Science of Computer Programming 154 (2018) 42–60

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Periodic scheduling for MARTE/CCSL: Theory and practice

Min Zhang [a,b], Feng Dai [a,b], Frédéric Mallet [b,c,d,*]

[a] Shanghai Key Laboratory of Trustworthy Computing, ECNU, Shanghai, China
[b] MoE International Joint Lab of Trustworthy Software, ECNU, Shanghai, China
[c] Université Cote d'Azur, CNRS, I3S, France
[d] INRIA Sophia Antipolis Méditerranée, France

ARTICLE INFO

Article history:
Received 1 May 2016
Received in revised form 7 July 2017
Accepted 25 August 2017
Available online 1 September 2017

Keywords:
Periodic scheduling
MARTE/CCSL
Maude
Rewriting logic
Model checking

ABSTRACT

The UML profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) is used to design and analyze real-time and embedded systems. The Clock Constraint Specification Language (CCSL) is a companion language for MARTE. It introduces logical clocks as first class citizens as a way to formally specify the expected behavior of models, thus allowing formal verification. CCSL describes the expected infinite behaviors of reactive embedded systems. In this paper we introduce and focus on the notion of periodic schedule to allow for a nice finite abstraction of these infinite behaviors. After studying the theoretical properties of those schedules we give a practical way to deal with them based on the executable operational semantics of CCSL in rewriting logic with Maude. We also propose an algorithm to find automatically periodic schedulers with the proposed sufficient condition, and to perform formal analysis of CCSL constraints by means of customized simulation and bounded LTL model checking.

1. Introduction

Lamport's logical clocks [10] originated from the need to synchronize distributed systems without assuming a reliable single common timing mechanism to compare events. Indeed, maintaining a global clock in a widely distributed system may be very costly. Rather than maintaining a total order on events, a partial view was proposed as sufficient to maintain causal relationships.

In a very different community, synchronous languages [3,2,21] use the word clock to emphasize a multiform notion of time where the notion of physical time is initially relaxed with the notion of ordering. The main important difference is that synchronous languages accept instantaneous causal relations, which are an important abstraction in synchronous circuits but are less relevant in widely distributed environments.

The Clock Constraint Specification Language (CCSL) [1,13] proposes a concrete syntax to handle logical clocks as first-class citizens. While synchronous languages mainly focus on signals and values and use logical clocks as a controlling mechanism, CCSL discards the values and only focuses on clock-related issues. In this paper, we focus on the periodic scheduling of CCSL from both theoretical and practical perspectives, considering that reactive embedded systems have recurrent behaviors for which the design of correct periodic schedulers is very important in the development of such systems [12,6]. While deciding on the existence of a schedule for a given set of CCSL constraints is still an open problem, we discuss here a

* Corresponding author at: MoE International Joint Lab of Trustworthy Software, ECNU, Shanghai, China.
E-mail addresses: zhangmin@sei.ecnu.edu.cn (M. Zhang), fdai_itlogic@163.com (F. Dai), Frederic.Mallet@unice.fr (F. Mallet).

http://dx.doi.org/10.1016/j.scico.2017.08.015
0167-6423/

sufficient condition for having a class of bounded schedules which can be extended to infinite but periodic ones. In our earlier work [26], we considered a pragmatic point of view and a very restrictive condition for the existence of periodic schedules. We propose here a much less restrictive condition, that (1) allows to find more periodic schedules, (2) is less restrictive on the condition to find a periodic schedule, and (3) considers a bigger subset of CCSL constraints. About (3), we consider specifically the *periodic filter* of CCSL, which offers a generic notion of periodicity on logical clocks that generalizes the classical definition of periodic activation. In this paper, we prove the correctness of our condition and present an operational method for building periodic schedules from a bounded schedule that satisfies the condition.

Furthermore, we also give a formal executable operational semantics of the extended CCSL language using Maude [4], an algebraic language based on rewriting logic. The formal operational semantics of CCSL was initially defined in a research report [1] in a bid to provide a reference semantics for building simulation tools, like TimeSquare [7]. Maude provides various formal analysis approaches, such as simulation, state space exploration (exhaustive or bounded, symbolic or explicit-state), by which we can analyze CCSL specifications to, for instance, check the existence of desired schedules with specific properties (e.g., reduce memory usage), produce a schedule or simulation by applying customized scheduling policies, and verify the satisfiability of expected properties.

The benefits of the new semantics defined in rewriting logic are multifold. The first benefit is that rewriting logic gives a direct implementation of the operational semantics while TimeSquare provides a Java-based implementation which is prone to introduce unexpected complexity. The second and most important benefit is that we can directly use the tools that are provided for rewriting logic to analyze a CCSL specification by means of simulation, state-space exploration, and even linear temporal logic model checking. Previous work on studying CCSL properties [14] relies on several intermediate transformations to automata and other specific formats so that model-checking becomes possible when a CCSL specification is finite. A CCSL specification is called finite if it can be transformed into a finite-state automaton [15]. However, some CCSL operators, which are called unsafe operators, cannot be transformed into finite-state automata. It either meant reducing to a safe subset of CCSL [9] or detecting that the specification was describing a finite reachable state-space even though relying on unsafe operators. In this contribution, we rely on the Maude environment [4] to provide a direct analysis support to CCSL specifications by formally defining its operational semantics in Maude, and we can explore unsafe specifications using bounded model checking and do not restrict to the safe subset.

For periodic scheduling, we provide a prototype implementation in Maude based on the new formal semantics of CCSL to detect the proposed conditions and build the satisfying schedule. As another contribution, we propose five arbitration policies to reduce the set of possible solutions when a CCSL specification is under-specified. Such arbitration policies can be seen as optimization criteria akin to those classically used in real-time scheduling to optimize memory or bandwidth. Those arbitration policies can also be naturally implemented in Maude based on the new formal semantics.

This paper is an extended version of our previous work [26]. Apart from adding much more detail on the formal semantics of CCSL, this extended version adds the following contributions:

1. We consider an operator for the CCSL language that was ignored in previous work, the so-called *periodic filter*, which serves for the specification of logical repetitive patterns between logical clocks.
2. We propose a less constraining sufficient condition for the existence of periodic schedules and give a formal proof of its correctness.
3. We present a formal executable operational semantics of the extended CCSL language in Maude and illustrate its applications to various formal analysis tasks, such as checking the existence of bounded and periodic schedules, deriving a customized simulation and performing LTL model-checking.
4. More user-defined arbitration policies are supported for customized simulations of schedules.

The rest of this paper is organized as follows. Section 2 introduces CCSL and the notion of schedule with bounded and periodic restrictions. It also gives conditions for being able to build a periodic schedule, and we prove that those conditions are sufficient. Section 3 gives a brief introduction to Maude. Section 4 discusses the encoding of the semantics of CCSL in Maude and details the way its environment can be used to compute bounded and periodic schedules. Section 5 considers several examples to illustrate the interest of this encoding and the usefulness of our tool. Finally, Section 6 compares this work to previous work, including our own, and Section 7 gives some concluding remarks.

2. The clock constraint specification language with periodic filter

2.1. Syntax and semantics of CCSL

CCSL relies on the notion of logical clocks, which are commonly used to express partial orders in distributed systems [10] or synchronization conditions in synchronous languages [2]. We use the wording clock or logical clock indistinctly in the following. While traditional synchronous languages give a syntax to combine signals, infinite sequences of values, and use clocks to express when the signals are present (have a value), CCSL discards the values on purpose to focus on relationships among clocks.

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

1. $\delta \models c_1 \prec c_2$	$\iff \forall n \in \mathbb{N}^+.\chi(c_2,n) = \chi(c_1,n) \Rightarrow c_2 \notin \delta(n)$	(Precedence)
2. $\delta \models c_1 \preceq c_2$	$\iff \forall n \in \mathbb{N}^+.\chi(c_1,n) \geq \chi(c_2,n)$	(Causality)
3. $\delta \models c_1 \subseteq c_2$	$\iff \forall n \in \mathbb{N}^+.c_1 \in \delta(n) \Rightarrow c_2 \in \delta(n)$	(Subclock)
4. $\delta \models c_1 \# c_2$	$\iff \forall n \in \mathbb{N}^+.c_1 \notin \delta(n) \vee c_2 \notin \delta(n)$	(Exclusion)
5. $\delta \models c_1 \triangleq c_2 + c_3$	$\iff \forall n \in \mathbb{N}^+.(c_1 \in \delta(n) \iff c_2 \in \delta(n) \vee c_3 \in \delta(n))$	(Union)
6. $\delta \models c_1 \triangleq c_2 \times c_3$	$\iff \forall n \in \mathbb{N}^+.(c_1 \in \delta(n) \iff c_2 \in \delta(n) \wedge c_3 \in \delta(n))$	(Intersection)
7. $\delta \models c_1 \triangleq c_2 \wedge c_3$	$\iff \forall n \in \mathbb{N}^+.\chi(c_1,n) = max(\chi(c_2,n),\chi(c_3,n))$	(Infimum)
8. $\delta \models c_1 \triangleq c_2 \vee c_3$	$\iff \forall n \in \mathbb{N}^+.\chi(c_1,n) = min(\chi(c_2,n),\chi(c_3,n))$	(Supremum)
9. $\delta \models c_1 \triangleq c_2 \$ d$	$\iff \forall n \in \mathbb{N}^+.\chi(c_1,n) = max(\chi(c_2,n) - d, 0)$	(Delay)
10. $\delta \models c_1 \triangleq c_2 \propto p$	$\iff \forall n \in \mathbb{N}^+.c_1 \in \delta(n) \iff c_2 \in \delta(n) \wedge \exists m \in \mathbb{N}^+.\chi(c_2,n) = m \times p - 1$	(Periodicity)

Fig. 1. Definition of the 10 primitive CCSL operators.

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. CCSL gives a syntax to define such relationships. 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 behavior of a system.

CCSL constraints express some relationships between clocks, and their underlying ticks. One possible behavior 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.

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 $\mathbb{N}^+$, $\delta(n) \neq \emptyset$.

$\mathbb{N}^+$ denotes the set of all the non-zero natural numbers. By the condition $\delta(n) \neq \emptyset$ in Definition 2 we exclude from schedules those *trivial* steps where there is no clock ticking.

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 3 *(History).* A 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 $n \in \mathbb{N}^+$:

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

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. 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.

In CCSL, there are four primitive constraint operators which are binary relations between clocks, and five kinds of clock definitions [13]. The four constraint operators are called *precedence*, *causality*, *subclock* and *exclusion*; and the five clock definitions are called *union*, *intersection*, *infimum*, *supremum*, and *delay*. For simplicity, we call all of them CCSL constraints in this paper. Besides the nine existing primitive operators, we consider a new one called *periodic filter*. The purely synchronous operators (*subclock*, *exclusion*, *union*, *intersection*) rely only on the notion of schedule. The other ones rely on history.

Fig. 1 shows the definition of the satisfiability of a constraint ϕ with regards to a schedule δ, which is denoted by $\delta \models \phi$. For example, given two clocks c_1 and c_2, $\delta \models c_1 \prec c_2$ holds if and only if for each n in $\mathbb{N}^+$, c_2 must not tick at step n if the number of ticks of c_1 is equal to the one of c_2 up to step n (not including step n). Intuitively, 'c_1 precedes c_2' means that c_1 must always tick earlier than c_2. *Precedence* and *causality* are asynchronous constraints and they forbid clocks to tick depending on what has happened on other clocks in the earlier steps. *Subclock* and *exclusion* are synchronous constraints and they force clocks to tick depending on whether another clock ticks or not in the same step. *Union* defines a clock c_1 which ticks whenever either c_2 or c_3 ticks; intersection defines a clock c_1 which ticks whenever both c_2 and c_3 tick; *supremum* defines the slowest clock c_1 which is faster than both c_2 and c_3; *infimum* defines the fastest clock c_1 which is slower than both c_2 and c_3; and *delay* defines the clock c_1 which is delayed by c_2 with d steps. *Periodic filter* is used to define a clock, e.g., c_1 in the figure, which ticks once periodically every pth tick of another clock c_2. Note that we need

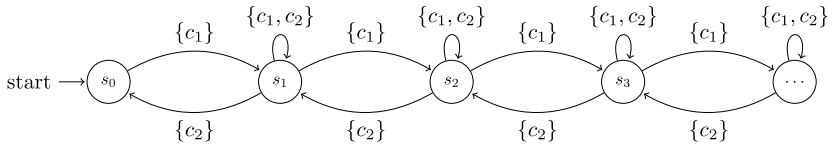

Fig. 2. State-based representation of the schedules satisfying $c_1 \prec c_2$.

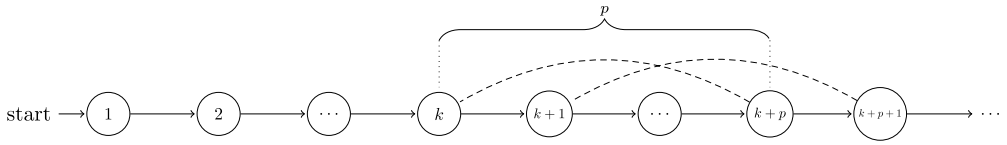

Fig. 3. Periodic schedule.

$m \times p - 1$ because the tick of c_2 at step n is not counted in the history $\chi(c_2, n)$. By construction, c_1 is a subclock of c_2 and is slower than c_2.

Subclock, exclusion, union and intersection are also called stateless constraints since the tick of a clock at a step only depends on the ticks of other clocks at the same step, while other constraints are stateful, when the ticks of a clock depend on the ticks of other clocks at previous steps.

Given a set Φ of clock constraints and a schedule δ, δ satisfies Φ (denoted by $\delta \models \Phi$) if $\delta \models \phi$ for each ϕ in Φ. We use $\delta; k \models \phi$ to denote that δ satisfies ϕ at step k ($k \in \mathbb{N}^+$). That is, $\delta; k \models \phi$ holds if and only if the counterpart formula of ϕ in Fig. 1 is true after the universally quantified variable n in the formula is eliminated by being instantiated with k. We use $\delta; k \models \Phi$ to denote that δ satisfies all the constraints in Φ at step k, i.e., $\forall \phi \in \Phi. \delta; k \models \phi$.

2.2. Bounded and periodic schedules

Given a set Φ of CCSL constraints, deciding if there exists at least one schedule that satisfies Φ is still an open problem. Intuitively, the satisfying schedule may be an infinite sequence of finite valuations, so it may require to explore an infinite number of states. For example, Fig. 2 shows a state-based representation of the schedules that satisfy the constraint $c_1 \prec c_2$ [14]. An arrow from one state to another represents one step and the label of the arrow is the set of all the clocks that tick at this step. An infinite path from the initial state s_0 represents a schedule. For precedence, it needs an infinite number of states to represent all the possible schedules [14]. In the work [23], such constraint is called *divergent*.

Finding a periodic schedule is a way to have a finite abstraction to characterize this infinite state space. We consider two special kinds of schedules called *bounded schedule* and *periodic schedule* from a pragmatic point of view. We show that the existence of a bounded schedule for a set of CCSL constraints among those introduced in the previous subsection is decidable. Additionally, we propose a simple sufficient condition for the existence of a periodic schedule.

Definition 4 *(Bounded schedule).* Given a set Φ of clock constraints on clocks in C and a function $\delta : \mathbb{N}^+_{\leq n} \to 2^C$, δ is called an n-bounded schedule of Φ if for each $0 < i \leq n$, $\delta; i \models \Phi$.

$\mathbb{N}^+_{\leq n}$ denotes the set of all the non-zero natural numbers that are less than or equal to n. The satisfiability of an n-bounded schedule with respect to a set Φ of constraints is denoted by $\delta \models_n \Phi$. It is obvious that given a bound n it is a decidable problem to check if there exists an n-bounded schedule for a set of CCSL constraints because the number of candidate schedules is finite, i.e., $(2^{|C|} - 1)^n$, where $|C|$ denotes the number of clocks in C. It is also obvious that there is no schedule that satisfies a set Φ of clock constraints if there does not exist an n-bounded schedule for Φ, although not vice versa. The existing tool TimeSquare [7] for the analysis of CCSL constraints can be considered as a bounded schedule solver because it builds a schedule up to a given number of steps and stops whenever it cannot progress anymore.

Bounded schedules are sometimes too restrictive in practice because we usually do not assign a bound to clocks in real-time embedded systems and assume that reactive systems can run forever and only terminate when shut down. Thus, clocks should tick infinitely often from the theoretical point of view. We identify another class of schedules which are unbounded and force all the clocks to occur periodically. We call them *periodic schedules*.

Definition 5 *(Periodic schedule).* A schedule δ is periodic if there exist k, p in $\mathbb{N}^+$ such that for each $k' \in \mathbb{N}^+$ and $k' \geq k$, $\delta(k' + p) = \delta(k')$.

Fig. 3 depicts a periodic schedule whose period is p. Each node denotes a time point, and each plain arrow denotes the elapse of a time unit. The dashed line indicates that, any clock ticking at either end of line also ticks at the other end. From step k, the schedule starts to repeat every pth step infinitely.

It is also an open problem to decide whether there exists a periodic schedule for a given arbitrary set of clock constraints. In the rest of this section, we propose a sufficient condition for the existence of a periodic schedule, and an approach to construct periodic schedules based on bounded ones.

Theorem 1. *Given a schedule $\delta : \mathbb{N}^+ \to 2^C$, a clock constraint ϕ, and two natural numbers k, k' with $k' > k$, then $\delta \models_k \phi \Rightarrow \delta; k' \models \phi$ if all the following five conditions hold:*

1. $\delta(k') = \delta(k)$;
2. *If ϕ is in form of $c_1 \prec c_2$ or $c_1 \preccurlyeq c_2$, then $\chi(c_1, k') - \chi(c_1, k) \geq \chi(c_2, k') - \chi(c_2, k)$;*
3. *If ϕ is in form of $c_1 \triangleq c_2 \$ d$, then $\chi(c_2, k) \geq d$ and $\chi(c_1, k') - \chi(c_1, k) = \chi(c_2, k') - \chi(c_2, k)$;*
4. *If ϕ is in form of $c_3 \triangleq c_1 \wedge c_2$ or $c_3 \triangleq c_1 \vee c_2$, then $\chi(c_1, k') - \chi(c_1, k) = \chi(c_2, k') - \chi(c_2, k) = \chi(c_3, k') - \chi(c_3, k)$;*
5. *If ϕ is in form of $c_1 \triangleq c_2 \propto p$, then $c_1 \in \delta(k') \iff (\chi(c_2, k') + 1) \mod p = 0$.*

Intuitively, condition 1 says that the clocks that tick at step k are the same as those at step k'; condition 2 says that the number of ticks of clock c_1 must be greater than or equal to the one of clock c_2 in a period if they are constrained by causality or precedence; and 3 and 4 say that the number of ticks of clocks that are constrained by a *delay*, infimum or supremum must be the same in a period. Note that we do not care about what has happened to clocks only constrained by stateless constraints. In addition, if c_1 and c_2 are constrained by a *delay*, the number of ticks of c_2 up to step k must be greater than or equal to d, as required by condition 3.

Proof. By case analysis:

The stateless (synchronous) constraints only use condition 1, we start with them.

1. $\phi \equiv c_1 \subseteq c_2$: $\delta; k \models c_1 \subseteq c_2$ implies that $c_1 \in \delta(k) \Rightarrow c_2 \in \delta(k)$. Obviously, we have $c_1 \in \delta(k') \Rightarrow c_2 \in \delta(k')$. Thus, $\delta; k' \models c_1 \subseteq c_2$.
2. $\phi \equiv c_1 \triangleq c_2 + c_3$: $\delta; k \models c_1 = c_2 + c_3$ means that $c_1 \in \delta(k) \iff c_2 \in \delta(k) \vee c_3 \in \delta(k)$. It is obvious that $c_1 \in \delta(k') \iff c_2 \in \delta(k') \vee c_3 \in \delta(k')$, and hence $\delta; k' \models c_1 \triangleq c_2 + c_3$.
3. $\phi \equiv c_1 \triangleq c_2 * c_3$: $\delta; k \models c_1 = c_2 * c_3$ means that $c_1 \in \delta(k) \iff c_2 \in \delta(k) \wedge c_3 \in \delta(k)$. It is obvious that $c_1 \in \delta(k') \iff c_2 \in \delta(k') \wedge c_3 \in \delta(k')$, and hence $\delta; k' \models c_1 \triangleq c_2 * c_3$.
4. $\phi \equiv c_1 \# c_2$: $\delta; k \models c_1 \# c_2$ means that $c_1 \notin \delta(k) \vee c_2 \notin \delta(k)$. Obviously, we have $c_1 \notin \delta(k') \vee c_2 \notin \delta(k')$, and hence $\delta; k' \models c_1 \# c_2$.
5. $\phi \equiv c_1 \prec c_2$: $\delta; k \models c_1 \prec c_2$ implies that $\chi_\delta(c_1, k) = \chi_\delta(c_2, k) \Rightarrow c_2 \notin \delta(k)$. It is obvious that $\chi_\delta(c_1, k) = \chi_\delta(c_2, k) \iff \chi_\delta(c_1, k') = \chi_\delta(c_2, k')$
   - If $\chi_\delta(c_1, k) \neq \chi_\delta(c_2, k)$, there must be $\chi_\delta(c_1, k) > \chi_\delta(c_2, k)$ because precedence implies causality [14], and hence $\chi_\delta(c_1, k') > \chi_\delta(c_2, k')$ by condition 2. Consequently, we have $\delta; k' \models c_1 \prec c_2$.
   - If $\chi_\delta(c_1, k) = \chi_\delta(c_2, k)$, then $c_2 \notin \delta(k)$, which implies $c_2 \notin \delta(k')$ because $\delta(k) = \delta(k')$ by condition 1. Thus, $\delta; k' \models c_1 \prec c_2$
6. $\phi \equiv c_1 \triangleq c_2 \$ d$: $\delta; k \models c_1 \triangleq c_2 \$ d$ implies that $\chi(c_1, k) = max(\chi(c_2, k) - d, 0)$. Because $\chi(c_2, k) \geq d$, there is $\chi(c_1, k) = \chi(c_2, k) - d$. We have $\chi(c_1, k) + n = \chi(c_2, k) + n - d$ for each n in $\mathbb{N}$. Let $n = \chi(c_1, k') - \chi(c_1, k)$. Thus, $\chi(c_1, k') = \chi(c_2, k') - d$ by condition 3. Because $n \geq 0$, we have $\chi(c_2, k') - d \geq 0$. Thus, $\chi(c_1, k') = max(\chi(c_2, k') - d, 0)$. Namely, $\delta; k' \models c_1 \triangleq c_2 \$ d$.
7. $\phi \equiv c_1 \preccurlyeq c_2$: $\delta; k \models c_1 \preccurlyeq c_2$. Namely, $\chi(c_1, k) \geq \chi(c_2, k)$. Since $\chi(c_1, k') - \chi(c_1, k) \geq \chi(c_2, k') - \chi(c_2, k)$ then $\chi(c_1, k) + (\chi(c_1, k') - \chi(c_1, k)) \geq \chi(c_2, k) + (\chi(c_2, k') - \chi(c_2, k))$, i.e., $\chi(c_1, k') \geq \chi(c_2, k')$. Thus, $\delta; k' \models c_1 \preccurlyeq c_2$.
8. $\phi \equiv c_3 \triangleq c_1 \wedge c_2$: $\delta; k \models c_3 \triangleq c_1 \wedge c_2$ means that $\chi(c_3, k) = max(\chi(c_1, k), \chi(c_2, k))$. Thus, we have $\chi(c_3, k) + m = max(\chi(c_1, k) + m, \chi(c_2, k) + m)$ for each m in $\mathbb{N}$. Let $m = \chi(c_1, k') - \chi(c_1, k)$. By condition 4, we have $\chi(c_3, k') = max(\chi(c_1, k'), \chi(c_2, k'))$, and hence $\delta; k' \models c_3 \triangleq c_1 \wedge c_2$.
9. $\phi \equiv c_3 \triangleq c_1 \vee c_2$: $\delta; k \models c_3 \triangleq c_1 \vee c_2$ means that $\chi(c_3, k) = min(\chi(c_1, k), \chi(c_2, k))$ for each m in $\mathbb{N}$. Let $m = \chi(c_1, k') - \chi(c_1, k)$. By condition 4, we have $\chi(c_3, k') = min(\chi(c_1, k'), \chi(c_2, k'))$, and hence $\delta; k' \models c_3 \triangleq c_1 \vee c_2$.
10. $\phi \equiv c_1 \triangleq c_2 \propto p$: the proof is straightforward by condition 5 and the definition of periodic filter.

The proof is done. □

We explain by an example that the conditions in Theorem 1 are less constraining than those defined in our previous work [26]. Fig. 4 shows three schedules for the precedence constraint $c_1 \prec c_2$. All three are periodic schedules according to Definition 5. For instance, we can find $k = 1$ and $p = 2$ for schedule (a), and $k = 1$ and $p = 4$ for schedule (b) such that k and p satisfy Definition 5. Schedule (a) and (b) also satisfy the condition in Theorem 1 with $k = 1, k' = 3$ and $k = 1, k' = 5$ respectively. It is also obvious that schedule (c) also satisfies Definition 5 with $k = 1, p = 1$ and the conditions in Theorem 1 with $k = 1, k' = 2$. By schedule (c), we have $\chi(c_1, k') - \chi(c_1, k) > \chi(c_2, k') - \chi(c_2, k)$ with $k = 1, k' = 2$. Schedule (c) does not satisfy the conditions defined in the work [26], where all the clocks are required to tick the same number of ticks from k to k'. In that sense, the conditions in Theorem 1 are less constraining and provide an important relaxation to find practical solutions.

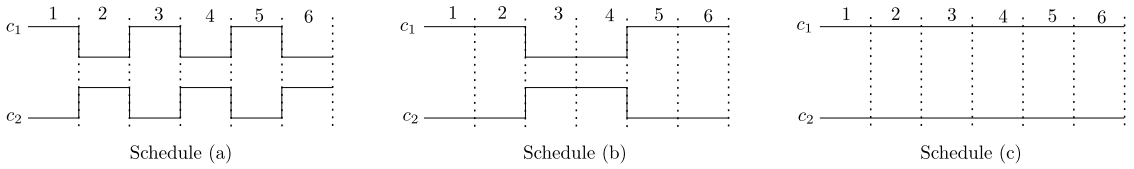

Fig. 4. Three schedules that satisfy the precedence constraint $c1 \prec c2$.

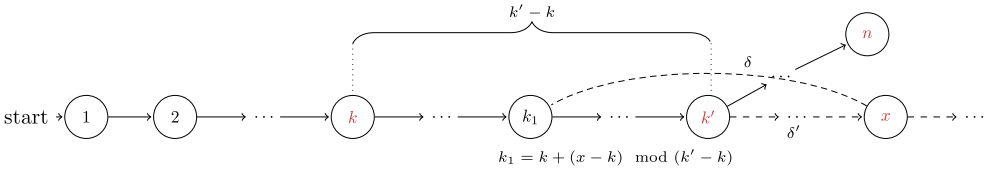

Fig. 5. Construction of periodic schedule δ' from an n-bounded schedule δ.

Corollary 1. *Given a schedule $\delta : \mathbb{N}^+ \to 2^C$, a set Φ of clock constraints, and two natural numbers $k, k'(k' > k)$, $\delta \models_k \Phi \Rightarrow \delta; k' \models \Phi$ if each constraint ϕ in Φ satisfies all the five conditions in Theorem 1.*

Given an n-bounded schedule δ of a set Φ of clock constraints, if there exist two natural numbers $k, k' \leq n$, which satisfy the five conditions in Theorem 1, we can define a periodic schedule δ' by extending δ such that δ' satisfies Φ.

$$\delta'(x) = \begin{cases} \delta(x) & \text{if } x \leq k' \\ \delta(k + (x - k) \mod (k' - k)) & \text{if } x > k' \end{cases}$$

Fig. 5 shows the construction of δ' based on δ. From k', the schedule δ' repeats infinitely the steps from k to $k' - 1$. We next prove that $\delta' \models \Phi$. By definition of δ', it is obvious that for all $k'' \leq k'$ there is $\delta'; k'' \models \Phi$. For all k'' such that $k'' > k'$, let $k_1 = k + (k'' - k) \mod (k' - k)$. Obviously, $k_1 < k'$, and hence $\delta'; k_1 \models \Phi$. By definition of δ', we have $\delta'(k'') = \delta(k_1)$. Because k and k' satisfy the five conditions, so do k_1 and k''. By Corollary 1, we have $\delta'; k'' \models \Phi$ for all k'' in $\mathbb{N}^+$ such that $k'' > k'$. Thus, we have $\delta' \models \Phi$.

Note that the five conditions in Theorem 1 are independent, which means that given a set Φ of CCSL constraints, if two clocks c_1 and c_2 are constrained by different stateful constraints the number of ticks from step k to k' of clock c_1 is not necessarily the same as the one of clock c_2. Namely, by the extended schedule δ', the clocks that are constrained by different stateful constraints in Φ can tick different times in a period. This makes a periodic schedule that is defined in the above-mentioned approach relatively general, compared with those schedules which force all the clocks to tick the same number of times in a period. However, if the stateful constraints *delay*, *infimum* and *supremum* where c_1 and c_2 are referenced are related to each other due to some other clocks, c_1 and c_2 must tick the same number of times in a period. This is straightforward to prove by Theorem 1.

We can also define infinite but non-periodic schedules with the proposed approach. Let us consider two disjoint CCSL constraints ϕ_1 and ϕ_2 such that $clock(\phi_1) \cap clock(\phi_2) = \emptyset$, where $clock : \Phi \to 2^C$ is a function that returns the set of all the clocks in a given constraint. We assume that ϕ_1 is a stateless constraint while ϕ_2 is a stateful one. Because ϕ_1 is stateless, we can find a non-periodic schedule for it and denote it by δ_1. For the stateful constraint ϕ_2, we assume there is a periodic schedule δ_2 found under the proposed sufficient condition. Let δ be a schedule such that for each i in $\mathbb{N}^+$, $\delta(i) = \delta_1(i) \cup \delta_2(i)$. Then, δ must be a schedule of $\{\phi_1, \phi_2\}$. However, δ may not be periodic because δ_1 is non-periodic.

The proposed approach is also suitable to more general cases where there are n sets of CCSL constraints and each set is disjoint from others.

Theorem 2. *Given n sets $\Phi_1, \ldots, \Phi_n$ of CCSL constraints on n sets $C_1, \ldots, C_n$ of clocks where $C_i \cap C_j = \emptyset$ with $1 \leq i < j \leq n$, let $\delta_1, \ldots, \delta_n$ be n schedules of $\Phi_1, \ldots, \Phi_n$ respectively and $\delta : \mathbb{N}^+ \to 2^{C_1 \cup \ldots \cup C_n}$ be a schedule such that $\forall i \in \mathbb{N}^+ . \delta(i) = \bigcup_{j \in n} \delta_j(i)$. Then, δ is a schedule of $\Phi_1 \cup \ldots \cup \Phi_n$.*

We explain the basic idea of the proof of Theorem 2. Let ϕ be a constraint in some $\Phi_i (1 \leq i \leq n)$. We prove $\delta \models \phi$ by case analysis on the form of ϕ. As an example, we consider precedence and assume that $\phi = c_1 \prec c_2$ with clocks c_1 and c_2 in C_i. For each j in $\mathbb{N}^+$, $\chi_{\delta_i}(c_1, j) = \chi_{\delta_i}(c_2, j) \implies c_2 \notin \delta_i(j)$. By definition of δ, there is $c_2 \notin \delta_i(j) \iff c_2 \notin \delta(j)$ and $\chi_\delta(c, j) = \chi_{\delta_i}(c, j)$ for each $c \in \{c_1, c_2\}$ and $j \in \mathbb{N}^+$. That is because $c_1, c_2 \notin C_k$ with $k = 1, \ldots, n$ and $k \neq i$. Thus, for each j in $\mathbb{N}^+$ we have $\chi_\delta(c_1, j) = \chi_\delta(c_2, j) \implies c_2 \notin \delta(j)$, and hence $\delta \models \phi$. Other cases can be proved likewise, and we omit the details in the paper.

By Theorem 2, we can divide a set Φ of constraints into several subsets such that the sets of clocks of these subsets of constraints are disjoint. Then, we search for a schedule for each subset of constraints with which we can define a schedule for Φ.

Definition 6 *(Connected clocks).* Given a set C of clocks and a set Φ of clock constraints, two clocks c and c' in C are *connected*, denoted by $c \asymp c'$, if and only if one of the following two conditions is satisfied:

1. There exists a constraint ϕ in Φ such that $c \in clock(\phi)$ and $c' \in clock(\phi)$, or
2. There exist n clocks ($n > 0$), i.e., $c_1, \ldots, c_n$ in C, such that $c \asymp c_1$, $c_1 \asymp c_2, \ldots, c_n \asymp c'$.

Let $\mathbb{C}$ be a partition of C, i.e., $\mathbb{C} \subseteq 2^C$ is a set of subsets of clocks in C such that:

1. For each C' in $\mathbb{C}$ and two arbitrary clocks c, c' in C', $c \asymp c'$;
2. For two arbitrary subsets C_1, C_2 in $\mathbb{C}$, $C_1 \cap C_2 = \emptyset$, and for each c_1 in C_1 and c_2 in C_2 there is $c_1 \not\asymp c_2$;
3. $C = \bigcup_{i \in |\mathbb{C}|} C_i$;
4. $\emptyset \notin \mathbb{C}$.

Intuitively, $\mathbb{C}$ is a set of disjoint subsets of clocks, and clocks in each subset are connected. A special case is that $\mathbb{C} = \{C\}$ if all the clocks in C are connected.

Given a set Φ of CCSL constraints on a set C of clocks, we assume that $\mathbb{C} = \{C_1, \ldots, C_n\}$ with some n in $\mathbb{N}$, and $\Phi_i \subseteq \Phi$ is the set of all the constraints in Φ which are defined on C_i. If there exists a schedule δ_i for each Φ_i such that $\delta_i \models \Phi_i$, let δ be a schedule such that $\forall j \in \mathbb{N}^+ . \delta(j) = \bigcup_{i \in |\mathbb{C}|} \delta_i(j)$. By Theorem 2, $\delta \models \Phi$ must hold.

3. Maude in a Nutshell

In this section, we introduce some background on Maude that is necessary to understand this paper. Maude is a rewriting-logic-based algebraic specification language and also an efficient rewrite engine [4]. The underlying rewriting logic of Maude is a logic of concurrent change well suited to formalize states and concurrent computations. Maude has been widely used as a logical and semantic framework for the formalization and verification of computer systems and programming languages [19,18,17,20]. Maude specifications are executable, and various formal analysis methods are provided, such as simulation, reachability analysis by state-space exploration, and LTL model checking.

3.1. Formalization in Maude

A *module* is the unit of Maude programs. A module specifies a *logical theory*. In Maude there are two kinds of modules, *functional modules* and *system modules*. A functional module specifies a *membership equational theory*, and a system model specifies a *rewrite theory*. A membership equational theory is a pair of the form $(\Sigma, E \cup A)$, whose underlying logic is called *membership equational logic* [16]. Σ is called a signature, specifying type structures as sorts and subsorts and function symbols; E is a collection of (possibly conditional) equations and memberships defined on Σ; and A is a collection of equational attributes such as associativity, commutativity and identity, declared for those operators in Σ. Equations are used to axiomatize the properties of the operators in Σ. Maude performs equational reduction with the equations E oriented from left to right modulo the axioms A.

A system module specifies a rewrite theory $(\Sigma, E \cup A, R)$, which consists of an underlying membership equational theory $(\Sigma, E \cup A)$ and a set R of rewrite rules, each of which has the form:

$$l : t \longrightarrow t' \text{ if } (\bigwedge_i p_i = q_i) \wedge (\bigwedge_j u_j := v_j) \wedge (\bigwedge_k w_k : s_k) \wedge (\bigwedge_m t_m \longrightarrow t'_m),$$

where l is a *label*, and t, t' are the terms that are constructed by the operators in Σ with some universally quantified variables. A rewrite rule may have conditions. There are four kinds of conditions which can be ordinary equations, matching equations, membership conditions, or rewrite conditions [4]. Mathematically, a matching equation $u_j := v_j$ is interpreted as an ordinary equation. Operationally, it is treated by matching the term e.g., u_j on the left-hand side against the one e.g., v_j on the right-hand side. A matching equation holds if and only if the matching succeeds. In u_j there may be new variables, i.e., the variables that occur in u_j but neither in t nor in any of the conditions before u_j in the rewrite rule. After matching succeeds, these new variables are instantiated by the corresponding subterms in the canonical form of v_j. We omit explanations of membership conditions and rewrite conditions because they are not used in our work. Interested readers can refer to the work [4] for more details. Computationally, given a term t_0 which has a subterm t'_0 that can be matched by t in the above rule with a substitution σ, t_0 can be rewritten into a new term with t'_0 being replaced by the corresponding substitution instance of t', provided that the conditions of the rule hold in the context of the substitution.

When specifying a system in Maude as a rewrite theory, the underlying membership equational theory of the rewrite theory specifies the "static aspects" of the system, i.e., the algebraic structure of the set of system states, and the rules in R specify the "dynamic aspects" of the system, i.e., all the possible transitions that the system can perform. In Maude, a system state is represented as a Σ-term. Transitions among the states are formalized by rewrite rules.

We briefly summarize the syntax of Maude (see [4] for more details). A functional module has the form of `fmod` *ModName* `is` *Decs&Stmts* `endfm`, where *ModName* is a user-defined name of the module and *Mod&Stmts* includes importations of other modules, declarations of sorts and subsort relations, declarations of operators, variables and equations.

Sorts and subsort relations are declared by the keywords `sorts` and `subsort`, respectively. Operators are declared with the `op` keyword in the form: `op` f `:` $s_1 \ldots s_n$ `->` s, where $s_i (i = 1, \ldots, n)$ and s are sorts. Maude allows for user-defined mixfix operators and uses underbars in operators to indicate each of the argument positions. An equation in Maude is declared in the form of `eq` t `=` t', where t and t' are two terms of the same sort. An equation can be conditional, which is declared by the keyword `ceq` and ended with the keyword `if`, followed by a conjunction of conditions. A rewrite rule in Maude is declared in the form of `rl [label]:` t `=>` t'. A conditional rewrite rule is declared by the keywords `crl` and `if` with a conjunction of rewrite conditions.

3.2. Formal analysis in Maude

Maude specifications are executable under certain conditions. Maude provides multiple formal analysis methods, including simulation, reachability analysis, and LTL model checking to formally analyze systems.

Simulation is achieved using Maude's `rewrite` command (abbreviated `rew`), which repeatedly applies the rewrite rules to transform a given term step by step. The transformation process simulates one behavior of the specified system. There may be some rules that can be applied an infinite number of times. In such cases, an upper bound to the rewrite steps is needed to force Maude to stop.

Maude provides the `search` command to explore the reachable state space of specified systems. The syntax of `search` command follows the general scheme:

`search [` n, m `]` `in` *ModName* `:` t_1 `=>*` t_2 `such that` *condition* `.`

where n, m are optional arguments providing a bound on the number of expected solutions and the maximum depth of the search; t_1 is the starting term; t_2 is a pattern which is essentially a term that contains some variables and has the same sort as t_1; and *condition* represents an optional condition which has the same form as the one of conditions in conditional rules. A term t is a solution if t matches against t_2 with a match satisfying the optional condition and there exists a rewriting sequence from t_1 to t. The arrow `=>*` is used to restrict the length of rewriting steps to none, one or more steps. If the arrow `=>!` is used instead of `=>*`, then a solution must be a term that cannot be further rewritten.

Maude has an efficient LTL model checker, supporting model checking of the properties expressed in linear temporal logic. Given a Maude specification of a system and a set of atomic propositions defined on system states, the Maude LTL model checker is invoked by the built-in function `modelCheck`, which takes two arguments, an initial state and an LTL formula, and returns `true` if no counterexample is found, and otherwise a counterexample as a witness to the violation. An LTL formula is built out of atomic propositions that need to be predefined and logical and temporal operators in LTL. A condition of doing LTL model checking in Maude is that the set of states that are reachable from the given initial state is finite.

4. Operational semantics of CCSL and its formalization in Maude

4.1. Operational semantics of CCSL

We define a *clock system*, denoted by $\langle X, \Phi \rangle$, consisting of a configuration X and a set Φ of CCSL constraints. Let C_Φ be the set of all the clocks in Φ. A configuration $X : C_\Phi \to \mathbb{N}$ is a function and for each c in C_Φ, $X(c)$ denotes the current number of ticks of clock c. Let F be a non-empty subset of C_Φ. If the ticking of every clock in F satisfies all the clock constraints in Φ, there is a transition from $\langle X, \Phi \rangle$ to $\langle X', \Phi \rangle$ with respect to F, denoted by $\langle X, \Phi \rangle \xrightarrow{F} \langle X', \Phi \rangle$, where:

$$\forall c \in C_\Phi . X'(c) = \begin{cases} X(c) + 1 & \text{if } c \in F \\ X(c) & \text{if otherwise.} \end{cases} \tag{1}$$

Fig. 6 shows the operational semantics of the 10 types of clock constraints in CCSL. Above a line is the condition of the transition under the line. We take the operational semantics of precedence for example. The rule labeled *Precedence* says that there is a transition from $\langle X, c_1 \prec c_2 \rangle$ to $\langle X', c_1 \prec c_2 \rangle$ with respect to F if the condition $X(c_1) = X(c_2) \Longrightarrow c_1 \notin F$ holds. The condition clearly corresponds to the definition of precedence in Fig. 1. For the definition of infimum, three cases are considered based on the relation between $X(c_2)$ and $X(c_3)$. For instance, when $X(c_2) > X(c_3)$ holds, c_1 is in F if and only if c_2 is in F, as formalized by the rule labeled by *Infimum-I*. Otherwise, $c_1 \triangleq c_2 \wedge c_3$ is violated in the new configuration X'. Two other cases are formalized by *Infimum-II* and *Infimum-III*, respectively. The operational semantics of other constraints are defined likewise.

4.2. Formalization of a clock system

The formalization of a clock system $\langle X, \Phi \rangle$ consists of formalizations of X and Φ. CCSL constraints can be naturally formalized in Maude. We use quoted identifiers to represent clocks in CCSL, e.g., `'c` represents a clock c. We declare a sort `Clock` whose elements are essentially the quoted identifiers that are represented by the sort `Qid` in Maude. In the following Maude module `CONSTRAINTS` two sorts `Constraint` and `Constraints` are declared to represent CCSL constraints

$$\frac{X(c_1) = X(c_2) \implies c_2 \notin F}{\langle X, c_1 \prec c_2\rangle \xrightarrow{F} \langle X', c_1 \prec c_2\rangle}\ [\text{Precedence}] \qquad \frac{c_1 \notin F \vee c_2 \in F}{\langle X, c_1 \subseteq c_2\rangle \xrightarrow{F} \langle X', c_1 \subseteq c_2\rangle}\ [\text{Subclock}] \qquad \frac{c_1 \notin F \vee c_2 \notin F}{\langle X, c_1 \# c_2\rangle \xrightarrow{F} \langle X', c_1 \# c_2\rangle}\ [\text{Exclusion}]$$

$$\frac{X(c_1) = X(c_2) \quad c_2 \notin F \vee \{c_1, c_2\} \subseteq F}{\langle X, c_1 \preceq c_2\rangle \xrightarrow{F} \langle X', c_1 \preceq c_2\rangle}\ [\text{Causality-I}] \qquad \frac{X(c_1) > X(c_2)}{\langle X, c_1 \preceq c_2\rangle \xrightarrow{F} \langle X', c_1 \preceq c_2\rangle}\ [\text{Causality-II}]$$

$$\frac{c_1 \notin F \vee c_2 \in F \vee c_3 \in F}{\langle X, c_1 \triangleq c_2 + c_3\rangle \xrightarrow{F} \langle X', c_1 \triangleq c_2 + c_3\rangle}\ [\text{Union}] \qquad \frac{c_1 \notin F \vee \{c_2, c_3\} \subseteq F}{\langle X, c_1 \triangleq c_2 \times c_3\rangle \xrightarrow{F} \langle X', c_1 \triangleq c_2 \times c_3\rangle}\ [\text{Intersection}]$$

$$\frac{X(c_2) > X(c_3) \quad c_1 \in F \iff c_2 \in F}{\langle X, c_1 \triangleq c_2 \wedge c_3\rangle \xrightarrow{F} \langle X', c_1 \triangleq c_2 \wedge c_3\rangle}\ [\text{Infimum-I}] \qquad \frac{X(c_2) < X(c_3) \quad c_1 \in F \iff c_3 \in F}{\langle X, c_1 \triangleq c_2 \wedge c_3\rangle \xrightarrow{F} \langle X', c_1 \triangleq c_2 \wedge c_3\rangle}\ [\text{Infimum-II}]$$

$$\frac{X(c_2) < X(c_3) \quad c_1 \in F \iff c_2 \in F}{\langle X, c_1 \triangleq c_2 \vee c_3\rangle \xrightarrow{F} \langle X', c_1 \triangleq c_2 \vee c_3\rangle}\ [\text{Supremum-I}] \qquad \frac{X(c_2) = X(c_3) \quad c_1 \in F \iff c_2 \in F \vee c_3 \in F}{\langle X, c_1 \triangleq c_2 \wedge c_3\rangle \xrightarrow{F} \langle X', c_1 \triangleq c_2 \wedge c_3\rangle}\ [\text{Infimum-III}]$$

$$\frac{X(c_2) > X(c_3) \quad c_1 \in F \iff c_3 \in F}{\langle X, c_1 \triangleq c_2 \vee c_3\rangle \xrightarrow{F} \langle X', c_1 \triangleq c_2 \vee c_3\rangle}\ [\text{Supremum-II}] \qquad \frac{X(c_2) = X(c_3) \quad c_1 \in F \iff c_2 \in F \wedge c_3 \in F}{\langle X, c_1 \triangleq c_2 \vee c_3\rangle \xrightarrow{F} \langle X', c_1 \triangleq c_2 \vee c_3\rangle}\ [\text{Supremum-III}]$$

$$\frac{X(c_2) \geq d \quad c_1 \in F \iff c_2 \in F}{\langle X, c_1 \triangleq c_2\ \$\ d\rangle \xrightarrow{F} \langle X', c_1 \triangleq c_2\ \$\ d\rangle}\ [\text{Delay-I}] \qquad \frac{X(c_2) < d \quad c_1 \notin F}{\langle X, c_1 \triangleq c_2\ \$\ d\rangle \xrightarrow{F} \langle X', c_1 \triangleq c_2\ \$\ d\rangle}\ [\text{Delay-II}]$$

$$\frac{\exists m \in \mathbb{N}^+. X(c_2) = p \times m - 1 \quad c_1 \in F \iff c_2 \in F}{\langle X, c_1 \triangleq c_2 \propto p\rangle \xrightarrow{F} \langle X', c_1 \triangleq c_2 \propto p\rangle}\ [\text{Periodicity-I}] \qquad \frac{\nexists m \in \mathbb{N}^+. X(c_2) = p \times m - 1 \quad c_1 \notin F}{\langle X, c_1 \triangleq c_2 \propto p\rangle \xrightarrow{F} \langle X', c_1 \triangleq c_2 \propto p\rangle}\ [\text{Periodicity-II}]$$

Fig. 6. Operational semantics of CCSL.

and sets of constraints, respectively. The constant `empty` denotes an empty set of constraints, and the infix operator `__` denotes the union of two sets of constraints. The operators declared from Line 6 to 15 are used to represent the ten CCSL constraints correspondingly.

```
1  fmod CONSTRAINTS is
2    sorts   Constraint  Constraints .
3    subsort Constraint < Constraints .
4    op empty : -> Constraints [ctor] .
5    op __ : Constraints Constraints -> Constraints [ctor assoc comm id: empty] .
6    op _<_     : Clock Clock       -> Constraint [ctor] .
7    op _!=_$_  : Clock Clock Nat   -> Constraint [ctor] .
8    op _!=_+_  : Clock Clock Clock -> Constraint [ctor] .
9    op _!=_*_  : Clock Clock Clock -> Constraint [ctor] .
10   op _!=_/\_ : Clock Clock Clock -> Constraint [ctor] .
11   op _!=_\/_ : Clock Clock Clock -> Constraint [ctor] .
12   op _<<_    : Clock Clock       -> Constraint [ctor] .
13   op _#_     : Clock Clock       -> Constraint [ctor] .
14   op _<=_    : Clock Clock       -> Constraint [ctor] .
15   op _!=_~_  : Clock Clock Nat   -> Constraint [ctor] .
16   ...
17 endfm
```

To formalize a configuration X in a clock system, we think of a logical clock in a configuration as a triple (c, ℓ, n), consisting of the clock identifier c, a list ℓ of records with each value being *tick* or *idle* (abbreviated by t or i respectively), which represents that the clock ticks or not at the step where the value is located in the list, and a natural number n that denotes the numbers of ticks in ℓ.[1] We call it a *clock triple*. Initially, ℓ is an empty list, and n is 0. In the following module `CONFIGURATION`, sort `ConfElt` is declared to represent clock triples. The infix operator `[_,_,_]` is used to construct a clock triple with a clock, a list of records and a natural number as its arguments. `Clock` and `TickList` are the sorts to

[1] We keep n in the triple for the efficiency purpose of formal analysis because in this way we do not need to count the number of ticks in ℓ every time it is needed.

represent clocks and lists of ticks. Each list consists of a finite number of `t` and `i`, which are constants to represent *tick* and *idle* respectively. Given a clock system, we use a collection of clock triples to represent a *configuration* of the clocks in the system, and declare a sort named `Conf` to represent the configurations.

```
fmod CONFIGURATION is
  including NAT + CLOCK + TICK-LIST . --- + is a union operator on modules
  sort ConfElt Conf .
  subsort ConfElt < Conf .
  op [_,_,_] : Clock TickList Nat -> ConfElt [ctor] .
  op empty : -> Conf [ctor] .
  op _,_ : Conf Conf -> Conf [ctor assoc comm id: empty]
endfm
```

Each state of a clock system for a set of CCSL constraints consists of a set of clocks, a set of CCSL constraints on the clocks and a configuration of all the clocks. Besides the three information, we introduce a natural number to each state to represent the elapsed time (the number of elapsed steps) of the system. Such information is not used for specifying the operational semantics of CCSL, but for the purpose of verification by fixing a bound for searching and state exploration. We declare a sort `CCSLState` and an operator `<_;_;_;_>` as shown below to formalize states of clock systems.

```
sort CCSLState .
op <_;_;_;_> : ClockSet Constraints Conf Nat -> CCSLState [ctor] .
```

In the initial state of a clock system, the configuration is the set of initial clock triples and the step is 0.

4.3. Formalization of the operational semantics of CCSL

The operational semantics of CCSL constraints can be naturally formalized using Maude rewrite rules. The main rule is defined as follows:

```
--- The transition rule that formalizes the operational semantics of CCSL
crl < (F, F') ; PHI ; X ; K > => < (F, F') ; PHI ; update(X,F) ; K + 1 >
 if F =/= empty /\ satisfy(F, X, PHI) .
```

In the above rule, `F`, `F'` are variables of sort `ClockSet`; and `PHI`, `X`, and `K` are variables of sorts `Constraints`, `Conf`, and `Nat`, respectively. `X` and `F` correspond to X and F in Fig. 6. Given a set C of clocks, `F` can be instantiated by any subset C' of C by matching `(F, F')` to C with the variable `F'` of sort `ClockSet` being instantiated by $C - C'$. As mentioned in Section 4, `F` must be a non-empty set and satisfy corresponding conditions for CCSL constraints so that the clocks in `F` can tick in the current configuration X. The first conjunct in the condition part of the above rule says that `F` is not empty, and the second one means that `F` satisfies the constraints `PHI` in the configuration `X`. The definition of the function `satisfy` is explained later. If the two conditions hold, we obtain a new configuration by updating configuration `X` with the set `F` of clocks according to Equation (1), as represented by `update(X,F)`.

To formalize the satisfiability of a bounded schedule with respect to a set Φ of CCSL constraints, we declare in Maude a predicate `satisfy` which takes three arguments: a set F of clocks, a configuration X, and a CCSL constraint ϕ. The predicate returns true if $\mathcal{C}$ satisfies Φ with respect to X, and otherwise false. We consider each possible constraint form in Φ when defining `satisfy`. As examples, we list below four equations that are defined in Maude to specify the satisfiability of a set F (represented by `F`) of clocks with precedence, delay, infimum and periodicity with respect to X, respectively:

```
--- Declaration of predicate satisfy:
op  satisfy : ClockSet Conf Constraint -> Bool .
--- Precedence
eq  satisfy(F, ([C1, TL1, N1], [C2, TL2, N2], X), C1 < C2) =
    if N1 == N2 then not (C2 in F) else true fi .
--- Delay
eq  satisfy(F, ([C2, TL2, N2], X), C1 != C2 $ N) =
    if N2 >= N then (C1 in F) == (C2 in F) else not (C1 in F) fi .
```

We first consider the Maude definition of precedence for example. The second argument of `satisfy` at Line 4 represents an arbitrary configuration X, where there must be at least two clock triples of `C1` and `C2` whose lists of ticks are `TL1` and `TL2` respectively. The variables `N1` and `N2` of sort `Nat` in the triples represent the numbers of ticks in `TL1` and `TL2`. The

right-hand side term of the equation represents the condition of the rule defined for precedence in Fig. 6. Because the correspondence between the condition and their definitions in the equation is clear, we omit detailed explanations in the paper. The second equation is defined to formalize the two cases (Delay-I and Delay-II in Fig. 6) of the operational semantics of delay.

Likewise, we consider the Maude definitions of infimum and periodicity. The equations defined for them are as follows:

```
--- Infimum
eq satisfy(F, ([C1, TL1, N1],[C2, TL2, N2],[C3, TL3, N3], X), C1 != C2 /\ C3) =
   (if N2 > N3 then (C1 in F) == (C2 in F) else
    (if N2 < N3 then (C1 in F) == (C3 in F) else
         (C1 in F) == (C2 in F or C3 in F)
      fi)
    fi) .
--- Periodicity
eq satisfy(F, ([C1, TL1, N1], [C2, TL2, N2], X), C1 != C2 ~ P) =
   if (N2 + 1) rem P == 0 then (C1 in F == C2 in F) else not (C1 in F) fi .
```

5. Formal analysis on CCSL in Maude

Given a set of CCSL constraints, several properties need to be checked or verified, e.g., is there a schedule that satisfies all the constraints? (are the constraints consistent?); if so, pick one specific schedule according to an external criteria (avoid deadlocks, reduce memory usage, force a fast pace). In this section, we show four applications that directly derive from our encoding of the operational semantics of CCSL in Maude.

5.1. Bounded scheduling

Given a bound n and a set of clock constraints Φ, we can use Maude's `search` command to search for all the n-bounded schedules that satisfy Φ. If Maude cannot find at least one of such schedules, it means that there does not exist such an n-bounded schedule, and further we can conclude that there does not exist a schedule that satisfies Φ, i.e., Φ is not valid. For instance, $\{c_1 \prec c_2, c_2 \prec c_1\}$ is invalid.

We show an example of finding bounded schedules for a given set of clock constraints using Maude's `search` command.

Example 1. Find bounded schedules that satisfy the set of constraints $\Phi_1 = \{c_1 \prec c_2, c_3 \triangleq c_1 \$ 1, c_2 \prec c_3\}$.

In Maude, we use `'c1 < 'c2`, `'c3 != 'c2 $ 1` and `'c2 < 'c3` to represent the three constraints, respectively. First, we use Maude's `search` command to find schedules with maximal search depth 30. The actual command and the result returned by Maude are as follows:

```
search [,30] < ('c1, 'c2, 'c3) ; ('c1 < 'c2) ('c2 < 'c3) 'c3 != 'c1 $ 1 ;
               ['c1,nil,0],['c2,nil,0],['c3,nil,0] ; 0 > =>*
               < CS:ClockSet ; CTS:Constraints ; X:ConfList ; 30 > .
Solution 1 (state 30)
X:ConfList -->
    ['c1,t i t i t i t i t i t i t i t i t i t i t i t i t i t i t i t i,15]
    ['c2,i t i t i t i t i t i t i t i t i t i t i t i t i t i t i t i t,15]
    ['c3,i i t i t i t i t i t i t i t i t i t i t i t i t i t i t i t i,14]
No more solutions.
```

In the above `search` command, we only provide the maximum depth of the search and do not specify the number of expected solutions. It means that Maude will return all the solutions it finds within the depth 30. The starting term represents the initial configuration where the list of each clock is nil, and the step is 0. The pattern represents an arbitrary configuration where the step is 30, which means a configuration after 30 steps of evolution. The collection of the tick lists of all the clocks in the configuration can be interpreted as a schedule of bound 30.

The search command returns only one solution, which means that there exists only one schedule whose bound is 30 within the search depth 30. The bounded schedule satisfies the three constraints in Φ_1. From the list of ticks of the three clocks, one can see that c_1 and c_3 only tick at all odd steps except that c_3 does not tick at the first step because of the constraint $c_3 \triangleq c_1 \$ 1$, and c_2 only ticks at all even steps. Namely, c_1 and c_2 tick alternatively by the returned bounded schedule. The result coincides with the definition of *alternation* constraint between clocks in an earlier work of the third author [14].

5.2. Periodic scheduling

We can also find periodic schedules in Maude using the operational semantics of CCSL. The basic idea is to formalize the sufficient conditions in Theorem 1 and at each step k' we check if all these conditions are satisfied by a k'-bounded schedule, that is, if there exists a step k such that $1 \leq k < k'$ and the k'-bounded schedule satisfies the five conditions with k' and k. If that is the case, a periodic schedule is found, whose period is $k' - k$.

We declare a function `getPeriod`, which returns the period for a given configuration $\mathcal{X}$ if the underlying bounded schedule in X satisfies the sufficient conditions, and 0, otherwise. As shown below, `getPeriod` takes four arguments, a configuration, a set of constraints, and two natural numbers corresponding to k and k', respectively. It checks each step from k to 0 in a backward manner until it finds one satisfying the sufficient conditions. Once (and if) such a step is found, the corresponding period is returned. In the equation, term `sd(K',K)` represents the value of subtracting `K` from `K'`, with `K'` greater than `K`. `isPeriodic` is an auxiliary function, which returns true when a bounded schedule satisfies the sufficient conditions with respect to a set of constraints and two given steps. It is defined inductively with respect to each kind of constraint. The second equation defined below considers the case of precedence. The conjuncts at Line 9 represent Condition 1 and the first conjunct at Line 10 represents Condition 2 in Theorem 1. Other conditions are formalized likewise, and we omit the details in the paper.

```
1  --- Declaration and definition of getPeriod and isPeriodic
2  op getPeriod  : Conf Constraints Nat Nat  -> Nat .
3  op isPeriodic : Conf Constraints Nat Nat  -> Bool .
4  eq getPeriod(X, PHI, K, K') =
5     if K > 0 then (if isPeriodic(X,PHI,K,K') then sd(K',K) else
6                    getPeriod(X,PHI, sd(K,1),K') fi)
7      else 0 fi .
8  eq isPeriodic(([C1, TL1, N1],[C2, TL2, N2], X), (C1 < C2) PHI, K, K') =
9     tick(TL1,K) == tick(TL1,K') and
10    tick(TL2,K) == tick(TL2,K') and
11    sd(N1,num(TL1,K)) >= sd(N2,num(TL2,K)) and
12    isPeriodic(X,PHI,K,K') .
```

Using the rewrite rule defined in Section 4.3, one can find periodic schedules with *search* command by specifying an extra condition. The condition is that `isPeriod` must return a non-zero period in the expected configuration. For instance, the following code shows the command to find a periodic schedule for the constraints in Φ_1 and the solution returned by Maude:

```
1 search [1] < ('c1, 'c2, 'c3);('c1 < 'c2)('c2 < 'c3)( 'c3 != 'c1 $ 1);['c1,nil,0] ['c2,nil,0] ['
      c3,nil,0] ; 0 > =>* < F ; PHI ; X' ; K' > such that P:NzNat := getPeriod(X',PHI,sd(K',1),K
      ') .
2 Solution 1 (state 4)
3 X':Conf --> ['c1,t i t i,2], ['c2,i t i t,2], ['c3,i i t i,1]
4 K':Nat --> 4
5 P:NzNat --> 2
```

The solution states that a periodic schedule is found, whose period is 2 and the starting step of the first period is 2. The returned periodic schedule is essentially the same as the one found by bounded scheduling in Section 5.1

We can search simultaneously more than one periodic schedules for a given set of constraints. Let us consider finding periodic schedules of the precedence constraint $c_1 \prec c_2$ as an example. We set the number of expected periodic schedules to search to be 4. The command used for the search is as follows:

```
1 search [4] < ('c1, 'c2) ; ('c1 < 'c2); ['c1,nil,0], ['c2,nil,0] ; 0 > =>*
2            < F ; PHI ; X' ; K' > such that P := getPeriod(X',PHI,sd(K',1),K') .
```

Table 1 shows four periodic schedules found by Maude for $c_1 \prec c_2$ when the bound is set to 4. The red steps in each schedule in the table are the starting steps of the first and second iterations of the period.

We can also provide a concrete period and use Maude to search if there exist such periodic schedules with the given specific period. However, due to the infinite number of steps, and like when searching bounded schedules, it cannot be guaranteed that there do not exist such periodic schedules even if Maude returns no result.

Table 1
Four periodic schedules found by Maude for $c_1 \prec c_2$ when the bound is set to 4.

Schedule	Clock/Step	1	2	3	4	5	6	...	Period p
1	c_1	t	t	t	t	t	t	...	1
	c_2	i	t	t	t	t	t	...	
2	c_1	t	i	t	i	t	i	...	2
	c_2	i	t	i	t	i	t	...	
3	c_1	t	t	t	t	t	t	...	1
	c_2	i	i	t	t	t	t	...	
4	c_1	t	t	t	i	t	i	...	2
	c_2	i	i	i	t	i	t	...	

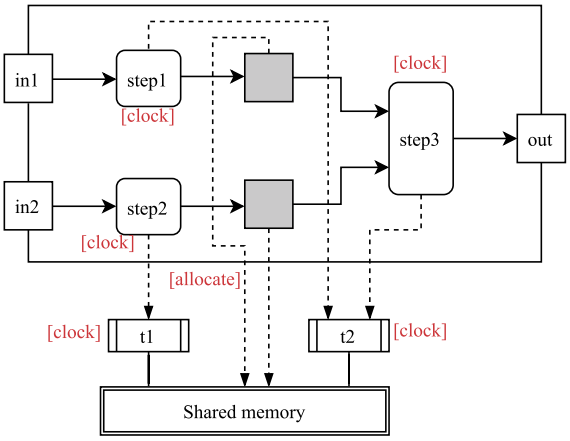

Fig. 7. A component in a practical application.

5.3. Formal verification by (bounded) model checking

Beside checking the existence of expected schedules, one may also want to explore the properties of the schedules that satisfy a set of constraints. For instance, one would expect that all the clocks must tick infinitely often, or a clock must tick some steps later after some other clock ticks. Another important application of the executable semantics of CCSL is to verify such properties by bounded model checking using the `search` command or Maude's LTL model checker.

5.3.1. Detection of deadlock schedules by bounded model checking

Finding deadlocks, or rather finding schedules without deadlocks, is one very important feature for a CCSL specification. The word deadlock usually means that after some steps, the system cannot evolve at all, i.e., no clock can tick. Due to the polychronous nature of CCSL, some parts of the system may be highly independent from other parts (see Definition 6 about connected clocks). Consequently, we use a stronger condition for deadlocks. Indeed, in CCSL, we expect all the clocks to tick infinitely often, so deadlock means that after some steps there is at least one clock that will never be allowed to tick.

Besides, since a CCSL specification may have conflicts, some schedules may end up in deadlocks while others may not. On this matter, we use the same definition as in [14]. We rely on our encoding in Maude to select those schedules that do not have deadlocks.

Example 2. The example is from a previous work [14]. It was previously used to perform flow latency analysis with the Architecture Analysis & Design Language (AADL) [8]. As shown in Fig. 7, the component takes two inputs in_1 and in_2, performs computations on the two inputs concurrently in $step_1$ and $step_2$, then combines the results to produce a final value in $step_3$ based on data produced at $step_1$ and $step_2$, and then outputs this final result through out.

The application continuously captures new inputs and produces one output for each pair of inputs in a streaming fashion.

Each processing unit is controlled by a clock, and a set Φ_2 of clock constraints specify the intentional semantics of the AADL specification (see Fig. 8). Let us note that tmp_1 and tmp_2 are two intermediate clocks required to specify constraints between in_1, in_2 and out.

Using the approach introduced in Section 5.2, we can find multiple periodic schedules in which each clock can tick infinitely often. A common feature of these periodic schedules is that in_1 must always tick simultaneously with in_2. If we

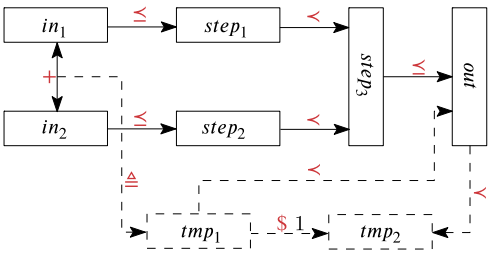

Fig. 8. An abstraction as a clock constraint specification.

Table 2
Six deadlock schedules found by Maude for CCSL constraints Φ_2.

No.	in_1	in_2	$step_1$	$step_2$	$step_3$	out	tmp_1	tmp_2
1	t	i	t	i	i	i	t	i
2	i	t	i	t	i	i	t	i
3	t i	i i	i t	i i	i i	i i	t i	i i
4	i i	t i	i i	i t	i i	i i	t i	i i
5	t i t	t i i	t i t	t i i	i t i	i t i	t i t	i i t
6	t i i	t i t	t i i	t i t	i t i	i t i	t i t	i i t

break this pattern (not explicitly specified), the application may go into a deadlock state. The reason for the deadlock is that the specification does not force in_1 and in_2 to tick simultaneously (see Φ_2). If in_1 ticks alone (resp. in_2), then out should tick next, but in order to tick out needs both ticks of in_1 and in_2, as defined by the constraints between in_1, in_2 and out. However, in_2 (resp. in_1) can only tick if out ticks before, which therefore leads to a deadlock.

In Maude, one can find such a deadlock by searching for a target state which cannot be further rewritten by any rewrite rules. For the above example, we search for the deadlock schedules within 3 steps in Maude with the following command:

```
search [,3] < 'in1, 'in2, 'out, 'step1, 'step2, 'step3, 'tmp, 'tmp2 ;
 ('out < 'tmp2)('step1 < 'step3)('step2 < 'step3) ('tmp < 'out)('in1 <= 'step1)
 ('in2 <= 'step2)  ('step3 <= 'out) ('tmp2 != 'tmp $ 1)('tmp != 'in1 + 'in2) ;
 ['in1,nil,0], ['in2,nil,0], ['out,nil,0],['step1,nil,0], ['step2,nil,0],
 ['step3,nil,0], ['tmp,nil,0], ['tmp2,nil,0] ; 0 > =>! S:CCSLState .
```

The search command finds six deadlock schedules, as shown in Table 2. A common feature of these deadlock schedules is that at some step in_1 and in_2 do not tick simultaneously, which finally results in a state where no clock can tick anymore.

One solution to the deadlock is to force in_1 and in_2 to tick simultaneously, which can be achieved by introducing two new constraints, i.e. $in_1 \subseteq in_2$ and $in_2 \subseteq in_1$. Maude returns no solution, which means that no deadlock schedule is found.[2] However, forcing in_1 and in_2 to tick simultaneously may be too restrictive from a pragmatic point of view since it may be difficult to synchronize two independent events. It suffices to enforce the slowest clock between in_1 and in_2 to tick before out. Thus, we replace $in_1 + in_2$ by $in_1 \wedge in_2$ in Fig. 8. We denote the new set after replacing the constraint by Φ_2'. Maude also returns no solution with a given maximum depth of the search for Φ_2'.

5.3.2. LTL model checking of periodic schedules

In this section, we explore the solutions that do not require to rely on bounded model-checking only. This is the reason that motivated the definition of periodic schedules since it allows a finite abstraction of the infinite state space.

By definition, in periodic schedules all clocks tick periodically. For properties where the exact number of ticks is not relevant (most often, only the difference between the number of ticks of two clocks is important), we can force each periodic schedule to go from step k' back to step k, where k and k' are the beginning and ending steps of the first period of the schedule ($k < k'$).

```
crl [next] : < (F, F'); PHI ; X ; K' > => < (F, F'); PHI ; X' ; K' + 1 >
 if F =/= empty /\
    satisfy(F,X,PHI) /\
    X' := update(X,F) /\
    getPeriod(X',K' + 1,PHI) == 0 .
```

[2] It is worth mentioning that it does not guarantee that there is no deadlock schedule because only a bounded state space is explored.

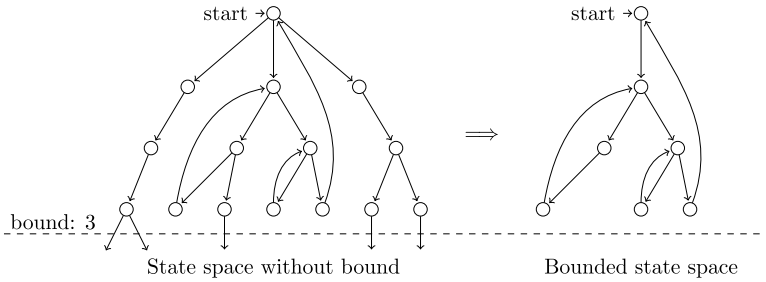

Fig. 9. Bounded state space of periodic schedulers.

```
6 crl [periodic] : < (F, F'); PHI ; X ; K' > =>
7                  < (F, F'); PHI ; rollback(X', P); sd(K' + 1,P) >
8  if F =/= empty /\
9     satisfy(F,X,PHI) /\
10    X':= update(X,F) /\
11    P := getPeriod(X',K' + 1,PHI) .
```

An extra condition is added to check if a successor configuration `X'` has an underlying periodic schedule. If not, configuration `X` is changed into a new one, i.e. `X'`. Otherwise, it is changed into an old one by rolling back `P` steps, where `P` is the period of the searched periodic schedule.

By excluding non-periodic schedules, we obtain a finite state space only from the searched periodic schedules, as depicted in Fig. 9. Fig. 9 (left) shows an example of a state space that contains both periodic and non-periodic schedules up to a bound, e.g. 3. The path with a loop represents a periodic schedule. In the figure there are three periodic schedules, which constitute a finite state space, as shown in Fig. 9 (right). The finite state space can then be model checked. It is worth mentioning that we still need to set a bound in order to search for periodic schedules. In that sense, model checking of periodic schedules is still not complete because it is possible that some periodic schedules are not found within a given search bound and hence cannot be model checked.

Next, we show some basic properties that clock constraints are expected to satisfy and their representations as LTL formulae. Let `tick` be a parameterized predicate on states, which takes a clock c as argument and returns `true` if c ticks in a state. The definition of `tick` in Maude is as follows:

```
1 subsort CCSLState < State .
2 op tick : Clock -> Prop .
3 eq < F ; PHI ; [ C, (TL t),N ] X ; K > |= tick(C) = true .
4 eq S:CCSLState |= P:Prop = false [owise] .
```

The operator `tick` defines a proposition `tick(`c`)` for each clock c. The first equation defines that `tick(`c`)` is true in a given state where clock c ticks. The second equation (line 4) means that `tick(`c`)` is false in other cases. Below we list four common properties of CCSL constraints and their corresponding LTL formulas.

- *Repeated ticking*: all clocks must tick infinitely often, which can be formalized as: $\bigwedge_{c\in C} \Box\Diamond \texttt{tick}(c)$.
- *Simultaneous ticking*: two clocks c_1 and c_2 must always tick simultaneously, which can be formalized as: $\Box(\texttt{tick}(c_1) \Longleftrightarrow \texttt{tick}(c_2))$.
- *Leading-to ticking*: if a clock c_1 ticks, it must cause another clock c_2 to tick eventually, which can be formalized as: $\Box(\texttt{tick}(c_1) \rightarrow \Diamond \texttt{tick}(c_2))$.
- *Alternative ticking*: two clocks c_1 and c_2 must always tick in alternation, which can be formalized as: $\Box\big(\texttt{tick}(c_1) \rightarrow \bigcirc(\neg\texttt{tick}(c_1)\ \mathbf{U}\ \texttt{tick}(c_2)) \wedge \texttt{tick}(c_2) \rightarrow \bigcirc(\neg\texttt{tick}(c_2)\ \mathbf{U}\ \texttt{tick}(c_1))\big)$.

We take the model checking of some LTL properties of the constraints in Φ_1 as an example. The intention of composing the three constraints in Φ_1 is to force clocks c_1 and c_2 to tick alternatively. By bounded scheduling and periodic scheduling analyses, we have indeed found a schedule that seemingly satisfies this property. By model checking, we can formally verify it. The command used for model checking the property is as follows:

```
1 red modelCheck(<('c1,'c2,'c3); ('c1 < 'c2) ('c2 < 'c3) ('c3 != 'c1 $ 1);
2                 ['c1,nil,0], ['c2,nil,0], ['c3,nil,0]; 0 >,
3                []((tick('c1) -> O (~ tick('c1) U tick('c2))) /\
4                   (tick('c2) -> O (~ tick('c2) U tick('c1))))) .
5 result Bool: true
```

Maude returns `true`, meaning that Φ satisfies the alternative ticking property.

5.4. Simulation with customized arbitration policies

Given a set Φ of clock constraints, there may be more than one schedule. In such cases, we can find an external criteria to pick one that has a given property. In a previous work [7], this was called an *arbitration/simulation policy*. One of the side effects of our work is that we can now formalize these arbitration policies and ask Maude to apply them for us. A clock must not tick if it does not have to tick. Such policies are usually user-definable. Another application of the executable operational semantics of CCSL is to *simulate* the behaviors of clocks under given constraints along with some arbitration policies.

We consider five basic arbitration policies, which we call *randomness*, *maximum minimum*, *laziness*, *activeness* and *mixity*, respectively.

- *Randomness*: to specify that at each step a successor is randomly chosen among all possible successors.
- *Maximum*: to specify that at each step the number of clocks that tick is maximal among all possible successors. Such a solution is not necessarily unique. This is useful to force the pace and execute as much as possible.
- *Minimum*: to specify that at each step the number of clocks that tick is minimal among all possible successors. Such a solution is not necessarily unique.
- *Laziness*: to specify a set of *lazy* clocks that only tick if they have to tick at each step.
- *Activeness*: to specify a set of *active* clocks that tick as soon as they can at each step. When all the clocks are active ones, the policy is equivalent to maximum.
- *Mixity*: to specify a set of lazy clocks and a set of active clocks. When all the clocks are lazy ones, the policy is equivalent to minimum.

Each of the above arbitration policies except randomness determines a unique successive behavior of each clock. By simulating the behaviors of clocks from initial state with an arbitration policy, it produces a trace which is essentially a schedule characterized by the policy.

The arbitration policies can be considered as internal strategies that control how the rewrite rules are applied. Formalizing strategies in Maude can be done using Maude's built-in meta-programming features, in which Maude programs are used as data in formalized strategies. Thanks to this feature, arbitration policies in CCSL can be naturally formalized in Maude.

Below we give partly the Maude definitions of the basic data types, functions and the rewrite rules used to achieve customized simulation.

```
1  sort Policy CCSLState4Sim .
2  ops rand max min : -> Policy .
3  ops lazy active : ClockSet -> Policy .  op mix : ClockSet ClockSet -> Policy .
4  op  [_`,_] : CCSLState Policy -> CCSLState4Sim .
5  op  getAllSuccessors : CCSLState Nat    -> ConfSet .
6  op  getConfbyPol     : ConfSet   Policy -> Conf .
7  crl [next] : [ < F ; PHI ; X ; K >, P ] => [ < F ; PHI ; X' ; K + 1 >, P ]
8   if X' := getConfbyPol(getAllSuccessors(< F ; PHI ; X ; K >,  0), P).
9  --- The Maude definition of getAllSuccessors
10 ceq getAllSuccessors(< F ; PHI ; X ;  K >, N) =
11     downTerm(T, nil), getAllSuccessors(< F ; PHI ; X ; K >, N + 1)
12  if RT := metaSearch(upModule('CCSL-SEMANTIC,false),upTerm(< F ; PHI ; X ; K >),
13            '<_;_;_;_>[upTerm(F),upTerm(PHI),'X':Conf,upTerm(M + 1)],nil,'+,1,N)/\
14     ('X':Conf <- T) := getSubstitution(RT) .
15 --- the Maude definition of getConfbyPol with laziness policy
16 eq  getConfbyPol(CF,lazy(C,CS)) = getConfbyPol(getNonTickConf(CF,C),lazy(CS)) .
```

We declare a sort `Policy` for arbitration policies, and six operators (as shown above at Lines 2 and 3) to represent the six kinds of arbitration policies respectively. A state of a clock system with policy consists of two parts, i.e., a state of the system and a policy, which is formalized as a term `[ S, P ]` with a state `S` of sort `CCSLState` and a policy `P` of sort `Policy`. We name the sort of such terms `CCSLState4Sim`. Behaviors of a clock system with policy can be formalized by the conditional rewrite rule defined at Lines 7 to 8. The main difference with Section 4.3 is that the successor `X'` of `X` in the new rule must be the one that follows the given arbitration policy. The basic idea for obtaining such `X'` is to search by one-step forward from `X` using the rule defined in Section 4.3 and find all possible successors, which is achieved by function `getAllSuccessors`. Then, `X'` is determined according to the given policy, which is achieved by function `getConfbyPol`.

The two functions `getAllSuccessors` and `getConfbyPol` are declared at Lines 5 and 6. In the declaration of `getAllSuccessors`, `ConfSet` is a sort to represent sets of configurations. In the definition of `getAllSuccessors` from Line 10 to 14, `metaSearch` is a built-in meta-level function corresponding to the `search` command. We omit detailed explanations about this function because it is out of the scope of this paper. Interested readers are referred to the

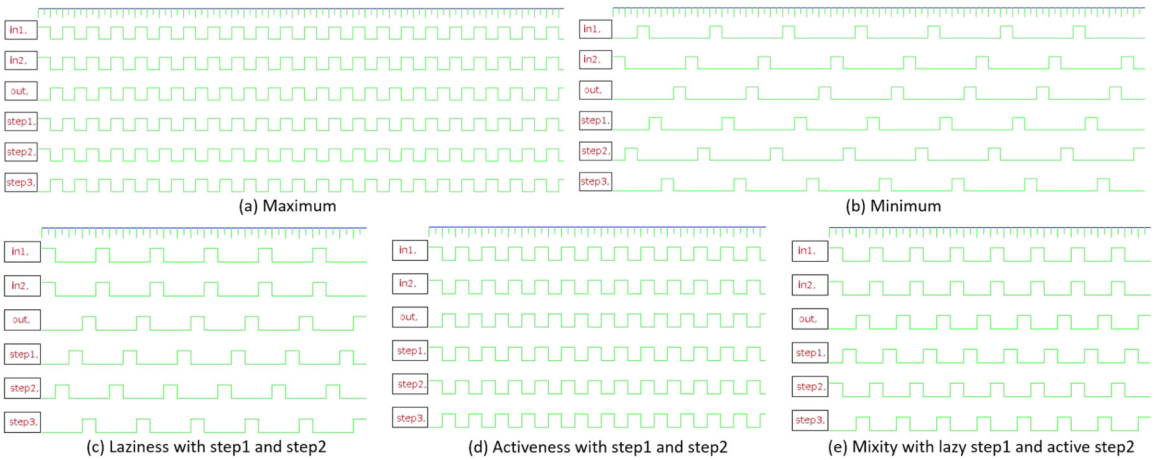

Fig. 10. The simulation results with the five arbitration policies.

work [4] for details. As an example, we give the definition of `getConfbyPol` with laziness policy, i.e., the equation at Line 16. `CF` is a variable of `ConfSet`, representing a set of configurations. The function `getNonTickConf` returns the sets of configurations in `CF` where clock `C` does not tick at the last step. If there is no such a configuration, `CF` is returned. We omit the definition of `getNonTickConf` because it is straightforward.

With the rule defined above, we can use Maude's `rew` command to simulate the behavior of a clock system with respect to a set of CCSL constraints and an arbitration policy. As an example, we simulate a clock system with respect to the constraints in Φ'_2 with different arbitration policies. Because simulation will not terminate unless the schedule being simulated is a deadlock one. We set a bound 50 to the maximal depth of simulation. The commands used for the simulations are of the following form:

```
1 rew [50] [ < 'in1, 'in2, 'out, 'step1, 'step2, 'step3, 'tmp, 'tmp2 ;
2  ('out < 'tmp2)('step1 < 'step3)('step2 < 'step3) ('tmp < 'out)('in1 <= 'step1)
3  ('in2 <= 'step2)  ('step3 <= 'out)  ('tmp2 != 'tmp $ 1)('tmp1 != 'in1 /\ 'in2);
4  ['in1,nil,0], ['in2,nil,0], ['out,nil,0],  ['step1,nil,0], ['step2,nil,0],
5  ['step3,nil,0], ['tmp,nil,0], ['tmp2,nil,0] ; 0 >, P ] .
```

We replace `P` in the above command with `max`, `min`, `lazy('step1,'step2)`, `active('step1,'step2)` and `mix('step1,'step2)` to simulate with the five arbitration policies, respectively. Maude returns different schedules. To make simulation results easier to read, we developed a prototype tool on the top of Eclipse using Maude as the underlying simulation engine. Fig. 10 shows the graphical representation of the five simulation results generated by the tool. All the five schedules satisfy the sufficient conditions of periodicity, and hence are periodic. There two main reasons for the periodicity of these schedules. One reason is the *alternation* constraint between the clocks *out* and tmp_1. However, only by alternation a schedule may not be periodic because there is no constraint on the interval between two alternative ticks. When an arbitration policy is provided, the clocks that should tick in next step may be deterministic. The schedule becomes periodic when all the intervals between alternative ticks are the same. However, the periods of schedules may be different under different policies. For instance, the periods under *maximum* and *activeness with* $step_1$ *and* $step_2$ are 1, while the periods under the other three policies are 5 (under *minimum*), 3 (under *laziness*) and 2 (under *mixity*), respectively.

6. Related work and discussion

CCSL mainly deals with logical clocks, i.e., unbounded increasing sequences (streams) of integers. The semantics of clock constraints may depend on Boolean parameters, in which case, we remain in a finite world and can rely on traditional verification and analysis results and tools. The constraints may also depend on unbounded natural numbers, for instance, the number of times a given clock has ticked. In this latter case, the constraint is called unsafe [15]. A specification is safe if it does not use any unsafe constraint.

The satisfiability of CCSL specifications has already been studied. Most of the time, decidability is achieved by reducing to a safe subset of CCSL constraints. The reference semantics of CCSL was given in a research report [1] mainly to be able to define a simulation tool called TimeSquare [7]. TimeSquare encodes the operational semantics of CCSL in Java and captures Boolean constraints symbolically using Binary Decision Diagrams (BDD). TimeSquare works step by step and at each step, finding a solution reduces to a satisfiability problem. After deciding if and how many valid solutions can be found at a step, the TimeSquare clock engine picks one solution according to its simulation policy, updates the state space and moves forward. TimeSquare does not consider the unbounded specification as a whole and only produces one finite possible trace

that satisfies all the constraints up to a given number of steps. In this work, we use bounded model-checking, and we can then explore all the solutions reached in a given number of steps, instead of only one.

Several solutions have been proposed to make an exhaustive exploration of the entire state space (not up to a pre-defined number of steps). A comprehensive list of references has been summarized in a recent survey [14]. However, one aspect is to be able to decide whether the state space can be represented with a finite abstraction even though the specification is unsafe [15]. Another way is to force a finite space by restricting to subsets of safe constraints [25,9,24]. In the work [22], a subset of CCSL constraints are transformed into timed automata to model check both logical and chronometric properties of CCSL constraints using the UPPAAL tool. In this work, we do not make any assumptions on whether the specification is safe or not by considering only periodic scheduling of ccsl from both theoretical and practical perspectives.

Like other existing approaches proposed for the formal analysis of ccsl, the formal analysis approach to periodic scheduling of ccsl is also based on its formal semantics. However, the main difference is that, thanks to the Maude environment, all the analyses performed result directly from the operational semantics without intermediate transformations, so without the need to prove that the semantics is preserved. Yu et al. [25] proposed to encode ccsl in Signal before transforming it to the internal format of Sigali. We hope that the encoding in Maude will allow to conduct automated verification for all the transformational approaches that use ccsl as a step. Maude also gives a framework to define the arbitration policies formally for the purpose of simulation. Some undocumented simulation policies are available in TimeSquare [7]. In Section 4, we give a simple formal interpretation for three of these simulation policies. We also propose three more and show how they can be implemented using Maude metasearch function.

Finally, abstract interpretation [5] or infinite model-checking [11] would allow reasoning on the global ccsl specification without restrictions. However, the encoding is likely to introduce semantic variations and we do not know at the moment how to encode ccsl constraints in a compositional way.

7. Conclusion and future work

We have proposed a new notion of bounded and periodic schedules to satisfy a ccsl specification. The satisfiability problem for ccsl specifications, which is still an open problem in the general case, is proved to be decidable with regards to bounded and periodic schedules even when using unsafe constraints. This is the first main result from a theoretical perspective. From a practical point of view, the second result of this paper is to present a Maude encoding of the formal semantics of an extended subset of ccsl to compute bounded and periodic schedules. We also describe how to implement customized arbitration policies within both simulation and bounded model-checking context.

Compared with our earlier work [26], we proposed in this paper less constraining conditions for being able to build more periodic schedules automatically with the tool. We have also improved our work by considering an extended subset of the language to support the notion of logical periodicity between logical clocks. We then had to define a new semantic model for considering this extension.

ccsl is still an emerging formal specification language which is under further study. We are considering introducing new types of constraints such as alternative operators. Maude, as a formal meta-tool, provides a formal environment to study the language itself by quickly developing formal analysis toolkit for ccsl in terms of the executable semantics of ccsl constraints. For its executability of the mathematical semantics, Maude can be used as a precise interpreter and state explorer for the formal analysis of ccsl constraints.

Acknowledgements

We thank all the anonymous reviewers including those who reviewed the workshop version published in FTSCS 2015 for their valuable comments on this work. This research was supported by The National Natural Science Foundation of China (NSFC) project: No. 61502171 and by the French Clarity Project.[3]

References

[1] C. André, Syntax and Semantics of the Clock Constraint Specification Language (CCSL), Research Report 6925, INRIA, 2009.
[2] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, R. de Simone, The synchronous languages 12 years later, Proc. IEEE 91 (2003) 64–83.
[3] G. Berry, G. Gonthier, The Esterel synchronous programming language: design, semantics, implementation, Sci. Comput. Program. 19 (1992) 87–152.
[4] M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, All About Maude, Lect. Notes Comput. Sci., vol. 4350, Springer, 2007.
[5] P. Cousot, Abstract interpretation, ACM Comput. Surv. 28 (1996) 324–328.
[6] L. Cucu, N. Pernet, Y. Sorel, Periodic real-time scheduling: from deadline-based model to latency-based model, Ann. Oper. Res. 159 (2008) 41–51.
[7] J. Deantoni, F. Mallet, TimeSquare: treat your models with logical time, in: C.A. Furia, S. Nanz (Eds.), TOOLS (50), in: Lect. Notes Comput. Sci., vol. 7304, Springer, 2012, pp. 34–41.
[8] P. Feiler, J. Hansson, Flow Latency Analysis with the Architecture Analysis and Design Language (AADL), Technical Note CMU/SEI-2007-TN-010, Carnegie Mellon, Software Engineering Institute, 2007.
[9] R. Gascon, F. Mallet, J. DeAntoni, Logical time and temporal logics: comparing UML MARTE/CCSL and PSL, in: C. Combi, M. Leucker, F. Wolter (Eds.), TIME, IEEE, 2011, pp. 141–148.

[3] http://www.clarity-se.org/.

[10] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Commun. ACM 21 (1978) 558–565.
[11] J. Leroux, G. Sutre, Flat counter automata almost everywhere!, in: 3rd International Symposium on Automated Technology for Verification and Analysis, in: Lect. Notes Comput. Sci., vol. 3707, Springer, 2005, pp. 489–503.
[12] C.L. Liu, J.W. Layland, Scheduling algorithms for multiprogramming in a hard-real-time environment, J. ACM 20 (1973) 46–61.
[13] F. Mallet, C. André, R. de Simone, CCSL: specifying clock constraints with UML/MARTE, Innov. Syst. Softw. Eng. 4 (2008) 309–314.
[14] F. Mallet, R. de Simone, Correctness issues on MARTE/CCSL constraints, Sci. Comput. Program. 106 (2015) 78–92.
[15] F. Mallet, J.V. Millo, R. de Simone, Safe CCSL specifications and marked graphs, in: 11th ACM/IEEE Int. Conf. on Formal Methods and Models for Codesign, IEEE, 2013, pp. 157–166.
[16] J. Meseguer, Membership algebra as a logical framework for equational specification, in: 12th International Workshop on Recent Trends in Algebraic Development Techniques, WADT, in: Lect. Notes Comput. Sci., vol. 1376, 1997, pp. 18–61.
[17] J. Meseguer, Twenty years of rewriting logic, J. Log. Algebraic Program. 81 (2012) 721–781.
[18] J. Meseguer, G. Rosu, The rewriting logic semantics project, Theor. Comput. Sci. 373 (2007) 213–237.
[19] J. Meseguer, G. Rosu, The rewriting logic semantics project: a progress report, Inf. Comput. 231 (2013) 38–69.
[20] P.C. Ölveczky, Real-Time Maude and its applications, in: 10th International Workshop on Rewriting Logic and Its Applications, WRLA, in: Lect. Notes Comput. Sci., vol. 8663, Springer, 2014, pp. 42–79.
[21] D. Potop-Butucaru, R. de Simone, J. Talpin, The synchronous hypothesis and polychronous languages, in: Embedded Systems Design and Verification, CRC Press, 2009, pp. 1–20.
[22] J. Suryadevara, C.C. Seceleanu, F. Mallet, P. Pettersson, Verifying MARTE/CCSL mode behaviors using UPPAAL, in: 11th International Conference on Software Engineering and Formal Methods, SEFM, in: Lect. Notes Comput. Sci., vol. 8137, Springer, 2013, pp. 1–15.
[23] Q. Xu, R. de Simone, J. DeAntoni, Divergence detection for CCSL specification via clock causality chain, in: 2nd International Symposium on Dependable Software Engineering: Theories, Tools, and Applications, SETTA, in: Lect. Notes Comput. Sci., vol. 9984, Springer, 2016, pp. 18–37.
[24] L. Yin, F. Mallet, J. Liu, Verification of MARTE/CCSL time requirements in Promela/SPIN, in: I. Perseil, K. Breitman, R. Sterritt (Eds.), ICECCS, IEEE Computer Society, 2011, pp. 65–74.
[25] H. Yu, J. Talpin, L. Besnard, T. Gautier, H. Marchand, P.L. Guernic, Polychronous controller synthesis from MARTE/CCSL timing specifications, in: 9th IEEE/ACM International Conference on Formal Methods and Models for Codesign, MEMOCODE, IEEE, 2011, pp. 21–30.
[26] M. Zhang, F. Mallet, An executable semantics of clock constraint specification language and its applications, in: 4th International Workshop on Formal Techniques for Safety-Critical Systems, FTSCS, in: Commun. Comput. Inf. Sci., vol. 596, Springer, 2015, pp. 37–51.