A synchronous language at work: the story of Lustre

Nicolas Halbwachs

A Synchronous Language at Work: the Story of Lustre

Nicolas Halbwachs
Vérimag,* Grenoble – France

Abstract

We recall the story of the development of the synchronous data-flow language LUSTRE *and of its industrial transfer inside the toolset* SCADE. *We try to analyse the reasons of its success, and to report the main lessons we got from the transfer of an academic concept into real industrial world.*

1 Introduction

The design of the synchronous language LUSTRE started more than 20 years ago, and resulted in an industrial software development tool, SCADE, which is now in use in many major companies developping embedded software (avionics, transportation, energy, ...). It seemed to us that this quite rare success story in the domain of formal methods deserves to be reported and analysed, from the point of view of the problems raised by the industrial transfer of a new technology: why did it succeed? how could it have succeeded better?.

So called "embedded systems" are much more fashionable nowadays than in the eighties, when they first appeared in large industrial applications. It is now admitted that this domain concerns systems presenting one or several of the following features: (1) they have to run in strong interaction with their — possibly physical — environment (real time systems, industrial control, ...), (2) their development combines software and hardware aspects, (3) they are submitted to strong non functional constraints (execution time, memory limitations, fault tolerance, power consumption, ...), (4) they are safety-critical — often because of (1), since they influence physical processes and devices. Because of (1) and (2), the domain of embedded systems is strongly related to both control theory and hardware design, and this is why the inspiration of synchronous languages comes both from control engineering formalisms and from hardware description languages.

One can wonder why the main three synchronous languages — Esterel [BG92], Signal [LGLL91] and LUSTRE [HCRP91] — were all born in France, approximately at the same time, and quite independently. Of course, at the beginning of the eighties, the idea of synchrony was already in the air, be it in theoretical works by Milner [Mil81, Mil83], or in "almost synchronous" formalisms, like Grafcet (or IEC 1131 Sequential Function Charts) [BJKS87, DA92] or Statecharts [Har87]. But the conditions were particularly favourable, both for academic and industrial reasons:

On the academic side, the three involved teams mixed researchers from control theory and computer science: Jean-Paul Rigault, Jean-Paul Marmorat and Gérard Berry for Esterel, Albert Benveniste and Paul Le Guernic for Signal, Paul Caspi and the author of this paper for LUSTRE. This double competence seems to have played an important role in the design of the languages.

On the other hand, strong industrial needs were appearing: the European and French industry of embedded software was faced with some big challenges:

- For the very first time, in the new family of French nuclear reactors, called N4, the most critical functions (in particular, the emergency stop) were realized by a computer system, the SPIN (for "integrated nuclear protection system").
- At the same time was designed the Airbus A320, which was the very first fully "fly-by-wire" aircraft.
- In railways industry, various automatic subways were designed (e.g., the VAL [Fer91]) and the successive versions of the French TGV (very high speed train) were more and more computerized.

Started in 1984, the development of LUSTRE beneficiated from these very good circumstances. After briefly recalling the principles of the language (Section 2), we will detail in Section 3 the main stages of its development, both from academic and industrial points of view. Section 4 analyses the feedback from industrial usages of the language. Finally, Section 5 outlines the current evolutions of the language and its associated tools.

*Verimag is a joint laboratory of Université Joseph Fourier, CNRS and INPG associated with IMAG.

2 A flavour of the language

Let us first recall, in a simplified way, the principles of LUSTRE: A LUSTRE program operates on *flows* of values. Any variable (or expression) x represents a flow, i.e., an infinite sequence $(x_0, x_1, \ldots, x_n, \ldots)$ of values. A program is intended to have a cyclic behavior, and x_n is the value of x at the nth cycle of the execution. A program computes output flows from input flows. Output (and possibly local) flows are defined by means of equations (in the mathematical sense), an equation "x=e" meaning "$\forall n, x_n = e_n$". So, an equation can be understood as a temporal invariant. LUSTRE operators operate globally on flows: for instance, "x+y" is the flow $(x_0 + y_0, x_1 + y_1, \ldots, x_n + y_n, \ldots)$. In addition to usual arithmetic, Boolean, conditional operators — extended pointwise to flows as just shown — we will consider only two temporal operators:

- the operator "pre" ("*previous*") gives access to the previous value of its argument: "pre(x)" is the flow $(nil, x_0, \ldots, x_{n-1}, \ldots)$, where the very first value "*nil*" is an undefined ("non initialized") value.
- the operator "->" ("followed by") is used to define initial values: "x -> y" is the flow $(x_0, y_1, \ldots, y_n, \ldots)$, initially equal to x, and then equal to y forever.

As a very simple and classical example, the program shown below is a counter of "events": It takes as inputs two Boolean flows "evt" (true whenever the counted "event" occurs), and "reset" (true whenever the counter should be reinitialized), and returns the number of occurrences of "events" which occured since the last "reset".

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

Intuitively, "true -> reset" is a Boolean flow, which is true at initial instant and whenever "reset" is true; when it is true, the value of "count" is 0; otherwise, when "event" is true, "count" is incremented, otherwise it keeps its previous value.

Once declared, such a "node" can be used anywhere in a program, as a user-defined operator. For instance, our counter can be used to generate an event "minute" every 60 "second", by counting "second" modulo 60 :

```
mod60 = Count(second, pre(mod60=59));
minute = (mod60 = 0);
```

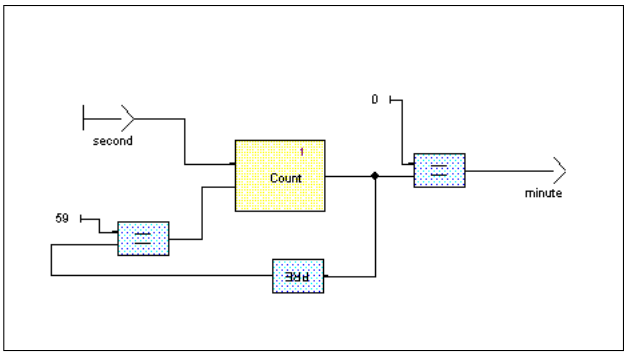


Figure 1. A graphical view in Scade

Here, "mod60" is the output of a "Count" node, counting "second", and reset whenever the previous value of "mod60" is 59. "minute" is true whenever "mod60" equals 0.

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.

The graphical counterpart of LUSTRE textual syntax is obvious; for instance, Fig. 1 is a a SCADE view of the "minute detector" described before.

3 The design and development of LUSTRE and SCADE

The initial idea of LUSTRE came from our previous works [CH86] about modelling real-time systems by means of time functions. Such a global handling of variable "histories", together with the inspirating proposal of the Lucid[1] language [AW85], suggested us to design a programming language describing variables as timed sequences of values. Moreover, from his background in control theory, Paul Caspi knew that this kind of description — we would say, today, "Matlab/Simulink-like" — was the natural one for control engineers: from their experience with previous technologies, they were used to declarative or data-flow formalisms, i.e., at high level, differential or finite-difference equations, and at lower levels, various kind of graphical networks (block-diagrams, analog networks, switches schemas, . . .).

[1] As a matter of fact, the name "LUSTRE" is a French acronym for "synchronous Lucid for real time".

3.1 The industrial story

Caspi's intuition was readily confirmed: we found, in many companies, many in-house development tools based on this kind of formalisms. The goal of such tools ranged from simple graphical description (without any idea of mechanical exploitation) to some attempts in consistency checking, simulation, and even automatic code generation. In particular, the Merlin-Gerin company (now Schneider-Electric), located in Grenoble, was in charge of the development of a large part of the SPIN ("integrated nuclear protection system"). Being aware that they were confronted to radically new problems, because of the software criticity, the management people decided to develop their own development environment, and they naturally chosed a data-flow formalism. Our great chance was to collaborate with them from the very beginning of the design of this environment, which was called SAGA (for "Assisted Specification and Automatic Generation"): SAGA was based on LUSTRE, provided with a mixed graphical/textual syntax, and offered a simple, but efficient, code generator. Two members of the LUSTRE team, Eric Pilaud and Jean-Louis Bergerand, were hired by Merlin-Gerin to supervise the development of the tool. The use of SAGA was very successful for the design of the SPIN and several other systems.

But after a few years, people at Merlin-Gerin understood that the maintenance and development of such a software toolset was not their job, so a software company, Verilog, was contacted for commercializing the SAGA product. Merlin-Gerin being already an important reference for SAGA, Verilog accepted the challenge: developping such a tool, with critical constraints of correctness, robustness, life duration, ..., was completely new for such a small company. Being located in Toulouse, Verilog contacted Aerospatiale (now part of Airbus), which was confronted with a very similar problem than Merlin-Gerin: for the design of the onboard software of the Airbus A320, Aerospatiale designed an in-house tool, called SAO (for "Computer Assisted Specification"), based on principles very similar to those of SAGA, but which was not intended, at the beginning, to perform the automatic generation of embeddable code. A kind of consortium was constituted between Aerospatiale, Merlin-Gerin, and Verilog to design a new tool, inspired by both SAO and SAGA. This new tool was called SCADE (for "Safety Critical Applications Development Environment"). At this time, VERIMAG was created as a common laboratory between Verilog and our academic institutions, in particular to make easier the cooperation about the design of SCADE. A member of the LUSTRE team, Daniel Pilaud, was hired by Verilog to head the SCADE team.

The new big challenge with SCADE, was to comply the requirements of the avionics certification authorities. In particular, the avionic norm DO-178B requires that any tool used for the development of a critical equipment be itself qualified at the same level of criticity than the considered equipment. As a consequence, for the code generated by SCADE to be embeddable, the SCADE code generator had to be qualified at the same level than the most critical software (flight software, level A). Let's say at once that such a qualification has nothing to do with a formal proof of the compiler, but is rather a matter of design process, test coverage, quality of the documentation, requirements traceability, etc. The SCADE code generator, KCG, is probably the first commercial compiler to be qualified for producing embedded software for civil avionics. KCG was a prominent argument for SCADE: not only, as any code generator, it suppresses the manual coding from data-flow specifications, but more importantly, since it is qualified, it suppresses the need for expensive *unit testing* for checking the correctness of this translation. This is what process people call the change from the "V" cycle to the "Y" cycle: the lowest part of the "V", which consist of coding and unit testing, becomes free and instantaneous.

In the nineties, the Swedish company Prover-Technology connected its SAT-based model checker PROVER with SCADE, thus providing the tool with an integrated verification capability. In doing so, they adopted our technique for specifying properties and assumptions by means of *synchronous observers*.

Then, the industrial story became complicated: Verilog was bought by the CS group, then sold to Telelogic, which finally sold SCADE to... Esterel-Technologies! Esterel-Technologies was founded in 1999, mainly to develop an industrial tool around the Esterel language. The main application domain of the Esterel language is circuit CAD, while the one of SCADE is embedded software, so the two languages do not compete with each other. They are quite different externally, but they share the same synchronous semantic model. So it makes sense to develop both tools together, and even to try to combine them (see §5). The purchase of SCADE by Esterel-Technologies was followed by a resumption of its development (in particular, a gateway from Simulink/Stateflow [CCM+03] was developed in the framework of the project IST-RISE), and a strong extension of its commercial promotion: SCADE is now used all over the world.

3.2 The research stages

The research about LUSTRE and its associated tools was driven in the very stimulating context of collaboration/competition with teams working on other synchronous languages, structured by successive French and European projects, called C2A, Eureka-SYNCHRON, Esprit/LTR-

SYRF[2], SafeAir-II[3].

3.2.1 Compilation

The first research topic, after the design of the language (Jean-Louis Bergerand's thesis), was of course its compilation. LUSTRE can be quite naturally and easily translated into sequential imperative code, as a single endless loop

```
initializations
loop
    acquire inputs ;
    compute outputs ;
    update memories
end
```

One just has to determine a correct order for computing outputs and memories, in order to minimize the number of memories (very often, there is no need of two memories to deal with x and pre(x)).

However, in the middle of the eighties, the only way of compiling ESTEREL was to produce an explicit automaton. We experienced a similar way of compiling LUSTRE (John Plaice's thesis). The idea was to specialize the code according to the known *previous* values of Boolean expressions: when "b" is true at some cycle, one knows that "pre(b)" will be true at the next cycle, and the code can be simplified accordingly. Of course, the initial cycle can also be specialized according to the semantics of the "->" operator. So, with each configuration of the Boolean memories (a state of the automaton) can be associated a specialized code (the outgoing transitions, carrying conditions and actions on non Boolean variables). In cooperation with the ESTEREL team, the common format OC was defined [PS87], as a target code for automata generators, in order to share common tools on this format, like generators to various target languages (C, Ada,...), minimizers, optimizers, graphical visualizers....

It readily appeared that, in contrast with ESTEREL, our generator produced automata with many redundant states, thus involving an explosion of the code even for simple programs. This is why we proposed the first algorithm of "symbolic bisimulation" [BFH90, HRR91], aiming at performing bisimulation reduction together with the generation of the automaton. This optimisation, implemented by means of BDDs technology, was the subject of Pascal Raymond's thesis.

Explicit automata can be much more efficient than the single loop code; however, it is generally also much bigger, and the size of the code is often a more important criterion than its efficiency. The size of the automaton may even be exponential w.r.t. the size of the source program, so the explicit automaton can rarely be used for real size programs.

[2]see www-verimag.imag.fr/SYNCHRONE/SYRF/syrf.html
[3]see www.safeair2.org/

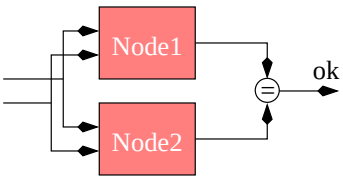


Figure 2. Comparing two nodes

However, the different ways of generating control automata were very useful for designing verification tools. Another influence from ESTEREL and the automaton generation was the introduction of *assertions* in the language: in ESTEREL, simple *relations* of exclusion or implication between input signals were introduced to allow the compiler to simplify the automaton accordingly. In LUSTRE, the relations were naturally generalized ("assert <*expression*>") to indicate the invariance of arbitrary Boolean expressions.

3.2.2 Specification and verification

LUSTRE program verification was the topic of two theses, by Anne-Cécile Glory and Christophe Ratel. As soon as we had a generator of control automata, we had also an automatic verification tool for Boolean programs. For instance, to verify that two nodes involving only Boolean variables and operators behave the same, one just has to connect them as in Fig. 2, compile the resulting program into an explicit automaton, and check on the resulting code that the output "ok" is never assigned false. This works also for comparing general nodes, but in a conservative way: if "ok" is never assigned false, the two nodes are equivalent, otherwise the verification is inconclusive. A more general verification scheme is given by Fig. 3: in general we want to prove that, as long as the environment behaves properly (i.e., satisfies some *assumption*), the program satisfies some *property*. Now, if we restrict ourselves to the verification of *safety properties*, both the assumption and the property can be expressed by some programs, called *synchronous observers* [HPOG89, HLR93], which receive as input all the relevant variables, and compute a single Boolean ouput, which is true as long as the inputs fulfill the considered property or assumption. The node to be verified is connected to the observers as in Fig. 3, and the verification consists of showing (by model-checking over the control automaton), that the output "prog_ok" can only be set to false if "env_ok" was set to false before. Of course, because of the state explosion, it is better to use specialised verification tools than the compiler. A specific model-checker, called LESAR, was developped for LUSTRE [HLR92]: it proceeds either by enumerative exploration of the automaton, or by symbolic forward or backward techniques. The model-checking techniques are classical, but the specifi-

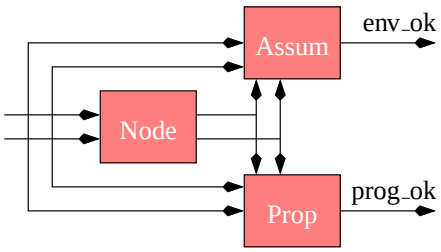


Figure 3. Specification with observers

cation by observers is especially natural in a synchronous declarative language. The same technique was adopted by Prover-technology when integrating its industrial verification tool with SCADE. We developped also a verification tool, called NBAC [HPR97, JHR99], based on linear relation analysis, and capable to handle simple (linear) behaviors of numerical variables (Bertrand Jeannet's thesis).

3.2.3 Automatic testing

The same technique of specification with observers was used to perform automatic testing [OP94, BORZ98, RWNH98, JRB04]: the assumption observer is used to generate only realistic test sequences, while the property is used as an "oracle" to determine whether each test passes or fails.

3.2.4 Hardware description and arrays

An experience was driven in 1989-92, in cooperation with the Paris Research Laboratory of Digital Equipment, to use LUSTRE for configuring large FPGAs, which were emerging at these times (Frédéric Rocheteau's thesis). For this, the language was extended with a mechanism of arrays (obviously needed to represent registers and regular architectures), which remained in the version 4 of the language [RH91].

3.2.5 Distributed code generation, dynamic scheduling

Code generation from synchronous programs to distributed architectures is a very challenging topic. We addressed the problem when the distribution is imposed by the user, e.g., by assigning a computation site with each variable. A first solution was proposed in Alain Girault's thesis [CGP99], which consists of (1) replicating the whole sequential code for each site, (2) slicing the code on each site according to the variables to be computed locally, and (3) adding communication and synchronization code, assuming a very simple communication mechanism (fixed size FIFOs). Another solution, inspired by control theory and industrial uses, was proposed in Rym Salem's thesis [CMSW99]: this proposal does not pretend to implement exactly the synchronous semantics of the centralized program, but takes advantage of the remaining non-determinism at the interface between synchronous and asynchronous worlds (mainly asynchronous input sampling). More recently, the problem of really multicycle programs was addressed: through the mechanism of *clocks*, LUSTRE allows variables to evolve at different rates; however, even if a variable is "slow", its computation must take place in the same "short" cycle than other variables. [SC04] proposes a way of generating truely multicycle code, where slow tasks are preempted by fast urgent ones, *while strictly respecting the synchronous semantics*.

4 Some lessons from industrial use

Let us now report the feedback we perceived from more than 10 years of industrial use of LUSTRE/SCADE. Some of our expectations were confirmed, some were not, unexpected qualities appeared, together with unexpected needs.

4.1 About our expectations

Synchronous data-flow. First, as said before, Caspi's initial idea about the convenience of the formalism for control engineer was completely confirmed. In particular, the graphical syntax was completely natural for SCADE users. This may look strange for programmers — who will generally prefer the textual LUSTRE syntax, and the capabilities of a text editor — but the graphical syntax is compulsory for reaching system designers; as a matter of fact, a graphical syntax, the SYNCCHARTS [And96], had also to be defined for ESTEREL.

Formal semantics. The fact that, in contrast with many control engineering formalisms, LUSTRE be equiped with a formal, simple, clean , and *abstract* semantics, is a more hidden advantage:

- it probably increases the quality of the programs, because it improves the quality of understanding of the language by the users; however, this impact is difficult to measure.
- it makes formal reasoning about programs easier, but it is not clear that formal reasoning is an important issue nowadays in the industry;
- however, it has a tremendous importance for the easiness of compilation, and for the quality of the generated code. Languages which — like Grafcet, Statecharts, or Simulink — are defined by means of a simulation algorithm, leave very little freedom for code generation and optimisation.

- our hope was that, using the formal semantics, a LUSTRE compiler could be formally *proven*. In spite of a successful attempt by Eduardo Gimenez with the proof assistant Coq [BC04], the industrial compiler was never proved. The main problem is probably that, for the time being, such a proof would not be accepted by certification authorities.

From clocks to activation conditions. In LUSTRE, the concept of clock is very similar to the one which plays a prominent role in SIGNAL: it allows a flow to have values only at some cycles, and then, to trigger operators only at these cycles. This quite abstract concept was seldom used by SCADE users, because it was considered too complicated. As a consequence, it was replaced by a simpler primitive, called "activation condition": a Boolean flow can be attached to a node call as an activation condition: the node is active only when the condition is true, otherwise its output keep their previous values (instead of being "absent" with the clock mechanism).

Formal verification and automatic testing. For the time being, the conclusion concerning formal verification is mitigated. We know of some experiences of property verification, using either the industrial Prover-plugin tool, or our academic prototypes. The technique of observers was apreciated, because it uses the same language for writing programs and properties. However, formal verification is far from being a routine task in software design. Even requiring that people express high level properties and assumptions is difficult. Moreover, it is difficult to quantify the benefits of using verification: nobody would admit that it improves the reliability of final programs ("anyway, my programs are zero-default!"), and concerning the reduction of costs, it is not yet admitted that it could reduce the amount of testing required for code certification. Our hope is to use automatic testing as a kind of "Trojan horse": since automatic testing is well-accepted (since it makes easier an already existing task), and since the effort needed for automatic testing (i.e., writing observers for the assumption and the property) is essentially the same than the one for verification, people can get used with verification tasks.

4.2 Unexpected features and needs

On the other hand, we discovered other advantages of LUSTRE/SCADE over existing tools, the importance of which was neglected before, mainly because they are classical in modern languages:

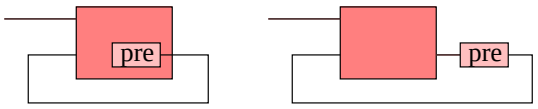
- Program structure: in contrast with many control engineering languages, LUSTRE proposes a structuration of programs at arbitrary depth, through the concept of node, which can be constructed by means of very few predefined operators. Tools like SAO only offered a partitionning of programs into *sheets*, the nesting of which was not possible; otherwise, the tool was based on a huge (and continuously increasing) library of predefined operators.
- Compiler efficiency: we expected the code efficiency to be the major gain with respect to previous tools. In fact, some users were also impressed by the efficiency of the compiler, some existing tools taking hours to compile even simple programs!
- Detection of instant loops: In LUSTRE, it is forbidden for a variable to instantly depend on itself (without a "pre" in the loop): it is callled a *causality error*. It was noticed that this detection by the compiler of instant loops may highlight specification inconsistencies which would be left hidden when programming with a sequential imperative language: in such a language, the order in which the statements are written just silently cut the loop, at some arbitrary point.

The question of causality: modular and separate compilation. As said before, synchronous languages raise the problem of causality errors (instantaneous self-dependence). The exact detection of causality errors is undecidable, since it can depend on arbitrary data properties. So it is necessarily approximated: some executable programs are rejected by the compiler. In LUSTRE, this approximation is especially rough, since the detection is done syntactically. However, it appeared not only that this rough detection was not disturbing for users, but that it was even too sophisticated: in SCADE, the criterion is even stronger[4], since it is forbidden that an input of a node instantaneously depend on an output of this node (i.e., not only there should be a "pre" in the loop, but it should appear outside the node, see Fig 4). This simplification has a huge consequence: nodes can be compiled separately (in contrast with all other synchronous languages), since the correct order of computations inside the node cannot depend on the way it is called. Of course, separate compilation is an important topic for real-life applications. It is also important for code traceability, which is often required for certification: the code generated for a node, or a node call, is clearly identified as a contiguous block in the whole object code.

5 And now ...

Let's conclude by an overview of ongoing research and development around LUSTRE and SCADE.

[4] according to some compiling options, which are used for separate compilation and code traceability. Other options allow the same criterion than in LUSTRE.



Allowed in LUSTRE
Forbidden in SCADE

Allowed in LUSTRE
Allowed in SCADE

Figure 4. Causality in LUSTRE and SCADE

First, some strong evolutions of the language are still going on. These evolutions are often prototyped thanks to Marc Pouzet's "Lucid synchrone" [CP95, CP98] which is a higher order extension of LUSTRE (or a kind of merge of LUSTRE and ML):

- *Arrays:* The array mechanism introduced in LUSTRE-V4 [RH91] aimed at describing regular circuits. It appeared that it is not convenient for software implementation, because V4 arrays cannot be compiled into arrays (updated with loops) in the target code: presently, they are expanded into as many variables as array elements. In [Mor02], a new mechanism of arrays is proposed, provided with a small number of *iterators*, similar to those of functional programming. Experiments show that this new mechanism is powerful enough for most pratical applications, and can lead, of course to tremendous reduction of the code size. Moreover, this mechanism permits also an original technique [MM04] for generic verification of programs containing arrays.

- *State machines:* It is known for long that the data-flow style is sometimes inconvenient in some situations, where the system to be described is naturally sequential, or automaton-like. Several attempts have been made to mix imperative and data-flow descriptionss [JLRM94, MR98]. Presently, a weakenned version of the SYNCCHARTS, called "Safe State Machines" (SSM), is being introduced into SCADE.

- *Packaging and genericity:* LUSTRE is still quite poor concerning such classical topics as encapsulation, packaging, genericity,..., which are essential for large designs, libraries definition and reuse. We are currently introducing mechanisms for defining generic packages, encapsulating definitions of constants, types, functions and nodes, and possibly taking as static parameters constants, types functions and nodes. This could lead, some day, to an "object oriented LUSTRE".

The new ways of compiling LUSTRE towards distributed architectures, and event-trigered tasking (cf. §3.2.5) remain to be transferred in the industrial tool. Another topic about code generation is intra-instant scheduling: in some applications, the basic cycle is considered too long with respect to some input-output required delays. In such cases, one would like to influence the static scheduler inside the compiler, so that some input acquisitions be scheduled "close to" some output computations and emissions. Such a feature is possible in the SAXO compiler [WBC+00] for ESTEREL. It is under investigation for LUSTRE [CCM+03].

Finally, we are also using LUSTRE for modelling, simulating, and verifying non synchronous systems: For instance, so called "globally asynchronous, locally synchronous" systems (GALS) can be modelled [HB02] by introducing controlled non-determinism through the use of additional inputs (oracles), possibly restricted by assertions. Some experiences are driven also concerning the simulation of circuits descriptions at transaction level in SystemC.

References

[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.

[BC04] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series, Springer Verlag, 2004.

[BFH90] A. Bouajjani, J.-C. Fernandez, and N. Halbwachs. Minimal model generation. In R. Kurshan, editor, *International Workshop on Computer Aided Verification*, Rutgers (N.J.), June 1990.

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

[BJKS87] A.D. Baker, T.L. Johnson, D.I. Kerpelman, and H.A. Sutherland. Grafcet and SFC as factory automation standards. In *American Control Conference*, pages 1725–1730, 1987.

[BORZ98] L. du 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.

[CCM+03] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From simulink to scade/lustre to tta: A layered approach for distributed embedded applications. In *LCTES 2003*, San Diego, CA, June 2003.

[CGP99] P. Caspi, A. Girault, and D. Pilaud. Automatic distribution of reactive systems for asynchronous networks of processors. *IEEE Transactions on Software Engineering*, 25(3):416–427, 1999. Research report INRIA 3491.

[CH86] P. Caspi and N. Halbwachs. A functional model for describing and reasoning about time behaviour of computing systems. *Acta Informatica*, 22:595–697, 1986.

[CMSW99] P. Caspi, C. Mazuet, R. Salem, and D. Weber. Formal design of distributed control systems with Lustre. In *Proc. Safecomp'99*, volume 1698 of *Lecture Notes in Computer Science*. Springer Verlag, September 1999.

[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.

[CP98] Paul Caspi and Marc Pouzet. A Co-iterative Characterization of Synchronous Stream Functions. In *Coalgebraic Methods in Computer Science (CMCS'98)*, Electronic Notes in Theoretical Computer Science, 28-29 March 1998.

[DA92] R. David and H. Alla. *Petri nets and Grafcet: tools for modelling discrete event systems*. Prentice Hall, New York, 1992.

[Fer91] D. Ferbeck. The VAL product line. In *APM'91 Conference*, Yokohama, 1991.

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

[HB02] N. Halbwachs and S. Baghdadi. Synchronous modeling of asynchronous systems. In *EMSOFT'02*. LNCS 2491, Springer Verlag, October 2002.

[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*, pages 785–793, September 1992.

[HLR93] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[HPOG89] N. Halbwachs, D. Pilaud, F. Ouabdesselam, and A.C. Glory. Specifying, programming and verifying real-time systems, using a synchronous declarative language. In *Workshop on Automatic Verification Methods for Finite State Systems, Grenoble*. LNCS 407, Springer Verlag, June 1989.

[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.

[JHR99] B. Jeannet, N. Halbwachs, and P. Raymond. Dynamic partitioning in analyses of numerical properties. In A. Cortesi and G. Filé, editors, *Static Analysis Symposium, SAS'99*, Venice (Italy), September 1999. LNCS 1694, Springer Verlag.

[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.

[JRB04] E. Jahier, P. Raymond, and P. Baufreton. Case studies with lurette v2. In *First International Symposium on Leveraging Applications of Formal Method, ISoLa 2004*, Paphos, Cyprus, October 2004.

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

[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.

[MM04] F. Maraninchi and L. Morel. Arrays and contracts for the specification and analysis of regular systems. In *Fourth International Conference on Application of Concurrency to System Design (ACSD)*, Hamilton, Ontario, Canada, June 2004.

[Mor02] L. Morel. Efficient compilation of array iterators for Lustre. In *First Workshop on Synchronous Languages, Applications, and Programming, SLAP02*, Grenoble, April 2002.

[MR98] F. Maraninchi and Y. Rémond. Mode-automata: About modes and states for reactive systems. In *European Symposium On Programming*, Lisbon (Portugal), March 1998. Springer Verlag.

[OP94] F. Ouabdesselam and I. Parissis. Testing synchronous critical software. In *5th International Symposium on Software Reliability Engineering (ISSRE'94)*, Monterey, USA, November 1994.

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

[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.

[RWNH98] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[SC04] N. Scaife and P. Caspi. Integrating model-based design and preemptive scheduling in mixed time- and event-triggered systems. In *Euromicro conference on Real-Time Systems (ECRTS'04)*, Catania, Italy, June 2004.

[WBC+00] D. Weil, V. Bertin, E. Closse, M. Poisse, P. Venier, and J. Pulou. Efficient compilation of Esterel for real-time embedded systems. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, San Jose, 2000.