Clock-directed Modular Code-generation for Synchronous Data-flow Languages

Dariusz Biernacki
Univ. of Worclaw (Poland)

Jean-Louis Colaço
Prover Technologies (France)

Grégoire Hamon
The MathWorks (USA)

Marc Pouzet
Université Paris-Sud (France)

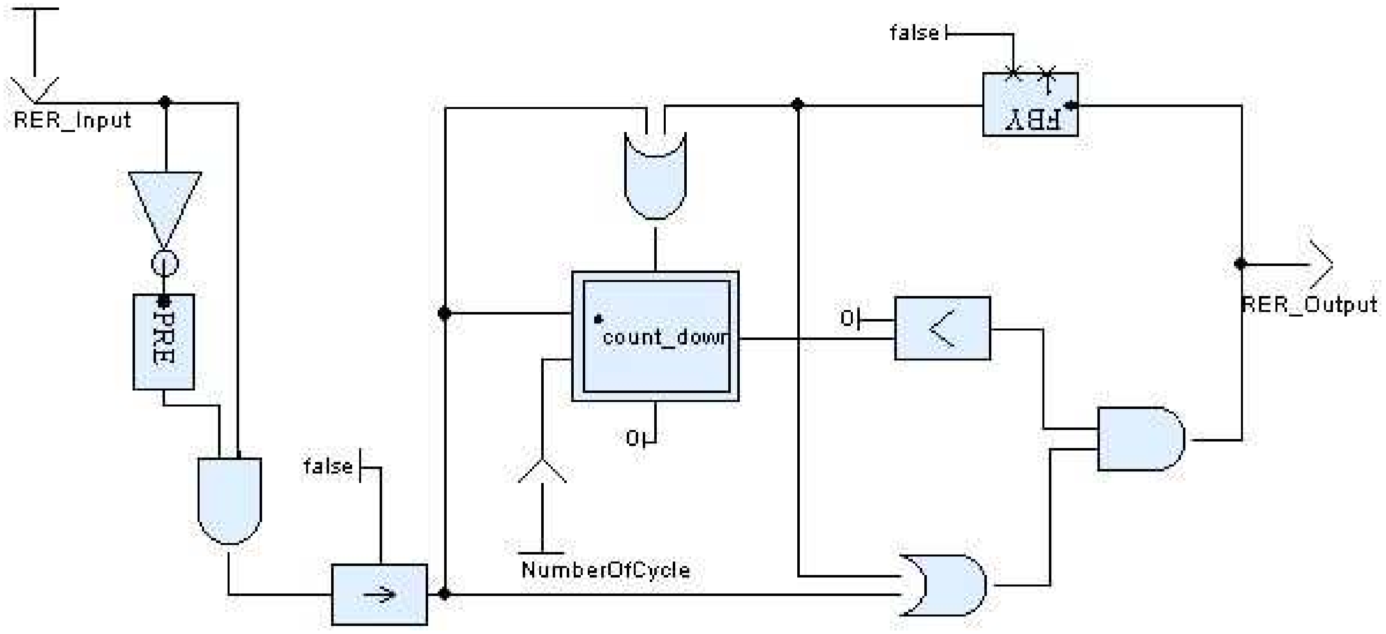
LCTES, Tucson

June 13th, 2008

A certified compiler for Lustre

Develop a certified compiler for synchronous block-diagram formalisms as found in SCADE/Lustre or a subset of Simulink with the help of the proof assistant Coq

Very popular formalism for embedded software (and in particular critical systems)



Existing certification:

- the existing SCADE compiler is “qualified” DO-178B (level A)
- this is mainly “process-based”: introduce more “functional” certification?
- this would allow to put more formal verification on the source code

Motivations (first step)

Code generation from synchronous block-diagram has been addressed in the early 80's and is part of "folklore"

Nonetheless, a minimal and precise description of how code-generation is made in industrial compilers is missing

First build a reference compiler, as small as possible, in (mostly) a purely functional way and based on local rewriting rules

- formalize the **code generation** into imperative sequential code (e.g., C)
- as **small** as possible but **realistic** (the code should be efficient)

as a way to:

- build a certified compiler inside a proof assistant
- complement previous works on the extension/formalization of synchronous languages

Code Generation

Principle:

A stream function $f : \mathit{Stream}(T) \longrightarrow \mathit{Stream}(T')$ is compiled into a pair:

- an initial state and a transition function: $\langle s_0 : S, f_t : S \times T \rightarrow T' \times S \rangle$

a stream equation $y = f(x)$ is computed sequentially by $y_n, s_{n+1} = f_t(s_n, x_n)$

An alternative (more general) solution:

- an initial state: $s_0 : S$
- a value function: $f_v : S \times T \rightarrow T'$
- a state modification (“commit”) function: $f_s : S \times T \rightarrow S'$

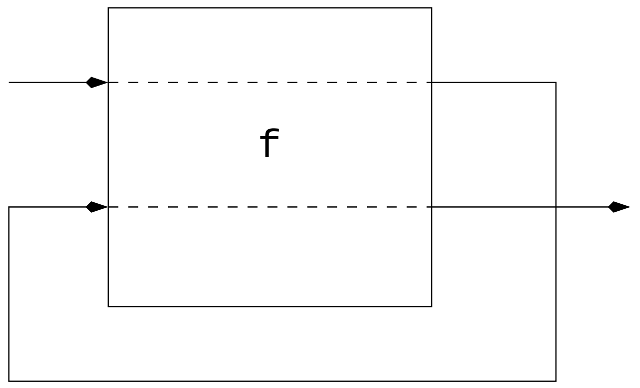
Final remarks:

- this generalises to MIMO systems
- in actual implementations, states are modified in place
- synchrony finds a very practicle justification here: a data-flow can be implemented as a single scalar variable

Modular Code Generation

- produce a transition function for each block definition
- compose them together to produce the main transition function
- static scheduling following data-dependences

But modular sequential code generation is not always feasible even in the absence of causality loops



$$(y, z) = f(t, y)$$

This observation has led to two different approaches to the compilation problem

Two Traditional Approaches

Non Modular Code Generation

- full static inlining before code generation starts
- enumeration techniques into (very efficient) automata ([Halbwachs et all., Raymond PhD. thesis, POPL 87, PLILP 91])
- keeps maximal expressiveness but at the price of modular compilation and the size of the code may explode
- difficult to find the adequate boolean variables to get efficient code in both code and size

Modular code generation

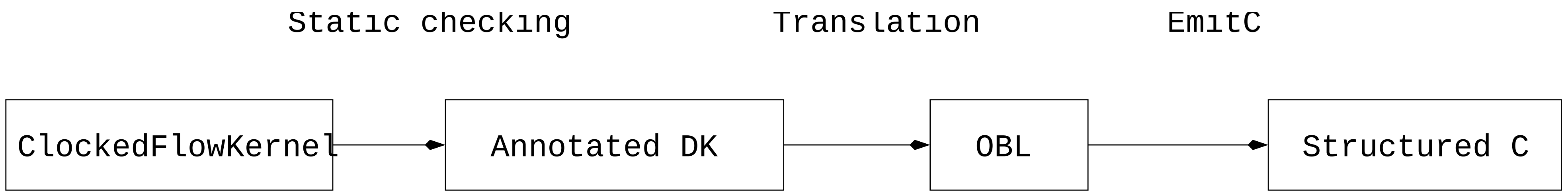
- mandatory in industrial compilers
- no preliminary inlining (unless requested by the user)
- imposes stronger causality constraints: every feedback loop must cross an explicit delay
- well accepted by SCADE users and justified by the need for *tracability*

Between the two, it is possible to decompose a function into a minimal set of atomic functions (Raymond, 1988).

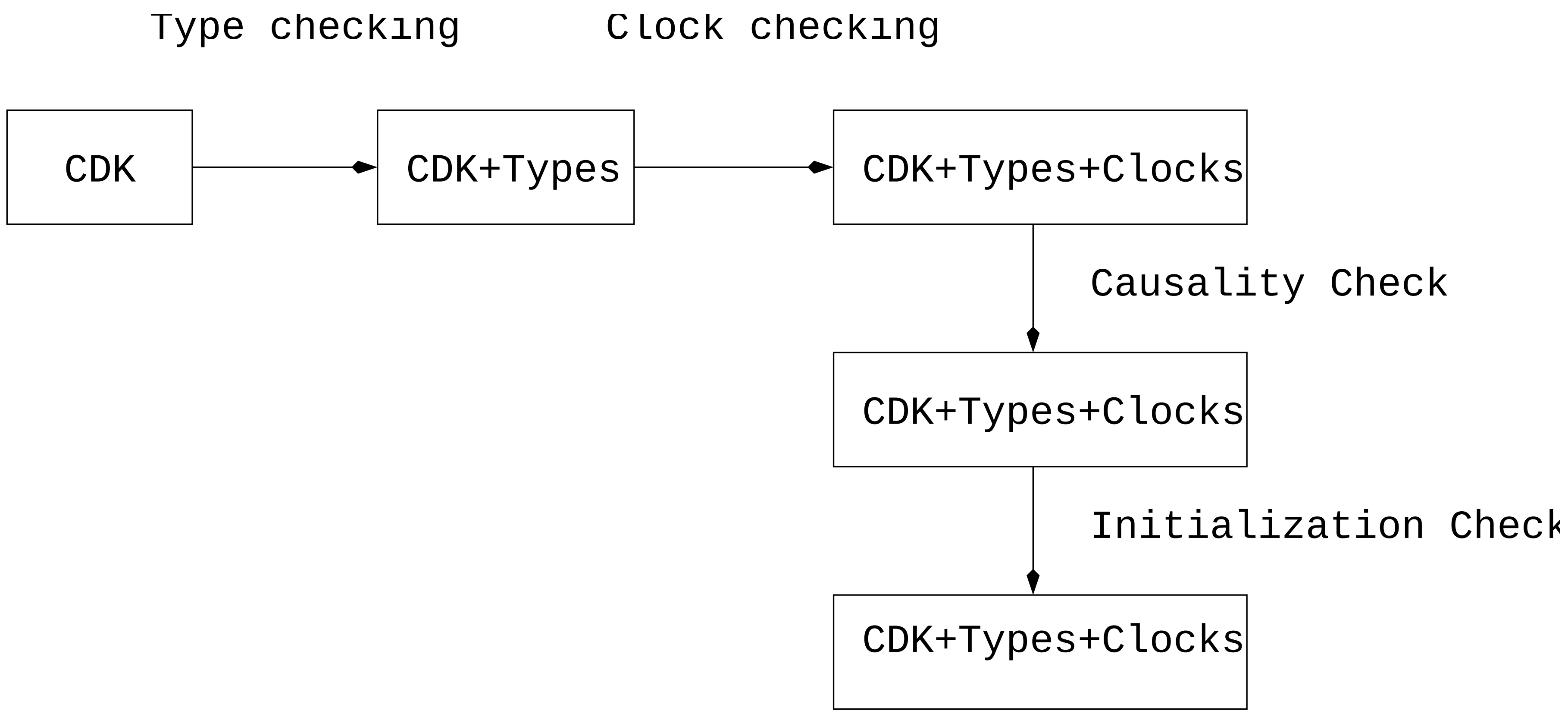
Proposal

- a compiler where everything can be “traced” with a precise semantics for every intermediate language
- introduce a basic **clocked** data-flow language as the input language
- general enough to be used as a input language for Lustre
- be a “good” input language for modern ones (e.g., mix of automata and data-flow as found in SCADE 6 or Simulink/StateFlow)
- provides a slightly more general notion of clocks
- and a reset construct
- compilation through an intermediate “object based” intermediate language to represent transition function
- provide a translation into imperative code (e.g., structured C, Java)

Organisation of the Compiler



Static Checking



A Clocked Data-flow Basic Language

A data-flow kernel where every expression is explicitely annotated with its clock

$$
\begin{array}{lcl}
a & ::= & e^{ct} \\
e & ::= & v \mid x \mid v \ \texttt{fby}\ a \mid a\ \texttt{when}\ C(x) \\
 & & \mid op\,(a, ..., a) \mid f\,(a, ..., a)\ \texttt{every}\ a \\
 & & \mid \texttt{merge}\ x\ (C \rightarrow a)\ ...\ (C \rightarrow a) \\
D & ::= & pat = a \mid D\ \texttt{and}\ D \\
pat & ::= & x \mid (pat, ..., pat)
\end{array}
$$

$$
\begin{array}{lcl}
d & ::= & \texttt{node}\ f(p) = p\ \texttt{with var}\ p\ \texttt{in}\ D \\
p & ::= & x : ty; ...; x : ty \\
v & ::= & C \mid i \\
ck & ::= & \textbf{base} \mid ck\ \textbf{on}\ C(x) \\
ct & ::= & ck \mid ct \times ... \times ct
\end{array}
$$

Informal Semantics

h	`True`	`False`	`True`	`False`	$\ldots$
x	x_0	x_1	x_2	x_3	$\ldots$
y	y_0	y_1	y_2	y_3	$\ldots$
v `fby` x	v	x_0	x_1	x_2	$\ldots$
$x + y$	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$\ldots$
$z = x$ `when` **`True`**(h)	x_0		x_2		$\ldots$
$t = y$ `when` **`False`**(h)		y_1		y_3	$\ldots$
`merge` h (**`True`** $\longrightarrow z$) (**`False`** $\longrightarrow t$)	x_0	y_1	x_2	y_3	$\ldots$

- z is at a slower rate than x. We say its clock is ck `on True`(h)
- the `merge` constructs needs its two arguments to be on complementary clocks
- statically checked through a dedicated type system (clock calculus)

Derived Operators

$$
\begin{aligned}
\texttt{if}\ x\ \texttt{then}\ e_2\ \texttt{else}\ e_3 &= \mathtt{\textbf{merge}}\ x \\
&\qquad (\mathtt{\textbf{True}} \to e_2\ \texttt{when}\ \mathtt{\textbf{True}}(x)) \\
&\qquad (\mathtt{\textbf{False}} \to e_3\ \texttt{when}\ \mathtt{\textbf{False}}(x))
\end{aligned}
$$

$$
\begin{aligned}
y = e_1\ \texttt{->}\ e_2 &= y = \texttt{if}\ \mathit{init}\ \texttt{then}\ e_1 \texttt{else}\ e_2 \\
&\quad \texttt{and}\ \mathit{init} = \mathtt{\textbf{True}}\ \texttt{fby}\ \mathtt{\textbf{False}}
\end{aligned}
$$

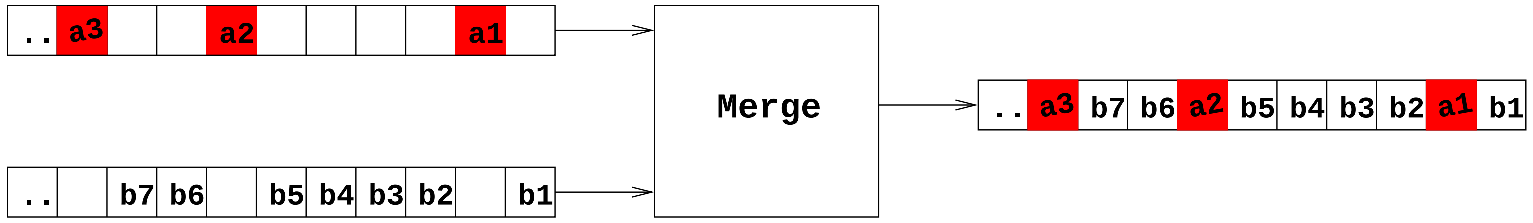
$$
\texttt{pre}\,(e) = \mathit{nil}\ \texttt{fby}\ e
$$

Example (counter)

```
node counting (tick:bool; top:bool) = (o:int) with
var v: int in
    o = if tick then v else 0 -> pre o + v
and v = if top then 1 else 0
```

N-ary Merge

`merge` combines two complementary flows (flows on complementary clocks) to produce a faster one:



Originaly introduced in Lucid Synchrone (1996), now in SCADE 6

Example: `merge c (a when c) (b whenot c)`

Generalization:

- can be generalized to n inputs with a specific extension of clocks with enumerated types
- the sampling e `when` c is now written e `when` $\texttt{True}(c)$
- the semantics extends naturally and we know how to compile it efficiently
- thus, **a good basic for compilation**

Reseting a behavior

- in Lustre, the “reset” behavior of an operator must be explicitely designed with a specific reset input

```
node count() returns (s:int);
let
 s = 0 fby s + 1
tel;

node resetable_counter(r:bool) returns (s:int);
let
  s = if r then 0 else 0 fby s + 1;
tel;
```

- painful to apply on large model
- propose a primitive that applies on node instance and allow to reset it

Modularity and reset

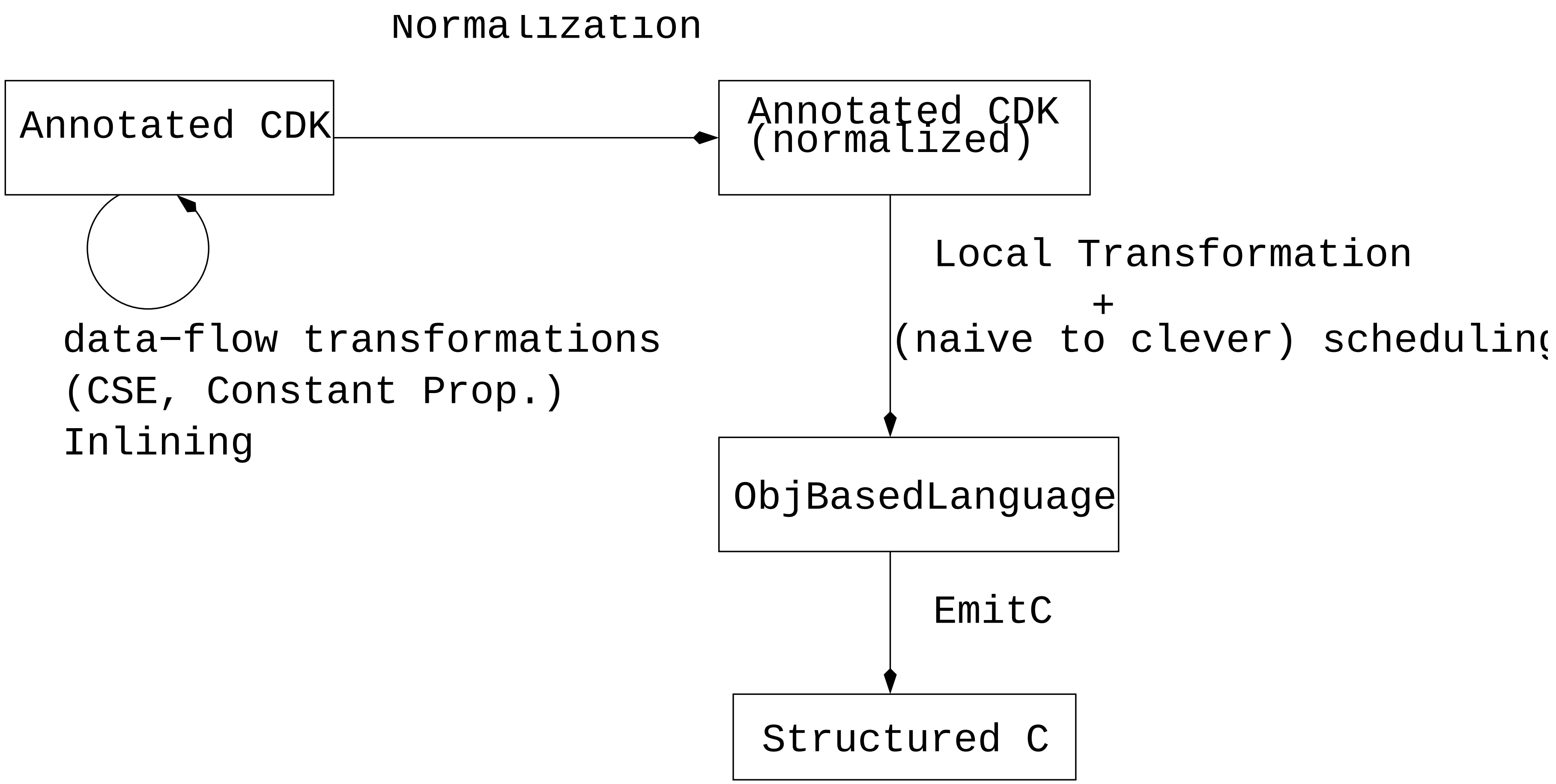
Specific notation in the basic calculus: $f\ (a_1, ..., a_n)$ `every` c

- all the node instances used in the definition of node f are reset when c is true
- the reset is “asynchronous”: no clock constraint between the condition c and the clock of the node instance

is-it a primitive construct? yes and no

- modular translation of the basic language with reset into the basic language without reset ([PPDP00], with G. Hamon)
- essentially a translation of the initialization operator `->`
- e_1 `->` e_2 becomes `if` c `then` e_1 `else` e_2
- very demanding to the code generator whereas it is trivial to compile!
- useful translation for verification tools, basic for compilation
- thus, **a good basic for compilation**

Translation



Putting Equations in Normal Form

- prepare equations before the translation
- extract delays from nested expressions by a linear traversal
- equations are transformed such that delays are extracted from nested computation.

Normal Form:

$$
\begin{array}{lcl}
a & ::= & e^{ck} \\
e & ::= & a\ \texttt{when}\ C(x) \mid op\,(a, ..., a) \mid x \mid v \\
ce & ::= & \texttt{merge}\ x\ (C \rightarrow ca)\ ...\ (C \rightarrow ca) \mid e \\
ca & ::= & ce^{ck} \\
eq & ::= & x = ca \mid x = (v\ \texttt{fby}\ a)^{ck} \\
 & & \quad \mid (x, ..., x) = (f\ (a, ..., a)\ \texttt{every}\, x)^{ck} \\
D & ::= & D\,\texttt{and}\,D \mid eq
\end{array}
$$

Example

$$
\begin{aligned}
z &= ((((4 \;\texttt{fby}\; o) * 3) \;\texttt{when}\; \mathtt{True}(c)) + k)^{ck \;\mathbf{on}\; \mathtt{True}(c)} \\
\mathbf{and}\; o &= (\mathtt{merge}\; c\; (\mathtt{True} \longrightarrow (5 \;\texttt{fby}\; (z+1)) + 2) \\
&\qquad (\mathtt{False} \longrightarrow ((6 \;\texttt{fby}\; x)) \;\texttt{when}\; \mathtt{False}(c)))^{ck}
\end{aligned}
$$

is rewritten into:

$$
\begin{aligned}
z &= (((t_1 * 3) \;\texttt{when}\; \mathtt{True}(c)) + k)^{ck \;\mathbf{on}\; \mathtt{True}(c)} \\
\mathbf{and}\; o &= (\mathtt{merge}\; c\; (\mathtt{True} \longrightarrow t_2 + 2) \\
&\qquad (\mathtt{False} \longrightarrow t_3 \;\texttt{when}\; \mathtt{False}(c)))^{ck} \\
\mathbf{and}\; t_1 &= (4 \;\texttt{fby}\; o)^{ck} \\
\mathbf{and}\; t_2 &= (5 \;\texttt{fby}\; (z+1))^{ck \;\mathbf{on}\; \mathtt{True}(c)} \\
\mathbf{and}\; t_3 &= (6 \;\texttt{fby}\; x)^{ck}
\end{aligned}
$$

Intermediate Language

$$
\begin{array}{rcl}
d & ::= & \texttt{class}\ f = \\
 & & \langle \texttt{memory}\ m, \\
 & & \quad \texttt{instances}\ j, \\
 & & \quad \texttt{reset}()\ \texttt{returns}\ () \ = \ S, \\
 & & \quad \texttt{step}\,(p)\ \texttt{returns}\,(p) = \texttt{var}\ p\ \texttt{in}\ S\rangle \\
S & ::= & x \ := \ c \mid \texttt{state}\,(x) \ := \ c \mid S\,;\,S \mid \texttt{skip} \\
 & & \qquad \mid o.\texttt{reset} \mid (x, ..., x) = o.\texttt{step}\,(c, ..., c) \\
 & & \qquad \mid \texttt{case}\ (x)\ \{C : S; ...; C : S\} \\
c & ::= & x \mid v \mid \texttt{state}\,(x) \mid op(c, ..., c) \\
v & ::= & C \mid i \\
j & ::= & o : f, ..., o : f \\
p, m & ::= & x : t, ..., x : t
\end{array}
$$

Intermediate Language

- the minimal need to represent transition functions
- we introduce an ad-hoc intermediate language to represent them
- it has an “object-based” flavor (with minimal expressiveness nonetheless)
- static allocation of states only
- it can be trivially translated into a imperative language
- we only need a subset set of C (functions and static allocation of structures, very simple pointer manipulation)

Principles of the translation

- hierarchical memory model which corresponds to the call graph: one local memory for each function call
- every function defines a machine (a "class")
- the translation is made by a linear traversal of the sequence of normalized and scheduled equations
- control-structure (invariant): a computation on clock ck is executed when ck is true
- a guarded equations $x = e^{ck}$ translates into a control-structure

  E.g., the equation:

  $$x = (y+2)^{\texttt{base on } C_1(x_1) \texttt{ on } C_2(x_2)}$$

  is translated into a piece of control-structure:

  $$\texttt{case } (x_1)\ \{C_1 : \texttt{case } (x_2)\ \{C_2 : x = y + 2\}\}$$

Clock-directed code generation

- local generation of a control-structure from a clock

$$\begin{array}{lcl} Control(\texttt{base}, S) & = & S \\ Control(ck \texttt{ on } C(x), S) & = & Control(ck, \texttt{case } (x)\ \{C : S\}) \end{array}$$

- merge them locally

$$\begin{array}{l} Join(\texttt{case } (x)\ \{C_1 : S_1; ...; C_n : S_n\}, \\ \qquad \texttt{case } (x)\ \{C_1 : S'_1; ...; C_n : S'_n\}) \\ \quad = \texttt{case } (x)\ \{C_1 : Join(S_1, S'_1); ...; C_n : Join(S_n, S'_n)\} \\ Join(S_1, S_2) = S_1; S_2 \end{array}$$

- control-optimization: find a static schedule of the equations which gather equations guarded by related clocks

Example (modularity)

- each function is compiled separately
- a function call needs a local memory

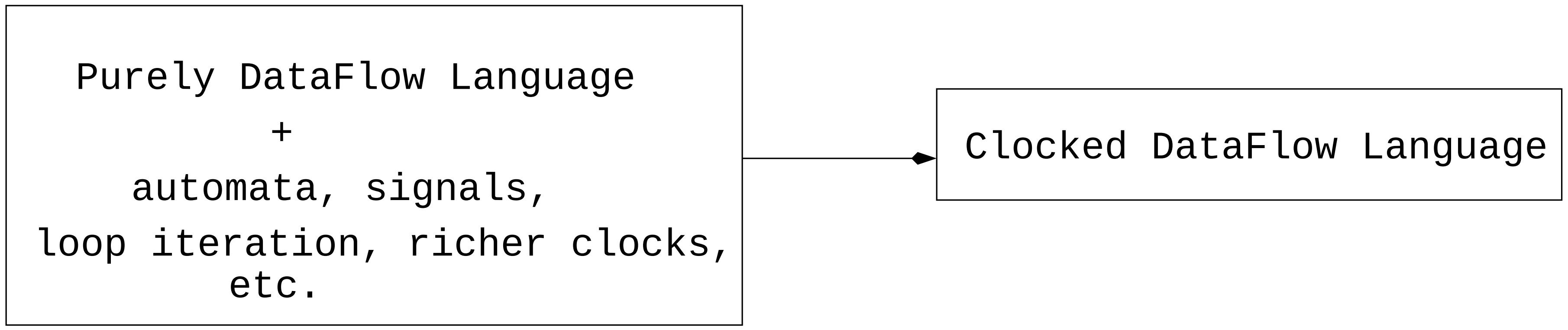
```
node count(x:int) returns (o:int)
let
  o = 0 fby o + x;
tel

node condact(c:bool;input:int) returns (o:int)
var v: int;
let
  v = count(input when true(c));
  o = merge c (true -> v)
              (false -> (0 fby o) when false(c));
tel
```

```
class condact {
  x_2 : int; x_4 : count;

  reset() {
    x_4.reset();
    mem x_2 = 0;
  }

  step(c : bool; input : int) returns (o : int) {
    x_3 : int;

    switch (c) {
      case true :
        (x_3) = x_4.step(input);
        o = x_3;
      case false :
        o = mem(x_2);
    };
    mem x_2 = o; }
```

MiniLustre in Numbers

Administrative code	Abstract syntax + printers	546
	Lexer&Parser	335
	main (misc, symbol tables, loader)	285
Basis	graph	74
	scheduling	67
	type checking	269
	clock checking	190
	causality check	30
	normalization	95
	translate (to ob)	136
Emitters to concrete languages	(C, Java and Caml)	arround 300 each
Optimizations	Inline + reset	250
	Dead-code Removal	42
	Data-flow network minimization	162

Extensions (towards a full language)



- extend the source language with new programming constructs
- translation semantics into the basic data-flow language
- this is essentially the approach taken in Lucid Synchrone and the ReLuC compiler of SCADE
- clocks play a central role
- simple and gives very good code
- reuse of the existing code generator (adequate in the context of a certification process)

Formal Certification (Coq development)

In parallel, we have implemented MiniLustre in the programming language of Coq + parts in Caml

- some part are directly programmed in Coq (2000 loc): E.g., translation function, checking clock/type annotation
- others (administrative) or less structural (e.g., scheduling) kept in Caml
- type and clock inference also done (4000 loc of Coq)

The semantics of the source and target with proof of equivalence between the two are done on paper (one year to find the adequate combination).

Proofs in Coq are under way

Conclusion

Results

- a minimal description of modular code-generation
- formalize and simplifies the compilation techniques we have developped for the ReLuC compiler (with Esterel-Technologies) used for SCADE 6

Current

- Coq development of the semantics preservation
- finish the Coq programming of the reference compiler (combines **translation validation**, e.g., for the clever scheduling of equations) and **certified compilation**)

Future (longer term)

- mixed systems (data-flow systems + mode-automata)
- source-to-source transformation into the data-flow system
- translation semantics (as done in ReLuC [EMSOFT'05, EMSOFT'06])