A Brief History of Synchronous Programming

Marc Pouzet
DIENS
Equipe INRIA PARKAS [1]

PSL/ENS
Marc.Pouzet@ens.fr

SYNCHRON
28 nov. 2022

[1] https://parkas.di.ens.fr

A French researcher go to a conference...


Central Authentication Service
LOGIN


flygskam!


flygskam!

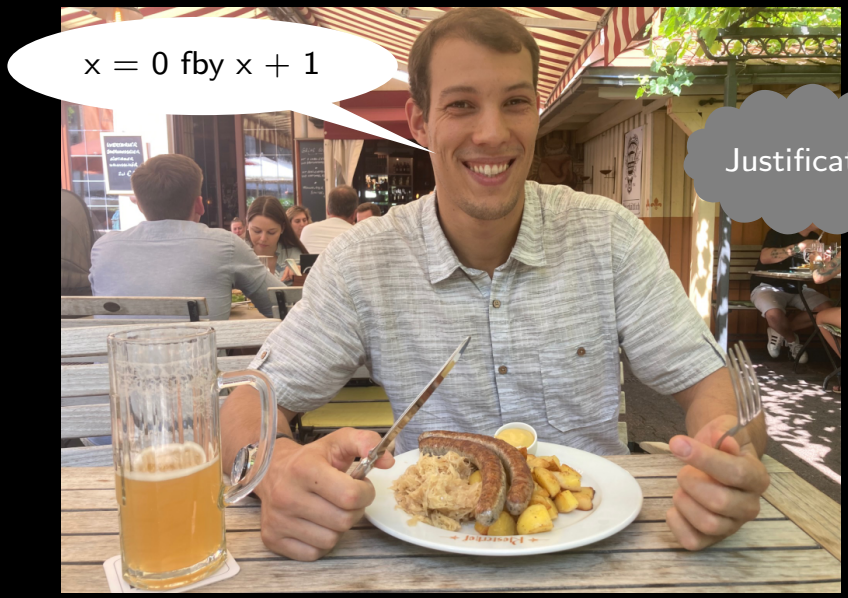

flygskam!


x = 0 fby x + 1


flygskam!


x = 0 fby x + 1
Justificatifs!


flygskam!


x = 0 fby x + 1
Justificatifs!

How to specify, program, verify the software of this small printer?

What is different from other programs?

Physical components, e.g., sensors, actuators, stepper motor (paper roll), etc.

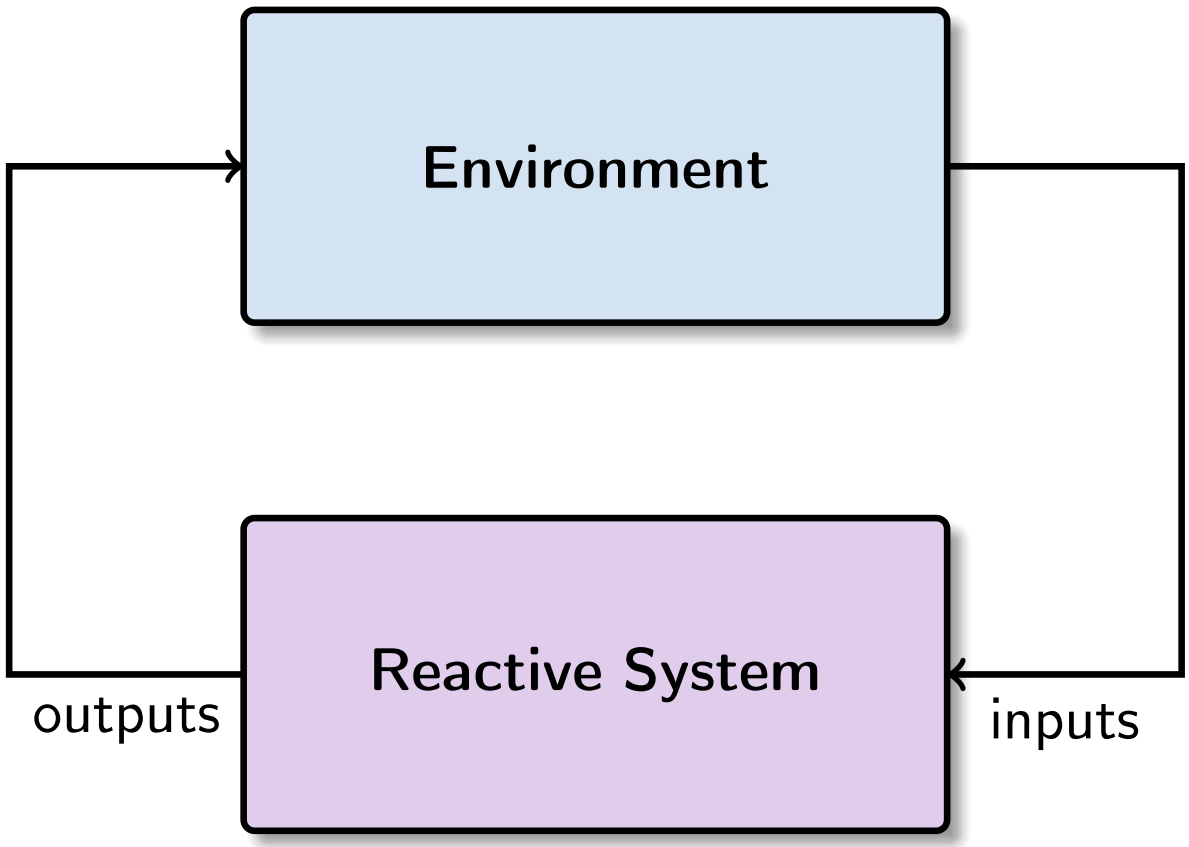
Control permanently (temperature), fast enough, regularly (the motor), etc.

More widely...

fly-by-wire command of a plane, on-board control of a train, the management of enery in a car, etc.

They are all embedded reactive systems.

What is it all about?



Write Assembly/C/C++/JavaScript/Python/OCaml/Coq by hand?

And compare it to what?

And compare it to what?

What is the specification?

And if we have a formal specification,

And if we have a formal specification,

in what programming language to write it?

And if we have a formal specification,

in what programming language to write it?

how to make sure the product code is safe and correct? E.g.,

And if we have a formal specification,

in what programming language to write it?

how to make sure the product code is safe and correct? E.g.,

there is no run-time error;

And if we have a formal specification,

in what programming language to write it?

how to make sure the product code is safe and correct? E.g.,

there is no run-time error;

the memory used is bounded and known statically;

And if we have a formal specification,

in what programming language to write it?

how to make sure the product code is safe and correct? E.g.,

there is no run-time error;

the memory used is bounded and known statically;

the worst-case execution time is bounded and known statically;

And if we have a formal specification,

in what programming language to write it?

how to make sure the product code is safe and correct? E.g.,

there is no run-time error;

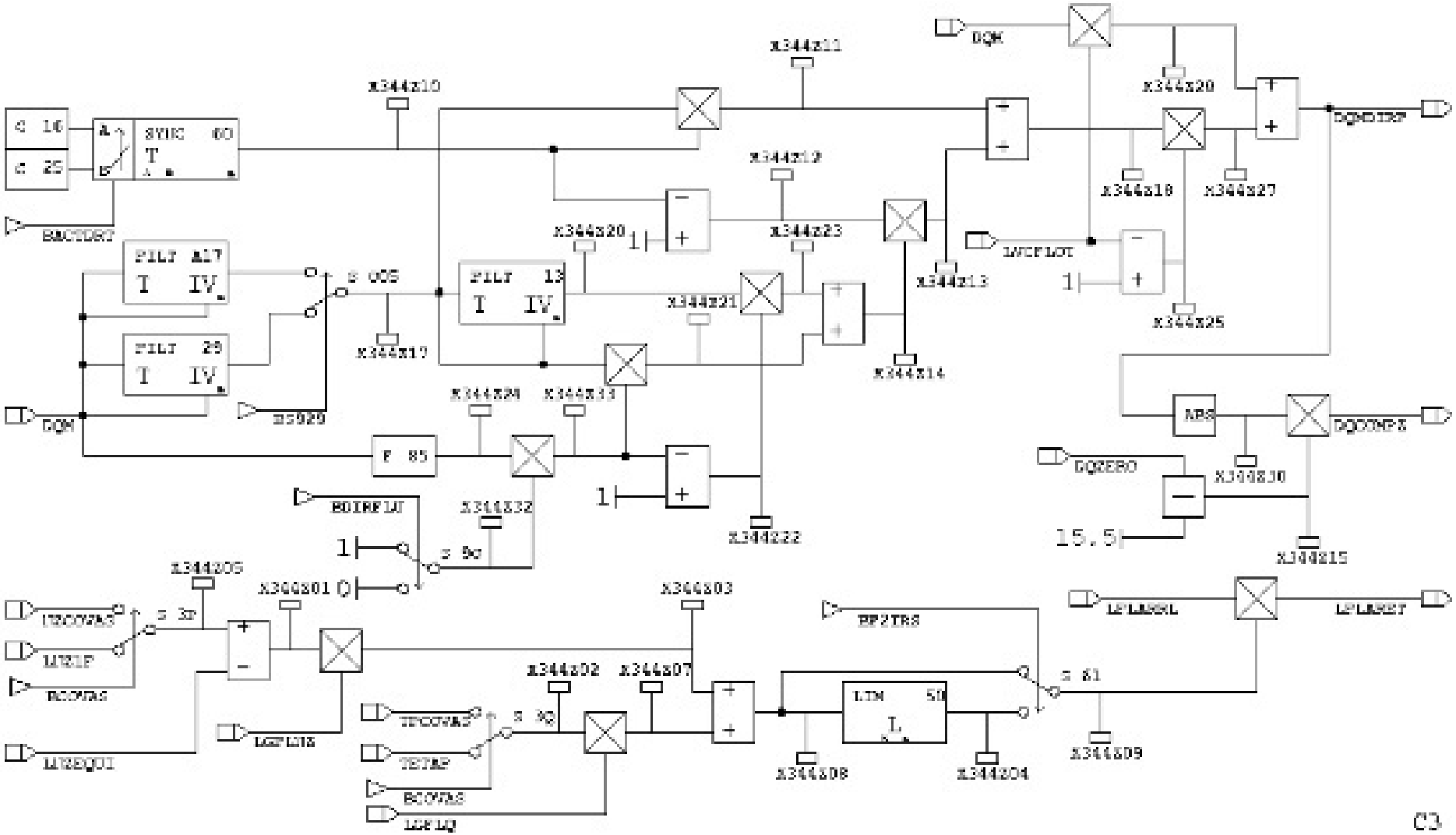
the memory used is bounded and known statically;

the worst-case execution time is bounded and known statically;

the executed code faithfully implements the specification.

A language for writing executable math specifications?

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



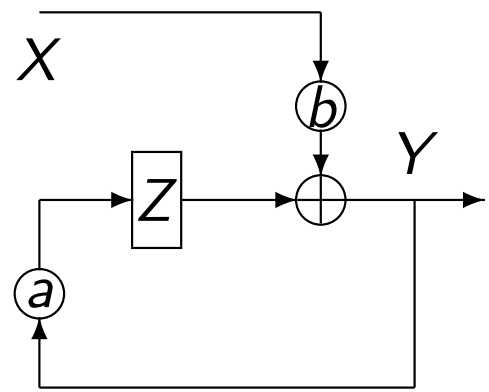
Accurate drawings

Engineers in control/signal processing had presise maths before the use of computers.

Sampled systems: stream equations, state machines, etc.

Example: a linear filter

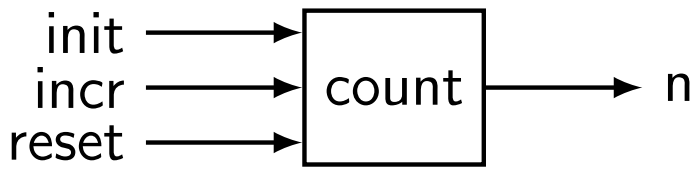
$$Y_0 = bX_0 \ , \ \forall n \ Y_{n+1} = aY_n + bX_{n+1}$$



...but not executable! Write code and convince that it is correct.

How to make those maths executable?

Somewhere in Grenoble... the language Lustre (1987) [CHPP87]




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

Programming with stream equations

A discrete system: a function of sequences; the sequences are synchronous.

X	1	2	1	4	5	6	...
Y	2	4	2	1	1	2	...
$X + Y$	3	6	3	5	6	8	...
pre X	*nil*	1	2	1	4	5	...
Y -> X	2	2	1	4	5	6	...

Equation $Z = 0$ -> *pre* $Z + X + Y$ means $Z_0 = 0 \wedge Z_n = Z_{n-1} + X_n * Y_n$.

Time is logical: inputs X and Y arrive "at the same time"; the output Z is produced "at the same time"

Euh... is-it real-time?

Reason in worst case: check that the generated code produces the output before the next input arrives.

The beautiful idea of Lustre

A synchronous interpretation of the Kahn and MacQueen networks [Kah74, KM77] and Lucid [AW85].

Express directly sampled models from control engineering.

Analyze/transform/simulate/test/verify them.

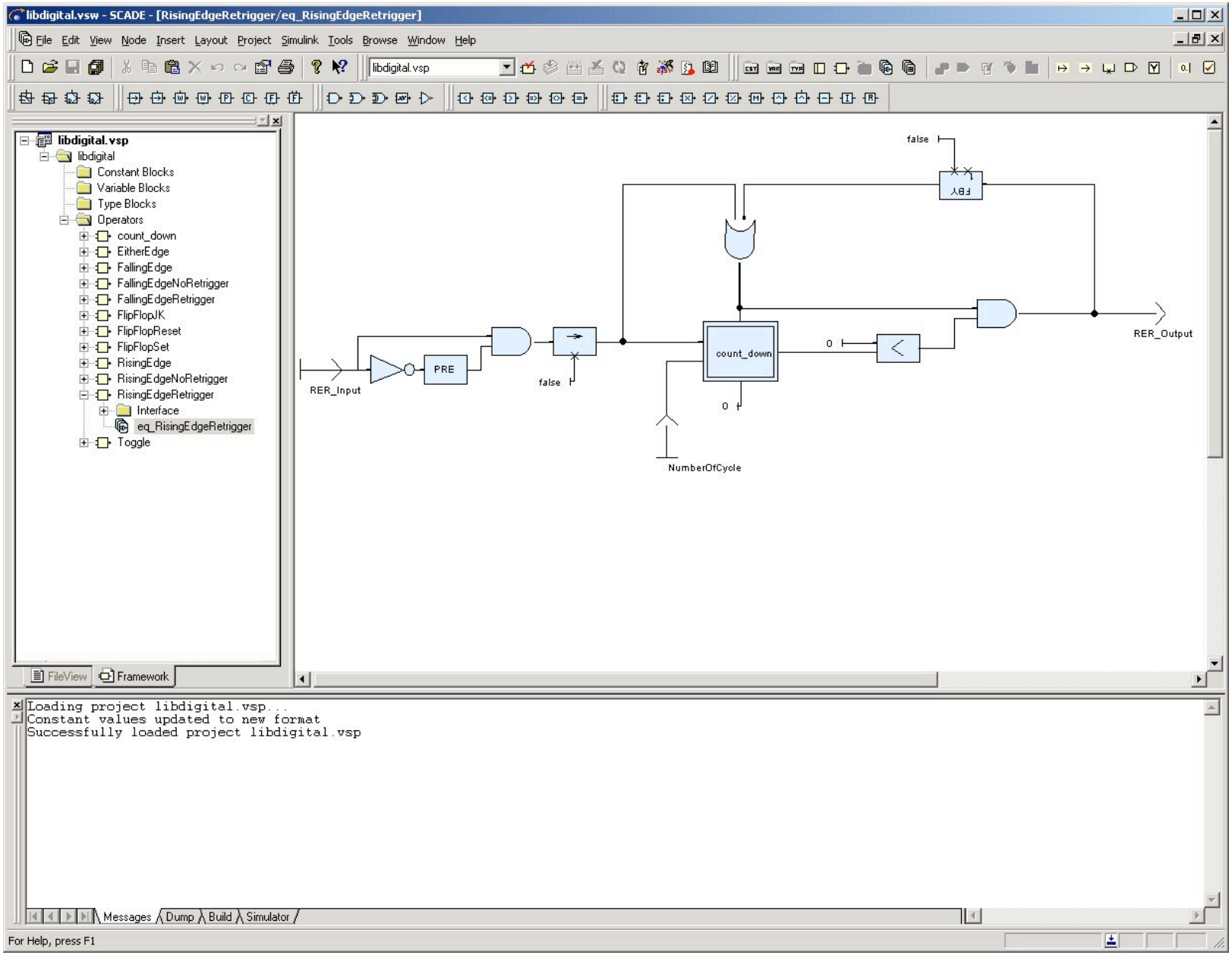
Automatically translate them into executable code.

A Lustre program is a precise specification.

An immediate resonance with industrial practice:

- SAO (Spécification Assistée par Ordinateur) - Airbus.
- Saga - Merlin Gerin

SCADE: Safety Critical Application Dev. Env. (Verilog, 95)



At the same time...

In Rennes: the Signal language (1987) [BLJ91]

Same influence and approach as Lustre but much more expressive.

A system: a relation between sequences.

Write specifications (partial, non-deterministic) of a system.

A Signal program is used to specify the interface of a component.

Study refinement relations.

Sildex industrial tool: based on Signal (TNI-Software then DS).

À Nice: the Esterel language (1984) [Gon88, BG92]

"Control" dominated systems (e.g., Mealy machines) and their transcription into sequential circuits.

Theoretical computer science (process calculus, lambda calculus, semantics).

Several radical ideas:

- A programming style closer to computer science: sequence, loops, interrupt, suspension of task, parallel composition, hierarchy, etc. to increase expressiveness.
- Several semantics, in SOS form.
- How to make this portable and deterministic?

Composition of boolean automata (e.g., Argos) [Mar92]

The beautiful idea of Esterel

Do as if the machine were computing infinitely fast!

Reconcile parallelism (for expressiveness) and determinism (for safety) [2].

A new discovery: compile Esterel into circuits (Lustre).

```
module ABRO
input A, B, R;
output C
loop
   [await A || await B];
   emit C
each R;
end module
```

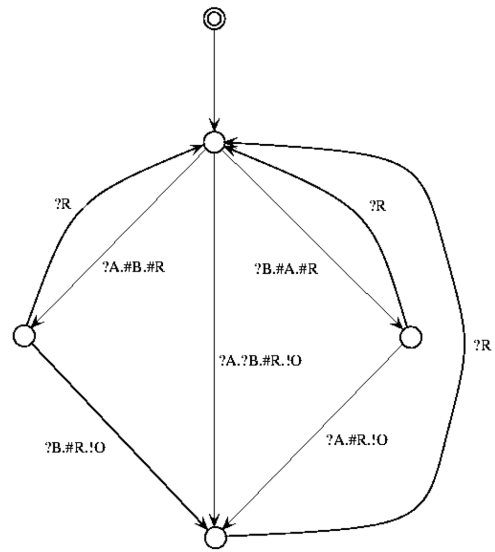


Figure 3.1: The ABRO Mealy machine

Industrial tool: Esterel-Studio then start-up Esterel-Technologies (1999).

[2] "Write things once" (The Esterel Language Primer Version5.91, G. Berry, 2000).

The same synchronous approach

With very different programming styles, all these languages share the same principles

(1) reason ideally;

(2) Compile the parallelism and compute the WCET of the generated code.

It's much easier.

Some programs are monsters...

how to reject them?

Different ́time scales

Synchronize slow and fast processes?

X	1	2	3	4	5	...
half	*true*	*false*	*true*	*false*	*true*	...
X *when half*	1		3		5	...
$X + (X$ *when half*$)$	2		5		8	...

```
let half = true -> not (pre half);
    o = x + (x when half);
tel
```

Define the sequence: $\forall n \in \mathbb{N}.o_n = x_n + x_{2n}$

- cannot be implemented with bounded memory (buffer);
- reject it statically: we can do it by typing [CP96, CP03].

Analyze dependences between signals

Programs have zero solutions (*deadlock*) or too many (*non-determinism*)

In Lustre/Signal

- `x = y + 1 and y = x + 2`
- `y = x and x = y`

For Lustre, a simple choice: "Syntactic causality" [CHPP87]:

"every loop must cross a delay", i.e., partial order between calculations.

This analysis can be done in a modular way, by typing and independent of clock calculus [CP01, BBC$^+$14]. And we can compile in independent blocks [PR09].

For Signal, conditional dependencies: "x depends on y if c".

Detect "true" cycles. Combines the computation of clocks and causality; analysis and code generation are more complex [ABG95].

Analyze dependences between signals

In Esterel

- `present S else emit S`
- `present S1 then emit S2 || present S2 then emit S1`
- ```
  present I then
    present O2 then emit O1 else present O1 then emit O2
  ```

Two discoveries [Draft book'02] [3]

Constructive semantics by calculating a fixed-point at each instant [Gon88, Ber02].

Are there any undetermined signals?

For Esterel, the "good" notion of causality is that of the électricity.

"If we wire the synchronous program, are the outputs stable?

Coincides with what is provable in constructive logic [MSB12]

[3] The Constructive Semantics of Pure Esterel Draft Version 3, 2002, G. Berry.
  ```

A bit later, somewhere between Grenoble and Jussieu...

Lucid Synchrone and ReactiveML

An idea of Paul Caspi, in 1994, in Grenoble.

> *"Marc, take a good look, we can write recursive Lustre programs in a few lines of LazyML!"*

Very expressive: higher order, type inference, recursion, etc.

but the "monsters" are still there.

typing, typing, typing... and adapt the compilation.

Lucid Synchrone (95-06) [CP96, Pou06]

Build a functional synchronous language with ML features.

ReactiveML (05-15) [MP05]

Synchronous parallelism in an ML-like language (OCaml)

Boussinot model [Bou91]: delayed reaction to absence.

Functional and synchronous: quesaco? [4]

Operations on streams in a language *a la ML*.

A synchronous interpretation [CP96, CP98] of Kahn & MacQueen's networks.

Make it a laboratory to experiment with new ideas.

New programming constructs: pattern matching, signals, last, automata, higher-order, etc.

Express the static semantics in term of dedicated type systems: clock calculus [CP96, CP03], causality analysis [CP01, BBC$^+$14], initialization analysis [CP04],etc.

A semantics by translation/collapse: being able to explain the high-level constructions in terms of clocked data-flow equations with reset.

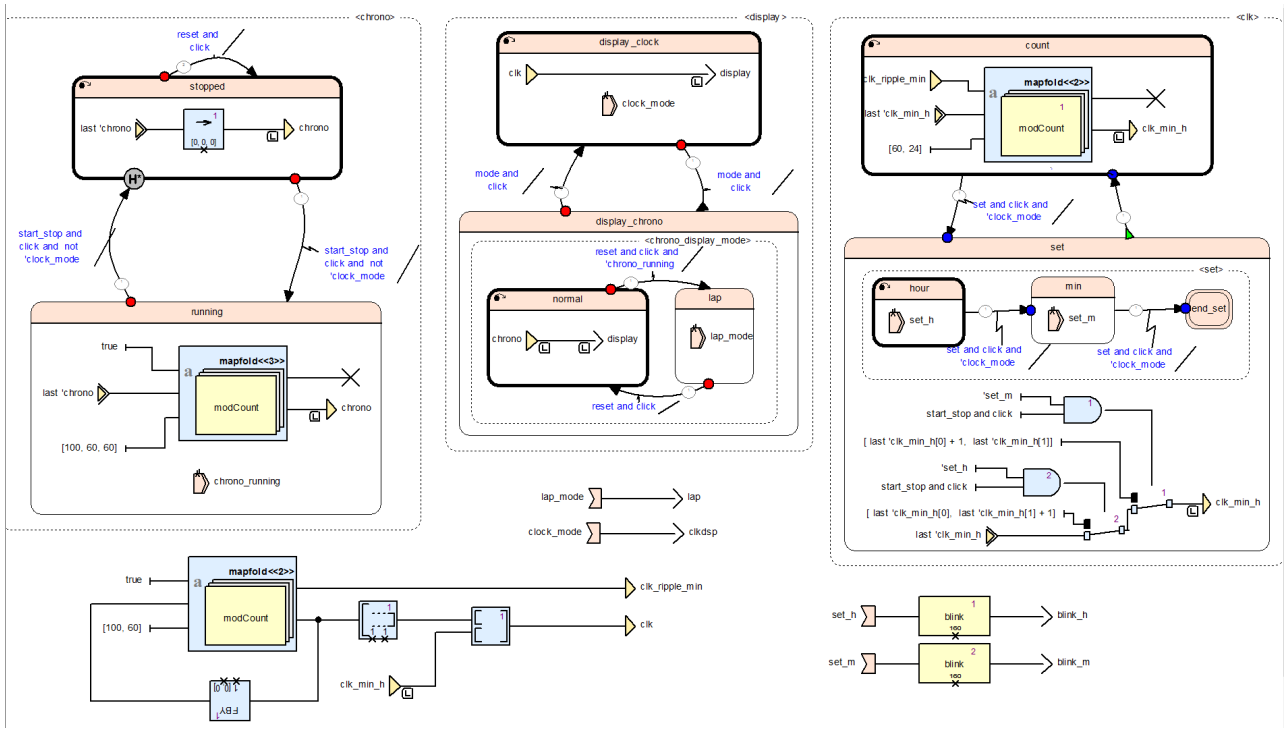
To ensure that the proposed new constructs are conservative.

A collaboration with Jean-Louis Colaço (Esterel-Tech./ANSYS).

[4] What is this?

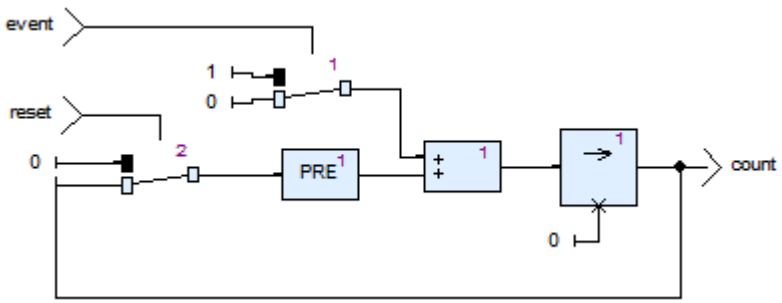
Scade/KCG 6 [CPP17]

- A new language and compiler, in 2008; written in OCaml;
- many ideas from Lucid Synchrone.



Are your drawings correct? Prove it!

‘What you prove is what you execute’ (Berry ’89)




```
L1 = pre L7;
L2 = (L11) -> (L5);
L3 = event;
L4 = reset;
count = L2;
L5 = L6 + L1;
L6 = if L3 then (L8) else (L9);
L7 = if L4 then (L10) else (L2);
L8 = 1;
L9 = 0;
L10 = 0;
L11 = 0;
```

code gen.

```
void counter_reset(outC_counter *outC)
{
  outC->init = kcg_true;
}

void counter(inC_counter *inC, outC_counter *o
{
  kcg_int tmp;

  if (outC->init) {
    outC->count = 0;
  }
  else {
    if (inC->event) {
      tmp = 1;
    }
    else {
      tmp = 0;
    }
    outC->count = tmp + outC->_L9;
  }
  if (inC->reset) {
    outC->_L9 = 0;
  }
  else {
    outC->_L9 = outC->count;
  }
  outC->init = kcg_false;
}
```

A certified compiler? an idea born in 1992/1993

Scade: a language from Lustre (common lab VERILOG/IMAG = VERIMAG).

Make a certified compiler for the strictest standards avionics.

Avoid having to recheck that the generated code is correct with respect to the source.

Based on the simple and precise definition of Lustre and its code generation.

The major evolutions of the language (Scade 6) and of the compiler were all guided by this objective.

An obligation of means vs an bligation of result (e.g., CompCert).

A certified compiler? an idea born in 1992/1993

Scade: a language from Lustre (common lab VERILOG/IMAG = VERIMAG).

Make a certified compiler for the strictest standards avionics.

Avoid having to recheck that the generated code is correct with respect to the source.

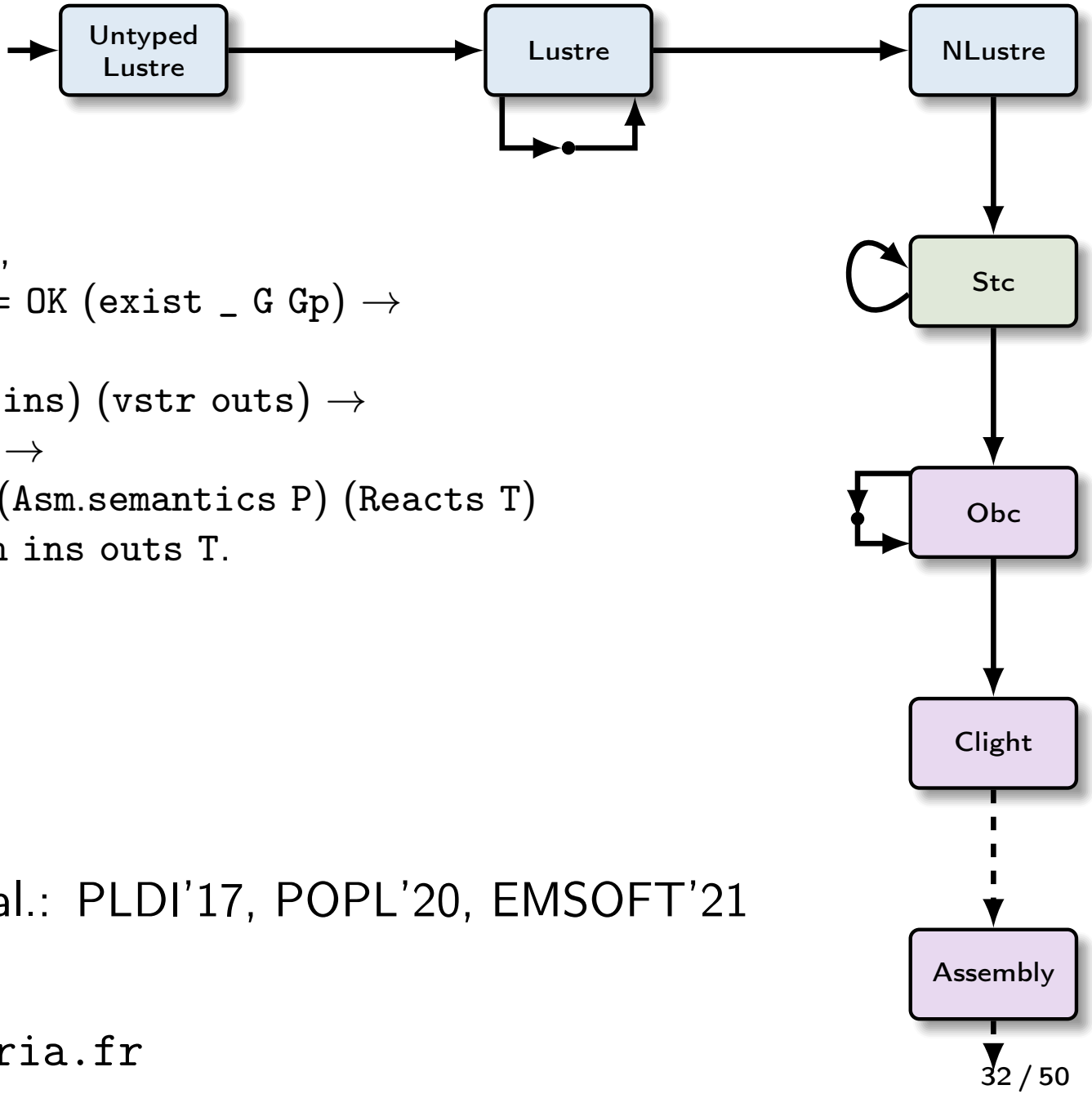
Based on the simple and precise definition of Lustre and its code generation.

The major evolutions of the language (Scade 6) and of the compiler were all guided by this objective.

An obligation of means vs an bligation of result (e.g., CompCert).

Can (and how) be developed a proven synchronous compiler?

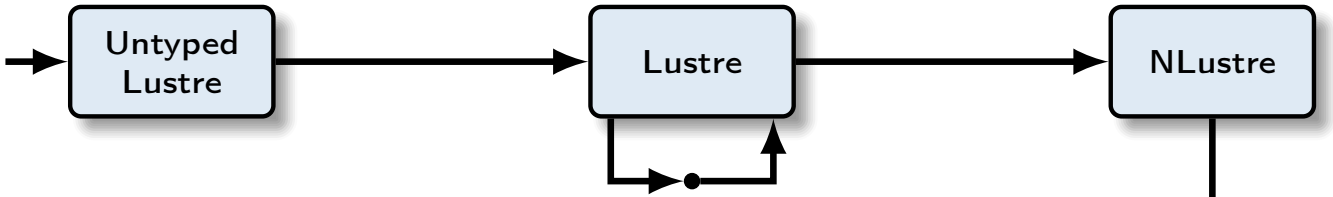
Vélus: a proven Lustre compiler to CompCert




```
Theorem behavior_asm:
  ∀ D G Gp P main ins outs,
    elab_declarations D = OK (exist _ G Gp) →
    wt_ins G main ins →
    sem_node G main (vstr ins) (vstr outs) →
    compile D main = OK P →
    ∃ T, program_behaves (Asm.semantics P) (Reacts T)
         ∧ bisim_io G main ins outs T.
```

- Timothy Bourke et al.: PLDI'17, POPL'20, EMSOFT'21
- 100kLOC de Coq.
- `https://velus.inria.fr`

Vélus: a proven Lustre compiler to CompCert




```
Theorem behavior_asm:
  ∀ D G Gp P main ins outs,
    elab_declarations D = OK (exist _ G Gp) →
    wt_ins G main ins →
    sem_node G main (vstr ins) (vstr outs) →
    compile D main = OK P →
    ∃ T, program_behaves (Asm.semantics P) (Reacts T)
         ∧ bisim_io G main ins outs T.
```

typing and elaboration have succeed,

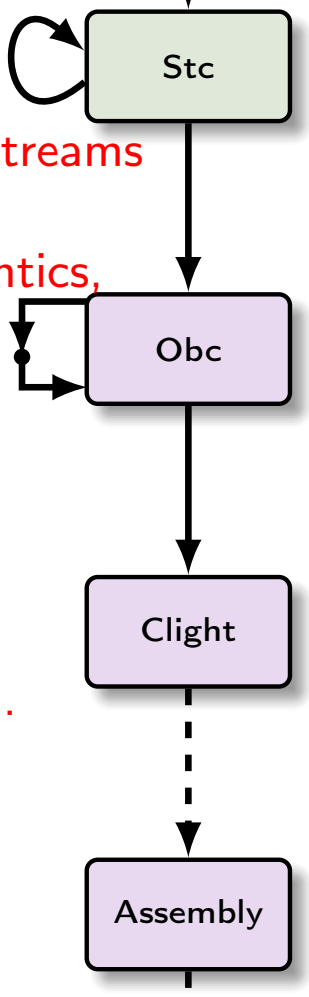
– ∀ well typed input streams

link to output streams by the data-flow semantics,

if the compilation succeed,

then, the generated assembly produces an infinite trace. . .

. . . which corresponds to the data-flow model.



- Timothy Bourke et al.: PLDI'17, POPL'20, EMSOFT'21
- 100kLOC de Coq.
- https://velus.inria.fr

The stepper motor?

`https://vertmo.org/jsofocaml/try-velus/` [5]

[5] Thanks to Timothy Bourke and Basile Pesin.

A short focus on Causality

It has been the subject of strong debates.

There is no absolute notion of causality: there is not one that is better than the other.

Some are more powerful (they accept more program).

The choice is determined by the code you target, e.g., circuit or software.

For circuits, if cyclic circuits are forbidden by the synthesis tool, why fighting for constructive causality?

For software, different compromises, e.g., code size of the target code.

A demo with Zrun

An experiment with the `zrun` interpreter of Zélus [6]

`https://github.com/INRIA/zelus/tree/work`

The causality in Lustre vs Signal vs Esterel correspond to different interpretations of the conditional. With `zrun`, you can try several.

Syntactic Causality (Lustre)

$$
\begin{array}{lcl}
^\star \mathit{if}\ \bot\ \mathit{then}\ \_\ \mathit{else}\ \_ & \stackrel{def}{=} & \bot \\
^\star \mathit{if}\ \_\ \mathit{then}\ \bot\ \mathit{else}\ \_ & \stackrel{def}{=} & \bot \\
^\star \mathit{if}\ \_\ \mathit{then}\ \_\ \mathit{else}\ \bot & \stackrel{def}{=} & \bot \\
^\star \mathit{if}\ \mathit{true}\ \mathit{then}\ x\ \mathit{else}\ \_ & \stackrel{def}{=} & x \\
^\star \mathit{if}\ \mathit{false}\ \mathit{then}\ \_\ \mathit{else}\ y & \stackrel{def}{=} & y
\end{array}
$$

[6] See previous talks at SYNCHRON - winter 2019 and 2021.

Causality

Lazy Causality

$$^\star \mathit{if}\ \bot\ \mathit{then}\ \_\ \mathit{else}\ \_ \quad \stackrel{def}{=} \quad \bot$$

$$^\star \mathit{if}\ \mathit{true}\ \mathit{then}\ x\ \mathit{else}\ \_ \quad \stackrel{def}{=} \quad x$$
$$^\star \mathit{if}\ \mathit{false}\ \mathit{then}\ \_\ \mathit{else}\ y \quad \stackrel{def}{=} \quad y$$

Constructive Causality (Esterel)

$$^\star \mathit{if}\ \bot\ \mathit{then}\ v_1\ \mathit{else}\ v_2 \quad \stackrel{def}{=} \quad \mathit{if}\ v_1 = v_2\ \mathit{then}\ v_1\ \mathit{else}\ \bot$$

$$^\star \mathit{if}\ \mathit{true}\ \mathit{then}\ x\ \mathit{else}\ \_ \quad \stackrel{def}{=} \quad x$$
$$^\star \mathit{if}\ \mathit{false}\ \mathit{then}\ \_\ \mathit{else}\ y \quad \stackrel{def}{=} \quad y$$

Causality

With the following definition for the or/and gates:

$$
\begin{array}{lcl}
{}^{\star}or(x,y) & \stackrel{def}{=} & \textit{if } x \textit{ then true else } y \\
{}^{\star}and(x,y) & \stackrel{def}{=} & \textit{if } x \textit{ then } y \textit{ else false}
\end{array}
$$

With the first interpretation, the two operators are strict. With the second one, they are "sequential" (left-to-right); with the third one, it coicindes with the 3-valued logic for boolean operators.

$$
\begin{array}{lcl}
{}^{\star}or(true, \_) & = & true \\
{}^{\star}or(\_, true) & = & true \\
{}^{\star}or(false, x) & = & x \\
{}^{\star}or(x, false) & = & x \\
{}^{\star}and(false, \_) & = & false \\
{}^{\star}and(\_, false) & = & false \\
{}^{\star}and(true, x) & = & x \\
{}^{\star}and(x, true) & = & x
\end{array}
$$

Examples in Zélus

Examples available at:
`https://github.com/INRIA/zelus/tree/work/zrun/tests/good/`.

A simple counter.

The cyclic circuit of Malik.

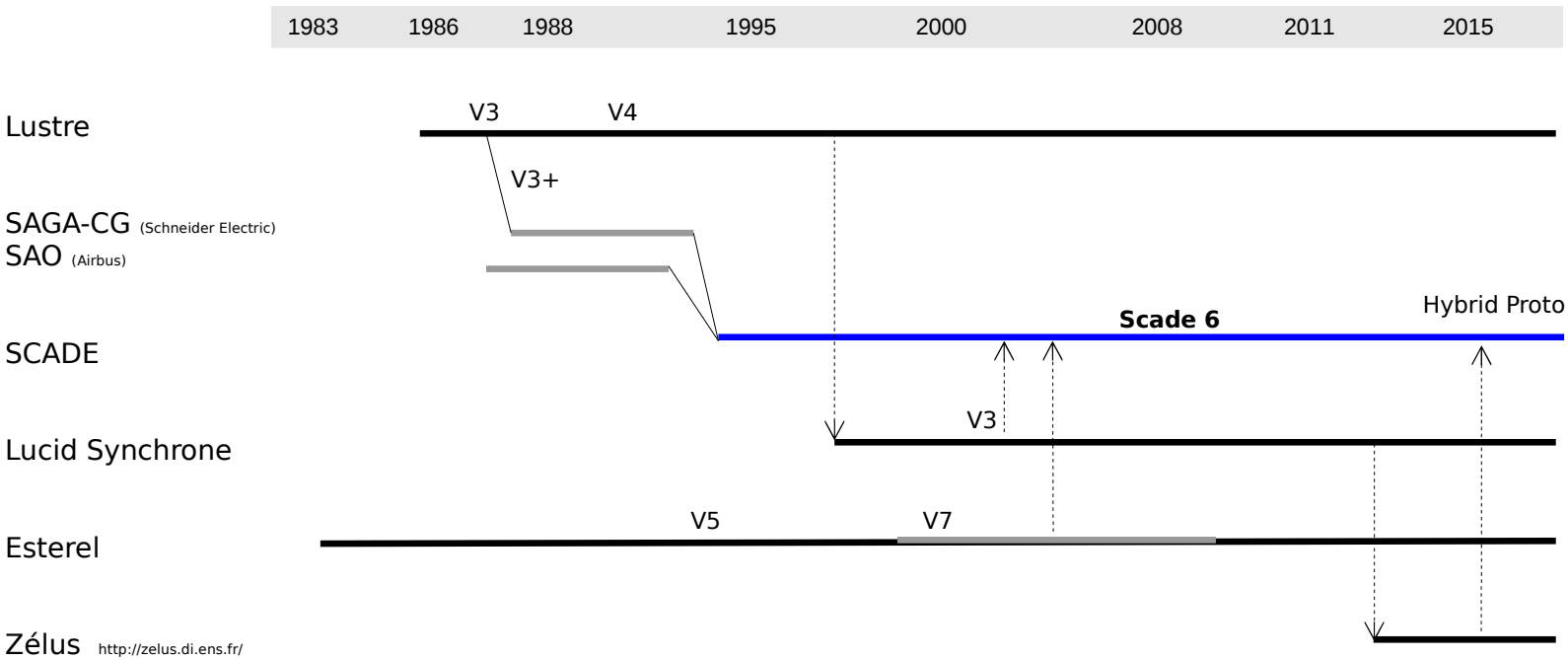
The Bus arbiter by R. de Simone.

A surprise: this later example does not need the interpretation given by Esterel for the *or* and *and* gate. It is enough to use a lazy conditional. See arbiter.zls.

Modular causality analysis

The causality analysis in Zélus and Scade is done modularily through a dedicated type system. In the academic Lustre compiler, it is done on the result of the static inlining of nodes.

Timeline


1983
1986
1988
1995
2000
2008
2011
2015
Lustre
V3
V4
V3+
SAGA-CG (Schneider Electric)
SAO (Airbus)
SCADE
Scade 6
Hybrid Proto
Lucid Synchrone
V3
Esterel
V5
V7
Zélus http://zelus.di.ens.fr/

I forgot very important subjects and works.

Synchronous observers [HLR92, HLR93]

A synchronous program that observes the inputs/outputs.

A wide set of formal verification techniques, actively participating in the progress of model-checking.

Direct links with circuit verification.

e.g., enumerative methods (before 90); symbolic by BDD (90-00); based on SAT (00-08); SMT (08-).

Academic tools (e.g., Lesar, Kind, Kind2 [CMST16]) and industrial.

The idea of synchronous observers has been taken up everywhere, e.g., Scade, Simulink.

Original synchronous languages for specifying properties: HLL (RATP and Prover) [OBC18], Lutin [RRJ08], Stimulus, CCSL [ZWCM21], etc.

An abundance of languages and compilers

Many languages/compilers to test/experiment with new ideas.

Integrate ideas from the synchronous into an existing language.

ReactiveC (from reactive in C), SugarCubes (from reactive in Java), ReactiveML (from reactive in ML), HipHop (from reactive to Esterel in JavaScript), etc.

To express finer (C-style) imperative features in a synchronous language, e.g. SCCharts [vHDM+14].

To express and exploit periodic calculations: Lucy-n [MPP10], Prelude [CBF+11].

And many others... SaxoRt compiler from Esterel, Quartz language, Shim, Blech, etc.

Distribute code, quasi-synchronous model, generation into tasks, etc.

The SYNCHRON workshop

An annual and uninterrupted workshop since 1994.

http://synchron2021.inria.fr

A precious and unique place to present on-going or more complete work.

Invite colleagues on new topics, e.g., synchronous programming for music, modeling of hybrid systems.

No publication, no selection committee, no referee, no program.

but open debates, confrontation of ideas.

Conclusion

- Dedicated, parallel and deterministic languages;
- adapted to the mathematical culture and the practice of engineers;
- adopted from the beginning.
- Do not compromise on principles: static and dynamic semantics specified in detail as well as the compiler.
- A continuous evolution of languages, compilation methods, compile-time checks.
- Ideas used in other applications: web, high frequency trading, mixed music, ChatBot, etc.
- A direct impact on industrial tools used every day.
- Some to build software; others to specify/verify properties of systems implemented otherwise.

And the story continues

- Industrial certification: can Coq specs complement/replace current (human) verification steps?
- Compilation that includes translation validation, independent testing that uses an executable semantics.
- Relax some synchronism constraints but control end-to-end latencies (Airbus's practice, etc.).
- Express real-time constraints and exploit them to generate sequential code, in tasks or parallel.
- Write hybrid models; probabilistic.
- An old topic: calculating with arrays. What's new?

A surprise: the principles and style of the language SISAL [7] works well with Lustre.

[7] Stream and Iteration in a Single Assignment Language [FCO90]

References I

T. Amagbegnon, L. Besnard, and P. Le Guernic.

Implementation of the data-flow synchronous language signal.
In *Programming Languages Design and Implementation (PLDI)*, pages 163–173. ACM, 1995.

E. A. Ashcroft and W. W. Wadge.

Lucid, the data-flow programming language.
A.P.I.C. Studies in Data Processing, Academic Press, 1985.

Albert Benveniste, Timothy Bourke, Benoit Caillaud, Bruno Pagano, and Marc Pouzet.

A Type-based Analysis of Causality Loops in Hybrid Systems Modelers.
In *International Conference on Hybrid Systems: Computation and Control (HSCC)*, Berlin, Germany, April 15–17 2014. ACM.

Gérard Berry.

The constructive semantics of pure esterel, draft, version 3.
Draft book. Available at:
`http://www-sop.inria.fr/members/Gerard.Berry/Papers/EsterelConstructiveBook.pdf`, 2002.

G. Berry and G. Gonthier.

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

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.

F. Boussinot.

Reactive C: An Extension of C to Program Reactive Systems.
Software Practice and Experience, 21(4):401–428, 1991.

References II

Mikel Cordovilla, Frédéric Boniol, Julien Forget, Eric Noulard, and Claire Pagetti.

Developing critical embedded systems on multicore architectures: the Prelude-SchedMCore toolset.
In *19th International Conference on Real-Time and Network Systems*, Nantes, France, September 2011. Irccyn.

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.

Adrien Champion, Alain Mebsout, Christoph Sticksel, and Cesare Tinelli.

The kind 2 model checker.
In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 510–517. Springer, 2016.

Paul Caspi and Marc Pouzet.

Synchronous Kahn Networks.
In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Philadelphia, Pensylvania, May 1996.

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, March 1998.
Extended version available as a VERIMAG tech. report no. 97–07 at `www.di.ens.fr/~pouzet/bib/bib.html`.

Pascal Cuoq and Marc Pouzet.

Modular Causality in a Synchronous Stream Language.
In *European Symposium on Programming (ESOP'01)*, Genova, Italy, April 2001.

References III

Jean-Louis Colaço and Marc Pouzet.

Clocks as First Class Abstract Types.
In *Third International Conference on Embedded Software (EMSOFT'03)*, Philadelphia, Pennsylvania, USA, october 2003.

Jean-Louis Colaço and Marc Pouzet.

Type-based Initialization Analysis of a Synchronous Data-flow Language.
International Journal on Software Tools for Technology Transfer (STTT), 6(3):245–255, August 2004.

Jean-Louis Colaco, Bruno Pagano, and Marc Pouzet.

Scade 6: A Formal Language for Embedded Critical Software Development.
In *Eleventh International Symposium on Theoretical Aspect of Software Engineering (TASE)*, Sophia Antipolis, France, September 13-15 2017.

J. T. Feo, D. C. Cann, and R. R. Oldehoeft.

A report on the Sisal language project.
Journal of Parallel and Distributed Computation, 10:349–366, 1990.

Georges Gonthier.

Sémantiques et modèles d'exécution des langages réactifs synchrones.
PhD thesis, Université d'Orsay, 1988.

N. Halbwachs, F. Lagnier, and C. Ratel.

Programming and verifying real-time systems by means of the synchronous dataflow language lustre.
IEEE Transaction on Software Engineering, 18(9):785–793, September 1992.
Available through anonymous ftp at `imag.fr:pub/SPECTRE/LUSTRE/PAPERS/lustre.tse.ps.gz`.

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.

References IV

Gilles Kahn.

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

Gilles Kahn and David B. MacQueen.

Coroutines and networks of parallel processes.
In *IFIP Congress*, pages 993–998, 1977.

Florence Maraninchi.

Operational and compositional semantics of synchronous automaton compositions.
In *CONCUR*, pages 550–564, 1992.

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.

Louis Mandel, Florence Plateau, and Marc Pouzet.

Lucy-n: a n-Synchronous Extension of Lustre.
In *10th International Conference on Mathematics of Program Construction (MPC'10)*, Manoir St-Castin, Québec, Canada, June 2010. Springer LNCS.

Michael Mendler, Thomas R. Shiple, and Gérard Berry.

Constructive boolean circuits and the exactness of timed ternary simulation.
Form. Methods Syst. Des., 40(3):283–329, June 2012.

Julien Ordioni, Nicolas Breton, and Jean-Louis Colaço.

HLL v.2.7 Modelling Language Specification.
Other STF-16-01805, RATP, May 2018.

References V

Marc Pouzet.

Lucid Synchrone, version 3. Tutorial and reference manual.
Université Paris-Sud, LRI, April 2006.
Distribution available at: `https://www.di.ens.fr/~pouzet/lucid-synchrone/`.

Marc Pouzet and Pascal Raymond.

Modular Static Scheduling of Synchronous Data-flow Networks: An efficient symbolic representation.
In *ACM International Conference on Embedded Software (EMSOFT'09)*, Grenoble, France, October 2009.

Pascal Raymond, Yvan Roux, and Erwan Jahier.

Lutin: A language for specifying and executing reactive scenarios.
EURASIP J. Embed. Syst., 2008, 2008.

Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien.

Scharts: sequentially constructive statecharts for safety-critical applications: Hw/sw-synthesis for a conservative extension of synchronous statecharts.
In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 372–383.
ACM, 2014.

Yuanrui Zhang, Hengyang Wu, Yixiang Chen, and Frédéric Mallet.

A clock-based dynamic logic for the verification of CCSL specifications in synchronous systems.
Sci. Comput. Program., 203:102591, 2021.