SMT-Based Bounded Schedulability Analysis of the Clock Constraint Specification Language

Min Zhang[1], Fu Song[2(✉)], Frédéric Mallet[3], and Xiaohong Chen[1]

[1] Shanghai Key Laboratory of Trustworthy Computing, ECNU, Shanghai, China
[2] ShanghaiTech University, Shanghai, China
songfu@shanghaitech.edu.cn
[3] Université Cote d'Azur, CNRS, Inria, I3S, Nice, France

Abstract. The Clock Constraint Specification Language (CCSL) is a formalism for specifying logical-time constraints on events for the design of real-time embedded systems. A central verification problem of CCSL is to check whether events are schedulable under logical constraints. Although many efforts have been made addressing this problem, the problem is still open. In this paper, we show that the bounded scheduling problem is NP-complete and then propose an efficient SMT-based decision procedure which is sound and complete. Based on this decision procedure, we present a sound algorithm for the general scheduling problem. We implement our algorithm in a prototype tool and illustrate its utility in schedulability analysis in designing real-world systems and automatic proving of algebraic properties of CCSL constraints. Experimental results demonstrate its effectiveness and efficiency.

Keywords: SMT · CCSL · Schedulability · Logical time · Real-time system

1 Introduction

Model-based design has been widely used, particularly in the design of safety-critical real-time embedded systems. It has achieved industrial successes through languages such as SCADE [12], AADL [15] and UML MARTE [26]. For example, UML MARTE provides syntactic annotations to implement, when the context allows, classical real-time scheduling algorithms such as EDF (Earliest Deadline First). It also provides a domain-specific language–Clock Constraint Specification Language (CCSL) [3], to express the real-time behaviors of a system under development as logical constraints on system events, but independently of any physical time and classical real-time scheduling algorithms. CCSL has been used on several industrial scenarios such as vehicle systems [16] and cyber-physical systems [10,22].

This work is supported by NSFC grants 61872146, 61532019 and 61761136011.

R. Hähnle and W. van der Aalst (Eds.): FASE 2019, LNCS 11424, pp. 61–78, 2019.
https://doi.org/10.1007/978-3-030-16722-6_4

Model-based design usually starts with coarse-grained logical models that are progressively refined into more concrete ones until the final code deployment. It is well-known that the earlier one can detect and fix bugs in the refinement process, the better [7]. Therefore, it is critical to provide efficient methods and tools to check safety, liveness and schedulability on the logical models and not only on the definite deployed system. This has motivated a large body of works on verifying whether events are schedulable under a set of constraints expressed in CCSL [11,21,28,33,35,36,38], though its decidability is still open. These works first transform CCSL constraints into other formal representations such as transition systems [21], Promela [35], Büchi automata [36], timed automata [33], rewriting logics [38], instant relations [28], or timed-interval logics [11], and then apply existing tools. However, their approaches usually suffer from the state explosion problem. Moreover, most of these works only deal with the so-called safe subset of CCSL and the other ones only provide semi-algorithms. In our earlier work [39], we proposed an SMT-based verification approach to CCSL and demonstrated several applications of the approach to finding schedules, verifying temporal properties, proving constraint entailment, and analyzing the validity of system traces. Based on the approach, we implemented an efficient tool for verifying LTL properties of CCSL [40].

In this work we are focused on the scheduling problem of CCSL, a fundamental problem to which the aforementioned verification problems of CCSL can be reduced. We first prove that the *bounded* scheduling problem of CCSL with fixed bounds is NP-complete. To our knowledge, this is the first result regarding the complexity of the scheduling problem with CCSL. Then, we propose a decision procedure for the bounded scheduling problem with a given bound. The decision procedure is based on the transformation of CCSL into SMT formulas [39]. Our decision procedure is sound, complete, and efficient in practice. Based on this decision procedure, we turn to the general (i.e. unbounded) scheduling problem and present a binary-search based algorithm. Our algorithm is sound, i.e., if it proves either schedulable or unschedulable, then the result is conclusive. We implemented our algorithms in a prototype tool. The tool was used to analyze a real-world interlocking system in a rail transit system. Using the proposed approach, we also prove some algebraic properties of CCSL. The experimental results demonstrate the effectiveness and efficiency of the SMT-based approach.

The rest of this paper is organised as follows: Section 2 introduces CCSL. Section 3 defines the (bounded) scheduling problem of CCSL and shows that the bounded case is NP-complete. Section 4 presents an SMT-based decision procedure for the bounded scheduling problem and a sound algorithm for the general scheduling problem. Section 5 shows a case study and experimental results. Section 6 discusses related work, and Section 7 concludes the paper.

2 The Clock Constraint Specification Language

2.1 Logical Clock, History and Schedule

In CCSL, clocks are used to model occurrences of events, where a clock ticks when the corresponding event occurs. For instance, a clock may represent an

event that is dispatch of a task, communications between tasks or acquisition of a shared resource by a task. Constraints over clocks are used to specify causal and temporal relations between system events. No global physical time is presumed for the clocks and their constraints. This feature allows CCSL to define a polychronous specification of a system at a logical level.

Definition 1 (Logical clock). *A* (logical) clock *c is an infinite sequence of ticks $(c^i)_{i\in\mathbb{N}^+}$ with each c^i being* tick *or* idle*, where $\mathbb{N}^+$ denotes the set of all the non-zero natural numbers.*

The value of c^i denotes whether an event associated with c occurs or not at step i. If c^i is *tick*, then the event occurs, otherwise not. In particular, we denote by **1** a global reference logical clock that always ticks at each step.

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 $\forall i \in \mathbb{N}^+$, $\delta(i) = \{c \in C \mid c^i = tick\}$ and $\delta(i) \neq \emptyset$.*

Intuitively, a schedule δ defines a partial order between the ticks of the clocks. $\delta(i)$ is a subset of C such that $c \in \delta(i)$ iff c ticks at step i. The condition $\delta(i) \neq \emptyset$ expresses that step i cannot be empty. This forbids stuttering steps in schedules. As one can add or remove finite number of empty steps without effect on schedulability, we exclude them from schedules for succinctness.

A clock can memorize the number of ticks that it has made. We use *history* to represent the memorization.

Definition 3 (History). *Given a schedule δ for a set C of clocks, a* history *of δ is a function $\chi_\delta : C \times \mathbb{N}^+ \to \mathbb{N}$ such that for each $c \in C$ and $i \in \mathbb{N}^+$:*

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

$\chi_\delta(c,i)$ represents the number of the ticks that the clock c has made immediately before step i. (Note that the tick of c at step i is excluded in $\chi_\delta(c,i)$.) For simplicity, we may write χ for χ_δ if it is clear from the context.

2.2 Syntax and Semantics of CCSL

CCSL consists of 11 kinds of constraints, 4 of them are binary relations for specifying the *precedence*, *causality*, *subclocking*, and *exclusion* relations between clocks, and the others are used to define clocks from existing ones. Clocks defined by constraints may correspond to system events or are just introduced as auxiliary clocks without corresponding to any events.

Table 1. Semantics of CCSL with respect to schedules

	ϕ	$\delta \models \phi$
Precedence	$c_1\ [b] \prec c_2$	$\forall n \in \mathbb{N}^+.\chi(c_2,n) - \chi(c_1,n) = b \Rightarrow c_2 \notin \delta(n)$
Causality	$c_1 \preccurlyeq c_2$	$\forall n \in \mathbb{N}^+.\chi(c_1,n) \geq \chi(c_2,n)$
Subclock	$c_1 \subseteq c_2$	$\forall n \in \mathbb{N}^+.c_1 \in \delta(n) \Rightarrow c_2 \in \delta(n)$
Exclusion	$c_1 \# c_2$	$\forall n \in \mathbb{N}^+.c_1 \notin \delta(n) \vee c_2 \notin \delta(n)$
Union	$c_1 \triangleq c_2 + c_3$	$\forall n \in \mathbb{N}^+.c_1 \in \delta(n) \Leftrightarrow c_2 \in \delta(n) \vee c_3 \in \delta(n)$
Intersection	$c_1 \triangleq c_2 * c_3$	$\forall n \in \mathbb{N}^+.c_1 \in \delta(n) \Leftrightarrow c_2 \in \delta(n) \wedge c_3 \in \delta(n)$
Infimum	$c_1 \triangleq c_2 \wedge c_3$	$\forall n \in \mathbb{N}^+.\chi(c_1,n) = \max(\chi(c_2,n), \chi(c_3,n))$
Supremum	$c_1 \triangleq c_2 \vee c_3$	$\forall n \in \mathbb{N}^+.\chi(c_1,n) = \min(\chi(c_2,n), \chi(c_3,n))$
Periodicity	$c_1 \triangleq c_2 \propto p$	$\forall n \in \mathbb{N}^+.c_1 \in \delta(n) \Leftrightarrow (c_2 \in \delta(n) \wedge \exists m \in \mathbb{N}^+.\chi(c_2,n) = m \times p - 1)$
Filtering	$c_1 \triangleq c_2 \blacktriangledown w$	$\forall n \in \mathbb{N}^+.c_1 \in \delta(n) \Leftrightarrow (c_2 \in \delta(n) \wedge w[n])$
DelayFor	$c_1 \triangleq c_2\ \$\ d\ on\ c_3$	$\forall n \in \mathbb{N}^+.c_1 \in \delta(n) \Leftrightarrow (c_3 \in \delta(n) \wedge \exists m \in \mathbb{N}^+.(c_2 \in \delta(m) \wedge \chi(c_3,n) - \chi(c_3,m) = d))$

Definition 4 (Syntax). *A* CCSL *constraint ϕ is defined by the following form:*

Precedence:	$c_1\ [b] \prec c_2$	*Causality:*	$c_1 \preccurlyeq c_2$
Subclock:	$c_1 \subseteq c_2$	*Exclusion:*	$c_1 \# c_2$
Union:	$c_1 \triangleq c_2 + c_3$	*Intersection:*	$c_1 \triangleq c_2 * c_3$
Infimum:	$c_1 \triangleq c_2 \wedge c_3$	*Supremum:*	$c_1 \triangleq c_2 \vee c_3$
Periodicity:	$c_1 \triangleq c_2 \propto p$	*Filtering:*	$c_1 \triangleq c_2 \blacktriangledown w$
DelayFor:	$c_1 \triangleq c_2\ \$\ d\ on\ c_3$		

where $b \geq 0$, $d \geq 0$ and $p > 0$ are natural numbers, c_1, c_2, c_3 are logical clocks and w is a (possibly infinite) word over $\{0,1\}$ expressed as a (ω-)regular expression.

For simplifying presentation, we denote by $c_1 \prec c_2$ the constraint $c_1\ [0] \prec c_2$, and $c_1 \triangleq c_2\ \$\ d$ the constraint $c_1 \triangleq c_2\ \$\ d\ on\ c_3$ such that $c_2 = c_3$.

The semantics of CCSL constraints is defined over schedules. Given a CCSL constraint ϕ and a schedule δ, the satisfiability relation $\delta \models \phi$ (i.e., δ satisfies constraint ϕ) is defined in Table 1.

The precedence constraint $c_1 \prec c_2$ (i.e., $c_1\ [0] \prec c_2$) expresses that the clock c_1 precedes the clock c_2. Suppose there is an unbounded buffer with two operations *fetch* and *store*, which respectively fetch data from and store data into the buffer. Fetch is only allowed when the buffer is nonempty. If the buffer is initially empty, store operation must strictly precede fetch operation. This behavior can be expressed by the constraint: *store* $\prec$ *fetch*. Likewise, the precedence constraint can be used to represent reentrant tasks by replacing *store* with *start* and *fetch* with *finish*.

The general precedence constraint $c_1\ [b] \prec c_2$ that can specify the differences b between the number of occurrences of two clocks before the precedence takes effect. Hence, it is able to express more complicated relations. For instance, if the buffer initially is nonempty, fetch operations can be performed prior to any

store operation. Figure 1 shows such a scenario where 4 elements are initially presented in the buffer. This behavior can be represented as: $store\ [4] \prec fetch$.

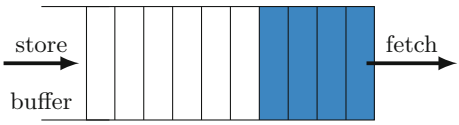

Fig. 1. Example for $store\ [4] \prec fetch$

The causality, subclock and exclusion constraints are straightforward. The causality constraint $c_1 \preccurlyeq c_2$ specifies that the occurrence of c_2 must be caused by the occurrence of c_1, namely at any moment c_1 must have ticked at least as many times as c_2 has. The subclock constraint $c_1 \subseteq c_2$ expresses that c_1 occurs at some step only if c_2 occur at this step as well. The exclusion constraint $c_1 \# c_2$ specifies that two clocks c_1 and c_2 are exclusive, i.e., they cannot occur simultaneously at the same step.

The union and intersection constraints are used to define clocks. $c_1 \triangleq c_2 + c_3$ defines a clock c_1 such that c_1 ticks iff c_2 or c_3 ticks. Similarly, $c_1 \triangleq c_2 * c_3$ defines a clock c_1 such that c_1 ticks iff both c_2 and c_3 tick. The infimum (resp. supremum) constraint $c_1 \triangleq c_2 \wedge c_3$ (resp. $c_1 \triangleq c_2 \vee c_3$) is used to define a clock c_1 that is the slowest (resp. fastest) clock that is faster (resp. slower) than both c_2 and c_3. These two constraints are useful for expressing delay requirements between two events. Remark that clocks c_1 defined by constraints may correspond to system events, otherwise are auxiliary clocks. In the former case, these constraints can be seen as constraints specifying relations between clocks c_1, c_2 and c_3.

The periodicity constraint $c_1 \triangleq c_2 \propto p$ defines a clock c_1 such that c_1 has to be performed once every p occurrences of clock c_2. It is worth mentioning that the periodicity constraint defined in such a way is relative because of the logical nature of CCSL clocks. That is, clock c_1 is relatively periodic with respect to clock c_2. CCSL does not assume the existence of a global reference clock, most relations are defined relative to other clocks. These notions extend the equivalent behaviors which are usually defined relative to physical time. If c_2 represents a sensor that measures physical time, then c_1 becomes physically periodic.

The filtering constraint $c_1 \triangleq c_2 \blacktriangledown w$ is used to define a clock c_1 which can be seen as snapshots of the clock c_2 at some steps according to the (ω-)regular expression w. For instance, $c_1 \triangleq c_2 \blacktriangledown (01)^\omega$ expresses that c_1 simulates c_2 at every even step. It defines a logically periodic behavior of c_1 with respect to c_2.

The delayFor constraint $c_1 \triangleq c_2\ \$\ d$ (i.e., $c_1 \triangleq c_2\ \$\ d\ on\ c_2$) defines a new clock c_1 that is delayed by the clock c_2 with d steps. The general form $c_1 \triangleq c_2\ \$\ d\ on\ c_3$ defines a new clock c_1 that is delayed by c_2 with d times of the ticks of c_3. c_1 can be seen as a *sampled* clock of c_2 on the basis of c_3. For instance, $c_1 \triangleq c_2\ \$\ 1\ on\ c_3$, denotes that whenever c_2 ticks at least once between two successive ticks of c_3 at steps m and n, c_1 must tick at step n.

3 Scheduling Problem of CCSL

3.1 Schedulability

Given a set Φ of CCSL constraints, a schedule δ satisfies Φ, denoted by $\delta \models \Phi$, iff $\delta \models \phi$ for all constraints $\phi \in \Phi$.

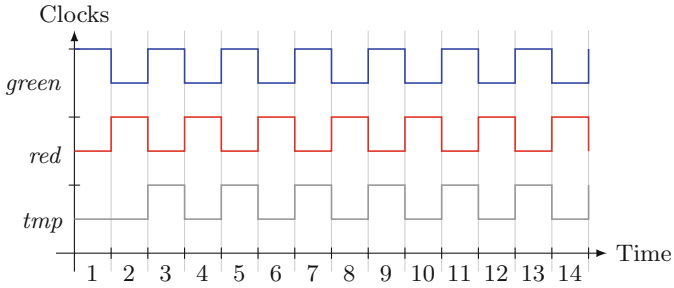

Fig. 2. The unique schedule that satisfies the three constraints in the example

Definition 5 (Logical time scheduling problem). *Given a set Φ of* CCSL *constraints,* the (logical time) scheduling problem *of* CCSL *is to determine whether there exists a schedule δ such that $\delta \models \Phi$.*

We illustrate the scheduling problem by a simple example. Consider alternative flickering between the green and red light using CCSL. We assume that green light starts first. The timing requirements can be formalized by the following three constraints:

$$green \prec red, \qquad tmp \triangleq green \$ 1, \qquad red \prec tmp,$$

where *green* and *red* are clocks respectively representing whether the green (resp. red) light is turned on, the clock *tmp* is an auxiliary clock used to help specify the constraints on clocks.

There exists exactly one schedule satisfying the three constraints, as shown in Fig. 2. In this schedule, the clock *tmp* has the same behavior as *green* from step 2, while the clock *red* has the opposite behavior to *green*. Namely, *red* and *green* operates in an alternative manner. For simplicity, we also write $green \sim red$ to denote the *alternation* relation of the two clocks.

Although one may be able to find one or more schedules for some simple constraints, to our knowledge, there is no generally applicable decision procedure solving the scheduling problem of full CCSL. There are two main challenges. First, schedules are essentially *infinite*, i.e., defined on all the natural numbers. Second, the *precedence* is *stateful*, i.e., it depends on the history, and there is no upper bound on how far in the history one must go back. It may then require an infinite memory to store the history. As a first step to tackle this challenging problem, in this work, we first consider the *bounded* scheduling problem.

3.2 Bounded Scheduling Problem

Given a bound $k \in \mathbb{N}^+$, let $\sigma : \mathbb{N}^+_{\leq k} \to 2^C$ be a function. σ is an *k-bounded schedule* of a set Φ of CCSL constraints, denoted by $\sigma \models_k \Phi$, iff there exists a schedule δ such that $\delta(i) = \sigma(i)$ for every $i \in \mathbb{N}^+_{\leq k}$ and $\delta \models \Phi$ from step 1 up to k, where $\mathbb{N}^+_{\leq k} := \{1, \cdots, k\}$.

Definition 6 (Bounded scheduling problem). *The* bounded scheduling problem *is to determine, for a given set Φ of* CCSL *constraints and a bound k, whether there is an k-bounded schedule σ for Φ, i.e., $\sigma \models_k \Phi$.*

Theorem 1 (Sufficient condition of unschedulability). *If a set Φ of constraints has no k-bounded schedule for some $k \in \mathbb{N}^+$, then Φ is unschedulable.*

The proof is straightforward by contradiction.

It is easy to see that the bounded scheduling problem is decidable, as there are finitely many potential k-bounded schedules, i.e., $(2^{|C|}-1)^k$, where $|C|$ denotes the number of clocks. Furthermore, the satisfiability problem of Boolean formulas can be reduced to the bounded scheduling problem in polynomial time.

Theorem 2. *The k-bounded scheduling problem of* CCSL *is* NP*-complete, even if $k = 1$.*

Proof. The NP upper bound can be proved easily based on the facts that the number of possible k-bounded schedules is finite and the universal quantification $\forall n \in \mathbb{N}^+_{\leq k}$ can be eliminated by enumerating all the possible values in $\mathbb{N}^+_{\leq k}$.

We prove the NP-hardness by a reduction from the satisfiability problem of Boolean formulas which is known NP-complete. Consider the Boolean formula $\phi = \bigwedge_{i=1}^{m}(l_i^1 \vee l_i^2 \vee l_i^3)$, where $m \in \mathbb{N}^+$ and l_i^j for $j \in \{1,2,3\}$ is either a Boolean variable x or its negation $\neg x$. Let $\mathsf{Var}(\phi)$ denote the set of Boolean variables appearing in ϕ. We construct a set of CCSL constraints Φ as follows.

For each $x \in \mathsf{Var}(\phi)$, we have two clocks x^+ and x^-. Let $\mathsf{enc}(x) = x^+$ and $\mathsf{enc}(\neg x) = x^-$. Each clause $l_i^1 \vee l_i^2 \vee l_i^3$ in ϕ is encoded as the CCSL constraint $c_i \triangleq \mathsf{enc}(l_i^1)+\mathsf{enc}(l_i^2)+\mathsf{enc}(l_i^3)$, denoted by ψ_i. Note that $c_i \triangleq \mathsf{enc}(l_i^1)+\mathsf{enc}(l_i^2)+\mathsf{enc}(l_i^3)$ can be transformed into CCSL constraints by introducing one auxiliary clock c, i.e., $\{c_i \triangleq \mathsf{enc}(l_i^1)+\mathsf{enc}(l_i^2)+\mathsf{enc}(l_i^3)\} \equiv \{c_i \triangleq \mathsf{enc}(l_i^1)+c, c \triangleq \mathsf{enc}(l_i^2)+\mathsf{enc}(l_i^3)\}$.

Let $\mathsf{enc}(\phi)$ denote the following set of CCSL constraints

$$\{\mathbf{1} \triangleq *_{i=1}^{m} c_i, \psi_1, ..., \psi_m, x^+ \,\#\, x^-, \mathbf{1} \triangleq x^+ + x^- \mid x \in \mathsf{Var}(\phi)\}$$

where $x^+ \,\#\, x^-$ and $\mathbf{1} \triangleq x^+ + x^-$ enforce that either x^+ or x^- ticks at each step, but not both. This encodes that either x is true or $\neg x$ is true. Note that $\tau \triangleq *_{i=1}^{m} c_i$ is a shorthand of $\tau \triangleq c_1 * \cdots * c_m$, and can also be expressed in CCSL constraints by introducing polynomial number of auxiliary clocks. For instance, $\{c \triangleq c_1 * c_2 * c_3\} \equiv \{c \triangleq c_1 * c', c' \triangleq c_2 * c_3\}$. We can show that ϕ is satisfiable iff $\mathsf{enc}(\phi)$ is 1-bounded schedulable. The satisfiability problem of Boolean formulas is NP-complete, we get that the 1-bounded scheduling problem of CCSL is NP-hard. The k-bounded scheduling problem for $k > 1$ immediately follows by repeating the ticks of clocks at the first step. □

Theorem 2 indicates the time complexity of the bounded scheduling problem. Thus, we need to find practical solutions that are algorithmically efficient for it. In the next section, we propose an SMT-based decision procedure for the bounded scheduling problem and a sound algorithm for the scheduling problem. Thanks to advances in state-of-the-art SMT solvers such as Z3 [25], our approach is usually efficient in practice.

4 Decision Procedure for the Scheduling Problem

4.1 Transformation from CCSL into SMT

Let us fix a set of CCSL constraints Φ defined over a set C of clocks. Each clock $c \in C$ is interpreted as a predicate $t_c : \mathbb{N}^+ \rightarrow \mathsf{Bool}$ such that for all $i \in \mathbb{N}^+$, $t_c(i)$ is true iff the clock c ticks at i, where Bool denotes Boolean sort. A schedule δ of Φ is encoded as a set of predicates $\mathcal{T}_C = \{t_c | c \in C\}$ such that the following condition holds: for all $t_c \in \mathcal{T}_C$,

$$\forall i \in \mathbb{N}^+ . t_c(i) \Leftrightarrow c \in \delta(i).$$

Recalling that schedules forbid stuttering steps, this condition is enforced by restricting the predicates t_c in $\mathcal{T}_C$ to satisfy the following condition:

$$\forall i \in \mathbb{N}^+ . \vee_{c \in C} t_c(i) \tag{F1}$$

Formula F1 specifies that at each step i at least one clock c ticks, i.e., $t_c(i)$ holds.

For each clock $c \in C$, we introduce an auxiliary function $h_c : \mathbb{N}^+ \rightarrow \mathbb{N}$ to encode its history. For each $i \in \mathbb{N}^+$,

$$h_c(i) := \begin{cases} 0, & \text{if } i = 1; \\ h_c(i-1), & \text{if } i > 1 \wedge \neg t_c(i-1); \\ h_c(i-1) + 1, & \text{if } i > 1 \wedge t_c(i-1). \end{cases} \tag{F2}$$

Intuitively, $h_c(i)$ is equivalent to $\chi(c, i)$ for each $i \in \mathbb{N}^+$. The set of all the auxiliary functions is denoted by $\mathcal{H}_C$.

By replacing each occurrence of clock c in $\delta(n)$ (resp. $c \notin \delta(n)$) with $t_c(n)$ (resp. $\neg t_c(n)$) and $\chi(c, n)$ with $h_c(n)$ in the definition of each CCSL constraint, each CCSL constraint ϕ can be encoded as an SMT formula $[\![\phi]\!]$.

We use $[\![\Phi]\!]$ to denote the conjunction of Formulas F1, F2 and the SMT encodings of CCSL constraints in Φ. Formally,

$$[\![\Phi]\!] := \text{F1} \wedge \text{F2} \wedge (\wedge_{\phi \in \Phi} [\![\phi]\!]).$$

Finding a schedule for Φ amounts to finding a solution, i.e., definitions of predicates in $\mathcal{T}_C$, which satisfies $[\![\Phi]\!]$.

Proposition 1. *Φ has a schedule iff $[\![\Phi]\!]$ is satisfiable.*

The scheduling problem of Φ is transformed into the satisfiability problem of the formula $[\![\Phi]\!]$. However, according to the SMT-LIB standard [4], $[\![\Phi]\!]$ belongs to the logic of UFLIA (formulas with Uninterpreted Functions and Linear Integer Arithmetic), whose satisfiability problem is undecidable in general. Nevertheless, the SMT encoding is still useful to solve the bounded scheduling problem, which we will present in the next subsection.

4.2 Decision Procedure for the Bounded Scheduling Problem

For k-bounded scheduling problem, it suffices to consider schedules $\delta : \mathbb{N}^+_{\leq k} \to 2^C$. Moreover, the quantifiers in $[\![\Phi]\!]$ can be eliminated once the bound k is fixed. Hence, we can resort to state-of-the-art SMT solvers. Formally, let $[\![\Phi]\!]_k$ be the formula obtained from $[\![\Phi]\!] = F1 \wedge F2 \wedge (\bigwedge_{\phi \in \Phi} [\![\phi]\!])$ by

- restricting the domain of predicates $t_c \in \mathcal{T}_C$ and functions $h_c \in \mathcal{H}_C$ to $\mathbb{N}^+_{\leq k}$;
- replacing quantifications $\forall n \in \mathbb{N}^+$ and $\exists m \in \mathbb{N}^+$ with $\forall n \in \mathbb{N}^+_{\leq k}$ and $\exists m \in \mathbb{N}^+_{\leq k}$ in $(\bigwedge_{\phi \in \Phi} [\![\phi]\!])$.

Proposition 2. *Φ is k-bounded schedulable iff $[\![\Phi]\!]_k$ is satisfiable.*
Moreover, if $[\![\Phi]\!]_k$ is satisfiable, then $[\![\Phi]\!]_{k'}$ is satisfiable for all $k' \leq k$.

4.3 A Sound Algorithm for the Scheduling Problem

According to Theorem 1, Propositions 1 and 2, (1) if $[\![\Phi]\!]$ is satisfiable, then Φ is schedulable, and (2) if $[\![\Phi]\!]_k$ for some $k \in \mathbb{N}^+$ is unsatisfiable, then Φ is unschedulable. We can deduce a sound algorithm for checking the general scheduling problem. However, randomly choosing a bound k and checking whether or not $[\![\Phi]\!]_k$ is unsatisfiable may be inefficient, as the k-bounded scheduling problem is NP-hard (cf. Theorem 2), and larger bound k may result in time out, but smaller bound k may result in that $[\![\Phi]\!]_k$ is satisfiable. Indeed, if we consider the maximal bound B, then the random approach may have to call SMT solving $\mathbf{O}(B)$ times. Alternatively, we propose a binary-search based approach as shown in Algorithm 1 for a given maximal bound B, which invokes SMT solving at most $\mathbf{O}(|\log_2 B|)$ times.

Algorithm 1: A sound algorithm for the scheduling problem

Input : a set of constraints Φ, a timeout threshold T, a maximal bound B
Output: $\{\texttt{SAT}, \texttt{UNSAT}, \texttt{Timeout}\} \times \mathbb{N}^+$

```
1  result_1 ← SMTSolver([[Φ]], T);
2  if result_1 = SAT then                                   /* Schedulable */
3      return (SAT, 0)
4  l ← 0; u ← B;
5  while l ≤ u do                                           /* Binary search */
6      k ← ⌊(l+u)/2⌋;
7      result_2 ← SMTSolver([[Φ]]_k, T);
8      if result_2 = SAT then l ← k + 1;                    /* Upper half */
9      else                                                 /* Lower half */
10         u ← k − 1;
11         if result_1 = UNSAT ∨ result_2 = UNSAT then
12             result_1 ← UNSAT;
13 if result_2 ≠ SAT then k ← k − 1;
14 return (result_1, k);
```

Given a set Φ of constraints in CCSL, a timeout threshold T and a maximal bound B, Algorithm 1 first invokes an `SMTSolver` to decide whether $[\![\Phi]\!]$ is satisfiable or not within `T` time. If $[\![\Phi]\!]$ is satisfiable, then Algorithm 1 returns (`SAT,0`), meaning that Φ is schedulable. Otherwise, it binary searches a bound $k \leq B$ such that $[\![\Phi]\!]_k$ is satisfiable while $[\![\Phi]\!]_{k+1}$ (if $k+1 \leq B$) is unsatisfiable or cannot be verified in time `T`.

Theorem 3. *Algorithm 1 has the following three properties:*

1. *If it returns* (`SAT,0`)*, then Φ is schedulable.*
2. *If it returns* (`UNSAT,k`)*, then Φ is unschedulable. If $k \neq 0$, then Φ has k-bounded schedulable, otherwise does not have any bounded schedulable.*
3. *If it returns* (`Timeout,k`)*, then Φ is k-bounded schedulable if $k \neq 0$, otherwise no bounded schedule is found for Φ.*

5 Case Study and Performance Evaluation

We implemented our approach in a prototype tool with Z3 [25] as its underlying SMT solver. We conduct a case study on expressing requirements of an interlocking system in CCSL constraints and analyzing its schedulability. Then, we prove 12 algebraic properties of CCSL constraints using the tool. Finally, we evaluate the performance of the tool using 9 sets of CCSL constraints.

5.1 Schedulability of an Interlocking System

The interlocking system is a subsystem of a rail transit system. It is used to prevent trains from collisions and derailments when they are moving under the control of signal lights. As shown in Fig. 3, the interlocking system monitors the occupancy status of the individual track section, and sends signals to inform drivers whether they are allowed to enter the route or not. The railway tracks are divided into sections. Each section is associated with a track circuit for detecting whether it is occupied by a train or not. Signal lights are placed between track sections. They can be red and green to indicate proceeding and stopping, respectively.

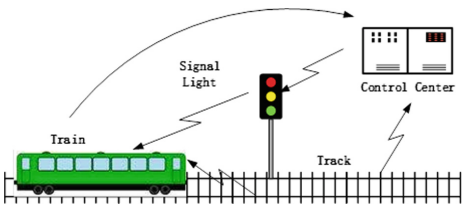

Fig. 3. Interlocking system

The mechanism and operation procedure of the interlocking system are summarized as follows.

1. To enter a track, a train first sends a request to the control center.
2. On receiving the request, the control center sends an inquiry to the track circuit to detect the status of the track.

Table 2. CCSL constraints of the interlocking system

request $\prec$ inquiry	responseOfTrack $\triangleq$ checkSucc + checkFail
checkFail $\prec$ redPulse	responseOfTrain $\triangleq$ enter + wait
redPulse $\preccurlyeq$ showRed	inquiry $\prec$ responseOfTrack
showRed $\prec$ wait	getOccupied $\sim$ getUnoccupied
checkSucc $\prec$ greenPulse	getOccupied # getUnoccupied
greenPulse $\preccurlyeq$ showGreen	request $\sim$ responseOfTrain
showGreen $\prec$ enter	inquiry $-$ responseOfTrack ≤ 40
enter $\prec$ leave	greenPulse $-$ showGreen ≤ 30
enter $\subseteq$ getOccupied	redPulse $-$ showRed ≤ 30
leave $\subseteq$ getUnoccupied	request $-$ responseOfTrain ≤ 50
getOccupied $\sim$ tmp_1	checkFail $-$ showRed ≤ 40
getUnoccupied $\sim$ tmp_1	checkSucc $-$ showGreen ≤ 40
checkFail $\subseteq$ tmp_1	getUnoccupied $\prec$ tmp_2
$tmp_2 \prec$ getOccupied	checkSucc $\subseteq$ tmp_2

3. If the track is occupied, it sends *checkFail* to the control center, and otherwise *checkSucc*.
4. On receiving the message *checkFail* (*resp. checkSucc*), the control center sends a red (*resp.* green) signal pulse to the signal light.
5. The signal light turns red (*resp.* green) on receiving the red (*resp.* green) signal pulse.
6. The train will enter after seeing the light is green, and the track becomes occupied. In case of the red light, the train must stop and wait.
7. The track becomes unoccupied after the train leaves. If the train is waiting, it must send a request again after some time.

There are time constraints on the above operations. For instance, the control center needs to get a response from the track circuit within 30 ms after sending an inquiry to it. The train must make decision within 50 ms after it sends a request to the control center. The light should turn to the corresponding color within 30 ms after it receives a pulse. After the track becomes occupied (*resp.* unoccupied), the light must turn red (*resp.* green) within 40 ms.

Table 2 shows the main logical constraints on the operations in the system and their timing constraints. We use some non-standard constraint expressions for the sake of compactness. Constraint $a - b \leq n$ denotes that b must tick within n steps after a ticks. It equals the set of the following three constraints:

$$a \prec b, \quad t \triangleq a \$ n \; on \; \mathbf{1}, \quad b \preccurlyeq t.$$

Note that in this example the unit of time is millisecond (ms). Thus, there is an implicit assumption in the constraints that every tick of a logic clock means the elapse of one millisecond.

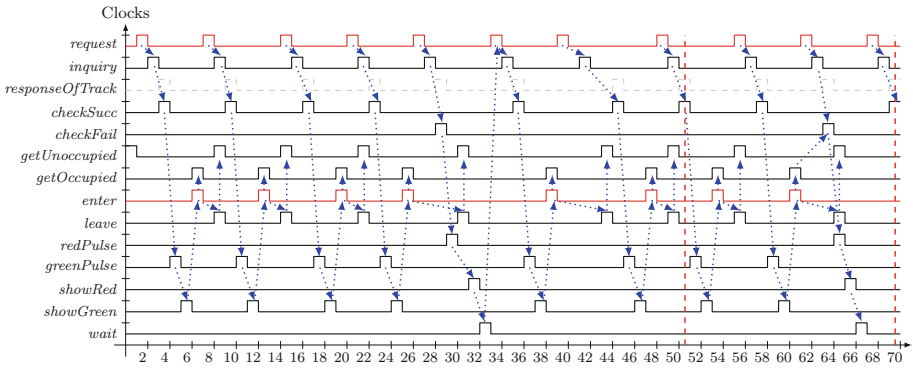

Fig. 4. A bounded schedule for the CCSL constraints in the case study

Most constraints in Table 2 are straightforward, except the six constraints marked with waved underlines. The first three constraints specify that `checkFail` only can occur between the occurrences of `getUnoccupied` and `getOccupied`. The others specify the following two requirements:

1. `checkSucc` only can occur after `getUnoccupied` and before `getOccupied`;
2. `getUnoccupied` precedes `getOccupied`.

Given these constraints, our tool found a bounded schedule as depicted in Fig. 4. From step 1 to step 7, one complete process is finished. Initially, the track gets unoccupied. At step 2, a request is made, which causes subsequent operations to occur from step 3 to step 7. At step 29, a fail case occurs because another train enters (step 26) but has not left (step 31). The train that made the request has to wait (step 33).

If we extend the bounded schedule by infinitely repeating the behaviors of all the clocks between step 51 and 69 from step 70, we obtain an infinite schedule. The extended schedule satisfies all the constraints, and thus it is a witness of the schedulability of designed mechanism for the interlocking system.

In this paper, we are only concerned with the schedulability of the constraints in the example. Some other kinds of temporal properties also need to verify. For instance, we must guarantee that whenever a train requests to enter the station, it must eventually enter. We also need to verify the system is deadlock-free. Such temporal properties can be verified by LTL model checking of CCSL constraints using SMT technique [40]. We omit it because it is beyond the scope of this paper.

5.2 Automatic Proof of CCSL Algebraic Properties

Using the proposed approach, we can also prove automatically algebraic properties of CCSL constraints such as the commutativity of exclusion and transitivity of causality. Algebraic properties of CCSL constraints can be represented as $\Phi \Rightarrow \phi$, where Φ is a set of CCSL constraints and ϕ is a constraint derived from Φ. Proving $\Phi \Rightarrow \phi$ is valid equals proving the unsatisfiability of $[\![\Phi]\!] \wedge \neg[\![\phi]\!]$, which can be solved by Algorithm 1.

Table 3. Proved algebraic properties of CCSL constraints

Algebraic property	Definition
Commutativity of exclusion	$c1 \# c2 \Rightarrow c2 \# c1$
Transitivity of causality	$c_1 \preccurlyeq c_2$, $c_2 \preccurlyeq c_3 \Rightarrow c_1 \preccurlyeq c_3$
Antisymmetry of causality	$c_1 \preccurlyeq c_2$, $c_2 \preccurlyeq c_1 \Rightarrow c_1 = c_2$
Fastness of infimum	$c_1 \triangleq c_2 \wedge c_3 \Rightarrow c_1 \preccurlyeq c_2, c_1 \preccurlyeq c_3$
Slowestness of infimum	$c_1 \triangleq c_2 \wedge c_3, c_4 \preccurlyeq c_2, c_4 \preccurlyeq c_3 \Rightarrow c_4 \preccurlyeq c_1$
Slowness of supremum	$c_1 \triangleq c_2 \vee c_3 \Rightarrow c_2 \preccurlyeq c_1, c_3 \preccurlyeq c_1$
Fastestness of supremum	$c_1 \triangleq c_2 \vee c_3, c_2 \preccurlyeq c_4, c_3 \preccurlyeq c_4 \Rightarrow c_1 \preccurlyeq c_4$
Causality of subclock	$c_1 \subseteq c_2 \Rightarrow c_2 \preccurlyeq c_1$
Causality of union	$c_1 \triangleq c_2 + c_3 \Rightarrow c_1 \preccurlyeq c_2, c_1 \preccurlyeq c_3$
Causality of intersection	$c_1 \triangleq c_2 * c_3 \Rightarrow c_2 \preccurlyeq c_1, c_3 \preccurlyeq c_1$
Subclocking of sampling	$c_1 \triangleq c_2 \lightning c_3 \Rightarrow c_1 \subseteq c_3$
Subclocking of union	$c_1 \triangleq c_2 + c_3 \Rightarrow c_2 \subseteq c_1, c_3 \subseteq c_1$
Subclocking of intersection	$c_1 \triangleq c_2 * c_3 \Rightarrow c_1 \subseteq c_2, c_1 \subseteq c_3$

Let us consider the proof of the slowestness of infimum as an example. The slowestness of infimum means that an infimum constraint $c_1 \triangleq c_2 \wedge c_3$ defines the slowest clock c_1 among those that are faster than both c_2 and c_3.

Proposition 3 (Slowestness of infimum). *Given two clocks c_2, c_3, let $c_1 \triangleq c_2 \wedge c_3$ and c_4 be an arbitrary clock such that $c_4 \preccurlyeq c_2$ and $c_4 \preccurlyeq c_3$, then $c_4 \preccurlyeq c_1$.*

This is proved by transforming CCSL constraints into the following SMT formula according the SMT encoding method:

$$[\![c_1 \triangleq c_2 \wedge c_3]\!] \wedge [\![c_4 \preccurlyeq c_2]\!] \wedge [\![c_4 \preccurlyeq c_3]\!] \wedge \neg[\![c_4 \preccurlyeq c_1]\!].$$

Algorithm 1 returns (`UNSAT`, 0), which means that the formula is proved unsatisfiable. The proposition is proved.

Table 3 lists the algebraic properties that have been successfully proved in our approach. Algebraic properties are useful to help understand the relation among CCSL constraints. Using them we can also verify whether some CCSL constraints are redundant or inconsistent for a given set of CCSL constraints.

5.3 Performance Evaluation

To evaluate the performance our tool, we collected 9 sets of CCSL constraints from the literature and real-world applications, and analyzed their schedulability using our tool. Under different time thresholds, we calculate the maximal bounds under which the constraints are schedulable.

Table 4 shows all the experimental results including the corresponding execution time. All the experiments were conducted on a Win 10 running on an i7 CPU with 2.70 GHz and 16 GB memory. The numbers followed by asterisks

Table 4. Experimental results of bounded schedulability analysis

CS	Clks.	Cons.	THD: 10 s		THD: 20 s		THD: 30 s		THD: 40 s	
			BD	TM	BD	TM	BD	TM	BD	TM
CS1	3	3	8	0.06	8	0.06	8	0.06	8	0.06
CS2	3	4	2^*	0.06	2^*	0.06	2^*	0.06	2^*	0.06
CS3	8	9	48	6.20	59	15.88	70	28.72	75	39.82
CS4	8	7	70	7.12	70	7.12	70	7.12	70	7.12
CS5	9	9	80	8.29	90	19.95	110	26.81	111	39.84
CS6	10	6	95	9.40	113	14.26	113	14.26	113	14.26
CS7	12	9	69	8.80	76	19.42	89	27.69	95	40.00
CS8	17	20	16	0.81	16	0.81	16^*	27.36	16^*	27.36
CS9	27	51	30	9.94	41	17.19	45	29.78	45	29.78

Remarks: CS: constraint set, Cons: the number of constraints, Clks: the number of clocks, THD: timeout threshold, TM: Time (second), BD: upper bound.

are the maximal bounds such that the corresponding constraints are bounded schedulable, but unschedulable in the next step. It is interesting to observe from Table 4 that time cost is loosely related to size (the number of clocks and constraints), thanks to efficient search strategies of SMT solvers. This is in striking contrasts to automata-based [29,35] and the rewriting-based approaches [38], whose scalability suffers from both the numbers of clocks and constraints.

6 Related Work

CCSL is directly derived from the family of synchronous languages, such as Lustre [9], Esterel [6] and Signal [5], and its the scheduling problem of CCSL is akin to what synchronous languages call clock calculus. The main differences are: CCSL is a specification language, while others are programming languages; and CCSL partially describes what is expected to happen in a declarative way and does not give a direct operational deterministic description of what must happen. Furthermore, CCSL only deals with pure clocks while the others deal with signals and extract the clocks when needed.

The Esterel compiler [31] applies a constructive approach to decide when a signal must occur (compute its clock) and what its value should be. This requires a detection of *causality cycles*, or intra-cycle data dependencies, which are also naturally addressed by our approach. However, the Esterel compiler compiles an imperative program into a Boolean circuit, or equivalently a finite state machine. Consequently, it cannot deal with CCSL unbounded schedules.

The clock calculus in Signal attempts to detect whether the specification is endochronous [30], in which case it can generate some efficient code. This analysis is mainly based on the subclock relationship that also exists in CCSL. In CCSL, we consider the problem whether there is at least one possible schedule or not.

In Lustre and its extensions, clocks are regarded as abstract types [13] and the clock calculus computes the relative rates of clocks while rejecting the program when computing the rates is not possible. In most cases, the compiler attempts to build bounded buffers and to ensure that the functional determinism can be preserved with a finite memory. In our case, we do not seek to reach a finite representation, as in the first specification steps this is not a primary goal for the designers. Indeed, this might lead to an over-specification of the problem.

Classical real-time scheduling problem [32] usually relies on task models, arrival patterns and constraints (e.g., precedence, resources) to propose algorithms for the scheduling problem with analytical results [19] or heuristics depending on the specific model (e.g., priorities, preemptive). Other solutions, based on timed automata [1,2,17] or timed Petri nets [8,18], propose a general framework for describing all the relevant aspects without assuming a specific task model. CCSL offers an alternative method based on logical time. It is believed that logical time and multiform time bases offer some flexibility to unify functional requirements and performance constraints. We rely on CCSL and we claim that after encoding a task model in CCSL, finding a schedule for the CCSL model also gives a schedule for the encoded task model [24].

There have been many efforts made towards the scheduling problem of CCSL, though no conclusion is drawn on its decidability. TimeSquare [14] is a simulation tool for CCSL which can produce a possible schedule for a given set of CCSL, up to a given user-defined bound. It also supports different simulation strategies for producing desired execution traces. Some earlier work [20] define the notion of *safe* CCSL specifications that can be encoded with a finite-state machine. The scheduling problem is decidable for safe specifications, as one can merely enumerate all the (finite) solutions. A semi-algorithm can build the finite representation when the specification is safe [21]. In [37], Zhang et al. proposed a state-based approach and a sufficient condition to decide whether safe and unsafe specifications accept a so-called *periodic schedule* [39]. This allows to build a finite solution for unsafe specifications, while there may also exist infinite solutions. Xu et al. proposed a notion of *divergence* of CCSL to study the schedulability of CCSL, and proved that a set of CCSL constraints is schedulable if all the constraints are divergent [34]. They resorted to the theorem prover PVS [27] to assist the divergence proof.

The scheduling problem of CCSL constraints in this work resorts to SMT solving to deal with the bounded and unbounded schedules. Using SMT solving has two advantages: (1) it is usually efficient in practice, and (2) it can deal with unsafe CCSL constraints such as infimum and supremum [21].

Some basic algebraic properties on CCSL relations have been established manually before [23] but we provide here an automatic framework to do so.

7 Conclusion and Future Work

In this work, we proved that the bounded scheduling problem of CCSL is NP-complete, and proposed an SMT-based decision procedure for the bounded

scheduling problem. The procedure is sound and complete. The experimental results also show its efficiency in practice. Based on this decision procedure, we devised a sound algorithm for the general scheduling problem. We evaluated the effectiveness of the proposed approach on an interlocking system. We also showed our approach can be used to prove algebraic properties of CCSL constraints.

Our approach to the bounded scheduling problem of CCSL makes us one step closer to tackling the general (i.e. unbounded) scheduling problem. As the case study demonstrates, one may find an infinite schedule by extending a bounded one such that the extended infinite schedule still satisfies the constraints. This observation inspires future work to investigate mechanisms of finding such bounded schedules, hopefully with SMT solvers by extending our algorithm. In our earlier work [37], we proposed a similar approach to search for periodical schedules in bounded steps. In that approach, CCSL constraints are transformed into finite state machine and consequently suffers from the state explosion problem. We believe our SMT-based approach can be extended to their work while still avoiding state explosion. We leave it to future work.

References

1. Abdeddaïm, Y., Asarin, E., Maler, O.: Scheduling with timed automata. Theor. Comput. Sci. **354**(2), 272–300 (2006)
2. Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: TIMES: a tool for schedulability analysis and code generation of real-time systems. In: Larsen, K.G., Niebert, P. (eds.) FORMATS 2003. LNCS, vol. 2791, pp. 60–72. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-40903-8_6
3. André, C., Mallet, F., de Simone, R.: Modeling time(s). In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 559–573. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75209-7_38
4. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB standard (2016)
5. Benveniste, A., Guernic, P.L., Jacquemot, C.: Synchronous programming with events and relations: the SIGNAL language and its semantics. Sci. Comput. Program. **16**(2), 103–149 (1991)
6. Berry, G., Gonthier, G.: The esterel synchronous programming language: design, semantics, implementation. Sci. Comput. Program. **19**(2), 87–152 (1992)
7. Boehm, B., Basili, V.R.: Software defect reduction top 10 list. Computer **34**(1), 135–137 (2001)
8. Bucci, G., Fedeli, A., Sassoli, L., Vicario, E.: Modeling flexible real time systems with preemptive time petri nets. In: Proceedings of the 15th ECRTS, Porto, Portugal, pp. 279–286. IEEE (2003)
9. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.: LUSTRE: a declarative language for programming synchronous systems. In: Proceedings of 14th POPL, Tucson, USA, pp. 178–188. ACM Press (1987)
10. Chen, X., Yin, L., Yu, Y., Jin, Z.: Transforming timing requirements into CCSL constraints to verify cyber-physical systems. In: Duan, Z., Ong, L. (eds.) ICFEM 2017. LNCS, vol. 10610, pp. 54–70. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68690-5_4
11. Chen, Y., Chen, Y., Madelaine, E.: Timed-pNets: a communication behavioural semantic model for distributed systems. Front. Comput. Sci. **9**(1), 87–110 (2015)

12. Colaço, J., Pagano, B., Pouzet, M.: SCADE 6: a formal language for embedded critical software development. In: Proceedings of the 11th TASE, Sophia Antipolis, France, pp. 1–11. IEEE (2017)
13. Colaço, J.-L., Pouzet, M.: Clocks as first class abstract types. In: Alur, R., Lee, I. (eds.) EMSOFT 2003. LNCS, vol. 2855, pp. 134–155. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45212-6_10
14. Deantoni, J., Mallet, F.: TimeSquare: treat your models with logical time. In: Proceedings of the 50th TOOLS, Prague, Czech Republic, pp. 34–41. IEEE (2012)
15. Feiler, P.H., Gluch, D.P.: Model-based engineering with AADL - an introduction to the SAE architecture analysis and design language. SEI, Addison-Wesley (2012)
16. Kang, E., Schobbens, P.: Schedulability analysis support for automotive systems: from requirement to implementation. In: Proceedings of the 29th SAC, Gyeongju, Korea, pp. 1080–1085. ACM (2014)
17. Krčál, P., Yi, W.: Decidable and undecidable problems in schedulability analysis using timed automata. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 236–250. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_20
18. Lime, D., Roux, O.: A translation based method for the timed analysis of scheduling extended time petri nets. In: Proceedings of the 25th RTSS, pp. 187–196. IEEE (2004)
19. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. J. ACM **20**(1), 46–61 (1973)
20. Mallet, F., Millo, J.-V.: Boundness issues in CCSL specifications. In: Groves, L., Sun, J. (eds.) ICFEM 2013. LNCS, vol. 8144, pp. 20–35. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41202-8_3
21. Mallet, F., de Simone, R.: Correctness issues on MARTE/CCSL constraints. Sci. Comput. Program. **106**, 78–92 (2015)
22. Mallet, F., Villar, E., Herrera, F.: MARTE for CPS and CPSoS. In: Nakajima, S., Talpin, J.-P., Toyoshima, M., Yu, H. (eds.) Cyber-Physical System Design from an Architecture Analysis Viewpoint, pp. 81–108. Springer, Singapore (2017). https://doi.org/10.1007/978-981-10-4436-6_4
23. Mallet, F., Millo, J., de Simone, R.: Safe CCSL specifications and marked graphs. In: Proceedings of the 11th MEMOCODE, Portland, OR, USA, pp. 157–166. IEEE (2013)
24. Mallet, F., Zhang, M.: Work-in-progress: from logical time scheduling to real-time scheduling. In: Proceedings of the 39th RTSS, Nashville, USA, pp. 143–146. IEEE (2018)
25. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
26. OMG: UML profile for MARTE: modeling and analysis of real-time embedded systems (2015)
27. Owre, S., Rushby, J.M., Shankar, N.: PVS: a prototype verification system. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 748–752. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55602-8_217
28. Peters, J., Przigoda, N., Wille, R., Drechsler, R.: Clocks vs. instants relations: verifying CCSL time constraints in UML/MARTE models. In: Proceedings of the 14th MEMOCODE, Kanpur, India, pp. 78–84. IEEE (2016)
29. Peters, J., Wille, R., Przigoda, N., Kühne, U., Drechsler, R.: A generic representation of CCSL time constraints for UML/MARTE models. In: Proceedings of the 52nd DAC, pp. 122:1–122:6. ACM (2015)

30. Potop-Butucaru, D., Caillaud, B., Benveniste, A.: Concurrency in synchronous systems. Formal Methods Syst. Des. **28**(2), 111–130 (2006)
31. Potop-Butucaru, D., Edwards, S.A., Berry, G.: Compiling Esterel. Springer, Boston (2007). https://doi.org/10.1007/978-0-387-70628-3
32. Sha, L., et al.: Real time scheduling theory: a historical perspective. Real-Time Syst. **28**(2–3), 101–155 (2004)
33. Suryadevara, J., Seceleanu, C., Mallet, F., Pettersson, P.: Verifying MARTE/CCSL mode behaviors using UPPAAL. In: Hierons, R.M., Merayo, M.G., Bravetti, M. (eds.) SEFM 2013. LNCS, vol. 8137, pp. 1–15. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40561-7_1
34. Xu, Q., de Simone, R., DeAntoni, J.: Divergence detection for CCSL specification via clock causality chain. In: Fränzle, M., Kapur, D., Zhan, N. (eds.) SETTA 2016. LNCS, vol. 9984, pp. 18–37. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47677-3_2
35. Yin, L., Mallet, F., Liu, J.: Verification of MARTE/CCSL time requirements in Promela/SPIN. In: Proceedings of the 16th ICECCS, USA, pp. 65–74. IEEE (2011)
36. Yu, H., Talpin, J., Besnard, L., et al.: Polychronous controller synthesis from MARTE/CCSL timing specifications. In: Proceedings of the 9th MEMOCODE, Cambridge, UK, pp. 21–30. IEEE (2011)
37. Zhang, M., Dai, F., Mallet, F.: Periodic scheduling for MARTE/CCSL: theory and practice. Sci. Comput. Program. **154**, 42–60 (2018)
38. Zhang, M., Mallet, F.: An executable semantics of clock constraint specification language and its applications. In: Artho, C., Ölveczky, P.C. (eds.) FTSCS 2015. CCIS, vol. 596, pp. 37–51. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-29510-7_2
39. Zhang, M., Mallet, F., Zhu, H.: An SMT-based approach to the formal analysis of MARTE/CCSL. In: Ogata, K., Lawford, M., Liu, S. (eds.) ICFEM 2016. LNCS, vol. 10009, pp. 433–449. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47846-3_27
40. Zhang, M., Ying, Y.: Towards SMT-based LTL model checking of clock constraint specification language for real-time and embedded systems. In: Proceedings of the 18th LCTES, Barcelona, Spain, pp. 61–70. ACM (2017)