Time: It is only Logical!

Frédéric Mallet(✉)

Université Côte d'Azur, CNRS, Inria, I3S, Nice, France
Frederic.Mallet@univ-cotedazur.fr

Abstract. Logical Clocks play an important role for the design and modelling of concurrent systems. The Clock Constraint Specification Language (CCSL) was built in 2009, as part of an annex of the UML Profile for MARTE, to give a proper syntax to handle logical clocks as first class citizens. The syntax gave rise to a series of different semantic interpretations along with various verification tools. Usecases are diverse and include languages to express timing requirements, temporal or spatio-temporal logics to capture expected safety properties, meta-languages to give an operational semantics to domain-specific languages. The application domains include avionics, safety-critical transportation systems, self-driving vehicles, systems engineering models, cyber-physical systems. This paper reviews the effort conducted since 2009 on CCSL. A large part of this effort was made possible by Professor He Jifeng and his will to build in Shanghai a research centre of excellence for trustworthy systems. Researchers there found inspiration in the heritage left by the different schools working around the world on concurrency theory, including the school of synchronous languages from which CCSL has emerged.

Keywords: Logical Time · Cyber-Physical Systems · Polychronous languages

1 Introduction

1.1 CCSL - Genesis

The Clock Constraint Specification Language (CCSL) was devised as an attempt to bring order into the galaxy of so-called standard notations and semantics that were emerging [37] following the adoption of the Unified Modelling Language 2.x [52]. Some complained that the official semantics was not precise enough [18], others that it was too constraining and not expressive enough. Each community working in the field of concurrency theory or formal languages was providing its contribution to give one interpretation, connected to its analysis or verification frameworks. The community of synchronous languages [7] was no exception and has provided its own contributions with Argos [40] and SyncCharts [2,4], as synchronous interpretations for the visual formalism of Harel's StateCharts [22] or also as a formal and sound alternative to UML State Machines.

The word *unified* was misleading as some people were trying to provide a unique, one size fits all, notation to do everything as the goal should have been to

J. P. Bowen et al. (Eds.): *Theories of Programming and Formal Methods*, LNCS 14080, pp. 323–347, 2023.
https://doi.org/10.1007/978-3-031-40436-8_12

provide a *unifying* framework to compare the legitimate different interpretations that are necessary to deal with the diversity of missions faced by software or system engineers. In his book on Unifying Theories of Programming [25] with C.A.R. Hoare, Professor He Jifeng gets us back on another path by stating *"A unifying theory is usually complementary to the theories that it links, and does not seek to replace them."*

CCSL was meant to give a complementary notation that would come as a companion of languages, whether visual or not, to clarify or make precise, if and when necessary, their interpretation and in particular, the subtle legitimate behavioural variations regarding temporal, timed or concurrent aspects of systems. Not a language to rule them all but rather a meta-language to allow different semantic interpretations to co-exist without ambiguities.

As UML semantic variation points were meant to be addressed in dedicated profiles, it was only natural to seek the definition of a profile for that purpose. Then what started as an attempt to build a synchronous reactive UML profile [49], soon became a participation of the Aoste I3S/Inria team to a Task Force within the Object Management Group. Our goal at that time, was to define something that would allow a synchronous reactive interpretation to co-exist with plenty other interpretations, including fully asynchronous ones. The UML Profile for Modelling and Analysis of Real-Time systems (MARTE) was adopted three years later in 2009 [53] and CCSL was described within annex C.3 of the specification.

1.2 Logical Time and Clocks

As the name suggests, logical clocks are the central and foundational element of CCSL. The word *clock* is also misleading as it has a deeply anchored popular meaning that turns out to characterize just a particular, although very important, kind of clock. A clock is *a device for measuring and showing time*. It usually works by comparing the duration of a phenomenon by counting the number of occurrences of ticks produced by a trustworthy source taken as a time reference. When the source is based on a regular physical phenomenon, for instance the resonant frequency of atoms in the case of atomics clocks, it is called a *physical clock*. However, the reference can also be an arbitrary recurrent event that is meaningful to the system for some reasons. In such cases, it is referred to as a *logical clock*.

Several digital systems react according to the speed of driving clocks that are not regular periodic physical events. A control system of a car engine reacts according to the rotation speed of the camshaft, this rotation speed changes continuously and can hardly refer to an absolute physical measure as 1) it would depend on too many external parameters; 2) it is neither necessary nor efficient to build the actual physical device.

In the origin of Communicating Sequential Processes (CSP [24]), we also find an interesting statement to support that position *"Another detail which we have deliberately chosen to ignore is the exact timing of occurrences of events. The advantage of this is that designs and reasoning about them are simplified, and*

furthermore can be applied to physical and computing systems of any speed and performance.".

Leslie Lamport [30] has made popular logical clocks in the context of distributed systems by considering that *"the concept of time is derived from the more basic concept of the order in which events occur"*. Logical clocks were used, as a pragmatic solution, to reconstruct a total order of events and therefore provide a simple method for synchronizing spatially-separated processes or even physical clocks. Further away from those practical considerations, event structures [43] provide a theoretical framework to study the relation *happens before* at the heart of Lamport's logical clocks.

In the field of programming languages, and mainly for reactive systems, synchronous languages [7], like Esterel [8], Lustre [11] or Signal [32], have long promoted logical clocks as native programming artefacts. Clocks are used as activation conditions to decide when it makes sense to activate the different parts of a program so as to make sure that they operate correctly, for instance because all the necessary inputs are available. Synchronous (logical) clocks rely on the concept of instant, that denotes atomic actions, and allows for deciding whether some occurrences of events happen instantaneously, i.e., within the same instant. This leads to the notion of *happens together* or coincidence. Even though coincidence is a mental construct, it proved to be useful for the design of reactive and/or safety-critical systems [13].

Professor He has also proposed his own view of a clock model suitable for the construction of hybrid systems [23]. In his work, clocks are increasing sequences of non-negative reals. Those clocks refer directly to synchronous signals and the real values carried by clocks are the dates at which events occur. By considering sequences of reals, it implicitly assumes a global common time base. Other frameworks, like polychrony [33], do not assume the existence of a common global clock and rather push for solutions where clocks are not inherently related to each other. However, the potential existence of a common clock, may still become a good property that has to be proven or disproven by the compiler.

As an attempt to unify theories of time structures, tagged systems [34] and then tag machines [6] have become mainstream theoretical and practical (within the scope of Ptolemy [17] and variants like ForSyDe [47] and ModHel'X [21]) solutions to compare and combine models of computations (and communications).

While tag machines provide a nice mathematical framework, they do not provide any concrete syntax to build tag structures and define relations among them. CCSL intended to do that by focusing only on the underlying orderings among events, leaving out the tags themselves. It combines the two notions of *happens before* and *happens together*. An extension of CCSL, called TESL [55], brings back the tags and define some operators that build clocks depending on the tags or derives tags based on clock relations.

Finally, one must note that logical clocks of CCSL strongly differ from the (dense) clocks of timed automata [1]. Timed automata rely on a dense time model, meaning that clocks take values in a dense set. This is very useful and

sometimes more natural for physical processes operating over continuous time. All these dense clocks increase at a uniform rate counting time with respect to a common global time frame. In certain conditions, the clocks can be stopped or reset. Timed automata, and their numerous derivatives, have given rise to a variety of powerful and very successful tools, like UPPAAL [31]. We show in Sect. 3 that we can combine such models with CCSL ones to benefit from both environments when one needs to access both logical and physical clocks.

The initial denotational semantics of CCSL [3] considered a model with dense-time but most CCSL-based tools [15] only work for discrete time and CCSL relies on timed automata [50] whenever it has to deal with dense-time relations.

1.3 Outline

This paper starts with a brief introduction to the syntax and semantics of CCSL in Sect. 2. Then, Sect. 3 describes two main use cases where CCSL is used not standalone, but as companion to other formalisms and notations. Section 4 describes some of the variants of CCSL. Then we briefly conclude.

2 Syntax and Semantics

A comprehensive theory of programming [25] treats a programming language under three styles of presentations: denotational, operational and algebraic. Here we do not go as far as proving consistency between the three definitions but we give a grasp of what it means in the context of CCSL.

We follow here the same path as Professor He. We start with the denotational semantics, then the operational one, and we end with a glimpse at the co-algebraic semantics.

2.1 Clocks, Schedules and History

Definition 1 (Logical clock). *A logical clock c is an infinite sequence (a stream) of ticks, $(c_n)_{n\in\mathbb{N}^+}$.*

While a logical clock can represent any kind of repetitive event, the ticks stand for their successive occurrences. All the events are assumed to be independent, so there is no relationship between the ticks of two clocks unless explicitly defined. Concretely, clocks can be used to observe the occurrence of events. In such cases, CCSL describes the expected observations. They can also be used as activation conditions to control the behaviour of a system.

CCSL constraints express some relationships between clocks, and their underlying ticks. One possible behaviour is captured as a synchronous schedule defined as an infinite sequence of steps. At each step, the schedule defines which clocks tick and which ones do not tick. A CCSL specification characterizes a set of valid schedules. Each constraint potentially reduces the number of valid schedules by forbidding some clocks to tick at some steps.

Definition 2 (Schedule). *Given a set C of clocks, a schedule of C is a total function $\delta : \mathbb{N} \to 2^C$ such that at each step n in $\mathbb{N}$, $\delta(n) \neq \emptyset$.*[1]

By the condition $\delta(n) \neq \emptyset$ in Definition 2 we exclude from schedules those *trivial*/stuttering steps where there is no clock ticking. As we deal with reactive systems, we expect the system not to stop, and therefore to have clocks that tick in infinitely many steps. As we show later, clocks that stop very often indicate a bad (or at least unexpected) behaviour of the system under consideration. Having this in mind, we thrive to build *good* schedules that have this property.

For a given schedule it may be useful to identify the step at which the i^{th} tick occurred.

Definition 3 (Dates and time). *Given a schedule δ for a set of clocks C, $dates^\delta : C \to 2^{\mathbb{N}}$ is a map defined as $\forall c \in C, dates^\delta(c) = \{i \in \mathbb{N} | c \in \delta(i)\}$.*

Then, $time^\delta$ is a map $time^\delta : C \times \mathbb{N}^+ \to \mathbb{N}$ defined as $\forall c \in C, \forall i \in \mathbb{N}^+, time^\delta(c, i) = j$ such that $|\{k \in dates^\delta(c) | k \leq j\}| = i$.

$dates^\delta$ gives the set of steps where a clock ticks for a given schedule δ, while $time^\delta$ gives the step at which the i^{th} tick of a given clock occurs, for a given schedule δ. If clocks tick infinitely many times, as they should, $dates^\delta$ is an infinite subset of natural numbers.

Purely synchronous constraints define when some clocks should tick together and when they cannot, i.e. synchronization conditions. Other more general constraints look at the past, the *history* (as far as they need) to decide what may happen at a given step.

Definition 4 (History). *The history of a schedule δ over a set C of clocks is a function $\chi^\delta : C \times \mathbb{N} \to \mathbb{N}$ such that for each clock $c \in C$ and $n \in \mathbb{N}$:*

$$\chi^\delta(c, n) = \begin{cases} 0 & \text{if } n = 0 \\ \chi^\delta(c, n-1) & \text{if } n > 0 \wedge c \notin \delta(n-1) \\ \chi^\delta(c, n-1) + 1 & \text{if } n > 0 \wedge c \in \delta(n-1) \end{cases}$$

Intuitively, $\chi^\delta(c, n)$ denotes the number of times that a clock c has ticked before reaching step n in the schedule δ. For simplicity, we write χ for χ^δ when the context is clear. The history computes the configuration for a given clock. This ability to look into the past as far as we need raises reachability problems unusual in traditional synchronous languages, which commonly look only at the preceding step.

The way history is built gives a natural carrier for the co-algebraic definition given in Sect. 2.5.

2.2 Syntax

The initial syntax of CCSL was defined in a research report [3]. It was defined under the form of a mathematical language. As CCSL became integrated into

[1] 2^C is the powerset of C.

programming environments, the syntax was modified to resemble more that of a programming language and be more tractable by standard text-based editors. TimeSquare [15] is the official tool to build and analyse CCSL specifications. The syntax in TimeSquare is meant to be integrated into modelling environments that stores artefacts as XML resources. A lighter syntax, called Light-CCSL[2], has then been defined to be more user-friendly. We use both the pure mathematical syntax and the light CCSL one here.

CCSL provides a set of binary or ternary **clock relations** that constrain the instants at which a clock can tick. When there is no constraint, all the schedules are possible. Each constraint reduces the set of possible schedules. For most specifications, an infinite number of schedules are valid. When only one schedule is possible, the system is fully determined. If no schedule is possible, the specification is inconsistent.

The two basic synchronous relations are *subclocking* ($c_1 \subseteq c_2$) and *exclusion* ($c_1 \# c_2$). *subclocking* is a relation that only allows c_1 to tick when c_2 ticks: $\forall s \in \mathbb{N}^+, c_1 \in \delta(s) \implies c_2 \in \delta(s)$. We get immediately that when $c_1 \subseteq c_2 \wedge c_2 \subseteq c_1$ then $dates(c_1) = dates(c_2)$, c_1 and c_2 are called synchronous ($c_1 \boxed{=} c_2$).[3]

Exclusion forbids c_1 and c_2 to tick at the same step: $\forall s \in \mathbb{N}^+, c_1 \notin \delta(s) \vee c_2 \notin \delta(s)$.

The Light-CCSL listing below defines two subclocking and one exclusion constraints, $c1 \subseteq c2 \wedge c2 \subseteq c3 \wedge c4 \# c3$.

```
Specification example1 {
  Clock c1 c2 c3 c4 [
    SubClocking c1 ← c2 ← c3
    Exclusion c4 # c3
  ]
}
```

The two basic asynchronous relations are *causality* ($c_1 \preccurlyeq c_2$) and *precedence* ($c_1 \prec c_2$). *Causality* is the *happen before* relationship of event structures. It means that $\forall d \in \mathbb{N}^+, time(c_1, d) \leq time(c_2, d)$, the d^{th} occurrence of c_1 cannot be after the d^{th} occurrence of c_2. *Precedence* is a bit stricter, it means that $\forall d \in \mathbb{N}^+, time(c_1, d) < time(c_2, d)$.

The Light-CCSL listing below defines precedences and causalities, $c1 \prec c2 \wedge c2 \preccurlyeq c3 \wedge c3 \prec c4$.

```
Specification example2 {
  Clock c1 c2 c3 c4 [ Precedence c1 < c2 <= c3 < c4 ]
}
```

While CCSL relations reduce the set of valid schedules, CCSL expressions build new clocks that preserve some relations by construction. Some expressions build concrete subclocks, like *union* and *intersection*.

[2] https://github.com/frederic-mallet/ccsl-sts/tree/main/Examples/CCSL_Primitives.

[3] The boxed equality ($\boxed{=}$) is there not to confuse clocks that are equal from clocks that tick synchronously.

$u \triangleq c_1 + c_2$ (union of c_1 and c_2) builds a clock u such that $dates(u) = dates(c_1) \cup dates(c_2)$. We get immediately that $c_1 \subseteq c_1 + c_2$ and $c_2 \subseteq c_1 + c_2$.
$i \triangleq c_1 * c_2$ (intersection of c_1 and c_2) builds a clock i such that $dates(i) = dates(c_1) \cap dates(c_2)$. We get immediately that $c_1 * c_2 \subseteq c_1$ and $c_1 * c_2 \subseteq c_2$.

Another way to build a new clock is to use affine functions. $c_1 \triangleq c_2 \propto p$ makes c_1 tick every p^{th} tick of c_2. $c_3 \triangleq c_1 \$ d$ makes c_3 tick synchronously with c_1 after its d^{th} tick. In Light-CCSL, on would write the following specification:

```
Specification Period {
  Clock c2 [
    repeat c1 every 3 c2
    Let c3 be c1 $ 2
  ]
}
```

From this listing we obtain the schedule shown in Fig. 1 as the only possible valid schedule since this specification is fully determined. In this schedule, $dates(c2) = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ as if $c2$ does not tick, none of the other clocks can tick. $dates(c1) = \{0, 3, 6, 9\}$ and $dates(c3) = \{6, 9\}$. Besides, $time(c2, 1) = 0$, $time(c1, 2) = 3$ and $time(c3, 1) = 6$.

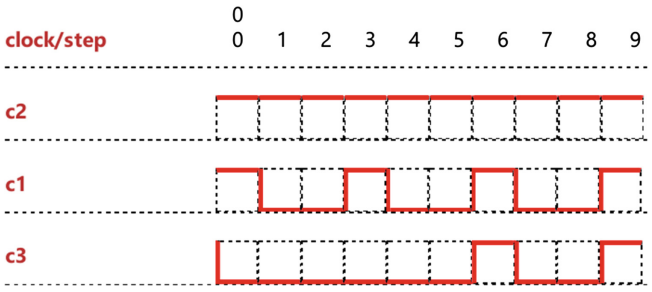

Fig. 1. A schedule with delays and periodic clocks.

Other expressions build new clocks that preserve causalities. $inf \triangleq c_1 \wedge c_2$ (infimum of c_1 and c_2) builds a clock inf such that $\forall d \in \mathbb{N}^+, time(inf, d) = min(time(c_1, d), time(c_2, d))$. We get immediately that $c_1 \preccurlyeq c_1 \wedge c_2$ and $c_2 \preccurlyeq c_1 \wedge c_2$.
$sup \triangleq c_1 \vee c_2$ (supremum of c_1 and c_2) builds a clock sup such that $\forall d \in \mathbb{N}^+, time(sup, d) = max(time(c_1, d), time(c_2, d))$. We get immediately that $c_1 \vee c_2 \preccurlyeq c_1$ and $c_1 \vee c_2 \preccurlyeq c_2$.

```
Specification Expressions {
  Clock a b c [
    Let i be inf(a,b,c)
    Let s be sup(a,b,c)
    Let union be a or b
    Let inter be a and b
```

]
}

2.3 Denotational Semantics

CCSL may serve different purposes. One main objective is to verify that a specification is consistent. This is for instance useful when using CCSL to build requirements. Informal or natural-language requirements are prone to errors. To check the consistency of requirements, we transform them into CCSL constraints and then we try to find at least one valid schedule for the derived specification [12]. TimeSquare [15] allows for making those transformations automatic by giving generic transformation rules from model elements and applying those transformation rules in a systematic way on a complete model (see Sect. 3.2). Checking the satisfaction of CCSL specifications has been done by many different methods [15,57,58] including through the use of an SMT-solver [58].

The semantics of CCSL given in Table 1 is interesting for that task as the encoding into an SMT solver is almost immediate. A schedule is defined as an undefined function[4], or rather as a set of undefined functions, one for each clock. Those undefined functions must satisfy all of the constraints in a specification. The SMT solver will then find a valid definition of those functions that satisfy all the constraints. If it manages to do so, that gives us immediately one valid schedule.

Table 1. Semantics of CCSL

1.	$\delta \vDash_{ccsl} a \subseteq b$	iff	$\forall i \in \mathbb{N}. a \in \delta(i) \rightarrow b \in \delta(i)$	(Subclock)
2.	$\delta \vDash_{ccsl} a \# b$	iff	$\forall i \in \mathbb{N}. a \notin \delta(i) \vee b \notin \delta(i)$	(Exclusion)
3.	$\delta \vDash_{ccsl} a \prec b$	iff	$\forall i \in \mathbb{N}. (\chi^{\delta}(a,i) > \chi^{\delta}(b,i) \vee (\chi^{\delta}(a,i) = \chi^{\delta}(b,i) \rightarrow b \notin \delta(i))$	(Precedence)
4.	$\sigma \vDash_{ccsl} a \preccurlyeq b$	iff	$\forall i \in \mathbb{N}. \chi^{\delta}(a,i) \geq \chi^{\delta}(b,i)$	(Causality)
5.	$\delta \vDash_{ccsl} c \triangleq a + b$	iff	$\forall i \in \mathbb{N}. c \in \delta(i) \leftrightarrow (a \in \delta(i) \vee b \in \delta(i))$	(Union)
6.	$\delta \vDash_{ccsl} c \triangleq a * b$	iff	$\forall i \in \mathbb{N}. c \in \delta(i) \leftrightarrow (a \in \delta(i) \wedge b \in \delta(i))$	(Intersection)
7.	$\delta \vDash_{ccsl} c \triangleq c' \propto n$	iff	$\forall i \in \mathbb{N}. c \in \delta(i) \leftrightarrow (c' \in \delta(i) \wedge \exists m \in \mathbb{N}^{+}. \chi^{\delta}(c',i) = m \cdot (n+1))$	(Periodicity)
8.	$\delta \vDash_{ccsl} c \triangleq c' \$ n$	iff	$\forall i \in \mathbb{N}. \chi^{\delta}(c,i) = max(\chi^{\delta}(c',i) - n, 0)$	(Delay)
9.	$\delta \vDash_{ccsl} c \triangleq a \wedge b$	iff	$\forall i \in \mathbb{N}. \chi^{\delta}(c,i) = max(\chi^{\delta}(a,i), \chi^{\delta}(b,i))$	(Infimum)
10.	$\delta \vDash_{ccsl} c \triangleq a \vee b$	iff	$\forall i \in \mathbb{N}. \chi^{\delta}(c,i) = min(\chi^{\delta}(a,i), \chi^{\delta}(b,i))$	(Supremum)

Pure synchronous constraints (Table 1, rules 1, 2, 5, 6) do not involve the history. They are called stateless constraints and result in solving a pure Boolean satisfaction problem. Other constraints rely on the history (see Definition 4) that relies on integer arithmetic. Table 1-7 is the most difficult of them all at it uses an existential quantifier. Actually, we can remove this quantifier by unfolding the formula based on the length of p. This is easy, or at least systematic. However, when p is big, this results in a highly inefficient system. Overall, when you

[4] In SMT, we would rely on the theory called UF_LIA, Undefined functions, an extension with free sorts and function symbols, combined with Linear Integer Arithmetic. The signature of those functions matches the one given in Definition 2.

combine Boolean logics, with integer arithmetic and undefined functions, there is no guarantee of having a result as the theories that are used are undecidable. However, in most practical cases we have encountered so far, SMT solvers do reach a verdict. Nevertheless, there have been many attempts over the last decade to improve the performances of CCSL solvers but there is no definitive answer to this problem at the moment.

On a pure denotational way, we can consider that a *Clock* is a pair $\langle \mathcal{I}, \prec \rangle$ where $\mathcal{I}$ is a set of instants, $\prec$ is a quasi-order relation on $\mathcal{I}$, named *strict precedence*, it is a total, irreflexive, and transitive binary relation on $\mathcal{I}$.

A *discrete-time clock* is a clock with a discrete set of instants $\mathcal{I}$. Since $\mathcal{I}$ is discrete, it can be indexed by natural numbers in a fashion that respects the ordering on $\mathcal{I}$: $\text{idx} : \mathcal{I} \to \mathbb{N}^+$, $\forall i \in \mathcal{I}$, $\text{idx}(i) = k$ if and only if i is the k^{th} instant in $\mathcal{I}$.

For any discrete time clock $c = \langle \mathcal{I}_c, \prec_c \rangle$, $c[k]$ denotes the k^{th} instant in $\mathcal{I}_c$ (*i.e.*, $k = \text{idx}_c(c[k])$). For any instant $i \in \mathcal{I}_c$ of a discrete time clock, $^\circ i$ is the unique immediate predecessor of i in $\mathcal{I}_c$. For simplicity, we assume the existence of a virtual instant, which is the (virtual) immediate predecessor of the first instant.

A *Time Structure* is a pair $\langle C, \preccurlyeq \rangle$ where C is a set of clocks, $\preccurlyeq$ is a binary relation on $\bigcup_{c \in C} \mathcal{I}_c$, named *causality*. $\preccurlyeq$ is reflexive and transitive. From $\preccurlyeq$ we derive two new relations: *Coincidence* ($\equiv \triangleq \preccurlyeq \cap \succcurlyeq$), *Precedence* ($\prec \triangleq \preccurlyeq \backslash \equiv$).

Then, given two clocks a and b, we can define the basic clock relations as follows.

Definition 5 (Subclocking). *a is said to be a sub-clock of b, and b a super-clock of a, denoted as $a \subseteq b$.*

$$\langle C, \preccurlyeq \rangle \models a \subseteq b \Leftrightarrow \forall i_a \in \mathcal{I}_a,\ \exists i_b \in \mathcal{I}_b,\ i_a \preccurlyeq i_b$$

Figure 2 gives an example of valid schedule for $a \subseteq b$, but there are infinitely many valid schedules.

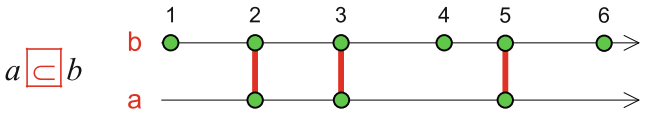

Fig. 2. Example of subclocking.

Note that this definition does not require the clocks to be discrete. Other CCSL relational operators are similar, see [3] for a comprehensive definition.

In a recent work [41], the semantics of CCSL has been mechanized in Agda. This dramatically improves the confidence we may have in reasoning with CCSL specifications. However, even though Agda gives some assistance to make proofs, it still needs some human interventions. One very interesting feature that was

introduced with the help of Agda was the notion of refinement of CCSL ticks [42]. This refinement is akin to the notion of instantaneous causality that is well-known in synchronous languages [7].

2.4 Operational Semantics

In TimeSquare [15] the operational semantics gives a way to compute one possible valid schedule for a given CCSL specification. This works by iterating over two phases. The first phase consists in deciding what subset of clocks (called a configuration) is fireable instantaneously. In CCSL, this can be done by solving a pure SAT problem. The second phase consists in picking one fireable configuration, firing it and rewriting the system to update the history of each clock that has ticked.

If rather than a unique valid schedule, one wants to build a symbolic representation of all the valid schedules, this can be done by synchronous transition system where the (infinitely many) states represent the history of clocks and the transitions are labelled by a set of clocks, the ones that can fired depending on the history. This transition system captures all the fireable clocks, selecting one transition follows only one of the (possibly infinite number of) paths. In practice, we use one transition system for each constraint and we build the synchronous composition of all the transition systems needed for each constraint in a given specification. As we may have an infinite number of states, we sometimes try to fold the transition system to retain only so-called *periodic schedules* [57]. The folding consists in keeping a finite number of states, equivalent, up to a particular equivalence relation, to (infinitely many) other states.

Definition 6 (cLTS). *A* Clock-Labelled 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.*

Pure synchronous constraints are represented by cLTS with only one state as the set of fireable clocks does not depend on the history. Figure 3(a) shows the cLTS for encoding $a \boxed{=} b$. Either a and b tick together, or neither of them can tick. Subclocking (see Fig. 3(b)) is weaker as b can also tick alone, but not a.

Other (stateful) constraints are represented with infinite-state transition systems. For instance, Fig. 4 gives the cLTS for the precedence ($a \prec b$). The state records the difference in the number of ticks between a and b (see Table 1, rule 3.) In the state, as both a and b have ticked as many times, we have $\chi^\delta(a, i) = \chi^\delta(b, i)$, and therefore b cannot tick. In other states, a and b can tick alone, can tick jointly or neither of them can tick. The state is updated accordingly.

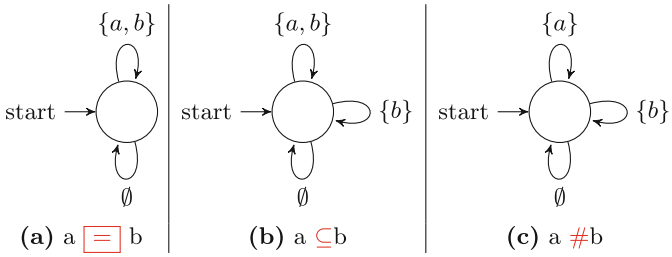

Fig. 3. CCSL synchronous relations as clock-Labelled Transition Systems

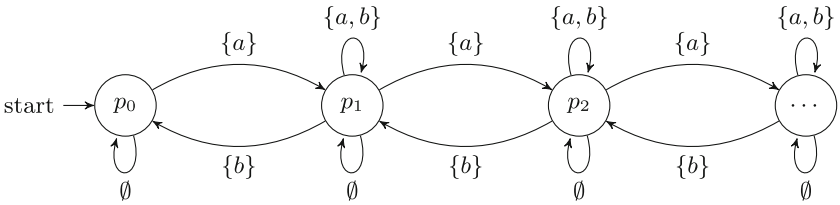

Fig. 4. CCSL precedence (infinite-state cLTS): $a \prec b$.

As there are an infinite number of states for some relations (like precedence), the set of potential execution is only intentional. *Safe* CCSL specifications [38] are the ones where only a finite number of states are actually reachable. We have established a sufficient condition for deciding whether or not a given CCSL specification is safe.

In a pure operational way, once the synchronous product of all the transition systems of all the CCSL constraints inside a specification has been computed (in intention or in extension), one can pick one path in this transition system to have a valid schedule. To get all the valid schedules, one must compute all the paths up to a given depth, depending on the length of the expected solution.

2.5 Coalgebraic Semantics

The theory of universal coalgebra [46] proposes a mathematical model, that differs from the approach of G. Plotkin for defining the operational semantics of software systems [45]. Indeed, considering transition systems as coalgebras gives useful insights for reactive systems and infinite data structures in general. Coalgebras appeared to be well fitted to capture the infinite-state transition systems underlying the semantics of some CCSL operators. We have used this style to define a notion of incompleteness for CCSL and then a possible generalized constraint model [39,60].

Definition 7 (Transition system). *A transition system is a structure* $\langle \Gamma, \longrightarrow \rangle$ *where* Γ *is a set (of elements,* γ*, called configurations) and* $\longrightarrow \subset \Gamma \times \Gamma$ *is a binary relation (called the transition relation). Read* $\gamma \longrightarrow \gamma'$ *as saying that there is a transition from configuration* γ *to configuration* γ'*.*

Using the notion of coalgebra we obtain an alternative way to describe transition systems.

Definition 8 (Coalgebra). *A (powerset) coalgebra* [46] *is a structure* $\langle \Gamma, \alpha \rangle$ *where* α *is a map from* Γ *into the set of all subsets of* Γ*,* 2^{Γ}*. In this context* Γ *is called the carrier of the coalgebra.*

It is evident that any transition system $\langle \Gamma, \longrightarrow \rangle$ determines the coalgebra $\langle \Gamma, \alpha \rangle$, where $\gamma' \in \alpha(\gamma)$ if and only if $\gamma \longrightarrow \gamma'$, and conversely, any coalgebra $\langle \Gamma, \alpha \rangle$ determines the transition system $\langle \Gamma, \longrightarrow \rangle$, where $\gamma \longrightarrow \gamma'$ if and only if $\gamma' \in \alpha(\gamma)$.

Definition 9 (Subcoalgebra). *Let* $\langle \Gamma, \alpha \rangle$ *be a coalgebra,* B *be a subset of* Γ *then the structure* $\langle B, \alpha \rangle$ *is called a subcoalgebra of* $\langle \Gamma, \alpha \rangle$ *if the embedding* $\alpha(\gamma) \subset B$ *is true for each* $\gamma \in B$*.*

One can check that any coalgebra $\langle \Gamma, \alpha \rangle$ is a subcoalgebra of itself and the intersection of a family of subcoalgebras is a subcoalgebra too. Hence, for each subset $X \subset \Gamma$ there exists a least one subcoalgebra whose carrier contains X. In this case, the carrier of this subcoalgebra is denoted by $\langle X \rangle$.

To calculate $\langle X \rangle$ one can use Tarski's fixed point theorem [51] for the monotonic operator Ψ_X on the lattice $\mathcal{P}_X(\Gamma)$, where $\mathcal{P}_X(\Gamma)$ is the set of all Γ subsets that cover X. This operator is defined by the following formula

$$\Psi_X(V) = V \cup \{\gamma' \in \Gamma \mid (\exists \gamma \in V)\ \gamma' \in \alpha(\gamma)\}.$$

This ensures that an element $\gamma \in \Gamma$ belongs to $\langle X \rangle$ if and only if there exists a finite sequence $\gamma_0, \ldots, \gamma_{n-1}, \gamma_n$ formed by elements of Γ such that

$$\gamma_0 \in X \text{ and } \gamma_n = \gamma\,; \tag{1}$$

$$\gamma_k \in \alpha(\gamma_{k-1}) \text{ for } k = 1, \ldots, n. \tag{2}$$

Finite or infinite Γ-valued sequences satisfying (2) are used below, we give them the name "tracks".

Hence, conditions (1) and (2) mean that an element $\gamma \in \Gamma$ belongs to $\langle X \rangle$ if and only if there exists a track that links some element of X and γ.

We now assume that some finite set of clocks $\mathcal{C}$ has been given. Let us define the constraint-free coalgebra over a clock set $\mathcal{C}$ as the coalgebra with the carrier $\mathbb{N}^{\mathcal{C}}$ and the map $\alpha : \mathbb{N}^{\mathcal{C}} \to 2^{\mathbb{N}^{\mathcal{C}}}$ defined by the formula:

$$\boldsymbol{\chi}' \in \alpha(\boldsymbol{\chi}) \quad \text{if and only if} \quad 0 \leq \chi'_a - \chi_a \leq 1 \quad \text{for all} \quad a \in \mathcal{C}.$$

It is evident that for any $\boldsymbol{\chi} \in \mathbb{N}^{\mathcal{C}}$ the map α is represented in the form

$$\alpha(\boldsymbol{\chi}) = \boldsymbol{\chi} + \{0, 1\}^{\mathcal{C}}.$$

This statement makes it obvious that a clock can only tick once at each instant and that all the evolutions are possible when no constraint is specified.

Proposition 1. *Let* $\langle \boldsymbol{\chi}(t) \mid t \in \mathbb{N} \rangle$ *be a sequence of configurations then there exists a schedule* σ *such that* $\chi_a(t) = \chi_a^\sigma(t)$ *for all* $t \in \mathbb{N}$ *and* $a \in \mathcal{C}$ *if and only if this sequence is a track in the coalgebra* $\langle \mathbb{N}^{\mathcal{C}}, \alpha \rangle$ *such that* $\boldsymbol{\chi}(0) = \mathbf{0}$.

A track $\langle \boldsymbol{\chi}(t) \mid t \in \mathbb{N} \rangle$ is called *initial* if the condition $\boldsymbol{\chi}(0) = \mathbf{0}$ holds.

One way to capture the notion of schedule, which are a sequence of steps where clocks tick simultaneously is to specify a map $\triangle : \mathbb{N}^{\mathcal{C}} \to 2^{\{0,1\}^{\mathcal{C}}}$ such that $\mathbf{0} \in \triangle(\boldsymbol{\chi})$ for any $\boldsymbol{\chi} \in \mathbb{N}^{\mathcal{C}}$ and to define

$$\alpha_\triangle(\boldsymbol{\chi}) = \boldsymbol{\chi} + \triangle(\boldsymbol{\chi}).$$

The map denotes at each step the set of clocks that tick.

A map $\triangle : \mathbb{N}^{\mathcal{C}} \to 2^{\{0,1\}^{\mathcal{C}}}$ that satisfies the condition $\mathbf{0} \in \triangle(\boldsymbol{\chi})$ for any $\boldsymbol{\chi} \in \mathbb{N}^{\mathcal{C}}$ is called an *actuation distribution* on $\mathcal{C}$. The actuation distribution captures the set of sets of clocks that are allowed to tick simultaneously at one instant given a configuration.

Definition 10 (Actuation distribution). *Let* $\triangle : \mathbb{N}^{\mathcal{C}} \to 2^{\{0,1\}^{\mathcal{C}}}$ *be an actuation distribution and* $\langle \mathbb{N}^{\mathcal{C}}, \alpha_\triangle \rangle$ *be a coalgebra, where* $\alpha_\triangle(\boldsymbol{\chi}) = \boldsymbol{\chi} + \triangle(\boldsymbol{\chi})$*, then an element of* $\mathbb{N}^{\mathcal{C}}$ *is called* $\triangle$*-reachable configuration if it belongs to the carrier of the minimal subcoalgebra containing* $\mathbf{0}$.

Such a set of reachable configurations is denoted below by $R(\triangle)$.

Definition 11 (Clock coalgebra). *Let* $\triangle : \mathbb{N}^{\mathcal{C}} \to 2^{\{0,1\}^{\mathcal{C}}}$ *be an actuation distribution then the coalgebra* $\langle R(\triangle), \alpha_\triangle \rangle$ *is called the* clock coalgebra *associated with* $\triangle$.

Actuation distributions of some clock constraints do not depend on the current configuration, so we define stationary distribution to denote particular interesting kinds of constraints.

Definition 12 (Stationary distribution). *An actuation distribution* $\triangle : \mathbb{N}^{\mathcal{C}} \to 2^{\{0,1\}^{\mathcal{C}}}$ *is called stationary if the map* $\triangle$ *is a constant map.*

Some primitive clock constraints, such as subclocking, exclusion, union and intersection, represent stationary actuation distributions. Therefore the question whether any stationary actuation distribution is represented by a set of stationary primitive clock constraints is interesting.

We have proven that this is true for 2-clock systems, but that this is not true in general [39]. Therefore, CCSL is incomplete as it should allow to build any actuation distribution. A very interesting construct that cannot be built is the n-m exclusion pattern, where n tasks share m resources. The 2–1 exclusion pattern is native (c_1 # c_2), and the n-1 can be built by parallel composition of multiple 2–1 exclusions. The n-m would be useful to represent a concurrent access to m cores by n computing tasks.

For this observation, one can build a generalization of CCSL that is complete. This languages is called *GenCCSL* [60]. While the language is complete, there is no operational way at the moment to build a solution for GenCCSL.

3 CCSL - A Companion Language

CCSL was never meant to be a programming language but rather it was meant to be a specification language. So it is not meant to be used standalone but rather to allow for complementing other specification with expected (temporal and timed) properties of a system. Additionally, CCSL is a companion language so it is expected that the main (functional) part of the system under consideration is given by another language or notation (e.g., UML for instance, or a programming language).

3.1 A Companion to UML MARTE

As it was defined in an annex of UML MARTE, users are inclined to use UML first, as much as possible, to describe, for instance, components or behavioural models. Then, they should use MARTE stereotypes when the semantics of UML is ambiguous.[5] Finally, use CCSL as a last resort when necessary. The two main useful MARTE stereotypes for that purpose are « clock » and « NFPConstraint ». « clock » identifies a model artefact that must be interpreted as a clock. « NFPConstraint » marks a constraint to be considered as a CCSL specification and potentially interpreted by adequate tools.

To give a simple example of what a companion language is, let us consider the BIP (Behaviour, Interaction, Priority) framework [5]. BIP is a framework for modelling heterogeneous real-time components with a correct-by-construction methodology. In BIP components, there are three layers. The lower layer describes the *behaviour* as transition systems. BIP uses a particular form of timed automaton. The intermediate layer includes a set of connectors describing the interactions between the transitions of the behaviour. The upper layer is a set of priority rules describing scheduling policies for interactions.

Figure 5 shows a small BIP example. Components have ports. Triangles denote so-called incomplete interactions while bullets identify complete ones. The upper connector with *tick*1, *tick*2 and *tick*3 implements a *rendez-vous*, i.e., the three ports are synchronized. The lower connector is a *broadcast*. CCSL provides no mechanism to build components or transition systems. It relies on other languages for that. One could use UML components and UML state machines for that purpose. However, UML state machines provide no built-in mechanism for describing rendez-vous. Using « clock » one would transform a UML event into a clock. Then using CCSL, one could enforce the semantics of BIP interactions (see the right-hand part of Fig. 5).

There are a bunch of papers [9,20,28,44,50,56] that show examples on how to use UML, MARTE and CCSL together, among those some prefer to use SysML instead of UML. There is a large contribution from the Software Engineering Institute in Shanghai. More importantly, each of these works provides a specific

[5] In UML wording, stereotypes are annotations of model elements that change the semantics of this element.

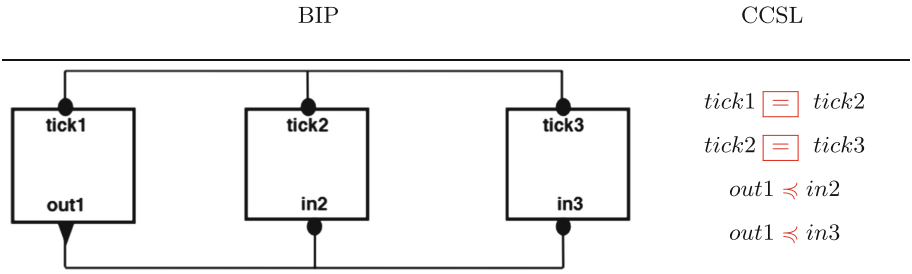

Fig. 5. Describing BIP interactions with CCSL

analysis tool to verify the temporal properties captured as a CCSL specification. These are either ad-hoc verification tools or transformations toward other mainstream verification languages (NuSMV, Timed Automata, VerilogHDL).

Figure 6 borrows an example of a temperature control system from [50] as an illustration. The temperature control system has two modes (*Diagnostic*, *Control*) depicted as a UML state machine. Moving from *Diagnostic* to *Control* is based on a time constraint. As UML does not have time units, we use MARTE to introduce them. In each mode, different constraints must hold to ensure the safety of the nuclear power plant. In *Diagnostic*, each diagnostic action (clock d) alternates with a reconfiguration action (clock c): $d \sim c$. A status update (clock s) is a particular kind of possible reconfiguration: $s \subseteq c$. Those constraints are captured in CCSL. To verify that the global model is consistent, a structural transformation based on the operational semantics (see Sect. 2.4) is performed to produce a timed automata (see right-hand side part of Fig. 6) that is fed into UPPAAL model-checker [31].

3.2 Semantic Adaptation of Domain-Specific Languages

While UML was the first language used as a support for CCSL specifications, it is not the only one. While UML has attempted a global union of lots of model elements, other approaches follow the *small is beautiful* mantra and advocate for the definition of small *Domain-Specification Languages* [19] just expressive enough for a given objective. Lots of dedicated modelling framework have emerged over the last two decades, the GeMoC studio [10] is one of them that was inspired by the international GeMoC initiative. Each language, or part of a language, is defined with its own abstract syntax and operational semantics. Then, to address large systems, several languages are composed to cover the different concerns (structure, states, data-flows, scenarios, properties). The languages are composed using a meta-language that derives from CCSL [29]. The approach is presented as a *unifying framework that reduces all structural composition operators to structural merging, and all composition operators acting on discrete behaviours to event scheduling*. This approach goes beyond what was discussed on UML as the *connectors* are defined between the two languages themselves and not on particular instances of those languages. Figure 7 shows an example

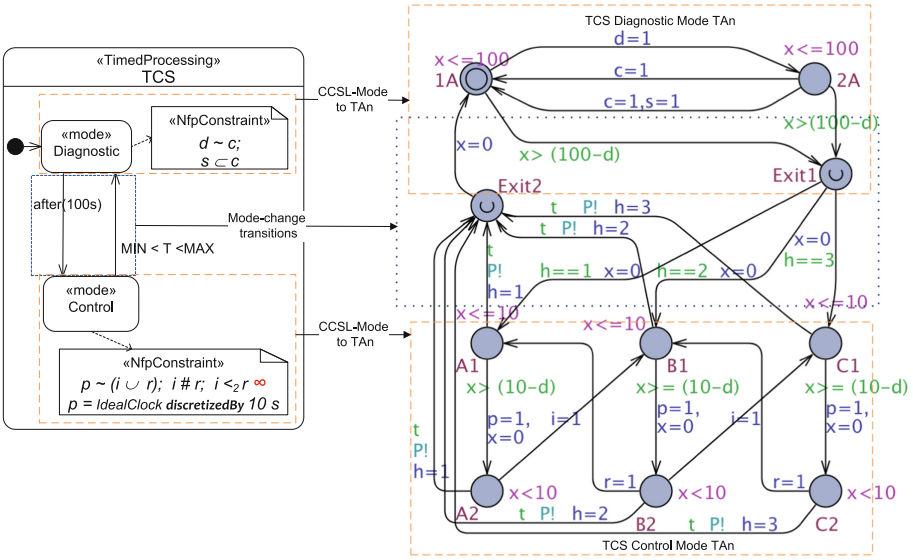

Fig. 6. A Temperature control system (TCS) with UML/MARTE/CCSL.

from that paper. We start with two languages (A and B). Language A involves some events a, b, c, while language B involves events 1, 2, 3, 4. The operational semantics of those two languages give execution rules. At the language level, we can build a constraint, say $a \preccurlyeq 1 \wedge b \preccurlyeq 4 \wedge 3 \preccurlyeq c$. Now, given two instances, one of A and one of B, we derive a set of possible traces for both models, a partial order, captured as event structures. Applying the composition rules (in black), we can reconstruct a global partial order that combines the traces from both languages (event structure es_c on the figure). Recently, this approach was applied to build a full-fledged simulator for Lingua Franca [14].

A similar exercise with a different tool/technology was done by another team [9] but still using CCSL to build a language for semantic adaptation.

4 CCSL Extensions and Derivatives

CCSL has led to several extensions or derivative languages that are briefly discussed in this section.

4.1 Valued Extensions

As CCSL was inspired by the Tagged Signal Model while removing the values of the tags and keeping only their orders, it was only natural to want to add the valued tags back into a language. In the Tagged-Event Specification Language (TESL) [54,55], clocks assign a time-stamp (aka a tag) to ticks with its own time scale. Tags represent the occurrence of the event at a specific

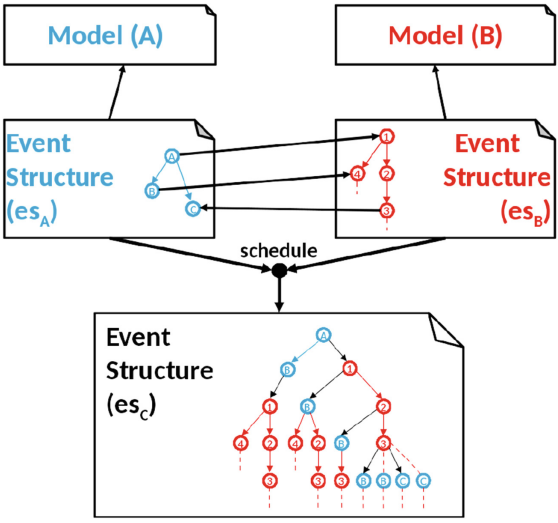

Fig. 7. DSL composition with clock relations.

time. Tag domains used for time must be totally ordered; typically, they are reals, rational numbers, integers, as well as the singleton Unit, which is used for purely logical clocks where (chronometric) time does not progress. TESL captures *event-triggered implications*, that are essentially CCSL-like clock constraints, time-triggered implications and *tag relations*. The time-triggered implication uses a chronometric delay (a reference to a physical time expression) to trigger an event. This delay is a duration (a difference between two tags) while in CCSL it would refer to a number of ticks or a difference between the number of ticks of two clocks. Tag relations link the different time scales. TESL allows for fairly general tag relations permitting acceleration and slow-down. Using affine tag relations makes the solving simpler as it amounts to handling linear equation systems. However, as the tags must be computed, TESL does not use any of the purely asynchronous constraints of CCSL (all the causality-based relations) as they do not allow for a constructive projection into the future and might lead to an infinity of possible futures.

Instead of schedules, TESL introduces so-called *runs*.

Definition 13 (Runs). *Given a set C of clocks, $\mathbb{B}$ the set of Booleans, $\mathbb{T}$ the ordered domain of timestamps. The set of runs is denoted Σ^{α} and defined by*

$$\Sigma^{\alpha} = \mathbb{N} \to C \to (\mathbb{B} \times \mathbb{T})$$

A (synchronous) run associates a pair to a step (a natural number) and a clock. The pair has a Boolean tag to identify whether the clock ticks or not and a timestamp that gives the current reading of the clock at this step. Compare to Definition 2.

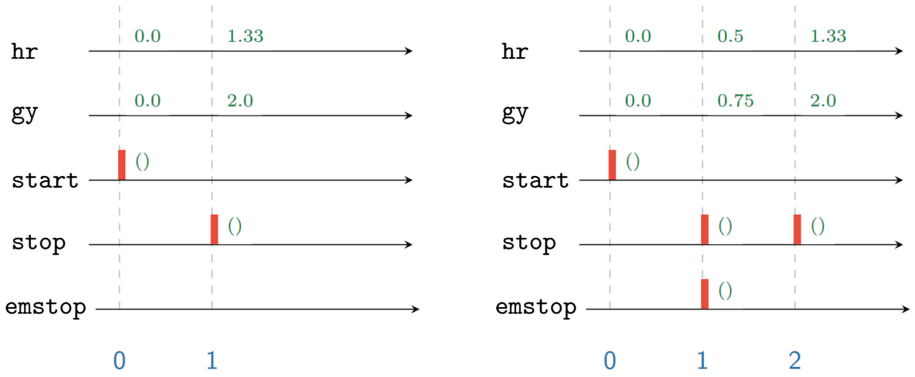

Fig. 8. Example of runs in TESL (Color figure online)

Figure 8 shows examples of runs taken from [55]. The black horizontal lines are temporal lines for clocks. The red rectangles denote ticks of clocks. The green annotations are timestamps, tags.

4.2 Other Extensions

There have been a variety of extensions, lots of them were proposed by teams at the Software Engineering Institute, following some work of Professor He.

In 2012, there was a first work [35] based on MARTE and CCSL to *"unify the logical time and the chronometric time variables, and extend the traditional events to CPS events"*. This work relied on Hybrid automata to introduce continuous evolutions of time and spatial constraints, as both constructs become necessary to model complex Cyber-Physical Systems (CPS).

This work was followed in two directions. On one side [48,59], trying to provide a spatio-temporal logics where both clocks and space are first-class citizens. On another side providing probabilistic extensions of CCSL.

For this second family of extensions, three alternative ways were studied. The first one used UML and profiling mechanisms as a support for extensions. This led to an hybrid form of MARTE state machines [36]. In that form, the systems was captured by combining UML, as much as possible, MARTE or stochastic stereotypes when needed and CCSL in the last resort. Verification was conducted by transforming the whole model into hybrid automata.

The two alternative solutions both considered extending CCSL with probabilistic parameters. This is necessary to capture the intrinsic uncertainty of complex environments for cyber-physical systems as for instance the temperature variation in a smart building, the likelihood of failure in an intelligent transport system.

Two such kinds of solutions were explored. The first solution called pCCSL [16], adds the notion of rate for the subclocking relation. When $a \subseteq b$, a may never tick, or always tick simultaneously with b. Both solutions, and all

the intermediate solutions are valid. The rate guarantees a probabilistic ratio between the number of ticks of a and the number of ticks of b. The second solution, called PrCCSL [27], adds instead a probability that a given clock relation is not satisfied. We present here pCCSL.

cLTS from Definition 6 are extended with a probability parameter as follows:

Definition 14 (Probabilistic CLTS). *A* Probabilistic Clock-Labelled Transition System *(PCLTS) is a CLTS with an extended transition relation* $\longrightarrow \subseteq S \times 2^C \times P \times S$*, where* $P \subseteq \mathbb{Q}$ *is the set of rational numbers between 0 and 1 (i.e., a probability).*

For a given transition $t = (s, \Gamma, p, s') \in \longrightarrow$, $\pi(t) = p$ denotes the probability p that the transition t is fired. It is akin of a discrete-time Markov chain, where the probability to reach the next state depends on the current state.

For a PCLTS $\langle S, C, \longrightarrow \rangle$, we call $s^\bullet$ the set of all transitions whose source is s:

$$s^\bullet = \{(s, \Gamma, p, s') \in \longrightarrow\}$$

Note that $s^\bullet$ can never be empty since it is always possible to do nothing in CCSL, i.e., $(s, \emptyset, p, s)$ is always in $\longrightarrow$ for all $s \in S$ and for some value p.

Given a clock $c \in C$, let us call $s^\bullet_c$ the set of all transitions whose source is s and such that the clock c ticks:

$$s^\bullet_c = \{(s, \Gamma, p, s') \in \longrightarrow | \ c \in \Gamma\}$$

For a PCLTS to be well-formed, it must satisfy the two following conditions:

$$\forall s \in S, \sum_{t \in s^\bullet} \pi(t) = 1 \tag{3}$$

$$\forall s \in S, \forall c \in C, \sum_{t \in s^\bullet_c} \pi(t) = p_c \tag{4}$$

In Eq. 4, for each clock $c \in C$, the probability p_c is either manually assigned by the user with a declaration 'Clock c probability p', or derived using the rate in a subclocking relation or assigned to the default value $1/|s^\bullet|$ otherwise.

A 'normal' CLTS can be seen as a probabilistic CLTS where all the probabilities are assigned with default values $1/|s^\bullet|$ for all the states $s \in S$.

Let $a, b \in C$ be two clocks and $r \in \mathbb{Q}$ a rational number such that $0 \leq r \leq 1$. The *subclocking* relation (see Fig. 9(a)), $b \subseteq a \,\mathsf{rate}\, r$ is defined as a PCLTS $\langle \{s_0\}, \{a, b\}, \longrightarrow_\subseteq \rangle$, such that $\longrightarrow_\subseteq = \{(s_0, \{\}, 1 - p_a, s_0), (s_0, \{a, b\}, p_a * r, s_0), (s_0, \{a\}, p_a * (1 - r), s_0)\}$, where $p_a \in \mathbb{Q}$ is the probability assigned to clock a. Let us note that Eq. 3 is satisfied since $\sum_{t \in s_0^\bullet} \pi(t) = (1 - p_a) + (p_a * r) + (p_a * (1 - r)) = 1$. Equation 4 is also satisfied since $\sum_{t \in s^\bullet_{0b}} \pi(t) = p_a * r = p_b$ and $\sum_{t \in s^\bullet_{0a}} \pi(t) = (p_a * r) + (p_a * (1 - r)) = p_a$.

If no probability is assigned then the default is 2/3. If no rate is assigned, then r defaults to 1/2. With default values, each one of the three transitions has a probability of 1/3, i.e., each transition has the same probability to be fired.

Transition $\{b\}$ however has a probability of 0 since it would otherwise contradict the subclocking relation.

Note that if both the probability of a is given and the rate of b relative to a are given, then $p_b = p_a * r$. In any other cases, the specification is ill-formed.

The synchrony constraint is a special case of subclock defined as follows $a = b \equiv b \subseteq a \,\mathsf{rate}\, 1$, which implies $p_a = p_b$.

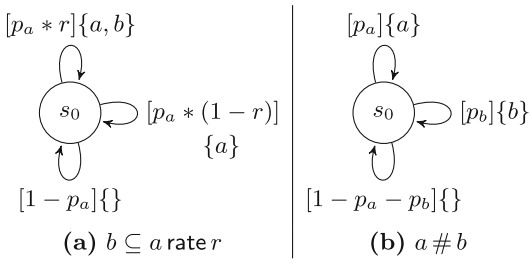

Fig. 9. PCLTS for subclocking and exclusion

Compare Fig. 9(a) to Fig. 3(a).

4.3 (Machine) Learning CCSL

A very recent work observes that using CCSL may be difficult for capturing requirements. This is especially true early in the design process when requirements are unclear. Indeed, using a formal language forces semantic choices. Early requirements must be flexible. While logical clocks allow some form of flexibility deciding which CCSL operator must be used may be a tough choice. What is usually easier for the designer is to give examples of expected or unexpected scenarios. The recent work [26] was trying to deduce a full CCSL specification from a set of scenarios/traces and from a partial specification.

The goal, and difficulty, is to find a specification that is as precise as possible while still satisfying all the constraints. To explore alternative specifications, we use reinforcement learning. We have a reward function that rewards *tight* specifications. Making one constraint too tight may result in a suboptimal solution as relaxing this constraint might allow to make another tighter. Figure 10 gives an overview of the proposed framework. Each layer explores alternative solutions for each hole. In the example, we have four holes, hence four layers.

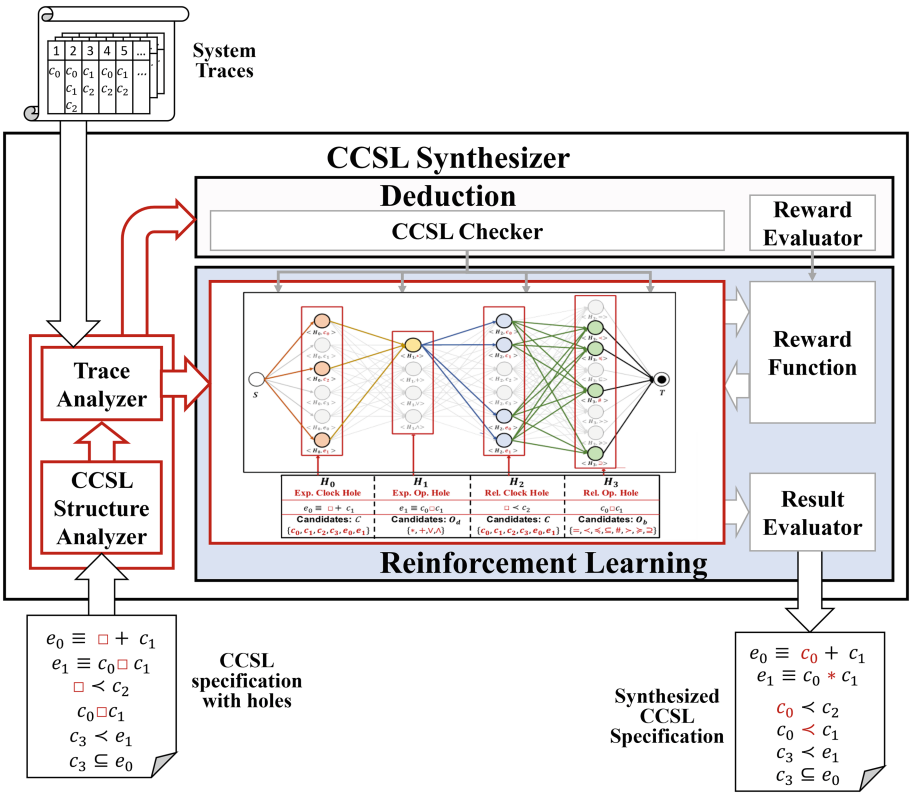

Fig. 10. Architecture and Workflow of a CCSL synthesizer accelerated by curiosity-driven exploration

5 Conclusion

The paper attempts to retrace part of the history of CCSL. A part of its evolution took place at the Software Engineering Institute in Shanghai. We also tried to retrieve the papers from Professor He Jifeng that impacted the most paths that were taken or ignored. Section 2 explores three semantics of CCSL. This is usually considered a good practice and this is recommended by the Unifying Theory of Programming. Hopefully, this section gives a good sense of why this could be useful to study languages under different perspectives. However, we did not go (yet) as far as showing the equivalence of the three semantics. This leaves some exciting perspectives for the future.

Acknowledgements. I would like to sincerely thank the reviewers for their very helpful and constructive comments. I would like to thank the Aoste team, in particular Charles André and Robert de Simone who were at the initiative of CCSL, and also Julien DeAntoni who joined soon after.

In SEI, under the leadership of He Jifeng, there also were academics from the beginning: 何积丰, 刘静, 陈仪香, 朱惠彪.. Then, a second generation: 张民 陈铭松，陈小红，杜德慧 and their students 尹玲, 张元瑞，稿费，胡铭.

This work was also made possible by Inria associated team Plot4IoT and by UCA DS4H (ANR-17-IDEX-0004).

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994). https://doi.org/10.1016/0304-3975(94)90010-8
2. André, C.: SyncCharts: a visual representation of reactive behaviors. Research report RR 95–52, rev. RR (96–56), I3S, Sophia-Antipolis, France (1996)
3. André, C.: Syntax and semantics of the clock constraint specification language (CCSL). Research report RR-6925, INRIA (2009). https://hal.inria.fr/inria-00384077
4. André, C., Peraldi-Frati, M.: Behavioral specification of a circuit using SyncCharts: a case study. In: EUROMICRO Conference, p. 1091. IEEE Computer Society (2000). https://doi.org/10.1109/EURMIC.2000.874620
5. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: Software Engineering and Formal Methods (SEFM), pp. 3–12. IEEE Computer Society (2006). https://doi.org/10.1109/SEFM.2006.27
6. Benveniste, A., Caillaud, B., Carloni, L.P., Sangiovanni-Vincentelli, A.L.: Tag machines. In: Wolf, W.H. (ed.) Embedded Software (EMSOFT), pp. 255–263. ACM (2005). https://doi.org/10.1145/1086228.1086276
7. Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Le Guernic, P., de Simone, R.: The synchronous languages 12 years later. Proc. IEEE **91**(1), 64–83 (2003). https://doi.org/10.1109/JPROC.2002.805826
8. Berry, G., Bouali, A., Fornari, X., Ledinot, E., Nassor, E., de Simone, R.: ESTEREL: a formal method applied to avionic software development. Sci. Comput. Program. **36**(1), 5–25 (2000). https://doi.org/10.1016/S0167-6423(99)00015-5
9. Boulanger, F., Dogui, A., Hardebolle, C., Jacquet, C., Marcadet, D., Prodan, I.: Semantic adaptation using CCSL clock constraints. Electron. Commun. Eur. Assoc. Softw. Sci. Technol. **50** (2011). https://doi.org/10.14279/tuj.eceasst.50.731
10. Bousse, E., Degueule, T., Vojtisek, D., Mayerhofer, T., Deantoni, J., Combemale, B.: Execution framework of the GEMOC studio (tool demo). In: van der Storm, T., Balland, E., Varró, D. (eds.) Software Language Engineering (SLE), pp. 84–89. ACM (2016)
11. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.: LUSTRE: a declarative language for programming synchronous systems. In: Annual Symposium on Principles of Programming Languages, pp. 178–188. ACM Press (1987). https://doi.org/10.1145/41625.41641
12. Chen, X., Liu, Q., Mallet, F., Li, Q., Cai, S., Jin, Z.: Formally verifying consistency of sequence diagrams for safety critical systems. Sci. Comput. Program. **216**, 102777 (2022). https://doi.org/10.1016/j.scico.2022.102777
13. Colaço, J., Pagano, B., Pouzet, M.: SCADE 6: a formal language for embedded critical software development. In: Theoretical Aspects of Software Engineering (TASE), pp. 1–11. IEEE Computer Society (2017). https://doi.org/10.1109/TASE.2017.8285623

14. Deantoni, J., Cambeiro, J., Bateni, S., Lin, S., Lohstroh, M.: Debugging and verification tools for lingua franca in GEMOC studio. In: Forum on Specification & Design Languages (FDL), pp. 1–8. IEEE (2021). https://doi.org/10.1109/FDL53530.2021.9568383
15. DeAntoni, J., Mallet, F.: TimeSquare: treat your models with logical time. In: Furia, C.A., Nanz, S. (eds.) TOOLS 2012. LNCS, vol. 7304, pp. 34–41. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30561-0_4
16. Du, D., Huang, P., Jiang, K., Mallet, F.: pCSSL: a stochastic extension to MARTE/CCSL for modeling uncertainty in cyber physical systems. Sci. Comput. Program. **166**, 71–88 (2018). https://doi.org/10.1016/j.scico.2018.05.005
17. Eker, J., et al.: Taming heterogeneity - the Ptolemy approach. Proc. IEEE **91**(1), 127–144 (2003). https://doi.org/10.1109/JPROC.2002.805829
18. Fecher, H., Schönborn, J., Kyas, M., de Roever, W.-P.: 29 new unclarities in the semantics of UML 2.0 state machines. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 52–65. Springer, Heidelberg (2005). https://doi.org/10.1007/11576280_5
19. Fowler, M.: Domain-Specific Languages. Addison Wesley (2010)
20. Ge, N., Pantel, M.: Time properties verification framework for UML-MARTE safety critical real-time systems. In: Vallecillo, A., Tolvanen, J.-P., Kindler, E., Störrle, H., Kolovos, D. (eds.) ECMFA 2012. LNCS, vol. 7349, pp. 352–367. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31491-9_27
21. Hardebolle, C., Boulanger, F.: ModHel'X: a component-oriented approach to multi-formalism modeling. In: Giese, H. (ed.) MODELS 2007. LNCS, vol. 5002, pp. 247–258. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69073-3_26
22. Harel, D.: StateCharts: a visual formalism for complex systems. Sci. Comput. Program. **8**(3), 231–274 (1987). https://doi.org/10.1016/0167-6423(87)90035-9
23. Jifeng, H.: A clock-based framework for construction of hybrid systems. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) ICTAC 2013. LNCS, vol. 8049, pp. 22–41. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39718-9_2
24. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
25. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice-Hall (1998)
26. Hu, M., Ding, J., Zhang, M., Mallet, F., Chen, M.: Enumeration and deduction driven co-synthesis of CCSL specifications using reinforcement learning. In: Real-Time Systems Symposium (RTSS), pp. 227–239. IEEE (2021). https://doi.org/10.1109/RTSS52674.2021.00030
27. Kang, E.-Y., Mu, D., Huang, L.: Probabilistic verification of timing constraints in automotive systems using UPPAAL-SMC. In: Furia, C.A., Winter, K. (eds.) IFM 2018. LNCS, vol. 11023, pp. 236–254. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98938-9_14
28. Khan, A.M., Rashid, M.: Generation of SystemVerilog observers from SysML and MARTE/CCSL. In: Real-Time Distributed Computing (ISORC), pp. 61–68. IEEE Computer Society (2016). https://doi.org/10.1109/ISORC.2016.18
29. Kienzle, J., Mussbacher, G., Combemale, B., Deantoni, J.: A unifying framework for homogeneous model composition. Softw. Syst. Model. **18**(5), 3005–3023 (2019). https://doi.org/10.1007/s10270-018-00707-8
30. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978). https://doi.org/10.1145/359545.359563
31. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. Int. J. Softw. Tools Technol. Transf. **1**(1–2), 134–152 (1997). https://doi.org/10.1007/s100090050010

32. Le Guernic, P., Benveniste, A., Bournai, P., Gautier, T.: Signal - a data flow-oriented language for signal processing. IEEE Trans. Acoust. Speech Sig. Process. **34**(2), 362–374 (1986). https://doi.org/10.1109/TASSP.1986.1164809
33. Le Guernic, P., Talpin, J., Lann, J.L.: POLYCHRONY for system design. J. Circ. Syst. Comput. **12**(3), 261–304 (2003). https://doi.org/10.1142/S0218126603000763
34. Lee, E.A., Sangiovanni-Vincentelli, A.L.: A framework for comparing models of computation. IEEE Trans. Comput. Aided Des. Integr. Circ. Syst. **17**(12), 1217–1229 (1998). https://doi.org/10.1109/43.736561
35. Li, T., et al.: Runtime verification of spatio-temporal specification language. Mob. Netw. Appl. **26**(6), 2392–2406 (2021). https://doi.org/10.1007/s11036-021-01779-5
36. Liu, J., Liu, Z., He, J., Mallet, F., Ding, Z.: Hybrid MARTE statecharts. Front. Comput. Sci. **7**(1), 95–108 (2013). https://doi.org/10.1007/s11704-012-1301-1
37. Lund, M.S., Refsdal, A., Stølen, K.: 4 semantics of UML models for dynamic behavior. In: Giese, H., Karsai, G., Lee, E., Rumpe, B., Schätz, B. (eds.) MBEERTS 2007. LNCS, vol. 6100, pp. 77–103. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16277-0_4
38. Mallet, F., Millo, J., de Simone, R.: Safe CCSL specifications and marked graphs. In: Formal Methods and Models for Codesign (MEMOCODE), pp. 157–166. IEEE (2013). https://ieeexplore.ieee.org/document/6670955/
39. Mallet, F., Zholtkevych, G.: Coalgebraic semantic model for the clock constraint specification language. In: Artho, C., Ölveczky, P.C. (eds.) FTSCS 2014. CCIS, vol. 476, pp. 174–188. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17581-2_12
40. Maraninchi, F., Rémond, Y.: Argos: an automaton-based synchronous language. Comput. Lang. **27**(1/3), 61–92 (2001). https://doi.org/10.1016/S0096-0551(01)00016-9
41. Montin, M., Pantel, M.: Mechanizing the denotational semantics of the clock constraint specification language. In: Abdelwahed, E.H., Bellatreche, L., Golfarelli, M., Méry, D., Ordonez, C. (eds.) MEDI 2018. LNCS, vol. 11163, pp. 385–400. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00856-7_26
42. Montin, M., Pantel, M.: Towards multi-layered temporal models: a proposal to integrate instant refinement in CCSL. In: Peters, K., Willemse, T.A.C. (eds.) FORTE 2021. LNCS, vol. 12719, pp. 120–137. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-78089-0_7
43. Nielsen, M., Plotkin, G.D., Winskel, G.: Petri nets, event structures and domains, part I. Theor. Comput. Sci. **13**, 85–108 (1981). https://doi.org/10.1016/0304-3975(81)90112-2
44. Peters, J., Przigoda, N., Wille, R., Drechsler, R.: Clocks vs. instants relations: verifying CCSL time constraints in UML/MARTE models. In: Formal Methods and Models for System Design (MEMOCODE), pp. 78–84. IEEE (2016). https://doi.org/10.1109/MEMCOD.2016.7797750
45. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebraic Program. **60–61**, 17–139 (2004)
46. Rutten, J.J.M.M.: Universal coalgebra: a theory of systems. Theor. Comput. Sci. **249**(1), 3–80 (2000). https://doi.org/10.1016/S0304-3975(00)00056-6
47. Sander, I., Jantsch, A.: System modeling and transformational design refinement in ForSyDe [formal system design]. IEEE Trans. Comput. Aided Des. Integr. Circ. Syst. **23**(1), 17–32 (2004). https://doi.org/10.1109/TCAD.2003.819898

48. Shao, Z., Liu, J., Ding, Z., Chen, M., Jiang, N.: Spatio-temporal properties analysis for cyber-physical systems, pp. 101–110 (2013)
49. de Simone, R., André, C.: Towards a "synchronous reactive" UML profile? Int. J. Softw. Tools Technol. Transf. **8**(2), 146–155 (2006). https://doi.org/10.1007/s10009-005-0206-9
50. Suryadevara, J., Seceleanu, C., Mallet, F., Pettersson, P.: Verifying MARTE/CCSL mode behaviors using UPPAAL. In: Hierons, R.M., Merayo, M.G., Bravetti, M. (eds.) SEFM 2013. LNCS, vol. 8137, pp. 1–15. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40561-7_1
51. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pac. J. Math. **5**(2), 285–309 (1955)
52. The Object Management Group: Unified Modeling Language, Version 2.0 (2005). https://www.omg.org/spec/UML/2.0
53. The Object Management Group: A UML Profile for MARTE, v. 1.0 (2009). https://www.omg.org/spec/MARTE/
54. Nguyen Van, H., Balabonski, T., Boulanger, F., Keller, C., Valiron, B., Wolff, B.: A symbolic operational semantics for TESL. In: Abate, A., Geeraerts, G. (eds.) FORMATS 2017. LNCS, vol. 10419, pp. 318–334. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65765-3_18
55. Van, H.N., Boulanger, F., Wolff, B.: TESL: a model with metric time for modeling and simulation. In: Muñoz-Velasco, E., Ozaki, A., Theobald, M. (eds.) Temporal Representation and Reasoning (TIME). LIPIcs, vol. 178, pp. 15:1–15:15. Schloss Dagstuhl (2020). https://doi.org/10.4230/LIPIcs.TIME.2020.15
56. Yang, J., Chen, X., Yin, L.: Eliciting timing requirements for cyber-physical systems: a multiform time based approach. In: Theoretical Aspects of Software Engineering (TASE), pp. 199–206 (2021). https://doi.org/10.1109/TASE52547.2021.00024
57. Zhang, M., Dai, F., Mallet, F.: Periodic scheduling for MARTE/CCSL: theory and practice. Sci. Comput. Program. **154**, 42–60 (2018). https://doi.org/10.1016/j.scico.2017.08.015
58. Zhang, M., Song, F., Mallet, F., Chen, X.: SMT-based bounded schedulability analysis of the clock constraint specification language. In: Hähnle, R., van der Aalst, W. (eds.) FASE 2019. LNCS, vol. 11424, pp. 61–78. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-16722-6_4
59. Zhang, Y., Mallet, F., Chen, Y.: A verification framework for spatio-temporal consistency language with CCSL as a specification language. Front. Comp. Sci. **14**(1), 105–129 (2018). https://doi.org/10.1007/s11704-018-7054-8
60. Zholtkevych, G., Labzhaniia, M.: Understanding safety constraints coalgebraically. In: Computational Linguistics and Intelligent Systems (COLINS), vol. 2604, pp. 1–19 (2020). http://ceur-ws.org/Vol-2604/paper1.pdf