Clock type soundness in synchronous languages

Martin Strecker

Université Paul Sabatier / IRIT, Toulouse
http://www.irit.fr/~Martin.Strecker

Abstract. Synchronous languages are based on the hypothesis that computations of a program are carried out at specific instants, as determined by the clocks of the signals processed by the program. A clock type system ensures that the program does not manipulate invalid data. In this paper, we establish an abstract type soundness result for synchronous languages: given a program, we can derive a system of set equations whose solution guarantees the absence of invalid data during execution. We then instantiate this result for synchronous languages with periodic clocks and show how to effectively solve the resulting set constraints. The development has been formalized in the Isabelle proof assistant.

1 Introduction

Synchronous languages [Hal98] are an interesting paradigm for the construction of reactive systems. Such a system is seen a network of communicating nodes. At a node, all "required" data are assumed to be available at specific instants, are processed and then forwarded to consumers nodes which in turn treat the data, eventually at a different pace. *Clocks* are a central notion for modeling the fact that nodes and the attached data channels operate at a different pace: A clock determines when data offered by a channel can be considered as "valid". In synchronous languages, a *clock type system*, orthogonal to a traditional value type system, is meant to ensure the "coherence" of data processed by the program.

Synchronous programming languages have been in use for quite a while, and there is a large number of formalisms, ranging from procedural [Ber00,Sch09] over functional [Hal05,Pou06,Bro93] to declarative [GTL03]. These languages are equipped with increasingly complex clock type systems [CDE+06,MP09] that aim at relaxing the notion of synchrony. However, one question remains unanswered: what is the precise relation between clock types and the semantics of the programming language? Which properties does a well-typed program ensure?

This paper tries to give one possible answer, by defining a concept of "type soundness" in synchronous languages. We first present a simplified synchronous language (Section 2) in the style of Signal / Polychrony [GTL03], and its semantics (Section 4) based on a notion of streams (Section 3). To be more precise, we define two kinds of semantics: one that ignores synchronization errors, and one

that flags the occurrence of these errors. We then define clocks and their semantics as sets of time instants (Section 5). Clock type checking of a program consists in verifying the validity of clock set constraints derived from that program, and type soundness expresses that the two above-mentioned semantics coincide. This result can be formulated abstractly, with an a priori non-constructive notion of validity of constraints. We then instantiate the framework to periodic clocks and give an effective procedure for solving the resulting constraints (Section 6).

The development presented here has been carried out in the Isabelle proof assistant [NPW02]. There have been previous codings of synchronous languages in proof assistants: [Now99] directly codes the semantics of the Signal language as a set of propositions, without defining a syntax, so that it is not possible to reason about the effects of syntax transformations. Some approaches [CDC00,PM08] formalize streams by co-induction, which does not seem to have decisive advantages, but imposes a discrete time model that we can avoid in our abstract setting. [BH01] use a tricky encoding of Lucid Synchrone's clock types within the type system of the proof assistant Coq. It is not clear whether this technique scales up to more expressive clock type systems, such as periodic clocks. [Spi08] presents a specification of streams and stream refinement in the context of the Focus formalism.

Some clock type systems with periodic clocks have been suggested. In a more general setting, [CDE+06,MP09] code the periodicity by infinite binary strings and develop specific unification algorithms for these strings. [FBLP08] presents a clock type system similar to ours, with the difference that union of clock constraints are not allowed. The type checking algorithm intertwines generation and solution of clock constraints (via unification), whereas our algorithm proceeds in two distinct phases.

2 A synchronous language

Our aim is to model synchronous dataflow languages in the style of Lustre [HCRP91] and Signal [GTL03]. As far as the basic constructs are concerned, these languages are similar, in that they allow to represent a parallel composition of assignments of values to variables (and that hierarchically).

Semantically, we are interested in *streams*, i.e. sequence of values computed in the course of time. The major difficulty resides in the fact that, as time passes, values can be present or absent – and that is where the aforementioned languages differ. In our exposition, we follow the semantic style of Signal.

Values can be computed by *combinatorial* operators that assume that their operands are synchronous, i.e. are either all present or all absent at the same time. Typically, these are the traditional arithmetic and boolean operators. The `pre` operator retrieves the previous value in the execution sequence, initializing the first value to a constant. There are two operators for combining sequences of values: `when` selects the current value from its first argument, provided the current value of its second argument is present and true. `default` merges two

streams by selecting the current value from its first argument if it is present; otherwise, it takes the current value from its second argument.

In analogy with traditional programming languages, we distinguish between expressions `expr` and statements `stmt` (in the following, `'n` is for variable names and `'v` for values):

datatype $('n,'v)$ *expr*
  = *Var* $'n$ — variables: $x, y, ...$
  | *PairE* $(('n,'v)$ *expr*$)$ $(('n,'v)$ *expr*$)$ — pairs: (e_0, e_1)
  | *Fun funid* $(('n,'v)$ *expr*$)$ — combinatorial function application: $f(e)$
  | *Pre* $'v$ $(('n,'v)$ *expr*$)$ — pre v e with initial value v
  | *When* $(('n,'v)$ *expr*$)$ $(('n,'v)$ *expr*$)$ — e_0 when e_1
  | *Default* $(('n,'v)$ *expr*$)$ $(('n,'v)$ *expr*$)$ — e_0 default e_1

datatype $('n,'v,'tp)$ *stmt*
  = *EmptyStmt* — neutral element of parallel compos.
  | *Eq* $'n$ $(('n,'v)$ *expr*$)$ — equality: $x := e$
  | *Par* $(('n,'v,'tp)$ *stmt*$)$ $(('n,'v,'tp)$ *stmt*$)$ — parallel compos.: $c_0||c_1$
  | *Letv* $'n$ $'tp$ $(('n,'v,'tp)$ *stmt*$)$ — local variable: let $x : T$ in c
  | *PCall* $('n$ *cname*$)$ $('n$ *list*$)$ — procedure call: $P(a_1...a_n)$

Local variables have to be typed; we abstract over a type `'tp` that will later be instantiated with the clock type of Section 5. Contrary to the expression constructor `Fun`, the statement constructor `PCall`, similar to a procedure call in traditional programming languages, does not assume that its arguments are synchronous. Quite conventionally, we can now define a procedure declaration (consisting of a header and a procedure body) and a procedure environment mapping a header to a procedure.

datatype $('n,'v,'tp)$ *proc*
  = *Proc* $(('n,$ *proc-header*$)$ *proc-interf*$)$ $(('n,'v,'tp)$ *stmt*$)$

types $('n,'v,'tp)$ *proc-env* = *proc-header* $\Rightarrow$ $('n,'v,'tp)$ *proc option*

On this basis, we can define a predicate `wtpd_stmt` that checks well-typing of a statement in an environment, according to a traditional *value type* system. Since the main concern of our investigations is a *clock type* system, the definition of this predicate just includes the strict minimum (variables and procedure names have to be declared in the context in which they are used), and we do not spell out the definition in more detail here.

3 Streams

Our semantics is based on streams, which are essentially maps from a temporal domain to values. An `event` is a mapping from variable names `"'n"` to "tagged" values `"'v option"` where `None` indicates the absence of a value, `"Some v"` the presence of a value `"v"`.

3.1 Stream functions

A `trace` describes the occurrence of events over time, thus the mapping from a temporal domain `'t` to events. For the time being, we make no specific assumptions about `'t` (discrete, continuous ...), but introduce them progressively. In a similar vein, assumptions about the value domain will be introduced as needed.

A `signal` is the projection of a trace to one specific variable.

types $('n,'v)$ *event* $= 'n \Rightarrow 'v$ *option*
types $('t,'n,'v)$ *trace* $= 't \Rightarrow ('n,'v)$ *event*
types $('t,'v)$ *signal* $= 't \Rightarrow 'v$ *option*

To prepare the definition of the semantics of expressions and statements, we first define functions manipulating signals that correspond to the intended semantics of our operators. The case of `Default` is easiest, it corresponds to the composition of two maps:

constdefs
default :: $[('t,'v)$ *signal*, $('t,'v)$ *signal*$] \Rightarrow ('t,'v)$ *signal*
default $s\ s' == s' ++ s$

For implementing the operator `When`, we have to assume that the value domain provides a function `true_val` for testing whether a value is true. This can be realized using Isabelle's type class mechanism: we introduce the type class `vals_cl` which provides the function `true_val` and also pairing and projection functions with the obvious properties.

constdefs
when :: $[('t,'v::$*vals-cl*$)$ *signal*, $('t,'v)$ *signal*$] \Rightarrow ('t,'v)$ *signal*
when $s\ s' == (\lambda\ t.$
case $(s'\ t)$ *of*
None $\Rightarrow$ *None*
$|$ *Some* $v \Rightarrow$ *if* (*true-val* v) *then* $(s\ t)$ *else None*)

We implement the operator `Pre` with function `delay`, which produces a signal which is synchronous with the original signal `s`, and such that at a given instant `t`, the delayed signal has the previous defined value of `s` (or the initial value if there is no previous defined one). We assume that the temporal domain is at least linearly ordered (type class `linorder`). If the set `A` of defined predecessor instants of `t` is empty, we return the initial value `v`, otherwise the value associated to the maximal predecessor of `t`.

constdefs
previous-defined :: $['v, ('t::$*linorder*$, 'v)$ *signal*, $'t] \Rightarrow 'v$
previous-defined $v\ s\ t ==$
$($*let* $A = \{t' \in$ *dom* $s.\ t' < t\}$ *in*
if $A = \{\}$ *then* v *else the* $(s\ ($*Max* $A)))$

delay :: $['v, ('t::$*linorder*$, 'v)$ *signal*$] \Rightarrow ('t,'v)$ *signal*
delay $v\ s ==\ \lambda\ t.$ *case* $(s\ t)$ *of*
None $\Rightarrow$ *None*
$|$ *Some* $v' \Rightarrow$ *Some* (*previous-defined* $v\ s\ t$)

In the definition of `previous_defined`, the function `the` is the inverse of `Some`. The definition is questionable if the maximum of `A` is not in the domain of `s`. Such a situation can be excluded for discrete event streams, having a discrete number of defined events (which still allows for a continuous temporal domain):

constdefs
discrete-event-signal :: $('t::order,'v)$ *signal* $\Rightarrow$ *bool*
discrete-event-signal $s == \forall\ t.$ *finite* $(\{t'.\ t' \leq t\} \cap$ *dom* $s)$

3.2 Errors

There remains the case of the *PairE* constructor and combinatorial operators, which assume that their arguments are synchronized. What happens if they are not? There are two possible solutions, which give rise to two different semantics: ignore the problem, or tag values resulting from badly synchronized computations as erroneous. In Section 5.3, we will show that these two semantics coincide, provided a program is well-typed according to the clock type system defined in Section 5.

In the first solution (ignoring synchronization errors), a pair of streams is formed by pointwise application of a function *lift-pair* that handles bad synchronization as absence of a value:

constdefs
lift-pair :: $[('v::$*vals-cl*$)$ *option*, $'v$ *option*$] \Rightarrow 'v$ *option*
lift-pair vo vo' ==
(*case vo of*
None $\Rightarrow$ *None*
| *Some v* $\Rightarrow$
(*case vo'* *of*
None $\Rightarrow$ *None*
| *Some v'* $\Rightarrow$ *Some* (*pair-val* (v, v'))))
signalpair :: $[('t,'v::$*vals-cl*$)$ *signal*, $('t,'v)$ *signal*$] \Rightarrow ('t,'v)$ *signal*
signalpair s s' == $(\lambda\ t.$ (*lift-pair* $(s\ t)$ $(s'\ t)$))

In the second solution, values are lifted to an error domain

datatype $'v$ *error* = *Err* | *OK* $'v$

In analogy to *lift-pair* and *signalpair*, we can now define *lift-pair-error* and

constdefs
signalpair-error ::
$[('t,$ $('v::$*vals-cl*$)$ *error*$)$ *signal*, $('t,'v$ *error*$)$ *signal*$] \Rightarrow ('t,'v$ *error*$)$ *signal*
signalpair-error s s' == $(\lambda\ t.$ (*lift-pair-err* $(s\ t)$ $(s'\ t)$))

where *lift-pair-error None* (*Some* v) = *Some Err* (similarly for arguments inverted) and we get a result of the form *Some* (*OK* v) only for correctly synchronized arguments.

Just as for the *PairE* constructor, there are two versions for *Fun*, applying a function to a $('t,'v)$ *signal* respectively a $('t,'v$ *error*$)$ *signal*. For easier handling

in the semantics, we bundle these functions in two records, *error-fns-flat* and *error-fns-lifted*.

4 Semantics

4.1 Language in canonical form

For subsequent treatment, especially derivation of clock constraints (see Section 5.1), it is inconvenient to have to deal with nested expressions, as long as these are not purely combinatorial, such as

```
y := when (f(x + 2)) (pre true b)
```

By introducing new local variables, we will rewrite these expressions in such a way that the operators *pre*, *when* and *default* are only applied to variables.

```
let x' : Tx', b' : Tb' in
 || y := when x' b'
 || x' := (f(x + 2))
 || b' := (pre true b)
```

Technically , this amounts to introducing new datatypes *canon-expr* (like *expr*, without the constructors *Pre*, *When*, *Default*) and *canon-stmt* (like *stmt*, with new constructors *CanPre*, *CanWhen*, *CanDefault*). The last line in the example is now represented as: *CanPre b′ True b*. We do not explicitly spell out the definition of these types, but will henceforth use them for the definition of the semantics.

4.2 Semantic rules

Since we now only have combinatorial expressions, their semantics can be computed pointwise, for a given trace, to yield a signal. The only difficulty can arise for pairing, as outlined in Section 3.2. We therefore parameterize the semantics with the error function to be applied:

consts
  interp-canon-expr ::
  [('*t*::*linorder*, '*v*::*vals-cl*, '*ve*) *error-fns*, ('*t*,'*n*, '*v*) *trace*, ('*n*,'*v*) *canon-expr*]
  ⇒ ('*t*, '*ve*) *signal*
primrec
  interp-canon-expr efs tr (*CanVar x*) =
      apply-fun-pointwise (*error-elem-lift efs*) (*trace-proj tr x*)
  interp-canon-expr efs tr (*CanPairE e e′*) =
      (*error-pair efs*
        (*interp-canon-expr efs tr e*)
        (*interp-canon-expr efs tr e′*))
  interp-canon-expr efs tr (*CanFun f e*) =

apply-fun-pointwise
(*error-fun-lift efs* (*interp-funid f*))
(*interp-canon-expr efs tr e*)

Remember that function application is always uncurried, so we have to use pairing for binary functions. Consequently, the semantics of the expression (*CanFun PlusF* (*CanPairE* (*CanVar* ″*x*″) (*CanVar* ″*y*″))) is

- (λ *t*. *if t mod 2 = 0 then Some* (*NatV 3*) *else None*) for the "flat" error function, if x is constantly 1, and y is 2 on even instants and otherwise undefined.
- (λ *t*. *if t mod 2 = 0 then Some* (*OK* (*NatV 3*)) *else Some Err*) for the "lifted" error function, for x and y as above.

The interpretation of statements is "relational". Thus, the interpretation of a statement does not produce a single stream (as is common in the semantics of languages like Lustre [HCRP91] or Lucid Synchrone [Pou06]). Rather, the semantics is in the spirit of Signal [GTL03]: We are interested in knowing whether a trace *tr* is a model of a statement. The semantics is parameterized by an error function selector, which can be either *Blocking* or *Permissive*. The distinction plays a role in the semantics of equalities $y = e$, where e is a combinatorial expression. In the permissive interpretation, we only verify that the interpretations of y and e coincide for the given trace *tr*. In addition, in the blocking case, the interpretation of e under *tr* has to be error-free.

The constructors *CanPre*, *CanWhen* and *CanDefault* are straightforward and make appeal to the functions defined in Section 3.1. Parallel composition is interpreted as conjunction.

For interpreting a procedure call, we look up the procedure definition in the procedure environment *penv*, given the procedure name *pn*. The procedure body *bd* is interpreted relative to a trace in which the arguments of the caller have been remapped to the formal parameters of the procedure.

Remember that we have local variable *declarations let v* : *clk in c* contrary to definitions of the style *let v* = *e in e*′ known from functional languages or their synchronous derivatives. The difference is essential: the value to be substituted for variable v is not tightly constrained by the value of e, but only by the clock *clk*. Adding the clock declaration v : *clk* is also a notable departure from the Signal language, indispensable for our type soundness result (see Section 5.3 for a discussion). The interpretation is as follows: To see whether a trace *tr* is a model of a *let*, we interpret the body c in a trace where v has been updated by an arbitrary stream s, provided s is compatible with the clock of the *let*. The parameter *tpi* is a type interpretation of the clock *clk* that will be discussed in the context of clock interpretations.

inductive
interp-canon-stmt ::
[*sync-error-handling*, (′*t*::*linorder*,′*n*,′*v*::*vals-cl*) *trace*, (′*n*,′*v*,′*tp*) *canon-proc-env*,
(′*t*,′*n*,′*v*) *trace* ⇒ ′*tp* ⇒ ′*t set*, (′*n*,′*v*,′*tp*) *canon-stmt*]
⇒ *bool*

where
interp-canonCanEmptyStmt [*simp*]:
interp-canon-stmt efs tr penv tpi CanEmptyStmt
| *interp-canonCanCombin*:
⟦ *case efs of*
Blocking ⇒ *error-free-signal*
(*interp-canon-expr error-fns-lifted tr e*)
| *Permissive* ⇒ *True*;
(*trace-proj tr y* =
interp-canon-expr error-fns-flat tr e) ⟧
⟹ *interp-canon-stmt efs tr penv tpi* (*CanCombin y e*)
| *interp-canonCanPre*:
trace-proj tr y = *delay v* (*trace-proj tr x*)
⟹ *interp-canon-stmt efs tr penv tpi* (*CanPre y v x*)
| *interp-canonCanWhen*:
trace-proj tr y = *when* (*trace-proj tr x*) (*trace-proj tr b*)
⟹ *interp-canon-stmt efs tr penv tpi* (*CanWhen y x b*)
| *interp-canonCanDefault*:
trace-proj tr y =*default* (*trace-proj tr x*) (*trace-proj tr z*)
⟹ *interp-canon-stmt efs tr penv tpi* (*CanDefault y x z*)
| *interp-canonCanPar*:
⟦*interp-canon-stmt efs tr penv tpi c*;
interp-canon-stmt efs tr penv tpi c′⟧
⟹ *interp-canon-stmt efs tr penv tpi* (*CanPar c c′*)
| *interp-canonCanPCall*:
⟦ *penv pn* = *Some* (*CanProc* (*ProcInterf pn′ pars*) *bd*);
interp-canon-stmt efs
(λ*t*. *remap-fun* (*tr t*) *pars args*)
penv tpi bd ⟧
⟹ *interp-canon-stmt efs tr penv tpi*
(*CanPCall* (*Cname pn cnum*) *args*)
| *interp-canonCanLetv*:
⟦ *interp-canon-stmt efs* (*trace-upd tr v s*) *penv tpi c*;
dom s = *tpi tr clk* ⟧
⟹ *interp-canon-stmt efs tr penv tpi* (*CanLetv v clk c*)

5 Clocks

Synchronous languages come equipped with two type systems: a traditional value type system, and a clock type system. We do not spell out the first in detail, and indeed make only minimal assumptions in our formalization (essentially: variables are declared before use).

Clock types are assimilated to sets. In Section 5.1, we present their syntax and semantics, and in Section 5.2, we show how to derive a system of clock constraints, given a synchronous program. The type soundness result of Section 5.3 will show that type correct programs are synchronization-error free.

5.1 Clock constraints

We lift the distinction between expressions and statements to clocks: *clexpr* correspond to sets of time instants, and *clstmt* to systems of set constraints. *clexpr* has the "universe" set *ClTop* and the usual boolean operators. There are two kinds of atoms: *ClOf* is the set of time instants when a variable is active. In *ClWhen*, we can directly give a set of time instants, more precisely: the representation of such a set in an abstraction domain $'a$. In Section 6, we will consider an instantiation of $'a$ to affine sets that arise in the analysis of periodic clocks.

datatype $('n,'a)$ *clexpr*
  = *ClOf* $'n$
  | *ClWhen* $'a$
  | *ClTop*
  | *ClInter* $(('n,'a)$ *clexpr*$)$ $(('n,'a)$ *clexpr*$)$
  | *ClUnion* $(('n,'a)$ *clexpr*$)$ $(('n,'a)$ *clexpr*$)$
  | *ClCompl* $(('n,'a)$ *clexpr*$)$

A clock constraint (*clstmt*) is essentially a (possibly empty) conjunction of set equations $ce_1 = ce_2$, and in addition *ClLet* for hiding local variables.

datatype $('n,'a)$ *clstmt*
  = *ClTrue*
  | *ClEq* $(('n,'a)$ *clexpr*$)$ $(('n,'a)$ *clexpr*$)$
  | *ClConj* $(('n,'a)$ *clstmt*$)$ $(('n,'a)$ *clstmt*$)$
  | *ClLet* $'n$ $(('n,'a)$ *clexpr*$)$ $(('n,'a)$ *clstmt*$)$

This informal description of the semantics is made precise by the functions *interp-clexpr* and *interp-clstmt*, which are both parameterized by a trace and by a "concretization" function *concr* converting the abstraction domain of *ClWhen* to a set of time instants.

consts
  interp-clexpr ::
  $[('t,'n,'v)$ *trace*, $('t,'n,'v)$ *trace* $\Rightarrow$ $'a \Rightarrow$ $'t$ *set*, $('n,'a)$ *clexpr*$]$
    $\Rightarrow$ $'t$ *set*
primrec
  interp-clexpr tr concr (*ClOf x*) = *dom* (*trace-proj tr x*)
  interp-clexpr tr concr (*ClWhen a*) = (*concr tr a*)
  interp-clexpr tr concr ClTop = *UNIV*
  interp-clexpr tr concr (*ClInter ce ce'*) =
    (*interp-clexpr tr concr ce*) $\cap$ (*interp-clexpr tr concr ce'*)
  interp-clexpr tr concr (*ClUnion ce ce'*) =
    (*interp-clexpr tr concr ce*) $\cup$ (*interp-clexpr tr concr ce'*)
  interp-clexpr tr concr (*ClCompl ce*) = $-$ (*interp-clexpr tr concr ce*)

consts
  interp-clstmt ::
  $[('t::order, 'n, 'v)$ *trace*, $('t,'n, 'v)$ *trace* $\Rightarrow$ $'a \Rightarrow$ $'t$ *set*,
    $('n,'a)$ *clstmt*$]$ $\Rightarrow$ *bool*
primrec

```
interp-clstmt tr concr ClTrue = True
interp-clstmt tr concr (ClEq e e') =
  ((interp-clexpr tr concr e) = (interp-clexpr tr concr e'))
interp-clstmt tr concr (ClConj c c') =
  (interp-clstmt tr concr c ∧ interp-clstmt tr concr c')
interp-clstmt tr concr (ClLet v e c) =
(∀ s. dom s = interp-clexpr tr concr e ⟶
      interp-clstmt (trace-upd tr v s) concr c)
```

5.2 Extracting clock constraints

The procedure we envisage for checking clock types is that of taking a program, extracting a system of set constraints (*clstmt*) and then passing these constraints on to a solver. Constraint generation and constraint solving are not intertwined as in traditional programming and as in the approach described in [FBLP08]. Decoupling generation and solving offers several advantages, among others that one can show a "generic" adequacy result (as in Section 5.3) without referring to specific instances, and that one might use off-the-shelf solvers for some classes of problems. We will look at a solver (however hand-made) for one instance of these constraints in Section 6.

Extraction of constraints is essentially performed by a recursive traversal of *canon-stmt*, which is favoured by their simple structure with almost no nested expressions. These can only occur for *CanCombin*. In a combinatorial statement, like for example $y = x_1 + x_2$, we equate the clock of y with the clocks of x_1 and x_2, thus obtaining the constraint $\texttt{ClOf}(y) = \texttt{ClOf}(x_1) \wedge \texttt{ClOf}(y) = \texttt{ClOf}(x_2)$. More in general, we use the function

```
constdefs
  clock-combin :: ['n, ('n,'v) canon-expr] ⇒ ('n,'a) clstmt
  clock-combin y e ==
  clConjs (map (λ x. (ClEq (ClOf y) (ClOf x))) (fv-canon-expr e))
```

where *clConjs* takes the conjunction of a list and *fv-canon-expr* calculates the free variables of an expression. With this, we can now define:

```
consts
  clock-canon-stmt ::
  ['n ⇒ 'a, ('n,'a) clock-env,
       ('n,'v, (('n,'a) clexpr)) canon-stmt] ⇒ ('n,'a) clstmt
primrec
  clock-canon-stmt abstr clenv CanEmptyStmt = ClTrue
  clock-canon-stmt abstr clenv (CanCombin y e) = clock-combin y e
  clock-canon-stmt abstr clenv (CanPre y v x) = (ClEq (ClOf y) (ClOf x))
  clock-canon-stmt abstr clenv (CanWhen y x b) =
       ClEq (ClOf y) (ClInter (ClOf x) (ClWhen (abstr b)))
  clock-canon-stmt abstr clenv (CanDefault y x z) =
       ClEq (ClOf y) (ClUnion (ClOf x) (ClOf z))
  clock-canon-stmt abstr clenv (CanPar c c') =
       ClConj (clock-canon-stmt abstr clenv c) (clock-canon-stmt abstr clenv c')
```

```
clock-canon-stmt abstr clenv (CanPCall cn args) =
  clock-param-form (the (clenv (cname-header cn))) args
clock-canon-stmt abstr clenv (CanLetv v clk c) =
  ClLet v clk (clock-canon-stmt abstr clenv c)
```

The definition is rather straightforward, so let us just comment two cases. For $y = \texttt{when}\ x\ b$, y has to be synchronous with the instants when x is active and b is active and true. The latter is approximated by an abstraction function *abstr* that maps variable b to the abstract domain. In the case of a procedure call (case *CanPCall*) of the form $cn(a_1 \ldots a_n)$, we look up the clock specification corresponding to cn in the clock environment *clenv*. This clock specification is composed of a parameter list $p_1 \ldots p_n$ and a body c (a *clstmt*), with $p_1 \ldots p_n$ free in c. We now form the constraint $\texttt{ClLet}\ p_1 = a_1, \ldots p_1 = a_1\ \texttt{in}\ c$.

5.3 Type soundness

Type soundness expresses that no "erroneous" results can be computed, in the sense of Section 4.2: The blocking and the permissive semantics coincide. More in detail: Given a clock statement c, we can compute a system of set constraints. If this system is satisfiable for a trace *tr*, then both interpretations are the same for *tr*. We need some well-formedness preconditions: the procedure environment *cpenv* has to be well-formed (i.e., all the procedures are well-typed according to the traditional value type system), the clock environment *clenv* has to be well-formed with respect to *cpenv* (i.e., procedure signatures have to correspond, and the body of each procedure has to be well-typed, according to the clock type system) and c itself has to be well-typed according to the value type system. The concretization function *concr* must be domain-indifferent, i.e. make no distinction between equally-clocked traces with possibly different values.

constdefs

```
dom-indifferent :: (('t,'n, 'v) trace ⇒ 'a ⇒ 't set) ⇒ bool
dom-indifferent concr == ∀ tr tr'.
(∀ x. dom (trace-proj tr x) = dom (trace-proj tr' x)) ⟶
      concr tr = concr tr'
```

theorem *clock-type-soundness*:

```
⟦wf-canon-proc-env cpenv; wf-clock-env concr cpenv clenv;
wtpd-canon-stmt cpenv tenv c;
dom-indifferent concr;
interp-clstmt tr concr (clock-canon-stmt abstr clenv c);
tpi =  (λ tr'. interp-clexpr tr' concr)
⟧
⟹ interp-canon-stmt Blocking tr cpenv tpi c
  = interp-canon-stmt Permissive tr cpenv tpi c
```

The proof is by induction on the relation defining *interp-canon-stmt* (for the direction "*Permissive* implies *Blocking*", the other direction being trivial). We will only discuss the case of *let*-expressions by means of an example, because the

proof has highlighted a problem in previous presentations of clock calculi, such as [OTBG08].

```
let x, y in z = x + y
```

it is not sufficient to generate a constraint where the "hidden" variables are just existentially quantified: $\exists XY.Z = X \wedge Z = Y$, if X, Y, Z are the "clock variables" corresponding to x, y, z. Obviously, the constraint is valid, and indeed, the variables x and y are *synchronizable*. The problem is that they are not *necessarily* synchronized, whence our suggestion to annotate *let*-bound variables with clock expressions:

```
let x: cl(z), y: cl(z) in z = x + y
```

From this, we derive a clock constraint which is roughly equivalent to

$$\forall x.\texttt{ClOf}(x) = \texttt{ClOf}(z) \rightarrow \forall y.\texttt{ClOf}(y) = \texttt{ClOf}(z) \rightarrow$$

$$\texttt{ClOf}(z) = \texttt{ClOf}(x) \wedge \texttt{ClOf}(z) = \texttt{ClOf}(y)$$

which is also valid. The difference: the annotations allow to alter the semantics of expressions, forcing the signals realizing x and y to be synchronous with z.

Some remarks are in order: Firstly, it is of course unsatisfactory to have the semantic notion of "interpretation of a clock statement for any trace *tr*" as precondition of type soundness, and the dependence of the postcondition on the individual trace *tr*. In Section 6, we will consider solvers deciding $\forall$ *tr*. *interp-clstmt tr concr c*. Using this fact and clock type soundness, we can show by elementary reasoning that

$\forall$ *tr*. *interp-canon-stmt Blocking tr cpenv tpi c* =
interp-canon-stmt Permissive tr cpenv tpi c

Differently said, the equivalence of blocking and permissive semantics for arbitrary traces can be reduced to a property that can be effectively decided by a constraint solver.

Another remark concerns the concept of "type soundness" itself: It is, in a sense, a weak notion which only makes a statement about the non-occurrence of synchronization errors, but says nothing about the existence of a trace *tr* which is model for a clock statement *c*. In the case of non-existence of a model for a specific *c*, the conclusion of the type soundness theorem is trivially satisfied.

6 Solving constraints

We will now instantiate the general framework, showing how to solve constraints arising when checking programs involving periodic clocks. For example, we would like to write

```
let cx : {|10t|}, cy : {|10t + 5|} in
   z = (Default (When x cx) (When y cy))
```

for merging two streams by picking elements from `x` every 10 time units (at $t = 0, 10, \ldots$) and elements from `y` every 10 time units, shifted by 5 (at $t = 5, 15, \ldots$).

We will show how to solve clock constraints in the spirit of Section 5.1 when instantiating the abstraction domain to "affine sets" of the form $\{\!|\text{a t + b}|\!\}$, where a and b are natural number constants (not variables!). The notation $\{\!|\text{a t + b}|\!\}$ is meant to represent the set $\{n | \exists t.n = at + b\}$. For readability, we will confound sets and set representations, thus writing X for @text "ClOf(x)" and $\{\!|\text{a t + b}|\!\}$ instead of @text "ClWhen(Affset(a, b))", etc.

The solver proceeds in stages, first eliminating bound, then free variables, and finally solving constraints among affine sets.

Eliminating bound variables of the form `let` $X = e$ `in` c. Here, `let` is `ClLet` of Section 5.1, e a clock expression and c a clock statement. We eliminate `let`s by progressively substituting e for X in c, starting from the innermost `let`s to avoid problems with bound variables. This transformation preserves interpretations. We are left with a constraint that is a conjunction of equalities of expressions $e_0 = e_1$. We rewrite these as conjunctions of subset constraints: $e_0 \subseteq e_1 \wedge e_1 \subseteq e_0$.

Massaging subset constraints: Given a subset constraint $e \subseteq e'$, we convert e into a disjunction of conjunctions $\bigcup_i \bigcap_j e_{i,j}$, then rewrite $\bigcup_i \bigcap_j e_{i,j} \subseteq e'$ into the conjunction $\bigwedge_i (\bigcap_j e_{i,j} \subseteq e')$. We proceed dually with e', which is converted to a conjunction of disjunction and similarly decomposed. Eventually, $e \subseteq e'$ is transformed into an equivalent conjunction of subset constraints $\bigcap_j e_j \subseteq \bigcup_k e_k$. We subsequently only work with constraints having this form.

Eliminating free variables: This transformation is not an equivalence transformation, but it preserves validity of the constraints. We progressively eliminate all variables on the left of $\subseteq$, then all variables on the right. There are several cases:

- Variable X occurs on both sides of $\subseteq$: Rewrite $X \cap e \subseteq X \cup e'$ to "true" (similarly if X is negated on both sides)
- Variable X occurs on the left of $\subseteq$ only, or negated in e': Rewrite $X \cap e \subseteq e'$ to $e \subseteq e'$.
- Variable X occurs on the right of $\subseteq$ only, or negated in e: Rewrite $e \subseteq X \cup e'$ to $e \subseteq e'$.

Since we are interested in preservation of validity, the last step can be justified by interpreting X as the empty set. The other cases are similar.

Solving inequalities of affine set expressions: We have now reduced the constraints to the form $\bigcap_j e_j \subseteq \bigcup_k e_k$, where the e_j and e_k are affine sets. There are two ways to proceed:

- We can use set comprehension to dissolve the set operations. For example, $\{\!|at + b|\!\} \subseteq \{\!|ct + d|\!\}$ would become $\forall n.(\exists t.n = at + b) \rightarrow (\exists t.n = ct + d)$, leaving us with a formula of Presburger arithmetic that could be solved with an appropriate decision procedure.

– We transform $\bigcap_j e_j \subseteq \bigcup_k e_k$ to $\bigcap_j e_j \cap -(\bigcup_k e_k) \subseteq \{\}$, thus to $\bigcap_j e_j \cap (\bigcap_k -e_k) \subseteq \{\}$. Affine sets are closed under intersection, complementation and finite union. In the present case, first get rid of complementation, using the equivalence (for $b < a$):

$$-\{\!|at + b|\!\} = (\bigcup_{c \in \{0 \ldots (a-1)\} - \{b\}} \{\!|at + c|\!\})$$

do massaging of subset constraints as described above and finally eliminate intersections of the form $\{\!|a_0 t + b_0|\!\} \cap \{\!|a_1 t + b_1|\!\}$. The latter can be calculated easily for some special cases, which are easy to produce: if $b_0 = 0$ and $gcd(a_0, a_1) = 1$, then the intersection is $\{\!|(a_0 a_1)\ \mathrm{t} + (a_1 n + b_1)|\!\}$, provided n is the least solution of $\exists n_0 . a_0 n_0 = (a_1 n + b_1)$ (the intersection is empty if no such solution exists).

The second procedure seems less heavy-weighted than a general quantifier elimination procedure for Presburger arithmetic, but experimentation will have to show which procedure is preferable.

7 Conclusion

This paper has presented a type soundness proof for synchronous languages in the style of Signal / Polychrony. The approach differs substantially from previous treatments of the subject [Bes92], in that it poses the problem of clock type checking as a problem of solving set constraints. The mechanical verification has allowed to highlight potential synchronization problems in previous language definitions.

We intend to instantiate of our framework with other type systems arising in embedded real time systems, such as time-bounded computations, by modelling worst-case execution times. Our solver for periodic clock constraints is loosely inspired by well-known set constraint solvers [Aik99], with a special adaptation to afine sets, as described in Section 6. A comparison with recent developments [KR07] is still extant.

As noted in Section 5.3, in the setting of a relational semantics for synchronous languages, the notion of type soundness is relatively weak. We are about to strengthen the result, by introducing a notion of "realizability": under which conditions does a program have a model? We hope that a constructive proof will pave the way for the compiler verification of synchronous languages.

References

Aik99. Alexander Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35:79–111, 1999.

Ber00. Gérard Berry. *The Esterel Language Primer*. Esterel Technologies, version v5.91 edition, 2000.

Bes92. Loïc Besnard. *Compilation de Signal: horloges, dépendances, environnement*. PhD thesis, Université Rennes, 1992.

BH01. Sylvain Boulmé and Grégoire Hamon. Certifying synchrony for free. In Robert Nieuwenhuis and Andrei Voronkov, editors, *LPAR*, volume 2250 of *Lecture Notes in Computer Science*, pages 495–506. Springer, 2001.

Bro93. Manfred Broy. Functional Specification of Time Sensitive Communicating Systems. In *ACM Transactions on Software Engeneering and Methodology 2:1*, pages 1 – 46, 1993.

CDC00. Cécile Canovas-Dumas and Paul Caspi. A PVS proof obligation generator for Lustre programs. *Lecture Notes in Computer Science*, 1955:179–??, 2000.

CDE+06. Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. N-synchronous Kahn networks: a relaxed model of synchrony for real-time systems. *ACM SIGPLAN Notices*, 41(1):180–193, January 2006.

FBLP08. J. Forget, F. Boniol, D. Lesens, and C. Pagetti. A multi-periodic synchronous data-flow language. In *11th IEEE High Assurance Systems Engineering Symposium (HASE08)*, December 2008.

GTL03. Paul Le Guernic, Jean-Pierre Talpin, and Jean-Christophe Le Lann. Polychrony for system design. *Journal of Circuits, Systems, and Computers*, 12(3):261–304, 2003.

Hal98. N. Halbwachs. Synchronous programming of reactive systems, a tutorial and commented bibliography. In *Tenth International Conference on Computer-Aided Verification, CAV'98*, Vancouver (B.C.), June 1998. LNCS 1427, Springer Verlag.

Hal05. Nicolas Halbwachs. A synchronous language at work: the story of Lustre. In *Third ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2005. MEMOCODE '05. Third ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2005. MEMOCODE '05.*, pages 3 – 11, Verona Italy, 07 2005.

HCRP91. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

KR07. Viktor Kuncak and Martin C. Rinard. Towards efficient satisfiability checking for boolean algebra with presburger arithmetic. In *CADE*, pages 215–230, 2007.

MP09. Louis Mandel and Florence Plateau. Abstraction d'horloges dans les systèmes synchrones flot de données. In *Vingtièmes Journées Francophones des Langages Applicatifs (JFLA 09)*, Saint-Quentin sur Isère, France, February 2009.

Now99. David Nowak. *Spécification et preuve de systèmes réactifs*. PhD thesis, Rennes 1, October 1999.

NPW02. Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL. A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer Verlag, 2002.

OTBG08. Julien Ouy, Jean-Pierre Talpin, Loïc Besnard, and Paul Le Guernic. Separate compilation of polychronous specifications. *Electr. Notes Theor. Comput. Sci*, 200(1):51–70, 2008.

PM08. Christine Paulin-Mohring. *From Semantics and Computer Science: Essays in Honor of Gilles Kahn*, chapter A constructive denotational semantics for Kahn networks in Coq. Cambridge University Press, 2008.

Pou06. Marc Pouzet. *Lucid Synchrone*. http://www.lri.fr/~pouzet/lucid-synchrone/, April 2006.

Sch09. K. Schneider. The synchronous programming language Quartz. Internal Report (to appear), Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, 2009.

Spi08. Maria Spichkova. Focus on Isabelle: From specification to verification. In *Theorem Proving in Higher-Order Logics (TPHOLs 2008)*, 2008.