

Dynamic Scheduling of Synchronous Programs
in
LUCID SYNCHRONE

Adrien Guatto
Joint work with L. Mandel and M. Pouzet

PARKAS team, LIENS & INRIA

SYNCHRON 2011

What this is about

Alternative titles

- ▶ Modular code generation for LUSTRE / LUCY-N without static clock information
- ▶ Experiments with Latency-Insensitive Design in LUCID SYNCHRONE
- ▶ One use of higher-order stream functions

Bottom line

A latency insensitive shallow embedding of LUSTRE/LUCY-N in LUCID SYNCHRONE.

Introduction

Context

A latency-insensitive protocol

Prototype implementation in LUCID SYNCHRONE

Conclusion

Original motivations

LUCY-N

A variant of LUSTRE with:

- ▶ ultimately periodic sampling/merging conditions;
- ▶ a buffer operator.

lucync

The compiler's role is to:

- ▶ infer clocks;
- ▶ compute buffer sizes;
- ▶ generate code.

LUSTRE 101

```
let node f c = o where
  rec o = merge c m 42
  and m = 0 fby (m + 1)
```

```
f(true fby false fby true fby true fby false fby true...)
```

time	t_0	t_1	t_2	t_3	t_4	t_5	...
c	<i>true</i>	<i>false</i>	<i>true</i>	<i>true</i>	<i>false</i>	<i>true</i>	
o	0	42	1	2	42	3	
m	0	.	1	2	.	3	

Clocks:

- ▶ $f :: 'a \rightarrow 'a$
- ▶ $m :: 'a$ on c

In the generated code, state changes for m must occur exactly when c is true.

LUCY-N (101)

```
let node f x = o where
  rec o = buffer v1 + v2
  and v1 = x when (10)
  and v2 = x when (01)
```

time	t_0	t_1	t_2	t_3	t_4	t_5	...
(10)	1	0	1	0	1	0	
x	x_0	x_1	x_2	x_3	x_4	x_5	
v1	x_0		x_2		x_4		
v2		x_1		x_3		x_5	
buffer v1		x_0		x_2		x_4	
o		$x_0 + x_1$		$x_3 + x_2$		$x_5 + x_4$	

Clocks:

- ▶ v1 :: 'a on (10)
- ▶ v2 :: 'a on (01)
- ▶ o :: 'a on (01)

Traditional modular code generation for LUSTRE

```
let node f c = o where
  rec o = merge c m 42
  and m = 0 fby (m + 1)
```

```
m :: 'a on c
```

```
o :: 'a
```

```
class f:
  mem m = 0;
  method step(in c, out o):
    if (c):
      o := m;
      m := m + 1;
    else:
      o := 42;
```

- ▶ Compiling means translating equations with (implicit) activation rhythms to guarded affectations.
- ▶ Code generation translates clock types to conditional statements.

Modular code generation for LUCY-N

```
let node f (x, y) = x when (1001) + y when (0110)
```

```
val f :: forall 'a.
```

```
'a on (011110) * 'a on (110011) -> 'a on (010010)
```

Clocking LUCY-N

- ▶ Clock types feature ultimately periodic binary words rather than names.
- ▶ Clocking a program amounts to solving some cyclic scheduling problem.
- ▶ Clocks are *schedules*, and thus `lucync` has to invent clocks that are not present in the source program.
- ▶ This may pose a practical problem for code generation with the previous method.

Circumventing the clock generation problem

```
let node g () = (o1, o2) where
  rec n = 0 fby (n + 1)
  and o1 = buffer (n when (00101)) + 1 when (10)
  and o2 = buffer (n when (01)) + 2 when (01)

  n ::  $\alpha$  on (1101010011001100110011010100110011001100)
```

Ideas

- ▶ Have the clocking pass generate simpler clocks;
- ▶ generate more efficient code for the given clocks:
 - ▶ try some compression methods on words;
 - ▶ decompose words into simpler ones thanks to algebraic properties;
- ▶ discard the static clock information and compute the activation rhythms on line (*“clocking” at run-time*).

Intuitions

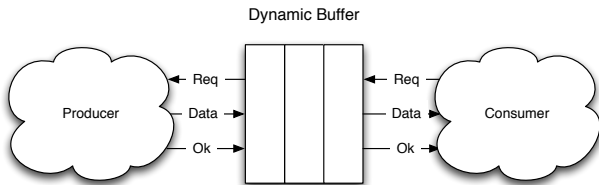
Where are clocks needed in LUCY-N?

- ▶ fby;
- ▶ node application;
- ▶ buffer.

Designing a protocol to compute clocks *on-line* means adding control signals and logic to the source program.

- ▶ Which control signals?
- ▶ What control logic?

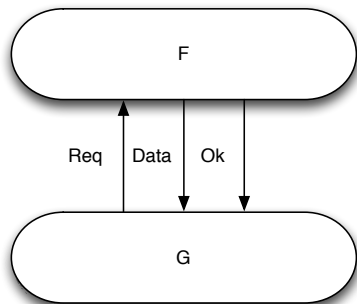
Understanding control signals through buffers



Which control signals for the buffer?

- ▶ *Req*: “I want to read in the buffer” bit.
- ▶ *Ok*: “I want to write in the buffer” bit.
- ▶ For modularity reasons, we add these signals everywhere.

The protocol



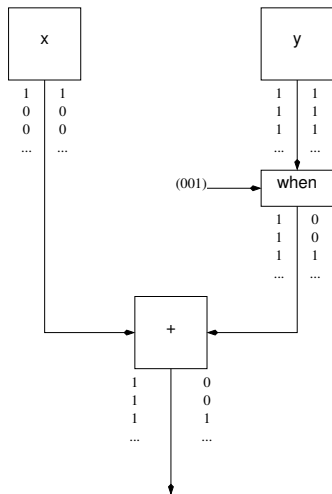
What's in an interface for source-level values of type α ?

- ▶ *req*, boolean: G tells F “Give me data!”;
- ▶ *data*, of type α : F sends G data of type α ;
- ▶ *ok*, boolean: F tells G “I’m giving you valid data”.

Behaviors for various constructs

- ▶ constants c :
 $ok = req, data = c$;
- ▶ synchronous operators ($+$, \dots):
force synchronization of operands;
- ▶ merge of e_1 and e_2 :
set either req_1 or req_2 according to condition;
- ▶ when:
set ok according to the sampling condition;
- ▶ buffer:
eager, always ask the producer for data when non-empty;
- ▶ fby:
initialized buffer of size one.

Local synchronization

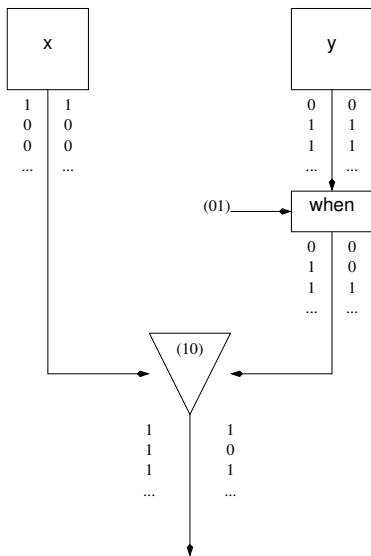


$$x + (y \text{ when } (001))$$

Behaviors for various constructs

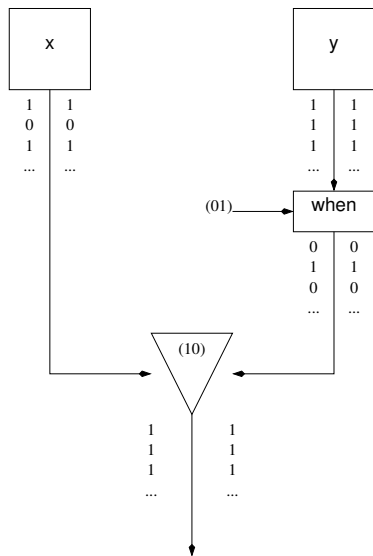
- ▶ constants c :
 $ok = req, data = c$;
- ▶ synchronous operators ($+$, \dots):
force synchronization of operands;
- ▶ merge of e_1 and e_2 :
set either req_1 or req_2 according to condition;
- ▶ when:
set ok according to the sampling condition;
- ▶ buffer:
eager, always ask the producer for data when non-empty;
- ▶ fby:
initialized buffer of size one.

Lazy sampling



`merge (10) x (y when (01))`

Eager sampling



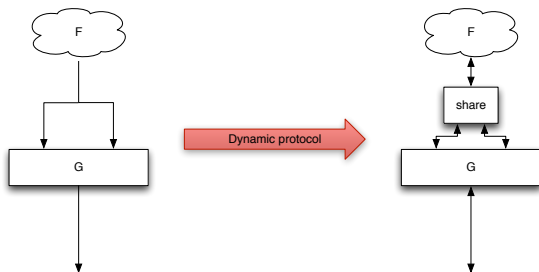
`merge (10) x (y when (01))`

Behaviors for various constructs

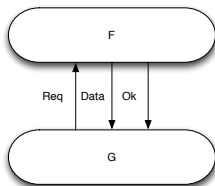
- ▶ constants c :
 $ok = req, data = c$;
- ▶ synchronous operators ($+$, \dots):
force synchronization of operands;
- ▶ merge of e_1 and e_2 :
set either req_1 or req_2 according to condition;
- ▶ when:
set ok according to the sampling condition;
- ▶ buffer:
eager, always ask the producer for data when non-empty;
- ▶ fby:
initialized buffer of size one.

Some remarks

- ▶ Invariant: it is impossible to receive data that was not asked for: $\neg req \Rightarrow \neg ok$.
- ▶ Each construct is naturally delay insensitive, in the sense that the functional behavior of the program do not change if it receives spurious 0 on its control wires.
- ▶ Multiple reads are no longer free, since we have to somehow merge the two *req* wires!



Programming the protocol in LUCID SYNCHRONE



Expressing the translation from the typing point of view?

$$\llbracket \alpha \rrbracket = \text{bool} \Rightarrow \alpha \times \text{bool}$$

In LUCID SYNCHRONE, we can use higher-order stream functions:

```
my_plus : (bool => int * bool) * (bool => int * bool)
          -> (bool => int * bool)
```


Synchronization

```
let node my_synchro e1 e2 (clock req) = (o, ok) where
  rec req1 = req && empty1 and req2 = ...

  and (v1, ok1) = run e1 req1 and (v2, ok2) = ...

  and ok1' = ok1 || not empty1 and ok2' = ...
  and v1' = if empty1 then v1 else b1 and v2' = ...

  and ok = ok1' && ok2'
  and o = (v1', v2')

  and b1 = v1 fby v1' and b2 = ...

  and empty1 = true fby (ok || (not ok1 && empty1))
  and empty2 = ...

val my_synchro :
  (bool => 'a * bool) * (bool => 'b * bool)
-> (bool => ('a * 'b) * bool)
```

DEMO

Remarks and perspectives

Related work

- ▶ Latency-Insensitive Design (Carloni et al.), and in particular. . .
- ▶ **S**ynchronous **E**Lastic **F**low (Kishinevsky et al.).

Remarks

- ▶ using statically scheduled code inside a dynamically scheduled context is easy;
- ▶ ignoring control-flow issues, a SELF-like protocol may be preferable.
- ▶ we do not target hardware implementation (combinatorial pathes everywhere!);
- ▶ we have experimented with a truly asynchronous implementation of the protocol in `ERLANG`.

Conclusion and future work

What we did present

A dynamic scheduling protocol for LUCY-N (or LUSTRE) akin to Latency-Insensitive Design.

TODO list

- ▶ Conjecture: well-clocked programs are live.
- ▶ Explore macro-expansion to imperative code or continuation-based functional code, and compare with the current static code generator.
- ▶ Does the ERLANG experiment has anything to do with asynchronous circuits?