An Overview of Synchronous Languages

Olivier Hermant

CRI, MINES ParisTech

13 Octobre 2014
Synchronous Languages

- Reactive programming:
  - from the *inside*, computation of a *reaction* to (external) stimuli
  - from the *outside*: immediate reaction to events

- A central notion:

Clocks

A clock is a succession of instants (ticks) that abstract time both for computation and environment:

1. computations are immediate *within* a tick
2. a stimulus can occur only at a tick

- Impact:
  - on the *design* of the language: control the primitives
  - check the Worst-Case Execution Time (blueWCET)
  - everything is *synchronous*: parallelism is real parallelism
Some synchronous languages: Esterel, Lustre, Lucid-synchrone ...

... and Faust!

- clocked: rather obvious
- the “synchronous abstraction”: everything is computed at every sample, instantly
- WCET $\sim$ length of the longest path (after delays have been disconnected)

This presentation: a survey of the domain, and its (potential) use for Faust.
Clocks

Clocks are usually represented as an infinite sequence of 0 and 1 (1 ≡ tick):

\[
\text{clk1} := 010010101001010100101\ldots \\
\text{clk2} := 101010101010101010101\ldots
\]

- this assumes the existence of a “root” clock. It can be irregular wrt physical time (abstracted).

This allows different clocks for different events [support for multirate].

Clocks can be:
- **periodic** (or ultimately periodic)
- **aperiodic**: in this case, we nevertheless have some hypothesis on the events (maximum rate, ...) [User events]

Explicit manipulation of clocks:
- negating, prefixing, filtering, ...
- communications may involve buffers
- need to check *causality*: no instantaneous feedback, clock precedence
This kind of language (Zélus) includes continuous functions:

▶ expression of the physical plant inside the language,
▶ through (ordinary) differential equations,
▶ an event is a zero-crossing of the modelized continuous function.
▶ involved semantics:
  ★ through nonstandard analysis: “infinitely small” is the time of response to events (computations).
  ★ operational semantics: simulated (coupled with a numerical solver) and executed (plant replace by sensors)

▶ or how to have a caterpillar in a paper ...
▶ Is this relevant for Faust ?
Clock Calculi
A. Bouakaz and JP. Talpin, *Design of safety-critical java level 1 applications using affine abstract clocks.*

Multiple clocks and operating on them - multiple problems:
- clock polymorphism ($\alpha$ on (10))
- compatibility between clk1 and clk2: subtyping relation
  - the end rate is the same
  - the 1s of clk1 arrive before clk2
- approximations through abstract interpretation (clock envelopes)
- (bounded) buffer calculation
- schedule of clocked tasks

In Faust:
- simplify clock relation, even in multirate
- can we think about use cases where the clock relations are not elementary?
- the semantics of the delay, no reasoning on clocks themselves
Multiprocessor Compilation

1. A. Bouakaz and JP. Talpin, *Design of safety-critical java level 1 applications using affine abstract clocks*.

2. G. Delaval, A. Girault, and M. Pouzet, *A Type System for the Automatic Distribution of Higher-order Synchronous Dataflow Programs*.

How to distribute computation on multiple targets:

1. compute a compatible **schedule** (given capacities of processors and WCETs).

2. **explicitely give locations** where each piece of code must be executed:
   * multiple processor,
   * multiple sensors or [devices].
Formalization
S. Boulmé and G. Hamon, A clocked denotational semantics for lucid-synchrone in COQ.

Derive formally some properties:
▶ on the programming language itself: semantics
▶ on the programs: semantics and proofs

Expressed in some proof assistant (Coq). For Faust [Emilio’s talk]:

\[
E[I]r = r(I) \\
E[E_1 : E_2]r = p_2 \circ p_1, \text{ with } p_i = E[E_i]r \\
E[E_1, E_2]r = \lambda m.p_1(m[1, d_1])\|p_2(m[d_1 + 1, d_1 + d_2]) \\
E[E_1 <: E_2]r = \lambda m.p_2(\|_{1,d_2,d_1,i}.\lambda i.p_1(m)) \\
E[E_1 :> E_2]r = \lambda m.p_2(\|_{1,d_2,i}.\lambda i.\text{mix}(p_1(m)[i, d_1', d_2]))
\]

where \(\text{mix}(m') = (m')\), if \(|m'| = 1\) and
\[
\text{mix}(m') = E[+]r(m'[1, 1]\|\text{mix}(m'[2, |m'|])), \text{ if } |m'| > 1
\]

\[
E[>]r = \lambda m.(\lambda t \in \bigcap_{i \in [1,|m|]} \text{dom}(m[i]).\lambda I \in .m^{-1}(I))(t) \\
E[<']r = \lambda (s).\|_i.\lambda I'.\lambda t.s(t)(I') \\
E[E_1 \sim E_2]r = \lambda m.\text{fix}(\lambda m'.p_1(p_2(@m'[1, d_2]))\|m))
\]

where \(@m) = \|_m \lambda s.\text{delay}(s, \lambda t.1)\)
Automation of proofs

Another way: high-level automated Hoare’s logic.

▶ express properties on the code
▶ automatically generate a proof

What do we need:

▶ A **language** for properties, provided as comments (annotations),
▶ A **framework** on which to express properties:
  ★ a formal semantics
  ★ or traces of a program,
▶ a **library** of basic lemmata and tactics in the proof assistant,
▶ an **automatic generation tool** (small, dedicated ATP).
- WCET computation: can it breaks the “synchronous abstraction”?
  - in an abstract way (capacity exceeded)
  - in a hard-time way
- does this work concern only core Faust?