Synchronous Parallelism and Languages: an Introduction

Marc Pouzet

UPMC/ENS/INRIA
Marc.Pouzet@ens.fr

MPRI, September 13, 2016

Trends for building safe and complex software

Interdule: Parallelism and time sharing

The synchronous model of time

The origin of Lustre

Several languages: Lustre, Esterel, Lucid Synchrone, ReactiveML

Trends for building safe and complex software

Write executable mathematical specifications in a high-level language so that a model is:

A reference semantics independent of any implementation.

A basis for simulation, testing, formal verification.

Then compiled into executable code, sequential or parallel.

A way to achieve correct-by-construction software.

Typed Functional Languages

Program in a mathematical language as an attempt to achieve code with zero defect.

High-level languages abstracting some details to focus on *what* computes a system.

A computation is a sequence of reductions:

$$fact(3) \to 3 \times fact(2) \to 3 \times 2 \times fact(1) \to 3 \times 2 \times 1 \to 3 \times 2 \to 6$$

Follow a few principles:

- Function composition.
- Types as specifications/properties of these functions.
- A method to check that a function agrees with its type.

Typed Functional Languages

A rich collection of languages:

- Lazy languages restricted to *pure* functions: Haskell, etc.
- Strict languages: Objective Caml, StandardML, etc.
- Proof assistants to write total functions.

An **important vehicule of ideas** for other languages and the use of formal methods in industry (Esterel-Tech., Microsoft, etc.)

These are **general purpose** programming languages for sequential algorithms and implementations.

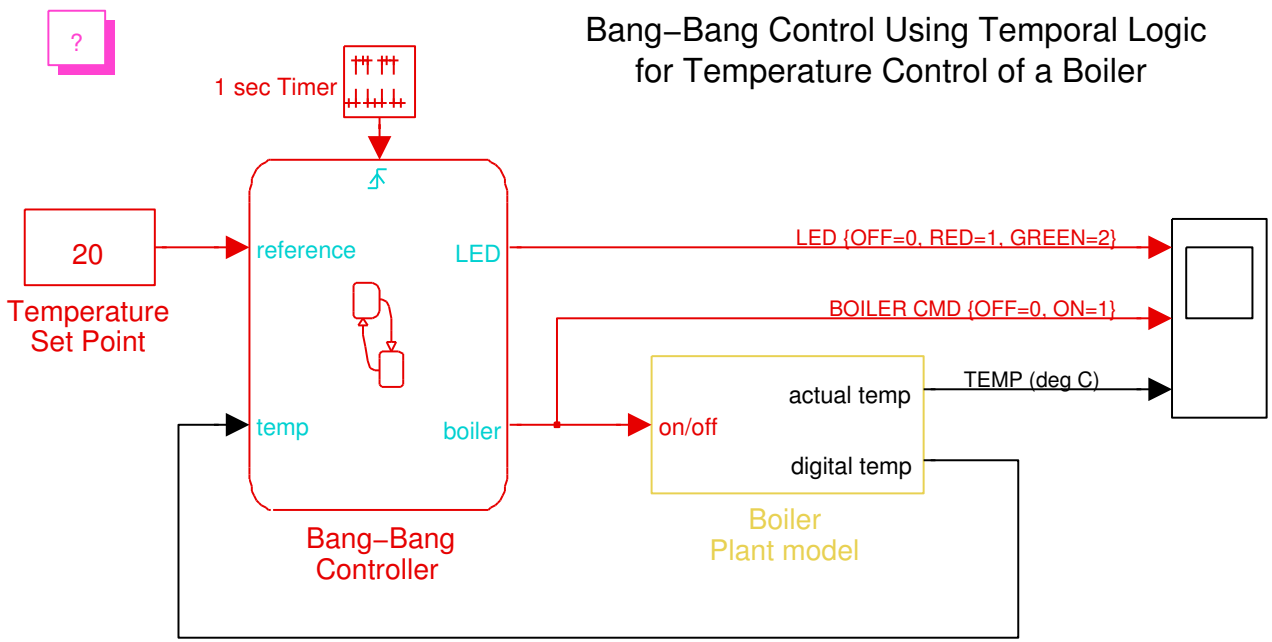
What about systems combining parallelism and time?

Real-time Systems

Focus on systems which continuously interact with each others.

- With a **physical environment** (e.g., fly-by-wire command, control-engine);
- or **other digital devices** (e.g., phone, TV boxes).

Example: a Bang-bang controller



Real-time Systems

Real time is always **related to the environment** and is not an absolute notion.

To ensure safety, think of **"what is the worst case"**?

The environment is not precisely known: systems run in **closed-loop**.

How can we program those systems, focusing on the **functionality** and abstracting from subtle implementation details?

What is new, why do we need mathematical languages?

Conciliate three notions:

- A **formal** (and computable) model of time.
  - Express deadlines, simultaneous events, etc.
- **parallelism** to describe complex systems from simpler ones
  - Control *at the same time* rolling and pitching.
  - *Closed-loop* systems (the controller and the plant run in parallel).
- **Statically guaranty safety properties** (both functional and non functional).
  - Determinism, dead-lock freedom.
  - Execution in bounded time and memory.

Safety is important:

- critical systems: fly-by-wire, braking, airbags, etc.
- some systems do not have a stable position (plane?)
  - properties must be guaranteed statically: **"dynamic" = "too late"**

Interlude: Parallelism and time sharing

- It is possible to launch several processes on a computer even if it has a single processor...???
- The computer fundamentaly does a single thing at a time.
- It gives the impression of doing several because it goes faster than our perception of change.
- Things are easy when applications are independent from each others.

Is this parallelism useful for programming a real-time system?

Read: ["The problem with threads", Edward Lee, 2006]

Let us draw a swing in OCaml...

```
let swing center radius alpha_init speed =
 let alpha = ref alpha_init in
 while true do
   alpha := !alpha +. speed;
   dessine center radius !alpha;
 done


let main = (swing c1 r pi) speed
```

(See OCaml code on the web page.)

Two swings?

Put two in parallel.

```
let swing center radius alpha_init speed =
 let alpha = ref alpha_init in
 while true do
   alpha := !alpha +. speed;
   dessine center radius !alpha;
 done

let main =
 Thread.create (swing c1 r pi) speed;
 Thread.create (swing c2 r 0.) speed
```

It does not work: why?

- The OS scheduler decides to execute some steps of the first then some of the second, then some of the first, etc.
- What happen if an extra swing is added?
- The programmer has no simple mean to control the scheduling policy of the OS such that swing move at the same pace.
- No precise time synchronization.

Synchrony/Asynchrony

Can we fix this by giving a hint to the scheduler so that it switches from a task to the other?

```
let swing center radius alpha_init speed =
 let alpha = ref alpha_init in
 while true do
   alpha := !alpha +. speed;
   dessine center radius !alpha;
   Thread.yield ()
 done

let main =
 Thread.create (swing c1 r pi) speed;
 Thread.create (swing c2 r 0.) speed
```

Synchrony/Asynchrony

Parallelism by interleaving

- `swing1; swing1; swing1; swing2;...`
- `swing1; swing2; swing1; swing2;...`
- Non reproducible behavior (depend on the scheduler and number of running processes).
- Add explicit synchronizations: expensive (at run-time), subtle (dead-locks, starvation), obfuscate the program.

This is even worst on multi-core processors

The semantics by interleaving is wrong: putting two threads running on a shared memory machine exhibits extra behaviors. [1]

[1] Cf. work by Zappa Nardelli [9].

Synchrony/Asynchrony

Add extra code to explicitely synchronise processes and ensure that every process do a single step.

```
let swing center radius alpha_init speed m1 m2 =
 let alpha = ref alpha_init in
 while true do
   alpha := move !alpha;
   draw center radius !alpha;
   Mutex.unlock m2; Mutex.lock m1
done

let main =
  let m1, m2 = Mutex.create (), Mutex.create () in
  Mutex.lock m1;
  Mutex.lock m2;
  Thread.create (swing c1 r a1) speed m1 m2;
  Thread.create (swing c2 r a2) speed m2 m1
```

Synchrony/Asynchrony

A solution with a synchronization barrier.

```
let barriere n =
  let mutex, attente =
    Mutex.create (), Mutex.create () in
  Mutex.lock attente;
  let nb_att = ref 0 in
  fun () ->
    Mutex.lock mutex;
    incr nb_att;
    if !nb_att = n then begin
      for i = 1 to n-1 do Mutex.unlock attente done;
      nb_att := 0; Mutex.unlock mutex
    end else begin
    Mutex.unlock mutex; Mutex.lock attente
end
```

Synchrony/Asynchrony

```
let stop = barriere 3

let swing center radius alpha_init speed =
  let alpha = ref alpha_init in
  while true do
    alpha := move !alpha;
    draw center radius !alpha;
    stop ()
done
let main =
  Thread.create (swing c1 r a1) speed;
  Thread.create (swing c2 r a2) speed;
  Thread.create (swing c3 r a3) speed
```

Not modular; error-prone (deadlock/starvation) and inefficient when systems are tightly coupled (at lot of synchronization between them).

Design domain specific languages (Berry'89 [4])

What's the matter with general purpose sequential languages and concurrency from the OS?

Sequential Programming:

- Turing complete, too much expressiveness, too hard to verify.
- Complexity is not where it is needed: pointer arithmetic, dynamic allocation, etc.
- Parallelism while ensuring determinism is absent whereas it is fundamental.

Concurrent (asynchronous) programming

- Time is not taken into account, both in language and in the semantics, non determinism is unavoidable:
  E.g., `wait 2 second; wait 3 second` $\neq$ `wait 5 second`
- No precise synchronization, no instantaneous broadcast:
  E.g., `wait 60 second; send minute to B? (send minute to B; send minute to C)?`
- Parallel composition and rendez-vous communication compose poorly: adding an extra listening process changes the semantics.
- Observing/debugging a program may change its behavior.

What if time was made logical?

Suppose that the machine is infinitetely fast...

Do as if there is an global time scale, shared by all processes. Do not try to go as fast as possible but synchronise/agree on it.

The swing in ReactiveML [8]

```
let process swing center radius alpha_init speed =
 let alpha = ref alpha_init in
 while true do
   alpha := !alpha +. speed;
   dessine center radius !alpha;
   pause
 done

let process main =
    run (swing c1 r pi speed)
 || run (swing c2 r 0. speed)
```

Synchony

It is possible to add as many swings as we want without modifying the overall behavior.

```
let process main =
    run (swing c1 r a1 speed)
 || run (swing c2 r a2 speed)
 || run (swing c2 r a3 speed)
```

The Synchronous Model of Time

A global logical time that is shared by all processes. This define instants on which processes can synchronize.

A sequence of ticks. Instruction pause means: wait for the next tick.

A bit of History

In the 80's, several team invented (at about the same time) languages dedicated to the design/implementation of control-systems.

- **Lustre** (Caspi & Halbwachs, Grenoble): data-flow (block-diagrams), functional model (deterministic);
- **Signal** (Benveniste & Le Guernic, Rennes): data-flow but relational (to define also non-deterministic systems);
- **Esterel** (Berry & Gonthier, Sophia): hierarchical automata and process algebra

Base it on the **mathematical culture and models** of the field of embedded control-systems.

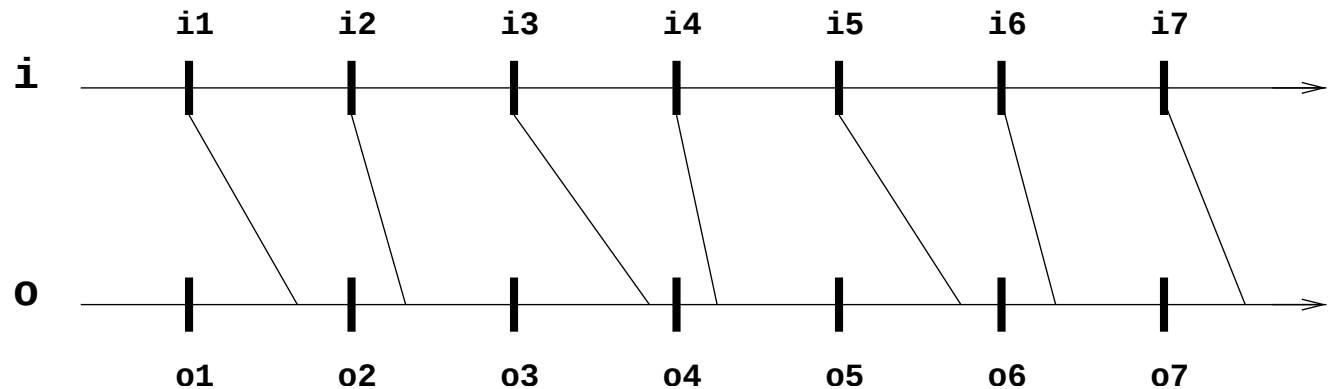
Quite a successful story: security systems in nuclear plants (Schneider Electric), fly-by-wire (Airbus, Ambraier, etc), automotive, trains, etc.

The Synchronous Model of Time

Simplify the programming of real-time system by considering first that exact time can be neglected, i.e, What is the worst case?

Time becomes logical: it is a sequence of instantaneous reactions.

1. Do as if the machine is infinitely fast: all processes listen to each others and see the very same inputs. E.g., the radio.
2. Check a posteriori the correspondance between logical time and real-time: is the machine fast enough? What is the worst case?



Worst case execution time (WCET): $max_{n \in \mathbb{N}}(t_n - t_{n-1}) \leq bound$.
This is checked on the actual platform.

Is it that original?

The conductor

All musician share a global time scale, that of the conductor.

Dancers:

They synchonise on music. This is the way several dancers agree and do the same thing.

Synchronous circuits:

A global clock shared by all gates/registers.

That is:

First reason ideally, neglecting light speed (orchestra), sound (dancers), electricity (circuits).

Then measure actual computational time and transmission delays.
Verify that the synchronous abstraction is reasonnable.

Trends for building safe and complex software

Interdule: Parallelism and time sharing

The synchronous model of time

The origin of Lustre

Several languages: Lustre, Esterel, Lucid Synchrone, ReactiveML

The Language LUCID

Ashcroft and Wadge introduced the language LUCID to program by writting (un-ordered) equations.

Programming Languages

J.J. Horning* Editor

Lucid, a Nonprocedural Language with Iteration

E.A. Ashcroft
Unversity of Waterloo
W.W. Wadge
University of Warwick

Lucid is a formal system in which programs can be written and proofs of programs carried out. The proofs are particularly easy to follow and straightforward to produce because the statements in a Lucid program are simply axioms from which the proof proceeds by (almost) conventional logical reasoning, with the help of a few axioms and rules of inference for the special Lucid functions. As a programming language, Lucid is unconventional because, among other things, the order of statements is irrelevant and assignment statements are equations. Nevertheless, Lucid programs need not look much different than iterative programs in a conventional structured programming language using assignment and conditional statements and loops.

Key Words and Phrases: program proving, formal systems, semantics, iteration, structured programming
CR Categories: 5.21, 5.24

Introduction

There has been much work done recently on techniques of program proving, but nevertheless most programmers still make little if any effort to verify their programs formally. Perhaps the main obstacle is the fact that most programming languages are not "mathematical" despite their use of some mathematical notation. This means that in proving a program it is necessary either to translate the program into mathematical notation (e.g. into the relational calculus) or to treat the program as a static object to which mathematical assertions are attached. In either case, the language in which assertions and proofs are expressed is different (often radically different) from the language in which programs are written. It can be argued that one of the best proof methods, that of Hoare [3], is successful partly because the programming language statements are brought into the proof language. We want to take this process to its logical conclusion.

Our aim is to unify the two languages with a single formal system called Lucid in which programs can be written and proofs carried out. A Lucid program can be thought of as a collection of commands describing an algorithm in terms of assignments and loops; but at the same time Lucid is a strictly denotational language, and the statements of a Lucid program can be interpreted as true mathematical assertions about the results and effects of the program. For example, an assignment statement in Lucid can be considered as a statement of identity, an equation. A correctness proof of a Lucid program proceeds directly from the program text, the statements of the program being the axioms from which the properties of the program are derived, the rules of inference being basically those of first-order logic with quantifiers. Furthermore, in Lucid we are not restricted to proving only partial correctness or only ter-

Kahn Process Networks

A network of deterministic processes communicating through FIFOs defines a continuous stream function.

INFORMATION PROCESSING 74 – NORTH-HOLLAND PUBLISHING COMPANY (1974)

THE SEMANTICS OF A SIMPLE LANGUAGE FOR PARALLEL PROGRAMMING

Gilles KAHN

IRIA-Laboria, Domaine de Voluceau, 78150 Rocquencourt, France

and

Commissariat à l'Energie Atomique, France

In this paper, we describe a simple language for parallel programming. Its semantics is studied thoroughly. The desirable properties of this language and its deficiencies are exhibited by this theoretical study. Basic results on parallel program schemata are given. We hope in this way to make a case for a more formal (i.e. mathematical) approach to the design of languages for systems programming and the design of operating systems.

Block diagram formalisms (control-theory, signal process.)

Difference equations, causal equations on sequences, z-transform:[2]

Time is discrete and logical (indices in $\mathbb{N}$)

Equation $o = x + y$ means $\forall n \in \mathbb{N}. o(n) = x(n) + y(n)$

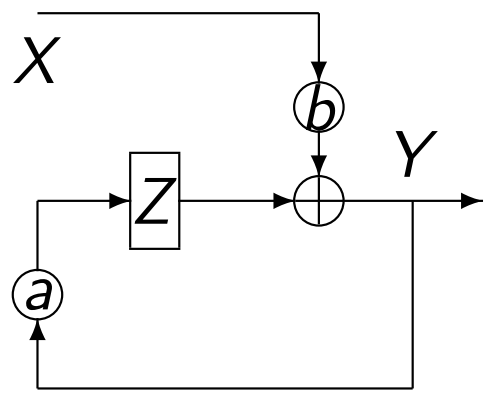
Equation $o = \frac{1}{z}(x)$ means $o(0) = 0$ and $\forall n > 0. o(n) = x(n-1)$

Manual transcription of these equations into imperative code.

Hard and error-prone.

Example: linear filtering (IIR)

$$Y_0 = bX_0 \ , \ \forall n \ Y_{n+1} = aY_n + bX_{n+1}$$



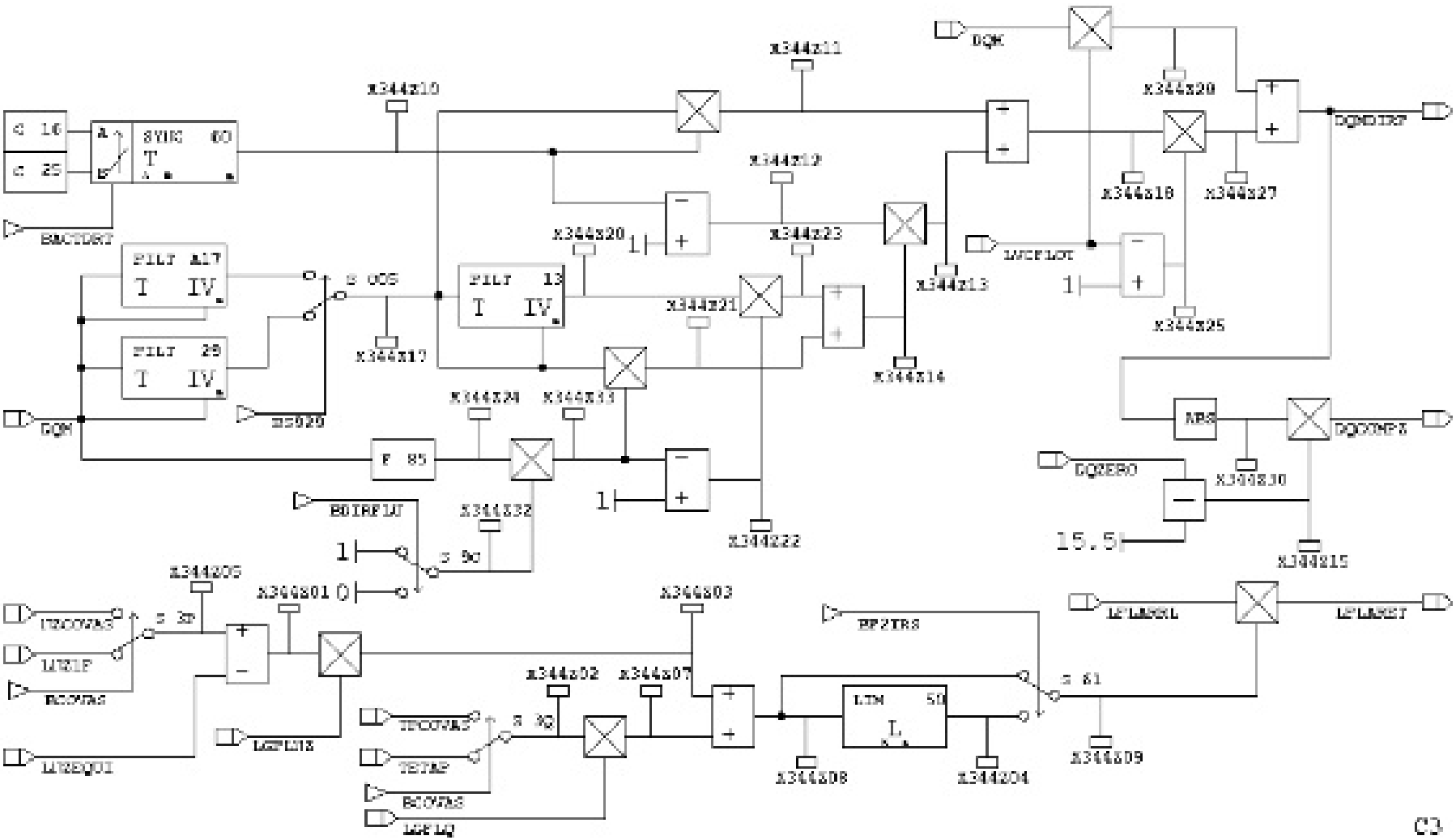
[2] Cf. Seminar by Juliette Leblond at CdF, Feb. 2014.

The beautiful idea of Lustre: Caspi & Halbwachs

- Write stream equations as **executable specifications**.
- A safety property is a program that observe an other program, called a **synchronous observer**.
- Restrict the expressiveness to ensure execution in **bounded time and memory**.
- Provide **static analysis/verification** tools and a compiler.
- The generated code is **correct-by-construction**

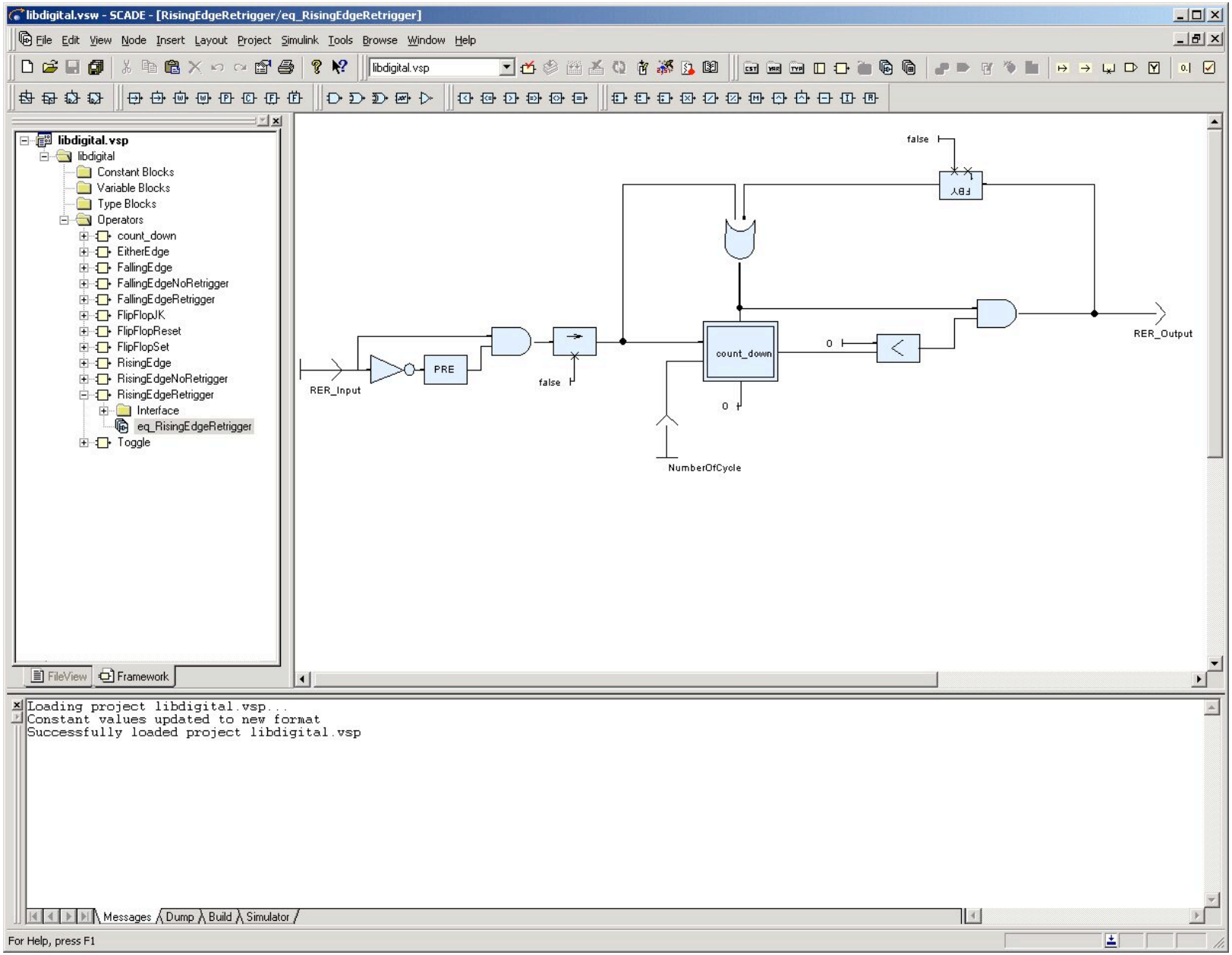
From drawings...

SAO (Spécification Assistée par Ordinateur) — Airbus 80's



... to programs: SCADE V5

Safety Critical Application Development Env (Esterel-Technologies)



Trends for building safe and complex software

Interdule: Parallelism and time sharing

The synchronous model of time

The origin of Lustre

Several languages: Lustre, Esterel, Lucid Synchrone, ReactiveML

An introduction to Lustre

An introduction to Esterel

- All synchronous languages share the same interpretation of time.
- One can be translated into the other.
- Several extensions has been considered: Lucid Synchrone, ReactiveC, ReactiveML.
- SCADE 6 (defined and implented by Esterel-Technologies) incorporates programming construts inspired by both that of Lustre, Lucid Synchrone and Esterel.
- Novel extensions are currently investigated, in particular the mix of discrete and continuous time.


E.A. Ashcroft and W.W. Wadge.
Lucid, a non procedural language with iteration.
Communications of the ACM, 20(7):519–526, 1977.


A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone.
The synchronous languages 12 years later.
Proceedings of the IEEE, 91(1), January 2003.


A. Benveniste, P. LeGuernic, and Ch. Jacquemot.
Synchronous programming with events and relations: the SIGNAL language and its semantics.
Science of Computer Programming, 16:103–149, 1991.


G. Berry.
Real time programming: Special purpose or general purpose languages.
Information Processing, 89:11–17, 1989.


G. Berry and G. Gonthier.

The Esterel synchronous programming language, design, semantics, implementation.
Science of Computer Programming, 19(2):87–152, 1992.


P. Caspi, N. Halbwachs, D. Pilaud, and J. Plaice.
Lustre: a declarative language for programming synchronous systems.
In *14th ACM Symposium on Principles of Programming Languages*. ACM, 1987.


Gilles Kahn.
The semantics of a simple language for parallel programming.
In *IFIP 74 Congress*. North Holland, Amsterdam, 1974.


Louis Mandel and Marc Pouzet.
ReactiveML, a Reactive Extension to ML.
In *ACM International Conference on Principles and Practice of Declarative Programming (PPDP)*, Lisboa, July 2005.
Recipient of the price for the “most influential PPDP’05 paper” given in July 2015 at PPDP’15.


Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen.
x86-tso: a rigorous and usable programmer's model for x86 multiprocessors.
Commun. ACM, 53(7):89–97, 2010.