WiP. A Co-Iterative Synchronous Interpreter

Guillaume Baudart,[*] Jean-Louis Colaço,[†] Louis Mandel,[*] Michael Mendler,[‡] Marc Pouzet[§]
[*]*IBM Research*, USA, [†]*ANSYS SBU*, France, [‡]*Otto-Friedrich-Universität Bamberg*, Germany
[§]*École normale supérieure, PSL Research University*, France

***Abstract*—Synchronous languages are routinely used to program industrial embedded controllers. These domain specific languages offer a mathematically precise semantics close to the block diagram formalism used by engineers, and reliable code generation. The semantics of these languages is typically defined on a kernel language. High-level constructs are then compiled into this kernel via a series of analyses and source-to-source transformations.**

In this paper, we describe a direct executable co-iterative semantics for dataflow synchronous programs that is the basis of an interpreter for a language that mixes data-flow equations and hierarchical automata à la Scade 6. This semantics can be used as a reference to prove properties of a program, test the compiled code, or experiment with language extensions at the specification level.

I. Introduction

Synchronous dataflow languages were introduced to ease the design of embedded systems. In these languages, computations are paced on a discrete global clock. A programmer writes high level specifications in the form of stream functions specifying variable values at each step.

Synchronous languages offer a mathematically precise semantics reminiscent of block diagrams, a popular formalism to describe control systems; specialized test and verification tools [1]; and optimized code generation [2]. Industrial synchronous languages such as Scade [3] are now routinely used to program safety critical embedded controllers, e.g., train braking system, or aircraft fly-by-wire.

Starting with the seminal Lustre paper [4] numerous formal semantics for synchronous dataflow languages have been proposed. However, these semantics either focus on a small language kernel and define high-level constructs, e.g., automata by compilation to the kernel [5], [6]; or the semantics of the full language is defined by predicates that are not executable [7], [8]. *Can we define a direct and executable semantics for a full language that also gives a reference interpreter?*

In this paper we consider a synchronous data-flow language where, as in Scade, data-flow equations and hierarchical automata can be arbitrarily mixed (Section II). We propose a direct denotational semantics based on a co-iteration interpretation of streams [9], [10]: a stream is characterized by an initial state and a transition function (Section III). We then discuss the future directions and potential benefits (Section IV).

Our work is closely related to [10]. However, we propose a deep embedding of the language that does not rely on the host language semantics. The semantics is thus simpler, and closer to the specifications. Compared to [11], the semantics includes for the first time the treatment of Scade like hierarchical automata.

```
 1  let node count_mod (n) returns (cpt)
 2    cpt = 0 -> (pre cpt + 1) mod n
 3
 4  let node edge (b) returns (o)
 5    o = b && (true -> not pre b)
 6
 7  let node chrono (stst, rst)
 8    returns (s init 0, m init 0)
 9    reset
10      automaton
11        | START -> local d
12            do d = count_mod 100
13            and if edge (d = 0) then s = count_mod 60
14            and if edge (s = 0) then m = count_mod 60
15            unless stst continue STOP
16        | STOP ->
17            do
18            unless stst continue START
19      end
20    every rst
```

Fig. 1. A simple chronometer.

II. Language: Example

Figure 1 illustrates the language with a simple chronometer (using Zelus syntax). The *node* `chrono` is a stream function from the boolean input streams `stst` (*start/stop*) and `rst` (*reset*) to the output streams `s` (*seconds*) and `m` (*minutes*). The outer `reset` construct (line 9-20) resets its body when the `rst` button is pressed (line 20). The inner automaton (line 10-19) alternates between the states `START` and `STOP` when the `stst` button is pressed (line 13 and 18).

In state `START` three mutually recursive equations increment the minutes and seconds counters (line 12-14) using a local variable `d`. In state `STOP`, the controller does nothing: `s` and `m` implicitly keep their last values because they are declared as initialized memories (`init` keyword line 8).

The node `counter` implements a counter modulo `n` using the unit delay operator `pre` and the initialization operator `->` (line 2). The node `edge` implements a boolean edge detector.

III. Co-Iterative Semantics

We now give the intuition of the co-iterative semantics. A *stream* of type T is defined by an initial state of type S and a transition function of type $S \to T \times S$. Repeatedly executing the transition function from the initial state yields a stream of values of type T. *Nodes* are length preserving stream processors, i.e., at each instant a node only requires the current value of its input to produce an output.

$$\begin{aligned} \mathit{CoStream}(T,S) &= (S \to T \times S) \times S \\ \mathit{CoNode}(T,T',S) &= (S \to T \to T' \times S) \times S \end{aligned}$$

$$
\begin{aligned}
&\llbracket \texttt{pre}\ e \rrbracket_\rho^{\mathit{Init}} = (\mathit{nil}, \llbracket e \rrbracket_\rho^{\mathit{Init}})\\
&\llbracket \texttt{pre}\ e \rrbracket_\rho^{\mathit{State}} = \lambda(m, s).m, \llbracket e \rrbracket_\rho^{\mathit{State}}(s)
\end{aligned}
$$

$$
\begin{aligned}
&\llbracket \texttt{reset}\ e_1\ \texttt{every}\ e_2 \rrbracket_\rho^{\mathit{Init}} = (\llbracket e_1 \rrbracket_\rho^{\mathit{Init}}, \llbracket e_1 \rrbracket_\rho^{\mathit{Init}}, \llbracket e_2 \rrbracket_\rho^{\mathit{Init}})\\
&\llbracket \texttt{reset}\ e_1\ \texttt{every}\ e_2 \rrbracket_\rho^{\mathit{State}} = \lambda(s_i, s_1, s_2).\\
&\quad \mathit{let}\ v_2, s_2' = \llbracket e_2 \rrbracket_\rho^{\mathit{State}}(s_2)\ \mathit{in}\\
&\quad \mathit{let}\ v_1, s_1' = \llbracket e_1 \rrbracket_\rho^{\mathit{State}}(\mathit{if}\ v_2\ \mathit{then}\ s_i\ \mathit{else}\ s_1)\ \mathit{in}\ v_1, (s_i, s_1', s_2')
\end{aligned}
$$

$$
\begin{aligned}
&\llbracket \texttt{present}\ e\ \texttt{->}\ e_1\ \texttt{else}\ e_2 \rrbracket_\rho^{\mathit{Init}} = (\llbracket e \rrbracket_\rho^{\mathit{Init}}, \llbracket e_1 \rrbracket_\rho^{\mathit{Init}}, \llbracket e_2 \rrbracket_\rho^{\mathit{Init}})\\
&\llbracket \texttt{present}\ e\ \texttt{->}\ e_1\ \texttt{else}\ e_2 \rrbracket_\rho^{\mathit{State}} = \lambda(s, s_1, s_2).\\
&\quad \mathit{let}\ v, s' = \llbracket e \rrbracket_\rho^{\mathit{State}}(s)\ \mathit{in}\\
&\quad \mathit{if}\ v\ \mathit{then}\ \mathit{let}\ v_1, s_1' = \llbracket e_1 \rrbracket_\rho^{\mathit{State}}(s_1)\ \mathit{in}\ v_1, (s, s_1', s_2)\\
&\qquad\quad \mathit{else}\ \mathit{let}\ v_2, s_2' = \llbracket e_2 \rrbracket_\rho^{\mathit{State}}(s_2)\ \mathit{in}\ v_2, (s, s_1, s_2')
\end{aligned}
$$

$$
\begin{aligned}
&\llbracket \texttt{do}\ x_1\ \texttt{=}\ e_1\ \texttt{and}\ x_2\ \texttt{=}\ e_2\ \texttt{done} \rrbracket_\rho^{\mathit{Init}} = (\llbracket e_1 \rrbracket_\rho^{\mathit{Init}}, \llbracket e_2 \rrbracket_\rho^{\mathit{Init}})\\
&\llbracket \texttt{do}\ x_1\ \texttt{=}\ e_1\ \texttt{and}\ x_2\ \texttt{=}\ e_2\ \texttt{done} \rrbracket_\rho^{\mathit{State}} = \lambda(s_1, s_2).\\
&\quad \mathit{let}\ f_s = \lambda(v_1, v_2).\ \llbracket (e_1, e_2) \rrbracket_{\rho+[v_1/x_1, v_2/x_2]}^{\mathit{State}}(s_1, s_2)\ \mathit{in}\\
&\quad \mathit{let}\ (v_1, v_2), (s_1', s_2') = \mathit{fix}\ (f_s)\ \mathit{in}\ [v_1/x_1, v_2/x_2], (s_1', s_2')
\end{aligned}
$$

Fig. 2. Excerpt of the co-iterative semantics.

An excerpt of the semantics is given in Figure 2.

Expressions: The semantics of an expression e is defined using two auxiliary functions. If ρ is an environment mapping values to variable name, $\llbracket e \rrbracket_\rho^{\mathit{Init}}$ denotes the initial state, and $\llbracket e \rrbracket_\rho^{\mathit{State}}$ denotes the transition function. For instance, the state of the unit delay operator `pre` e contains the state of the sub-expression e and the value at the previous step initialized with nil (any value of the correct type). At each step, the output is the previous value stored in the state.

Control structures: `reset` e_1 `every` e_2, is equivalent to a single-state automaton. In addition to the state of the two sub-expressions of `reset` e_1 `every` e_2, the state also stores the initial state of e_1. When the reset condition e_2 evaluates to *true*, we reinitialize the state of e_1 with the copy of the initial state. `present` e `->` e_1 `else` e_2 executes one of its branch depending on the activation condition.

Equations: Following denotational semantics of functional languages, sets of mutually recursive equations are interpreted using a fix-point operator. To define this fix-point operator, we assume that all functions are total on the CPO $T_\perp = T + \perp$, where $\perp$ is the bottom absorbing value: $\forall v : T_\perp,\ v > \perp$. We then write $\mathit{fix}\ (f_s) : T_\perp \times S$ the smallest fix-point of a transition function with a fixed input state $f_s : T_\perp \to T_\perp \times S$, i.e, the smallest solution of $X_s = (\mathit{let}\ v, s' = X_s\ \mathit{in}\ f_s(v))$:

$$\mathit{fix}\ (f_s) = \mathit{let\ rec}\ v, s' = f_s(v)\ \mathit{in}\ v, s'$$

Using this fix-point operator, as an example, Figure 2 gives the semantics of two mutually recursive equations `do` x_1 `=` e_1 `and` x_2 `=` e_2 `done` which returns a environment defining x_1 and x_2 and the updated state for e_1 and e_2.

IV. Research Plan

The semantics described in Section III is the basis of an interpreter written in OCaml. This interpreter: 1) gives a reference semantics for the language, 2) can be used as a debugging tool, and 3) can be extended to prototype new language features.

Reference Semantics: A program is *causal* if it produces streams without $\perp$. The fix-point based semantics is by construction modular: we do not lose causality by node abstraction. In practice, the causality analysis of the compiler relies on a conservative syntactic criterion: *all dependency cycles must be broken by a unit delay*, which may require inlining node applications. The semantics can thus be used as a reference to precisely characterize the subset of causal programs that are accepted by the compiler.

For instance, for the following program [12], the Scade compiler returns `Causality error: the definition of flow x2 depends on flow y1; the definition of flow y1 depends on flow x1; the definition of flow x1 depends on flow y2; the definition of flow y2 depends on flow x2;`.

```
let node mux(c, x, y) returns (o)
  if c then o = x else o = y

let node fog_gof(c, x) returns (y)
  local x1, x2, y1, y2
  do  x1 = mux(c, x, y2) and y1 = f(x1)
  and x2 = mux(c, y1, x) and y2 = g(x2)
  and  y = mux(c, y2, y1) done
```

But this program can be safely executed with our interpreter since the `mux` operator prevents cycles at runtime.

Dynamic Debugging: Static analysis errors returned by the compiler can be difficult to understand, especially for programs comprising many hierarchical automata running in parallel. The interpreter can be used as a debugging tool to execute failing programs to understand precisely when the program failed, or identify problematic variable values and mode transitions.

Language features: Finally, the core of the interpreter is about $1,000$ LoC and can be extended to prototype new language features. For instance adding support for arrays and arrays iterators, static arguments (evaluated only once during instantiation), or external functions with side-effects.

References

[1] A. Champion, A. Mebsout, C. Sticksel, and C. Tinelli. The kind 2 model checker. In *CAV*. Springer, 2016.

[2] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *PLILP*. Springer, 1991.

[3] J.-L. Colaço, B. Pagano, and M. Pouzet. SCADE 6: A formal language for embedded critical software development (invited paper). In *TASE*. IEEE, 2017.

[4] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: A declarative language for programming synchronous systems. In *POPL*. ACM, 1987.

[5] J.-L. Colaço, B. Pagano, and M. Pouzet. A conservative extension of synchronous data-flow with state machines. In *EMSOFT*. ACM, 2005.

[6] G. Baudart, L. Mandel, E. Atkinson, B. Sherman, M. Pouzet, and M. Carbin. Reactive probabilistic programming. In *PLDI*. ACM, 2020.

[7] T. Bourke, L. Brun, P.-É. Dagand, X. Leroy, M. Pouzet, and L. Rieg. A formally verified compiler for lustre. In *PLDI*. ACM, 2017.

[8] T. Bourke, L. Brun, and M. Pouzet. Mechanized semantics and verified compilation for a dataflow synchronous language with reset. In *POPL*. ACM, 2020.

[9] C. Paulin-Mohring. Circuits as streams in coq: Verification of a sequential multiplier. In *TYPES*. Springer, 1995.

[10] P. Caspi and M. Pouzet. A co-iterative characterization of synchronous stream functions. In *CMCS*. Elsevier, 1998.

[11] G. Hamon. A denotational semantics for stateflow. In *EMSOFT*. ACM, 2005.

[12] S. Malik. Analysis of cyclic combinational circuits. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 13(7), 1994.