Correctness Issues on MARTE/CCSL constraints

Frédéric Mallet, Robert De Simone

Correctness issues on MARTE/CCSL constraints

Frédéric Mallet[a,d,c,*], Robert de Simone[b,d]

[a] *Université Nice Sophia Antipolis*
[b] *INRIA Sophia Antipolis Méditerranée*
[c] *East China Normal University, Software Engineering Institute*
[d] *I3S Laboratory, UMR 7271 CNRS*

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 available verification techniques. 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 analysis on MARTE models.

This work relies on a state-based semantics for CCSL to establish correctness properties on MARTE/CCSL specifications. We propose and compare two different techniques to build the state-space of a specification. One is an extension of some previous work and is based on extended finite state machines. It relies on integer linear programming to solve the constraints and reduce the state-space. The other one, is based on an intentional representation and uses pure boolean abstractions but offers no guarantee to terminate when the specification is not safe.

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 execution platform through allocation.

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

*Corresponding author: Frederic.Mallet@unice.fr

1. Introduction

The Unified Modeling Language (UML 2.x) proposes a simplistic and informal model of time, called *Simple Time*. This model has been extended in the UML Profile for Modeling and Analysis of Real-Time and Embedded systems [1] (MARTE), adopted in November 2009. MARTE introduces a richer *Time model* [2] 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 *Clock Constraint Specification Language* (CCSL) has been initially defined in an annex of the MARTE specification to provide a concrete syntax for handling these logical clocks as first-class citizens. It was endowed with a formal operational semantics [3] to breathe life into UML models by defining synchronization and coordination schemes between the various modeling elements.

The operational semantics of CCSL is adequate to build a simulation framework, like TimeSquare [4][1] but less appropriate to conduct exhaustive analyzes. We rely for that purpose on a state-based semantics to establish correctness properties on CCSL specifications. A CCSL specification is called *safe* if and only if it can be represented with a finite state machine. Some of the CCSL constraints are not safe and their semantics can only be captured with an infinite number of states or a finite symbolic representation of these infinite states. In a previous work [5], we have proposed to use extended state machines, *i.e.,* finite state machines extended with (unbounded) integer variables, to capture the semantics of unsafe constraints. This abstraction lead to the generation of observers for simulation or model-checking verification. This abstraction is very convenient since it does not need to assume that the specification is actually safe. In [6] we have proposed an algorithm to detect safe specifications. Having that we propose an alternative solution that does not rely on extended finite state machines but rather on a *intentional* data structure. In this paper, we propose an extension of [5] with state invariants to further reduce the size of the product. Then, we describe the alternative solution. Both solutions have advantages and flaws that are explored in details.

Building the synchronized product of a CCSL specification is key to conduct model-checking on properties. Indeed, we also discuss some classical

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

liveness issues that may arise with CCSL specifications and that can actually be checked with our proposal. Safety and liveness issues are illustrated on a simple example borrowed from AADL and in which the platform, the architecture and the binding are captured in MARTE/CCSL.

Section 2 starts with a positioning with respect to related works. Section 3 gives some background about CCSL and transition systems for clock systems. Section 4 is the main part of the contribution. It discusses first the solution relying on extended finite state machines and integer linear programming. Second, it compares it to another solution relying on purely Boolean analysis and using an intentional data structure. Section 5 gives an illustrative example and discusses typical correctness properties.

2. Related work

Logical clocks have been introduced in distributed systems [7] to loosely synchronize communicating systems and order their events. This appealing concept of logical clock is central in many contexts and for several purposes, including in process networks or in Petri nets [8], however its usage in CCSL is mainly inspired from their central role as activation conditions in Synchronous languages [9, 10]. Compared to Lamport's clocks, synchronous languages introduce the notion of atomic reaction (also called *instant*) in which several events occur simultaneously. Consequently, the behavior of a system can be defined based on what has happened during the reaction, but also on what has NOT happened, hence leading to the so-called reaction to absence. CCSL operators forbid the occurrence of some events based on what has happened or not in previous reactions.

Whereas usually in synchronous languages, the programmer handles signals (sequences of values) as a primitive construct, the notion of signals does not exist in CCSL and the language only focuses on the clocks themselves. In synchronous languages, clocks tend to be handled mainly by the compiler, through the process called *clock calculus*, to decide when the values are valid and when the computations must be performed. In Lustre [11], clocks of inputs are usually given indirectly while in Esterel [9] they are rarely handled explicitly by the programmer. In polychronous extensions, like Signal [12], the clocks constrain the system to become endochronous, when the system is underspecified. CCSL was devised as a language focusing on clocks independently of signals and values.

Later, the notion of tag system [13] and then tag structure [14] was proposed as a mathematical framework to compare models of computations and orchestrate heterogeneous models. CCSL provides a concrete syntax [15] for building such orchestration models by focusing only on clocks (or tags) and not on values.

Initially the operational semantics of CCSL was given as a set of rewriting rules [3] in order to build a simulation engine that performs the clock calculus dynamically on the fly. To conduct exhaustive analyzes on CCSL specifications we propose to encode the semantics using transition systems. Some CCSL operators cannot be represented with finite transition systems and symbolic representations must be proposed to deal with these so-called *unsafe* operators. In [5], finite state machines extended with unbounded integer variables where proposed for that purpose. Integer variables symbolically captured the infinite number of states. We consider in this paper an alternative encoding that maintains an intentional representation of the infinite transition systems and that expands them on-demand when the synchronized product is built. The contribution of this paper is to compare the two alternative approaches and to discuss solutions to establish correctness properties on CCSL specifications.

Clearly, the proposed structure comes close to pushdown automata. The literature is abundant on pushdown automata (PDA). The class needed here is strictly weaker than PDA since the operations on the stack are very limited. Indeed, the stack would just be used for counting (+1, -1, zero-test). Counter automata [16, 17] appears to be a subclass closer to our needs. The reachability problem for counter-automata has been studied a lot and subclasses for which reachability is decidable have been identified [18, 19]. A naive encoding of CCSL in counter automata would probably result in having one counter per clock or at least one counter per clock domain. Such an encoding is out of the scope of this paper. Relying on acceleration techniques [20] to compute the reachability set from a CCSL specification is clearly an interesting problem that is also beyond the goals of this contribution. This paper does not propose a new mathematical abstraction but rather explores two practical solutions to conduct exhaustive verifications on CCSL specifications.

Several attempts have been made before to perform exhaustive analyses of CCSL specifications. Gaston et al. [21] proposed an encoding as Büchi automata to compare the expressiveness of CCSL with temporal logics. Yin et al. [22] proposed to encode CCSL operators in Promela to perform model-checking with the SPIN model-checker. In both attempts, only a safe subset

of CCSL was considered. In this paper, the full *unsafe* semantics of CCSL operators is captured. When the composition of unsafe operators is safe, the state space can be built and model-checked.

Yu et al. [23] have proposed to encode some CCSL operators in Signal. The purpose was to rely on the signal compiler to perform clock hierarchization and synthesize the controller. Only a subset of CCSL was addressed. Moreover, Signal relies on a three-valued logics to deal both with clocks and values. Since CCSL does not consider values, a more direct encoding, as proposed here, is likely to give better results. Indeed, going from CCSL to Signal, then to the three-valued logics for representing Boolean operators is likely to introduce accidental complexity.

Suryadevara et al. [24] have compared the expressiveness of CCSL with that of timed automata and shown that logical clocks of CCSL where complementary to real-valued clocks of timed automata. Indeed, timed automata can introduce integer variables but they need to be bounded if one is to use the model-checker. Only when a CCSL specification is safe can the integer variables be bounded (this is the definition of safety). In that context, UppAal can verify (by model-checking) only the safe and physically-timed constraints of CCSL, when all the integer variables are bounded. Additionally, the study in [24] has shown that some forms of synchronizations supported by CCSL are difficult to express in a simple way with classical timed automata. The idea was then to process those synchronizations with ad-hoc techniques. BIP [25] proposes to use an algebra of connectors on top of timed automata to capture the interactions and alleviate this issue.

Finally, in classical or real-time schedulability analysis there are many efficient analytic results to compute a schedule. However, the assumptions made on the underlying model of computation or communication are usually very strong. In our case, CCSL captures logical and temporal constraints of various shapes without any assumptions on the global behavior (periodic tasks, harmonic executions, preemptive communications). It allows the quick prototyping of models. The system is modeled in a progressive way by the conjunction of constraints. Because of this large expressiveness we do not expect to fall always in a case where analysis is possible and/or meaningful. When a system clearly matches the assumptions of some analytical solution, our solution is most certainly not efficient.

3. Background

This section gives the necessary background to understand the contribution. It starts with an introduction to the Clock Constraint Specification Language and continues by describing CCSL constraints. It concludes by recalling the notations used on labeled transition systems and their synchronized product.

3.1. The Clock Constraint Specification Language

This section briefly introduces the logical time model [2] of MARTE and the Clock Constraint Specification Language (CCSL). A technical report [3] describes the syntax and the operational semantics of CCSL constraints. CCSL was built by defining some constraints with a simple physical interpretation. Other constraints are building as conjunctions of such *kernel* constraints.

CCSL clocks measure event occurrence dates in a system. Logical clocks [7] replace physical dates by a logical sequencing. We never presume that clocks or events are described relative to a global physical time but we rather consider that clocks are independent of each other.

Definition 1 (Logical clock). A *logical clock* c is defined as an infinite sequence (a stream) of *ticks*: $(c_n)_{n=1}^{\infty}$.

Clocks describes noticeable events in a system. In CCSL, the expected behavior of the system is described by a specification that constrains the way the clocks can tick. Basically, a CCSL specification prevents clocks from ticking when some conditions hold.

Definition 2 (CCSL specification). A CCSL *specification* is a tuple $Spec = \langle C, Cons \rangle$, where C is a finite set of clocks and $Cons$ is a finite set of constraints.

During the execution of a system, clocks tick according to occurrences of related events. The *schedule* captures what happens during one particular execution. A CCSL specification denotes a set of schedules. If empty, there is no solution, the specification is invalid. If there are many possible schedules, it leaves some freedom to make some choices depending on additional criteria. For instance, some may want to run everything as soon as possible (ASAP), others may want to optimize the usage of resources (processors/memory/bandwidth).

Definition 3 (Schedule). A *schedule* is a function $Sched : \mathbb{N} \to 2^C$. Given an *execution step* $s \in \mathbb{N}$, and a schedule $\sigma \in Sched$, $\sigma(s)$ denotes the set of clocks that *tick* at step s.

Definition 4 (Valid schedule). For a given specification $Spec = \langle C, Cons \rangle$, a schedule σ is *valid* ($\sigma \models Spec$) if and only if it satisfies all the constraints of $Spec$: $\forall cons \in Cons, \sigma \models cons$.

Note that there are usually an infinite number of valid schedules for a specification, we only consider the ones that do not have empty steps: $\forall n \in \mathbb{N}, \sigma(n) \neq \emptyset$. Physically, empty steps represent instants where nothing relevant happens. There is an infinite number of ways to add stuttering steps for a given schedule and adding a finite number of empty steps does not bring any useful information to the safety issue.

Given a specification, the goal of CCSL clock calculus is to find whether or not there is at least one valid schedule. Exhaustive analysis consists in establishing common properties on all the valid schedules that satisfy a given set of constraints.

The mechanism described here is general and does not assume any specific way to define the semantics of CCSL constraints. In practice some CCSL constraints are primitive and are used to define other (non-primitive) constraints by composition.

3.2. Primitive constraints in CCSL

Some CCSL constraints are stateless, *i.e.,* the constraint imposed on a schedule is identical at all steps; others are stateful, *i.e.,* they depend on what has happened in previous steps.

The first basic stateless CCSL constraint is Subclocking, which prevents a (sub)clock from ticking when a (master) clock cannot tick.

Definition 5 (Subclocking). Let a, b be two logical clocks. A schedule σ satisfies the subclocking constraint on a and b ($a \boxed{\subseteq} b$) if the following condition holds: $\sigma \models a \boxed{\subseteq} b \iff \big(\forall n \in \mathbb{N}, a \in \sigma(n) \implies b \in \sigma(n)\big)$.

Contrary to fully synchronous systems, we never assume the existence of a global master clock from which all the other clocks should derive. When $a \boxed{\subseteq} b$ and $b \boxed{\subseteq} a$, the two clocks a and b are said to be synchronous, this is denoted as $a \boxed{=} b$.

Two simple stateless CCSL constraints are Union and Exclusion.

Definition 6 (Union). Let a, b be two logical clocks. A schedule σ satisfies the union constraint on a and b if the following condition holds:
$\sigma \models u \triangleq a \boxed{+} b \iff \big(\forall n \in \mathbb{N}, u \in \sigma(n) \iff (a \in \sigma(n) \vee b \in \sigma(n))\big)$

Note that Union is commutative and associative, we use in next sections an n-ary extension of this binary definition. A dual operator is the Intersection $(a \boxed{*} b)$ defined by replacing the disjunction by a conjunction.

Definition 7 (Exclusion). Let a, b be two logical clocks. A schedule σ satisfies the exclusion constraint on a and b if the following condition holds:
$\sigma \models a \boxed{\#} b \iff \big(\forall n \in \mathbb{N}, a \notin \sigma(n) \vee b \notin \sigma(n)\big)$

Stateful constraints rely on the history of clocks for a specific schedule.

Definition 8 (History). Given a schedule σ, the *history* over a set of clocks C is a function $H_\sigma : C \to (\mathbb{N} \to \mathbb{N})$ defined inductively for all clocks $c \in C$:

$$\begin{aligned} H_\sigma(c)(0) &= 0 \\ \forall n \in \mathbb{N}, c \notin \sigma(n) &\implies H_\sigma(c)(n+1) = H_\sigma(c)(n) \\ \forall n \in \mathbb{N}, c \in \sigma(n) &\implies H_\sigma(c)(n+1) = H_\sigma(c)(n) + 1 \end{aligned}$$

For a clock $c \in \mathcal{C}$, and a step $n \in \mathbb{N}$, $H_\sigma(c)(n)$ denotes the number of times the clock c has ticked when reaching step n within the schedule σ.

The primitive stateful CCSL clock constraint is Causality. When an event causes another one, the effect cannot occur if the cause has not. In CCSL, causality can be instantaneous.

Definition 9 (Causality). Let a, b be two logical clocks. A schedule σ satisfies the causality constraint on a and b if the following condition holds:
$\sigma \models a \boxed{\preccurlyeq} b \iff \big(\forall n \in \mathbb{N}, H_\sigma(a)(n) \geq H_\sigma(b)(n)\big)$

A small extension of Causality includes a notion of temporality and is called Precedence.

Definition 10 (Precedence). Let a, b be two logical clocks and $\delta \in \mathbb{Z}$. A schedule σ satisfies the precedence constraint on a and b if the following condition holds:
$\sigma \models a \boxed{\delta \prec} b \iff \big(\forall n \in \mathbb{N}, H_\sigma(b)(n) - H_\sigma(a)(n) = \delta \implies b \notin \sigma(n)\big)$

This is a generalization of the primitive notion of precedence. For instance, *put* $\boxed{n \prec}$ *get* denotes an infinite FIFO with n tokens initially present. When *put* ticks, one token is added into the FIFO. When *get* ticks, one token is removed. When n tokens have been removed then removing is not possible anymore.

The primitive CCSL precedence is defined as: $a \boxed{\prec} b \equiv a \boxed{0 \prec} b$, *i.e.,* an initially empty infinite FIFO. When $a \boxed{\prec} b$, clock a is said to be faster than clock b. A bounded version of precedence (*e.g.,* bounded FIFO) is defined as $a \boxed{\prec_N} b \equiv a \boxed{\prec} b \wedge b \boxed{N \prec} a$.

Given two clocks a and b, their Infimum (resp. Supremum) is informally defined as that the slowest (resp. fastest) clock faster (resp. slower) than both a and b.

Definition 11 (Infimum). Let a, b, inf be three clocks. A schedule σ satisfies the infimum constraint if the following condition holds:
$\sigma \models inf \triangleq a \boxed{\wedge} b \iff \big(\forall n \in \mathbb{N}, H_\sigma(inf)(n) = max(H_\sigma(a)(n), H_\sigma(b)(n))\big)$

The supremum is dual and is built by replacing max by min. Infimum and Supremum are useful to group events occurring at the same pace and decide which one occurs first and which one occurs last.

Another example of a stateful constraint used in this paper is the DelayFor constraint. Such a constraint delays a 'base' clock by counting the ticks of a 'reference' clock.

Definition 12 (DelayFor). Let $base$, ref and res be three logical clocks and $N \in \mathbb{N}$. A schedule σ satisfies constraint DelayFor if the following condition holds:

$$\sigma \models res \triangleq base \$ N \ on \ ref \iff$$
$$\big(\exists n \in \mathbb{N}, res \in \sigma(n) \iff ref \in \sigma(n) \wedge$$
$$\exists m \leq n, base \in \sigma(m) \wedge H_\sigma(ref)(n) - H_\sigma(ref)(m) = N\big)$$

Note that this operator has both a sampling and a synchronization effect. Indeed, the delayed clock is synchronous with the ref clock but some ticks of the base clock may be lost if the ref clock is faster than the $base$ clock. When $base$ and ref are the same clock, we get the simple synchronous delay operator of Signal: $c \$ N \ on \ c \iff c \$ N$

Definition 13 (PeriodicOn). Let *base* and *res* be two logical clocks and a period $P \in \mathbb{N} \setminus \{0\}$. A schedule σ satisfies constraint PeriodicOn if the following condition holds:

$$\sigma \models res \triangleq \texttt{PeriodicOn}\, base\, \texttt{period=}P \iff$$
$$\forall n \in \mathbb{N}, res \in \sigma(n) \iff \big(H_\sigma(base)(n) = P * H_\sigma(res)(n) \wedge base \in \sigma(n)\big)$$

It is easy to show that when a clock is periodic on another one, then it is a subclock of it: $res \triangleq \texttt{PeriodicOn}\, base\, \texttt{period=}P$ then $res \boxed{\subseteq} base$. This is a logical definition of periodicity, where *res* is P-times slower than *base*. If the period is one, then the two clocks are synchronous. If we assume now that s is a physical (discrete) clock, for instance ticking every second, then $\_10s \triangleq \texttt{PeriodicOn}\, s\, \texttt{period=}10$ defines a periodic clock $\_10s$ ticking every 10^{th} second. Since CCSL is a relational language, the same operator can also be used to build faster clocks. For instance, still assuming the existence of s, $s \triangleq \texttt{PeriodicOn}\, \_10Hz\, \texttt{period=}10$ defines a clock $\_10Hz$ ticking ten times per second, which does not necessarily mean every 0.1 s though.

Definition 14 (sampledOn). Let *base*, *c*, and *res* be three logical clocks. A schedule σ satisfies constraint sampledOn if the following condition holds:

$$\sigma \models res \triangleq c\, \texttt{sampledOn}\, base \iff$$
$$\forall n \in \mathbb{N}, res \in \sigma(n) \iff \big(\exists m < n, base \in \sigma(m) \wedge base \in \sigma(n) \wedge$$
$$H_\sigma(base)(n) - H_\sigma(base)(m) = 1 \wedge$$
$$H_\sigma(c)(n) - H_\sigma(c)(m) \geq 1\big)$$

This is a sampling operator. It takes two clocks (*c* and *base*) and produces the fastest clock slower than *c* that is a subclock of *base*, *i.e.,* it synchronizes *c* on *base*. It is easy to show that $res \boxed{\subseteq} base \wedge c \boxed{\preccurlyeq} res$.

Note that if *c* is too fast (compared to *base*), then some occurrences of *c* are lost in the sampled clock.

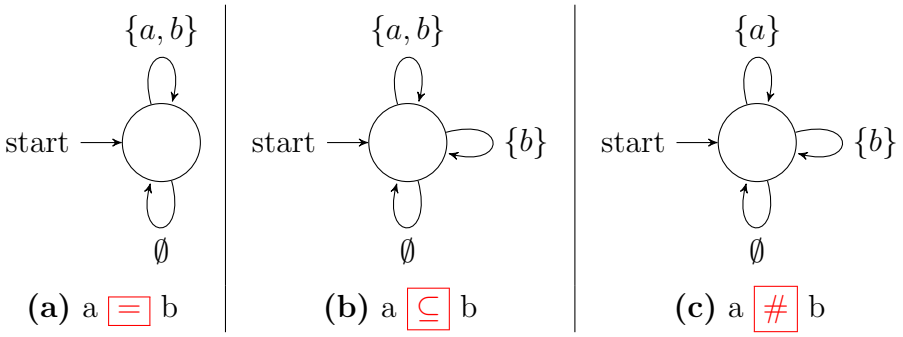
3.3. Synchronized product of transition systems

Labeled transition systems are often used to capture the semantics of languages [26]. We explain here the usage made of such transition systems in the context of our clock specification and recall briefly basic notions on the synchronized product.

Definition 15. A *Clock-Labeled Transition System (cLTS)* is defined as a tuple $\mathcal{A} = \langle S, T, s0, C \rangle$ where

- S is a set of *states*,
- $s0 \in S$ is the *initial state*,
- C is a finite set of clocks,
- $T \subseteq S \times 2^C \times S$ is a set of *transitions*, with $(s, Y, s') \in T$ means that all the clocks in $Y \subseteq C$ tick when the transition from s to s' is fired.

CCSL constraints can be captured as a cLTS. For instance, compare Figure 1(b) to the semantics given in Definition 5. There is only one state since this constraint does not depend on history. There are three transitions out of the four possible ones. The one left out is $\{a\}$, where a would tick alone without b, which is obviously forbidden by the subclocking relation $a \boxed{\subseteq} b$.



(a) a $\boxed{=}$ b **(b)** a $\boxed{\subseteq}$ b **(c)** a $\boxed{\#}$ b

Figure 1: Primitive CCSL relations as clock-Labeled Transition Systems

Note that it is always possible to do nothing (see transitions with empty set). This transition is required for composition purpose. However, once the full transition system is built, the resulting composed transitions with empty set are deleted to remove stuttering steps as discussed before.

Some constraints are stateful and may even require an infinite number of states. Consider for instance, the primitive precedence $a \boxed{\prec} c$, its transition system is given in Figure 2. Such constraints, with an infinite number of states, are called *unsafe*.

A CCSL specification is a conjunction of constraints and its behavior is computed as the synchronized product of all the transition systems for each

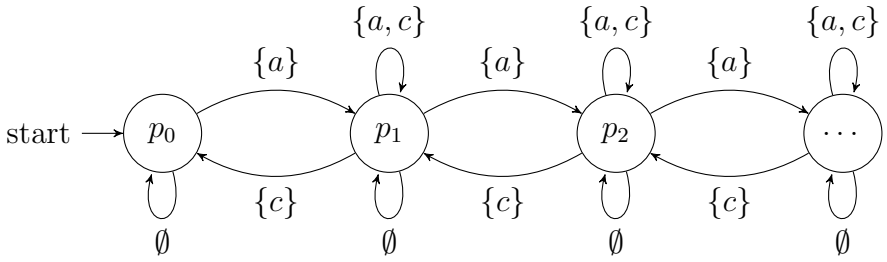


Figure 2: CCSL precedence (infinite state cLTS): a precedes c.

constraint. The synchronization occurs on the clocks and the transition systems need, at each step, to agree on the fate of each clock.

Definition 16. If, for $i \in \{1..n\}$, $\mathcal{A}_i = \langle S_i, T_i, s0_i, C_i \rangle$ is a clock-labeled transition system, the *synchronized product* [26, 8] of $\mathcal{A}_i$ is the labeled transition system $\langle S, T, s0, C \rangle$ defined by

- $S = S_1 \times \ldots \times S_n$,
- $T = \{\big\langle (s_1, Y_1, s'_1), \ldots, (s_n, Y_n, s'_n) \big\rangle \in T_1 \times \ldots \times T_n\}$
  $\iff \forall i, j \in \{1..n\}, (C_i \cap Y_j) = (C_j \cap Y_i)$,
- $s0 = (s0_1, \ldots, s0_n)$,
- $C = \bigcup_{i \in \{1..n\}} C_i$.

Two cLTS synchronize only on their common clocks when they both agree on whether the common clocks tick or not. This definition is a classical definition introduced by Arnold et al. [26] and later extended [8]. However it is adapted to the cLTS and in particular to the fact that we have sets of clocks as labels for the transitions. It does not assume that the number of states is finite. Even though the basic transition systems are infinite, the number of reachable states in the synchronized product may be finite [6]. The following section discusses two alternative solutions to encode the unsafe constraints and to compute the synchronized product of infinite transition systems. Both solutions use the same algorithm, which terminates if and only if the product has a finite number of states.

Algorithm 1. *Synchronized product through reachability analysis*

1. Let $S \leftarrow \emptyset$, $T \leftarrow \emptyset$,
2. Let $S' \leftarrow \{(s0_1, \ldots, s0_n)\}$
3. while S' is not empty {
4. Let $st = (st_1, \ldots, st_n)$ be one element of S'
5. Let $S \leftarrow S \cup \{st\}$
6. Let $S' \leftarrow S' \setminus \{st\}$
7. $\forall \big\langle \langle s_1, Y_1, s'_1 \rangle, \ldots, \langle s_n, Y_n, s'_n \rangle \big\rangle \in T_1 \times \ldots \times T_n$ such that
8. $\forall i \in \{1..n\}$
9. $\big(s_i = st_i \wedge (\forall j \neq i \in \{1 \ldots n\})(C_i \cap Y_j = C_j \cap Y_i)\big)$ {
10. Let $st' = st'_1 \times \ldots \times st'_n$
11. if $st' \notin S$ then $S' \leftarrow S' \cup \{st'\}$
12. $T \leftarrow T \cup \{\langle st, \bigcup_{i \in \{1..n\}} Y_i, st' \rangle\}$
13. }
14. }

4. State-space exploration for CCSL specifications

4.1. Introduction

This section compares two techniques to compute the synchronized product and to explore the state-space of a CCSL specification. To illustrate and compare the two techniques we use a simple specification. We are looking for all the schedules σ such that: $\sigma \models a \boxed{\prec} c \wedge \sigma \models b \triangleq a \$ 1 \wedge \sigma \models c \boxed{\prec} b$

In this example, the LTS for the two precedes operators are infinite (see Def. 10). Consequently, the product also has a infinite number of states. However, as we will see some of the states in the product are not reachable leaving only a finite number of reachable states. We explore two alternative techniques to build this finite set of states and compare them.

4.2. Extended finite state machines

The first solution is an extension of our earlier work [5]. It consists in encoding the infinite transition systems with extended finite state machines, *i.e.,* state machines with a finite number of states and a finite set of unbounded integer variables. Compared to the previous work, it introduces the notion of state invariant (see Fig. 3) that helps reducing the product to a smaller automaton by removing states that are not reachable.

Each state machine has a finite number of integer variables. These variables, called δ-counters, compute the advance of one clock over another one for a given schedule.

Definition 17 (δ-counters). Given a schedule σ, *δ-counter* is a function $\delta_\sigma : (C \times C) \to (\mathbb{N} \to \mathbb{N})$ defined as follows: $\forall c1, c2 \in C$ and $\forall s \in \mathbb{N}$: $\delta_\sigma(c1, c2)(s) = H_\sigma(c1)(s) - H_\sigma(c2)(s)$.

Each transition owns a Boolean guard built on top of the δ-counters. Each transition is labelled with a set of clocks that tick when the transition is fired. Figure 3 shows an example where the infinite cLTS from Figure 2 is encoded using an extended finite state machine. The two transitions from state $P1$ and labeled {c} have a guard. Depending on the history, either the transition leads to state $P0$ or remains in state $P1$. Note that the states are not built arbitrarily. The states determine precisely which clocks are forbidden to tick. The guards on the transitions are not used to prevent a given configuration of ticking clocks but only to decide what is the next state. This is very important to determine possible schedules at a given step without having to interpret the guards.

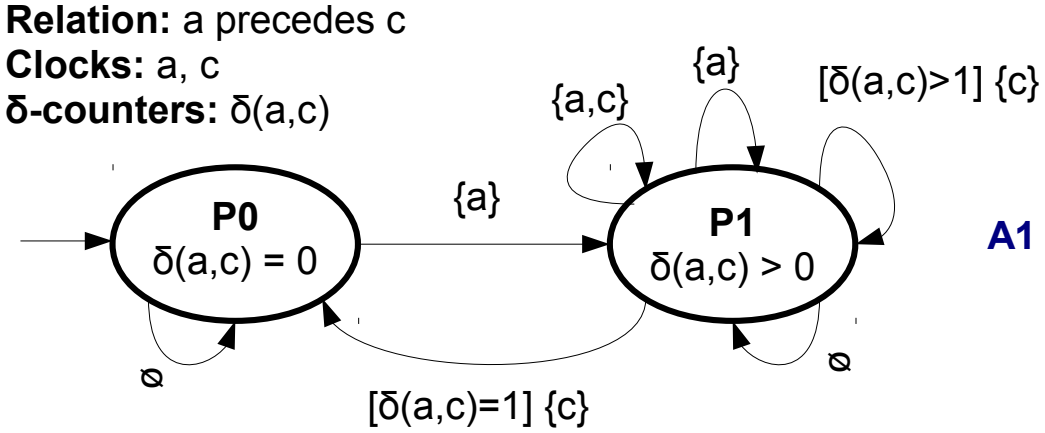


Figure 3: $a \prec c$ as an Extended Finite State Machine

Each state of the state machine is associated with a state invariant. A state invariant gives a minimal and maximal bound for some δ-counters.

Definition 18 (State invariant). Given a cLTS $\mathcal{A} = \langle S, T, s0, C \rangle$, a *state invariant* is a function $inv : S \to (C \times C \to (\mathbb{Z} \cup \{-\infty\}) \times (\mathbb{Z} \cup \{\infty\}))$.

For instance, on Figure 3, the invariant $inv(P0)(a, c) = (0, 0)$ (and denoted $\delta(a, c) = 0$) means that in state $P0$, clocks a and c have had exactly the same number of ticks. This is initially true since no clock has ticked and this is also maintained by the guards on transitions. $inv(P1)(a, c) = (1, \infty)$ (denoted $\delta(a, c) > 0$) means that in state $P1$, a has ticked strictly more often than c.

Figure 4 gives the two other extended finite state machines for the two other constraints of our example. Each one of the three state machines has two states, then the product has 2x2x2=8 states. However, not all these states are actually reachable. To explore which state is reachable and which one is not, we need to interpret the guards and combine them with the state invariants. This means building a set of inequations on integer variables and solving such a system. We have decided to restrain ourself to linear inequations and to use integer linear programming (ILP) to solve the system. Our implementation relies on CPlex.

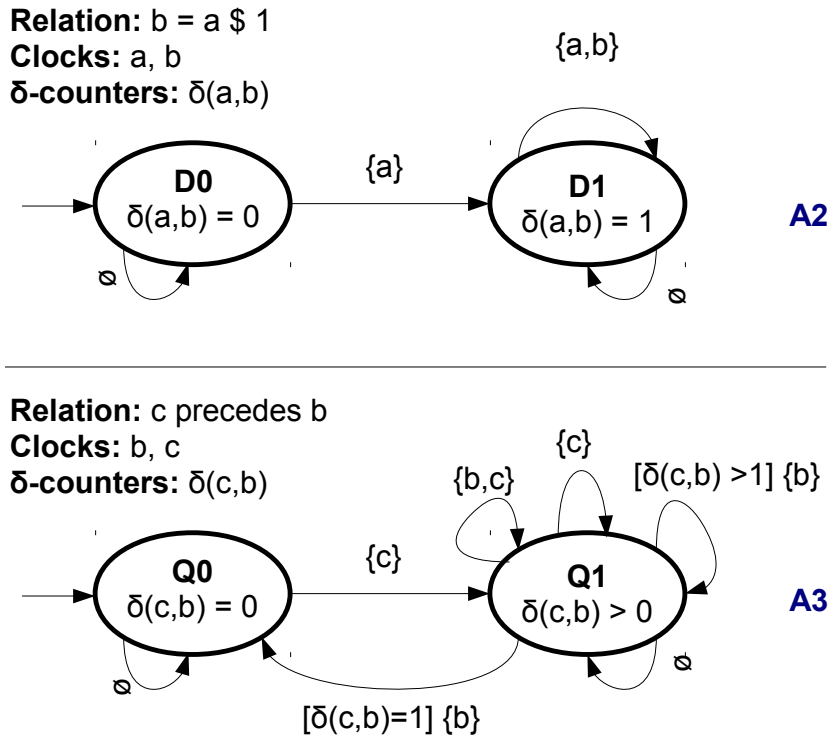


Figure 4: Extended Finite State Machines example

We proceed by extending Algo. 1 as follows.

Algorithm 2. *Synchronized product of extended FSM*

1. Let $S \leftarrow \emptyset$, $T \leftarrow \emptyset$, $S' \leftarrow \{(s0_1, \ldots, s0_n)\}$
2. While S' is not empty {
3. Pick a state s from S' and remove it
4. $\forall t = (s, Y, s')$ that satisfies the rules of the synchronized product between the n transitions $t_i = (s_i, Y_i, s'_i)$ {
5. $StateInvariant(s) = \bigwedge_{i \in \{1..n\}} StateInvariant(s_i)$,
6. $StateInvariant(s') = \bigwedge_{i \in \{1..n\}} StateInvariant(s'_i)$,
7. $Guard(t) = \bigwedge_{i \in \{1..n\}} Guard(t_i)$,
8. If $Guard(t)$ violates $StateInvariant(s)$ then discard t
9. If $\neg Guard(t)$ violates $StateInvariant(s)$ then keep t but discard its guard (which is useless)
10. If $StateInvariant(s')$ has no solution then discard s'
11. Otherwise, $S' \leftarrow \{s'\}$ and $T \leftarrow \{t\}$
12. }
13. }

Figure 5 shows the resulting product which has only three reachable states out of the eight initially possible. Only the four states that are actually explored are shown.

We start from the product of the initial states $P0 \times D0 \times Q0$. The state invariant is the conjunction of the invariants of each three composed states ($P0$, $D0$, $Q0$). Exploring the outgoing transitions, we can eliminate right away two solutions where synchronizations are not possible (see Line 4 in Algo. 2, which is equivalent to Lines 7-9 of Algo. 1). There is one solution (apart from doing nothing that is always possible) which is firing a alone. For instance, firing c alone is not possible since it would imply synchronizing transition $\{c\}$ from $Q0$ with the empty transitions from both $P0$ and $D0$. However, because $(\{a, c\} \cap \{c\}) \neq (\emptyset \cap \{b, c\})$, transition $\{c\}$ cannot synchronize with the empty transition from $P0$. Intuitively, the empty self-transition from $P0$ means neither a nor c must tick, which contradicts c ticking alone. Taking transition $\{a\}$ leads from $P0 \times D0 \times Q0$ to $P1 \times D1 \times Q0$.

All the transitions with the stop sign are the ones that can be excluded by this simple synchronization analysis. Other transitions need to take into

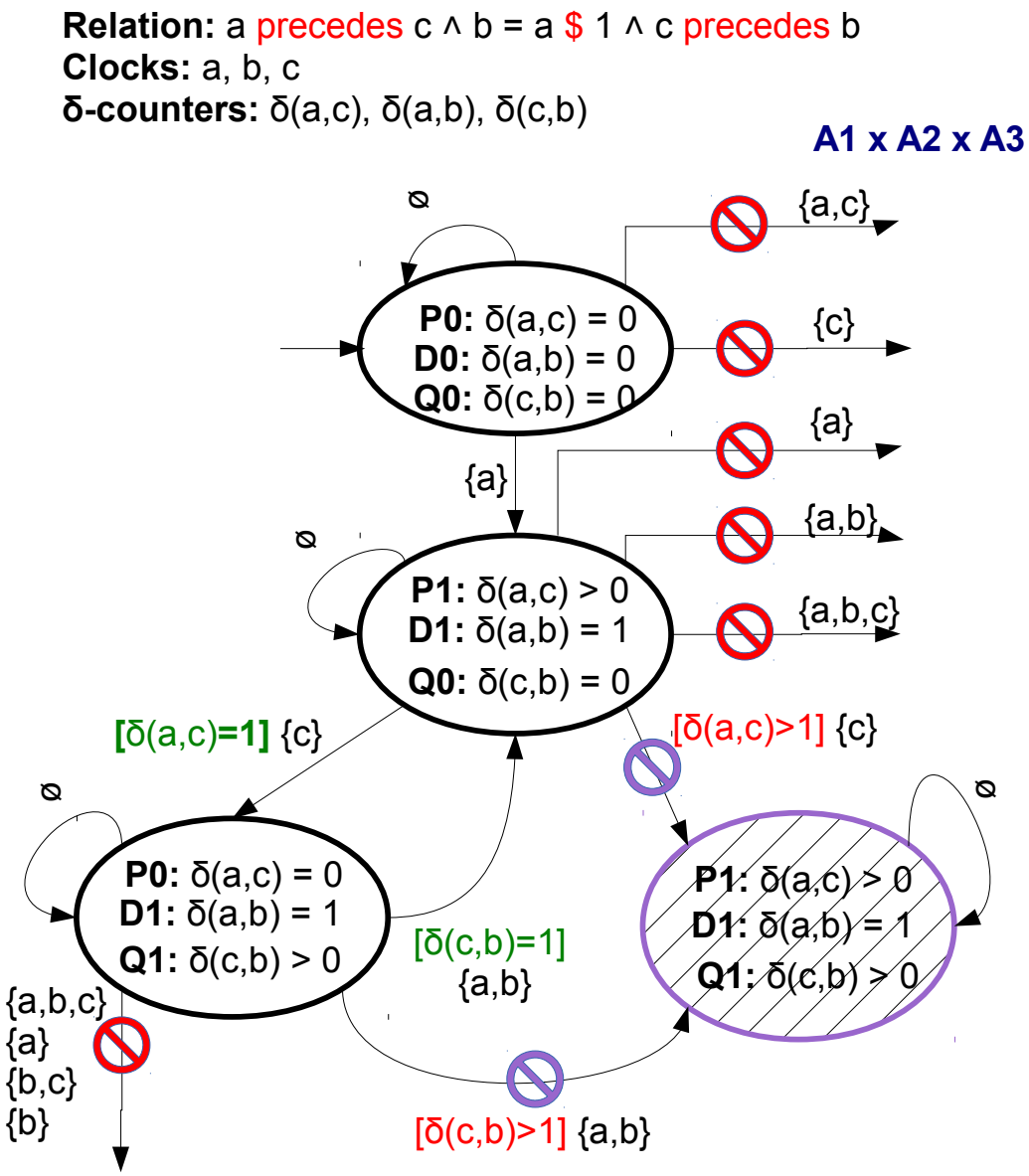


Figure 5: Product of extended Finite State Machines

account the guards and the invariants. Let us consider for instance, state $P1 \times D1 \times Q0$. Applying the synchronization analysis some transitions are discarded. However, there are two remaining outgoing transitions (apart from doing nothing) that involve guards. From the state invariant we deduce

that $\delta(a, c) = 1$ which contradicts the guard of transition $\{c\}$ going to $P1 \times D1 \times Q1$. Then this transition is discarded (see line 8 in Algo. 2).

Because, the state invariant already implies $\delta(a, c) = 1$ then the guard of the guard of transition $\{c\}$ going to $P0 \times D1 \times Q1$ can never be falsified. The transition is maintained, but the guard is discarded (set to true) as described in line 9 of Algo. 2.

Finally, because the two transitions entering state $P1 \times D1 \times Q1$ have been discarded, then the state has to be discarded as well since it is not reachable. Eventually, we get a purely boolean cLTS with only three states all the δ-counters become useless since they are not used in any guard. This was the expected result since we know that this specification is safe [6].

This first solution introduces unbounded integer variables and relies on integer linear programming to reduce the size of the automaton. It is important to reduce the size since this is one of the main limitations to address larger problems. The question of whether we get a minimal solution is still open. Acceleration techniques [20] might provide a more efficient encoding but this question still needs to be explored.

4.3. Lazy evaluation

We now explore a second solution that relies only on a purely boolean abstraction and abstract the infinite state space with an intentional data structure based on lazy evaluation [27]. However, in that case Algo. 1 is not guaranteed to terminate since the number of states is potentially infinite. This second method should be preferred when we know that the CCSL specification is safe.

We use a data structure that *intentionally* captures the whole set of unbounded natural numbers. Of course the data structure is never built in extension, as this would blow the memory off. Instead, only the part required to build the synchronized product is produced as soon as required. However, when there is an infinite number of reachable states then the states are progressively built until the memory limit is reached. So, our main assumption is that we work with products where only a finite number of states are reachable. In [6] we have proposed an algorithm to decide whether a CCSL specification is safe or not.

Figure 6 shows the lazy data structure for building the CCSL precedence. The structure is intentionally infinite. Unbounded natural numbers have at most one successor and exactly one predecessor. *Zero* is an exception since it has no predecessor and its potential successor is one. Each natural number

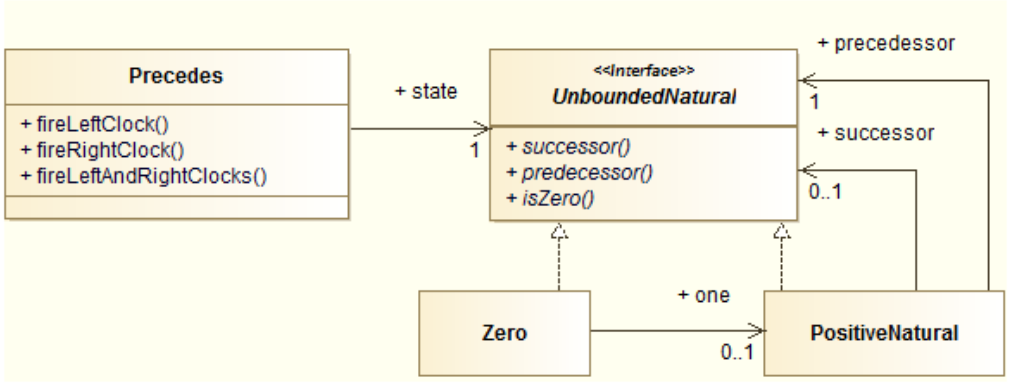


Figure 6: Lazy data structure for precedence

is defined has the successor of another natural number but is only built if required. These unbounded natural numbers encode the δ-counters. Hence, there is no need to have guards since all states have their own identity. In the previous solution, the guards and δ-counters were maintaining the history of the system, or at least part of it.

Figure 7 shows the result of the synchronized product applied to our small example. Note that Algo. 1 terminates only if there is a finite number of states in the product (which is the case here). On the left side of Fig. 7, the states are built according to Algo. 1. Each state is always built in extension and therefore we do not need guards to tell them apart. On the right hand side, the objects are instances of classes shown on Fig. 6. The state information for the delay are built using a standard integer (type int) since the automaton for the delay is finite (see δ(a,b) in italic on the figure). The two top-most objects on the right-hand side are the two ones that are actually built in extension. In each state, the δ-counters point to the right 'unbounded integer'. As soon as one δ-counter needs to access a *new* natural number, it is effectively built along with all of its predecessors. All the successors of one are left intentional here (within the dash-circled area) since this particular product only needs two natural numbers (0 and 1). The intentional domain is however potentially infinite.

4.4. Comparing the two approaches

Let us now resume the main differences between the two approaches.

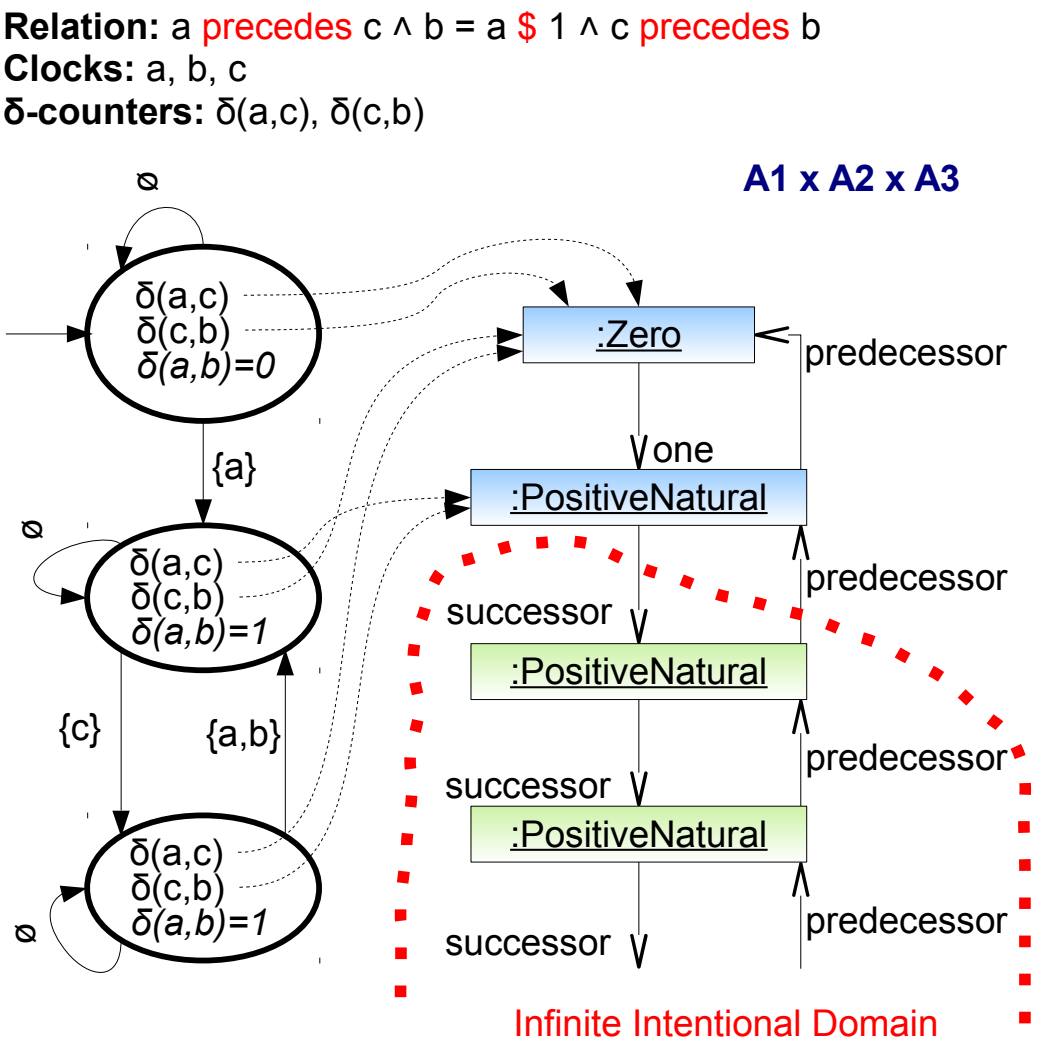


Figure 7: Synchronized product with lazy data structure

When using extended finite state machines, Algo. 1 always terminates since it enumerates all the solutions on a finite abstraction. The state invariants are used to reduce the number of states and we need to solve a system of integer linear inequations for that purpose. When some of the integer guards could not be reduced, it can be either because the system is actually infinite or because the state invariants were not strong enough. We use only linear inequations in state invariants to ensure that CPlex can solve it, but in some cases, there are stronger non linear inequations that can be established.

When using the intentional data structure, it is unfolded on demand. If the system is finite and memory allows, then the algorithm terminates. When we run out of memory, either the system is infinite or it is too large to be built. The interest of that method is that it does not require state invariants or integer linear programming. The flaw is that it consumes much more memory than the first solution. Acceleration techniques may here be key to address larger systems. This second method should only be applied when we know the system is finite, however, finiteness can now be decided [6].

5. Example: CCSL for capturing the architecture, application and allocation

To illustrate the approach, we take an example inspired by [28], that was used for flow latency analysis on AADL[2] specifications [29].

5.1. Application

Figure 8 (on the top) considers a simple application described as a UML structured class. 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.

To abstract this application and capture a CCSL specification, we assign one clock for each action. The clock has the exact same name as the associated action (*e.g.,* $step1$). We also associate one clock with each input (*e.g.,* $in1$), and one clock to the production of the output (*e.g.,* out). The successive instants of the clocks represent successive executions of the actions or capturing of inputs or output production.

The basic CCSL specification is given as follows:

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

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

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

[2]AADL stands for Architecture & Analysis Description Language

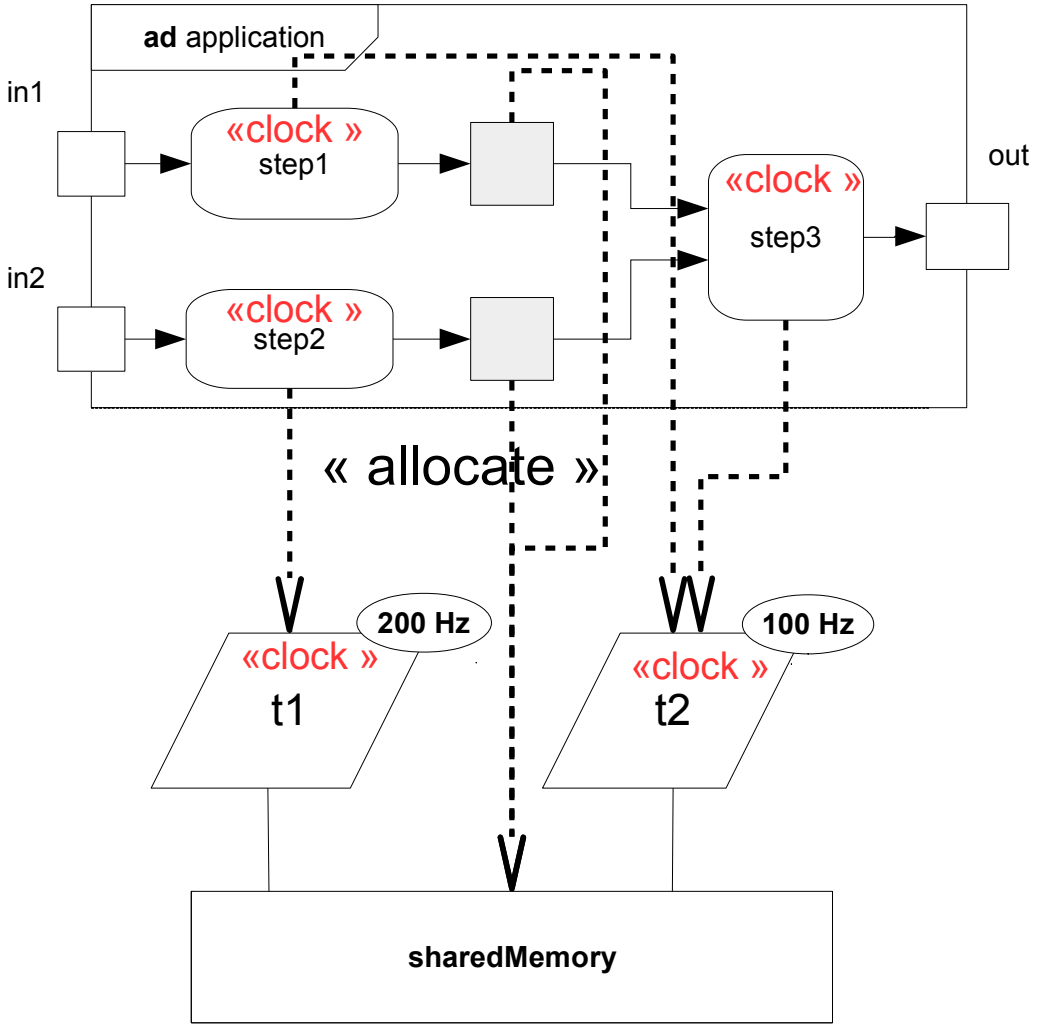


Figure 8: Simple application

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 for *in*2 and *step*2. Finally, Eq. 3 states that an output can be produced as soon as *step*3 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 safe, therefore TimeSquare cannot perform any kind of exhaustive analysis and can only produce a particular schedule that matches the specification (see Fig. 9).

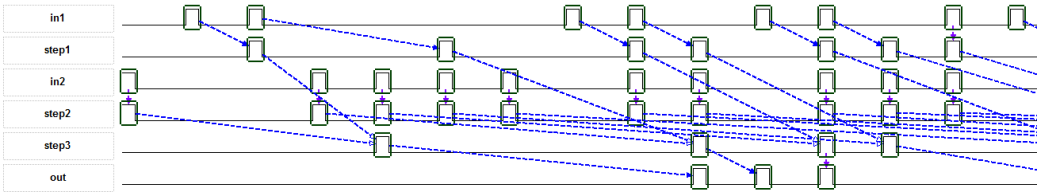


Figure 9: A valid schedule for the application part of Fig. 8

On 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.

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

The effect of this constraint can be seen on Figure 10. Looking carefully at this schedule, we can note that the arrival of $in2$ has been slowed 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.

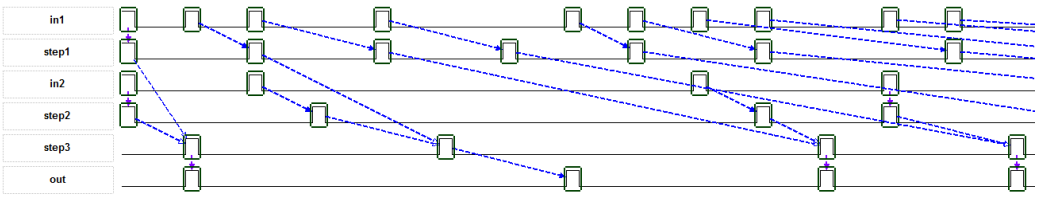


Figure 10: Another valid schedule for the application part of Fig. 8

Reachability analysis as described in Section 4 tells us that the composition is still not bounded because bounds on $in1 \boxed{\vee} in2$ does not imply bounds on both $in1$ and $in2$ but bounding only one the two clocks is enough. To have a complete finite systems, we can for instance replace Eq. 4 by Eq. 5.

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

By doing so, our reachability analysis algorithm converges and produces the state-space shown in Figure 11[3].

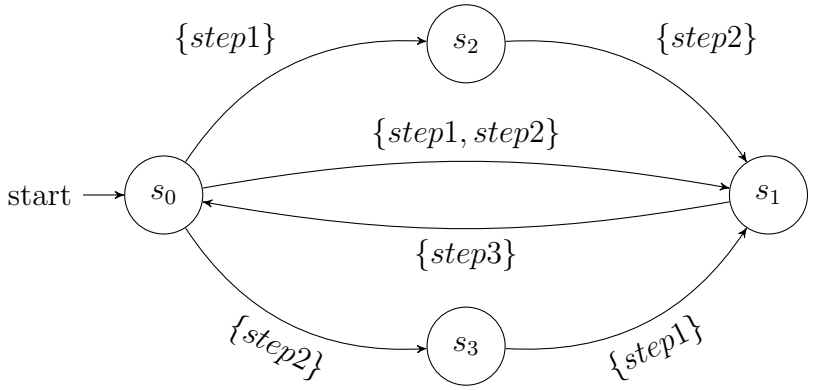


Figure 11: Synchronized product of Eqs. 1-3 and Eq. 5.

We have removed $in1$, $in2$, and *out* since they were just adding interleaving without offering more actual parallelism in the execution of actions.

Building the state-space is also useful to detect liveness issues in 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 12. This figure shows a typical case of (partial) deadlock in CCSL. Indeed, if from the initial state s_0, we decide to fire $in1$ (resp. $in2$) alone, then Eq. 6 prevents $in1 + in2$ from ticking again before *out* ticks, but $in2$ (resp. $in1$) 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.

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

This is a partial deadlock that illustrates that CCSL specifications can have conflicts. There are some schedules that are live and in which all clocks can tick forever (*step*1 and *step*2 tick together). There are other schedules that are not live and that we call *bad-paths* (*step*1 or *step*2 tick alone).

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

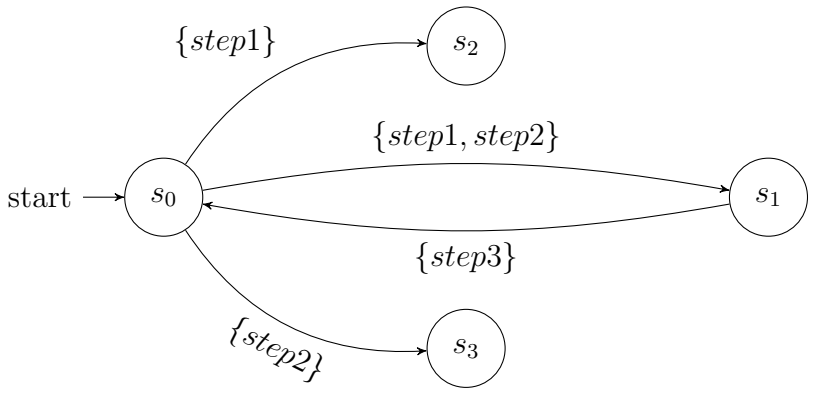


Figure 12: Synchronous products of Eqs. 1-3 and Eq. 6.

5.2. Execution platform and allocation

Once the application is designed, then CCSL can also be used to capture the execution platform. Figure 8 (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:

$$t1 \triangleq \texttt{PeriodicOn}\, ms\, \texttt{period=}5 \tag{7}$$

$$t2 \triangleq \texttt{PeriodicOn}\, t1\, \texttt{period=}2 \tag{8}$$

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. 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 *step*2 and *step*3 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). *step*3 needs inputs from both *step*1 and *step*2 before executing but it can execute only according to the sampling period of $t1$ since *step*3 is allocated to $t1$ (Eq. 12). Finally, all steps can only

execute when their input data have been sampled (Eq. 13).

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

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

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

$$d3\_s \triangleq (step1 \boxed{\wedge} step2) \text{ sampledOn } t1 \tag{12}$$

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

All these new constraints do not change anything on the finiteness of the whole system. They only reduce the set of possible executions and interlivings. If the application specification was finite, then its allocated version is still finite. If it was infinite, then it remains infinite. Whether it is finite or not, TimeSquare can produce an execution of this specification (see Fig. 13). 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.

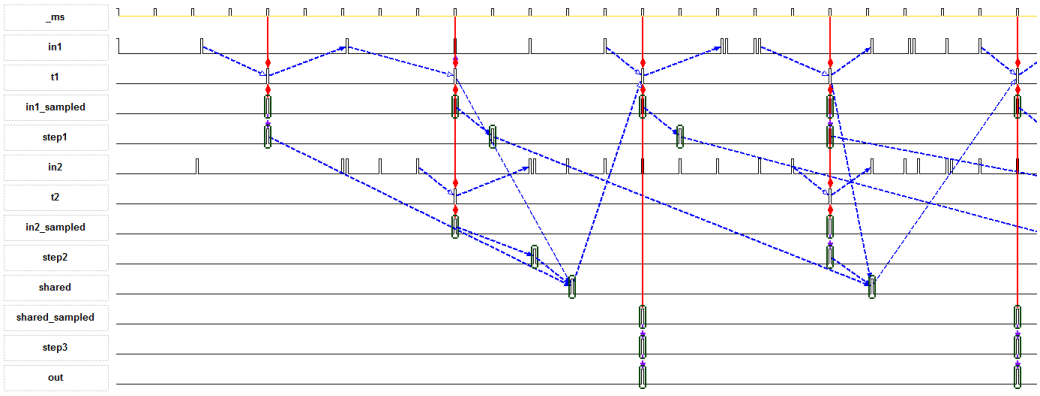


Figure 13: A valid schedule for the allocated application (Fig. 8)

6. Conclusion

The paper has discussed and compared two practical solutions to encode the semantics of CCSL operators with extended finite state machines and infinite intentional labeled transition systems. These two encodings lead to

two different ways to compute the parallel composition of these automata in order to explore correctness issues that may arise.

The first solution is an extension of some previous work and relies on integer linear programming (ILP) to compute the product. Our extension in this paper proposes to add state invariants to help discard some additional unreachable states thus reducing the size of the resulting state machine. However, we have not proved that this resulting state machine is minimal in states. We leave that part for another work.

The second solution relies on an intentional data structure and does not use ILP but only the traditional synchronized product algorithm [26]. The algorithm is adapted to process through reachability analysis to build only the reachable states. However, since the result may have an infinite number of reachable states, the semi-algorithm may not always terminate.

Finally, we also show that some safety and liveness properties can be effectively checked using both techniques on an example borrowed from the AADL community. This example serves to show how CCSL can be used to capture data dependencies on applications. The execution platform is also modeled in CCSL with a uniform notation to deal with both logical and physical periodic relations. The allocation is finally allocated onto the execution platform to produce a schedule of the mapped application.

In this work, the synchronized product is, in the two proposed solutions, built explicitly. This obviously may lead to practical limitations when the state-space is too large. In future works, we intend to improve the data structure to be able to address larger cases. One hint would be to use symbolic representations of the transitions rather than explicit ones as here. In the simulation engine of TimeSquare, we actually use binary decision diagrams to compute the solutions at one particular simulation step.

References

[1] OMG, UML Profile for MARTE, v1.0, Object Management Group, formal/2009-11-02 (November 2009).

[2] C. André, F. Mallet, R. de Simone, Modeling time(s), in: 10th Int. Conf. on Model Driven Engineering Languages and Systems (MODELS '07), no. 4735 in LNCS, ACM-IEEE, Springer, Nashville, TN, USA, 2007, pp. 559–573. doi:10.1007/978-3-540-75209-7_38.

[3] C. André, Syntax and semantics of the Clock Constraint Specification Language (CCSL), Research Report 6925, INRIA (May 2009).
URL http://hal.inria.fr/inria-00384077/

[4] J. Deantoni, F. Mallet, Timesquare: Treat your models with logical time, in: C. A. Furia, S. Nanz (Eds.), TOOLS (50), Vol. 7304 of Lecture Notes in Computer Science, Springer, 2012, pp. 34–41. doi:10.1007/978-3-642-30561-0_4.

[5] F. Mallet, Automatic generation of observers from MARTE/CCSL, in: 23rd IEEE Int. Symp. on Rapid System Prototyping, RSP 2012, IEEE, 2012, pp. 86–92. doi:10.1109/RSP.2012.6380695.

[6] F. Mallet, J.-V. Millo, R. de Simone, Safe CCSL specifications and marked graphs, in: 11th ACM/IEEE Int. Conf. on Formal Methods and Models for Codesign, IEEE, 2013, pp. 157–166.

[7] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Commun. ACM 21 (7) (1978) 558–565.

[8] A. Arnold, G. Point, A. Griffault, A. Rauzy, The altarica formalism for describing concurrent systems, Fundam. Inform. 40 (2-3) (1999) 109–124. doi:10.3233/FI-1999-402302.

[9] G. Berry, L. Cosserat, The esterel synchronous programming language and its mathematical semantics, in: S. D. Brookes, A. W. Roscoe, G. Winskel (Eds.), Seminar on Concurrency, Vol. 197 of LNCS, Springer, 1984, pp. 389–448.

[10] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, R. de Simone, The synchronous languages 12 years later, Proceedings of the IEEE 91 (1) (2003) 64–83.

[11] P. Caspi, D. Pilaud, N. Halbwachs, J. Plaice, Lustre: A declarative language for programming synchronous systems, in: POPL, ACM Press, 1987, pp. 178–188.

[12] A. Benveniste, P. L. Guernic, C. Jacquemot, Synchronous programming with events and relations: the signal language and its semantics, Sci. Comput. Program. 16 (2) (1991) 103–149.

[13] E. A. Lee, A. L. Sangiovanni-Vincentelli, A framework for comparing models of computation, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 17 (12) (1998) 1217–1229.

[14] A. Benveniste, B. Caillaud, L. P. Carloni, P. Caspi, A. L. Sangiovanni-Vincentelli, Composing heterogeneous reactive systems, ACM Transactions on Embedded Computing Systems 7 (4).

[15] B. Combemale, J. DeAntoni, M. V. Larsen, F. Mallet, O. Barais, B. Baudry, R. B. France, Reifying concurrency for executable metamodeling, in: Software Language Engineering - 6th Int. Conf., SLE, Vol. 8225 of Lecture Notes in Computer Science, Springer, 2013, pp. 365–384. doi:10.1007/978-3-319-02654-1_20.

[16] L. G. Valiant, M. Paterson, Deterministic one-counter automata, J. Comput. Syst. Sci. 10 (3) (1975) 340–350. doi:10.1016/S0022-0000(75)80005-5.
URL http://dx.doi.org/10.1016/S0022-0000(75)80005-5

[17] S. Bardin, A. Finkel, J. Leroux, L. Petrucci, Fast: Fast acceleration of symbolic transition systems, in: CAV, Vol. 2725 of Lecture Notes in Computer Science, Springer, 2003, pp. 118–121.

[18] A. Finkel, G. Sutre, Decidability of reachability problems for classes of two counters automata, in: H. Reichel, S. Tison (Eds.), STACS 2000, 17th Annual Symp. on Theoretical Aspects of Computer Science, Vol. 1770 of Lecture Notes in Computer Science, Springer, 2000, pp. 346–357.

[19] J. Hopcroft, J.-J. Pansiot, On the reachability problem for 5-dimensional vector addition systems, Theoretical Computer Science 8 (2) (1979) 135–159. doi:10.1016/0304-3975(79)90041-0.

[20] S. Bardin, A. Finkel, J. Leroux, Faster acceleration of counter automata in practice, in: K. Jensen, A. Podelski (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, 10th Int. Conf., TACAS 2004, Vol. 2988 of Lecture Notes in Computer Science, Springer, 2004, pp. 576–590. doi:10.1007/978-3-540-24730-2_42.

[21] R. Gascon, F. Mallet, J. DeAntoni, Logical time and temporal logics: Comparing UML MARTE/CCSL and PSL, in: C. Combi, M. Leucker,

F. Wolter (Eds.), TIME, IEEE, 2011, pp. 141–148. doi:10.1109/TIME.2011.10.

[22] L. Yin, F. Mallet, J. Liu, Verification of MARTE/CCSL time requirements in Promela/SPIN, in: I. Perseil, K. Breitman, R. Sterritt (Eds.), ICECCS, IEEE Computer Society, 2011, pp. 65–74.

[23] H. Yu, J.-P. Talpin, L. Besnard, T. Gautier, H. Marchand, P. Le Guernic, Polychronous controller synthesis from MARTE CCSL timing specifications, in: S. Singh, B. Jobstmann, M. Kishinevsky, J. Brandt (Eds.), MEMOCODE, IEEE, 2011, pp. 21–30.

[24] J. Suryadevara, C. C. Seceleanu, F. Mallet, P. Pettersson, Verifying marte/ccsl mode behaviors using uppaal, in: SEFM, Vol. 8137 of Lecture Notes in Computer Science, Springer, 2013, pp. 1–15.

[25] S. Bliudze, J. Sifakis, The algebra of connectors - structuring interaction in BIP, IEEE Trans. Computers 57 (10) (2008) 1315–1330. doi:10.1109/TC.2008.26.

[26] A. Arnold, Finite transition systems - semantics of communicating systems, Int. Series in Computer Science, Prentice Hall, 1994.

[27] T. Johnsson, Efficient compilation of lazy evaluation, in: SIGPLAN Symposium on Compiler Construction, ACM, 1984, pp. 58–69.

[28] P. H. Feiler, J. Hansson, Flow latency analysis with the architecture analysis and design language, Tech. Rep. CMU/SEI-2007-TN-010, CMU (June 2007).

[29] S. of Automotive Engineers, SAE Architecture Analysis and Design Language (AADL), document number: AS5506/1 (June 2006).
URL http://www.sae.org/technical/standards/AS5506/1