choices allowed the development of a first generation of compilers and code generators. Conveniently, unlike the then-standard asynchronous interleaving approach to concurrency, this definition of parallel composition (4) greatly reduces the state-explosion problem and made program verification feasible.

Compiling functional concurrency into embedded code running under the simple schemes of Fig. 1 allowed critical applications to be deployed without the need for any operating system scheduler. This was particularly attractive to certification authorities since it greatly reduced system complexity. For them, standard operating systems facilities such as interrupts were already dangerously unpredictable. The desire for simplicity in safety-critical real-time systems appears to be universal [67], so the synchronous approach seems to be an excellent match for these systems.

In the late 1980s, it appeared that both imperative and dataflow styles were useful and that mixing them was desirable. The common, clean underlying mathematics allowed multiple synchronous formalisms to be mixed without compromising rigor or mathematical soundness. For example, the imperative Esterel language was compiled into a logic-netlist-based intermediate representation that enabled existing logic optimization technology to be used to optimize Esterel programs.

In the early 1990s, research on program verification and synthesis lead to the need for expressing specifications in the form of invariants. Unlike programs, invariants are generally not deterministic, but fortunately nondeterminism fits smoothly into the synchronous framework. Signal has considered nondeterministic systems from the very beginning but has supplied it in a disciplined forms. Unlike other formalisms in which parallel composition is nondeterministic, nondeterminism in Signal is due exclusively to a fixpoint equation having multiple solutions, a property that can be formally checked whenever needed. Similarly, assertions were added to Lustre to impose constraints on the environment for use in proofs.

Defining the behavior in a reaction as a fixpoint equation resulted in subtle causality problems, mainly for Esterel and Signal (Lustre solves this trivially by forbidding delay-free loops). For years this was seen a drawback of synchronous languages. The hunt for causality analysis in Esterel was like a Mary Higgins Clark novel: each time the right solution was reported found, a flaw in it was reported soon thereafter. However, G. Berry's constructive semantics does appear to be the solution and has stood unchallenged for more than five years. A similar struggle occured for Signal behind the scenes, but this mathematical difficulty turned out to be a definite advantage. In 1995, dependency was added as an first-class operator in both Signal and the DC and DC$_+$ common formats for synchronous languages. Besides encoding causality constraints, this allowed scheduling constraints to be specified explicitly and has proven to be an interesting alternative to the use of the imperative ";".

In the late 1990s, the focus has extended beyond simple programming to the design of systems. This calls for models of components and interfaces with associated behavioral semantics. While this is still research under progress, initial results indicate the synchronous model will continue to do the job.

Everything has its limits, including the model of synchrony. But its clean mathematics has allowed the study of synchrony intertwined with other models, e.g., asynchrony.

B. Compilation Has Been Surprisingly Difficult but Was Worth the Effort

Of the three languages, Esterel has been the most challenging to compile because its semantics include both control and data dependencies. We described the various compilation technologies in subsection III-B, and observe that while many techniques are known, none is considered wholly satisfactory.

The principle of Signal compilation has not changed since 1988: it consists of solving the abstraction of the program described by the clock and causality calculus shown in table III. However it was not until 1994 that it was proven that the internal format of the compiler was a canonical form for the corresponding program [68], [69], therefore guaranteeing the uniqueness of this format and establishing the compilation strategy on a firm basis.

Properly handling arrays within the compilation process has been a difficult challenge, and no definitive solution exists yet.

Compilation usually requires substantially transforming the program, usually making it much harder to trace, especially in the presence of optimization. Unfortunately, designers of safety-critical systems often insist on traceability because of certification constraints. Some effort has been made to offer different trade-offs between efficiency and traceability. Separate compilation is one possible answer, and direct solutions are also available. This has been an important focus for the Scade/Lustre compiler, which is DO178B certified.

Overall, the developed compilation techniques can be regarded as a deep and solid body of technologies.

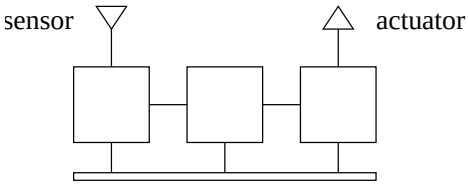


Fig. 5. Synchronous computers communicating via serial lines and a bus.

V. Future Challenges

Rather than a complete methodology, synchronous programming is more a step in the overall system development process, which may include assembling components, defining architecture, and deploying the result. A frequently-leveled complaint about the synchronous model is that not all execution architectures comply with it, yet the basic paradigm of synchrony demands global specifications. Exploring the frontiers of synchrony and beyond is therefore the next challenge. In this section we discuss three topics that toe this boundary: architecture modeling, deployment on asynchronous architectures, and building systems from components.

A. Architecture Modeling

As seen before, synchronous languages are excellent tools for the *functional* specification of embedded systems. As discussed in more details in subsection V-B, embedded applications are frequently deployed on architectures which do not comply with the execution model of synchrony examplified in Fig. 1. Typical instances are found in process industries, automobiles or aircrafts, in which the control system is often distributed over a communication infrastructure consisting of buses or serial lines. Fig. 5 shows a situation of interest. In this figure, a distributed architecture for an embedded system is shown. It consist of three computers communicating via serial lines and a bus. The computer on the left processes data from some sensor, the computer on the right computes controls for some actuator, and the central computer supervises the system. There are typically two sources of deviation from the synchronous execution model in this architecture:

1. The bus and the serial lines may not comply with the synchronous model, unless they have been carefully designed with this objective in mind. The family of Time Triggered Architectures (TTA) is such a synchrony compliant architecture [70], [71], [72]. But most are not, and still are used in embedded sytems.
2. The A/D and D/A converters and more generally the interfaces with the analog world of sensors and actuators, by definition, lie beyond the scope of the synchronous model.

While some architectures avoid the difficulty 1, difficulty 2 cannot be avoided. Whence the following

Problem 1 (Architecture Modeling) *How to Assist the Designer in Understanding how Her/His Synchronous Hpecification Behaves, when Faced with Difficulties 1 and/or 2?*

Assume that the considered synchronous specification decomposes as $P = P_1 \parallel P_2 \parallel P_3$ and the three components are mapped, from left to right, onto the three processors shown in Fig. 5. Assume, for the moment, that each P_i can run "infinitely fast" on its processor. Then the difficulties are concentrated on the communications between these processors: they cannot be considered synchronous. Now, the possible behaviors of each communication element (sensor, actuator, serial line, bus) can be modeled by resourcing to the *continuous* real-time of analog systems. *Discrete time* approximations of these models can be considered.

Take the example of a serial line consisting of a FIFO queue of length 1, its continuous time model says:

always:
FIFO can be written if empty just before, and
FIFO can be read if full just before

where "empty just before" means that the FIFO has been empty for some time interval until now. Now, the corresponding discrete time approximation reads exactly the same. But the above model is just an invariant synchronous system, performing a reaction at each discretization step—it does not, however, behave functionally. As explained in subsection I-C.3, this invariant can be specified in Signal. It can also be specified using the mechanism of Lustre assertions which are statements of the form assert B, where B is a boolean flow—the meaning is that B is asserted to be always true.

This trick can then be applied to (approximately) model all communication elements. The deployment of specification P on the architecture of Fig. 5 is then modelled as P', where

$$P' = ((\text{sensor} \parallel P_1) \parallel \text{serial} \parallel P_2 \parallel \text{serial} \parallel (P_3 \parallel \text{actuator})) \parallel \text{bus}$$

The resulting program can be confronted to formal verifications. By turning nondeterminism into additional variables, it can also be simulated.

Now, what if the assumption that the three processors run infinitely fast cannot be accepted? All compilers of synchronous languages compute schedulings that comply with the causality constraints imposed by the specification, see the related discussions in subsection I-C. This way, each reaction decomposes into atomic actions that are partially ordered, call a *thread* any maximal totally ordered sequence of such atomic actions. Threads, that are too long for being considered instantaneous with respect to the time discretization step, are broken into successive *microthreads* executed in successive discretization steps. The original synchronous semantics of each P_i is preserved if we make sure that microthreads from different reactions do not overlap. We just performed synchronous *time refinement*.

The above technique has been experimented in the SafeAir project using the Signal language, and in the Crisys project using the Lustre language [73], [74]. It is now available as a service for architecture modelling with Sildex-V6[9]. So much for problem 1. However, the alert reader must have noticed in passing that the breaking of threads into microthreads is generally subtle, and therefore calls for assistance. To this end we consider the next

Problem 2 (Architecture Profiling) *How to Profile a Scheduled Architecture?*

Look closely at the discussions on causality and scheduling in Section I-C, and particularly in Section I-C.3. Reactions decompose into atomic actions that are partially ordered by the causality analysis of the program. Additional *scheduling constraints* can be enforced by the designer, provided they do not contradict the causality constraints. This we call the *scheduled* program; note that it is only partially ordered, not totally. Scheduled programs can be, for instance, specified in Signal by using the statement "$U \rightarrow X$ when B" introduced in table III—in doing so, the latter statement is used for encoding both the causality constraints and the additional scheduling constraints set by the designer.

Now, assume that $(U,V) \rightarrow X$ holds in the current reaction of the considered scheduled program, i.e., (U,V) are the closest predecessors of X in the scheduled program. Denote by d_X the earliest date of availability of X in the current reaction. We have

$$d_X = \max(d_U, d_V) + \delta_{op}, \tag{6}$$

where δ_{op} is the additional duration of the operator needed (if any) to produce X from U and V. Basically, the trick used consists in replacing the scheduled program

$X := U \text{ op } V \mid (U,V) \rightarrow X$

by (6). If we regard (6) as a Signal program, the resulting mapping is an *homomorphism* of the set of Signal programs into itself. Performing this systematically, from a scheduled program, yields an associated *profiling* program, which models the timing behavior of the original scheduled program. This solves problem 2. This technique has been implemented in Signal [75].

A related technique has been developed at Verimag and FTR&D Grenoble in the framework of the TAXYS project [76], for profiling Esterel designs.

B. Beyond Synchrony

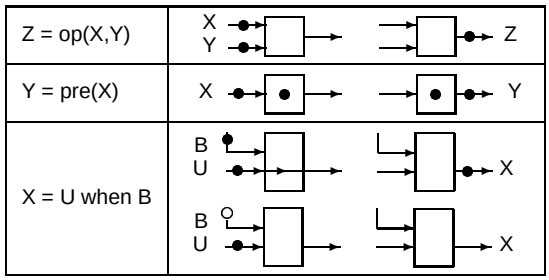
The execution schemes of Fig. 1 are relevant for embedded systems, but they do not encompass all needs. While the synchronous model is still pervasive in synchronous hardware, Globally Asynchronous Locally Synchronous (GALS) architectures are now considered for hardware built from components, or for hardware/software hybrid architectures such as encountered in ystem-on-a-chip designs [77]. Similarly, embedded control architectures, e.g., in automobiles or aircraft, are distributed ones. While some recent design choices favor so-called Time-Triggered architectures [70] complying with our model of synchrony, many approaches still rely on a (partially) asynchronous medium of communication.

It turns out that the model (3) and (4) of synchrony was clean enough to allow a formal study of its relationships with some classes of asynchronous models. This allowed the development of methods for deploying synchronous designs on asynchronous architectures. It also allowed the study how robust the deployment of a synchronous program can be, on certain classes of asynchronous distributed real-time architectures.

[9] http://www.tni-valiosys.com

TABLE IV

From Lustre statements (left) to dataflow actors (right).



Synchronous programs can be deployed on GALS architectures satisfying the following assumption:

Assumption 1: the architecture obeys the model of a network of synchronous modules interconnected by point-to-point wires, one per each communicated signal; each individual wire is lossless and preserves the ordering of messages, but the different wires are not mutually synchronized.

An important body of theory and techniques have been developed to support the deployment of synchronous programs onto such GALS architectures [21], [78], [79]. We shall give a flavor of this by explaining how Lustre and Signal programs can be maped to an asynchronous network of dataflow actors in the sense of Ptolemy [80], [81], an instance of an architecture satisfying assumption 1. In this model, each individual actor proceeds by successive salvos; in each salvo, input tokens are consumed and output tokens are produced. Hence each individual actor can be seen as a synchronous machine. Then, tokens travel along the network wires asynchronously, in a way compliant with assumption 1.

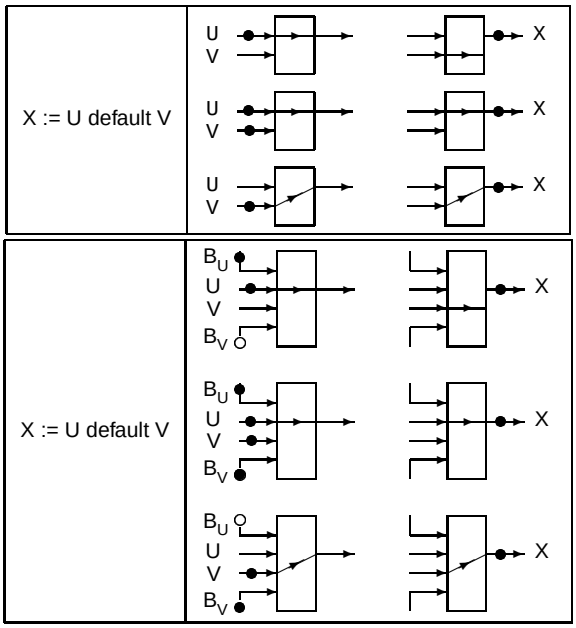
Table IV shows the translation for Lustre. The aim is that each Lustre statement would be replaced by its associated dataflow actor; thus a Lustre program would result in a token based dataflow network. The right hand side of table IV reads as follows : the diagrams on the left show the enabling condition of the dataflow actor; 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 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 operator pre is modeled by the presence of an initial token inside the actor. For the when statement, in 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, the U token passes the "gate", whereas it is lost when the boolean guard holds a *false* token. This performs data dependent downsampling. Note that, unlike in the when operator of Signal (table II), both inputs U and B must have the same clock in the Lustre when statement.

The firing of an actor is interpreted as the corresponding Lustre statement performing a reaction. It is proved [82] that a structural translation of any clock consistent[10] Lustre program using this table yields a dataflow network. Furthermore, in this network a single-token buffer execution is possible. Therefore any Lustre program possesses a fully asynchronous interpretation as well. Based on a similar remark, Lustre programs were deployed on a distributed architecture via the objecto code (OC) automaton-format [83]. Since then, it appeared that this deployment technique closely relates to the old-fashioned Gilles Kahn's dataflow networks [84].

[10]As said at the end of subsection I-C.1, Lustre clocks can be used as a type system, and corresponding type consistency can be checked.

TABLE V

The Signal default statement and its associated dataflow actors: a first and incorrect attempt—top, and the correct version—bottom.



This was easy, since Lustre is a *functional* language, i.e, a language in which each correct program behaves functionally. The case of Signal is different, as can be expected from the fact that Signal regards reactions as relations, not functions.

Referring to table II, a first attempt of translation of the default is depicted in table V, top. However this translation is incorrect, as the actor cannot tell the difference between "no token" and "late token." This missing information is provided in the correct translation shown in table V, bottom. The main point here is that *additional signaling is needed* in the form of the boolean guards B_U and B_V indicating the presence (value *true*) and absence (value *false*) of U and V. These guards are not part of the original Signal statement. This augmentation can be systematically performed, for each Signal primitive statement. This translation requires introducing additional signaling and operators. This overhead is the price to pay for the multiclock nature of Signal: the ambient reaction is not visible in Signal, it has to be made explicit prior to the translation into dataflow actors.

Clearly, a naive structural mapping would result in an unacceptable overhead and is not applicable. Instead, by performing a powerful symbolic analysis of the clock set of this program, one can transform it into a Lustre-like program by synthesizing a minimal additional signaling. Then translation into dataflow actors becomes straightforward, as seen from table IV. For a formal theory supporting this technique, the reader is referred

to [85], [21], [78], [86], where the two fundamental concepts of *endochrony* (informally discussed here) and *isochrony* (not discussed here) were introduced. 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[11].

The method applies to any asynchronous communication infrastructure satisfying the above assumption 1. GALS implementations can be derived in this way. Extensions of this method can be used in combination with separate compilation, to design systems by assembling components within a GALS architecture.

C. Real-time and logical time

There is a striking difference between the two preceding approaches which both describe some aspects of desynchronisation: in the GALS situation V-B, desynchronised programs stay functionally equivalent to their original synchronised versions and thus any design and validation result that has been obtained in the synchronous world remains valid in the desynchronised one. This is not the case in the approach of Fig. 5; here, we have shown how to faithfully mimic the architecture within the synchronous framework but it is quite clear that the program P' that mimics the implementation will in general not behave like the synchronous specification P. This is due to the fact that the added architectural features do not behave like ideal synchronous communication mechanisms.

But this is a constant situation: in many real time systems, real devices do not behave like ideal synchronous devices, and some care has to be taken when extrapolating design and validation results from the ideal synchronous world to the real real-time world.

This kind of problems have been investigated within the Crisys Esprit project and some results of this investigation can be found in [87]. Some identified reasons for such a thorough extrapolation are as follows:

1. Continuity—The extrapolation of analog computation techniques has played an important part in the origin of synchronous programming, and continuity is clearly a fundamental aspect of analog computing. Therefore, it is not surprising that it also plays a part in synchronous programming.
2. Bounded variability—Continuity is important because it implies some bandwidth limitation. This can be extended to non-continuous cases, provided systems don't exhibit unbounded variability.
3. Race avoidance—However, bounded variability is not enough when sequential behaviours are considered, because of intrinsic phenomena like "essential hazards." This is why good designers take a great care to avoid critical races in their designs, so as to preserve validation results. This implies using in some cases asynchronous programming techniques, e.g., causality chains.

D. From Programs to Components and Systems

Building systems from components is accepted as the today and tomorrow solution for constructing and maintaining large and complex systems. This also holds for embedded systems, with the additional difficulty that components can be hybrid hardware/software made. Object oriented technologies have had this as their focus for many years. Object oriented design of systems has been supported by a large variety of notations and methods, and this profusion eventually converged to the the Universal Modeling Language (UML)[12]) standard [88].

Synchronous languages can have a significant contribution for the design of embedded systems from components, by allowing the designer to master accurately the behavioral aspects of her/his application. To achieve this, synchronous languages must support the following:

1. Genericity and inheritance—While sophisticated typing mechanisms have been developed to this end in object oriented languages, the behavioral aspects are less understood, however. Some behavioral genericity is offered in synchronous languages by providing adequate polymorphic primitive operators for expressing control, see section I-C for instances of such operators. Behavioral inheritance in the framework of synchronous languages is far less understood and is still a current topic for research.
2. Interfaces and abstractions—We will show that the synchronous approach offers powerful mechanisms to handle the behavioral facet of interfaces and abstractions.
3. Implementations, separate compilation, and imports—Handling separate compilation and performing imports with finely tuning the behavioral aspects is a major contribution of synchronous languages, which we shall develop hereafter.
4. Multi-faceted notations, à la UML.
5. Dynamicity—Dynamic creation/deletion of instances is an important feature of large systems. Clearly, this is also a dangereous feature when critical systems are considered.

In this section we concentrate on the items 2, 3, 4, and provide some hints for 5.

D.1 What are the proper notions of abstraction, interface and implementation, for synchronous components?

See [89] and references therein for general discussions on this topic. We discuss these matters in the context of Signal. Consider the following Signal component P:

```
Y := f(X) | V := g(U)
```

It consists of the parallel composition of two single-clocked statements Y := f(X) and V := f(U), be careful that no relation is specified between the clocks of the two inputs X and U, meaning that they can independently occur at any reaction of P. A tentative implementation of P could follow the second execution scheme of Fig. 1, call it P':

```
for each clock tick do
    check presence of X ; present X then (compute Y:=f(X)   ; emit Y) ;
    check presence of U ; present U then (compute V:=g(U) ; emit V) ;
end
```

We can equivalently represent implementation P' by the following Signal program, we call it again P':

```
Y := f(X) | Y → U | V := g(U)
```

[11] http://www.irisa.fr/espresso/welcome_english.html

[12] http://uml.systemhouse.mci.com/

The added scheduling constraint "Y → U" in P' expresses that, when both Y and U occur in the same reaction, emission of Y should occur prior the reading of U. Now, consider another component Q:

```
X := h(V)
```

It can be implemented as Q':

```
for each clock tick do
    check presence of V ; present V then (compute X:=h(V) ; emit X) ;
end
```

Now, the composition R ≡ P || Q of these two components is the system

```
Y := f(X) | V := g(U) | X := h(V)
```

it can, for instance, be implemented by R':

```
for each clock tick do
    check presence of U ; present U then (compute V:=g(U) ; emit V) ;
    check presence of V ; present V then (compute X:=h(V) ; emit X) ;
    check presence of X ; present X then (compute Y:=f(X)  ; emit Y) ;
end
```

Unfortunately, the parallel composition of the two implementations P' and Q' is blocking, since P' will starvate, waiting for X to come from component Q', and symmetrically for Q'. This problem is indeed best revealed using the Signal writing of the parallel composition of P' and Q':

```
( Y := f(X) | Y → U | V := g(U) ) | X := h(V)
```

which exhibits the causality circuit:

```
X → Y | Y → U | U → V | V → X
```

Therefore, according to *[89]*, P' cannot be considered as a valid implementation of P for subsequent reuse as a component, since composing P' and Q' does not yield a valid implementation of R! This is too bad, since a standard compilation of P would typically yield P' as an executable code. The following lessons can be derived from this example, about abstraction and implementations of synchronous components:

1. Brute force separate compilation of components can result in being unable to reuse components for designing systems.
2. Keeping causality and scheduling constraints explicit is required when performing the abstraction of a component.

What can be valid abstractions and implementations, for synchronous components? Consider the following component P':

```
Y := f(X) | X := h(W) | V := g(U)
```

in which X is a local signal. A valid implementation of P' is P":

```
  Y := f(X) | X := h(W) | V := g(U)
| Y ← X    | X ← W     | V ← U
```

which is obtained by simply making the causality constraints explicit. P" can be rewritten as:

```
  ( Y := f(X) | X := h(W) | Y ← X ← W )
| ( V := g(U)             | V ← U )
```

which is more illuminating, since it shows the structuration of P" into two sequential *threads*, which must be kept concurrent. Therefore, a valid implementation of component P' does *not* obey any of the execution schemes of Fig. 1, but rather has the form of *concurrent threads supervised by a scheduler, where each thread can be safely separately compiled*—for the particular case of P" the scheduler is trivial, since the two threads can freely occur at each reaction.

Next, we claim that a valid abstraction of P' is obtained as follows. Start from P". Abstract Y := f(X) into its clock abstraction $h_Y = h_X$, and similarly for other statements. This yields the program:

```
  h_Y = h_X | h_X = h_W | h_V = h_U
| Y ← X     | X ← W     | V ← U
```

and we can hide in it the local signal X. The result is a valid abstraction for P', denote it by P:

```
  h_Y = h_W | h_V = h_U
| Y ← W     | V ← U
```

Of course, less wild abstractions can be performed, in which not every signal is replaced by its clock—e.g., boolean signals can be kept.

To summarize, a theory of synchronous components uses causality/scheduling constraints as a fundamental tool. This is definitely a novel feature, compared with usual approaches relying on trace- or bisimulation-based refinements (see [89] and references therein). Component-based design is supported by Signal in its latest version[13], by means of a set of services, including abstractions, imports, and implementations.

Performing separate compilation for Esterel, based on the *constructive causality*, raises similar difficulties. In fact the current algorithm checking constructive causality computes a more refined result asserting the necessary preconditions for causal correctness; this plays a role alike Signal's causality constraints just discussed before. Research is in progress to figure how this information can be best kept and exploited, for separate compilation. Also, it appears that the new compilations techniques developed by Edwards, or Weil *et al.* (see subsection III-B) use an internal representation of programs similar to the above discussed structures developed in the context of Signal.

D.2 Multi-faceted notations à la UML, and related work

The UML community has done a very nice job at defining a multi-faceted set of views for a system and its components. The different views provide information on the structure of the system (class and object diagrams), some high-level scenarios for use, involving different objects of the system (use cases, sequence diagrams), and behavioral notations to specify the behavior of components (state diagrams, Statecharts). How can the advantages of this multifaceted approach be combined with those of the synchronous approach?

The Esterel Studio toolset from Esterel Technologies offers a coupling with the class and object diagrams to help structuring the system description. However, lifting the multi-faceted modeling approach to the synchronous paradigm is still under progress. Charles André *et al.* [90] have proposed a version of the sequence diagrams, which is fully compliant with the synchronous model. Benoît Caillaud *et al.* [91] have proposed the formalism BDL as a semantic backbone supporting the different behavioral notations of UML (scenarios and state diagrams), and providing a clear semantics to systems, not just components.

[13] http://www.irisa.fr/espresso/welcome_english.html

D.3 Dynamic instantiation, some hints

We just conclude this subsection by discussing this much more prospective topic: can synchronous components be dynamically instantiated? A truly functional version of Lustre has been proposed by Caspi and Pouzet [32], which offers certain possibilities for dynamic instanciation of nodes. The semantics either can be strictly synchronous or can be derived from the asynchronous Kahn networks [84]. On the other hand, the work of Benveniste *et al.* on endo/isochrony [21], [78] opens possibilities for the dynamic instanciation of synchronous components in a GALS architecture, in which the communication between components is asynchronous. But the most advanced proposal for an implementation is the toolset Reactive Programming and the language Sugar Cubes for Java [14] proposed by Frédéric Boussinot. Its core consists of a set of Java classes implementing logical threads and a mechanism of global reactions relying on a global "stop" control point, shared by all objects. The corresponding model is synchronous (it is even single-clocked). Reactive Programming can also be used with a more usual but less formal asynchronous interpretation, and dynamic instanciation can be used in the latter case.

VI. Conclusion and perspectives

The synchronous programming approach has found its way in the application area of embedded systems. We have described some important features of them, mentioned some applications, and discussed some recent advances and future plans. One question remains about the future of this paradigm: where do we go?

For some time it was expected that synchronous technology would operate by hiding behind dominant methods and notations. One example is the Matlab/Simulink/Stateflow tool commonly used for signal processing algorithm design. Another example is UML and the general trend toward model engineering. This has happened to some extent, since Scade can import discrete time Simulink diagrams and Sildex can import Simulink/Stateflow discrete time diagrams. Here, the user can ignore the synchronous formalisms and notations and just and use the synchronous tools. A similar situation is occuring with Esterel Studio and Rational Rose.

Another way to bring synchronous technology to bear involves blending synchronous features for control into widespread languages. An example of this is the ECL language, mentioned in subsection I-C.4, in which features from Esterel are embedded into C. Synchronous analysis techniques are currently limited to the Esterel-derived portion of the language, but this could be enlarged in the future by adding more static analysis techniques.

An often-asked question is, why several languages instead of just one? The common formats OC, DC, and DC_+ (see subsection I-C.4) were attempts to achieve this quietly that have not panned out. Instead, more dataflow-like constructs have been merged into Esterel, such as Lustre's pre. Esterel Technologies' recent acquisition of the Lustre-derived Scade tool paves the way for more such additions.

Today, we see with some surprise that visual notations for synchronous languages have found their way to successful industrial use with the support of commercial vendors. This probably reveals that building a visual formalism on the top of a mathematically sound model gives actual strength to these formalisms and makes them attractive to users. The need toward integrating these technologies in larger design flows remains, but is achieved via suitable links with other tools. And this seems to be the winning trend nowadays.

[14] http://www-sop.inria.fr/mimosa/rp/

References

[1] A. Benveniste and G. Berry, "Prolog to the special section on another look at real-time programming," *Proceedings of the IEEE*, vol. 79, pp. 1268–1269, Sept. 1991.

[2] A. Benveniste and G. Berry, "The synchronous approach to reactive real-time systems," *Proceedings of the IEEE*, vol. 79, pp. 1270–1282, Sept. 1991.

[3] F. Boussinot and R. de Simone, "The Esterel language," *Proceedings of the IEEE*, vol. 79, pp. 1293–1304, Sept. 1991.

[4] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, pp. 1305–1320, Sept. 1991.

[5] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire, "Programming real-time applications with SIGNAL," *Proceedings of the IEEE*, vol. 79, pp. 1321–1336, Sept. 1991.

[6] IEEE Computer Society, 345 East 47th Street, New York, New York, *IEEE Standard VHDL Language Reference Manual (1076-1993)*, 1994. http://www.ieee.org/.

[7] IEEE Computer Society, 345 East 47th Street, New York, New York, *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language (1364-1995)*, 1996. http://www.ieee.org/.

[8] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, pp. 231–274, June 1987.

[9] R. David and H. Alla, *Petri nets and Grafcet: tools for modelling of discrete event systems*. Eglewood Cliffs, NJ: Prentice Hall, 1992.

[10] R. David, "Grafcet: a powerful tool for the specification of logic controllers," *IEEE Transactions on Control Systems Technology*, vol. 3, pp. 253–268, 1995.

[11] M. von der Beeck, "A comparison of Statecharts variants," in *Formal Techniques in Real-Time and Fault-Tolerant Systems: Third International Symposium Proceedings*, vol. 863 of *Lecture Notes in Computer Science*, Springer-Verlag, 1994.

[12] D. Harel and A. Naamad, "The Statemate semantics of Statecharts," *ACM Transactions on Software Engineering and Methodology*, vol. 5, pp. 293–333, Oct. 1996.

[13] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, "LUSTRE: A declarative language for programming synchronous systems," in *ACM Symposium on Principles of Programming Languages (POPL)*, (Munich), Jan. 1987.

[14] G. Berry, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, ch. The Foundations of Esterel. MIT Press, 2000.

[15] G. Berry, "The constructive semantics of pure Esterel." Book in preparation available at http://www.Esterel.org, 1999.

[16] D. E. Thomas and P. R. Moorby, *The Verilog Hardware Description Language*. Kluwer, fourth ed., 1998.

[17] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, "SIS: A system for sequential circuit synthesis," Tech. Rep. UCB/ERL M92/41, University of California, Berkeley, May 1992.

[18] A. Bouali, "Xeve: an Esterel verification environment," in *Tenth International Conference on Computer-Aided Verification, CAV'98*, (Vancouver (B.C.)), LNCS 1427, Springer Verlag, June 1998.

[19] A. Benveniste and P. L. Guernic, "Hybrid dynamical systems theory and the SIGNAL language," *IEEE Trans. Automat. Contr.*, vol. AC-35(5), pp. 535–546, 1990.

[20] A. Benveniste, P. L. Guernic, and C. Jacquemot, "Programming with events and relations: the SIGNAL language and its semantics," *Science of Computer Programming*, vol. 16, pp. 103–149, 1991.

[21] A. Benveniste, B. Caillaud, and P. L. Guernic, "Compositionality in dataflow synchronous languages: specification & distributed code generation," *Information and Computation*, vol. 163, pp. 125–171, 2000. http://www.irisa.fr/sigma2/benveniste/pub/BCLg99a.html.

[22] J.-P. Paris, G. Berry, F. Mignard, P. Couronné, P. Caspi, N. Halbwachs, Y. Sorel, A. Benveniste, T. Gautier, P. L. Guernic, F. Dupont, and C. L. Maire., "Projet synchrone : les formats communs des langages synchrones," Tech. Rep. 157, Irisa, june 1993.

[23] C. André, "Representation and analysis of reactive behaviors: A synchronous approach," in *Proceedings of Computational Engineering in*

Systems Applications (CESA), (Lille, France), pp. 19–29, July 1996. http://www-sop.inria.fr/meije/Esterel/syncCharts/.

[24] F. Maraninchi, "The Argos language: Graphical representation of automata and description of reactive systems," in *Proceedings of the IEEE Workshop on Visual Languages*, (Kobe, Japan), Oct. 1991.

[25] F. Maraninchi and Y. Rémond, "Mode-automata: About modes and states for reactive systems," in *Proceedings of the European Symposium On Programming (ESOP)*, (Lisbon (Portugal)), Springer-Verlag, Mar. 1998.

[26] Y. Wang, J. P. Talpin, A. Benveniste, and P. Le Guernic, "Compilation and distribution of state machines using SPOT S," in *16th IFIP World Computer Congress (WCC'2000)*, Aug. 2000. ftp://ftp.irisa.fr/local/signal/publis/articles//WCC-00.ps.gz.

[27] J.-R. Beauvais, E. Rutten, T. Gautier, P. Le Guernic, and Y.-M. Tang, "Modelling statecharts and activitycharts as signal equations," *ACM Transactions on Software Engineering and Methodology*, vol. 10, no. 4, 2001.

[28] L. Lavagno and E. Sentovich, "ECL: A specification environment for system-level design," in *Proceedings of the 36th Design Automation Conference*, (New Orleans, Louisana), pp. 511–516, June 1999.

[29] M. Antoniotti and A. Ferrari, "Jester, a reactive java extension proposal by Esterel hosting," 2000. http://www.parades.rm.cnr.it/projects/jester/jester.html.

[30] A. Poigne, M. Morley, O. Maffeis, L. Holenderski, and R. Budde, "The synchronous approach to designing reactive systems," *Formal Methods in System Design*, vol. 12, no. 2, pp. 163–187, 1998.

[31] P. Caspi and M. Pouzet, "Synchronous Kahn networks," in *Int. Conf. on Functional Programming*, ACM SIGPLAN, Philadelphia May 1996.

[32] P. Caspi and M. Pouzet, "A co-iterative characterization of synchronous stream functions," in *Proceedings of the Workshop on Coalgebraic Methods in Computer Science, Lisbon*, vol. 11 of *Electronic Notes in Theoretical Computer Science*, Elsevier, 1998. http://www.elsevier.nl/locate/entcs.

[33] K. Schneider, "A verified hardware synthesis for esterel programs," in *Proceedings of the International IFIP Workshop on Distributed and Parallel Embedded Systems (DIPES)*, (Paderborn, Germany), 2000.

[34] K. Schneider and M. Wenz, "A new method for compiling schizophrenic synchronous programs," in *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, (Atlanta, Georgia), Nov. 2001.

[35] G. Berry, A. Bouali, X. Fornari, E. Ledinot, E. Nassor, and R. De Simone, "Esterel: A formal method applied to avionic software development," *Science of Computer Programming*, vol. 36, pp. 5–25, Jan. 2000.

[36] O. Hainque, L. Pautet, Y. L. Biannic, and E. Nassor, "Cronos: A separate compilation toolset for modular Esterel applications," in *FM'99: World Congress on Formal Methods*, Lecture Notes in Computer Science 1709, 1999.

[37] Y. L. Biannic, E. Nassor, E. Ledinot, and S. Dissoubray, "Uml object specification for real-time software," in *Proceedings of the RTS 2000 Show*, (Paris), 2000. http://www.Esterel-technologies.com/Esterel/article3.pdf.

[38] L. Arditi, A. Bouali, H. Boufaied, G. Clave, M. Hadj-Chaib, L. Leblanc, and R. de Simone, "Using Esterel and formal methods to increase the confidence in the functional validation of a commercial DSP," in *Proceedings of the ERCIM Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, (Trento, Italy), June 1999.

[39] F. Rocheteau and N. Halbwachs, "POLLUX, a lustre-based hardware design environment," in *Conference on Algorithms and Parallel VLSI ArchitecturesII, Chateau de Bonas* (P. Quinton and Y. Robert, eds.), June 1991.

[40] L. Morel, "Efficient compilation of array iterators for lustre." In preparation, 2002.

[41] G. D. Plotkin, "A structural approach to operational semantics," Tech. Rep. DAIMI FN-19, Aarhus University, Åarhus, Denmark, 1981.

[42] G. Berry and L. Cosserat, "The ESTEREL synchronous programming language and its mathematical semantics," in *Seminar on Concurrency* (S. D. Brooks, A. W. Roscoe, and G. Winskel, eds.), pp. 389–448, Springer-Verlag, 1984.

[43] J. A. Brzozowski, "Derivates of regular expressions," *Journal of the Association for Computing Machinery*, vol. 11, pp. 481–494, Oct. 1964.

[44] G. Gonthier, *Sémantiques et modèles d'exécution des langages réactifs synchrones; application à Esterel. [Semantics and models of execution of the synchronous reactive languages: application to Esterel]*. Thèse d'informatique, Université d'Orsay, 1988.

[45] G. Berry and G. Gonthier, "The Esterel synchronous programming language: Design, semantics, implementation," *Science of Computer Programming*, vol. 19, pp. 87–152, Nov. 1992. ftp://cma.cma.fr/Esterel/BerryGonthierSCP.ps.Z.

[46] M. Chiodo, P. Giusto, A. Jurecska, L. Lavagno, H. Hsieh, K. Suzuki, A. Sangiovanni-Vincentelli, and E. Sentovich, "Synthesis of software programs for embedded control applications," in *Proceedings of the 32nd Design Automation Conference*, (San Francisco, California), pp. 587–592, June 1995. ftp://ic.eecs.berkeley.edu/pub/HWSW/dac95.ps.gz.

[47] C. Castelluccia, W. Dabbous, and S. O'Malley, "Generating efficient protocol code from an abstract specification," *IEEE/ACM Transactions on Networking*, vol. 5, pp. 514–524, Aug. 1997.

[48] G. Berry, "Esterel on hardware," *Philosophical Transactions of the Royal Society of London. Series A*, vol. 339, pp. 87–104, 1992.

[49] H. Toma, E. Sentovich, and G. Berry, "Latch optimization in circuits generated from high-level descriptions," in *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, (San Jose, California), p. FIXME, Nov. 1996.

[50] S. Malik, "Analysis of cyclic combinational circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, pp. 950–956, July 1994.

[51] T. R. Shiple, G. Berry, and H. Touati, "Constructive analysis of cyclic circuits," in *Proceedings of the European Design and Test Conference*, (Paris, France), pp. 328–333, Mar. 1996. ftp://ic.eecs.berkeley.edu/pub/Memos_Conference/edtc96.SBT.ps.Z.

[52] S. A. Edwards, "An Esterel compiler for large control-dominated systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, Feb. 2002.

[53] D. Weil, V. Bertin, E. Closse, , M. Poize, P. Venier, and J. Pulou, "Efficient compilation of Esterel for real-time embedded systems," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, (San Jose, California), pp. 2–8, Nov. 2000.

[54] R. S. French, M. S. Lam, J. R. Levitt, and K. Olukotun, "A general method for compiling event-driven simulations," in *Proceedings of the 32nd Design Automation Conference*, (San Francisco, California), pp. 151–156, June 1995. http://suif.stanford.edu/papers/rfrench95.ps.

[55] V. Bertin, M. Poize, J. Pulou, and J. Sifakis, "Towards validated real-time software," in *Proceedings of the 12th Euromicro Conference on Real-Time Systems (ECRTS)*, (Stockholm, Sweden), pp. 157–164, June 2000. http://ecrts00.twi.tudelft.nl/.

[56] N. Halbwachs, F. Lagnier, and P. Raymond, "Synchronous observers and the verification of reactive systems," in *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93* (M. Nivat, C. Rattray, T. Rus, and G. Scollo, eds.), (Twente), Workshops in Computing, Springer Verlag, June 1993.

[57] N. Halbwachs and P. Raymond, "Validation of synchronous reactive systems: from formal verification to automatic testing," in *ASIAN'99, Asian Computing Science Conference*, (Phuket (Thailand)), Dec. 1999.

[58] M. Y. Vardi and P. Wolper, "An automata-theoretic approach to automatic program verification," in *Symposium on Logic in Computer Science*, June 1986.

[59] N. Halbwachs, F. Lagnier, and C. Ratel, "Programming and verifying real-time systems by means of the synchronous data-flow programming language Lustre," *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, Sept. 1992.

[60] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic, "Synthesis of discrete-event controllers based on the signal environment," *Discrete Event Dynamic System: Theory and Applications*, vol. 10, pp. 325–346, Oct. 2000.

[61] H. Marchand, E. Rutten, M. Le Borgne, and M. Samaan, "Formal verification of signal programs: Application to a power transformer station controller," *Science of Computer Programming*, 2000. to appear.

[62] B. Jeannet, "Dynamic partitioning in linear relation analysis. application to the verification of synchronous programs," *Formal Method in System Design*, 2001. to appear.

[63] M. Westhead and S. Nadjm-Tehrani, "Verification of embedded systems using synchronous observers," in *FTRTFT'96*, (Uppsala), LNCS 1135, Sept. 1996.

[64] L. Holenderski, "Compositional verification of synchronous networks," in *FTRTFT'2000*, (Pune, India), LNCS 1926, Sept. 2000.

[65] L. Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon, "Lutess: testing environment for synchronous software," in *Tool support for System Specification Development and Verification*, Advances in Computing Science, Springer, 1998.

[66] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs, "Automatic testing of reactive systems," in *19th IEEE Real-Time Systems Symposium*, (Madrid, Spain), Dec. 1998.

[67] N. Wirth, "Embedded systems and real-time programming," in *Embedded Software, First international workshop, EMSOFT 2001*, (Tahoe city, CA, USA), LNCS 2211, Springer Verlag, Oct. 2001.

[68] T. Amagbegnon, L. Besnard, and P. L. Guernic, "Arborescent canonical form of boolean expressions," Tech. Rep. 2290, Irisa Res. Rep., June 1994. ftp://ftp.inria.fr/INRIA/publication/RR/RR-2290.ps.gz.

[69] T. Amagbegnon, L. Besnard, and P. Le Guernic, "Implementation of the data-flow synchronous language signal," in *Programming Languages Design and Implementation*, pp. 163–173, ACM, 1995. ftp://ftp.irisa.fr/local/signal/publis/articles/PLDI-95:compil.ps.gz.

[70] H. Kopetz, *Real-time systems, design principles for distributed embedded applications, 3rd edition*. London: Kluwer academic publishers, 1997. ISBN 0-7923-9894-7.

[71] H. Kopetz and G. Bauer, "The Time Triggered Architecture," *Proceedings of the IEEE*, 2002. this special issue.

[72] T. Henzinger, B. Horowitz, and C. M. Kirsch, "A time-triggered language for embedded programming," *Proceedings of the IEEE*, 2002. this special issue.

[73] P. Caspi, C. Mazuet, R. Salem, and D. Weber, "Formal design of distributed control systems with Lustre," in *Proc. Safecomp'99*, vol. 1698 of *Lecture Notes in Computer Science*, Springer Verlag, September 1999.

[74] P. Caspi, C. Mazuet, and N. Reynaud-Parigot, "About the design of distributed control systems: the quasi-synchronous approach," in *Proc. Safecomp'01*, vol. 2187 of *Lecture Notes in Computer Science*, Springer Verlag, September 2001.

[75] A. Kountouris and P. Le Guernic, "Profiling of signal programs and its application in the timing evaluation of design implementations," in *Proceedings of the IEE Colloq. on HW-SW Cosynthesis for Reconfigurable Systems*, (HP Labs, Bristol, UK), pp. 6/1–6/9, IEE, Feb. 1996. ftp://ftp.irisa.fr/local/signal/publis/articles/HWSWCRS-96:profiling.ps.gz.

[76] V. Bertin, E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine, "Taxys = Esterel + Kronos, a tool for verifying real-time properties of embedded systems," in *Proceedings of the 40th IEEE conference of decision and control CDC2001*, (Orlando), p. 137, Dec. 2001.

[77] G. D. Micheli, R. Ernst, and W. Wolf, *Readings in Hardware/Software co-design*. San Francisco: Morgan Kaufmann publishers, Academic Press, 2002. ISBN 1-55860-702-1.

[78] A. Benveniste, B. Caillaud, and P. L. Guernic, "From synchrony to asynchrony," in *Proc. of CONCUR'99*, Aug 1999.

[79] A. Benveniste, "Some synchronization issues when designing embedded systems from components," in *Embedded Software, T.A. Henzinger and Chistoph Hirsch eds.*, vol. 2211, pp. 32–49, Berlin: LNCS, Springer Verlag, 2001.

[80] J. Buck, S. Ha, E. Lee, and D. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *International Journal of computer Simulation*, 1994. special issue on Simulation Software Development.

[81] J. Janneck, E. Lee, J. Liu, S. Neuendorffer, and S. Sachs, "Taming heterogeneity with hierarchy—the Ptolemy approach," *Proceedings of the IEEE*, 2002. this special issue.

[82] P. Caspi, "Clocks in dataflow languages," *Theoretical Computer Science*, vol. 94, pp. 125–140, 1992.

[83] P. Caspi, A. Girault, and D. Pilaud, "Automatic distribution of reactive systems for asynchronous networks of processors," *IEEE Transactions on Software Engineering*, vol. 25, pp. 416–427, May 1999. http://www.inrialpes.fr/bip/people/girault/Publications/Tse99/.

[84] G. Kahn, "The semantics of a simple language for parallel programming," in *Information Processing 74: Proceedings of IFIP Congress 74*, (Stockholm, Sweden), pp. 471–475, North-Holland, Aug. 1974.

[85] T. Gautier and P. Le Guernic, "Code generation in the sacres project," in *Towards System Safety, Proceedings of the Safety-critical Systems Symposium, SSS'99*, (Huntingdon, UK), Springer, Feb. 1999. ftp://ftp.irisa.fr/local/signal/publis/articles/SSS-99:format_dist.ps.gz.

[86] A. Benveniste and P. Caspi, "Distributing synchronous programs on a loosely synchronous, distributed architecture," Tech. Rep. 1289, Irisa, Dec. 1999. http://www.irisa.fr/sigma2/benveniste/pub/BC99.html.

[87] P. Caspi, "Embedded control: from asynchrony to synchrony and back," in *First International Wokshop on Embedded Software* (T. Henzinger and C. Kirsch, eds.), vol. 2211 of *Lecture Notes in Computer Science*, 2001.

[88] J. Rumbaugh, I. Jacobson, and G. Booch, *Tne Unified Modeling Language reference manual*. Object technologies series, Addison-Wesley, 1999.

[89] L. de Alfaro and T. Henzinger, "Interface theories for component-based designs," in *Embedded Software, T.A. Henzinger and Chistoph Hirsch eds.*, vol. 2211, pp. 148–165, Berlin: LNCS, Springer Verlag, 2001.

[90] C. André, M. Peraldi-Frati, and J. Rigault, "Scenario and property checking of real-time systems using a synchronous approach," in *Proceedings of ISORC'2001*, (Lille, France), July 2001.

[91] J. Talpin, A. Benveniste, B. Caillaud, C. Jard, Z. Bouziane, and H. Canon, "Bdl, a language of distributed reactive objects," in *Proceedings of ISORC'1998*, (Kyoto, Japan), 1998. see also http://www.irisa.fr/sigma2/benveniste/pub/CTJBJ2000.html.