Schedulability analysis with CCSL specifications

Ling Yin, Jing Liu, Zuohua Ding, Frédéric Mallet, Robert de Simone

To cite this version:

Ling Yin, Jing Liu, Zuohua Ding, Frédéric Mallet, Robert de Simone. Schedulability analysis with CCSL specifications. APSEC 2013 - 20th Asia-Pacific Software Engineering Conference, Dec 2013, Bangkok, Thailand. pp.414-421, 10.1109/APSEC.2013.62 . hal-00926305

HAL Id: hal-00926305
https://inria.hal.science/hal-00926305

Submitted on 3 Dec 2022

Schedulability analysis with CCSL specifications

Ling Yin[1], Jing Liu*[1], Zuohua Ding[2],Frédéric Mallet[3,4], Robert de Simone[4]
[1]*Shanghai Key lab of Trustworthy Computing, East China Normal University, Shanghai, China*
[2]*Center of Math Computing and Software Engineering, Zhejiang Sci-Tech University, Hangzhou, Zhejiang, China*
[3]*Univ. Nice Sophia Antipolis, I3S, UMR 7271, CNRS,*
[4]*INRIA Sophia Antipolis Méditerranée 06900 Sophia Antipolis, France*

***Abstract*—The Clock Constraint Specification Language (CCSL) is a formal polychronous language based on the notion of logical clock. It defines a set of kernel constraints that can represent both asynchronous and synchronous relations. It was originally developed as part of the UML Profile for MARTE to express causal and temporal constraints of Real-time and Embedded Systems. In this paper, we explore the use of CCSL for modeling scheduling requirements and to conduct schedulability analysis. For this purpose, a dedicated scheduling library of CCSL has been built. This library is endowed with a state-based operational semantics, and is applied to solve issues related to schedulability analysis and latency-insensitive design. We establish schedulability categories and latency-insensitiveness property in the context of the semantics, and solve those issues by using model checking techniques.**

***Keywords*-CCSL; schedulability analysis; model checking;**

I. Introduction

The Clock Constraint Specification Language (CCSL)[1] has originally been defined as the time model of the UML profile for Modeling and Analysis of Real-Time Embedded systems (MARTE), expressing timing constraints between events or operations. It captures essential causal and temporal notions from both synchronous and asynchronous specifications. Now, it has evolved beyond the timing specification of MARTE models, and has become a full-fledged domain-specific modeling language widely used in automotive and avionics domains[2][3][4]. In this paper, we explore the use of CCSL for scheduling. Whereas there were other attempts[5][6] to use CCSL in scheduling at different design phases, these works mainly focused on the modeling part, the simulation and consistency validation. However, schedulability analysis has not been thoroughly discussed. In this paper, we mainly focus on the schedulability analysis of CCSL specifications.

CCSL is based on logical clocks. A logical clock is not a physical device that provides regular physical time intervals (as a watch). It is instead an abstract measure unit, possibly user-defined, by which durations and dates may be set or measured as long as a sequence of occurrences ("ticks") on that clock can be observed or produced. In that sense a logical clock largely corresponds to an abstract *event* (considered as a sequence of event occurrences), simply stressing its temporal relevance and role in the design. A logical clock needs not in any case be related to regular physical realities (though it can). Logical clocks have first been introduced in that context by synchronous languages[7]. They have been widely been used in other domains ever since and proved to provide a high degree of flexibility in modeling[8].

CCSL provides a set of kernel constraints to specify relations among clocks, considering both synchronous and asynchronous aspects. Clocks and clock constraints can be used to model scheduling issues: Tasks/jobs are associated with clocks that represent their triggering conditions; scheduling requirements (timed and functional causal relations between tasks/jobs, constraints introduced by the execution platform, resource limitations and scheduling expectations) are then expressed as CCSL constraints among the corresponding clocks. For scheduling purpose, we have built a new CCSL library, which contains a set of clock constraints dedicated to the scheduling domain. These constraints are endowed with informal and formal state-based operational semantics (called ccLTS). Some of them are selected from the kernel set of CCSL constraints; some are newly defined; the others are composed from existing constraints in the library.

Then a scheduling model of a system can be built as a CCSL specification consisting of a set of clocks and clock constraints (from the scheduling library). The specification formally describes expected requirements and its behaviors are solutions to the conjunction of all contained constraints. A behavior is a sequence of *steps*, for each step (index), several clocks are assigned to tick simultaneously at that step; it can be regarded as a valid schedule of the system if every clock proceeds as far as it is meant to. For each step, the specification allows usually more than one choices of ticking clocks satisfying its constraints. That is to say a specification represents all the solution behaviors (thus all the valid schedules), departing from each other by choosing different set of clocks to tick at the same step index. Schedulability for a given system can be decided by checking which category a CCSL specification falls into:

- Category 1, the most favorable case, all clocks can proceed as far as they are meant to, no matter what choice is made. Scheduling is then just a matter of choosing the "time placement" of tasks/jobs associated with clocks, without worrying about bad consequences (*e.g.,* not enough time slots left for some tasks/jobs).
- Category 2, the least favorable case (unschedulable), bad

consequences cannot be avoided. Some (or all) clocks will eventually be halted no matter what choice is made.

- Category 3 is in the middle, that some behaviors lead to valid schedules, while others do not. Users then need to be careful to pick on the valid solutions.

Very often, distinct schedules depart only from one another in a non-essential way, by shifting the exact timing placement they each assign to the same amount of clock ticks. This property (that distinct totally ordered schedules can in fact refer to the same partial order trace), called latency-insensitiveness[9], can be used later in scheduling.

In this paper, we focus on the analysis of schedulability and latency-insensitiveness. Generic examples are provided to display typical phenomena to be taken into account. We provide a label transition system based semantics, ccLTS, for CCSL. Then, schedules can been seen as runs of a ccLTS, the schedulability categories and latency-insensitive property are formalized in the context of ccLTS. Another benefit of the state-based semantics is that it brings the possibility of using model-checking techniques in scheduling analysis. We translate CCSL specifications into NuSMV models based on ccLTS, express conditions for schedulability categories and latency-insensitiveness in CTL formulas, and then check them upon the translated NuSMV models.

The paper is constructed as follows: We discuss related work in section II. The kernel of CCSL is briefly introduced in section III. Then we explore the use of CCSL in models for scheduling, and give the syntax and semantics of our scheduling library in section IV. The schedulability issues are discussed in section V. In section VI, we provide methods for detecting the schedulability categories and latency-insensitiveness. Then we conclude the paper and discuss future plans in section VII.

II. Related Work

We have done some work on analysis of CCSL specifications, such as simulation[4], exhaustive verification[10] and observer generation[11]. But this paper is the first one targeting schedulability analysis. Besides CCSL, many scheduling frameworks have indeed been introduced for schedulability analysis and efficient scheduling generation. We discuss the differences and similarities between our work and them in the next paragraph. As an overview, we are not trying to compete with these in the current paper, but to provide an alternative method relying on a very different formalism based on the notion of logical clock, *i.e.,* CCSL. We take full advantages of CCSL in modeling, such as the flexibility provided by logical time, multi-form time bases, unified expressing of both functional requirements and extra performance constraints. This requires the extension of the semantics of CCSL to provide a new model adequate for (1) deciding whether a specification do admit *at least one solution* (schedulability), and, (2) computing one valid schedule, without considering a potential criterion of optimality.

Most of the existing frameworks, *e.g.,* timed automata [12][13], timed Petri nets [14][15], timed process algebra [16], rely "physical-by-nature" timing. The distinctive difference is that their systems are presented as synchronously timed (on a single global continuous time), with various timed events being constrained by value relations between so-called clocks (a different notion from our logical clocks in CCSL), which are devices measuring physical time as it elapses. Thus, correspondence should rather be between our logical clocks and the timed events. Data flow graphs provide scheduling models where the initial constraints are less on timing and more on dependencies or on exclusive resource allocation. The resulting schedules are almost always of *modulo* periodic nature, matching the CCSL expressiveness. A possible schedule we provide, a sequence of steps filled with simultaneous clock ticks, often falls into the periodic nature too (state space of a CCSL specification is finite while a schedule is usually infinite). The *step* can be seen as a global ticking clock, and the scheduling decision (which clock ticks at which step) then can be seen as periodic binary word masks of the `filteredBy` constraint; this corresponds to the binary word encoded schedule for the firing of each actor in a data flow model. The relationships between data flow graphs and CCSL is discussed in more details in other works[17][18].

Our latency-insensitiveness can be regarded as a generalization of the Mazurkiewicz trace theory and the confluence in process algebra theory[19] to a synchronous setting. It also seems closely related to the conflict-free in Process Networks[20][21] and the weak endochrony in SIGNAL[22]. They focus on the desynchronization from synchronous models to asynchronous implementations, only requiring that two states reached from a common state can join in a common state again, what happened along the different paths is not relevant. While our work concentrates on the latency-insensitiveness between scheduling results, the number of ticks for each clock is crucial. The mix of synchrony and asynchrony and the clock terminations in CCSL rise the complexity of our checking. In the future, we want to inspect how relations between clocks can be used to detect unschedulability and latency-insensitiveness at construction time prior to building the full state space or even syntactically without building the full state space at all.

III. A brief introduction to CCSL

As already mentioned, clock ticks amount to event occurrences. Two event occurrences e and e' may temporally be: *simultaneous/coincident*, denoted as $e \equiv e'$, meaning that the two occurrences occur (are always observed) at the same time step; or one may *precede* the other one, denoted as $e \prec e'$, introducing a temporal ordering between them. We also note $e \# e'$ for *not* $(e \equiv e')$, and $e \preceq e'$ for $(e \prec e'$ *or* $e \equiv e')$. Note that causality intentions in design in general allows relation $\preceq$, when instantaneous/combinatorial causality is also legible.

At the clock level (again, a clock being a sequence of ticks/instants), two basic ordering relations are defined based on the above ticking relations. First, on the synchronous side, *subClock*: c *subClock* c' iff $\forall i, j : c_i \preceq c_j \ \exists k, l : (c_i \equiv c'_k) \wedge (c_j \equiv c'_l) \wedge (c'_k \preceq c'_l)$, c_i being the i^{th} tick of clock c; this definition means that every tick of c has to occur synchronously with a tick of c', but not always the other way around, so there are potentially more ticking instants in c'. Moreover, this relation is order-preserving. Second, on the asynchronous side, *faster than* (*strictPre* or *causes*, strictly vs. weakly): c *strictPre* c' iff $\forall i,\ c_i \prec \ c'_i$; c *causes* c' iff $\forall i,\ c_i \preceq \ c'_i$. *faster than* can a priori lead to unbounded drifts between clocks (to check that the second clock does not tick too early, we need to monitor how many ticks the first one has performed "in advance", just so it does not drop below 0, and this requires an integer in the course of simulation). In frequent cases the specification provides an interval between the clocks, so that we get bounded drifts. In addition to these two clock relations, two other kernel relations, *coincides* and *exclusion*, are defined by extending corresponding ticking relations into clock level respectively. Kernel CCSL functions that build new clocks from existing ones are also defined, including *union, intersect, wait, inf, sup, upto* and *concat* (we give their semantics in next section).

The expressiveness of the CCSL kernel library is limited, for instance by the lack of support for parameterized constructs[23]. This is overcome by allowing new libraries and user-defined constraints in order to fit the use for a specific domain. In this paper, we build a library dedicated to schedulability analysis.

IV. A CCSL LIBRARY FOR SCHEDULABILITY ANALYSIS

Scheduling theories rely on task/job models that supposedly abstract real applications. In our view, the successive timing values for characteristic feature of successive executions of a task/job can be seen as a logical clock. Scheduling requirements based on numerous distinct parameters of tasks (dispatch time, period, deadline, jitter...) are essentially constraints among corresponding clocks. Results of classical scheduling algorithms are almost always of modulo periodic nature (subClock design style with various filter masks).

A. CCSL *scheduling model*

We present primitive constraints first, explain their intuitive semantics and the reason to include them in the library.

- |c| denotes the length of clock c (number of ticks). Often, tasks/jobs are assumed to execute infinitely. But users should also be able to say that a task/job should execute a specific finite number of times and then terminate (static condition), or a task should terminate when some condition is satisfied (dynamic condition). So we need to consider finite clocks and their proper terminations. Length limitation is modeled by $|c| \leq k, k \in N$. $|c| = \infty$ when the clock is infinite. When we define constraints, we take clock termination into account and apply time ordering which encodes causality only where ticks are defined (otherwise the causality is defeated).
- `filteredBy` uses an explicit mask to decide on each new tick (of the superclock) whether it should be preserved or discarded in the subclock definition. A mask is a binary word (with "1" meaning *keep* and "0" *discard*) of length at least equal to the length of the superclock. It preserves termination, and may create a finite clock from an infinite one (if the mask is of finite length). In practice, the mask is an ultimately periodic pattern, consisting of an initialization prefix part and a repeated part. Periodic subsystems or even periodically patterned ones fall into this design style. Moreover, filteredBy also provides a means to describe (full or partial) solutions to the system ordering. When original requirements are themselves not synchronously patterned, the *results* of scheduling are often computed of that nature. Having a way to deal with partial scheduling results inside the syntax of the model itself is useful, as it allows solving approaches by incremental time refinements. Common used forms of filteredBy are extracted by suitable masks. `delay_n` discards the n first ticks of its input clock.; `wait_n` only ticks once, on the original n^{th} tick of its input clock.
- `upto` takes two input clock arguments, and ticks with the first one as long as the second has not started. It provides a way to define dynamic terminations of clocks.
- `concat` takes two inputs and ticks with the first until it (the first input) terminates, then starts ticking as the second. This corresponds to sequential composition. It is of real use only if its first clock input terminates, requiring to check the lifespan of it (at least at run time).
- `union` (or `intersect`) has two inputs and creates a new clock that ticks when at least one (or both) input clocks do. They provide means to model merge and join. Finite `union` requires the two clock arguments to be finite, while `intersect` may produce a finite clock even both arguments are infinite.
- `sampledOn` function is a mix of synchronous and asynchronous spirit. It takes two input clocks and ticks synchronously with the second, but only in case the first input clock ticked since the previous tick of the second. So, its role is essentially to record whether there was a tick of the first clock since "last time". This can be used to model the classical data access problem, though it has larger use. It create a finite clock if one of its inputs is.
- `inf` (or `sup`) takes two input clocks, and creates a new clock that is faster (or slower) than both of the inputs. The i^{th} tick of the new clock is coincident with the earlier (or later) of the i^{th} tick of input clocks. `inf` provides an infinite clock as soon as one arguments is infinite, while `sup` provides a finite one as soon as one input is finite.

- The fundamental synchronous relation is the subClock discussed above, restricting that the subclock can tick only if its superclock does.
- exclusion relation specifies that the two clocks can never tick at the same step. It expresses the exclusive access to critical resources naturally.
- The basic asynchronous relations are the two versions of fasterThan, the strictPre $\prec$ and causes $\preceq$. They both say that the first clock is quicker than the second and $|slower| \leq |faster|$. They express precedences, *e.g.,* meeting a deadline, priority comparison. Their difference is that when the index difference between the two clocks is 0, the strict one forbids the tick of the second, while the weak one allows.
- boundedDiff_i_j bounds an integer interval [i,j] to the drift of two clocks, denoted as $i \leq left - right \leq j$. It can be viewed as an extension of fasterThan, not restricting which clock is faster, but stating their index difference cannot exceed the range. A clock in bounded drift with a finite (resp. infinite) one is finite (resp. infinite) as well. It can be used to express buffer size limitations, transmission delay, jitter.... For instance, using 0 as the left bound and *buffer size* as the right bound between two $write$ and $read$ clocks.
- alternate indicates the alternate ticking of two clocks, starting from the left one. It can model asynchronous communications. A clock that alternates with a finite (resp. infinite) one is finite (resp. infinite) as well.

The we provide some often used patterns of grouping primitive constraints to express the scheduling concerns.

- We build a constraint called duration that has four parameters, $task_{start}$, $task_{end}$, $measure$ and d. It says that the execution duration of the task is less than "d" units of the $measure$ clock. It could be used to express a deadline, the hard real time limit, using a physical clock as $measure$ clock.
  $\texttt{duration}(clock\ task_{start}, task_{end}, measure; int\ d)$
  $$\triangleq \begin{cases} measure \prec task_{start} \\ task_{end} \prec (measure\ \texttt{delayedBy}\ d) \end{cases}$$
- Constraint repetition is defined to model successive occurrences of an event, with a repetition rate (nominal duration) and a jitter.
  $\texttt{repetition}(clock\ task, measure; int\ rate, jitter)$
  $$\triangleq \begin{cases} standard = measure\ \texttt{filteredBy}\ (1.0^{rate-1})^{\omega} \\ -jitter\ \leq standard - task\ \leq jitter \end{cases}$$
- Constraint sporadicity states that a task is invoked randomly but with a minimum inter-invocation interval.
  $\texttt{sporadicity}(clock\ task, measure; int\ interval)$
  $$\triangleq \begin{cases} standard = measure\ \texttt{filteredBy}\ (1.0^{rate-1})^{\omega} \\ 0\ \leq measure - c\ \leq\ 1 \\ measure \prec (task\ \texttt{wait\_1}) \end{cases}$$

B. State-based operational semantics

We define a label transition system ccLTS to express the operational semantics of CCSL. It is explicitly defined for only primitive constraints, since composed constraints are defined by grouping primitive ones with composition rules.

Definition 1. A ccLTS is a tuple $L =< S, Clocks, T, \hat{s} >$,

- S is a set of states and $\hat{s} \in S$ is the initial state.
- $Clocks$ is a finite set of clock names, its powerset is denoted as $ClockSets$.
- $T \subseteq S \times\ ClockSets\ \times S$ defines the transition relation, each transition is labeled by a set of clocks that tick simultaneously in that step. $(s, C, s^{'}) \in T$ is also denoted as $s \xrightarrow{C}_T s^{'}$ simply or $s \xrightarrow{C} s^{'}$ if clear from context.

A ccLTS is deterministic iff $\forall s \in S, (s \xrightarrow{C} s_1 \wedge s \xrightarrow{C} s_2) \Rightarrow (s_1 = s_2)$. A path of a ccLTS is a sequence of transitions, either finite or infinite, $\rho = s_0 \xrightarrow{C_0} s_1 \xrightarrow{C_1} \cdots \xrightarrow{C_{k-1}} s_k \cdots$. A run is a path from the initial state, and its trace is the series of transition labels: $C_0; C_1; \cdots C_k; \cdots$. We extend the transition relation to $\Rightarrow$: $s \overset{M}{\Longrightarrow} s_k$ iff there exists a path from s_0 to s_k, and M is the union the transition labels, $M = \biguplus_{i=0}^{i=k-1} C_i$. The number of times that clock c appears in M is denoted as $M(c)$. We say $s^{'}$ is reachable from s, if there exists $s \overset{M}{\Longrightarrow} s^{'}$. The set of states that are reachable from s is noted as $Reach(s)$.

One can easily build a ccLTS for a primitive clock constraint, with $Clocks$ as the set of involved clocks, and transitions modeling the transformations allowed by the constraint. Every state has a stutter transition $s \xrightarrow{\emptyset} s$ since no constraint is violated or changed if no clock ticks. If a state has no non-stutter outgoing transitions, it is called a *deadlock* state. Due to page limitation, we only show the ccLTSs of constraints used in the later examples in Figure 1. Stutter transitions are not shown explicitly and please ignore the boolean expressions on $Term(c)$ right now (they are discussed in section V-B).

A specification is a conjunction of constraints, so its behavior has to respect all its constraints. Transitions from two ccLTSs can be composed only if there is no conflict; conflict here means violation of the conjunction, a common involved clock ticks in one transition while does not in the other.

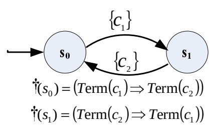
Definition 2. Let $L_i = \{S_i, Clocks_i, T_i, \hat{s}_i\}$ for $i \in \zeta$, where $\zeta = \{1, ..., n\}$ is a finite index set, be a set of ccLTSs, their *composition* is a synchronized product, $||\{L_i\}_{i \in \zeta} = \{S, Clocks, T, \hat{s}\}$, where $S = S_1 \times \cdots \times S_n; \hat{s} = \hat{s}_1 \times \cdots \times \hat{s}_n; Clocks = Clocks_1 \cup \cdots \cup Clocks_n;$

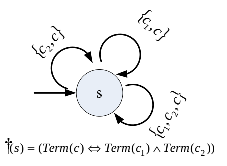
$$\frac{\forall i,j \in \zeta (i \neq j), s_i \xrightarrow{C_i}_{T_i} s_i^{'}, s_j \xrightarrow{C_j}_{T_j} s_j^{'}, \forall c \in Clocks_i \cap Clocks_j, c \in C_i \Leftrightarrow c \in C_j}{(s_1, ..., s_n) \xrightarrow{C_1 \cup \cdots \cup C_n} (s_1^{'}, ..., s_n^{'})}$$

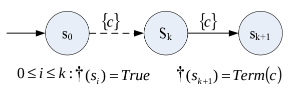
The semantics of a specification is given by the runs of its corresponding ccLTS, which is a composition result of the ccLTSs of its constraints.

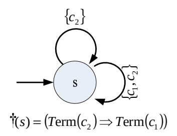
V. Schedulability and latency-insensitive analysis with the CCSL scheduling library

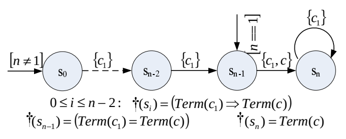
TThe purpose of scheduling is to assign resources and time slots to tasks so that all tasks are accomplished under

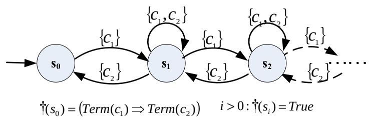

(a) c_1 `alternate` c_2

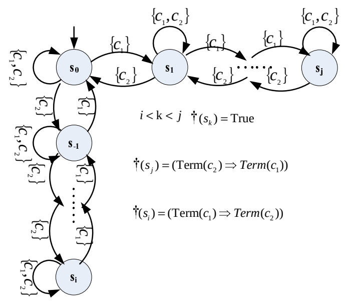

(b) $c = c_1$ `union` c_2


(c) $|c| \leq\ k$


(d) c_1 `subClock` c_2


(e) $c = c_1$ `wait_n`


(f) c_1 `strictPre` c_2


(g) $i \leq c_1 - c_2 \leq j$

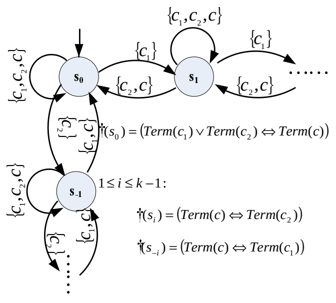

(h) $c = c_1$ `sup` c_2

Figure 1: ccLTS encodings of clock constraints

requirements. In that sense, a valid schedule in CCSL context should be a sequence of steps where every clock ticks as far as it is meant to (infinitely often or up to its proper termination). A number of concerns in CCSL schedulability analysis are revealed by some generic examples in section V-A. Then we give the formal definitions of schedulability classification and latency-insensitiveness in section V-B.

A. *Motivation examples*

For simplicity, clock declarations and the stutter transitions are ignored in their ccLTS graphs given in Figure 2.

Example 1. Specification $S1 :\ c = b$ `sup` $a, -1 \leq b{-}a \leq 1$ belongs to category 1 introduced in the introduction. As Figure 2(a) shows, no matter which choice is made, every clock can tick infinitely often.

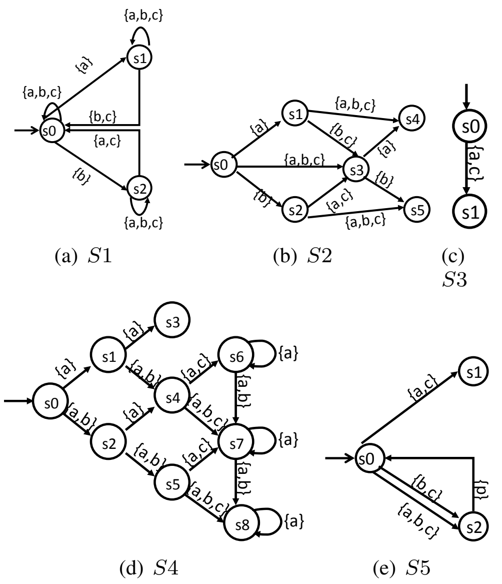
Example 2. If we specify clock c to be finite, whose maximal length is 1, we get $S2 :\ c = b$ `sup` $a,\ -1 \leq b{-}a \leq 1,\ |c| \leq 1$. As Figure 2(b) shows, clock c cannot tick since states s_3, s_4, s_5, a and b cannot tick from s_4 and s_5. But those "cannot tick anymore" are expected, called proper terminations: for c, it is directly specified; for a and b, they are dynamically derived: $c = b$ `sup` a states that c ticks together with the slower one of a and b. At state s_4 (resp. s_5) clock b (resp. s_5) is the slower one, so the termination of c derives the termination of b (resp. a), then because of the length match required by $-1 \leq b - a \leq 1$, a (resp. b) also terminates. No matter which choice is made, every clock can tick up to its proper termination.

Example 3. $S3 : c = a$ `union` b, b `strictPre` d, c `alternate` d, a `strictPre` b. Both a and c can tick once, b can never tick, even through all of them are supposed to infinite. This halt should be distinguished from the proper termination. $S3$ is unschedulable.

Example 4. $S4 : b$ `subClock` $a,\ c = a$ `wait_3`, b `strictPre` $c, 0 \leq b - c \leq 2$. At least one tick of b should be chosen to tick before the third tick of a, then c ticks together with the third tick of a and then terminates properly. Clock b terminates properly when the index difference between b and c reaches 2, and a ticks infinitely. However, none of the constraints forbids a to tick two times without one b. If b is not chosen in time, the specification goes into state s_3 where every clock gets halted.

Example 5. $S5 : c = a$ `union` b, b `strictPre` $d, 0 \leq b{-}d \leq 2, c$ `alternate` d is in category 3. Valid schedules can be obtained only if users do not choose $\{a, c\}$ when the specification is in state s_0.

Proper termination and halt should be distinguished although they both act as "cannot tick anymore". $S2$ belong to category 1 while $S3$ is in category 2. Specification $S1$ should be regarded as latency-insensitive, since in each valid schedule every clock ticks infinitely often, the only difference is the exact step placement. While $S2$ is not latency-insensitive: when it is in state s_3, choosing $\{a\}$ or $\{b\}$ is different. Choosing $\{a\}$ will produce a valid schedule that has two ticks of a and one tick for b and c, while choosing b will allow two ticks of b, one tick for a and one tick for c. The latency-insensitive criteria should consider the amount of ticking times of finite clocks. The schedulability categories and latency-insensitiveness will be formally defined and discussed in next subsection, based on a refined ccLTS (attaching proper termination information


(a) $S1$ (b) $S2$ (c) $S3$

(d) $S4$ (e) $S5$

Figure 2: ccLTSs of example specifications

to states to distinguish proper terminations from halts).

B. Schedulability classification and latency-insensitiveness

We use predicate $Term(c)$ to indicate the proper termination of c (ticked up to its length limit), which can be statically specified or dynamically computed.

Definition 3. A refined ccLTS is a tuple $rL =< L, Terms, \dagger >$, where

- $L =< S, Clocks, T, \hat{s} >$ is a basic ccLTS defined above,
- $Terms$ is a set of termination predicates $Terms = \{Term(c) | c \in Clocks\}$,
- $\dagger : S \to \Phi$ is a function mapping S to Φ, where Φ is a set of boolean formulas over $Terms$.

Sometimes the proper termination of a clock is defined in a constraint. While sometimes, it cannot be decided by a single constraint and is a global decision on several constraints. See $S2$ as an example, the proper termination of c at state s_3 is explicitly stated by $c = a$ `wait_3`, while the terminations of b and a are not stated in any individual constraint; they result from global decisions on the whole specification. The $\dagger$ function provides a way to express the possibilities of terminations symbolically for each constraint and to determine the global decisions of the whole specification. For each constraint, the $\dagger$ function definition should remain consistent with the constraint semantics. A terminated clock cannot occur in any further outgoing transition and cannot come back to life either. For example, for $c = c_1$ `wait_n` in Figure 1(f), when c terminates at s_n, there is no c labeling on any further transitions and $Term(c)$ is true in all further states.

To make global decisions on a specification, the boolean termination expressions should be intersected along with the composition of individual ccLTSs, *i.e.,* for $||\{T_i\}_{i \in \zeta = \{1,...,n\}} = \{S, Clocks, T, \hat{s}, Terms, \dagger\}$ ($i \in \zeta$), $S, Clocks, T, \hat{s}$ are obtained by basic ccLTS composition (definition 2), $Terms = Terms_1 \cup \cdots \cup Terms_n$, and $\forall s = (s_1, ..., s_n) \in S, \dagger(s) = \dagger_1(s_1) \wedge \cdots \wedge \dagger_n(s_n)$.

It is easy to see that the composition preserves the termination consistency rules described above. A clock c is properly terminated at s only if $Term(c)$ is assigned to true in every evaluation that satisfies $\dagger(s)$, denoted as $\dagger(s) \Rightarrow Term(c)$.

A valid schedule, which requires every clock either to tick infinitely often or up to its proper termination, is defined as:

Definition 4. Let ρ be a run of a specification, $\rho = s_0 \xrightarrow{C_0} s_1 \xrightarrow{C_1} \cdots \xrightarrow{C_{k-1}} s_k \cdots$, the trace of ρ is a *valid schedule* called ξ (or say the run ρ produces a valid schedule ξ), iff ρ satisfies $\forall i, j : 0 \leq i \leq j \leq k, \forall c \in Clocks$, $c \notin C_j \Rightarrow (\dagger(s_i) \Rightarrow Term(c))$. The ticking number of a clock $c \in Clocks$ in ξ is denoted as $\mathsf{TN}_c(\xi)$. If c ticks infinitely many times, then $\mathsf{TN}_c(\xi) = \infty$.

Recall that a run can be finite or infinite. We call a finite run *extendable* if its last state has non-stutter outgoing transitions. A run is *maximal* if it is infinite, or finite and unextendable. Obviously, the trace of an extendable finite run is not a valid schedule. And we cannot determine the schedulability of a specification by extendable finite runs.

Definition 5. A CCSL specification falls into

- *category 1* iff all maximal runs produce valid schedules.
- *category 2* iff no run can produce a valid schedule.
- *category 3* iff some maximal runs produce valid schedules while others do not.

Definition 6. A specification is *latency-insensitive* if all its reachable states satisfy the following conditions:

1. Determinism: $s \xrightarrow{C} s_1 \wedge s \xrightarrow{C} s_2 \Rightarrow s_1 = s_2$.
2. For each pair of outgoing transitions of s, $s \xrightarrow{C_1} s_1, s \xrightarrow{C_2} s_2$, $\exists s^{'}$ that $s_1 \xrightarrow{C_3} s^{'}, s_2 \xrightarrow{C_4} s^{'}$ and $C_1 \uplus C_3 = C_2 \uplus C_4$, where $\uplus$ is the multi-set union.

Latency-insensitiveness leads to some delightful results to be used in later on scheduling or simulating. Due to page limitation, we only present the results here. Please check the proofs in our technique report.

Proposition 1. If a specification is latency-insensitive, its ccLTS satisfies that for any reachable state $s \in S$, if $s \overset{M_1}{\Longrightarrow} s_1$ and $s \overset{M_2}{\Longrightarrow} s_2$, then $\exists s^{'} \in S, s_1 \overset{M_3}{\Longrightarrow} s^{'}, s_2 \overset{M_4}{\Longrightarrow} s^{'}$ and $M_1 \uplus M_3 = M_2 \uplus M_4$.

Corollary 1. If a specification is latency-insensitive, then in its ccLTS, if a deadlock state s_d is reachable from the initial state $\hat{s}$ ($\hat{s} \overset{M_1}{\Longrightarrow} S_d$), let $s_1 \in S$, $\hat{s} \xrightarrow{C_1} s_1$, then $s_1 \overset{M_2}{\Longrightarrow} s_d$ and $M_1 = C_1 \uplus M_2$.

Theorem 1. Let $Sched$ be the set of all the valid schedules of a latency-insensitive specification, then $\forall \xi_i, \xi_j \in Sched, \forall c \in Clocks$, $TN_c(\xi_i) = TN_c(\xi_j)$.

The theorem states that all the valid schedules of a latency-insensitive specification depart from each other only by the step placement of clock ticks. Knowing the schedulability classification and latency-insensitiveness ahead can help produce a schedule: If a specification belongs to category 1, users can relax a little bit since no choice may ever cause bad consequences; and the situation is event better for latency-insensitive, users can pick choices freely at any step. If the specification is category 2, any attempts to find a correct schedule is useless. If category 3, users need to be careful to avoid bad choices leading to invalid schedules.

VI. Schedulability category detection and latency-insensitive checking

A CCSL specification represents all possible runs thus all valid schedules; this allows the use of model-checking for scheduling analysis. We choose NuSMV as the model-checker. CCSL specifications are translated into NuSMV models, conditions of category classification and latency-insensitiveness are expressed as CTL formulas and checked upon the NuSMV models.

CCSL to NuSMV The translation is a direct mapping from ccLTS to FSM. It requires the state space of the specification to be finite. So we demand that if a constraint that may cause infinite state space (`faster than`, `sup` and `inf`), is contained, there must be a `boundedDiff` restricting finite drifts. Clocks and their termination predicts are declared as boolean variables, for instance c : *boolean*, *c_term* : *boolean*, and they are evaluated during the execution according to the ccLTS semantics, $c = TRUE$ means that c ticks in that step and $c\_term \rightarrow c_1\_term$ corresponds to $Term(c) \Rightarrow Term(c_1)$ in ccLTS. For translation details, please check our technique report.

Schedulability classification It is not convenient to establish the schedulability category from definition 5 directly since it requires to build all maximal runs. So we consider it in an equal but more easy checking way, through clock halts.

Definition 7. A clock $c \in Clocks$ *is halted* at state $s \in S$ iff it cannot tick anymore since s is not properly terminated at s, *i.e.,* $\exists Term(c) = 0$ s.t. $\dagger(s) = 1$ and $\forall t \in Reach(s), \forall s' \in S$,if $t \xrightarrow{C} s'$, then $c \notin C$. A state is called *clockhalt* if some clocks are halted at this state.

The classification of schedulability amounts to the reachability problem of clockhalt states, which are formulated as $ds : \bigvee_{c \in Clocks}(c.dead = FALSE \wedge AG\ c = FALSE)$. A specification falls into: *category 1* iff no reachable clockhalt state exists, formulated as $AG\ \neg ds$; *category 2* iff clock halt is inevitable $AFds$; *category 3* if it does not satisfy the above two conditions, that is some runs can reach clockhalt states eventually while others do not.

Latency-insensitiveness checking Note that latency-insensitiveness is not preserved by composition. Specification $S1$ is latency-insensitive, while adding a latency-insensitive constructor $|c| \leq 1$, we get $S2$ which is not latency-insensitive. It is also possible to build a latency-insensitive specification from specifications which are not. For instance, $S5$ is not latency-insensitive. By adding a `strictPre` b, we get $S3$ which is latency-insensitive. And unlike the weakly endochrony in SIGNAL[22] which checks the property on state-less abstractions, latency-insensitive is state dependent. Checking it from the syntactic constructors in a compositional way is not straightforward. So, we define an easy check sufficient condition for it. Please check its proof in our report.

Theorem 2. A specification is latency-insensitive if:
1. Constraints `upto, exclusion` are not used.
2. Input clocks for `union`, `intersect`, `concat`, and `sampledOn` are exclusive.
3. Any two clocks c_1 and c_2 that are involved in one clock relation or are input clocks of one clock function satisfies: for each reachable state s in its ccLTS, $\forall s_1, s_2 \in S$ that $s \xrightarrow{C_1} s_1, s \xrightarrow{C_2} s_2$, if $c_1 \in C_1 \wedge c_2 \in C_2 \wedge c_2 \notin C_1$, then $\exists s' \in S$ that $s_1 \xrightarrow{C_3} s'$, $c_2 \in C_3$ and $c_1 \notin C_3$.

Condition 1 and condition 2 can be easily checked by static analysis. Condition 3 is checked by the following CTL formulas for each pair of clocks that are involved in one clock relation or as input clocks of one clock expression.

$$AG((EX(c_1 = true \& c_2 = false) \& EX(c_2 = true)))$$
$$\rightarrow (AX((c_1 = true \& c_2 = false) \rightarrow EX(c_2 = true \& c_1 = false)))$$
$$\& \ AG((EX(c_1 = false \& c_2 = true) \& EX(c_1 = true))$$
$$\rightarrow (AX((c_1 = false \& c_2 = true) \rightarrow EX(c_1 = true \& c_2 = false))))$$

VII. Conclusion and future work

In this paper, we establish a scheduling library for CCSL to easy its use in scheduling modeling and analysis. We provide a state-based semantics ccLTS for it. Based on that, we study how a general translation expanding the semantics into an adequate model and provide a generic way to do schedulability analysis. With the consideration of some typical phenomenons during our practice, we define the schedulability categories and latency-insensitive property in CCSL scheduling. We believe the relevance of these properties may beyond the specific context, to other cases where a global total order (as a valid schedule) needs to be calculated form local partial orders. In the future, we may develop this work in several directions. First, a strict subset of primitive clock constraints or some patterns of compositions of clock constraints could be recognized; then a simple syntactic restriction can be used as an efficient way to build a latency-insensitive specification. Second, we could trim the state space of a specification, then the remained part would contain valid schedules only. Third, we are going to check the efficiency of our methods by case studies. Finally, our current transformation into NuSMV models requires the assumption that the specification only uses operators with bounded drift, *i.e.,* operators represented with finite state automaton. Safety analysis [24] allows deciding statically that a CCSL specification is actually bounded (finite). Using a similar technique would allow a greater class of CCSL specifications to be handled with our technique.

Acknowledgments

We would like to thank the reviewers of this paper for their valuable comments. This work is partially supported by the projects funded by the 973 Program 2009CB320702, NSFC 91318301 and 61170084, Zuohua Ding is supported by the NSFC No.61170015. The Shanghai Key Lab is supported by Shanghai Knowledge Service Platform ZF 1213 and 863 Project 2011AA010101.

References

[1] Charles André. Syntax and semantics of the clock constraint specification language (CCSL). Technical Report RR-6925, INRIA, 2009.

[2] Su-Young Lee, Frédéric. Mallet, and Robert de Simone. Dealing with AADL end-to-end flow latency with UML MARTE. In *13th IEEE Int. Conf. on Engineering of Complex Computer Systems (ICECCS)*, pages 228–233, 2008.

[3] Frédéric Mallet and Robert de Simone. MARTE vs. AADL for discrete-event and discrete-time domains. In *Languages for Embedded Systems and their Applications*, volume 36 of *Lecture Notes in Electrical Engineering*, pages 27–41. 2009.

[4] Frédéric Mallet, Marie-Agnès Peraldi-Frati, and Charles André. Marte CCSL to execute East-ADL timing requirements. In *IEEE Int. Symp. on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, pages 249–253, 2009.

[5] Marie-Agnès Peraldi-Frati and Yves Sorel. From high-level modelling of time in MARTE to real-time scheduling analysis. *ACESMB 2008*, page 129.

[6] Marie-Agnes Peraldi-Frati and Julien DeAntoni. Scheduling multi clock real time systems: From requirements to implementation. In *ISORC*, pages 50–57, Washington DC, USA, 2011.

[7] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[8] Edward A. Lee and Alberto Sangiovanni-vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17:1217–1229, 1998.

[9] Luca P. Carloni, Kenneth L. McMillan, and Alberto L. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 20(9):1059–1076, 2001.

[10] Ling Yin, Frédéric Mallet, and Jing Liu. Verification of MARTE/CCSL time requirements in Promela/SPIN. In *16th IEEE Int. Conf. on Engineering of Complex Computer Systems (ICECCS)*, pages 65–74, 2011.

[11] Frédéric Mallet. Automatic generation of observers from MARTE/CCSL. In *23rd IEEE Int. Symp. on Rapid System Prototyping (RSP)*, pages 86–92, 2012.

[12] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times: A tool for schedulability analysis and code generation of real-time systems. In *Formal Modeling and Analysis of Timed Systems*, volume 2791 of *Lecture Notes in Computer Science*, pages 60–72. Springer Berlin Heidelberg, 2004.

[13] Yasmina Abdedda˙ Scheduling with timed automata. *Theoretical Computer Science*, 354(2):272 – 300, 2006.

[14] G. Bucci, A. Fedeli, L. Sassoli, and E. Vicario. Modeling flexible real time systems with preemptive time petri nets. In *15th Euromicro Conference on Real-Time Systems*, pages 279 – 286, 2003.

[15] D. Lime and O.H. Roux. A translation based method for the timed analysis of scheduling extended time petri nets. In *25th IEEE International Symposium on Real-Time Systems*, pages 187 – 196, 2004.

[16] Anna Philippou, Insup Lee, and Oleg Sokolsky. Pads: An approach to modeling resource demand and supply for the formal analysis of hierarchical scheduling. *Theoretical Computer Science*, 413(1):2 – 20, 2012.

[17] Calin Glitia, Julien DeAntoni, Frédéric Mallet, Jean-Vivien Millo, Pierre Boulet, and Abdoulaye Gamatié. Progressive and explicit refinement of scheduling for multidimensional data-flow applications using UML MARTE. *Design Automation for Embedded Systems*, 16(2):137–169, 2012.

[18] Calin Glitia, Julien DeAntoni, and Frédéric Mallet. Logical time @work: Capturing data dependencies and platform constraints. In *System Specification and Design Languages*, volume 106 of *Lecture Notes in Electrical Engineering*, pages 223–238. Springer New York, 2012.

[19] Robbin Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[20] Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Readings in hardware/software co-design. chapter Ptolemy: a framework for simulating and prototyping heterogeneous systems, pages 527–543. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[21] Gilles Kahn. The semantics of a simple language for parallel programming. In *IFIP74*, pages 471–475. North-Holland, 1974.

[22] Dumitru Potop-Butucaru, Robert de Simone, Yves Sorel, and Jean-Pierre Talpin. From concurrent multi-clock programs to deterministic asynchronous implementations. In *ACSD*, pages 42–51, 2009.

[23] Charles André, Julien DeAntoni, Frédéric Mallet, and Robert de Simone. The time model of logical clocks available in the OMG MARTE profile. In *Synthesis of Embedded Software*, pages 201–227. Springer, 2010.

[24] Frédéric Mallet, Jean-Vivien Millo, and Robert de Simone. Safe ccsl specifications and marked graphs. In *MEMOCODE*, pages 1–10, 2013. To appear.