Synchronous Programming of Reactive Systems*
A Tutorial and Commented Bibliography

Nicolas Halbwachs

Vérimag**, Grenoble – France
e-mail: Nicolas.Halbwachs@imag.fr

1 Reactive Systems

The term "*reactive system*" was introduced by David Harel and Amir Pnueli [HP85], and is now commonly accepted to designate permanently operating systems, and to distinguish them from "*transformational systems*" — i.e, usual programs whose role is to terminate with a result, computed from an initial data (e.g., a compiler). In synchronous programming, we understand it in a more restrictive way, distinguishing between "*interactive*" and "*reactive*" systems:
Interactive systems permanently communicate with their environment, but at their own speed. They are able to synchronize with their environment, i.e., making it wait. Concurrent processes considered in operating systems or in data-base management, are generally interactive.
Reactive systems, in our meaning, have to react to an environment which cannot wait. Typical examples appear when the environment is a physical process. The specific features of reactive systems have been pointed out many times [Hal93,BCG88,Ber89]:

- In contrast with most interactive systems, they are generally intended to be *deterministic*.
- Their description involves *concurrency*, for several different reasons:
  1. They run in parallel with their environment;
  2. They are often implemented on distributed architectures, for reasons of speed, fault-tolerance, or physical distribution requirements;
  3. Most of the time, it is convenient to describe them as sets of concurrent processes.

  Cases (2) and (3) must be distinguished. In the later case, concurrency is nothing but a description facility; we call it *logical concurrency*. Generally, it has nothing to do with *physical concurrency* involved in case (2), and is not submitted to the same constraints.
- They are submitted to critical reliability requirements. In fact, most critical systems either are reactive, or contain reactive parts.

* This work has been partially supported by the ESPRIT-LTR project "SYRF".

** Verimag is a joint laboratory of Université Joseph Fourier (Grenoble I), CNRS and INPG

```
< Initialize Memory >
foreach input_event do
    < Compute Outputs >
    < Update Memory >
end
```

a. "Event driven"

```
< Initialize Memory >
foreach period do
    < Read Inputs >
    < Compute Outputs >
    < Update Memory >
end
```

b. "Sampling"

Fig. 1. Execution schemes for reactive systems

2 Synchronous Programming

All control engineers know a simple way to implement a reactive system by a single loop, of the form shown by Fig. 1.a. This program scheme is "*event driven*" since each reaction is triggered by an input event.

Fig. 1.b shows an even simpler and more common scheme, which consists in periodically sampling the inputs. This "*sampling*" scheme is mainly used in numeric systems which solve, e.g., systems of differential equations. These two schemes do not deeply differ, but they correspond to different intuitive points of view. In both cases, the program typically implements an *automaton*: the states are the valuations of the memory, and each reaction corresponds to a transition of the automaton. Such a transition may involve many computations, which, from the automaton point of view, are considered *atomic* (i.e., input changes are only taken into account between two reactions). This is the essence of the synchronous paradigm, where such a reaction is often said to *take no time*. An atomic reaction is called an *instant* (logical time), and all the events occurring during such a reaction are considered *simultaneous*.

Now, automata are useful tools — from their simplicity, expressive power, and efficiency —, but they are very difficult to design by hand[1]. Synchronous languages aim at providing high level, modular, constructs, to make the design of such an automaton easier. The basic construct that all these languages provide, is a notion of synchronous concurrency, inspired by Milner's synchronous product [Mil81,Mil83]: in the sampling scheme, when automata are composed in parallel, a transition of the product is made of "*simultaneous*" transitions of all of them; in the event-driven scheme, some automata can stay idle, when not triggered by events coming either from the environment or from other automata. In any case, when participating in such a compound transition, each automaton considers the outputs of others as being part of its own inputs. This "*instantaneous*" communication is called the *synchronous broadcast* [BCG88,Ber89,BB91]. The important point is that, in contrast with the asynchronous concurrency considered in asynchronous languages like ADA [ADA83,Coh96], this synchronous product can preserve *determinism*, a highly desirable feature in reactive systems design.

There are two fields where this synchronous model has been used for years:

[1] Consider, e.g., scanners and parsers, and the usefulness of tools like LEX and YACC!

In synchronous circuit design, it is the usual model of communicating Mealy machines (FSM). Most hardware description formalisms (e.g., [Bli90,CLM91]) are naturally synchronous, or contain a significant synchronous subset [Per93]. As a matter of fact, the compilation and verification of synchronous programs borrow many techniques from circuit CAD. However, while hardware description languages can be directly used to describe the data part of a circuit, they are of little help in designing complex hardware controllers. This explains the success of synchronous imperative languages, like ESTEREL, in this field.

In control engineering, high level specification formalisms are often *data-flow* synchronous formalisms, inherited from earlier analog technology: differential or finite-difference equations, block-diagrams, analog networks. Interpreted in a discrete world, these models can be formalized using the data-flow paradigm [Kah74,AW85,PP83]. However, these formalisms are seldom used as programming languages, and automatic code generation is not available. On the other hand, more imperative languages used for programming automatic controllers (e.g., Sequential Function Charts [LM93,AG96]) generally follow the same cyclic execution scheme.

3 Synchronous Languages

[Hal93,IEE91] are general references on synchronous languages.

Statecharts [Har87] is probably the first, and the most popular, formal language designed in the early eighties for the design of reactive systems. However, they were proposed more as a specification and design formalism, rather than as a programming language. Many features (synchronous product and broadcast) of the synchronous model are already present in Statecharts, but determinism is not ensured, and many semantic problems were raised [vdB94]. Almost at the same time, three programming languages were proposed by French academic groups:

- ESTEREL[2] [BCG88,BS91,BG92,Ber93,Ber98] is an imperative language developed at the "Ecole des Mines" and Inria, in Sophia Antipolis.
- SIGNAL[3] [LGLL91,BL90] and LUSTRE[4] [HCRP91,CPHP87] are data-flow languages, respectively designed at Inria (Rennes) and CNRS (Grenoble). SIGNAL is more "event-driven", while LUSTRE mainly corresponds to the "sampled" scheme.

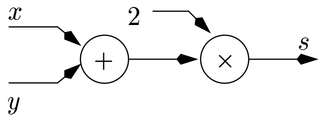
Also, following the formal definition of the synchronous model, a purely synchronous variant of the Statecharts was proposed: ARGOS[5] [Mar92]. The ideas of ARGOS are currently used as a basis for a graphical version of ESTEREL, named SYNCCHARTS[6] [And96], proposed at the University of Nice.

[2] see `http://www.inria.fr/meije/esterel/esterel-eng.html`
[3] see `http://www.inria.fr/Equipes/EPATR-eng.html`
[4] see `http://www.imag.fr/VERIMAG/SYNCHRONE/lustre-english`
[5] see `http://www-verimag.imag.fr/SYNCHRONE/argonaute-english.html`
[6] see `http://www.inria.fr/meije/esterel/syncCharts/`

In this section, we use simple examples to give a flavor of the programming styles in LUSTRE and ESTEREL.

3.1 Overview of the synchronous data-flow language Lustre

LUSTRE is based on the synchronous data-flow model, i.e., on a synchronous interpretation of block-diagrams. A block diagram may be viewed as a network of operators (or as a system of equations, see opposite) running in parallel at the rate of their inputs.



$$s = 2 \times (x + y)$$

The synchronous interpretation of such a description consists in considering each variable as taking a value at each cycle of the program. According to this interpretation, the above description means: "*at any cycle* n, $s_n = 2 * (x_n + y_n)$".

A LUSTRE program defines its output variables as functions of its input variables. Each variable or expression E denotes a function of discrete time, giving its value E_n at each "instant" n. Variables are defined by means of equations: an equation "X=E", specifies that the variable X is always equal to expression E.

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

- $(\mathsf{pre(E)})_n = \mathsf{E}_{n-1}$, while $(\mathsf{pre(E)})_0$ has the undefined value *nil*.
- $(\mathsf{E \text{ -> } F})_n = \mathsf{F}_n$, while $(\mathsf{E \text{ -> } F})_0 = \mathsf{E}_0$.

For instance, if x_n, y_n denote the respective values of x and y at "instant" n, the equation "z = 0->(pre(x) + y)" means that the initial value z_0 of z is 0, and that, at any non initial instant n, $\mathsf{z}_n = \mathsf{x}_{n-1} + \mathsf{y}_n$.

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

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

```
mod5 = Counter(0, 1, pre(mod5)=4);
```

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

(a) Program

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

(b) Behavior

Fig. 2. Example of LUSTRE program: A counter

which instanciates the node Counter, with 0 and 1 as constant initial and increment values, and resets it whenever the previous value of its output is 4. The variable mod5 is then the cyclic sequence of integers modulo 5.

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

From a temporal point of view, industrial applications show that several processing chains, evolving at different rates, can appear in a single system. LUSTRE offers a notion of boolean clock, allowing the activation of nodes at different rates.

3.2 Overview of the synchronous imperative language Esterel

Being an imperative language, ESTEREL looks more familiar at first glance, since it provides usual constructs, like assignments, sequences, loops, ... However, its synchronous semantics makes this apparent friendliness somewhat deceiptive: one must keep in mind that, apart from a few statements that explicitly take time (e.g., "await $< signal >$"), most ESTEREL statements are conceptually *instantaneous*, i.e., are executed in the same reaction than other statements that sequentially precede or follow them in the program.

ESTEREL provides a lot of constructs that we cannot present in detail. We only comment a small example, which is a speed supervisor (see Fig. 3): the program is intended to measure the speed of a vehicle, and to detect when this speed exceeds a maximum bound.

Fig. 3.a describes a speedometer: it is an ESTEREL *module*, receiving two *signals*, Second and Meter, which occur, respectively, whenever the vehicle has travelled for 1 meter and a second has elapsed. It emits a valued signal Speed, carrying the current value of the speed, an integer, measured in m/s. The *body* of the module is an infinite loop (lines 4–13) which initializes a local variable Distance — that will measure the number of Meters received within a Second — and enters a "do ... upto Second" construct (lines 6–10). This construct executes its body — a loop incrementing Distance on every occurrence of Meter (lines 7–9) —, until being interrupted by the next occurrence of the signal Second. So, on the first occurrence of Second following the entering in the global loop, the "do ... upto Second" statement is terminated and the counter Distance contains

```
1   module Speedometer:
2   input Second, Meter;
3   output Speed : integer in
4     loop
5       var Distance := 0 : integer in
6         do
7           every Meter do
8             Distance := Distance+1
9           end every
10        upto Second;
11        emit Speed(Distance)
12      end var
13    end loop
14  end module
```

(a)

```
1   module SpeedSupervisor:
2   input Second, Meter;
3   output TooFast in
4     signal Speed : integer in
5     [ run Speedometer
6     ||
7       every Speed do
8         if ?Speed > MaxSpeed
9         then emit TooFast
10        end if
11      end every
12    ]
13    end signal
14  end module
```

(b)

Fig. 3. A speed supervisor in ESTEREL

the number of Meters received during this time. The signal Speed, carrying the value of Distance, is simultaneously emitted (line 11), and the loop is entered again for a new Second. So, the signal Speed is emitted *exactly* each Second.

Fig. 3.b shows the speed supervisor, which makes use of the Speedometer module. Here the input signals are Second and Meter, and the output is a signal TooFast that is emitted whenever the speed excceeds the bound MaxSpeed. A local signal Speed is used to transmit the result of the speedometer. Within the scope of this signal, the speedometer is instanciated[7], through the run construct (line 5), in parallel with a process comparing the speed to the bound: this process is triggered whenever the speedometer emits a Speed signal, whose current carried value ?Speed is compared with the bound, with the possible effect of emitting the signal TooFast.

4 Compilation of Synchronous Languages

This section is an overview of the various approaches related to the compilation of synchronous languages into sequential or distributed code. Beforehand, we have to tackle a static semantic problem, which is specific to synchronous languages: *causality*.

4.1 Causality analysis

Generally speaking, the problem of causality comes from the fact that not all synchronous programs have a unique, deterministic meaning. In the data-flow model,

[7] In the run statement, the parameters of the Speedometer module could have been renamed.

this problem has a very simple statement, since it boils down to the well known problem of *combinational loops* in synchronous circuits [Mal93,Kau70,Sto92]: consider the following LUSTRE equations:

$$(a)\ \mathsf{x = not\ x} \qquad (b)\ \mathsf{y = y} \qquad (c)\ \mathsf{z = (z*z + 1.0)/2.0}$$
$$(d)\ \mathsf{u = if\ c\ then\ v\ else\ w;\ v = if\ c\ then\ w\ else\ u}$$

Case (a) is clearly a nonsense: the equation doesn't have any solution. Case (b) can be viewed as non deterministic, since y can have any value. Case (c) is an equation with one and only one solution (z=1), but solving such implicit algebraic equations is clearly unfeasible, in general. Case (d) can be disputed: apparently, u depends on v, that depends on u, so there is a combinational loop. Now, whatever be the value of the condition c, this loop is semantically cut.

In data-flow languages, all these situations are quite unnatural — because the user keeps the data dependences in mind — and generally easy to avoid. This is why, in LUSTRE, all the preceding examples are rejected by the compiler. However, in imperative languages like ESTEREL, ARGOS, or Statecharts, it is extremely easy to write programs with apparent causality problems — i.e., where, in some states, the presence of a signal seems to depend on itself —, to which users want to give meaning. More precise criteria must be applied to identify really problematic programs. Most of the various semantics that have been proposed for Statechart [vdB94] differ from each other by the way these problems are solved.

Let's go further into these problems, by means of some ESTEREL examples (see Fig. 4), taken from [Ber95]. A simple way of examining the correctness of these examples is by considering all the cases of presence/absence of signals: we want to have one (reactivity) and only one (determinism) consistent solution, for each configuration of input signals. For the module P1 of Fig. 4, either O is present, in which case the else part of the "present ... else ..." statement is not executed, so O is not emitted, and O is not present; this assumption is not consistent; or O is absent, the else part is executed, so O is emitted, and the assumption is again violated. This module doesn't have any consistent behavior. In fact, this example shows exactly the same kind of inconsistency as the equation "O = not O" in LUSTRE. Consider now the module P2 of Fig. 4. If we assume that O is present, the then part of the "present ... then ..." statement is executed, so O is emitted, and our assumption is satisfied. Now, assuming that O is absent, O is not emitted, and our assumption is satisfied again. Here, we have two consistent behaviors, it is a case of non determinism similar to the LUSTRE equation "O = O". The module P3 is a similar case of non determinism, showing that the problem can result from dependence paths of arbitrary length. For P4, if I is present, the first process in the parallel construct emits S, and the presence of S makes the second process to emit O. Conversely, if I is absent, neither S nor O is emitted. So P4 is correct; it corresponds to the LUSTRE fragment "S = I ; O = S", which doesn't show any loop. The case of P5 is more questionable: the first process in the parallel seems to be non deterministic (like in P2). Now, if we assume that O1 is present, we find that the second process in

```
module P1:
output O;
  present O
    else emit O
  end present
end module
```

```
module P2:
output O;
  present O
    then emit O
  end present
end module
```

```
module P3:
output O1, O2;
  present O1 then emit O2 end
||
  present O2 then emit O1 end
end module
```

```
module P4:
input I;
output O;
  signal S in
    present I then emit S end
  ||
    present S then emit O end
  end signal
end module
```

```
module P5:
output O1, O2;
  present O1 then emit O1 end
||
  present O1 then
    present O2 else emit O2 end
  end present
end module
```

Fig. 4. Causality problems in ESTEREL

the parallel doesn't have any behavior (like in P1). So, P5 has one and only one consistent behavior, where neither O1 nor O2 is emitted.

All the semantics proposed for synchronous languages reject modules P1, P2, and P3, and accept P4. The *Boolean causality* considered in [HM95], analyzes the problem in classical logic, and accepts also P5, since it has one and only one solution. The *constructive causality* [SBT96,Ber95] rejects P5, because the only solution doesn't have a constructive explanation, by means of causes and effects. Moreover, the constructive causality has been shown [SBT96] to coincide with *electric stability*: a constructively causal circuit will stabilize whatever be the traversal delays of its gates.

Causality problems are an obstacle to separate compilation and distributed code generation. Several authors[8] [Bou91,BdS96,Bon95] propose a weakenning of the synchronous communication, to get round these problems.

4.2 Sequential code generation

The straightforward way for compiling a data-flow synchronous language like LUSTRE into sequential code, is by generating a single loop, after conveniently sorting the equations according to their dependences. Sequential code generation from an imperative language like ESTEREL is less obvious: in the compilers ESTEREL-V2 and -V3 [BCG88,BG92], the control part of the program was compiled into an explicit automaton, representing the control structure of the code. This approach has also been applied to LUSTRE [CPHP87], with on-the fly minimization (by bisimulation) of the automaton [BFH+92,HRR91]. The explicit automaton is a very efficient implementation — since the whole internal synchronization of the program is computed at compile time —, with the drawback

[8] See also `http://www.inria.fr/meije/rc/rc-project.html`.

of possibly involving an exponential expansion of the code size. This is why it is now generally abandonned for single loop compilation. However, it played a central role in the development of verification tools.

The single loop code generation for ESTEREL was a side-effect of the development of a silicon compiler [Ber92] (see §4.5): compiling ESTEREL into circuits is mainly a translation into a data-flow network, which can be easily implemented by a single loop program. The ESTEREL-V4 and -V5 compilers [Ber95] are based on this principle.

About the compilation of synchronous imperative languages into data-flow networks, let us mention also the translation of ARGOS [MH96] into the DC common format (see §4.3) and the REGLO tool [Ray96] which produces recognizers (in LUSTRE) for regular expressions.

4.3 Common formats

The LUSTRE, SIGNAL, and ESTEREL compilers were developed in tight cooperation. In order to share common tools, and to make the languages integration easier, common formats were defined and used as intermediate codes in the compilers: the OC ("object code") format [PS87] was used to encode explicit automata in the earlier versions of the compilers. Another format [CS95], named DC/DC+ (for "declarative code") is used now to encode implicit automata, at the equational level.

4.4 Distributed code generation

Compiling synchronous languages into code for distributed architectures is obviously a challenge. In [ML94], techniques are proposed to separate a clocked data-flow networks into sub-networks independent enough to be sperately compiled into processes. The SYNDEX tool [LSSS91] can be used to schedule the resulting tasks on various architectures, and to study adjust the performances of the resulting system. Another approach is presented in [CGP94,CG95], which starts from the sequential code produced by the standard compilers, and from distribution directives given by the user; it consists of (1) replicating the code on each site of the architecture, (2) simplifying the code on each site according to its assigned role, and (3) adding communications along simple bounded FIFOs, to ensure both communication and synchronization.

4.5 Silicon compiling

Compiling synchronous programs into circuits is also an important goal, particularly in a codesign approach. Synchronous data-flow languages are very close to hardware description languages (HDLs), and their compilation to circuits is quite easy (see, e.g., [RH91]). The translation into circuits of ESTEREL was a much more difficult task, but the result is more interesting, since ESTEREL is of much higher level than usual HDLs for describing controllers. The translation proposed in [Ber92] is structural, and must be completed by deep optimizations, using both standard CAD tools [SSL+92], and specific techniques [STB96,STB97] that take advantage of knowledge coming from the structure of the source code.

5 Verification of Synchronous Programs

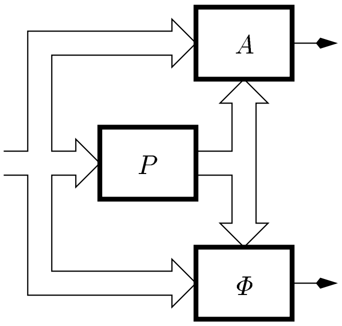
Since synchronous programming is mainly devoted to the field of critical embedded systems, the formal verification of synchronous programs is a particularly important goal. By chance, it can take advantage of some specific features of the application field and of the synchronous model:

- Experience shows that critical properties that must be verified are generally *simple*, *safety* properties. By "simple properties", we mean logical dependency relations between events, in contrast with deep arithmetic properties. As a consequence, these properties can often be model-checked on an abstraction of the program [Hal94], the most natural of which is the control automaton generated by the earlier versions of the compilers (see §4.2).
- The control automaton, being obtained as a synchronous product, is generally much smaller that models obtained by asynchronous composition (no interleaving, no need for partial orders, ...).
- The transition relation is a *vectorial function*, which allows particularly efficient BDD-based techniques [CBM89a] to be applied for the symbolic construction of the reachable state space.
- Thanks to the synchronous model, a specific approach can be applied to express safety properties by means of *synchronous observers*, i.e., special programs possibly written in the same language as the program under verification.

The verification methods for synchronous programs are all based on the control automaton.

Reduction methods have been applied [dSR94], mainly to automata compiled from ESTEREL: using the tools AUTO/AUTOGRAPH [RdS90] reduced views of the automaton can be obtained and compared.

Other methods are based on *model-checking* [QS82,CES86], and mainly symbolic model-checking [CBM89b,BCM+90]. TEMPEST [JPV95] is a model-checker of temporal logic formulas dedicated to ESTEREL. SIGALI [LDBL93] model-checks SIGNAL programs, using the symbolic resolution of SIGNAL *clocks* constraints.



The tool LESAR [HLR92] is a symbolic, BDD-based, model-checker of LUSTRE programs; it is based on the use of *synchronous observers*, to describe both the property to be checked and the assumptions on the program environment under which these properties are intended to hold: an observer of a safety property is a program, taking as inputs the inputs/outputs of the program under verification, and deciding (e.g., by emitting an *alarm* signal) at each instant whether the property is violated. Running in parallel the program, P, an observer Φ of the desired property, and an observer A of the assumption made about the environment one has just to check that either the alarm signal of Φ is never emitted (property satisfied) or the alarm signal of A is emitted (assumption violated), which can be done by a simple traversal of the reachable states of the compound program. Besides this only need of considering reachable states (instead of paths) this specification technique has several advantages:

- observers are written in the same language as the program under verification;
- observers are *executable*; they can be tested, or even kept in the actual implementation (redundancy, autotest).

Notice that, with an asynchronous language, observers would have to be explicitly synchronized with the program under verification, with the risk of changing its behavior. For an application of this technique, see also [WNT96].

Taking into account some numerical aspects — in particular delay counting — is considered in [HPR97], using abstract interpretation techniques [CC77]. In [LHR97], observers and abstract intepretation are used for verifying parameterized networks of synchronous processes.

6 Other Related Topics

[Le94] is an interesting comparative study of formal description techniques, including synchronous languages. This section lists some other works related to synchronous programming.

Program testing will remain a validation technique complementary to formal verification. The automatic testing of synchronous programs is studied in [MHMM95,TMC94,OP94,NRW98].

Integration of synchronous/asynchronous aspects: In general, only some parts of a complex system can be suitably described in the synchronous model. The language ELECTRE [CR95] is devoted to the synchronous control of asynchronous tasks. CRP [BSR93] allows ESTEREL modules to be composed asynchronously.

Combination of formalisms: Data-flow and imperative synchronous languages are complementary, and several attempts have been made, either to combine them [JLRM94], or to introduce imperative concepts in data-flow languages [RM95,MR98]. The most advanced work on combining synchronousa languages is probably the "SYNCHRONIE WORKBENCH[9]", developped at GMD.

Higher order data-flow languages: Data-flow synchronous languages can be viewed as lazy functional languages working on infinite sequence. "Lucid synchrone" [Cas93,CP95,CP96] is a higher order extension of LUSTRE, where the synchronous execution (bounded memory) is preserved.

[9] See `http://set.gmd.de/EES/synchronie/swb.html`.

Aknowledgements: I hope all prominent contributors to synchronous languages have been properly cited. However, the theses on the subject are not referenced, because most of them are written in French. Nevertheless, I would like to aknowledge the significant contribution of the following students[10]:

P. Amagbégnon, M. Belhadj, J.-L. Bergerand, R. Bernhard, C. Bodennec, F. Boniol, A. Bouali, B. Chéron, L. Cosserat, E. Coste-Maniere, P. Couronné, B. Dutertre, X. Fornari, D. Gaffé, G. Gherardi, A. Girault, G. Gonthier, A.-C. Glory, M. Jourdan, V. Lecompte, B. Le Goff, D. Lesens, D. L'Her, O. Maffĕis, H. Marchand, C. Mazuet, F. Mignard, J.-P. Paris, I. Parissis, J. Plaice, C. Ratel, P. Raymond, A. Ressouche, F. Rocheteau, V. Roy, J.-B. Saint, T. Shiple, J.-M. Tanzi, H. Toma, D. Weber

References

[ADA83] ADA. *The Programming Language* ADA *Reference Manual*. LNCS 155, Springer Verlag, 1983.

[AG96] C. André and D. Gaffé. Proving properties of GRAFCET with synchronous tools. In *IEEE-SMC'96, Computational Engineering in Systems Applications*, Lille, France, July 1996.

[And96] C. André. Representation and analysis of reactive behaviors: a synchronous approach. In *IEEE-SMC'96, Computational Engineering in Systems Applications*, Lille, France, July 1996.

[AW85] E. A. Ashcroft and W. W. Wadge. LUCID*, the data-flow programming language.* Academic Press, 1985.

[BB91] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.

[BCG88] G. Berry, P. Couronné, and G. Gonthier. Synchronous programming of reactive systems, an introduction to ESTEREL. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers.* Elsevier Science Publisher B.V. (North Holland), 1988. INRIA Report 647.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Fifth IEEE Symposium on Logic in Computer Science, Philadelphia*, 1990.

[BdS96] F. Boussinot and R. de Simone. The SL synchronous language. *IEEE Transactions on Software Engineering*, 22(4):256–266, April 1996.

[Ber89] G. Berry. Real time programming: Special purpose or general purpose languages. In *IFIP World Computer Congress*, San Francisco, 1989.

[Ber92] G. Berry. Esterel on hardware. *Philosophical Transactions Royal Society of London*, 339:217—248, 1992.

[Ber93] G. Berry. Preemption and concurrency. In *Proc. FSTTCS 93*, Lecture Notes in Computer Science 761, pages 72–93. Springer-Verlag, 1993.

[Ber95] G. Berry. The constructive semantics of esterel. Draft book available by ftp at `ftp://ftp-sop.inria.fr/meije/esterel/papers/constructiveness.ps.gz`, 1995.

[10] Of course, most of them are no longer students!

[Ber98] G. Berry. The foundations of Esterel. In C. Stirling G. Plotkin and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998.

[BFH+92] A. Bouajjani, J.-C. Fernandez, N. Halbwachs, P. Raymond, and C. Ratel. Minimal state graph generation. *Science of Computer Programming*, 18:247–269, 1992.

[BG92] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[BL90] A. Benveniste and P. LeGuernic. Hybrid dynamical systems theory and the SIGNAL language. *IEEE Transactions on Automatic Control*, 35(5):535–546, May 1990.

[Bli90] Blif-MV: An interchange format for design verification and synthesis. Technical report, Berkeley Logic Synthesis Group, 1990.

[Bon95] F. Boniol. Synchronous communicating reactive processes. In *2nd AMAST Workshop on Real-Time Systems*, Bordeaux, June 1995.

[Bou91] F. Boussinot. Reactive C: An extension of C to program reactive systems. *Software Practice and Experience*, 21(4):401–428, 1991.

[BS91] F. Boussinot and R. de Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9):1293–1304, September 1991.

[BSR93] G. Berry, R. K. Shyamasundar, and S. Ramesh. Communicating reactive processes. In *Proc. 20th ACM Conf. on Principles of Programming Languages, POPL'93*, Charleston, Virginia, 1993.

[Cas93] P. Caspi. Lucid synchrone. In *International Workshop on Principles of Parallel Computing (OPOPAC)*, November 1993.

[CBM89a] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using boolean functional vectors. In L. J. M. Claesen, editor, *Formal VLSI Correctness Verification*. North-Holland, November 1989.

[CBM89b] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble*. LNCS 407, Springer Verlag, 1989.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages, POPL'77*, Los Angeles, January 1977.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2), 1986.

[CG95] P. Caspi and A. Girault. Execution of reactive distributed systems. In *EURO-PAR'95 Stockholm*, volume 966 of *LNCS*. Springer Verlag, August 1995.

[CGP94] P. Caspi, A. Girault, and D. Pilaud. Distributing reactive systems. In *Seventh International Conference on Parallel and Distributed Computing Systems, PDCS'94*, Las Vegas, USA, October 1994. ISCA.

[CLM91] E. M. Clarke, D. E. Long, and K. L. McMillan. A language for compositional specification and verification of finite state hardware controllers. *Proceedings of the IEEE*, 79(9):1283–1292, September 1991.

[Coh96] N. H. Cohen. ADA *as a second langguage*. McGraw-Hill Series in Computer Science, 1996.

[CP95] P. Caspi and M. Pouzet. A functional extension to LUSTRE. In *Eighth International Symp. on Languages for Intensional Programming, ISLIP'95*, Sidney, May 1995.

[CP96] P. Caspi and M. Pouzet. Synchronous Kahn networks. In *Int. Conf. on Functional Programming, Philadelphia*. ACM SIGPLAN, May 1996.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE: a declarative language for programming synchronous systems. In *14th ACM Symposium on Principles of Programming Languages, POPL'87*, Munchen, January 1987.

[CR95] F. Cassez and O. Roux. Compilation of the ELECTRE reactive language into finite transition systems. *Theoretical Computer Science*, 144, June 1995.

[CS95] C2A-SYNCHRON. The common format of synchronous languages – The declarative code DC. Technical report, Eureka-SYNCHRON Project, October 1995.

[dSR94] R. de Simone and Annie Ressouche. Compositional semantics of ESTEREL and verification by compositional reductions. In *CAV'94*, Stanford, June 1994.

[Hal93] N. Halbwachs. *Synchronous programming of reactive systems.* Kluwer Academic Pub., 1993.

[Hal94] N. Halbwachs. About synchronous programming and abstract interpretation. In B. LeCharlier, editor, *International Symposium on Static Analysis, SAS'94*, Namur (Belgium), September 1994. LNCS 864, Springer Verlag.

[Har87] D. Harel. Statecharts: A visual approach to complex systems. *Science of Computer Programming*, 8(3), 1987.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HLR92] 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*, September 1992.

[HM95] N. Halbwachs and F. Maraninchi. On the symbolic analysis of combinational loops in circuits and synchronous programs. In *Euromicro'95*, Como (Italy), September 1995.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. In *Logic and Models of Concurrent Systems,* NATO *Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems.* Springer Verlag, 1985.

[HPR97] N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, August 1997.

[HRR91] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991. LNCS 528, Springer Verlag.

[IEE91] Another look at real-time programming. *Special Section of the Proceedings of the IEEE*, 79(9), September 1991.

[JLRM94] M. Jourdan, F. Lagnier, P. Raymond, and F. Maraninchi. A multiparadigm language for reactive systems. In *5th IEEE International Conference on Computer Languages*, Toulouse, May 1994. IEEE Computer Society Press.

[JPV95] L. J. Jagadeesan, C. Puchol, and J. E. Von Olnhausen. Safety property verification of ESTEREL programs and applications to telecommunication software. In P. Wolper, editor, *7th International Conference on Computer Aided Verification, CAV'95*, Liege (Belgium), July 1995. LNCS 939, Springer Verlag.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In *IFIP 74*. North Holland, 1974.

[Kau70] W. H. Kautz. The necessity of closed loops in minimal combinatorial circuits. *IEEE Trans. on Computers*, pages 162–164, 1970.

[LDBL93] M. Le Borgne, Bruno Dutertre, Albert Benveniste, and Paul Le Guernic. Dynamical systems over Galois fields. In *European Control Conference*, pages 2191–2196, Groningen, 1993.

[Le94] C. Lewerentz and Th. Lindner (eds.). *Case Study "Production Cell": a Comparative Study in Formal Software Development*. FZI-Publikation 940001, ISSN 0944-3037, Forschungszentrum Informatik, Karlsruhe, 1994.

[LGLL91] P. LeGuernic, T. Gautier, M. LeBorgne, and C. LeMaire. Programming real time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, September 1991.

[LHR97] D. Lesens, N. Halbwachs, and P. Raymond. Automatic verification of parameterized linear networks of processes. In *24th ACM Symposium on Principles of Programming Languages, POPL'97*, Paris, January 1997.

[LM93] P. LeParc and L. Marcé. Synchronous definition of Grafcet with Signal. In *IEEE SMC'93*, 1993.

[LSSS91] C. Lavarenne, O. Seghrouchni, Y. Sorel, and M. Sorine. The SynDEx software environment for real-time distributed systems design and implementation. In *European Control Conference, ECC'91*, July 1991.

[Mal93] S. Malik. Analysis of cyclic combinational circuits. In *ICCAD'93*, Santa Clara (Ca), 1993.

[Mar92] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR'92*, Stony Brook, August 1992. LNCS 630, Springer Verlag.

[MH96] F. Maraninchi and N. Halbwachs. Compiling Argos into boolean equations. In *Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT)*, Uppsala (Sweden), September 1996. LNCS 1135, Springer Verlag.

[MHMM95] M. Müllerburg, L. Holenderski, O. Maffeis, and M. Morley. Systematic testing and formal verification to validate reactive programs. *Softaware Quality Journal*, 4(4):287–307, 1995.

[Mil81] R. Milner. On relating synchrony and asynchrony. Technical Report CSR-75-80, Computer Science Dept., Edimburgh Univ., 1981.

[Mil83] R. Milner. Calculi for synchrony and asynchrony. *TCS*, 25(3), July 1983.

[ML94] O. Maffeis and P. Le Guernic. Distributed implementation of SIGNAL: scheduling and graph clustering. In *3rd Internationl Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*. LNCS 863, Springer Verlag, September 1994.

[MR98] F. Maraninchi and Y. Rémond. Mode-automata: About modes and states for reactive systems. In *European Symposium on Programming, ESOP'98*, Lisbon, April 1998.

[NRW98] X. Nicollin, P. Raymond, and D. Weber. Automatic testing of reactive programs. In preparation 1998.

[OP94] F. Ouabdesselam and I. Parissis. Testing synchronous critical software. In *5th International Symposium on Software Reliability Engineering (ISSRE'94)*, Monterey, USA, November1994.

[Per93] D. Perry. *VHDL*. McGraw-Hill, 1993.

[PP83] N.S. Prywes and A. Pnueli. Compilation of nonprocedural specifications into computer programs. *IEEE Transactions on Software Engineering*, SE-9(3), May 1983.

[PS87] J. A. Plaice and J-B. Saint. The LUSTRE-ESTEREL portable format. Unpublished report, INRIA, Sophia Antipolis, 1987.

[QS82] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*. LNCS 137, Springer Verlag, April 1982.

[Ray96] P. Raymond. Recognizing regular expressions by means of dataflows networks. In *23rd International Colloquium on Automata, Languages, and Programming, (ICALP'96)* Paderborn, Germany. LNCS 1099, Springer Verlag, July 1996.

[RdS90] V. Roy and R. de Simone. Auto and Autograph. In R. Kurshan, editor, *International Workshop on Computer Aided Verification*, Rutgers (N.J.), June 1990.

[RH91] F. Rocheteau and N. Halbwachs. Implementing reactive programs on circuits, a hardware implementation of LUSTRE. In REX *Workshop on Real-Time: Theory in Practice, DePlasmolen (Netherlands)*, pages 195–208. LNCS 600, Springer Verlag, June 1991.

[RM95] E. Rutten and F. Martinez. SIGNALGTI, implementing task preemption and time interval in the synchronous data-flow language SIGNAL. In *7th Euromicro Workshop on Real Time Systems*, Odense (Denmark), June 1995.

[SBT96] T. R. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *International Design and Testing Conference IDTC'96*, Paris, France, 1996.

[SSL+92] E. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Aldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. SIS: a system for sequential circuit synthesis. Technical report memorandum nr. ucb/erl m92/41, University of California at Berkeley, 1992.

[STB96] E. Sentovich, H. Toma, and G. Berry. Latch optimization in circuits generated from high-level descriptions. In *ICCAD'96*, November 1996.

[STB97] E. Sentovich, H. Toma, and G. Berry. Efficient latch optimization using incompatible sets. In *34th Design Automatio Conference*, June 1997.

[Sto92] L. Stok. False loops through resource sharing. In *ICCAD'92*, Santa Clara (Ca), 1992.

[TMC94] P. Thevenod-Fosse, C. Mazuet, and Y. Crouzet. On statistical testing of synchronous data flow programs. In *1st European Dependable Computing Conference (EDCC-1)*, pages 250–67, Berlin, Germany, 1994.

[vdB94] M. von der Beeck. A comparison of Statecharts variants. In *FTRTFT*. LNCS 863, Springer Verlag, 1994.

[WNT96] M. Westhead and S. Nadjm-Tehrani. Verification of embedded systems using synchronous observers. In *FTRTFT'96*, Uppsala, September 1996. LNCS 1135.