

# Synchronous Kahn Networks (ten years later)

Marc Pouzet  
LRI

Marc.Pouzet@lri.fr

Workshop SYNCHRON, 27/11/06

## Overview

- The origins
- From Lustre to Lucid Synchrone
- Developing a Language
- Conclusion

# **The origins (sept. 94 – june 96)**

(VERIMAG, Montbonnot & McGill University, Montreal)

## The origins

What are the relationships between:

- Kahn Process Networks
- Synchronous Data-flow Programming (e.g., Lustre)
- (Lazy) Functional Programming (e.g., Haskell)
- Types and Clocks
- State machines and stream functions

**What can we learn from the relationships between synchronous and functional programming?**

## The first intuitions

In 1993 (I was still doing my PhD thesis in Rocquencourt...) Paul noticed the relationships between functional programming and synchronous programming in two papers:

- **Lucid Synchrone** [Caspi, OPOPAC 93],
- **Towards Recursive Block Diagrams** [Caspi, RTP 94]

He made the following observations:

- Synchronous dataflow can be simulated in a functional lazy language in a few lines (LazyML at that time).
- We can program in the semantics.
- We can benefit from all the features of the host language: type inference, modularity, higher-order, recursion.
- with the idea that extending Lustre with functional features was both natural and fruitful

# Lustre and Lazy streams

```
module Streams where
```

```
-- lifting constants
```

```
constant x = x : (constant x)
```

```
-- pointwise application
```

```
extend (f:fs) (x:xs) = (f x):(extend fs xs)
```

```
-- delays
```

```
(x:xs) 'fby' y = x:y
```

```
pre x y = x : y
```

```
-- sampling
```

```
(x : xs) 'when' (True : cs) = (x : (xs 'when' cs))
```

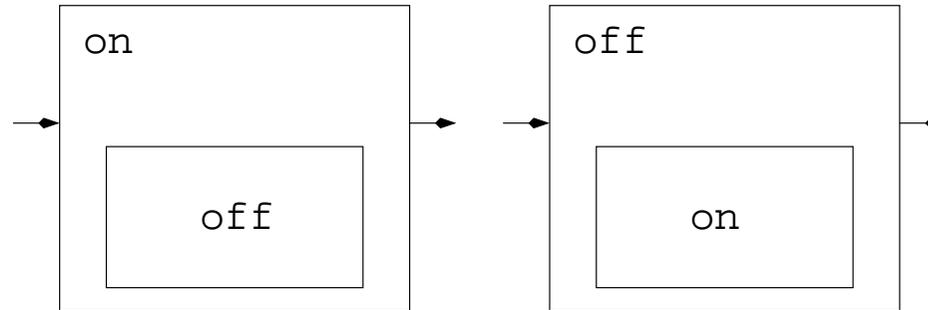
```
(x : xs) 'when' (False : cs) = xs 'when' cs
```

```
merge (True : c) (x : xs) y = x : (merge c xs y)
```

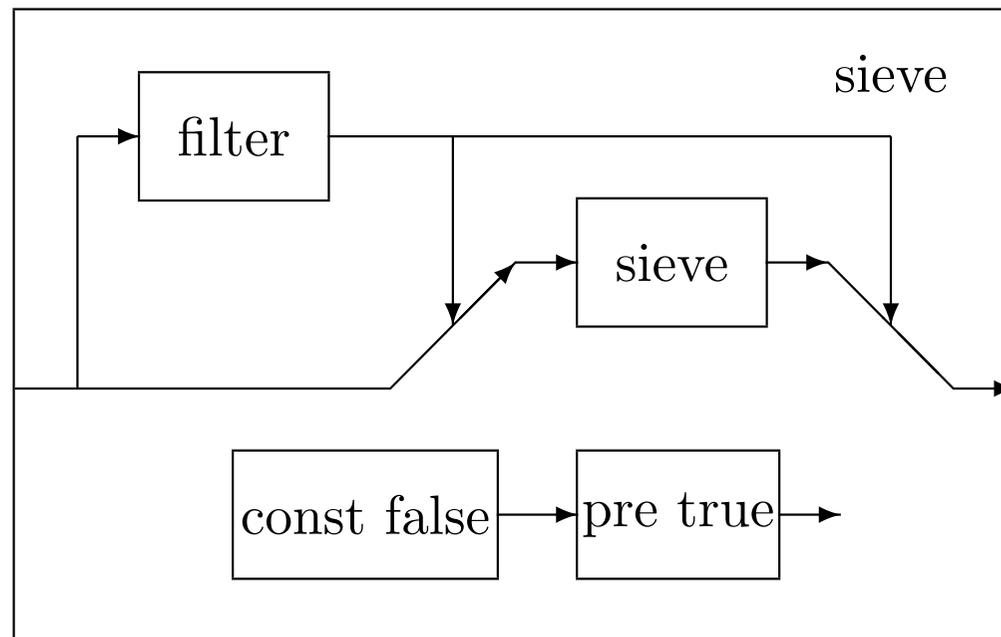
```
merge (False : c) x (y : ys) = y : (merge c x ys)
```

# Recursive Block Diagrams

- Dynamic reconfiguration (i.e., imperative constructs of Esterel) can be encoded as classical tail-recursive functions



- More general (dynamic) networks can be considered such as the Eratosthene sieve, etc.



## Extending Synchronous Data-flow

- Synchronous data-flow can be characterized as some class of static bounded memory data-flow networks.
- They are not recursively defined and obey some “synchronous” constraints (*clock calculus*).
- Those networks enjoy efficient compilation techniques.

Based on Kahn’s relationship between data-flow and stream functions, synchrony can be related to Wadler’s listlessness and deforestation techniques ([Wadler, LFP 84, TCS 90])

- Listlessness is a compilation techniques which eliminates intermediate data-structures from stream programs, e.g.,  $\text{hd } (x : y) = x$
- avoid the need of a lazy evaluation mechanism

Can we extend the class of static synchronous data-flow to higher-order and dynamical networks, thus giving sense to a larger class of synchronous data-flow?

## Synchrony and Listlessness evaluation

But deforestation techniques fail to deforest (i.e., diverge) on some simple programs such as `current v (x 'when' c) c` which are trivially accepted in Lustre.

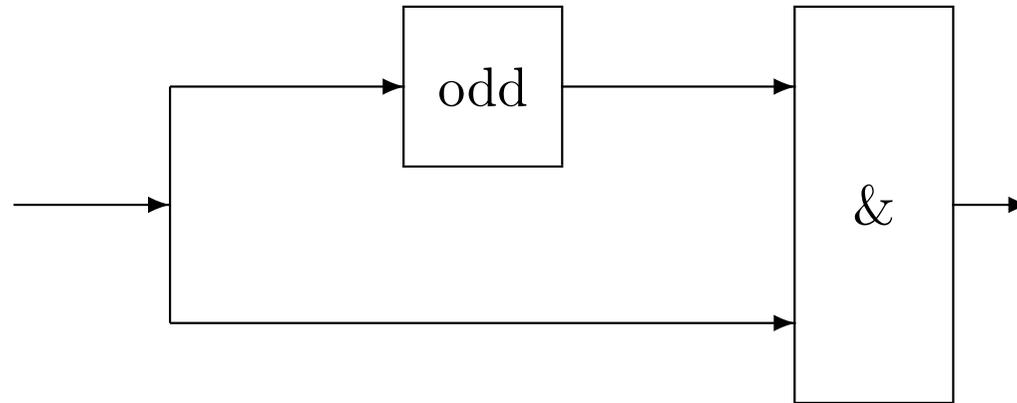
$$(x_0 : x) \text{ 'when' } (\text{True} : c) = x_0 : (x \text{ 'when' } c)$$
$$(x_0 : x) \text{ 'when' } (\text{False} : c) = x \text{ 'when' } c$$
$$\text{curr } v (x_0 : x) (\text{True} : c) = x_0 : (\text{current } x_0 x c)$$
$$\text{current } v x (\text{False} : c) = v : (\text{current } v x c)$$

In **Lustre** syntax: `(current (x when c))`

In **SCADE**: `conduct(c; v; id)` (where `id x = x`)

Existing deforestation techniques at that time (e.g., “A Short Cut to Deforestation” [FPCA 93]) failed to deforest many useful synchronous programs

# Clock Constraints and Synchrony



The computation of  $(x_n \& x_{2n})_{n \in \mathbb{N}}$  is not real-time

```
let odd x = x when half
```

```
let non_synchronous x = x & (odd x)
                        ^^^^^^^^
```

This expression has clock 'a on half, but is used with clock 'a.

## Execution with unbounded FIFOs!!!

- Deforestation techniques diverge on those programs
- We should statically reject those programs
- This is the purpose of the *clock calculus* in synchronous data-flow languages

## Synchronous Kahn Networks [ISLIP'95, ICFP'96]

Provide an extension of synchronous data-flow by:

- defining a functional kernel with abstraction, application and recursion and whose first order restriction is reminiscent to Lustre,
- define a synchronous operational semantics for it, generalizing existing ones; this allows us to characterize “synchronous data-flow behaviors”,
- define a clock calculus for this language and express it as a type-system, which, in turn, allows us to generalize it to functional features

**Property:** well clocked program can be executed synchronously

# From Lustre to Lucid Sychrone

(McGill University, Université Paris 6)

# A Co-iterative Characterization of Synchronous Stream Functions [CMCS 98]

These first works with Paul showed that it was possible to implement an extension of Lustre and it was of course called Lucid Synchrone

- a small functional kernel (higher-order and recursion)
- a dependent type clock calculus
- causality check was trivial (graph based)
- a partial compilation method: this was the hard part (we failed to make it modular)

We started working on the problem in june 96 (after ICFP), from the work of Jacob & Rutten (pre-version of “A tutorial on (co)algebras and (co)induction”, EATCS Bulletin 97

- find a modular compilation technique for the extended kernel
- explain the classical compilation of SCADE in a few lines
- the method is time resistant: it is still in use in the current compiler

## Functional Programming and Reactive Systems

At about the same period, several projects identified the interest of functional (data-flow) programming to model reactive systems.

- Mary Sheeran noticed in 84 the interest of functional programming for describing synchronous circuits in  $\mu$ FP [LFP 84]
- Various embedding of Hardware Description Languages in Haskell: Lava [Sheeran & all, ICFP 98], Hawk [Launchbury & all, ICFP 99], etc.
- Functional Reactive Programming [Hudak, Petterson, ICFP 99]
- Fran (Functional Reactive Animation) [Elliot & Hudak, ICFP 97]
- Now Multi-stage programming techniques [Taha PhD. 99, etc.], ReFlect (Intel), etc.

## Functional Reactive Programming (FRP, FRAN)

- accept too many programs: synchronous and asynchronous
- no *guaranty* at compile time
- memory leaks, unbounded recursion, etc.
- no compilation (or simply macro-expansion)
- Haskell run-time (but a macro-expansion to C is feasible)
- the type system of Haskell is not sufficient

## Synchronous circuits (e.g., Lava)

- very elegant and very well adapted to the design of synchronous circuits
- “two stage” approach: the execution of the program produces a net-list
- length-preserving functions only (circuits with a base clock)
- the type system of Haskell is not sufficient
- no static analysis (e.g., causality analysis) nor modular compilation

# Developping a language (sept. 96 – )

(LIP6, Univ. Paris 6 & LRI, Univ. Paris-Sud 11)

# Lucid Synchrone

How to extend Lustre in a conservative way (without breaking it)?

Build a “laboratory” language

- study (prototype) extensions of Lustre
- experiment things, manage all the compilation chain and write programs!
- Version 1 (1995), Version 2 (2001), V3 (2006)

Follow a few principles:

- types everywhere
- clock based approach: everything should be explained in term of a basic clocked language
- modularity everywhere (type analysis, separate compilation)

# Some developments I

## Typing:

- Automatic type inference with polymorphism; various versions (latest [Emsoft'04])

**Clock calculus:** Clocks play a central role both on the semantics side and the implementation side.

- same philosophy as Lustre (differs from the one of Signal)
- clocks as types (provides both polymorphism and inference) making them more usable
- defined as a dependent type system [ICFP 96]
- start of the collaboration with Jean-Louis Colaço (Esterel-Technologies) on the design of a prototype compiler for SCADE (~ 2000)
- the prototype ReLuC compiler uses the first-order version of this calculus
- programming constructs (e.g., `merge`)
- then a simpler calculus reminiscent to Milner-type system [Emsoft 2003]

## Some developments II

### Type-based program analysis:

- initialization analysis (with JL. Colaço, [SLAP 02, STTT 04])
- both implemented in Lucid Sychrone and ReLuC
- greatly reduces the number of false alarms

### Mixing imperative construct and data-flow:

- PhD. thesis of G. Hamon [PPDP 00, SLAP 04]
- work with Colaço & Pagano (ET) on the design of the mix of automata and data-flow systems
- translation semantics relying on the clock mechanism [Emsoft 05], direct synchronous semantics [Emsoft 06]
- both implemented in ReLuC and Lucid Sychrone

Recently, we came back to the origins (N-Synchronous Kahn Networks) to relax the semantics to allow non strictly synchronous systems for the implementation of video systems (project with Philips semiconductor, now NXP)

## Main results

- Synchronous Kahn networks [ICFP'96]
- Clocks as dependent types [ICFP'96]
- Modular compilation [CMCS'98]
- Control-structures and data-flow [PPDP'00]
- causality analysis [ESOP'01]
- initialization analysis [SLAP'02, STTT'04]
- ML-like clock calculus [Emsoft'03]
- higher-order and typing [Emsoft'04]
- data-flow and state machines [Emsoft'05, Emsoft'06]
- N-Synchronous Kahn Networks [Emsoft'05, POPL'06]

## Laboratory language?

Many of these ideas, originally introduced by Paul, are now integrated in two industrial tools of the field.

- the ReLuC compiler of SCADE is based (and improves) techniques introduced in Lucid Synchrone
- same philosophy: types everywhere, modularity, etc.
- typing, clock calculus
- program constructs (e.g., merge)
- static analysis (initialization)
- design/semantics of ReLuC (next SCADE)

Athys (Dassault-Systèmes) is developing a programming environment into the Catia suite for PLC:

- basing it on an imperative kernel (reminiscent to Esterel)
- automatic type synthesis (with polymorphism), module systems

## Conclusion

This work was initiated in 93 by Paul Caspi and we started our collaboration in sept. 94

- The direction was clearly drawn from the very beginning and only a few things really changed.
- Reformulating synchronous data-flow in the functional setting was very fruitful and many extensions came naturally.
- The language exists and contains most of the features we were looking at at the very beginning.
- The goal to make it a laboratory language succeed.
- Paul had a strong influence in it (simplicity of constructions, orthogonality of concept, unified semantics).