

Logical Clock Constraint Specification in PVS

Qingguo Xu, Robert de Simone, Julien Deantoni

▶ **To cite this version:**

Qingguo Xu, Robert de Simone, Julien Deantoni. Logical Clock Constraint Specification in PVS. [Research Report] 8748, Inria Sophia Antipolis. 2015, pp.11. hal-01192839

HAL Id: hal-01192839
https://inria.hal.science/hal-01192839

Submitted on 3 Sep 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Logical Clock Constraint Specification in PVS

Qingguo Xu , Robert de Simone , Julien Deantoni

RESEARCH REPORT

N° 8748

25/06/2015

Project-Team AOSTE

ISSN 0249-6399

Qingguo Xu[1†], Robert de Simone[2],

Julien Deantoni[3]

Project-Teams Aoste

Research Report N° 8748 — 25/06/2015 — 11 pages.

Abstract: The Clock Constraint Specification Language (CCSL), first introduced as a companion language for Modeling and Analysis of Real-Time and Embedded systems (MARTE), has now evolved beyond the time specification of MARTE, and has become a full-fledged domain specific modeling language widely used in many domains. This report demonstrates the encoded PVS (Prototype Verification System) theories for interpreting clock relation and clock expression based on schedules as a sequence of clock set. In order to ensure the correctness of the encodings, we prove some interesting properties about the clock constraint. Finally, we give an example to illustrate the approach.

Key-words: CCSL, Syntax & semantics, PVS

1 School of Computer Engineering and Science, Shanghai University – qgxu@mail.shu.edu.cn

2 INRIA Sophia Antipolis Méditerranée AOSTE – Robert.de_simone@inria.fr

3 University of Nice Sophia Antipolis, University of Nice Sophia Antipolis, – Julien.Deantoni@plytech.unice.fr

† This work is partially supported by the Natural Science Foundation of China (Grant No. 61170044).

1. Introduction 1
2. Clock Relations & Expressions Interpretation 2
2.1 Preliminaries 2
2.2 Clock Relations interpretation 4
2.3 Clock Expressions interpretation 4
3. Clock specification & its Components 6
3.1 Clock as ADT 6
3.2 Specification as ADT 7
3.3 CCSL specification' s interpretation 7
4. Bounded Relation 8
5. Case Study 9
6. Related Work and Conclusion 10

1. Introduction

The Unified Modeling Language (UML) Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE)[1], adopted in November 2009, has introduced a *Time Model* [2] that extends the informal Simple Time of UML2. This time model is general enough to support different forms of time (discrete or dense, chronometric or logical). Its so-called clocks allow enforcing as well as observing the occurrences of events and the behavior of annotated UML elements. The *Time Model* comes with a companion language named the Clock Constraint Specification Language (CCSL) [3] defined in the annex of the MARTE specification. Initially devised as a language for expressing constraints between clocks of a MARTE model, CCSL has evolved and has been developed independently of the UML. CCSL is now equipped with a formal semantics [3] and is supported by a software environment (TimeSquare [4]) that allows for the specification, solving, and visualization of clock constraints.

Lots of works have been published on the modeling capabilities offered by MARTE, much less on verification techniques supported. Inspired by the works about state-based semantics interpretation for the kernel CCSL operators [5]. This report focuses on formalizing CCSL semantics and then defining the CCSL specification by Abstract Data Type (ADT)[6] in terms of syntax of CCSL.

Section 2 introduces a state-transition based semantics for CCSL in PVS language [7]. Section 3 shows how to express a given CCSL specification by an ADT constant, and the semantics on a set of schedules. Section

4 proposes a method to analyze boundedness of a CCSL specification. A case is investigated to determine its boundedness in section 5. Finally, section 6 makes a comparison with related works, concludes the contribution, and outlines some future works.

2. Clock Relations & Expressions Interpretation

This section briefly formalizes the logical time model [2] of the Clock Constraint Specification Language (CCSL) in PVS language. A technical report [3], [5] and it latest update [8] describes the syntax and the semantics of a kernel set of CCSL constraints. We don't give again the formal definitions for some clock relations and clock expressions, which can be found in section 3 in report [5]. One can cross refer to it if you have difficulties in understanding some logic semantics in this section.

The notion of multiform logical time has first been used in the theory of synchronous languages and its polychronous extensions. CCSL is a formal declarative language to specify polychronous clock specification. It provides a concrete syntax to make the clocks and clocks constraints first-class citizens of UML-like models. Clocks in CCSL are used to measure the number of occurrences of events in a system. Logical clocks replace physical times by a logical sequencing. A CCSL specification do not need for clocks to be relative to a global physical time. They are by default independent of each other and what matter is the partial ordering of their ticks induces by the constraints between them.

A clock belongs to a clock set $\mathcal{C}$. A *clock c* can be seen as a totally ordered set of *instants*, $\square_c$. In the following, *i* and *j* are instants. A *time structure* is a set of clocks $\mathcal{C}$ and a set of relations on instants $\square = \bigcup_{c\in\mathcal{C}} \square_c$. CCSL considers two kinds of relations: *causal* and *temporal* ones. The basic causal relation is causality/dependency, a binary relation on $\square$: $\preccurlyeq \subset \square \times \square$. $i \preccurlyeq j$ means *i* causes *j* or *j* depends on *i*. $\preccurlyeq$ is a pre-order on $\square$, i.e., it is reflexive and transitive. The basic temporal relations are precedence ($\prec$), coincidence ($\equiv$), and exclusion (#), three binary relations on $\square$. For any pair of instants $(i, j) \in \square \times \square$

In a time structure, $i \prec j$ means that the only acceptable execution traces are those where *i* occurs strictly before *j* (*i* precedes i). $\prec$ is transitive and asymmetric. $i \equiv j$ imposes instants *i* and *j* to be coincident, i.e., they must occur at the same execution step, both of them or none of them. $\equiv$ is an equivalence relation, i.e., it is reflexive, symmetric and transitive. $i \# j$ forbids the coincidence of the two instants, i.e., they cannot occur at the same execution step. # is both irreflexive and symmetric. A consistency rule is enforced between causal and temporal relations. $i \preccurlyeq j$ can be refined either as $i \prec j$ or $i \equiv j$, but *j* can never precede *i*. CCSL defines a concrete syntax to specify instant relation or more generally clock relations, which represent infinitely many instant relations.

In this paper, we consider discrete sets of instants only, so that the instants of a clock can be indexed by natural numbers. For a clock $c \in \mathcal{C}$, and for any $k \in \mathbb{N}_{>0}$, $c[k]$ denotes the k^{th} instant of *c*.

During the execution of a system, an execution step is defined at a given step, every clock in $\mathcal{C}$ can tick or not according to the constraints defined in the specification. A schedule captures what happens during one particular execution.

2.1 Preliminaries

1. (**Schedule**): A *schedule* over clock set $\mathcal{C}$ is defined as a function *Sched* : $\mathbb{N}_{>0} \rightarrow 2^{\mathcal{C}}$. ■

For a given schedule, the configurations of the clocks tell us the advance of the clocks, relative to the others. We can define schedule by a sequence of set of clock as a parameter introduced in a PVS theory:

```
basic_def[Clock: TYPE]: THEORY BEGIN
  c, c1, c2: VAR Clock
  SCHEDULE: TYPE = sequence[set[Clock]]
  sgm: VAR SCHEDULE
```

sequence[set[Clock]] is a prelude function defined as $\mathbb{N} \rightarrow 2^{Clock}$ in PVS[9].

2. (**Clock configuration**): For a given schedule σ, clock $c \in \mathcal{C}$ and a natural number $n \in \mathbb{N}$, the configuration $\chi_\sigma : \mathcal{C} \times \mathbb{N} \rightarrow \mathbb{N}$ is defined recursively as:

$$\chi_\sigma(c,n) = \begin{cases} 0, & if \ \ n = 0 \\ \chi_\sigma(c,n-1), & if \ \ c \notin \sigma(n) \\ \chi_\sigma(c,n-1)+1, & if \ c \in \sigma(n) \end{cases} \qquad \text{(F.1)} \quad \blacksquare$$

Clock configuration shown below is a recursive function, where *measure n* is used to ensure the termination of calculation with the deceasing of n. Lemmas *Config_nonDec* and *Config_Invar* shows two properties about schedule.

```
n : VAR nat
config(sgm)(c)(n): RECURSIVE nat = IF n = 0 THEN 0 ELSE
LET pre = config(sgm)(c)(n - 1) IN IF sgm(n)(c) THEN pre + 1 ELSE pre ENDIF
   ENDIF MEASURE n
Config_nonDec: LEMMA  FORALL (i, j: nat): i <= j => config(sgm)(c)(i) <= config(sgm)(c)(j)
Config_Invar: LEMMA FORALL (m: upfrom(n)): config(sgm)(c)(m) = config(sgm)(c)(n) IFF
      FORALL (i: subrange(n+1, m)): NOT sgm(i)(c)
```

The former can be proved by mathematic induction scheme, and the latter can be proved via the former and the induction. Proof for *Config_nonDec* is completed by the following proof scripts:

```
%|- Config_nonDec : PROOF
%|- (spread (induct "j") ((grind) (then (skeep*) (grind))))
%|- QED
```

Hereafter, we don't give the proof scripts for every formulas because they can be seen in the attached PVS dump file. We may sketch the proof method for some of them. One wants to know in detail need also refer to Prover Guide [10].

For a clock $c \in \mathcal{C}$, and a step $n \in \mathbb{N}$, $\chi_\sigma(c,n)$ counts the the number of times the clock c has ticked at step n for the given schedule σ. Therefore, the value of $\chi_\sigma(c,n)$ denotes the index of a certain instant for clock c. Over a schedule σ, c can tick $k>0$ times if and only if $\exists n \in \mathbb{N}_{>0}, \chi_\sigma(c,n) = k$.

For a given schedule σ, and a clock $c \in \mathcal{C}$, here we interpret $\mathbb{I}_c$ as $\mathbb{I}_{c,\sigma} = \{i : \mathbb{N}_{>0} \mid c \in \sigma(i)\}$, which is a subset of $\mathbb{N}_{>0}$, containing and only containing the step i such that $\sigma(i)$ includes the clock c. A step $i \in \mathbb{I}_{c,\sigma}$ coincides with the $\chi_\sigma(c, i)^{th}$ instant $c[\chi_\sigma(c, i)]$, i.e., $i \equiv c[\chi_\sigma(c, i)]$ if $i \in \mathbb{N}_{c,\sigma}$.

The following *I?(sgm)(c)* defines the Instant set of clock c along a schedule. *idx(sgm)(c)(I: (I?(sgm)(c)))* defines the index of a certain Instant I of clock c. *idx_bij* juduges one-to-one relation between Instant set and the corresponding index set for every clock. *kth_ID* describes the fact that a clock c's k^{th} Instant is step i if and only if c's configuration is k at step i. Note that the datatype of i and k determines that the exists of index k and c ticks at step i exactly.

```
Index?(sgm)(c)(k: posnat): bool = EXISTS (n: posnat): k = config(sgm)(c)(n)  % Index 1,2,3,...
I?(sgm)(c)(n:posnat): bool = sgm(n)(c)  % Instants for c, (I_c)
idx(sgm)(c)(I: (I?(sgm)(c))): (Index?(sgm)(c)) = config(sgm)(c)(I)
idx_bij: JUDGEMENT idx(sgm)(c) HAS_TYPE  (bijective?[(I?(sgm)(c)), (Index?(sgm)(c))])
kth_Instant(sgm)(c)(x: (Index?(sgm)(c))): (I?(sgm)(c)) = inverse(idx(sgm)(c))(x)
```

```
kth_ID: LEMMA FORALL (k: (Index?(sgm)(c))), (i: (I?(sgm)(c))):
                                    kth_Instant(sgm)(c)(k) = i IFF config(sgm)(c)(i) = k
```

To prove kth_ID, the most of works requires us to prove the existence of index and instant. The existence can be guaranteed by the following frequently used auxiliary lemma. Pre_tick_ex, which is proved using Config_Invar, the monotony of config ensured by definition in (F.1).

```
Pre_tick_ex: LEMMA  FORALL (k: posnat): config(sgm)(c)(n) = k =>
(EXISTS (j: below[n]):  (config(sgm)(c)(j) = k - 1 AND config(sgm)(c)(j + 1) = k))
```

2.2 Clock Relations interpretation

We can formally define clock relations based on state-based (upon a schedule defined over clock set sequence) representation given in [5].

```
% Synchronous relations
SubClock?(sgm)(c1, c2): bool = FORALL (n:posnat): sgm(n)(c1) IMPLIES sgm(n)(c2)
Exclusion?(sgm)(c1, c2): bool = FORALL (n: posnat): NOT sgm(n)(c1) OR NOT sgm(n)(c2)
Coincidence?(sgm)(c1, c2): bool = FORALL (n: posnat): sgm(n)(c1) IFF sgm(n)(c2)
%Asynchronous relations
Causal?(sgm)(c1, c2): bool =   FORALL n : config(sgm)(c1)(n) >= config(sgm)(c2)(n)
Precedence?(sgm)(c1, c2): bool = FORALL n: config(sgm)(c1)(n) = config(sgm)(c2)(n) => NOT
sgm(n+1)(c2)
% Precedence implies causality
 Preced_Causal: PROPOSITION Precedence?(sgm)(c1, c2) => Causal?(sgm)(c1, c2)
% Subclocking implies causality
Sub_Causal: PROPOSITION SubClock?(sgm)(c1, c2) => Causal?(sgm)(c2, c1)
PrecedFun: LEMMA FORALL (a, b: Clock): Precedence?(sgm)(a, b) IFF
    (EXISTS (h: (injective?[(I?(sgm)(b)), (I?(sgm)(a))])): (FORALL (i, j: (I?(sgm)(b))): i < j => h(i) < h(j))
AND (FORALL (i: (I?(sgm)(b))): h(i) < i)
```

Some properties, shown as the lemmas, such as Preced_Causal and Sub_Causal, among these clock relations are also deduced and given. Their proof is a pure translation the ideas in [5] into PVS proof scripts.

We can establish the semantics' equivalence based on between states and instants. PrecedFun, as an example shows the equivalence, can be proved by instancing h(i) = j such that instants i and j has the same index , i.e., config(sgm)(b)(i) = config(sgm)(a)(j). We also need kth_ID above in the process of proof.

2.3 Clock Expressions interpretation

The followings are some clock expressions defined as the constraint between output clock and input one(s) with the given argument if any. Their formal semantics can be found in [5].

```
Union?(sgm)(u: Clock)(c1, c2): bool =   FORALL (n: posnat): sgm(n)(u) IFF (sgm(n)(c1) OR sgm(n)(c2))
 Intersection?(sgm)(i: Clock)(c1, c2): bool =  FORALL (n: posnat): sgm(n)(i) IFF (sgm(n)(c1) AND
sgm(n)(c2))
 Inf?(sgm)(inf: Clock)(c1, c2): bool = FORALL (n: nat):
config(sgm)(inf)(n) = max(config(sgm)(c1)(n), config(sgm)(c2)(n))
 Sup?(sgm)(sup: Clock)(c1, c2): bool = FORALL (n: nat):
config(sgm)(sup)(n) = min(config(sgm)(c1)(n) , config(sgm)(c2)(n))
Delay?(sgm)(c: Clock, d: nat)(c_del: Clock): bool = FORALL (n: nat):
```

```
config(sgm)(c_del)(n) = max(config(sgm)(c)(n) - d, 0)
SampledOn?(sgm)(trig, base: Clock)(samp: Clock): bool =  %weak version: non strict
   FORALL (n: posnat): sgm(n)(samp) IFF (sgm(n)(base) AND
 (EXISTS (j:subrange(1, n)): sgm(j)(trig) AND  (FORALL (m: subrange(j,n-1)): NOT sgm(m)(base))))
 StrictSampledOn?(sgm)(trig, base: Clock)(samp: Clock): bool =  %strict version
    FORALL (n: posnat): sgm(n)(samp) IFF (sgm(n)(base) AND
       (EXISTS (j:subrange(1, n-1)): sgm(j)(trig) AND (FORALL (m: subrange(j+1,n-1)): NOT
sgm(m)(base))))
END basic_def
```

We can also define the filtering expression, with the argument defined as binary word datatype *BW* shown below in PVS theory *binword* (shown below). Meanwhile, some associated datatypes and some frequently used constant are also given. PVS delectation *CONVERSION w, v, fin , inf,* as one of type conversion mechanism, supplies the possibilities to write argument of *BW* type as the one of type *finseq[nbit]* or *sequence[nbit].* One can refer to the technique report [6] if there are some difficulties to understand. Lemma *Periodic_uvomg* tell us the fact that the periodicity of a regular binary word given by the form of $u.(v)^{\omega}$, $u_0,u_1,\ldots,u_n\ ,v_0,v_1,\ldots,v_m,\ v_0,v_1,\ldots,v_m,\ v_0,v_1,\ldots v_m,\ldots$ with $|u|=n$ and $|v|=m$.

```
binword: THEORY BEGIN
   BW: DATATYPE  BEGIN
    fin(v: finseq[nbit]): fin?
    inf(w: sequence[nbit]): inf?
  END BW
 CONVERSION w, v, fin , inf
 zero: (fin?) = (# length := 1, seq := LAMBDA (i: below(1)): 0 #)
one: (fin?) = (# length := 1, seq := LAMBDA (i: below(1)): 1 #);
 ^(v: (fin?), n: nat): (fin?) = LET l = v`length * n IN
     (# length := l, seq := LAMBDA (i: below(l)): v`seq(rem(v`length)(i)) #)
 NonEmptyFinBW: TYPE = {u | u`length > 0}
 v: VAR NonEmptyFinBW
 omg(v): (inf?) = LAMBDA (i: nat): v`seq(rem(v`length)(i));
 o(u: (fin?), bw: BW): BW = CASES bw OF fin(v): fin(u o v),
     inf(w): inf(LAMBDA (i: nat): LET lu = u`length IN
             IF i < lu THEN u`seq(i) ELSE w(i - lu) ENDIF) ENDCASES
 uv_omg(u, v: (fin?)): (inf?) = u o omg(v)
 Periodic_uvomg: LEMMA FORALL (w:(inf?),u,v:(fin?)): w = uv_omg(u, v) =>
(FORALL (j: below(v`length), n: nat):w(w)(n * v`length + j + u`length) = w(w)(j + u`length))
END binword
```

By importing theories *basic_def* and *binword*, we can define clock expression *FilterBy?*, which has the periodic behavior *Periodic_Config*.

```
clock_expr[Clock: TYPE]: THEORY BEGIN
 IMPORTING basic_def[Clock]
 IMPORTING binword
 sgm: VAR SCHEDULE
```

```
    FilterBy?(sgm)(c: Clock,w: (inf?))(f: Clock): bool = FORALL (n: posnat): sgm(n)(f)  IFF
        (IF sgm(n)(c) THEN w(w)(config(sgm)(c)(n) - 1) = 1 ELSE FALSE  ENDIF)

    FilterBy?(sgm)(c: Clock,u: (fin?), v: NonEmptyFinBW)(f: Clock): bool =
      FilterBy?(sgm)(c,uv_omg(u, v))(f)
    Periodic?(sgm)(base: Clock, period: posnat, offset: nat)(c: Clock):  bool =
      FilterBy?(sgm)(base,zero ^ offset, one o zero ^ (period - 1))(c)

    Periodic_Config: LEMMA FORALL (base: Clock), (f: Clock), (period: posnat), (offset: nat):
      Periodic?(sgm)(base, period, offset)(f) =>
       (FORALL (n: nat): LET nb = config(sgm)(base)(n), nf = config(sgm)(f)(n) IN
          IF nb <= offset THEN nf = 0 ELSE nf = div(nb - offset - 1, period) + 1 ENDIF)
  END clock_expr
```

3. Clock specification and its Components

First, we express clock type *AClock (all clock)* including explicit clock and implicit ones. Second, we give the clock relation defined over *AClock*.

3.1 Clock as ADT

We can distinguish two kinds of clock like 0.

(**Clock set**) An element in the clock set $\mathcal{C}$ can be given by the specification writer explicitly (**explicit clock**), or by one of the following clock expressions (**implicit clock**):

$$Clock ::= ref(a) \mid a + b \mid a * b \mid sup(a, b) \mid inf(a, b) \mid dealy(a , n) \mid SampledOn(a, b) \mid StrictSampledOn(a, b) \mid FilteredBy(a, u, v) \quad \text{(F.2)}$$

where $a, b \in \mathcal{C}$ are clocks, $u, v \in (0+1)^*$ are finite binary words, and $n \in \mathbb{N}$ is a natural number. ■

We can express CCSL specification as set of clock relation between clocks which are explicitly given (via constructor *ref*) or derived by clock defined by clock expression.

```
ccsl_spec[Clock: TYPE]: THEORY BEGIN
 AClock: DATATYPE BEGIN
   ref(clk: Clock): clock?
   +(a, b: AClock): union?
   *(a, b: AClock): intersection?
   /\(a, b: AClock): inf?
   \/(a, b: AClock): sup?
   delay(c: AClock, d: nat): delay?
   sampledon(trig, base: AClock): sampledon?
   strictsampledon(trig, base: AClock): strictsampledon?
   filterby(c: AClock, u: finseq[nbit],v:{vv:finseq[nbit]|vv`length>0}): filterby?
 END AClock
 CONVERSION clk, ref
```

Via PVS conversion mechanism in PVS language [7], the declaration "*CONVERSION clk, ref*" allows us to choose *ref*(*c*) or *c* freely for a given clock *c*.

3.2 Specification as ADT

(**CCSL specification**): A CCSL specification $\mathcal{SPEC}$ is a pair <$\mathcal{C}$, *CConstr* >, where $\mathcal{C}$ is a set of clocks, *CConstr* is a set of formulae (including SubClock, Causality, Precedence, Exclusion and Coincidence) used to specify the relations among the clocks in the set $\mathcal{C}$. ■

Clock constrains are finally expressed by clock relation between *AClock*s including explicit clock and implicit ones. We can use the constructor & or the recognizer *AND?* to collect all the necessary relations that the specification must hold.

```
SPEC: DATATYPE  BEGIN
  <|(a, b: AClock): sub?
  <=(a, b: AClock): causal?
  <(a, b: AClock): precedence?
  #(a, b: AClock): exclusion?
  ==(a, b: AClock): coincidence?
  &(s1, s2: SPEC): and?
 END SPEC;
~(a, b:AClock): SPEC = a < b & b < delay(a, 1)
sychronize(a, b:AClock): SPEC = a < delay(b, 1) & b < delay(a, 1)
RelatedClock(e: AClock): RECURSIVE setof[AClock] = LAMBDA (c: AClock): CASES e
    OF ref(clk): c = e,
      delay(a, d): RelatedClock(a)(c),
      sampledon(trig, base): RelatedClock(trig)(c) OR RelatedClock(base)(c),
      strictsampledon(trig, base):  RelatedClock(trig)(c) OR RelatedClock(base)(c),
      filterby(a, u,v): RelatedClock(a)(c)   ELSE RelatedClock(a(e))(c) OR RelatedClock(b(e))(c)
    ENDCASES MEASURE e BY <<;
ConsideredClock(s: SPEC): RECURSIVE setof[AClock] = LAMBDA (c: AClock): CASES s
  OF &(s1, s2): ConsideredClock(s1)(c) OR ConsideredClock(s2)(c)  ELSE
LET a = a(s), b = b(s) IN c = a OR c = b OR RelatedClock(a)(c) OR RelatedClock(b)(c)
    ENDCASES MEASURE s BY <<;
```

For a given CCSL specification *s*, *ConsideredClock(s)* collects all the explicit and implicit clocks occur in *s* by calling *RelatedClock* used to collect clocks occurs in clock expressions.

3.3 CCSL specification's interpretation

By importing PVS theory *clock_expr* (indirectly importing theories *basic_def* and *binword*), we can associate the different clock relation constructors with the corresponding interpretations defined in PVS theory *basic_def*. If there are some clocks occurs in CCSL specification, we may call the semantics of clock expression define in PVS theory *basic_def or clock_expr*. Up to now, we have completed the syntax and semantics model of CCSL specification in PVS. From the definition below, we can see it is necessary to recursively call clock expression's interpretation *|=(sgm, c)* when we try to interpret the clock relation *|=(sgm, s)*.

```
IMPORTING clock_expr[AClock]
sgm: VAR SCHEDULE[AClock]
s: VAR SPEC;  c: VAR AClock;
|=(sgm, c): RECURSIVE bool = (NOT and?(s) => (sgm |= a(s)) AND (sgm |= b(s))) AND
CASES c OF ref(clk): Coincidence?(sgm)(c, clk),
```

```
        +(a, b): Union?(sgm)(c)(a, b),
        *(a, b): Intersection?(sgm)(c)(a, b),
        /\(a, b): Inf?(sgm)(c)(a, b),
        \/(a, b): Sup?(sgm)(c)(a, b),
        delay(a, d): Delay?(sgm)(a, d)(c),
        sampledon(trig, base): SampledOn?(sgm)(trig, base)(c),
        strictsampledon(trig, base): StrictSampledOn?(sgm)(trig, base)(c),
        filterby(a, u,v): FilterBy?(sgm)(a, uv_omg(u,v))(c)
    ENDCASES  MEASURE  c BY <<;
 |=(sgm, s): RECURSIVE bool = IF and?(s) THEN (sgm |= s1(s)) AND (sgm |= s2(s))
     ELSE (sgm |= a(s)) AND (sgm |= b(s)) AND CASES s OF
<|(a, b): SubClock?(sgm)(a, b),
        <=(a, b): Causal?(sgm)(a, b),
        <(a, b): Precedence?(sgm)(a, b),
        #(a, b): Exclusion?(sgm)(a, b),
        ==(a, b): Coincidence?(sgm)(a, b)
      ENDCASES  ENDIF MEASURE s BY <<;
END ccsl_spec
```

4. Bounded Relation

We can analyze some interesting features owned by the CCSL specification itself. In this section, we take the boundedness as an example to show how to do the analysis.

For a given CCSL specification, if the difference between the speeds of two clocks $a, b \in \mathcal{C}$ is limited in an allowed boundary, we say the clock pair (a, b) has a bounded relation.

(**Bounded relation)** For a given clock set $\mathcal{C}$, two clocks $a, b \in \mathcal{C}$, and a schedule σ over $\mathcal{C}$, a and b has the bounded relation with a given boundary $m \in \mathbb{N}$, denotes $|a, b| \leq m$:

$$\sigma \vDash |a, b| \leq m \ \textbf{\textit{iff}} \ \forall n \in \mathbb{N}, |\chi_\sigma(a, n) - \chi_\sigma(b, n)| \leq m$$

m (resp. $-m$) is called *upper* (resp. *lower*) *bound*. ■

(**Bounded Specification)** For a given CCSL specification $\mathcal{SPEC}$ = <$\mathcal{C}$, *CConstr*>, $\forall a, b \in \mathcal{C}$, $r \in$ *CConstr*, $\mathcal{SPEC}$ is **bounded** if and only if any clock pair has the bounded relation:

$$\forall \sigma, \sigma \vDash \mathcal{SPEC} \Longrightarrow \exists m \in \mathbb{N}, \sigma \vDash |a, b| \leq m \ ■$$

0 (resp. 0) can be expressed *maxDrift* (resp. *bounded_ccsl*) in PVS as below. *ConsideredClock* provides the possibility to get the clock set from the given specification rather than specify them explicitly.

```
delta(sgm)(n: nat)(c1, c2: AClock): int = LET X = config(sgm) IN X(c1)(n) - X(c2)(n)
maxDrift(sgm)(a, b: AClock)(m: nat): bool =
  FORALL (n: nat): LET dif = delta(sgm)(n)(a, b) IN -m <= dif AND dif <= m
bounded_ccsl(s: SPEC)(sgm): bool = (sgm |= s) =>
    (FORALL (a, b: (ConsideredClock(s))): EXISTS (m: nat): maxDrift(sgm)(a, b)(m))
```

If we check every clock pairs among all clocks in $\mathcal{C}$ to decide whether a specification is divergence-free or not, there are $\binom{|\mathcal{C}|}{2} = |\mathcal{C}| \times (|\mathcal{C}| - 1)/2$ pairs need to be checked. The number of checks then totals to $\mathcal{O}(|\mathcal{C}|^2)$. But in practice many checks can be safely neglected when the bounded relation is implied by the already

checked one according to the following theorem *boundsEx_ccsl*. *boundsEx_ccsl* is proved by instancing the boundedness of *top* and *bot* as that of all the clock pairs.

```
bounds_exists(s): bool = FORALL sgm: EXISTS (bot, top: AClock, m: nat): FORALL sgm:  (sgm |= s) =>
    ((sgm |= (bot <= top)) AND maxDrift(sgm)(bot, top)(m) AND
      (FORALL (c: (ConsideredClock(s))): sgm |= (bot <= c & c <= top)))
boundsEx_ccsl: THEOREM bounds_exists(s) IMPLIES bounded_ccsl(s)(sgm)
```

bounds_exists shown above is used to check whether we can find the fastest clock *bot* and the slowest one *top* such that all other clocks' speed lie between them. If so, theorem *boundsEx_ccsl* tell us the specification is bounded.

5. Case Study

To illustrate the PVS approach for analyzing CCSL specification, we take an example inspired by[11], that was was used for flow latency analysis on Architecture Analysis and Design Language(AADL) specifications. However, with CCSL we are conducting different kinds of analyses.

FIGURE 1 considers a simple application described as a UML activity. This application captures two inputs *in1* and *in2*, performs some calculations (*step1*, *step2* and *step3*) and then produces a result *out*. This application has the possibility to compute *step1* and *step2* concurrently depending on the chosen execution platform. This application runs in a streaming-like fashion by continuously capturing new inputs and producing outputs.

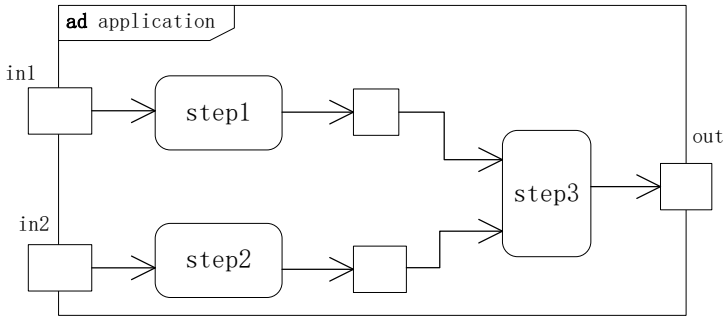


FIGURE 1 : Simple application

To abstract this application as a CCSL specification, we assign one clock to each action. The clock has the exact same name as the associated action (e.g., *step1*). We also associate one clock with each input, this represents the capturing time of the inputs, and one clock with the production of the output (*out*). The successive instants of the clocks represent successive executions of the actions or input sensing time or output release time. The basic CCSL specification is $\mathcal{SPEC}_{simp}$ = <$\mathcal{C}$,CConstr>, where $\mathcal{C}$={*in1*,*in2*,*step1*,*step2*,*step3*,*out*}, *CConstr* includes the following clock constraints*:*

$in1 \preccurlyeq step1 \wedge step1 \prec step3$ (F.3)

$in2 \preccurlyeq step2 \wedge step2 \prec step3$ (F.4)

$step3 \preccurlyeq out$ (F.5)

(F.3) specifies that step1 may begin as soon as an input *in1* is available. Executing *step3* also requires step1 to have produced its output. (F.4) is similar for *in2* and *step2*. (F.5) states that an output can be produced as soon as *step3* has executed. Note that CCSL precedence is well adapted to capture infinite FIFOs denoted on the figure as object nodes. Such a specification is clearly not bounded because of none of clock constraint can

ensure the slower clock tick. Previous work hints us we can introduce the bounded relation as (F.6) shows to bound the speed between the faster and slower clock.

$$inf(in1, in2) \sim out \qquad \text{(F.6)}$$

Based on subsection 3.3, we formalize this CCSL specification as below:

```
simpApp: THEORY  BEGIN
 Clock: TYPE = {in1, in2, step1, step2, step3, out}
 IMPORTING ccsl_spec[Clock]
 spec: SPEC =  in1 <= step1 & step1 < step3 & step3 <= out &
  in2 <= step2 & step2 < step3 & (in1 /\ in2) ~ out
 safety: THEOREM  FORALL (sgm: SCHEDULE): bounded_ccsl(spec)(sgm)
end simpApp
```

Theorem *safety* asserts all the clock pairs have the bounded relation. *safety* can be deduced via theorem *boundsEx_ccsl* by instancing *bot* = *inf*(*in1, in2*), *top* = *delay(inf*(*in1, in2*)*,1)* with the following two facts deduced from equations (F.3), (F.4), (F.5) and (F.6).

1. bounded relation with the boundary 1 between clocks *bot* and *top*,
2. causality relations $\forall c$: {c|*ConsideredClocks(spec)(c)*}, *bot* $\preccurlyeq$ c$\preccurlyeq$ *out*.

Noted that *ConsideredClocks(spec)* = {*in1*, *in2*, *step1*, *step2*, *step3*, *out*, *inf*(*in1, in2*), *delay(inf*(*in1, in2*),1)} by the definition in section 0. The readers who want to get the proof detail can refer to the attached PVS dump file.

6. Related Work and Conclusion

Some techniques were provided as an effort to analyze CCSL specifications. Exhaustive analysis of CCSL through a transformation into labeled transition systems has already been attempted in [12]. However, in those attempts, the CCSL operators were bounded because the underlying model-checkers cannot deal with infinite labeled transition systems.

In [13], the authors showed that even though the primitive constraints were unbounded, the composition of these primitive constraints could lead to a system where only a finite number of states were accessible. [14] defines a notion of safety for CCSL and establish a condition to decide whether a specification is safe on the transformed marked graph from CCSL.

All the above works share one common point: the specification analysis were done by some transformation and performed on the transformed target. The results were depended on the correctness and efficiency of the mechanized transformation.

Our contribution is the first try to express and verify CCSL specification using a proof assist. We provide a framework to describe a CCSL specification as an ADT and to deduce some interested properties owned by the schedule satisfied by that specification via a set of available lemmas to support.

Based on the state-based semantics of a kernel subset of CCSL, We have translated the clock constraints into PVS function in a logic form. And then we describe the relations, using PVS formula, between different kind of clock constraints. By founding the boundedness of a clock pair with the clock causality relation of all clocks, we found a method to determine whether a given CCSL specification is bounded or not.

As a future work, we plan to enrich the properties about clock constraints. For instance, how to find the potential causality conflict between clocks? How to decide whether a schedule shows periodicity against by a CCSL specification? etc. Developing some PVS proof strategies dedicated to simplify semantics

interpretation may be help to reduce the human reaction when we try to prove some properties about specification.

Bibliography

[1] OMG, "UML Profile for MATRE v1.0. Object Management Group," November 2009 ed, 2009 formal/2009-11-02.

[2] C. André, F. Mallet, R. D. Simone, "Modeling time(s)," presented at the Proceedings of the 10th international conference on Model Driven Engineering Languages and Systems, Nashville, TN, 2007.

[3] C. André, "Syntax and Semantics of the Clock Constraint Specification Language (CCSL)," Inria I3S Sophia Antipolis 15 June 2009.

[4] J. DeAntoni, F. Mallet, "TimeSquare: Treat Your Models with Logical Time," in *Objects, Models, Components, Patterns*. vol. 7304, C. Furia, S. Nanz, Eds., ed: Springer Berlin Heidelberg, 2012, pp. 34-41.

[5] F. Mallet, J.-V. Millo, Y. Romenska, "State-based representation of CCSL operators," 2013-07-19 2013.

[6] S. Owre, N. Shankar, "Abstract Datatypes in PVS," Computer Science Laboratory, SRI International, Menlo Park, C.A. December 1993.

[7] S. Owre, N. Shankar, J. M. Rushby, D. W. J. Stringer-Calvert, "PVS Language Reference," ed. Menlo Park, CA, 1999.

[8] J. Deantoni, C. André, R. Gascon, "CCSL denotational semantics," Inria I3S Sophia Antipolis 13 November 2014.

[9] S. Owre, N. Shankar, "The PVS Prelude Library," Computer Science Laboratory, SRI International, Menlo Park, C.A. March 7 2003.

[10] N. Shankar, S. Owre, J. M. Rushby, D. W. J. Stringer-Calvert, "PVS Prover Guide," ed. Menlo Park, CA, 1999.

[11] P. H. Feiler, J. Hansson, "Flow latency analysis with the architecture analysis and design language (AADL)," CMU Technical Note CMU/SEI-2007-TN-010, 2007.

[12] R. Gascon, F. Mallet, J. DeAntoni, "Logical Time and Temporal Logics: Comparing UML MARTE/CCSL and PSL," in *Temporal Representation and Reasoning (TIME), 2011 Eighteenth International Symposium on*, 2011, pp. 141-148.

[13] F. Mallet, J.-V. Millo, "Boundness Issues in CCSL Specifications," in *ICFEM 2013 - 15th International Conference on Formal Engineering Methods*, 2013, pp. 20-35.

[14] F. Mallet, J.-V. Millo, R. De Simone, "Safe CCSL Specifications and Marked Graphs," in *MEMOCODE - 11th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, 2013, pp. 157-166.



RESEARCH CENTRE
SOPHIA ANTIPOLIS
MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr
ISSN 0249-6399