Automated Clock Inference for Stream Function-Based System Level Specifications

Jean-Pierre Talpin[1] and Sandeep Kumar Shukla[2]

[1]INRIA-IRISA and [2]Virginia Tech

Abstract

Among system-level design frameworks and languages, system modeling approaches based on functional programming aim at rapid specification and prototyping of processing of data streams. Models constructed with functional programs provide highly modular and computationally correct models for prototyping and simulation purposes.

For design space exploration, the functional programming frameworks provide semantically sound methodologies for establishing formal refinement relations and secure a trustable design automation flow. However, the stream computations expressed by functions embody an implicitly single clocked model of computation (untimed or fully synchronous) which does not take advantage of possible polychronous (multi-clock) computation inherent to the system's dataflow.

To this end, we propose a type inference system for representing a synchronous and multi-clocked model of computation in the typed and functional programming language ML. Along the way, we address the issue of performing the automated refinement of implicitly timed stream functions in a model of computation that supports reasoning on partially ordered signal clocks, allowing for formal design transformation and verification to be performed in the context of a functional programming environment.

1. Introduction

Synchronous data-flow programming languages, such as Lustre [2] and Signal [6], are domain-specific languages dedicated to the implementation of embedded software. These programming languages implement a synchronous model of computation in which time is abstracted to facilitate reasoning and verification of symbolic synchronization and scheduling relations. Synchronous data-flow languages are commonly used in Europe for embedded software design, especially in avionics, to rapidly prototype, simulate, verify and synthesize embedded software for mission critical applications. As similar needs for rapid prototyping and simulation as well as system-level verification and synthesis have been expressed in the design automa-

tion industry, it is desirable to propose such a model in the context of system-level design and raise both its expressive capability and ease of use.

On the other hand, in the context of multi-media embedded systems data stream based computation is important. Therefore, several proposals [10, 7] have recently been made to provide functional programming capabilities to system-level environments geared towards rapid prototyping and simulation of mainly stream based computations inherent in multi-media. Functional programming framework both provide the necessary sound semantic framework to formally reason on system modeling, and significantly raise design productivity by offering programming environments in which error-prone and time-consuming engineering tasks such as type consistency and memory management [13] are automatically handled by the compiler.

Multi-media computation architectures modeled with functional programming concepts allow to rapidly obtain prototyping and simulation evidences and, design space exploration and/or of design refinement checking.

However, functional programming frameworks used so far for capturing such models [10], [7] are not capable of expressing multi-clocked computations. The model of computation expressed in [10] is perfectly synchronous [5], and in [7], untimed, perfect and clock synchronous as well as explicitly timed [5]. The ability to automatically recognize the independence of computation fragments from each other, and therefore the ability to assign distinct clocks to these computations can have both performance implications, as well as other resource scheduling implications.

To this end, we propose a type inference system for representing a synchronous and multi-clocked model of computation in the typed and functional programming language ML. Along the way, we address the issue of performing an automated refinement of implicitly timed stream functions in a model of computation that supports reasoning on partially ordered signal clocks, allowing for formal design transformation and verification to be performed in the context of a functional programming environment.

2. Background

We start with the definition of the elementary Standard ML data-structures along which the presentation of our model of computation will be carried out. Our aim is to provide a a implicitly timed simulation environment that implements a multi-clocked synchronous model of computation in plain Standard ML. An extension to the Standard ML type inference system is defined to analyze programs and statically check conformance to design correctness properties.

Events and signals The data-type `event` is defined by two rules. It is either absent and carries the symbolic value `nil` of present and carries a value of some type `'a` tagged by the constructor `one`. A signal consists of a series of event. It can either be represented by a list of event, that is either empty and denoted by `[]` or non-empty. The head of a non-empty signal is an event of type `'a` and its tail a signal of type `'a`.

```
datatype 'a event  = nil | one of 'a
datatype 'a signal =
    [] | 'a event :: 'a signal
```

It might be suitable to define possibly infinite signal co-inductively, as in Haskell for instance. This can alternatively be done by considering a delayed function call to access the tail of a signal and implement a lazy evaluation semantics. A lazy signal of type `'a` consists of an event of type `'a` (its head) and of a function that takes no argument (type `unit`) and returns a signal of type `'a` (the tail of the signal). Its signature is then `datatype 'a signal = [] | 'a event :: (unit -> 'a signal)`. For the sake of simplicity, our presentation will be illustrated by examples on inductive signals.

Signals with explicit clocks Assigning clocks to signals consists of a refinement of the data-type definitions for signals with extra information carried by supplementary type variables. An event or a signal of type `'a` is consistently related to a variable `'b` which symbolically denotes its clock and is used to relate it to other events or signals.

```
datatype 'a 'b event  = nil | one of 'a
datatype 'a 'b signal =
    [] | 'a 'b event :: 'a 'b signal
```

Multi-clocked sampling For instance, consider a definition of the down-sampling function `when`. The function `when` takes two signals of type `'a` and of type `boolean`. It returns a signal of type `'a` that is a sampling of events from the first argument when the second argument is present and holds the value true. The function `when` is defined by four rules. The first rules says that if both signals are empty then so is the result of the function. The second rule says that if both signals are absent (their heads carry the symbolic value `nil`) then the result is absent and the function is recursively called with the tails `s` and `t` as arguments. The third rule says that if the second arguments is present with the value false, pattern `one false`, then the result is as for rule two whatever the value of the first argument is (denoted by the wildcard _). Finally, the fourth rule says that if the first argument is present with some value `v` and the second holds the value true with `one true` then the event `one v` is copied to the output signal.

```
fun when [][]= []
  | when nil::s _::t = nil::when s t
  | when _::s one false::t = nil::when s t
  | when one v::s one true::t =
      one v::when s t
```

Our explanations on the definition of function `when` involved logical timing relations between signals which we can put into equations by considering the type of function `when` annotated with clocks:

```
fun when: 'a 'b signal * boolean 'c signal
    -> 'a 'd signal
```

Analyzing clock relations Now, let us denote by `@b` the presence of an event along the signal associated to the clock variable `'b`. Next write c and `not c` the presence of the value true and false along the signal of clock `'c`. Finally let `or, and, =>, <=>` be logical connectors. Each rule in the definition of `when` corresponds to an implication as to the presence or absence of event along the output signal:

- rule 2: not $@b$ and not $@c$ implies not $@d$
- rule 3: not c implies not $@d$
- rule 4: $@b$ and c implies $@d$

As a result, the type of function `when` can be associated to the assertion `@d <=> @b and c` which says that the result of clock `'d` is present iff the first argument of clock `'b` is present and the second is present holds the event true. It will be handy to define a particular class of boolean events in order to represent control by using assertions.

```
datatype 'a bool = boolean 'a event
val true  : 'a bool / a
val false : 'a bool / not a
```

Generalization of the approach Using clocked boolean events allows us to operate the same analysis on the definition of boolean functions and assign them to assertions describing their behaviors.

```
fun not: 'a bool -> 'b bool / b <=> not a
fun and: 'a bool -> 'b bool -> 'c bool
    / c <=> a and b
fun or : 'a bool -> 'b bool -> 'c bool
    / c <=> a or b
```

3. Formal presentation

The formalization of our logic for reasoning on clocks consists of an extension of the Standard ML type system with assertions on signal clocks. The abstract syntax of Standard ML which we consider for this formalization consists of atomic values and patterns, constructors, rules and expressions.

Atoms x are either defined identifiers or pre-defined constructors such as the wildcard _, the absence nil, the empty list [], the boolean values true and false and constructors one for events, $(\cdot,\cdot)$ for pairs and :: for signals. A pattern p is either an atom or a constructor x applied to a possibly empty sequence of patterns p^*.

A rule $p = e$ associates the input pattern p of a function to the expression e and is composed with |. An expression e is either an atom x, a function x defined by a rule r, the application of a function e to a sequence e^* of arguments or the binding of a rule r to the lexical scope of e.

$$
\begin{array}{ll}
x \quad \ni \{\text{true}, \text{false}, \text{nil}, \text{one}, ::, [], (\cdot,\cdot), \dots\} & \text{(id)} \\
p ::= x \mid x\,p^* & \text{(pattern)} \\
r ::= p = e \mid (r\,|\,r) & \text{(rule)} \\
e ::= x \mid \text{fun}\,x\,r \mid e\,e^* \mid \text{let}\,r\,\text{in}\,e & \text{(expression)}
\end{array}
$$

Next, we augment the Standard ML type system with datatypes and assertions to reason on signal clock relations. A type either denotes a variable $'a$, a pair $t * u$, a function $t \rightarrow u/C$ of argument type t, of result type u and of assertions C. Clocked booleans, events and signals are our predefined data-types.

$$
\begin{array}{llll}
t,u ::= {}'a & \text{(variable)} & \mid {}'a\,\text{bool} & \text{(boolean)} \\
\mid t * u & \text{(pair)} & \mid t'a\,\text{event} & \text{(event)} \\
\mid t \rightarrow u/C & \text{(function)} & \mid t'a\,\text{signal} & \text{(signal)}
\end{array}
$$

Assertions C are defined by relations between clock expressions c expressed using a Boolean logic to characterize invariants. A clock expressions is either 0 to mean never, 1 to mean always, $@a$ to denote the presence of a signal of clock $'a$, a to mean its value is true and $\text{not}\,a$ to mean it is false. Connectors and and or define conjunction and disjunction so that clocks expressions inherit natural Boolean relations such as $@x \Leftrightarrow x\,\text{or}\,\text{not}\,x$ or $0 \Leftrightarrow x\,\text{and}\,\text{not}\,x$.

$$
c,d ::= 0 \mid 1 \mid @a \mid a \mid \text{not}\,a \mid c\,\text{and}\,d \mid c\,\text{or}\,d \quad \text{(clock)}
$$

Assertions C are the conjunction or disjunction of implication relations between clock expressions. We write $C :: D$ for an assertion of pre-condition C and of post-condition D.

$$
C,D ::= \emptyset \mid c \Rightarrow d \mid C \wedge D \mid C \vee D \mid C :: D \quad \text{(constraint)}
$$

An example Let us put our type and clock system to work on a second example describing a delay function

pre. It takes a signal of type $'a$ and clock $'b$, a value of type $'a$ and returns a signal of type $'a$ and clock $'c$. If the first argument is absent, condition $\text{not}\,@b :: \emptyset$, then the result is absent, condition $\text{not}\,@c :: C$. The term C is a variable we use to denote the predicted assertion of function pre itself as it is recursively called with the signal s and the value w.

```
fun   pre []  v      = []
  | pre nil::s v   = nil::pre s v
  | pre one w::s v = one v::pre s w
```

If the first argument is present (condition $@b :: C$) then so is the result at c with the stored value v in place of which the current value w of the input signal is stored until the next recursive call of the function. From the deductions $\text{not}\,@b :: C \Rightarrow \text{not}\,@c :: C$ and $@b :: C \Rightarrow @c :: C$ follows a fixed-point resolution which yields the type and assertion C of the function pre. Namely, b is present iff c is present.

```
fun pre : 'a 'b signal * 'a -> 'a 'c signal
    / @b <=> @c
```

4. Clock inference

Clock synthesis consists of a logics that automates the informal deduction depicted in the previous example. It consists of building a relation of the form $E \vdash e : t/C$ for an expression e in a given environment E. The environment E associates defined identifiers x to type hypothesis t, written $x : t$ and the result of a deduction $e : t/C$ consists of the type t of the expression e and of an assertion C pertaining on its clock relations.

$$
E \vdash e : t/C \quad \text{(type inference relation)}
$$

Subtyping Types annotated with clock assertions naturally inherit a sub-typing relation from the logical structure of assertions they are built upon. We say that a type t is a subtype of u, written $t \leq u$, iff u makes stronger assertions than t. The relation $\leq$ is the largest equivalence relation that satisfies the following structural rules (as well as refexivity $t \leq t$, for all t).

$$
t_1 \leq t_2/C \Rightarrow \begin{array}{l} t_1\,'a\,\text{event} \leq t_2\,'b\,\text{event}/C \wedge (@a \Leftrightarrow @b) \\ t_1\,'a\,\text{signal} \leq t_2\,'b\,\text{signal}/C \wedge (@a \Leftrightarrow @b) \end{array}
$$

In the case of function types, in particular, we say that $t_1 \rightarrow u_1/C \leq t_2 \rightarrow u_2/D$ iff the input t_2 is a subtype of t_1, the output u_1 is a subtype of u_2 and the assertions D imply C to mean weaker hypothesis and stronger conclusion imply containment.

$$
\frac{t_1 \leq t_2/C_1 \quad u_1 \leq u_2/C_2}{t_1 * u_1 \leq t_2 * u_2/C_1 \wedge C_2}
$$

$$
\frac{t_1 \leq t_2/C_1 \quad u_1 \leq u_2/C_2 \quad D \Rightarrow C \wedge C_1 \wedge C_2}{t_1 \rightarrow u_1/C \leq t_2 \rightarrow u_2/D}
$$

Type assignment for atoms   We start the definition of the relation $E \vdash e : t/C$ by considering the axioms related to atomic expressions.

The type t of a bound identifier x should be known from the environment and be assumed left of the relation $\vdash$ in order to be guaranteed on its right.

$$(a) \; E, x : t \vdash x : t$$

The wirldcard $\_$ can be regarded as an identifier bound to any context. It has any type t and no assertion in any environment E, written $E \vdash \_ : t$.

$$(b) \; E \vdash \_ : t$$

The empty signal $[]$ can take any event type t and clock $'a$ so that $E \vdash [] : t'a$ signal.

$$(c) \; E \vdash [] : t'a \; \mathsf{signal}$$

Similarly, the event `nil` is any $t'a$ event yet with the assertion that a is absent, written $\mathsf{not} \, @a$.

$$(d) \; E \vdash \mathsf{nil} : t'a \; \mathsf{event}/\mathsf{not} \, @a$$

Type assignment for constructors   We build deduction rules using these axioms. Most of them are defined by induction on the structure of expressions e.

Rule (1) says that if the expression e satisfies the relation $E \vdash e : t/C$ then the constructor $\mathsf{one}\,e$ is a $t'a$ event with $'a$ present, written $@a$.

$$(1) \; \frac{E \vdash e : t/C}{E \vdash \mathsf{one}\,e : t'a \; \mathsf{event}/@a}$$

Similarly, rule (2) says that if the expressions e_1 and e_2 are respectively events and signals of type t and clock $'a$, then the constructor $e_1 :: e_2$ is a $t'a$ signal that inherits the sequence of assertions $C :: D$ from its sub-expressions.

$$(2) \; \frac{E \vdash e_1 : t'a \; \mathsf{event}/C \quad E \vdash e_2 : t'a \; \mathsf{signal}/D}{E \vdash e_1 :: e_2 : t'a \; \mathsf{signal}/C :: D}$$

Rule (3) performs an equally simple structural decomposition of deductions for a pairing expressions. Rule (4) defines the time of the rule $f \, p = e$ of a function f. In its hypothesis, it assumes an environment F to give type t_1/C_1 to its input pattern p.

$$(3) \; \frac{E \vdash e_1 : t_1/C_1 \quad E \vdash e_1 : t_2/C_2}{E \vdash (e_1, e_2) : t_1 * t_2/C_1 \wedge C_2}$$

Type assignment for expressions   The side condition (a) affects the choice of F in that the variables defined in the domain of E and the free variables of the pattern p should be disjoint sets. Rule (4) continues with the assignment of type and assertion t_2/C_2 to the expression e before concluding that $p = e$ has the function type $t_1 \rightarrow t_2/C_1 \Rightarrow C_2$.

$$(4) \; \frac{EF \vdash p : t_1/C_1 \quad EF \vdash e : t_2/C_2 \quad {}^{(a)}}{E \vdash p = e : (t_1 \rightarrow t_2/C_1 \Rightarrow C_2)/\emptyset}$$

Rule (5) defines the type t of a function definition $\mathsf{fun}\,x\,r$ by a fixed-point relation between the type of its rule $r : t$ and the hypothesis $x : t$ in its premiss.

$$(5) \; \frac{E, x : t \vdash r : t}{E \vdash \mathsf{fun}\,x\,r : t}$$

Rule (6) completes the processing of function definitions with the assignment of a type t to a pair of rules $r_1 | r_2$ by expecting both r_1 and r_2 to have the same type t.

$$(6) \; \frac{E \vdash r_1 : t \quad E \vdash r_2 : t}{E \vdash r_1 | r_2 : t}$$

This match requires the use of the sub-typing rule (7) in order to possibly lift the type t_1 of r_1 and t_2 of r_2 to the common upper-bound t.

$$(7) \; \frac{E \vdash e : t/C \quad t \leq u \quad D \Rightarrow C}{E \vdash e : u/D}$$

Rule (8) defines the type deduction procedure for the application of a function e_1 to an argument e_2. Its hypothesis is the deduction of a type $t_2 \rightarrow t/C$ and an assertion C_1 for e_1 and t_2/C_2 for e_2. The type of the application $e_1 e_2$ is t and its assertion the conjunction of C, C_1 and C_2.

$$(8) \; \frac{E \vdash e_1 : (t_2 \rightarrow t/C)/C_1 \quad E \vdash e_2 : t_2/C_2}{E \vdash e_1 e_2 : t/C \wedge C_1 \wedge C_2}$$

Similarly, rule (9) defines the type deduction for the definition of a rule $x = e_1$ in the lexical scope of an expression e_2. It says that, if e_1 admits at least one pair of (discarded) type and assertion t_1/C_1, then the expression has exactly the same type and assertion as the substitution $e_2[e_1/x]$ of x by e_1 in e_2.

$$(9) \; \frac{E \vdash e_1 : t_1/C_1 \quad E \vdash e_1[e_2/x] : t/C}{E \vdash \mathsf{let} \, x = e_1 \; \mathsf{in} \; e_2 : t/C}$$

A more algorithmic presentation of rule (9), presented next, uses a *polymorphic* type instead of e_1 in place of each occurrence of x in e_2. A polymorphic type consists of the universal quantification of all elements of the type and assertion (type, clock or assertion) of an expression that are

independent of the context of that assertion (the environment E). Rule (9) equally specifies polymorphism and effectively associates Standard ML expressions to assertions on clocks.

Example Let us depict the deduction performed with the relation $E \vdash e : t/C$ by considering a `merge` function. It takes two signals as arguments, if the first holds a value, it is copied to the output. Otherwise, the output signal is defined by the second input signal.

```
fun default [] [] = []
  | default nil::s nil::t = nil::default s t        (A)
  | default nil::s one v::t = one v::default s t    (B)
  | default one v::s _::t = one v::default s t      (C)
```

We shall first make an expectation D on the assertion of `default`. Assume:

$$\texttt{default} :' a'b\,\mathsf{signal} *'a'c\,\mathsf{signal} \rightarrow 'a'd\,\mathsf{signal}/D$$

Rule (A) initially assumes b and c absent, then nothing, and initially guarantees d absent and then D. Rule (B) assumes b absent and c present with v and guarantees d is present with v. Rule (C) assumes b present with v and discards c to guarantee d is present with v, therefore:

$$D = \begin{pmatrix} \mathsf{not}\,@b\,\mathsf{and}\,\mathsf{not}\,@c :: \emptyset \Rightarrow \mathsf{not}\,@d :: D \\ \wedge \quad \mathsf{not}\,@b\,\mathsf{and}\,@c :: \emptyset \Rightarrow @d :: D \\ \wedge \quad\quad\quad @b :: \emptyset \Rightarrow @d :: D \end{pmatrix}$$

The least fixed-point of the system of Boolean equations that defines D by the rules $(A - C)$ is $D =^{def} @d \Leftrightarrow @b\,\mathsf{or}\,@c$ and obtained after the following simplification steps.

$$\begin{aligned} D &= ((\neg(@b\,\mathsf{or}\,@c) \Rightarrow \neg@d) \wedge ((@b\,\mathsf{or}\,@c) \Rightarrow @d)) :: D \\ &= (@d \Leftrightarrow @b\,\mathsf{or}\,@c) :: D \\ &= (@d \Leftrightarrow @b\,\mathsf{or}\,@c) \end{aligned}$$

Correctness The correctness of our type inference system directly follows from known results on ML by considering the operational semantics of Standard ML [8] which structurally defines a relation $S \vdash e \Rightarrow v$ saying that expression e has value v in a context S that associates variables to values by $x : v$. In particular, and since we consider finite signals, we can actually consider Standard ML's big-step operational semantics.

Property 1 (type soundness [8]) *If $S : E$ and $S \vdash e \rightarrow v$ and $E \vdash e : t$ then $E \vdash v : t$.*

We call an environment E the type of a context S and write $S : E/C$ since it associates variables x to types with $x : t$. Stating the correctness of clock assertion can equally be done by considering assumptions C on input signals in S and guarantees D on output signals in v for an operation e over signals. Hence the result:

Property 2 (assertion soundness) *If $S : E/C$ and $S \vdash e \rightarrow v$ and $E \vdash e : t/D$ then $E \vdash v : t/D$.*

Extension with polymorphism Just as for the type inference of Standard ML programs, it is easy to extend our clock inference system with polymorphism and provide a deduction system whose deterministic processing closely mimics that of its algorithmic implementation.

Type polymorphism consists of the universal quantification of free variables (type $'a$, clock $'c$, behavior $'b$ variables) in the relation $E \vdash e : t/C$. A variable in t/C is said free if it does not occur in E. Since a free variable is not constrained by E in the relation $E \vdash v : t/D$, it can be substituted by any term of the same kind (type, clock or behavior). This substitution is called instantiation.

$$T ::= t/C \mid \forall'a.T \mid \forall'b.T \mid \forall'c.T$$

Generalization Type polymorphism avoids costly recalculation of a function's type when its name is referenced several times in a `let` expression. Generalization, function *gen*, consists of universally quantifying the typing judgment t/C over variables not occurring in the type environment E/D of the program. We write *fv* for the function that returns the set of variables free in its input term.

$$gen(E,t/C) = \forall V.(t/C), \ V = fv(t/C) \setminus fv(E)$$

As we wrote earlier, rule (9) can be formulated in a more algorithmic way by using generalization:

$$(9') \quad \frac{E \vdash e_1 : t_1/C_1 \quad E, x : gen(E, t_1/C_1) \vdash e_2 : t/C}{E \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : t/C}$$

Instantiation The converse operation of is called instantiation and consists of substituting quantified variables with fresh ones. We write $T[t/'a]$ for the substitution of the variable $'a$ by the term t in T.

$$\begin{aligned} \forall t,d,C, \ inst(E, \forall'a.T) &\supseteq inst(E, T[t/'a]) \\ inst(E, \forall'b.T) &\supseteq inst(E, T[C/'b]) \\ inst(E, \forall'c.T) &\supseteq inst(E, T['d/'c]) \\ inst(E, t/C) &\ni t/C \end{aligned}$$

The instantiation rule (a) is then defined as follows

$$(a') \quad \frac{t/C \in inst(E,T)}{E, x : T \vdash x : t/C}$$

Correctness The correctness of polymorphic clock inference can be established by stating the following property that is just an instance of the known results on polymorphic type inference.

Property 3 *If $E \vdash e_1 : t_1/C_1$ then $E, x : gen(E, t_1/C_1) \vdash e_2 : t/C$ iff $E \vdash e_2[e_1/x] : t/C$.*

5. A loosely time-triggered protocol

We exemplify the use of our multi-clocked model of computation by considering the case-study of a protocole for loosely time-triggered architecture that we proposed and proved in [12].

Specification The LTTA is composed of three devices, a *writer,* a *bus,* and a *reader*. Each device is activated by its own, approximately periodic, clock. At the nth clock tick (time $t^{\mathbf{w}}(n)$), the *writer* generates the value $x^{\mathbf{w}}(n)$ and an alternating flag $b^{\mathbf{w}}(n)$ s.t.:

$$b^{\mathbf{w}}(n) = \begin{cases} false & \text{if } n = 0 \\ not\ b^{\mathbf{w}}(n-1) & \text{otherwise} \end{cases}$$

Both values are stored in its output buffer, denoted by $y^{\mathbf{w}}$. At any time t, the writer's output buffer $y^{\mathbf{w}}$ contains the last value that was written into it:

$$y^{\mathbf{w}}(t) = (x^{\mathbf{w}}(n), b^{\mathbf{w}}(n)), \text{ where } n = \sup\{n' \mid t^{\mathbf{w}}(n') < t\} \quad (1)$$

At $t^{\mathbf{b}}(n)$, the *bus* bus fetches $y^{\mathbf{w}}$ to store in the input buffer of the reader, denoted by $y^{\mathbf{b}}$. Thus, at any time t, the reader input buffer is defined by:

$$y^{\mathbf{b}}(t) = y^{\mathbf{w}}(t^{\mathbf{b}}(n)), \text{ where } n = \sup\{n' \mid t^{\mathbf{b}}(n') < t\} \quad (2)$$

At $t^{\mathbf{r}}(n)$, the *reader* loads the input buffer $y^{\mathbf{b}}$ into the variables $x(n)$ and $b(n)$:

$$(x(n), b(n)) = y^{\mathbf{b}}(t^{\mathbf{r}}(n))$$

Then, in a similar manner as for an alternating bit protocol, the reader extracts $x(n)$ iff $b(n)$ has changed.
This is by the sequence m of ticks where b changes:

$$\begin{aligned} m(0) = 0, m(n) &= \inf\{k > m(n-1) \mid b(k) \neq b(k-1)\} \\ x^{\mathbf{r}}(k) &= x(m(k)) \end{aligned} \quad (3)$$
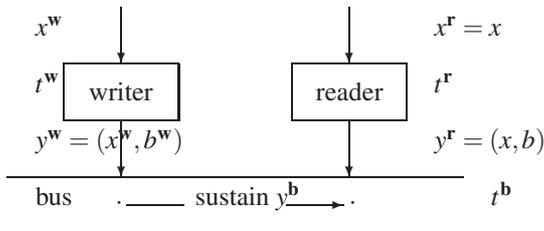
Example trace We illustrate the protocol by the following picture. Notice the role of the flag b: if the writer sends the same value along $x^{\mathbf{w}}$ twice, the boolean flag switch ensures that this value will be read twice on $x^{\mathbf{r}}$. On the opposite, if the value is sent once along $x^{\mathbf{w}}$ and read twice along $x^{\mathbf{r}}$, the boolean flag samples the excess of reading.
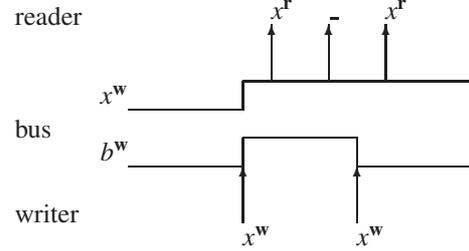
Flag switches are detected by the reader by a non predictable but bounded delay according to *physical* time: perfect physical synchrony is lost.

Correctness In any execution of the protocol, the sequences $x^{\mathbf{w}}$ and $x^{\mathbf{r}}$ must coincide, i.e.,

$$\forall n \cdot x^{\mathbf{r}}(n) = x^{\mathbf{w}}(n) \quad (4)$$

In order to prove the correctness of the protocol, we need to prove that, under some hypotheses on the clocks, the property (4) is true. In [12], we show that the LTTA protocol satisfies the property (4) iff the following conditions hold:

$$w \geq b \quad , \text{ and } \quad \left\lfloor \frac{w}{b} \right\rfloor \geq \frac{r}{b}, \quad (5)$$

where w, b and r are the respective periods of the clocks of the writer, the bus and the reader, and where, for $x \in \mathbb{R}$, $\lfloor x \rfloor$ denotes the largest integer less or equal to x.

Synchronous abstraction A clock is associated with any device d of a LTT architecture to provide the sampling frequency. It is possible to access a value at any time thanks to the function $l_{\mathbf{d}}$ associated to the device d which enables to access the value carried at the previous tick:
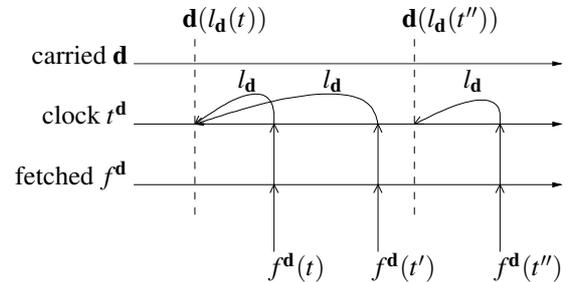
In a synchronous model of computation, the time continuum is abstracted. Only the notions of precedence and

simultaneity are relevant. It is therefore very simple to abstract the time domain using sampling events. In this approach, the clock $t^{\mathbf{d}}$ only defines the ordered set of sampling instants, and the carried values $\mathbf{d}$ are represented by a *signal* synchronized with $t^{\mathbf{d}}$. In order to make it possible to fetch the carried values *at any time*, we introduce a signal $f^{\mathbf{d}}$ whose clock[1] is completely free. The following picture illustrates this abstraction:
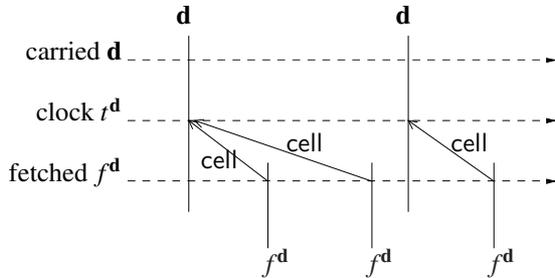
Programming methodology The methodology we use to implement the LTTA consists of its progressive and compositional refinement of the requirements specification: $x^r(n) = x^w(n), \forall n \geq 0$ that preserves the property of flow equivalence: xr and xw hold the same successive values. This yields the function ltta.

```
let fun ltta (xw cw, cb, cr) =
    let (xb, bb, sbw) =
            bus (xw, writer(xw, cw), cb)
    and (xr, br, sbb) = reader (xb, bb, cr)
    and _ = objective (sbw, sbb, cb, cr)
    in (xr, i, zi)
end;
```

The function ltta is decomposed into its three components reader, bus and writer connected by one-place buffers. The objective function describes the synchronization constraints under which the protocol correctness holds.

```
fun objective (sbw, sbb, cb, cr)
    let _ = sync cb (default sbw cb)
    and _ = sync cr (default sbb cr)
    in sync sbb (default sbb (switch sbb))
end;
```

The objective function uses the function sync to specify the synchronization relations that need to be maintained between the signals sbw, sbb, cb, cr for the requirements to hold.

```
fun sync [] [] = ()
  | sync nil::s nil::t = sync s t
  | sync one _::s one _::t = sync s t

fun sync: 'a signal * 'b signal -> void
    / @a <=> @b
```

[1]in a data-flow synchronous approach, by *clock* we mean the ordered set of instants where a signal is present

Notice that sync is best seen as the constraint `@a <=> @b` rather than as a function that operationally synchronizes two signals. Nonetheless, type inference allows to check the conformance of this objective with the type inferred for the rest of the LTTA specification (the writer, bus and reader) by simply showing that they are not contradictory.

```
fun objective: 'a signal * 'b signal
    * 'c signal * 'd signal -> void
    / @c <=> @a or @c
    / @d <=> @b or @d
```

The writer accepts an input $x^{\mathbf{w}}$ and defines the boolean flag $b^{\mathbf{w}}$ that will be carried along with it over the bus. The bus forward its inputs $x^{\mathbf{w}}$ and $b^{\mathbf{w}}$ to the reader as the result $x^{\mathbf{b}}$ and $b^{\mathbf{b}}$ of a one-place buffer.

```
fun writer (xw, cw) =
    let _ = sync bw xw and _ = sync xw cw
    in not (pre bw true)
end;
fun bus (xw, bw, cb) = buffer (xw, bw, cb)
```

The reader loads its inputs $x^{\mathbf{b}}$ and $b^{\mathbf{b}}$ from the bus and samples $x^{\mathbf{r}}$ from $x^{\mathbf{b}}$ upon a switch of $b^{\mathbf{b}}$. Each of the functions reader, bus and writer operate at independent (input) clocks $c^{\mathbf{w}}$, $c^{\mathbf{b}}$ and $c^{\mathbf{r}}$.

```
fun reader (xb, bb, cr) =
    let (yr, br, sbb) = buffer (xb, bb, cr)
    in (when yr (switch br), br, sbb)
end;
```

The switch function emits an output iff two successive occurrences *zb* and *b* of the boolean flag differ.

```
fun switch (b) =
    let zb = pre b true
    in default (when true (when b (not zb))
            (when true (when (not b) zb)
end;
```

We now detail the definition of the desynchronizing one-place buffer which simulates asynchrony. The function buffer alternates between the receipt of an input (x, b) and the emission of an output (bx, bb).

```
fun shift (x, b)
    let (sx, sb) = current (x, b, c)
    and _ = alternate (x, sx)
    in (sx, sb)
end;
fun buffer (x, b, c) =
    let (sx, sb) = shift (x, b, c)
    and (bx, bb) = current (sx, sb, c)
    in (bx, bb, sb)
end;
```

The alternate function makes these operations exclusive by using a boolean flip-flop signal b (notice, again, the importance of the initial condition: zb must be initialized to false for receive to precede send).

```
fun alternate (x, sx) =
    let b = not (pre b false)
    and _ = sync sx (when true (not b))
    in sync x (when true b)
end;
```

The function current sustains its input signals (wx, wb) and allows to retrieve them at a given clock c.

```
fun current (wx, wb, c) =
    let rx = when (cell wx c false) c
    and rb = when (cell wb c true)  c
    in (rx, rb)
end;
```

The function buffer introduces an unspecified delay (materialized by the input clock c), hence we can synchronize it with the output of the protocol xr without affecting the bus or the writer.

6. Related work

Functional programming languages, such as Milner's Standard ML [8], are high-level, declarative, strongly and statically-typed languages, which allow them to express computations concisely. Programs written in functional languages are also precise at a higher level of abstraction.

Research related to formal frameworks for heterogeneous modeling is a relatively new area of research. An example is the simulation framework for heterogeneous SystemC models proposed in [7], which aims at gaining fast simulation models and at allowing designers to compose heterogeneous components without worrying about the target simulation kernel. We use the approach of [5] to define a generic framework of models of computation for SoC design.

In this aim, a related work is ForSyDe [10], which consists of a standard library that provides a synchronous computational model implemented in Haskell. The objective of the ForSyDe methodology is to allow system design to be performed at higher level of abstraction and to bridge the abstraction gap by transformation design refinement. The most related work in the vast domain of type-based program analysis is the work of Pouzet et al. on Lucid Synchrone [2]. The dependent type system of Lucid Synchrone allows to check that a functional program is well-synchronized: that all streams in this program have totally related clocks.

7. Conclusion

The approach introduced in this paper differs by considering the global design of an architecture from elementary functions with partially related clocks. Design proceeds in a compositional and refinement-based manner using expressive temporal propositions on Boolean expressions to gradually check correctness of the assembled components using an assumption/guarantee reasoning.

In this aim, the clock inference system we presented is the first to relate implicitly typed stream functions with an expressive propositional logics that abstract the behavior of stream functions by Boolean expressions on clocks and allow for an expressive reasoning on properties pertaining to static verification and optimization. We demonstrated that this verification and optimization framework fully integrates to a model of computation described with %100 standard ML programs and no ad-hoc hypothesis.

References

[1] J.T. BUCK, S. HA, E.A. LEE AND D.G. MESSERSCHMITT. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems. In *International Journal of Computer Simulation, special issue on Simulation Software Development*. v. 4, pp. 155-182. Ablex, 1994.

[2] COLACO, J. L., GIRAULT, A., HAMON, G., POUZET, M. Towards a higher-order synchronous dataflow language. In *Embedded Software Conference*, Springer Verlag lectures notes in computer science, 2004.

[3] GAJSKI, D., ZHU, J., DÖMER, R., GERSTLAUER, A., ZHAO, S. SpecC: Specification Language and Methodology. Kluwer Academic Publishers, 2000.

[4] GROETKER, T., LIAO, S., MARTIN, G., SWAN, S. System Design with SystemC. Kluwer Academic Publishers, 2002.

[5] JANTSCH, A. *Modeling embedded systems and SOCs, concurrency and time in models of computation*. Morgan Kuafmann Publishers, 2003.

[6] LE GUERNIC, P., TALPIN, J.-P., LE LANN, J.-L. "Polychrony for system design". In *Journal of Circuits, Systems and Computers*. World Scientific, 2002.

[7] MATHAIKUTTY, D. A., PATEL, H. D., SHUKLA, S. K. "A functional programming framework of heterogeneous models of computation for system design". In *Forum on Specification and Design Languages*, 2004.

[8] MILNER, R., TOFTE, M., HARPER, R. The definition of Standard ML. *The MIT Press*, Cambridge, 1990.

[9] POTOP, D., CAILLAUD, B. "Correct-by-construction asynchronous implementation of modular synchronous specifications". In *Applications of Concurrency to System Design*. IEEE Press, 2005.

[10] SANDER, I. *System modeling and design refinement in ForSyde, PhD Thesis*. Royal Institute of Technology, 2003.

[11] TALPIN, J. P., JOUVELOT P. The type and effect discipline. *Information and Computation*, v. 115. Academic Press, 1994.

[12] BENVENISTE, A., CASPI, P., LE GUERNIC, P., MARCHAND, H., TALPIN, J.-P., TRIPAKIS, S. "A protocol for loosely time-triggered architectures". *Embedded Software Conference*. Springer Verlag, October 2002.

[13] TOFTE, M., TALPIN, J. P. Region-based memory management. *Information and Computation*. Academic Press, 1997.