Boundness issues in CCSL specifications

Frédéric Mallet[1]

Université Nice Sophia Antipolis,
Aoste team-project, INRIA/I3S, F-06902, France
Frederic.Mallet@unice.fr

Abstract. The UML Profile for Modeling and Analysis of Real-Time and Embedded systems promises a general modeling framework to design and analyze systems. Lots of works have been published on the modeling capabilities offered by MARTE, much less on verification techniques supported. The Clock Constraint Specification Language (CCSL), first introduced as a companion language for MARTE, was devised to offer a formal support to conduct causal and temporal analyses on MARTE models.
This work introduces a state-based semantics for CCSL operators and then focuses on the analysis capabilities of MARTE/CCSL and more particularly on boundness issues.
The approach is illustrated on one simple example where the architecture plays an important role. We describe a process where the logical description of the application is progressively refined to take into account the candidate execution platforms through allocation.

Keywords: Logical Time, Architecture-driven analysis, UML MARTE, Reachability analysis

1 Introduction

The UML Profile for Modeling and Analysis of Real-Time and Embedded systems [1] (MARTE), adopted in November 2009, has introduced a *Time model* [2] that extends the informal *Simple Time* of The Unified Modeling Language (UML 2.x). 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 called *Clock Constraint Specification Language* (CCSL) [3] and defined in an annex of the MARTE specification. Initially devised as a simple 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][1]) that allows the specification, resolution, and visualization of clock constraints.

[1] http://timesquare.inria.fr

MARTE promises a general modeling framework to design and analyze systems. Lots of works have been published on the modeling capabilities offered by MARTE, much less on verification techniques supported. While the initial semantics of CCSL is described as a set of rewriting rules [3], this paper proposes as a first contribution a state-based semantics for each of the kernel CCSL operators. The global semantics emerging of the parallel composition of CCSL constraints then becomes the synchronized product of the automaton of each individual constraint. Since automaton for some CCSL operators can be infinite, this requires specific attention to compute the synchronized product. The second contribution is an algorithm that builds the synchronized product. The algorithm terminates when the set of states reachable through the synchronized product is finite. The third contribution is a discussion on a sufficient condition to guarantee that the synchronized product is actually finite.

Section 2 proposes a state-based semantics for CCSL. Section 4 illustrates the use of CCSL for architecture-driven analysis. It shows how abstract representations of the application and the architecture are built and how the two models are mapped through an allocation process. Section 3 discusses boundness issues on CCSL specifications. Section 5 makes a comparison with related works.

2 A state-based semantics for CCSL operators

This section starts with a brief introduction to CCSL and then gives a formal definition of CCSL operators in terms of labeled transition systems. Some of the CCSL operators require an infinite number of states.

2.1 The Clock Constraint Specification Language

The Clock Constraint Specification Language (CCSL) has been developed to elaborate and reason on the logical time model [2] of MARTE. A technical report [3] describes the syntax and the semantics of a kernel set of CCSL constraints.

The notion of multiform logical time has first been used in the theory of Synchronous languages [5] and its polychronous extensions [6]. The use of tagged systems to capture and compare models of computations was advocated by [7]. CCSL provides a concrete syntax to make the polychronous clocks become first-class citizens of UML-like models.

A *clock* c is a totally ordered set of *instants*, $\mathcal{I}_c$. 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 $\mathcal{I} = \bigcup_{c \in \mathcal{C}} \mathcal{I}_c$. CCSL considers two kinds of relations: *causal* and *temporal* ones. The basic causal relation is causality/*dependency*, a binary relation on $\mathcal{I}$: $\preccurlyeq \subset \mathcal{I} \times \mathcal{I}$. $i \preccurlyeq j$ means i causes j or j depends on i. $\preccurlyeq$ is a pre-order on $\mathcal{I}$, *i.e.,* it is reflexive and transitive. The basic temporal relations are *precedence* ($\prec$), *coincidence* ($\equiv$), and *exclusion* ($\#$), three binary relations on $\mathcal{I}$. For any pair of instants $(i, j) \in \mathcal{I} \times \mathcal{I}$ in a time structure, $i \prec j$ means that the only acceptable execution traces are those where i occurs strictly before j (i precedes j). $\prec$ is transitive and asymmetric (reflexive and antisymmetric). $i \equiv j$ 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 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.

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 $k \in \mathbb{N}^\star$, $c[k]$ denotes the k^{th} instant of c.

2.2 CCSL clocks and relations

Definition 1. *A* Labeled Transition System *[8] over a set A of actions is defined as a tuple $\mathcal{A} = \langle S, T, s0, \alpha, \beta, \lambda \rangle$ where*

- *S is a set of* states*,*
- *T is a set of* transitions*,*
- *$s0 \in S$ is the initial state,*
- *$\alpha, \beta : T \to S$ denote respectively the* source state *and the* target state *of a transition,*
- *$\lambda : T \to A$ denotes the action responsible for a transition,*
- *the mappings $\langle \alpha, \lambda, \beta \rangle : T \to S \times A \times S$ are one-to-one so that T is a subset of $S \times A \times S$.*

In the context of CCSL, the actions are clocks. For each CCSL **clock** c, we build the Labeled Transition System $Clock_c = \langle S, T, \alpha, \beta, \lambda \rangle$ over $A_c = \{c, \epsilon\}$ such that

- $S = \{s\}$, $T = \{t, e\}$, $s0 = s$,
- $\alpha(t) = \alpha(e) = \beta(t) = \beta(e) = s$,
- $\lambda(t) = c$ and $\lambda(e) = \epsilon$.

The ϵ action allows for doing nothing. This is to allow composition with other LTSs. $Clock_a$ is given in Figure 1.a as an illustration.[2]

Definition 2. *Given n sets of actions $A_1, \ldots, A_n$, a* synchronization constraint *is a subset I of $A_1 \times \ldots \times A_n$.*

Definition 3. *If, for $i = 1, \ldots, n$, $\mathcal{A}_i = \langle S_i, T_i, s0_i, \alpha_i, \beta_i, \lambda_i \rangle$ is a labeled transition system over A_i, and if $I \subseteq A_1 \times \ldots \times A_n$ is a synchronization constraint, the* synchronized product *[8] of $\mathcal{A}_i$ with respect to I is the labeled transition system $\langle S, T, s0, \alpha, \beta, \lambda \rangle$ over the set I defined by*

- *$S = S_1 \times \ldots \times S_n$, $s0 = s0_1 \times \ldots \times s0_n$,*
- *$T = \{\langle t_1, \ldots, t_n \rangle \in T_1 \times \ldots \times T_n | \langle \lambda_1(t_1), \ldots, \lambda_n(t_n) \rangle \in I\}$,*
- *$\alpha(\langle t_1, \ldots, t_n \rangle) = \langle \alpha_1(t_1), \ldots, \alpha_n(t_n) \rangle$,*

[2] The ϵ transitions are not shown to simplify the drawings. In all the presented LTSs, it is always possible to do nothing by remaining in the same state.

- $\beta(\langle t_1, \ldots, t_n\rangle) = \langle \beta_1(t_1), \ldots, \beta_n(t_n)\rangle$,
- $\lambda(\langle t_1, \ldots, t_n\rangle) = \langle \lambda_1(t_1), \ldots, \lambda_n(t_n)\rangle$.

Synchronization constraints allow for capturing the semantics of CCSL polychronous operators. In this section, we focus on CCSL (binary) relations.

Relation 1. *Given two clocks* c1 *and* c2*, the* **coincidence** *operator c1* $\boxed{=}$ *c2 is the synchronized product of* $Clock_{c1}$ *and* $Clock_{c2}$ *with respect to the synchronization constraint* $I = \{\langle c1, c2\rangle, \langle \epsilon, \epsilon\rangle\}$ *(Fig. 1.b).*

Relation 2. *The* CCSL **subclock** *operator (c1* $\boxed{\subset}$ *c2) is the synchronized product of* $Clock_{c1}$ *and* $Clock_{c2}$ *with respect to the synchronization constraint* $I = \{\langle c1, c2\rangle, \langle \epsilon, c2\rangle, \langle \epsilon, \epsilon\rangle\}$ *(Fig. 1.c).*

Relation 3. *Figure 1.d illustrates the* CCSL **excludes** *operator (c1* $\boxed{\#}$ *c2) defined as the synchronized product of* $Clock_{c1}$ *and* $Clock_{c2}$ *with respect to the synchronization constraint* $I = \{\langle c1, \epsilon\rangle, \langle \epsilon, c2\rangle, \langle \epsilon, \epsilon\rangle\}$.

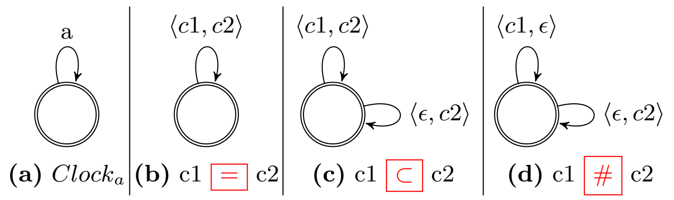
a
(a) $Clock_a$
$\langle c1, c2\rangle$
(b) c1 $\boxed{=}$ c2
$\langle c1, c2\rangle$
$\langle \epsilon, c2\rangle$
(c) c1 $\boxed{\subset}$ c2
$\langle c1, \epsilon\rangle$
$\langle \epsilon, c2\rangle$
(d) c1 $\boxed{\#}$ c2

Fig. 1. Primitive CCSL relations as Labeled Transition Systems

2.3 CCSL bounded expressions

In CCSL, expressions allow for the creation of new clocks based on existing ones. Expressions can also be represented as labeled transition systems. Union and intersection are two simple examples of CCSL expressions.

Expression 1. $u \triangleq c1 + c2$ *(u is the* **union** *of* c1 *and* c2*) is represented by the synchronized product of* $Clock_{c1}$*,* $Clock_{c2}$ *and* $Clock_u$ *with respect to the synchronization constraint* $I = \{\langle c1, c2, u\rangle, \langle c1, \epsilon, u\rangle, \langle \epsilon, c2, u\rangle, \langle \epsilon, \epsilon, \epsilon\rangle\}$ *(Fig. 2.a).*

Expression 2. $i \triangleq c1.c2$ *(i is the* **intersection** *of* c1 *and* c2*) is represented by the synchronized product of* $Clock_{c1}$*,* $Clock_{c2}$ *and* $Clock_i$ *with respect to the synchronization constraint* $I = \{\langle c1, c2, i\rangle, \langle c1, \epsilon, \epsilon\rangle, \langle \epsilon, c2, \epsilon\rangle, \langle \epsilon, \epsilon, \epsilon\rangle\}$ *(Fig. 2.b).*

Those two expressions are stateless (one state). Other expressions are stateful and require building dedicated LTS to express their semantics.

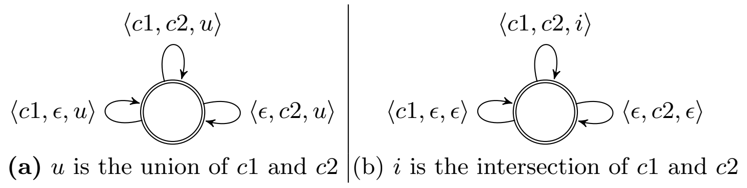
$\langle c1, c2, u\rangle$
$\langle c1, \epsilon, u\rangle$
$\langle \epsilon, c2, u\rangle$
$\langle c1, c2, i\rangle$
$\langle c1, \epsilon, \epsilon\rangle$
$\langle \epsilon, c2, \epsilon\rangle$
(a) u is the union of $c1$ and $c2$
(b) i is the intersection of $c1$ and $c2$

Fig. 2. Union and intersection of clocks

Expression 3. *The* **binary delay** *(delayed $\triangleq$ base $\$$ n) is represented by a dedicated labeled transition system $Delay(n) = \langle S, T, s0, \alpha, \beta, \lambda\rangle$ over $A = \{init, steady, \epsilon\}$ with $n + 1$ states such that*

- $S = \{d_0, d_1, \ldots, d_n\}$, $T = \{t_0, t_1, \ldots, t_n, e_0, \ldots, e_n\}$, $s0 = d_0$,
- $\alpha(t_i) = d_i$ *and* $\alpha(e_i) = d_i$ *for* $i \in \{0 \ldots n\}$,
- $\beta(t_i) = d_{i+1}$ *for* $i \in \{0 \ldots n\}$ *and* $\beta(t_n) = d_n$, $\beta(e_i) = d_i$ *for* $i \in \{0 \ldots n\}$,
- $\lambda(t_i) = init$ *for* $i \in \{0 \ldots n-1\}$ *and* $\lambda(t_n) = steady$ *and* $\lambda(e_i) = \epsilon$ *for* $i \in \{0 \ldots n\}$.

init denotes a preliminary phase during which the *base* clock must tick alone. *steady* is a phase where both clocks *base* and *delayed* become synchronous for ever. Figure 3 gives as an illustration the resulting transition systems to denote $b \triangleq a \$ 1$ (actions *init* and *steady* are hidden).

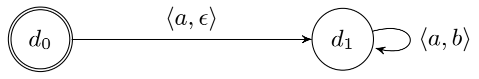

Fig. 3. Binary delay: $b \triangleq a \$ 1$

The binary delay is a particular case of a more general synchronous expression called *FilteredBy* (denoted ▼). $f \triangleq c \blacktriangledown u.(v)^\omega$ defines the clock f as a subclock of c according to two binary words u and v.

Definition 4. *A* binary word *w is a function, $w : \mathbb{N}^\star \to \{0, 1, \perp\}$, such that* $(\exists l \in \mathbb{N}^\star, w(l) = \perp) \implies ((\forall i > l)(w(i) = \perp))$.

Definition 5. *If w is a binary word, $len(w)$ (denoted $|w|$) is called its* length. *$len : (\mathbb{N}^\star \to \{0, 1, \perp\}) \to \mathbb{N} \cup \{\omega\}$. If $\forall i \in \mathbb{N}^\star, w(i) \neq \perp$ then $|w| = \omega$ and w is said to be an* infinite word, *otherwise w is a* finite word. *When w is finite, $|w| = min(i \in \mathbb{N}, w(i+1) = \perp)$.*

Definition 6. *Let n be a positive natural number ($n \in \mathbb{N}^\star$). Let v be a finite binary word. $w = v^n$ is a finite binary word such that $|w| = n * |v|$ and $\forall i \in 1..n, \forall j \in \{1..|v|\}, w(i * j) = v(j)$.*

Definition 7. *Let v be a finite binary word. $w = (v)^\omega$ is an infinite binary word such that $\forall i \in \mathbb{N}, \forall j \in \{1..|v|\}, w(i * |v| + j) = v(j)$.*

Definition 8. *Let u and v be two binary words. $w = u.v$ is a binary word such that $\forall i \in \mathbb{N}^\star, (i \le |u| \implies w(i) = u(i)) \wedge (i > |u| \implies w(i) = v(i - |u|))$. If either u or v is infinite, then w is infinite. If both u and v are finite, then w is finite and such that $|w| = |u| + |v|$.*

Expression 4. *If u and v are two finite binary words, the LTS for the* CCSL *expression* **FilteredBy** *is defined as follows. $f \triangleq c \blacktriangledown u.(v)^\omega$ is the LTS $Filter(u, v) = \langle S, T, s0, \alpha, \beta, \lambda \rangle$ over $A = \{zero, one, \epsilon\}$ with $n + 1$ states such that*

- $S = \{s_1, \dots, s_{|u|+|v|}\}$, $T = \{t_1, \dots, t_{|u|+|v|}, e_1, \dots, e_{|u|+|v|}\}$, $s0 = s_1$,
- $\alpha(t_i) = s_i$ *for* $i \in \{1 \dots |u| + |v|\}$,
- $\beta(t_i) = s_{i+1}$ *for* $i \in \{1 \dots |u| + |v| - 1\}$ *and* $\beta(t_{|u|+|v|}) = s_{|u|+1}$,
- $\lambda(t_i) = zero$ *if* $u(i) = 0$ *and* $\lambda(t_i) = one$ *if* $u(i) = 1$, *for* $i \in \{1 \dots |u|\}$
- $\lambda(t_{i+|u|}) = zero$ *if* $v(i) = 0$ *and* $\lambda(t_{i+|u|}) = one$ *if* $v(i) = 1$, *for* $i \in \{1 \dots |v|\}$
- $\alpha(e_i) = s_i$ *and* $\beta(e_i) = s_i$ *and* $\lambda(e_i) = \epsilon$ *for* $i \in \{1 \dots |u| + |v|\}$.

The label *one* denotes instants where both f and c tick together. The label *zero* when c ticks alone. Actually, Delay is just a particular case of filter with $u = 0^n$ and $v = 1$.

Another interesting special case is when $u = 0^{off}$ and $v = 1.0^{p-1}$, for $off \in \mathbb{N}^\star$ and $p \in \mathbb{N}$. This defines a **periodic pattern** $Periodic(off, p)$, where off is called the *offset* and p the *period*. $Delay(n)$ is also a particular periodic case with an offset of n and a period of 1.

Figure 4 gives an example of a periodic filter, where b is periodic on a with a period of 3 and an offset of 1: $b \triangleq a \blacktriangledown 0.(1.0.0)^\omega$.

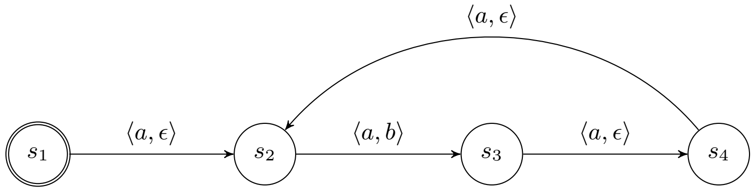
s_1
$\langle a, \epsilon \rangle$
s_2
$\langle a, b \rangle$
s_3
$\langle a, \epsilon \rangle$
s_4
$\langle a, \epsilon \rangle$

Fig. 4. Example of periodic filter with offset: $b \triangleq a \blacktriangledown 0.(1.0.0)^\omega$

SampledOn is an expression that produces a clock s if and only if a clock called *trigger* has ticked between the two previous successive ticks of a sampling clock (*base*).

Expression 5. *sampled $\triangleq$ trigger* ***sampledOn*** *base is the LTS $Sampled = \langle S, T, s0, \alpha, \beta, \lambda \rangle$ over $A = \{base, trig, sample, all\epsilon\}$ with 2 states such that*

- $S = \{s_1, s_2\}$, $T = \{b, bs, sa_1, sa_2, t_1, t_2, e_1, e_2\}$, $s0 = s_1$,

- $\alpha(b) = \beta(b) = s_1$ *and* $\lambda(b) = base$*,*
- $\alpha(sa_i) = \beta(sa_i) = s_i$ *and* $\lambda(sa_i) = all$ *for* $i \in \{1 \dots 2\}$*,*
- $\alpha(t_i) = s_i$ *and* $\beta(t_i) = s_2$ *and* $\lambda(t_i) = trig$ *for* $i \in \{1 \dots 2\}$*,*
- $\alpha(bs) = s_2$ *and* $\beta(bs) = s_1$ *and* $\lambda(bs) = sample$*,*
- $\alpha(e_i) = \beta(e_i) = s_i$ *and* $\lambda(e_i) = \epsilon$ *for* $i \in \{1 \dots 2\}$*.*

Labels *base* and *trig* respectively denote instants where clocks *base* and *trigger* tick alone. Label *sample* denotes instants where both clocks *base* and *sampled* tick simultaneously. Label *all* denotes instants where all the three clocks *base*, *trigger* and *sampled* tick simultaneously. Figure 5 gives the LTS for the sampling operator.

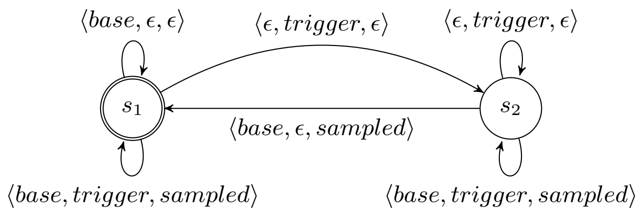
$\langle base, \epsilon, \epsilon \rangle$
$\langle \epsilon, trigger, \epsilon \rangle$
$\langle \epsilon, trigger, \epsilon \rangle$
s_1
s_2
$\langle base, \epsilon, sampled \rangle$
$\langle base, trigger, sampled \rangle$
$\langle base, trigger, sampled \rangle$

Fig. 5. Sampling: *sampled* $\triangleq$ *trigger* **sampledOn** *base*

2.4 Unbounded relations

Unbounded operators can be modeled with labeled transition systems that have an infinite but countable number of states.

Relation 4. CCSL **precedence** *left* $\boxed{\prec}$ *right is modeled as a labeled transition system* $Precedes = \langle S, T, s0, \alpha, \beta, \lambda \rangle$ *over* $A = \{left, right, both, \epsilon\}$ *such that*

- $S = \{p_i | i \in \mathbb{N}\}$*,* $T = \{l_i, r_i, lr_i, e_i | i \in \mathbb{N}\}$*,* $s0 = p_i$*,*
- $\alpha(l_i) = \alpha(e_i) = \alpha(lr_i) = p_i \wedge \alpha(r_i) = p_{i+1}, \forall i \in \mathbb{N}$*,*
- $\beta(l_i) = p_{i+1} \wedge \beta(r_i) = \beta(e_i) = \beta(lr_i) = p_i, \forall i \in \mathbb{N}$*,*
- $\lambda(l_i) = left \wedge \lambda(r_i) = right \wedge \lambda(lr_i) = both \wedge \lambda(e_i) = \epsilon, \forall i \in \mathbb{N}$*.*

Label *left* denotes instants where clock *left* must tick alone. Label *right* denotes instants where clock *right* must tick alone. Label *both* denotes instants where the two clocks must tick simultaneously. Figure 6 shows the transition system for the CCSL relation a $\boxed{\prec}$ b, *i.e.,* the synchronized product of $Clock_a$, $Clock_b$ and *Precedes* with respect to the synchronization constraint $I = \{\langle a, \epsilon, left \rangle, \langle \epsilon, b, right \rangle, \langle a, b, both \rangle, \langle \epsilon, \epsilon, \epsilon \rangle\}$ (*left*, *right* and *both* are hidden for the sake of simplicity).

This operator is called unbounded because the drift between a and b is not bounded, *i.e.,* a can tick infinitely often without b ticking at all. This operator

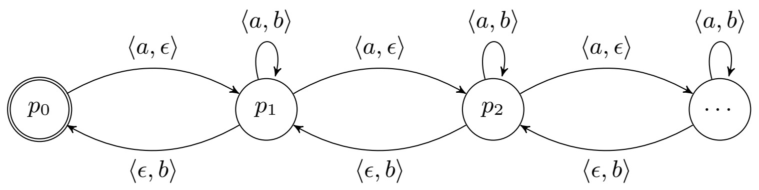
$\langle a, \epsilon \rangle$
$\langle a, b \rangle$
$\langle a, \epsilon \rangle$
$\langle a, b \rangle$
$\langle a, \epsilon \rangle$
$\langle a, b \rangle$
p_0
p_1
p_2
$\cdots$
$\langle \epsilon, b \rangle$
$\langle \epsilon, b \rangle$
$\langle \epsilon, b \rangle$

Fig. 6. CCSL precedence (infinite state LTS): a precedes b.

is not symmetrical. Even though a is unconstrained and can tick whenever it wants and as fast as it wants, b on the contrary is constrained to be always a little late compared to a. So b is said to be slower than a, or a is faster than b.

2.5 Unbounded expressions

In CCSL, there are two unbounded expressions that constrain neither a nor b: **Inf** and **Sup**.

Inf(a, b) is the slowest clock that is faster than both a and b. In most cases, $Inf(a, b)$ is neither a nor b but a clock that sometimes tick simultaneously with a (when a is in advance over b), sometimes it ticks simultaneously with b (when a is late compared to b) and sometimes it ticks simultaneously with a and b (when none of them precedes the other one).

Expression 6. *$Inf(a, b)$ is the labeled transition system $Inf = \langle S, T, s0, \alpha, \beta, \lambda \rangle$ over $A = \{left, right, both, left\_inf, right\_inf, \epsilon\}$ such that*

- $S = \{s_i | i \in \mathbb{Z}\}$, $T = \{inc_i, dec_i, t_i, e_i | i \in \mathbb{Z}\}$, $s0 = s_0$,
- $\alpha(inc_i) = \alpha(dec_i) = \alpha(both_i) = \alpha(e_i) = s_i$, $\forall i \in \mathbb{Z}$,
- $\beta(both_i) = \beta(e_i) = s_i$ *and* $\beta(inc_i) = s_{i+1}$ *and* $\beta(dec_i) = s_{i-1}$, $\forall i \in \mathbb{Z}$,
- $\lambda(inc_i) = left\_inf$ *if* $i \geq 0$, *and* $\lambda(inc_i) = left$ *if* $i < 0$, $\forall i \in \mathbb{Z}$
- $\lambda(dec_i) = right\_inf$ *if* $i \leq 0$, *and* $\lambda(dec_i) = right$ *if* $i < 0$, $\forall i \in \mathbb{Z}$
- $\lambda(both_i) = both$ *and* $\lambda(e_i) = \epsilon$, $\forall i \in \mathbb{Z}$

Figure 7 shows the transition systems for $i \triangleq Inf(a, b)$. This LTS is infinite on both sides. Let us note than by definition $Inf(a, b) \preccurlyeq a$ and $Inf(a, b) \preccurlyeq b$, which means that if $Inf(a, b)$ is somehow constrained (*i.e.,* by a synchronous operator like filter), then this propagates the constraint on both a and b. Additionally, whenever a clock c is known to be faster than either a or b, then $c \preccurlyeq Inf(a, b)$, *i.e.,* the tickings of $Inf(a, b)$ are constrained (and bounded) by all the clocks faster than either a or b.

Sup(a, b) is defined as the fastest clock that is slower than both a and b. In most cases, $Sup(a, b)$ is neither a nor b.

Expression 7. *$Sup(a, b)$ is a labeled transition system $Sup = \langle S, T, s0, \alpha, \beta, \lambda \rangle$ over $A = \{left, right, both, left\_sup, right\_sup, both\_sup, \epsilon\}$ such that*

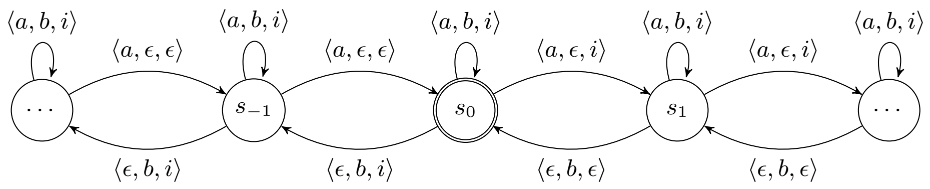
$\langle a,b,i\rangle$
$\langle a,\epsilon,\epsilon\rangle$
$\langle a,b,i\rangle$
$\langle a,\epsilon,\epsilon\rangle$
$\langle a,b,i\rangle$
$\langle a,\epsilon,i\rangle$
$\langle a,b,i\rangle$
$\langle a,\epsilon,i\rangle$
$\langle a,b,i\rangle$
$\cdots$
s_{-1}
s_0
s_1
$\cdots$
$\langle \epsilon,b,i\rangle$
$\langle \epsilon,b,i\rangle$
$\langle \epsilon,b,\epsilon\rangle$
$\langle \epsilon,b,\epsilon\rangle$

Fig. 7. CCSL Inf (infinite state LTS): $i \triangleq \mathit{Inf}(a,b)$.

- $S = \{s_i | i \in \mathbb{Z}\}$, $T = \{inc_i, dec_i, t_i, e_i | i \in \mathbb{Z}\}$, $s0 = s_0$,
- $\alpha(inc_i) = \alpha(dec_i) = \alpha(both_i) = \alpha(e_i) = s_i$, $\forall i \in \mathbb{Z}$,
- $\beta(both_i) = \beta(e_i) = s_i$ *and* $\beta(inc_i) = s_{i+1}$ *and* $\beta(dec_i) = s_{i-1}$, $\forall i \in \mathbb{Z}$,
- $\lambda(inc_i) = left$ *if* $i \geq 0$, *and* $\lambda(inc_i) = left\_sup$ *if* $i < 0$, $\forall i \in \mathbb{Z}$
- $\lambda(dec_i) = right$ *if* $i \leq 0$, *and* $\lambda(dec_i) = right\_sup$ *if* $i < 0$, $\forall i \in \mathbb{Z}$
- $\lambda(both_i) = both$ *if* $i \neq 0$ *and* $\lambda(e_i) = \epsilon$, $\forall i \in \mathbb{Z}$, *and* $\lambda(both_0) = both\_sup$

Figure 8 shows the transition systems for $s \triangleq \mathit{Sup}(a,b)$. Let us note than by definition $a \preccurlyeq \mathit{Sup}(a,b)$ and $b \preccurlyeq \mathit{Sup}(a,b)$, which means that the constraints imposed on $\mathit{Sup}(a,b)$ do not directly impact neither a or b. However, whenever a clock c is known to be slower than either a or b, then it is also slower than $\mathit{Sup}(a,b)$, *i.e.,* ($\exists c$ such that $a \preccurlyeq c \vee b \preccurlyeq c) \implies \mathit{Sup}(a,b) \preccurlyeq c$.

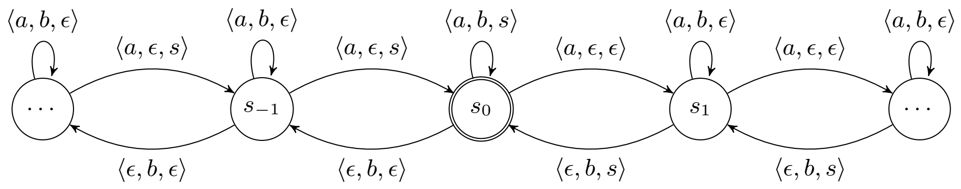
$\langle a,b,\epsilon\rangle$
$\langle a,\epsilon,s\rangle$
$\langle a,b,\epsilon\rangle$
$\langle a,\epsilon,s\rangle$
$\langle a,b,s\rangle$
$\langle a,\epsilon,\epsilon\rangle$
$\langle a,b,\epsilon\rangle$
$\langle a,\epsilon,\epsilon\rangle$
$\langle a,b,\epsilon\rangle$
$\cdots$
s_{-1}
s_0
s_1
$\cdots$
$\langle \epsilon,b,\epsilon\rangle$
$\langle \epsilon,b,\epsilon\rangle$
$\langle \epsilon,b,s\rangle$
$\langle \epsilon,b,s\rangle$

Fig. 8. CCSL sup (infinite state LTS): $s \triangleq \mathit{Sup}(a,b)$.

3 Boundness issues on CCSL specifications

When several CCSL constraints are put in parallel, the composition is simply defined as the synchronized product of the LTSs of the operators. However, since some of the LTSs for the primitive operators are infinite (*e.g.,* Fig. 6, Fig. 7, Fig. 8), the synchronized product might end up being infinite. However, even though the product is potentially infinite, in some cases, only a finite subset of the synchronized product is reachable from the initial state. Section 3.1 shows a case where the product of infinite LTSs is finite. The algorithm used in that subsection only terminates when the product is actually finite. The following section discusses a sufficient condition to decide whether the product is actually finite and therefore whether the algorithm proposed in Section 3.1 actually terminates.

3.1 Finite synchronized product of infinite LTSs

.

Considering n LTSs such that, for $i = 1, \ldots, n$, $\mathcal{A}_i = \langle S_i, T_i, s0_i, \alpha_i, \beta_i, \lambda_i \rangle$ and one synchronization constraint $I \subseteq A_1 \times \ldots \times A_n$, the *synchronized product* of $\mathcal{A}_i$ with respect to I is a labeled transition system $\langle S, T, s0, \alpha, \beta, \lambda \rangle$ over the set I constructed as described in Algorithm 1. This algorithm terminates only when the product has a finite number of states. Indeed, S is a set initialized with only one state. At each iteration, one state st is removed from S and added to S'. All the outgoing transitions of st are computed. If C is the set of clocks, there are at most $2^{|C|}$ outgoing transitions. Some of these transitions may be inconsistent. For each transition the target state st' is computed and added to S if not already present in S'. This condition guarantees that the same state is not visited twice. The algorithm terminates when S is empty. S becomes empty when all the targeted state are already in S' (have already been visited). If the set of reachable states is finite then when all the states are in S' then S is necessarily empty. Therefore, when the set of reachable states is finite the algorithm terminates.

Algorithm 1. Synchronized product through reachability analysis

```
Let $S' \leftarrow \emptyset$
Let $s0 \leftarrow s0_1 \times \ldots \times s0_n$
Let $S \leftarrow \{s0\}$
while $S$ is not empty {
    Let $st = st_1 \times \ldots \times st_n$ be one element of $S$
    Let $S' \leftarrow S' \cup \{st\}$
    Let $S \leftarrow S \setminus \{st\}$
    $\forall \langle t_1, \ldots, t_n \rangle \in I$ such that $(\forall i \in \{1 \ldots n\})(\alpha_i(t_i) = st_i)\}$ {
        Let $st' = \beta_1(t_1) \times \ldots \times \beta_n(t_n)$
        if $st' \notin S'$ then $S \leftarrow S \cup \{st'\}$
    }
}
```

Let us take as an example the following CCSL specification: $(a \boxed{\prec} c) \wedge (b \triangleq a \$ 1) \wedge (c \boxed{\prec} b)$. This specification is defined as the synchronized product of *Precedes* (Relation 4), *Delay*(1) (Expression 3), *Precedes* (Relation 4 again).

Initially, $s0 = p_0 \times d_0 \times p_0$. The first precedes relation (state p_0) imposes c not to tick, the second precedes (state p_0) prevents b from ticking whereas the delay expression (state d_0) only allows a to tick alone without c. Therefore the only outgoing transition consists in making a ticks alone going into the state $s1 = p_1 \times d_1 \times p_0$. At this stage $S = \{s1\}$ and $S' = \{s0\}$. From $s1$, the first precedes relation (state p_1) does not impose any constraint while the second one (state p_0) still prevents b from ticking. The delay expression (state $d1$) only allows doing nothing or making a and b tick simultaneously. Since b cannot tick, then a cannot tick either, so only c can tick leading to state $s2 = p_0 \times d_1 \times p_1$. Therefore $S' = \{s0, s1\}$ and $S = \{s2\}$. From $s2$, the first precedes relation prevents c from ticking, the second relation also prevents c from ticking. The delay expression only allows a and b to tick simultaneously. Taking this (sole) solution leads to $s1$. Since $s1$ is already in S', no new state is added to S, which is therefore empty. This terminates the algorithm. The resulting LTS has only three states (Fig. 9).

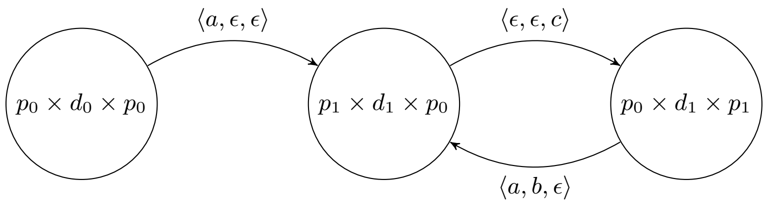
$\langle a, \epsilon, \epsilon \rangle$
$\langle \epsilon, \epsilon, c \rangle$
$p_0 \times d_0 \times p_0$
$p_1 \times d_1 \times p_0$
$p_0 \times d_1 \times p_1$
$\langle a, b, \epsilon \rangle$

Fig. 9. CCSL alternation: synchronized product of two precedences and one delay

This particular construction is very frequent, it has been called **Alternation** and is denoted $a \boxed{\sim} c$. Increasing the delay from 1 to n makes a particular relation, called **bounded precedence** and denoted as a $\boxed{\prec_n}$ c. $a \boxed{\sim} c \equiv a \boxed{\prec_1} c$. Previous works on CCSL where always assuming a bound for all CCSL operators, whereas here the bound is computed by reachability analysis. However, the (semi) algorithm sketched above may not terminate when the synchronized product is not finite. Therefore it is necessary to know beforehand whether the result is finite or not.

3.2 A sufficient condition for having a bounded CCSL specification

We have seen in the previous subsection that knowing in advance whether the synchronized product is finite is important. Indeed, when it is not finite, then Algorithm 1 does not terminate. This section discusses ways to determine whether the system is finite or not. The problem is similar to safety issues in process networks [9]. In process networks, a channel is *k-safe* it the channel can contain at most k tokens. A process network is k-safe if all its channels are k-safe. However, testing that a process network is k-safe is undecidable in the general case. However, the problem becomes decidable if we restrict to a special kind of process networks, *i.e.,* the marked graphs [10] or their extension, the synchronous data flow (SDF) graphs [11].

So the idea here is to transform the CCSL specification into a SDF graph. Since SDF do not have any notion of simultaneous action, the full semantics of

CCSL cannot be captured as an SDF graph but it can still capture an abstraction of relative rates at which the clocks execute. Then, if the resulting SDF graph is safe, the corresponding CCSL specification would be bounded. If the SDF graph is not safe, then it does not say much of the CCSL specification since we consider only an approximation of the CCSL semantics. Therefore, the safety of the underlying SDF graph would only be a sufficient condition for a CCSL specification to be bounded.

CCSL clocks define triggering conditions just like SDF computation nodes. CCSL constraints impose conditions that determine the relative rates at which each clock can tick. Similarly, SDF edges also determine some dependencies and evolution rates between the SDF actions.

For instance, the precedence relation of CCSL simply states that one clock cannot tick faster than the other one. It can be captured through the SDF graph shown in Figure 10. The boxes are computation nodes. The circle is an infinite channel working as a FIFO with non-blocking writes and blocking reads. Every time the node a executes, it produces a token (data) that enters the communication channel. The node b can only execute if tokens are available in the channel. When b executes it consumes one token from the (input) channel. Therefore, the node b can compute its i^{th} execution only after a has computed its i^{th} execution. This is the exact same semantics that $a \prec b$ except that there is no temporal notion associated with SDF graphs, and no notion of simultaneity, only of data dependency or causality.

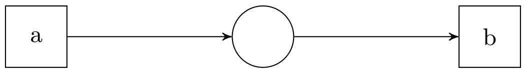

Fig. 10. CCSL precedence as an Synchronous Data Flow graph

Even though SDF does not have any notion of temporality, it can still provide a support for abstracting the synchronous CCSL operators. Let us consider the delay operator used in Figure 3 as an example. This operator says that ultimately the two clocks a and c execute at the same rate (synchronously). Since the delay is of one unit, it also means that a is always one tick ahead of c. In other words, the difference of ticks between a and c is exactly one, not more, not less. This can be expressed as the SDF graph shown in Figure 11. The one in the bottom channel means that there is one initial token here, so that a can execute right away. c cannot execute since there is no token in its incoming channel. When a executes, it consumes a token in the bottom channel and produces one in the top channel allowing c to execute. As long as c does not execute then a cannot execute again. In marked graphs and in SDF graphs, the number of tokens over a cycle is a constant. Having one initial token in this cycle guarantees that a and c execute at the same rate in alternation.

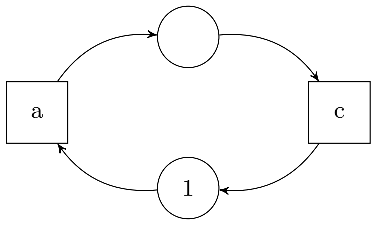

Fig. 11. CCSL delay as an Synchronous Data Flow graph

Now, if we take the same example as in the previous subsection, we can build an SDF graph for the CCSL specification: $(a \boxed{\prec} c) \wedge (b \triangleq a \,\$\, 1) \wedge (c \boxed{\prec} b)$. Each operator brings its own data/rate dependencies. The parallel composition of the operators consists in putting together all these dependencies. The resulting SDF graph is shown in Figure 12.

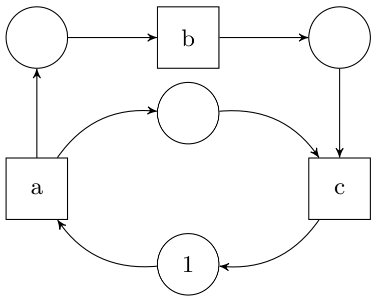

Fig. 12. CCSL delay as an Synchronous Data Flow graph

Classical results on data flow process networks show that this graph is 1-safe since no computation node is a source[3] or a sink[4] and all the two cycles (a-b-c and a-c) have exactly one initial token. This means that all the three clocks a, b and c must execute at the same speed and therefore that the corresponding CCSL specification has a finite number of states. The state being the differences of ticks between the different clocks.

[3] a source is a computation node without input edge
[4] a sink is a computation node without output edge

4 Example: CCSL for capturing the architecture, application and allocation

To illustrate the approach, we take an example inspired by [12], that was used for flow latency analysis on AADL[5] specifications [13]. However, with CCSL we are conducting different kinds of analyses, section 5 discusses some common points with classical real-time scheduling analysis.

4.1 Application

Figure 13 (on the top) considers a simple application described as a UML structured class. This application captures two inputs *in*1 and *in*2, performs some calculations (*step*1, *step*2 and *step*3) and then produces a result *out*. This application has the possibility to compute *step*1 and *step*2 concurrently depending on the chosen execution platform. This application runs in a streaming-like fashion by continuously capturing new inputs and producing outputs.

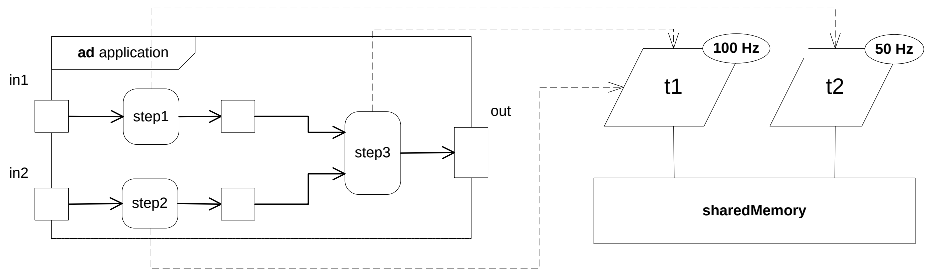

Fig. 13. Simple application

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.*, *step*1). 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:

$$in1 \boxed{\preccurlyeq} step1 \wedge step1 \boxed{\prec} step3 \tag{1}$$

$$in2 \boxed{\preccurlyeq} step2 \wedge step2 \boxed{\prec} step3 \tag{2}$$

$$step3 \boxed{\preccurlyeq} out \tag{3}$$

Eq. 1 specifies that *step*1 may begin as soon as an input *in*1 is available. Executing *step*3 also requires *step*1 to have produced its output. Eq. 2 is similar

[5] AADL stands for Architecture & Analysis Description Language

for $in2$ and $step2$. Finally, Eq. 3 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 unbounded, therefore TimeSquare cannot perform any kind of exhaustive analysis and can only produce a particular schedule that matches the specification (see Fig. 14).

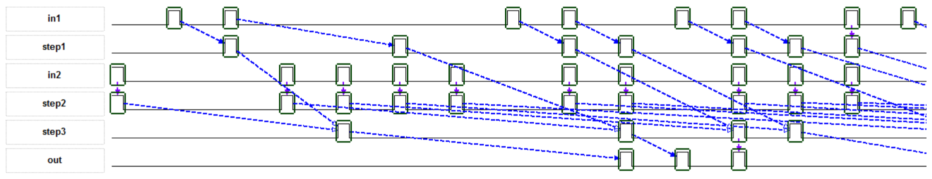

Fig. 14. A valid schedule for the application part of Fig. 13

One way to reduce the state-space is to bound the drift between the inputs and the outputs. This means limiting the parallelism by slowing down the production of outputs when several computations are still on-going. This can easily be done by adding a CCSL constraint like Eq. 4.

$$Sup(in1, in2) \boxed{\sim} out \tag{4}$$

The effect of this constraint can be seen on Figure 15. Looking carefully at this schedule, we can note that the arrival of $in2$ has been slown down to avoid large accumulation of computations. For instance, the third occurrence of $in2$ is delayed after the second occurrence of out. However, we can see that the input $in1$ keeps arriving at a fast rate allowing executions of $step1$. However, the execution of $step3$ is stalled after the corresponding occurrence of $in2$ has been dealt with by $step2$ as required by Eq. 2.

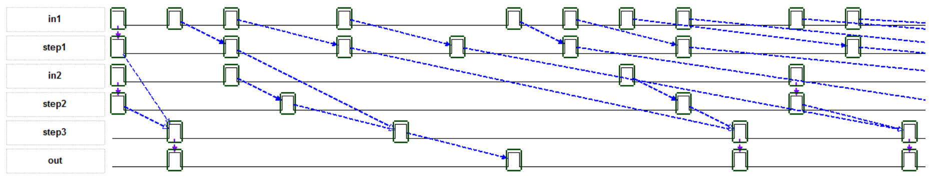

Fig. 15. Another valid schedule for the application part of Fig. 13

Reachability analysis as described in Section 3 tells us that the composition is still not bounded because bounds on $Sup(in1, in2)$ do not imply bounds on both $in1$ and $in2$. To have a complete finite systems, we can for instance replace Eq. 4 by Eq. 5.

$$Inf(in1, in2) \boxed{\sim} out \tag{5}$$

By doing so, our reachability analysis algorithm converges and produces the state-space shown in Figure 16[6]. We have removed *in*1, *in*2, and *out* since they were just adding interleaving without offering more actual parallelism in the execution of actions.

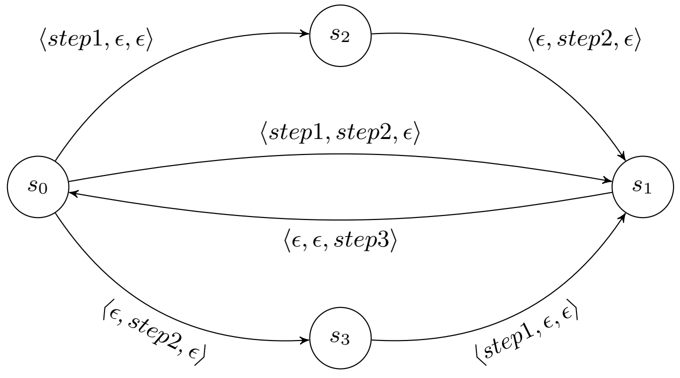
s_0
s_1
s_2
s_3
$\langle step1, \epsilon, \epsilon\rangle$
$\langle \epsilon, step2, \epsilon\rangle$
$\langle step1, step2, \epsilon\rangle$
$\langle \epsilon, \epsilon, step3\rangle$
$\langle \epsilon, step2, \epsilon\rangle$
$\langle step1, \epsilon, \epsilon\rangle$

Fig. 16. Synchronous products of Eqs. 1-3 and Eq. 5.

This kind of analysis is useful to detect invalid CCSL specifications. For instance, had we replaced Eq. 4 by Eq. 6 instead of Eq. 5, we would have obtained a finite result but with the state-space shown in Figure 17. This figure shows a typical case of deadlock in CCSL. Indeed, if from the initial state s_0, we decide to fire *in*1 (resp. *in*2) alone, then Eq. 6 prevents $in1 + in2$ from ticking again before *out* ticks, but since *in*2 (resp. *in*1) was not produced and therefore *step*2 was not executed. Then *step*3 cannot execute either since it requires both *step*1 and *step*2. If *step*3 cannot execute, then *out* cannot be produced, which then results in a deadlock.

$$in1 + in2 \boxed{\sim} out \tag{6}$$

4.2 Execution platform and allocation

Once the application is designed, then CCSL can also be used to capture the execution platform. Figure 13 (bottom part) shows the selected execution platform: two tasks with different activation periods. The basic CCSL specification of the execution platform is given as follows:

$$t1 \triangleq ms \blacktriangledown (1.0^9)^\omega \tag{7}$$

$$t2 \triangleq t1 \blacktriangledown (1.0)^\omega \tag{8}$$

[6] The algorithm is available as an Eclipse update site on http://timesquare.inria.fr/sts/update_site/

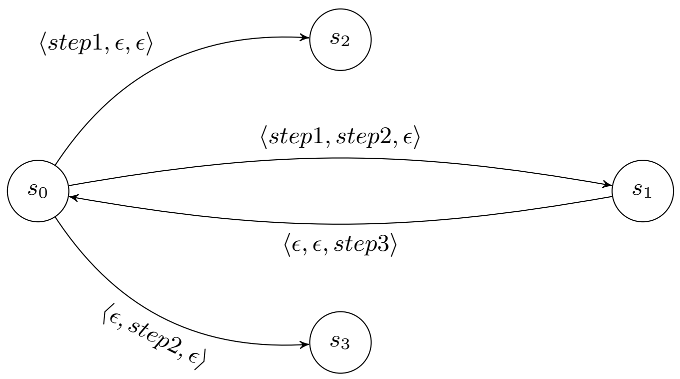
$\langle step1, \epsilon, \epsilon \rangle$
s_2
$\langle step1, step2, \epsilon \rangle$
s_0
s_1
$\langle \epsilon, \epsilon, step3 \rangle$
$\langle \epsilon, step2, \epsilon \rangle$
s_3

Fig. 17. Synchronous products of Eqs. 1-3 and Eq. 6.

Eq. 8 is a pure logical relationship between $t1$ and $t2$ that states that thread $t2$ is twice slower than thread $t1$, *i.e.,* it is periodic on $t1$ with period 2 and offset 0. Eq. 7 is also a periodic relation, but relative to ms, a particular clock that denotes milliseconds. Being periodic on ms with a period of 10 makes $t1$ a 100 Hz clock and therefore $t2$ a 50 Hz clock.

When the execution platform is specified, the remaining task is to map the application onto the execution platform. In MARTE, this is done through an allocation. In CCSL, this is done by refining the two specifications with new constraints that specify this allocation. Since both $step2$ and $step3$ are allocated on the same thread, then their execution is exclusive (Eq. 9). Then, the thread being periodic, the inputs are sampled according to the period of activation of the threads (Eqs. 10-11). Then $step3$ needs inputs from both $step1$ and $step2$ before executing but it can execute only according to the sampling period of $t1$ since $step3$ is allocated to $t1$ (Eq. 12). Finally, all steps can only execute when their input data have been sampled (Eq. 13).

$$step2 \boxed{\#} step3 \tag{9}$$

$$in1\_s \triangleq in1 \textbf{ sampledOn } t1 \tag{10}$$

$$in2\_s \triangleq in2 \textbf{ sampledOn } t2 \tag{11}$$

$$d3\_s \triangleq \textbf{Inf}(step1, step2) \textbf{ sampledOn } t1 \tag{12}$$

$$in1\_s \boxed{\preccurlyeq} step1 \wedge in2\_s \boxed{\preccurlyeq} step2 \wedge d3\_s \boxed{\preccurlyeq} step3 \tag{13}$$

All these new constraints do not change anything on the finiteness of the whole system. They only reduce the set of possible executions. If the application specification was finite, then its allocated version is still finite. If it was infinite, they it remains infinite. Whether it is finite or not, TimeSquare can produce an execution of this specification (see Fig. 18). On this schedule the dashed arrows denote precedence relations, while the (red) vertical lines denote coincidence relations. Note that the fact that *ms* is a physical clock does not impact the calculus, it only impacts the visual representation of the schedule.

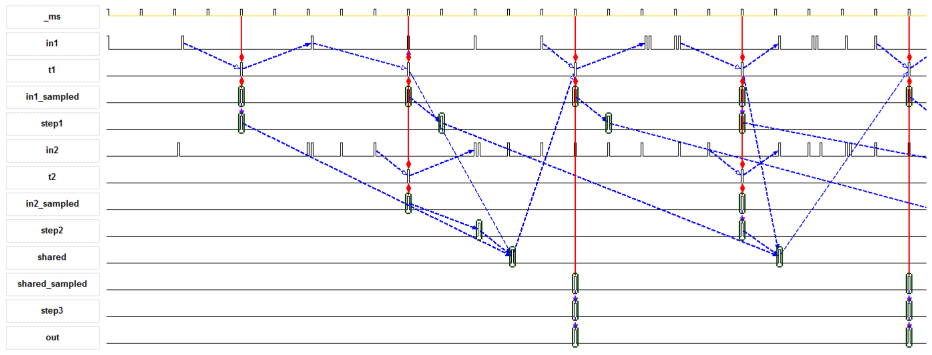

Fig. 18. A valid schedule for the allocated application (Fig. 13)

5 Related work

The transformation of CCSL into labeled transition systems has already been attempted in [14,15]. However, in those attempts, the CCSL operators were bounded because the underlying model-checkers cannot deal with infinite labeled transition systems. The purpose of this work is to deal with unbounded operators.

In [16], there was an initial attempt to provide a data structure suitable to capture infinite transition systems based on a lazy evaluation technique. A similar structure could be used in our case except that we consider clocks with only two states (instead of three): tick or stall. Clock death is still to be further explored.

The kind of applications addressed in section 4 is very close to models usually used in real-time scheduling theories. However, such theories usually rely on task models that abstract real applications. Originally they were rather simple (e.g., independent periodic tasks only for Rate Monotonic Analysis). Always more sophisticated models now appear in the literature. They are all based on numerous distinct parameters, providing numerical constraint values for timing aspects (dispatch time, period, deadline, jitter drift. . .). Tasks are considered as iterations of jobs (or jobs as instances of tasks). In our view, the successive timing values for characteristic feature of successive jobs can each be seen as a logical clock, and the time constraint relations between such clocks are usually expressed as simple equalities and bounded inequalities that fall well into the range of CCSL constructs descriptive power.

Classical (non real-time) scheduling, on its side, provides generally models where the initial constraints are less on timing and more on dependencies or on exclusive resource allocation. But resulting schedules are almost always of *modulo* periodic nature, here again matching the CCSL expressiveness.

Usually, authors [17,18,19] rely on "physical-by-nature" timing, found in theoretical models such as Timed Automata [20]. The distinctive difference is that timed automata assume a global physical time. Timed events are then constrained by value relations between so-called clocks (a different notion from our logical clocks), which are devices measuring physical time as it elapses.

Our work also bears some similarity with previous attempts by Alur and Weiss [21,22], which define schedules as infinite words expressed in regular expressions and then construct corresponding Büchi automata.

6 Conclusion

We have presented a state-based semantics of a kernel subset of CCSL, a language that relies on logical clocks to express logical and temporal constraints. Each CCSL operator (relation or expression) is defined as a label transition system, that may have either a finite or infinite number of states. The parallel composition of CCSL constraints is defined as the synchronized product of the primitive label transition systems. A (semi)algorithm is proposed to actually build the synchronized product of infinite transition systems by assuming that only a finite number of states are accessible in the product. The algorithm only terminates on that condition. Then a discussion is made on how data flow process networks could be used as a sufficient condition to decide that the synchronized product is actually finite. All the approach is illustrated on a simple example often used in AADL and where a simple application is allocated onto a two processor architecture. The work presented here improves on previous attempts to support exhaustive analyses of CCSL specifications. Indeed, previous works were only considering *a priori* bounded CCSL operators to guarantee the finiteness of the composition, while here no assumption is made on the boundness of primitive operators.

As a future work, we should extend and prove that data flow process networks can actually be used to detect finite compositions of any unbounded CCSL operators. Whereas it is pretty much clear that synchronous operators and regular asynchronous operators (like precedes, inf, sup) are always covered by synchronous data flow graphs, it is much less clear for mix operators like sampledOn. This aspect has only been briefly touched here to underline the fact that on simple examples our algorithm is actually useful.

References

1. OMG: UML Profile for MARTE, v1.0. Object Management Group. (November 2009) formal/2009-11-02.
2. André, C., Mallet, F., de Simone, R.: Modeling time(s). In: 10th Int. Conf. on Model Driven Engineering Languages and Systems (MODELS '07). Number 4735 in LNCS, Nashville, TN, USA, ACM-IEEE, Springer (September 2007) 559–573

3. André, C.: Syntax and semantics of the Clock Constraint Specification Language (CCSL). Research Report 6925, INRIA (May 2009)
4. Deantoni, J., Mallet, F.: Timesquare: Treat your models with logical time. In: TOOLS (50). Volume 7304 of LNCS., Springer (2012) 34–41
5. Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Le Guernic, P., de Simone, R.: The synchronous languages 12 years later. Proc. of the IEEE **91**(1) (2003) 64–83
6. Le Guernic, P., Talpin, J.P., Le Lann, J.C.: Polychrony for system design. Journal of Circuits, Systems, and Computers **12**(3) (2003) 261–304
7. Lee, E.A., Sangiovanni-Vincentelli, A.L.: A framework for comparing models of computation. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **17**(12) (December 1998) 1217–1229
8. Arnold, A.: Finite transition systems - semantics of communicating systems. Int. Series in Computer Science. Prentice Hall (1994)
9. Kahn, G.: The semantics of simple language for parallel programming. In: IFIP Congress. (1974) 471–475
10. Commoner, F., Holt, A.W., Even, S., Pnueli, A.: Marked directed graphs. J. Comput. Syst. Sci. **5**(5) (1971) 511–523
11. Lee, E., Messerschmitt, D.: Synchronous data flow. Proceedings of the IEEE **75**(9) (1987) 1235–1245
12. Feiler, P.H., Hansson, J.: Flow latency analysis with the architecture analysis and design language. Technical Report CMU/SEI-2007-TN-010, CMU (June 2007)
13. of Automotive Engineers, S.: SAE Architecture Analysis and Design Language (AADL). (June 2006) document number: AS5506/1.
14. Yin, L., Mallet, F., Liu, J.: Verification of MARTE/CCSL time requirements in Promela/SPIN. In Perseil, I., Breitman, K., Sterritt, R., eds.: ICECCS, IEEE Computer Society (2011) 65–74
15. Gascon, R., Mallet, F., DeAntoni, J.: Logical time and temporal logics: Comparing UML MARTE/CCSL and PSL. In Combi, C., Leucker, M., Wolter, F., eds.: TIME, IEEE (2011) 141–148
16. Romenska, Y., Mallet, F.: Lazy parallel synchronous composition of infinite transition systems. In: ICTERI. Volume 1000 of CEUR Workshop Proc. (2013) 130–145
17. Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: Times: A tool for schedulability analysis and code generation of real-time systems. In: Formal Modeling and Analysis of Timed Systems. Volume 2791 of LNCS. Springer (2004) 60–72
18. Krcál, P., Yi, W.: Decidable and undecidable problems in schedulability analysis using timed automata. In: Tools and Algorithms for the Construction and Analysis of Systems. Volume 2988 of LNCS. Springer (2004) 236–250
19. Abdedda Y.: Scheduling with timed automata. Theoretical Computer Science **354**(2) (2006) 272–300
20. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2) (1994) 183–235
21. Alur, R., Weiss, G.: Regular specifications of resource requirements for embedded control software. In: Proc. of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium. RTAS '08, Washington DC, USA, IEEE Computer Society (2008) 159–168
22. Alur, R., Weiss, G.: Rtcomposer:a framework for real-time components with scheduling interfaces. In: Proc. of the 8th ACM Int. Conf. on Embedded software. EMSOFT '08, New York, NY, USA, ACM (2008) 159–168