Science of Computer Programming 202 (2021) 102546

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

A clock-based dynamic logic for schedulability analysis of CCSL specifications ☆

Yuanrui Zhang [a,b,c], Frédéric Mallet [d], Huibiao Zhu [e], Yixiang Chen [c], Bo Liu [b], Zhiming Liu [b,*]

[a] School of Mathematics and Statistics, Southwest University, China
[b] RISE, College of Computer & Information Science, Southwest University, China
[c] MoE Engineering Research Center for Software/Hardware Co-design Technology and Application, East China Normal University, China
[d] I3S Laboratory, UMR 7271 CNRS, INRIA, Université Nice Sophia Antipolis, France
[e] Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, China

ARTICLE INFO

Article history:
Received 19 November 2019
Received in revised form 17 September 2020
Accepted 17 September 2020
Available online 14 October 2020

Keywords:
Clock constraint specification language
Dynamic logic
Real-time embedded systems
Schedulability analysis
Theorem proving

ABSTRACT

The Clock Constraint Specification Language (CCSL) is a clock-based formalism for the specification and analysis of real-time embedded systems. The major goal of schedulability analysis of CCSL specifications is to solve the schedule problem, which is to answer 'whether there exists a clock behaviour (also called a 'schedule') that conforms to a given CCSL specification'. Existing works on schedulability analysis of CCSL specifications are mainly based on model checking or SMT-solving. In this paper, however, we propose a theorem-proving approach to the problem. To this end, we define a clock-based dynamic logic (cDL) in which we can specify the clock behaviours and the clock relations in CCSL. With cDL, given a CCSL specification *SP*, we can express its schedule problem as a cDL formula ϕ_{sp}. Then solving the schedule problem is equivalent to checking the validity of ϕ_{sp} in the proof system of cDL. By analyzing the proof tree of ϕ_{sp}, we can generate a concrete schedule satisfying *SP*. Compared to the previous approaches, our method is not limited to special types of CCSL specifications and schedules and does not depend on the bounds that are set for approximate checking. We implement our cDL in Coq. We use an example throughout the paper to illustrate our method.

1. Introduction

The clock constraint specification language (CCSL) [2] is a specification language for specifying the constraints between the occurrences of events in real-time embedded systems (RTESs). It was firstly defined as an annex of UML/MARTE [3], but later developed as an independent language equipped with a formal semantics [4]. CCSL gives a concrete syntax to deal with logical clocks, made popular by Leslie Lamport [5] and synchronous languages (such as Esterel). In CCSL, 'clocks' are treated as first-class citizens for capturing discrete-time events, and clock expressions are used for specifying the logical and chronometrical constraints between the occurrences of events. CCSL is a specification language and not a programming

☆ This paper is an extended version of the conference paper [1] at TASE 2019: 13th International Symposium on Theoretical Aspects of Software Engineering.

* Corresponding author.
E-mail addresses: zhangyrmath@126.com (Y. Zhang), zhimingliu88@swu.edu.cn (Z. Liu).

https://doi.org/10.1016/j.scico.2020.102546
0167-6423/

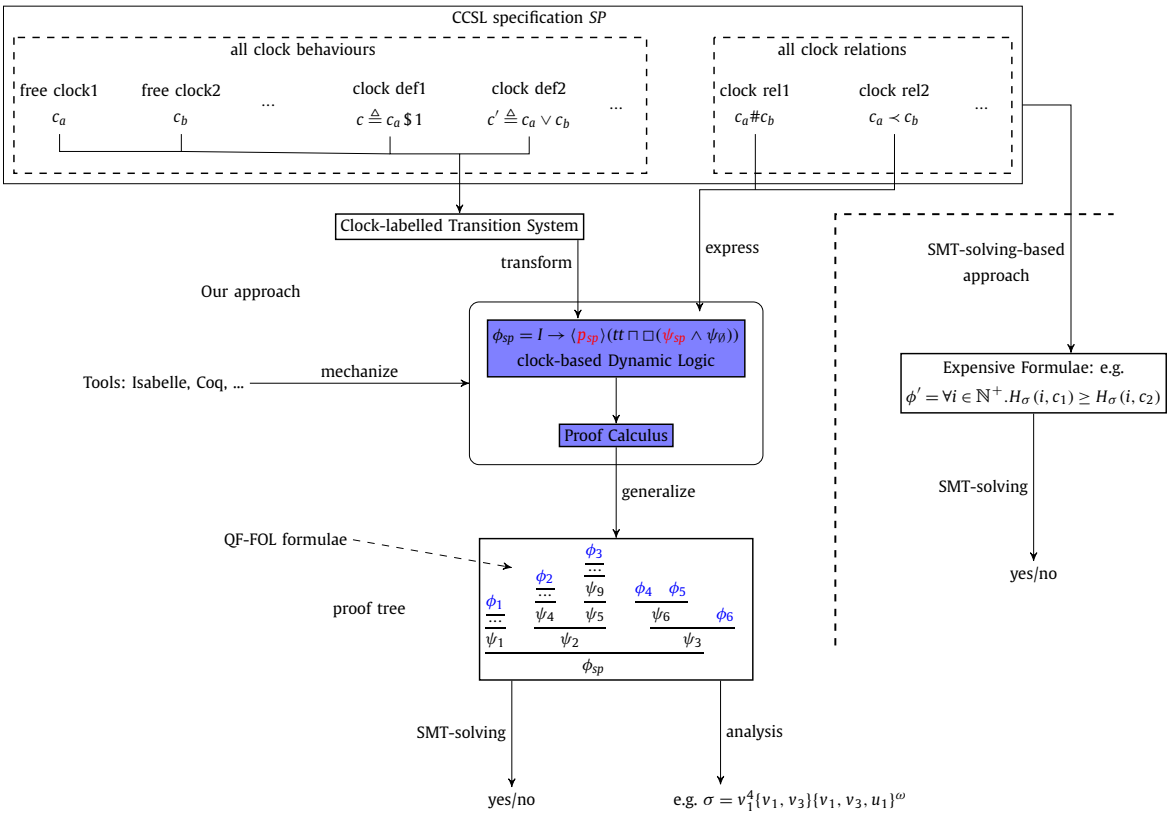


Fig. 1. Theorem-proving approach for schedulability analysis of CCSL specifications.

language. It only allows an abstract specification of a set of possible behaviours and does not attempt to provide a single operational deterministic execution. All the values are ignored to focus only on clock issues. CCSL has been widely used in the specification and analysis of different RTESs, e.g. see [6–8].

One important aspect in the formal analysis of CCSL is schedulability analysis of CCSL specifications. The major goal of the analysis is to answer whether 'there exists a schedule for a given CCSL specification'. Here a schedule is a clock behaviour expressed as a sequence of the occurrences of clocks (whose formal definition is given in Sect. 2). The decidability of this problem still remains open [9]. However, there are existing methods based on model checking and SMT-solving that give a partial solution of this problem [10,11,9,12]. The model-checking-based approach [10,11] relies on a transformation from CCSL specifications into finite automata, but only a part of CCSL specifications whose corresponding automata are finite (which are also called 'safe CCSL specifications' [13]) can be treated in this way. On the other hand, [9,12] proposed an SMT-solving-based approach which relies on encoding a CCSL specification directly into a first-order logical (FOL) formula. In this approach, the authors focus on searching a type of schedules called 'periodic schedules' [9] for a given CCSL specification by SMT-solving the FOL formula. However, solving the FOL formula always needs to set a bound (a positive number that decides the iterative steps in the SMT-solving procedure) and the result of the search of the schedules depends on this bound.

In this paper, we propose a theorem-proving approach to schedulability analysis of CCSL specifications, which is not limited to special types of CCSL specifications and does not depend on the bounds set to FOL formulae. To this end, we define a variation of dynamic logic called 'clock-based dynamic logic' (cDL) and develop a proof calculus in order to specify and analyze the schedule problem. We build cDL by extending first-order dynamic logic (FODL) [14] with clocks as primitives and inheriting the so-called 'normalized trace formula' of differential temporal logic^2 (dTL^2) [15]. With these features the schedule problem of a CCSL specification can be specified as a cDL dynamic formula. cDL supports both modelling the dynamic clock behaviour of the specification as its program model and specifying the static clock relations of the specification as its logical expression in a single formalism. The key idea behind our approach is that, the proof in cDL makes use of the syntactical structure of cDL program models so that a cDL dynamic formula can be decomposed into quantifier-free FOL (QF-FOL) formulae (i.e. verification conditions), which in turn can be proved using SMT-solving. With cDL the schedulability analysis can be made at an abstraction level where the concept of clock and the synchronous execution mechanism can be fully stressed.

Our method is illustrated in Fig. 1. A CCSL specification *SP* consists of a set of clocks and a set of relations between clocks. The schedule problem of *SP* can be captured as a dynamic formula ϕ_{sp} in cDL. This is achieved by transforming all

clocks whose behaviours can be captured as a clock-labelled transition system (see Sect. 2.3) into a cDL program model p_{sp}, and expressing all clock relations as a cDL formula ψ_{sp}. In this way, we reduce the schedule problem of *SP* into a verification problem of the cDL formula ϕ_{sp}. With the inference rules of cDL, ϕ_{sp} can be proven with the generation of a proof tree. All verification conditions on the leaf nodes of the proof tree are QF-FOL formulae, which can thus be solved by an SMT-solving procedure [16]. By analyzing the proof tree, a concrete schedule that satisfies the specification can be generated (e.g. schedule σ). Except for the process of generating the proof tree, all other procedures can be carried out automatically. The whole cDL system can be mechanized by popular theorem provers like Isabelle [17], Coq [18], etc.

Compared to the previous approaches mentioned above, the whole analysis process of our approach does not rely on state exploration so it does not require the specification *SP* to be a safe one. And since all verification conditions are quantifier-free, we do not need to set a bound for an approximate solving. This is different from the existing SMT-solving-based approach mentioned above, where the FOL formula that directly encodes the schedule problem contains quantifiers (e.g. formula ϕ' in Fig. 1) and a bound has to be set in order to eliminate them. Because of this reason, in our approach the search for a schedule does not depend on the bound but on whether we can prove the cDL formula that captures the schedule problem.

To summarize, the contributions of this paper are:

(i) We construct cDL and its proof calculus, and we mechanize cDL using Coq.
(ii) We propose a method for schedulability analysis of CCSL specifications based on cDL.

For (ii), we focus on the encoding from CCSL specifications into cDL formulae and how the schedule problem can be solved through derivations of cDL.

cDL is partially based on 'CCSL dynamic logic' [19], which is designed for characterization and verification of a simple CCSL specification $\widetilde{Rel}$ (consisting of a set of clock relations) of a given synchronous system p. In CCSL dynamic logic, clock relation *Rel* is taken as a primitive, and the satisfaction of the specification can be captured as a formula of the form '$[p] \lambda (Rel_1, ..., Rel_n)$'. In cDL we use temporal formulae of the form '$\Box\psi$' to express clock relations, which is more general than $[p] \lambda (Rel_1, ..., Rel_n)$. It is known that CCSL dynamic logic is not expressive enough to handle the schedule problem in CCSL, which can be expressed as a normalized trace formula of the form '$\langle p\rangle\phi \sqcap \Box\psi$' in cDL (see Sect. 4). Normalized trace formulae can express the existence of a trace satisfying both a state property[1] (ϕ) and a temporal property ($\Box\psi$).

This paper is an extended version of the conference paper [1], where cDL was defined and the method for schedulability analysis of CCSL specifications was proposed. There the algorithm for analyzing the proof tree of a valid cDL formula can only generate a bounded schedule, which is, a finite prefix of a schedule. In this paper, we take one step further by improving the algorithm there, so that the modified algorithm (Algorithm 2) can generate a complete schedule. Furthermore, this paper is more comprehensive with well and completely defined preliminaries, introductions to FODL and dTL^2, and a full definition of the substitution in cDL (Sect. 4.2). Also, we add the proofs of important propositions (Appendix A) and of the soundness of the cDL proof system (Appendix B). Graphical illustrations of some concepts and examples are provided throughout the paper for better understanding (e.g. Examples 2.3, 4.3, 4.4 and Example 5.2).

The rest of this paper is organized as follows: Sect. 2 briefly introduces the formalism CCSL, which is necessary for understanding the content of this paper. Sect. 3 introduces an illustrative example, which is used throughout the paper to explain our contributions. We define in Sect. 4 the syntax and semantics of cDL. In Sect. 5, we present the proof system of cDL and analyze its soundness, completeness and decidability. Sect. 6 proposes a method for schedulability analysis of CCSL in cDL. In Sect. 7 we discuss our implementation of cDL in Coq, and give an example of how cDL formulae can be captured and proved in Coq. We discuss the related work in Sect. 8, and draw our conclusions in Sect. 9 with a discussion about possible future work.

2. The clock constraint specification language

The version of CCSL presented here is based on [9,20]. CCSL consists of two parts: logical clocks and the constraints between clocks. In Sect. 2.1, we introduce the logical clock and the related concepts. In Sect. 2.3, we introduce the constraints between clocks and the related concepts. We give the semantics of CCSL and the definitions of CCSL specifications and the schedule problem. In Sect. 2.3 we introduce clock-labelled transition systems — an automata semantics of CCSL.

2.1. Logical clock

Logical clock In CCSL, a logical clock captures the occurrences of an event in RTESs over a discrete time model. It is an infinite sequence, defined as a function $c : \mathbb{N}^+ \to \{0, 1\}$, where $\mathbb{N}^+ = \{1, 2, ..., n, ...\}$ is the set of natural numbers. Each $c(i)$ ($i \in \mathbb{N}^+$) can be either 'tick' (represented as 1) or 'idle' (represented as 0), representing that the event associated to c occurs or not at the instant i. We use $\mathcal{C}$ to denote a finite set of clocks.

[1] I.e., a property that is evaluated at a state.

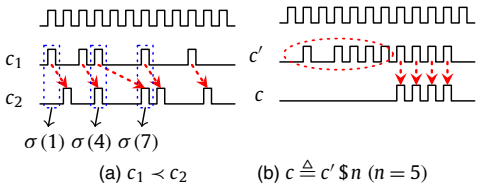


Fig. 2. A possible schedule of CCSL constraints.

Clock sequence & Schedule Given a set of clocks $\mathcal{C}$, the state of clocks in $\mathcal{C}$ at an instant captures the ticks of all clocks in $\mathcal{C}$ at that instant. It can be denoted as a function $\eta : \mathcal{C} \to \{0, 1\}$. Given a set of clocks $\mathcal{C}$, a clock sequence (or simply 'sequence') is a finite or infinite sequence of states of clocks in $\mathcal{C}$. It is defined as a down-closed partial function $\kappa : \mathbb{N}^+ \twoheadrightarrow (\mathcal{C} \to \{0, 1\})$, which satisfies for any $i \in \mathbb{N}^+$, if $i \in dom(\kappa)$, then for any $j < i$, $j \in dom(\kappa)$.

Since for any function η, there exists exactly one set of ticked clocks: $\alpha^\eta = \{c \mid \eta(c) = 1\}$ corresponding to it, we can use α^η to denote η. Hence a sequence can be also written as: $\kappa = \alpha_1\alpha_2...\alpha_n...$. If a set of ticked clocks $\alpha = \{c\}$, we simply write it as c.

A schedule is an infinite clock sequence, denoted by σ. A finite clock sequence is also called a 'bounded schedule' in some references, e.g. [9,12].

Configuration Given a set of clocks $\mathcal{C}$, the number of times each clock has ticked at an instant is denoted as a function $h : \mathcal{C} \to \mathbb{N}$ (where $\mathbb{N} = \mathbb{N}^+ \cup \{0\}$). Given a set of clocks $\mathcal{C}$ and a clock sequence κ, a configuration H_κ is an infinite sequence that keeps track of the number of times each clock has ticked at each instant in clock sequence κ. It is defined as a function $H_\kappa : \mathbb{N} \to (\mathcal{C} \to \mathbb{N})$ s.t.:

$$H_\kappa(i, c) =_{df} \begin{cases} 0, & \text{if } i = 0 \\ H_\kappa(i-1, c) + 1, & \text{if } i > 0, c \in \kappa(i) \\ H_\kappa(i-1, c), & \text{if } i > 0, c \notin \kappa(i). \end{cases}$$

$H_\kappa(0, c) = 0$ indicates that at the beginning no clock ticks.

Example 2.1. In Fig. 2(a), there are two clocks: c_1, c_2, $\mathcal{C} = \{c_1, c_2\}$. Clock $c_1 = 101100100100...$, clock $c_2 = 010100110010...$. Schedule $\sigma = \{c_1\}\{c_2\}\{c_1\}\{c_1, c_2\}\emptyset\emptyset\{c_1, c_2\}\{c_2\}\emptyset\{c_1\}\{c_2\}\emptyset...$, where for example we have $\sigma(1) = \{c_1\}$, $\sigma(4) = \{c_1, c_2\}$ and $\sigma(7) = \{c_1, c_2\}$ (they are indicated by the dashed rectangle in Fig. 2). The configuration H_σ for example satisfies: $H_\sigma(0, c_1) = 0$, $H_\sigma(1, c_1) = 1$, $H_\sigma(2, c_1) = 1$, $H_\sigma(3, c_1) = 2$.

2.2. Clock constraint

Clock Constraint In CCSL, a clock constraint captures a constraint between clocks. It can be either a clock relation or a clock definition.

Clock relations describe binary relationships between clocks, their syntax is defined as:

$$Rel ::= c_1 \prec c_2 \mid c_1 \preceq c_2 \mid c_1 \subseteq c_2 \mid c_1 \# c_2,$$

where c_1, c_2 are arbitrary clocks. The semantics of clock relations $\sigma \vDash_{ccsl} Rel$ is defined in Fig. 3. 'Causality' and 'Precedence' describe a asynchronous dependence relation between two events. 'Causality' captures a constraint between c_1 and c_2 in which the ticks of clock c_2 are caused by the ticks of clock c_1, in other words, c_2 cannot tick before c_1 ticks. 'Precedence' captures a similar constraint as 'Causality' but it does not allow that two clocks tick at the same instant, in other words, it expresses that c_1 ticks strictly faster than c_2 ticks. 'Subclock' and 'Exclusion' describe a synchronous relation between two events. 'Subclock' captures a constraint between c_1 and c_2 in which c_1 can only tick if c_2 ticks at the same instant; 'Exclusion' captures a constraint between c_1 and c_2 in which c_1, c_2 cannot tick at the same instant.

Clock definitions define new clocks by composing the existing clocks in different ways. A clock definition is of the form:

$$Cdf ::= c \triangleq E$$

where E is a clock expression defined by the following grammar:

$$E ::= c_1 + c_2 \mid c_1 * c_2 \mid c_1 \blacktriangleright c_2 \mid c_1 \rhd c_2 \mid c_1 \curvearrowright c_2 \mid c \propto n \mid c \$ n \mid c_1 \vee c_2 \mid c_1 \wedge c_2.$$

c_1, c_2 are arbitrary clocks, $n \geq 1$. The semantics of clock definitions $\sigma \vDash_{ccsl} Cdf$ are defined in Fig. 3. 'Union' defines a clock c that ticks at any instant when either c_1 or c_2 ticks. 'Intersection' defines a clock c that ticks at any instant when both c_1 and c_2 tick. 'Sample' defines a clock c that samples c_1 based on c_2. c ticks at any instant when c_2 ticks for the first time after a tick of c_1. 'Strict Sample' is similar to 'Sample', but it does not allow clock c to tick at any instant when both c_2 and c_1 tick. 'Interruption' defines a clock c that ticks at all instants when c_1 ticks until c_2 ticks for the first time. 'Periodicity'

Causality:	$\sigma \vDash_{ccsl} c_1 \preceq c_2$ iff $\forall i \in \mathbb{N}^+.H_\sigma(i,c_1) \geq H_\sigma(i,c_2)$
Precedence:	$\sigma \vDash_{ccsl} c_1 \prec c_2$ iff $\forall i \in \mathbb{N}^+.H_\sigma(i,c_1) > H_\sigma(i,c_2) \vee (H_\sigma(i,c_1) = H_\sigma(i,c_2) \wedge c_1 \notin \sigma(i))$
Subclock:	$\sigma \vDash_{ccsl} c_1 \subseteq c_2$ iff $\forall i \in \mathbb{N}^+.c_1 \in \sigma(i) \rightarrow c_2 \in \sigma(i)$
Exclusion:	$\sigma \vDash_{ccsl} c_1 \# c_2$ iff $\forall i \in \mathbb{N}^+.c_1 \notin \sigma(i) \vee c_2 \notin \sigma(i)$
Union:	$\sigma \vDash_{ccsl} c \triangleq c_1 + c_2$ iff $\forall i \in \mathbb{N}^+.c \in \sigma(i) \leftrightarrow (c_1 \in \sigma(i) \vee c_2 \in \sigma(i))$
Intersection:	$\sigma \vDash_{ccsl} c \triangleq c_1 * c_2$ iff $\forall i \in \mathbb{N}^+.c \in \sigma(i) \leftrightarrow (c_1 \in \sigma(i) \wedge c_2 \in \sigma(i))$
Strict Sample:	$\sigma \vDash_{ccsl} c \triangleq c_1 \blacktriangleright c_2$ iff $\forall i \in \mathbb{N}^+.c \in \sigma(i) \leftrightarrow ($ $c_2 \in \sigma(i) \wedge$ $\exists j.(0 < j < i) \wedge c_1 \in \sigma(j) \wedge (\forall k.(j \leq k < i) \rightarrow c_2 \notin \sigma(k))$ $)$
Sample:	$\sigma \vDash_{ccsl} c \triangleq c_1 \rhd c_2$ iff $\forall i \in \mathbb{N}^+.c \in \sigma(i) \leftrightarrow ($ $c_2 \in \sigma(i) \wedge$ $\exists j.(0 < j \leq i) \wedge c_1 \in \sigma(j) \wedge (\forall k.(j \leq k < i) \rightarrow c_2 \notin \sigma(k))$ $)$
Interruption:	$\sigma \vDash_{ccsl} c \triangleq c_1 \curvearrowright c_2$ iff $\forall i \in \mathbb{N}^+.c \in \sigma(i) \leftrightarrow (c_1 \in \sigma(i) \wedge \forall j.(0 < j \leq i) \rightarrow c_2 \notin \sigma(j))$
Periodicity:	$\sigma \vDash_{ccsl} c \triangleq c' \propto n$ iff $\forall i \in \mathbb{N}^+.c \in \sigma(i) \leftrightarrow (c' \in \sigma(i) \wedge \exists m \in \mathbb{N}^+.H_\sigma(i,c') = m \cdot (n+1))$
Delay:	$\sigma \vDash_{ccsl} c \triangleq c' \$ n$ iff $\forall i \in \mathbb{N}^+.H_\sigma(i,c) = max(H_\sigma(i,c') - n, 0)$
Infimum:	$\sigma \vDash_{ccsl} c \triangleq c_1 \wedge c_2$ iff $\forall i \in \mathbb{N}^+.H_\sigma(i,c) = max(H_\sigma(i,c_1), H_\sigma(i,c_2))$
Supremum:	$\sigma \vDash_{ccsl} c \triangleq c_1 \vee c_2$ iff $\forall i \in \mathbb{N}^+.H_\sigma(i,c) = min(H_\sigma(i,c_1), H_\sigma(i,c_2))$

Fig. 3. Semantics of CCSL.

defines a clock c that ticks every n ticks of clock c'. c ticks at any instant when c' ticks, and there are n ticks of c' (not including the tick of c' at the instant) after the last instant when c ticks. 'Delay' defines a clock c that ticks at any instant when c' ticks and before the instant c' has ticked for n or more than n times. 'Infimum' defines the slowest clock that is faster than both c_1 and c_2. c ticks at any instant when the faster clock between c_1 and c_2 ticks. 'Supremum' defines the fastest clock that is slower than both c_1 and c_2. c ticks at any instant when the slower clock between c_1 and c_2 ticks.

Example 2.2. Fig. 2(a) illustrates the clock relation $c_1 \prec c_2$. We can see that the tick of c_2 always depends on the tick of c_1 (the dependence relation is indicated by the red arrows) and c_1 and c_2 do not tick at the same instant. At each instant the semantics of Precedence is satisfied, e.g., at the instant 3, $H_\sigma(3,c_1) > H_\sigma(3,c_2)$.

Fig. 2(b) illustrates the clock definition $c \triangleq c' \$ n$ (when $n = 5$).

Clock Specification & Free Clock Given a set of clock constraints C, $\sigma \vDash_{ccsl} C$ is defined s.t. $\sigma \vDash_{ccsl} cn$ for all $cn \in C$. A CCSL specification is a pair

$$SP ::= \langle \widetilde{Cdf}, \widetilde{Rel} \rangle,$$

where $\widetilde{Cdf}$ is a set of clock definitions, $\widetilde{Rel}$ is a set of clock relations. $\sigma \vDash_{ccsl} SP$ is defined s.t. $\sigma \vDash_{ccsl} \widetilde{Rel}$ and $\sigma \vDash_{ccsl} \widetilde{Cdf}$. We use $\mathcal{C}(SP)$ to denote all clocks appearing in SP.

Given a CCSL specification $SP = \langle \widetilde{Cdf}, \widetilde{Rel} \rangle$, we use $\mathcal{F}(SP)$ to denote the set of all free clocks appearing in $\widetilde{Cdf} \cup \widetilde{Rel}$. A 'free clock' is a clock that does not appear on the left side of any clock definitions of the form '$c \triangleq E$' in a specification.

Schedule Problem Given a CCSL specification $SP = \langle \widetilde{Cdf}, \widetilde{Rel} \rangle$, the schedule problem is to determine whether 'there exists a schedule σ of $\mathcal{C}(SP)$ s.t. $\sigma \vDash_{ccsl} SP$'. In a schedule of a CCSL specification, we are only interested in those instants at which at least one clock ticks. According to the semantics of CCSL (Fig. 3), instants at which no clock ticks have no impact on the satisfaction relation between a schedule and a specification. In other words, if σ is a schedule of a specification, and σ' is the schedule obtained by inserting or removing arbitrary number of instants at which no clock ticks into the schedule σ, then σ' is also a schedule of the specification. For example, if $\sigma = \{c_1\}\{c_2\}\{c_1\}\{c_2\}...$ is a schedule of the specification $SP = \langle \emptyset, \{c_1 \preceq c_2\} \rangle$, then schedule $\sigma' = \{c_1\}\{c_2\}\emptyset\{c_1\}\{c_2\}...$ (obtained by inserting an instant at which no clock ticks into σ) is also a schedule of SP, because at the instant 3, $H_\sigma(3,c_1) \geq H_\sigma(3,c_2)$ holds since no clock ticks. Therefore, in the schedule problem, we always focus on the schedules that contain no instants at which no clock ticks (except for instant 0), i.e., every schedule σ must satisfy: $\sigma(i) \neq \emptyset$ for any $i \in \mathbb{N}^+$.

2.3. Clock-labelled transition system

Clock-labelled Transition System In CCSL, the clock behaviour can be captured as a special type of finite transition systems, called a 'clock-labelled transition system' (cLTS) [20]. A cLTS is a tuple

$$\mathcal{A} = \langle L, T, l_0, \mathcal{C} \rangle$$

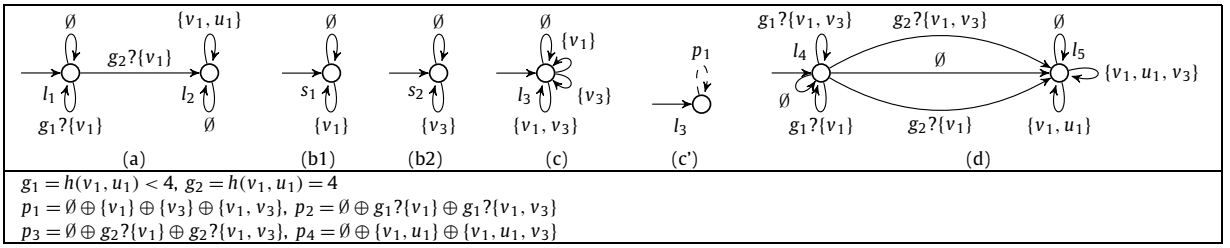


Fig. 4. Examples of cLTSs.

where L is a set of locations, l_0 is an initial location. $T \subseteq L \times (G \times (\mathcal{C} \to \{0,1\})) \times L$ is a set of transitions. A transition $(l, g?\alpha, l') \in T$ (also denoted as $l \xrightarrow{g?\alpha} l'$) can be fired from l to l' when guard g is true and exactly the clocks in $\alpha \subseteq \mathcal{C}$ tick. G is a set of guards. The guard g is of the form

$$g =_{df} h(c_1, c_2) \bowtie k,$$

where $k \in \mathbb{Z}$, $\bowtie \in \{<, \leq, >, \geq, =\}$. When g is the Boolean true expression, it can be omitted and we simply write α. We use $h(c_1, c_2)$ to represent the difference between $h(c_1)$ and $h(c_2)$: $h(c_1, c_2) =_{df} h(c_1) - h(c_2)$, where the unary function h is defined in Sect. 2.1. In cLTSs, T satisfies that for each location $l \in L$, $(l, \emptyset, l) \in T$.

A clock sequence $\sigma = \alpha_1\alpha_2...\alpha_i...$ is accepted by a cLTS if from the initial location l_0 there is a path $l_0 \xrightarrow{g_1?\alpha_1} l_1 \xrightarrow{g_2?\alpha_2} ... \xrightarrow{g_i?\alpha_i} l_i...$ in the cLTS, where each guard g_i $(i \geq 1)$ is evaluated to be true. We use $Seq(\mathcal{A})$ to denote the set of all sequences accepted by $\mathcal{A}$, and we use $Sch(\mathcal{A})$ to denote the set of all schedules accepted by $\mathcal{A}$.

Example 2.3. Fig. 4(a) shows the cLTS of the constraint $u_1 \triangleq v_1 \$ 5$, where $L = \{l_1, l_2\}$, the initial location is l_1, $\mathcal{C} = \{u_1, v_1\}$. There are 5 transitions in T: $(l_1, \emptyset, l_1)$, $(l_1, g_1?\{v_1\}, l_1)$, $(l_1, g_2?\{v_1\}, l_2)$, $(l_2, \{v_1, u_1\}, l_2)$, $(l_2, \emptyset, l_2)$. g_1, g_2 are given in the lower part of the table. Let $\sigma = v_1v_1v_1v_1\{v_1, u_1\}^\omega$, then $\sigma \in Sch(\mathcal{A})$.

Note that p_2, p_3, p_4 will be used later in conjunction with this example.

Synchronous Product The synchronous product of $\mathcal{A}_1$,...,$\mathcal{A}_n$, denoted by $\mathcal{A}_1 \parallel ... \parallel \mathcal{A}_n$, captures the common behaviours of all n cLTSs. They synchronize only when they all agree on whether their common clocks tick or not. Formally, let $\mathcal{A}_i = \langle L_i, T_i, l_{0,i}, \mathcal{C}_i \rangle$ $(1 \leq i \leq n)$, then $\mathcal{A}_1 \parallel ... \parallel \mathcal{A}_n$ is defined as a tuple $\langle L, T, l_0, \mathcal{C} \rangle$ where

(1) $L = L_1 \times ... \times L_n$;
(2) $(\langle l_1, ..., l_n \rangle, (\bigwedge_{i=1}^n g_i)?(\bigcup_{i=1}^n \alpha_i), \langle l'_1, ..., l'_n \rangle) \in T$ iff $\langle l_i, g_i?\alpha_i, l'_i \rangle \in T_i$ for $1 \leq i \leq n$ and $\alpha_j \cap \mathcal{C}_k = \alpha_k \cap \mathcal{C}_j$ for any $1 \leq j \leq k \leq n$;
(3) $l_0 = \langle l_{0,1}, ..., l_{0,n} \rangle$;
(4) $\mathcal{C} = \bigcup_{i=1}^n \mathcal{C}_i$.

The condition '$\alpha_j \cap \mathcal{C}_k = \alpha_k \cap \mathcal{C}_j$' guarantees that all n cLTSs agree on whether their common clocks tick or not in their own transitions. Refer to [20] for more explicit explanations.

In cLTS we use a 'compositional transition' $[l, g_1?\alpha_1 \oplus ... \oplus g_n?\alpha_n, l']$ as a shorthand to express the set of transitions $(l, g_1?\alpha_1, l')$,...,$(l, g_n?\alpha_n, l')$ with the same locations l, l'. Later in Sect. 4, we see that the operator $\oplus$ here is actually the choice operator of the program model of cDL.

Example 2.4. Fig. 4(c) shows the synchronous product $\mathcal{A}_c = \mathcal{A}_{b1} \parallel \mathcal{A}_{b2}$ where $\mathcal{A}_{b1}$ and $\mathcal{A}_{b2}$ (shown in Fig. 4(b1) and (b2) respectively) are the cLTSs of the free clocks v_1 and v_3 respectively. Fig. 4(d) shows the synchronous product $\mathcal{A}_d = \mathcal{A}_a \parallel \mathcal{A}_c = \mathcal{A}_a \parallel \mathcal{A}_{b1} \parallel \mathcal{A}_{b2}$. In $\mathcal{A}_c$, e.g., the transition $(l_3, \{v_1\}, l_3)$ of $\mathcal{A}_c$ is the synchronization of the transition $(s_1, \{v_1\}, s_1)$ of $\mathcal{A}_{b1}$ and the transition $(s_2, \emptyset, s_2)$ of $\mathcal{A}_{b2}$, where $l_3 = \langle s_1, s_2 \rangle$. In $\mathcal{A}_d$, e.g., the transition $(l_4, g_2?\{v_1, v_3\}, l_5)$ is the synchronization of the transition $(l_1, g_2?\{v_1\}, l_2)$ of $\mathcal{A}_a$ and the transition $(l_3, \{v_1, v_3\}, l_3)$ of $\mathcal{A}_c$, where $l_4 = \langle l_1, l_3 \rangle$, $l_5 = \langle l_2, l_3 \rangle$. Let $\mathcal{C}_a$, $\mathcal{C}_c$ be the set of clocks of $\mathcal{A}_a$ and $\mathcal{A}_c$ respectively. Since $\mathcal{C}_a = \{v_1, u_1\}$ and $\mathcal{C}_c = \{v_1, v_3\}$, we have $\{v_1\} \cap \mathcal{C}_c = \{v_1, v_3\} \cap \mathcal{C}_a = \{v_1\}$ holds.

Fig. 4(c') shows the same cLTS as Fig. 4(c), but we use a compositional transition $[l_3, p_1, l_3]$ to express all 4 transitions $\langle l_3, \emptyset, l_3 \rangle$, $\langle l_3, \{v_1\}, l_3 \rangle$, $\langle l_3, \{v_3\}, l_3 \rangle$ and $\langle l_3, \{v_1, v_3\}, l_3 \rangle$. p_1 is shown in the lower part of the table.

cLTS Semantics of CCSL Any CCSL definition $c \triangleq E$ (or any free clock c) can be captured as a cLTS $\mathcal{A}^{c \triangleq E}$ (or $\mathcal{A}^c$). For any schedule σ, there is

$$\sigma \vDash_{ccsl} c \triangleq E \text{ iff } \sigma \in Sch(\mathcal{A}^{c \triangleq E}).$$

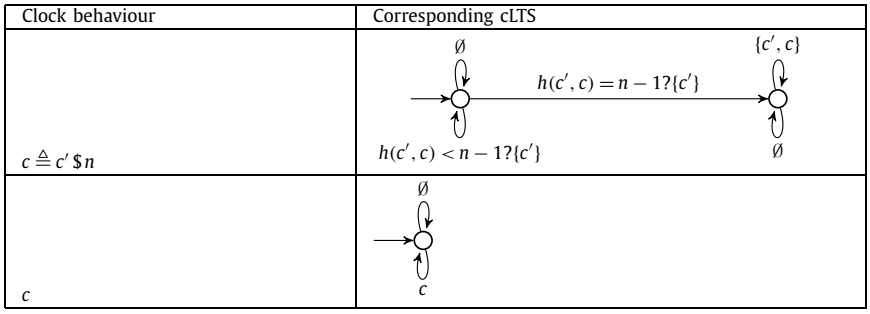


Fig. 5. Encodings of some clock behaviours.

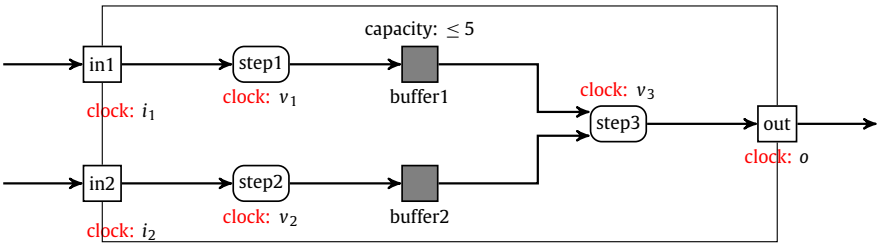


Fig. 6. A component of an application.

Fig. 5 gives the encoding of the clock definition $c \triangleq c' \$ n$ and the general free clock c, which we use in our example in Sect. 3. Fig. 4(a) is a special case of the cLTS of $c \triangleq c' \$ n$ when $n = 5$. The encoding of other clock definitions can be found in [20–22].

With the cLTS semantics of CCSL definitions, the behaviour of clocks of any CCSL specification $SP = \langle \widetilde{Cdf}, \widetilde{Rel} \rangle$ can be encoded as a cLTS $\mathcal{A}^{SP}$ by making the synchronous product of the cLTSs of all clock definitions and all free clocks in SP. Formally, $\mathcal{A}^{SP} = (\|_{c \triangleq E \in \widetilde{Cdf}} \mathcal{A}^{c \triangleq E}) \| (\|_{c \in \mathcal{F}(SP)} \mathcal{A}^{c})$. The following proposition shows that $\mathcal{A}^{SP}$ exactly captures the behaviour of SP.

Proposition 2.1. *Given a specification $SP = \langle \widetilde{Cdf}, \widetilde{Rel} \rangle$, for any schedule σ of $\mathcal{C}(SP)$,*

$$\sigma \vDash_{ccsl} \widetilde{Cdf} \text{ iff } \sigma \in Sch(\mathcal{A}^{SP}). \tag{1}$$

Proposition 2.1 is a direct result from the cLTS theory proposed in [20], where there is

$$\sigma \vDash_{ccsl} SP \text{ iff } \sigma \in Sch(\mathcal{A}'), \tag{2}$$

with $\mathcal{A}' = (\|_{Rel \in \widetilde{Rel}} \mathcal{A}^{Rel}) \| (\|_{c \triangleq E \in \widetilde{Cdf}} \mathcal{A}^{c \triangleq E})$ being the synchronous product of the cLTSs of all clock relations and clock definitions of SP. Here in Proposition 2.1, the proposition (1) is a special case of (2), where we only consider the behaviour of the clocks of SP. For the free clocks $\mathcal{F}(SP)$ which are in $\widetilde{Rel}$ but not in any clock definition of $\widetilde{Cdf}$, we encode them into cLTSs to capture their behaviours.

3. An illustrative example

In this section we consider an illustrative example which is used throughout this paper. This example is originally from [20]. As Fig. 6 shows, a component of a practical application contains two inputs *in*1, *in*2, three computations *step*1, *step*2, *step*3, two buffers *buffer*1, *buffer*2 and an output *out*. *step*1, *step*2 and *step*3 are three independent modules running concurrently. *step*1 (resp. *step*2) needs an input from *in*1 (resp. *in*2) for a computation and after the computation it produces a result in the buffer. *step*3 needs intermediate results from both *step*1 and *step*2 for a computation and after the computation it returns a result to the output *out*. The component continuously receives inputs and produces outputs in a streaming fashion.

In this component, by associating each action with a clock, we can use CCSL, as an annex language of this model, to capture the logical constraints between clocks. As a simple case, let us consider two basic specifications SP_1, SP_2 in the following table:

	$\widetilde{Cdf}$	$\widetilde{Rel}$	$\mathcal{F}(SP_i)$ $(i \in \{1,2\})$
SP_1	$u_1 \triangleq v_1\,\$\,5$	$v_1 \prec v_3,\ v_3 \preceq u_1$	v_1, v_3
SP_2	$u_1 \triangleq v_1\,\$\,5,\ u_2 \triangleq v_1 \vee v_2$	$v_1 \prec v_3, v_3 \preceq u_1, i_1 \preceq v_1,$ $i_2 \preceq v_2, u_2 \prec v_3, v_3 \preceq o$	$i_1, i_2, v_1,$ v_2, v_3, o

where SP_1 specifies a basic relation between $step1$ and $step3$: $step3$ must occur later than $step1$, but before $buffer1$ reaches its maximum capacity: 5 outputs of $step1$. This constraint can be expressed by two clock relations: $v_1 \prec v_3$, $v_3 \preceq u_1$. Here the clock u_1 is newly defined, it ticks as the clock v_1 but delayed by 5 ticks. This can be expressed as a clock definition: $u_1 \triangleq v_1\,\$\,5$. SP_2 defines a more refined specification by adding more clock constraints in the sets $\widetilde{Cdf}$, $\widetilde{Rel}$. SP_2 specifies all dependency relationships between actions in the application. See [20] for more complex specifications of this example.

After obtaining the clock specifications, designers can have a better understanding of this component by performing schedulability analysis of these specifications. One important problem in schedulability analysis is the schedule problem as mentioned in Sect. 1. For a CCSL specification, there may be no schedules, or one or more schedules satisfying it. E.g., Fig. 7 shows two possible schedules satisfying the specification SP_1. By analyzing the schedule problem, unimplementable specifications can be found as early as possible in the development process of an RTES.

In this paper, we take SP_1 as an example. We show how our proposed method can be used to describe and analyze the schedule problem of SP_1.

4. Syntax and semantics of cDL

In this section we propose a logic called 'clock-based dynamic logic' (cDL) in order to suitably characterize the CCSL specifications. cDL extends FODL with clocks as primitives and inherits the concept of normalized trace formulae from dTL^2 to express and verify temporal properties. cDL provides a proof system for reasoning about the schedule problem at a high level. It can be encoded into higher-order logic for implementation issues. We firstly give a brief introduction to FODL and dTL^2 as a background. Then we define the syntax of cDL in Sect. 4.2. At last we give its semantics in Sect. 4.3.

4.1. First-order dynamic logic and dynamic temporal logic dTL^2

First-order dynamic logic. Dynamic logic [23] is an extension of modal logic for reasoning about programs. FODL [14,24] is a type of dynamic logic able to express programs in the domain of first-order arithmetic theory. A program model p in FODL is a regular program, defined as follows:

$$p ::= x := e \mid P? \mid p; p \mid p \oplus p \mid p^*,$$

where e is an arithmetical expression and P is a quantifier-free Boolean expression. $x := e$ is an assignment, it assigns to the variable x the value of an arithmetical expression e. $P?$ is a test, it means at the current state, the program proceeds if the proposition P is true. ;, $\oplus$, $*$ are the sequence, non-deterministic choice and finite loop operator respectively. $p; q$ means the program first executes p, and after p terminates, it executes q. $p \oplus q$ means the program either executes p or executes q, it is a non-deterministic choice. p^* means the program executes p for a finite number of times.

An FODL formula ϕ is defined as follows:

$$\phi ::= tt \mid e \leq e \mid \langle p \rangle \phi \mid \neg\phi \mid \phi \wedge \phi \mid \forall x.\phi,$$

where tt is Boolean true, $\leq$ represents the 'less than' relation in the arithmetic theory. $\langle p \rangle \phi$ is called a 'dynamic formula'. It is formed by embedding a regular program p into a modal logical formula $\Diamond\phi$. $\langle p \rangle \phi$ expresses that 'after some execution of p, ϕ holds', which is a state property of p. One can refer to [24] for more details.

The dynamic temporal logic dTL^2. dTL^2 [15] is an augmentation of differential dynamic temporal logic (DDTL) [25], which enriches FODL with differential equations for expressing the behaviour of hybrid systems and 'dynamic temporal formulae' of the form '$\langle p \rangle \Diamond \phi$' and '$\langle p \rangle \Box \phi$' for expressing the temporal properties of p. Dynamic temporal formulae $\langle p \rangle \mathfrak{d}$ $(\mathfrak{d} \in \{\Diamond\phi, \Box\phi\})$ express that there exists a trace of p satisfying the temporal formula $\mathfrak{d}$. In DDTL, there are no suitable rules to support the derivations of formulae of the form $\langle p \rangle \Box \phi$. Based on DDTL, dTL^2 introduces a type of more general formulae called

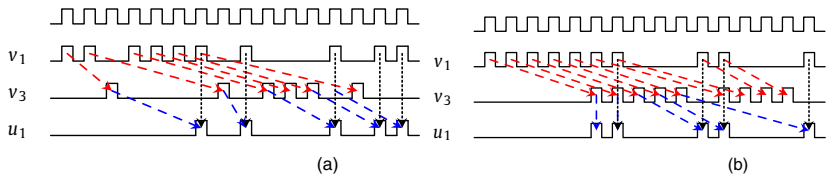


Fig. 7. Two possible schedules for SP_1.

'normalized trace formulae' which are of the forms '$\langle p\rangle\phi \sqcup \Diamond\psi$' and '$\langle p\rangle\phi \sqcap \Box\psi$', and proposes relative rules to support their derivations. Normalized trace formula is able to express both state and temporal properties of a single trace (whose meaning will be given in Sect. 4.2). One can refer to [15] for more details about dTL^2.

4.2. The syntax of cDL

In order to characterize the behaviour model of CCSL clocks (i.e., the cLTS) in logic, we introduce a program model called 'clock program model' (CPM) based on the regular program model of FODL [14,24].

Definition 4.1 *(Syntax of CPM).* The syntax of CPM is defined in BNF as follows:

$$p ::= \alpha \mid g?\alpha \mid \varepsilon \mid \ddagger \mid p; p \mid p \oplus p \mid p^* \mid p^\omega.$$

The intuitive meaning for each sentence is as follows. α is a set of ticked clocks and g is a guard in cLTS (see Sect. 2 for their definitions). We also call α a 'clock event' (or simply 'event') in CPM. Each event consumes one unit of time. The guarded clock event $g?\alpha$ means 'at current time, if g is true, then α executes, else the program halts'. The evaluation of g does not consume any time. ε represents an 'empty program', it does nothing and does not consume time. $\ddagger$ represents a 'halting program', it halts the program and nothing can happen after that. $;, \oplus, *$ are the sequence, non-deterministic choice and finite loop operator that are directly inherited from FODL. $p; q$ means that the program first executes p, and after p terminates, it executes q. $p \oplus q$ means that the program either executes p, or executes q, it is a non-deterministic choice. p^* means that the program executes p for a finite number of times. ω is the infinite loop operator. p^ω means that the program p executes for infinitely many times and never terminates.

Note that we use the same symbol 'ω' to express the infinite loop in CPM (Definition 4.1) and an infinite clock sequence that repeats a segment (Sect. 2) at the same time. The precedence of the operators in CPM is declared from the highest to the lowest as: ω, $*$, $;$, $\oplus$.

As we will see in Sect. 6.1, any cLTS can be encoded as a CPM. The schedules accepted by a cLTS exactly correspond to the words accepted by its corresponding CPM (as stated in Proposition 6.4).

Example 4.1. The behaviour of clocks u_1, v_1, v_3 in SP_1 (given in Sect. 3, whose cLTS corresponds to Fig. 4(d) in Sect. 2.3) can be captured as a CPM $p_{sp1} = p_2^\omega \oplus p_2^*; p_3; p_4^\omega$, where p_2, p_3, p_4 are given in the lower part of the table in Fig. 4.

cDL extends FODL with CPM as its program model and inherits normalized trace formula of the form $\langle p\rangle\phi \sqcap \Box\psi$ from dTL^2 [15]. The following definition gives the syntax of cDL formulae.

Definition 4.2 *(Syntax of cDL formulae).* The cDL formula ϕ is defined in BNF as follows:

$$\phi ::= \phi_s \mid \langle p\rangle\phi \sqcap \Box\phi \mid \neg\phi \mid \phi \wedge \phi$$

where

$$\phi_s ::= tt \mid e \le e \mid \neg\phi_s \mid \phi_s \wedge \phi_s \mid \forall x.\phi_s,$$
$$e ::= x \mid h(c) \mid \eta(c) \mid k \mid e + e \mid e \cdot e.$$

ϕ_s represents static formulae. In ϕ_s, tt is Boolean true, e is an integer arithmetic expression, x is a general variable in the domain $\mathbb{Z}$. We use Var to denote a set of general variables. The function $h : \mathcal{C} \to \mathbb{N}$ (given in Sect. 2.1) records the number of ticks for each clock at the current instant. And the function $\eta : \mathcal{C} \to \{0, 1\}$ (given in Sect. 2.1) records the state of each clock at the current instant. Because clocks do not appear alone in a cDL formula (they only appear alone in the programs of a cDL formula), we can take $h(c), \eta(c)$ as special variables related to the clock $c \in \mathcal{C}$. We use $Var(\mathcal{C})$ to denote the set of all 'clock-related variables' $h(c), \eta(c)$ for any c in $\mathcal{C}$. $k \in \mathbb{Z}$ is a constant. $\langle p\rangle\phi \sqcap \Box\psi$ is a dynamic formula. The term $\phi \sqcap \Box\psi$ describes both a state property and a temporal property of an execution trace in clock programs. It consists of a state formula ϕ and a temporal formula $\Box\psi$, with a conjunction operator $\sqcap$ linking them. The formula $\langle p\rangle\phi \sqcap \Box\psi$ means that there exists some execution of p s.t. (1) the execution trace satisfies the temporal property $\Box\psi$, and (2) after the execution terminates (if it does), the state property ϕ also holds. For non-terminating executions of p, they do not need to satisfy the condition (2).

As it will be seen in Sect. 6.1, the schedule problem for a CCSL specification can be expressed as a cDL formula. The truth of the formula indicates the existence of a schedule of the specification (as stated in Theorem 6.1).

Example 4.2. The schedule problem of SP_1 in Example 4.1 can be captured as a dynamic formula $I_{sp1} \to \langle p_{sp1}\rangle tt \sqcap \Box(\psi_{sp1} \wedge \psi_\emptyset)$, where the program p_{sp1} can never terminate. It means that 'under the initial condition I_{sp1}, there exists an infinite trace of p_{sp1} satisfying $\Box(\psi_{sp1} \wedge \psi_\emptyset)$'. The formula ψ_{sp1} captures the set of clock relations in SP_1 and the formula $\psi_\emptyset$ plays the role of filtering out the traces of p, more details will be given in Sect. 6.

$[\cdot]$ is the dual operator of $\langle\cdot\rangle$. Formulae of the form '$[p]$...' mean that 'all execution traces of p satisfy ...'. The disjunction operator $\sqcup$ is the dual operator of the conjunction operator $\sqcap$. Formula $[p]\phi \sqcup \Diamond\psi$ means that for each trace of p, it either satisfies $\Diamond\psi$, or terminates and satisfies ϕ. It can be expressed as formula $\neg\langle p\rangle\neg\phi \sqcap \Box\neg\psi$. Other logical terms and expressions such as *ff* (the Boolean false), $\phi_1 \vee \phi_2$, $\phi_1 \rightarrow \phi_2$, $\exists x.\phi$, $e_1 - e_2$, $e_1 = e_2$, $e_1 < e_2$,... can be expressed by the terms and expressions defined in Definition 4.2.

In cDL, given a formula ϕ, we say a variable X is 'bound' in ϕ if

(1) $X \in Var$ and X is in the scope of some quantifier $\forall X$, or
(2) $X \in Var(\mathcal{C})$ (assume $X = h(c)$ or $X = \eta(c)$),
  (i) there exists a subformula of ϕ of the form $\langle p\rangle\psi_1 \sqcap \Box\psi_2$ s.t. X is in $\psi_1 \sqcap \Box\psi_2$ and c is in p, or
  (ii) there exists a subprogram of ϕ of the form $q; r$ s.t. X is in r and c is in q.

We say a variable X is 'free' in ϕ if it is not bound in ϕ. A substitution $\phi[e/X]$ replaces every free occurrence of the variable X of ϕ with the expression e. An 'admissible substitution' guarantees that the meaning of a formula is the same before and after the substitution. $\phi[e/X]$ is 'admissible' iff there exists no variable Y s.t. (1) Y is in e; and (2) Y is bound in $\phi[e/X]$. Unless specially mentioned, all substitutions in this paper are assumed to be admissible.

4.3. The semantics of cDL

Kripke Frame & Trace The semantics of cDL is based on the Kripke frame (S, val) [24], where S is a set of states, *val* interprets a program as a set of traces on S and a logical formula as a subset of S. A trace tr is a finite or infinite sequence of states. Given a finite trace $tr_1 = s_0 s_1 ... s_n$ and a (possibly infinite) trace $tr_2 = u_0 u_1 ... u_m ...$, we define $tr_1 \cdot tr_2 =_{df} s_0 s_1 ... s_n u_1 u_2 ... u_m ...$ provided that $s_n = u_0$. Given any tr_1, tr_2, we define

$$tr_1 \circ tr_2 =_{df} \begin{cases} tr_1 \cdot tr_2, & \text{if } tr_1 \text{ is finite} \\ tr_1, & \text{otherwise} \end{cases}.$$

Given two sets of traces S_1, S_2, $S_1 \circ S_2$ is defined as:

$$S_1 \circ S_2 =_{df} \{tr_1 \circ tr_2 \mid tr_1 \circ tr_2 \text{ is defined}, tr_1 \in S_1, tr_2 \in S_2\}.$$

We use $tr(i)$ to denote the i^{th} element of the trace tr, $i \geq 0$. We use tr_b to denote the first element of the trace tr, $tr_b = tr(0)$. We use tr_e to denote the last element of the trace tr, provided that tr is finite.

Definition 4.3 *(State and evaluation in cDL).* Given a set of clocks $\mathcal{C}$ and a set of variables *Var*, a state s in cDL is defined as a total function as follows:

(i) s maps each variable $h(c) \in Var(\mathcal{C})$ to a value in domain $\mathbb{N}$.
(ii) s maps each variable $\eta(c) \in Var(\mathcal{C})$ to a value in domain $\{0, 1\}$.
(iii) s maps each variable $x \in Var$ to a value in domain $\mathbb{Z}$.

Given an expression e and a state s, an evaluation $Eval_s(e)$ is defined as:

(1) If $e = a$, where $a \in \{x, h(c), \eta(c)\}$, then $Eval_s(a) =_{df} s(a)$.
(2) If $e = k$, then $Eval_s(k) =_{df} k$.
(3) If $e = e_1 \triangle e_2$, where $\triangle \in \{+, \cdot\}$, then $Eval_s(e) =_{df} Eval_s(e_1) \triangle Eval_s(e_2)$.

With Definition 4.3 we can link the concept of traces in cDL and the concept of clock sequences in CCSL by defining the traces of a clock sequence as follows.

Definition 4.4 *(Trace of a clock sequence).* Given a set of clocks $\mathcal{C}$, the corresponding set of traces of a clock sequence κ, denoted as Tr^{κ}, is defined s.t. for all clocks $c \in \mathcal{C}$ and $i \in \mathbb{N}^+$, the following conditions hold:

(i) $tr(0)(\eta(c)) = 0$ and $tr(0)(h(c)) = 0$;
(ii) $tr(i)(\eta(c)) = 1$ iff $c \in \kappa(i)$;
(iii) $tr(i)(h(c)) = H_{\kappa}(i, c)$.

Definition 4.4 indicates that there exists a connection between the clock sequences in CCSL and the traces in cDL. Intuitively, a clock sequence corresponds to a set of traces whose clock-related variables (of the form '$h(c)$, $\eta(c)$') exactly record the information reflected by the sequence at each instant. From Definition 4.4 we note that only traces with all variables $\eta(c)$, $h(c)$ (for any $c \in \mathcal{C}$) being set to 0 at the beginning can capture the behaviour of clock sequences since we

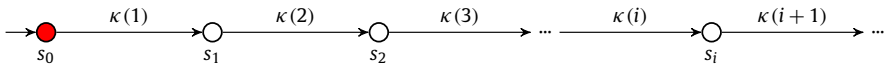


Fig. 8. The correspondence relation between a sequence κ and one of the traces $tr \in Tr^{\kappa}$.

assume $H_\kappa(0, c) = 0$ for any κ (Sect. 2.1). Thus we only focus on those traces that satisfy this condition. We call them 'standard traces'.

Definition 4.5 *(Standard traces).* Given a set of clocks $\mathcal{C}$, a standard trace tr (w.r.t. $\mathcal{C}$) is defined s.t. $tr(0)(\eta(c)) = 0$ and $tr(0)(h(c)) = 0$ for any $c \in \mathcal{C}$.

Example 4.3. Fig. 8 shows a correspondence relation between a clock sequence κ and a trace $tr \in Tr^{\kappa}$. Let $\kappa(1) = \{c_1\}$, $\kappa(2) = \{c_1, c_2\}$, then s_0, s_1, s_2 satisfy that $s_0(\eta(c_1)) = s_0(\eta(c_2)) = s_0(h(c_1)) = s_0(h(c_2)) = 0$, $s_1(\eta(c_1)) = 1$, $s_1(h(c_1)) = H_\kappa(1, c_1) = 1$, $s_1(\eta(c_2)) = 0$, $s_1(h(c_2)) = H_\kappa(1, c_2) = 0$, $s_2(\eta(c_1)) = 1$, $s_2(h(c_1)) = H_\kappa(2, c_1) = 2$, $s_2(\eta(c_2)) = 1$ and $s_2(h(c_2)) = H_\kappa(2, c_2) = 1$.

The semantics of cDL is given as the following definition.

Definition 4.6 *(Semantics of cDL formulae).*
Given a set of clocks $\mathcal{C}$ and a set of variables Var, the semantics of cDL formulae is given as a Kripke frame (S, val), where S is the set of all states defined in Definition 4.3, val is defined as follows:

- For CPM:
  (1) $val(\varepsilon) =_{df} S$.
  (2) $val(\ddagger) =_{df} \emptyset$.
  (3) $val(\alpha) =_{df} \{ss' \mid s, s' \in S;$ for any $c \in \alpha, s'(h(c)) = s(h(c)) + 1 \wedge s'(\eta(c)) = 1$; for any $d \in \mathcal{C} - \alpha, s'(h(d)) = s(h(d)) \wedge s'(\eta(d)) = 0$; for any $x \in Var, s'(x) = s(x)\}$.
  (4) $val(g?\alpha) =_{df} \{ss' \mid s \in val(g), ss' \in val(\alpha)\}$.
  (5) $val(p; q) =_{df} val(p) \circ val(q)$.
  (6) $val(p \oplus q) =_{df} val(p) \cup val(q)$.
  (7) $val(p^*) =_{df} val(\varepsilon) \cup \bigcup_{n \geq 1} \underbrace{val(p) \circ ... \circ val(p)}_{n}$.
  (8) $val(p^{\omega}) =_{df} \underbrace{val(p) \circ val(p) \circ ....}_{\infty}$
- For cDL formula:
  (i) $val(tt) =_{df} S$.
  (ii) $val(e_1 \leq e_2) =_{df} \{s \mid Eval_s(e_1) \leq Eval_s(e_2)\}$.
  (iii) $val(\langle p \rangle \phi \sqcap \Box \psi) =_{df} \left\{ s \mid \text{there is a } tr \in val(p) \text{ s.t. } s = tr_b,\ tr \vDash \Box\psi \text{ and } tr_e \in val(\phi) \text{ if } tr_e \text{ exists} \right\}$.
  (iv) $val([p]\phi \sqcup \Diamond \psi) =_{df} \left\{ s \mid \text{for all } tr \in val(p) \text{ s.t. } s = tr_b,\ tr \vDash \Diamond\psi \text{ or } tr_e \text{ exists and } tr_e \in val(\phi) \right\}$.
  (v) $val(\neg\phi) =_{df} \{s \mid s \notin val(\phi)\}$.
  (vi) $val(\phi \wedge \psi) =_{df} val(\phi) \cap val(\psi)$.
  (vii) $val(\forall x.\phi) =_{df} \{s \mid$ for any $v_0 \in \mathbb{Z}, s \in val(\phi[v_0/x])\}$.
  The trace semantics of temporal formulae $\Box\psi$, $\Diamond\psi$ is defined as follows:
  (a) $tr \vDash \Box\psi$ iff every state s in tr $(s \neq tr_b)$ satisfies $s \in val(\psi)$.
  (b) $tr \vDash \Diamond\psi$ iff there exists a state s in tr $(s \neq tr_b)$ that satisfies $s \in val(\psi)$.

The semantics of each CPM corresponds to a set of traces on S. ε defines the set of all traces with length 1. $\ddagger$ defines the empty set. The event α defines a transition from a state s to a state s'. Intuitively, at the current instant, if a clock c $(c \in \alpha)$ ticks, variable $h(c)$ is increased by 1 and variable $\eta(c)$ is set to 1; if the clock c does not tick $(c \notin \alpha)$, $h(c)$ does not change and $\eta(c)$ is set to 0. Other variables in both s and s' are kept the same. Traces satisfying $g?\alpha$ are exactly those traces satisfying α whose beginning states satisfy g. Since the guard g is in fact a cDL formula, the definition of $val(g)$ makes sense. Each trace of $p; q$ is formed by concatenating a trace of p and a trace of q. Each trace of $p \oplus q$ is either a trace of p or a trace of q. The traces of program p^* are defined as all finite traces with length 1, or traces of the form $tr_1 \circ tr_2 \circ ... \circ tr_n$ where $n \geq 1$, $tr_i \in val(p)$ is finite $(1 \leq i < n)$. The traces of p^{ω} consist of all infinite traces of the form $tr_1 \circ tr_2 ...$ where each $tr_i \in val(p)$ is finite $(i \geq 1)$, or of the form $tr_1 \circ tr_2 \circ ... \circ tr_n$ where $n \geq 1$, $tr_1, ..., tr_{n-1} \in val(p)$ are finite, but $tr_n \in val(p)$ is infinite.

The semantics of each cDL formula corresponds to a set of states in S. (iii), (iv) are similar to the corresponding definitions in dTL^2 [15]. Note that in cDL, for any temporal formulae $\Box\psi$ and $\Diamond\psi$, ψ is a state formula. The semantics of

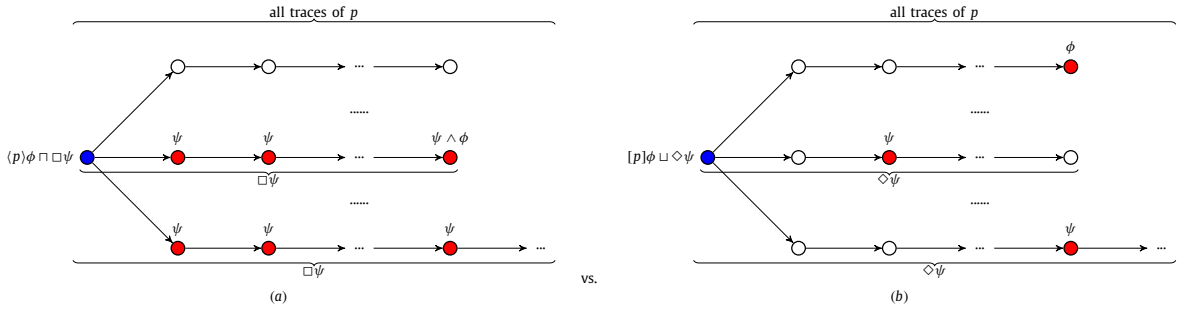


Fig. 9. The semantics of $\langle p\rangle\phi \sqcap \Box\psi$ and $[p]\phi \sqcup \Diamond\psi$.

temporal formulae $\Box\psi$ and $\Diamond\psi$ is given in (a) and (b), where we do not require that the first state of a trace tr satisfies formula ψ. This stipulation helps traces better correspond to schedules because according to Definition 4.4 the 1st element of a schedule exactly corresponds to the 2nd element of a trace. For a state property ϕ, we only consider its truth for terminating traces. (v), (vi), (vii) directly come from the corresponding definitions in FODL [24].

Example 4.4. Fig. 9 gives a graphical illustration of the semantics of formulae $\langle p\rangle\phi \sqcap \Box\psi$, $[p]\phi \sqcup \Diamond\psi$. In Fig. 9(a), a state satisfies formula $\langle p\rangle\phi \sqcap \Box\psi$ (the blue state) iff there exists a trace of p (the 2nd and 3rd trace from top) satisfying the temporal formula $\Box\psi$ and if it terminates, the terminating state satisfies ϕ. In Fig. 9(b), a state satisfies formula $[p]\phi \sqcup \Diamond\psi$ (the blue state) iff any trace of p either satisfies the temporal formula $\Diamond\psi$ (the 2nd and 3rd trace from top), or it terminates and satisfies ϕ (the 1st trace from top).

From Fig. 9 we see that the infinite traces are only required to satisfy the temporal formula because they never terminate. With the semantics of cDL we introduce the satisfaction relation of cDL.

Definition 4.7 *(Satisfaction relation).* Given a Kripke frame (S, val), for any state $s \in S$ and a cDL formula ϕ, the satisfaction relation $s \models_{cdl} \phi$ is defined as:

$$s \models_{cdl} \phi \text{ iff } s \in val(\phi).$$

If for all state $s \in S$, $s \models_{cdl} \phi$, then we call ϕ is 'valid', denoted by $\models_{cdl} \phi$.

CPM is in fact an ω-regular expression of an ω-regular language [26] based on clock events and guarded clock events as words. This observation is important for the encoding from cLTS into CPM introduced in Sect. 6.1. We first define the concept of 'string', as the basic element of the ω-regular language. Then we show how CPM denotes the ω-regular language by defining a semantics of CPM based on strings.

Definition 4.8 *(String).* A string is a finite or infinite sequence $\rho = a_1a_2...a_n...$ where a_i $(1 \le i \le n)$ is a clock event of the form α or $g?\alpha$, called a 'word'. The concatenation between strings is defined as follows:

$$\rho_1\rho_2 =_{df} \begin{cases} \rho_1, & \text{if } \rho_1 \text{ is infinite} \\ \rho_1 \centerdot \rho_2, & \text{if } \rho_1 \text{ is finite} \end{cases},$$

where for any finite string $x_1 = a_1...a_n$ and string $x_2 = b_1...b_m...$, $x_1 \centerdot x_2 =_{df} a_1...a_nb_1...b_m...$.

We use ρ^ω to represent an infinite string that infinitely repeats string ρ, i.e., $\rho^\omega =_{df} \underbrace{\rho\rho...}_{\infty}$.

Let λ be the empty string that satisfies for any ρ, it holds that $\rho\lambda = \lambda\rho = \rho$.

We define 4 operators on sets of strings:

(1) Concatenation: $L_1L_2 =_{df} \{\rho_1\rho_2 \mid \rho_1 \in L_1, \rho_2 \in L_2\}$.
(2) Union: $L_1 \cup L_2 =_{df} \{\rho \mid \rho \in L_1 \text{ or } \rho \in L_2\}$.
(3) Star Operator: $L^* =_{df} \cup_{n\ge 0}L^n$, where $L^n =_{df} \underbrace{LL...L}_{n}$.
(4) Omega Operator: $L^\omega =_{df} \underbrace{LL...L...}_{\infty}$.

From the definition of string in Definition 4.8, we can observe that clock sequence is in fact a special type of strings where there is no guard in each clock event.

Definition 4.9 *(CPM as an ω-regular expression).*

CPM is an ω-regular expression, it denotes an ω-regular language in the sense of the semantics given as follows. For any CPM p, the set of strings of p, denoted by $Str(p)$ is defined as:

(i) $Str(a) =_{df} \{a\}$, $Str(\varepsilon) =_{df} \{\lambda\}$, $Str(\ddagger) =_{df} \emptyset$, where a is of the form α or $g?\alpha$.
(ii) $Str(q; r) =_{df} Str(q)Str(r)$.
(iii) $Str(q \oplus r) =_{df} Str(q) \cup Str(r)$.
(iv) $Str(q^*) =_{df} Str(q)^*$.
(v) $Str(q^\omega) =_{df} Str(q)^\omega$.

We use Str to denote the language denoted by CPM in the sense of the semantics given above.

We use symbol $\equiv$ to denote the equivalence between CPMs on strings, i.e., for any CPMs p, q, $p \equiv q$ iff $Str(p) = Str(q)$.

Intuitively, a string captures one deterministic behaviour of CPM and it can be seen as a sequential program of CPM. We show this by defining the semantics of strings in Kripke frame as the following definition.

Definition 4.10 *(Semantics of strings).* Given a set of clocks $\mathcal{C}$ and a set of variables Var, the semantics of a string $\rho = a_1 a_2 ... a_n ...$ is given as a Kripke frame (S, val), where S is the set of all states defined in Definition 4.3, $val(\rho)$ is defined as:

$$val(\rho) =_{df} val(a_1) \circ val(a_2) \circ ... \circ val(a_n) \circ ....$$

For a set of strings A, we define

$$val(A) =_{df} \bigcup_{\rho \in A} val(\rho).$$

Since a string captures one behaviour of CPM, the set of strings of a CPM should capture all of its behaviours. Therefore, there exists a correspondence relation between the string semantics and the trace semantics of CPM, stated as the following proposition.

Proposition 4.1 *(Relation between string semantics and trace semantics). Given a CPM p, we have*

$$val(p) = val(Str(p)).$$

Proof. The proposition can be proved by induction on the structure of p. The base case is trivial, for example, we have $val(Str(a)) = val(\{a\}) = val(a)$ where a is of the form α or $g?\alpha$.

For the induction step, we only give $q \oplus r$ for example, other cases are similar. By Definition 4.9, we have $val(Str(q \oplus r)) = val(Str(q) \cup Str(r)) = val(Str(q)) \cup val(Str(r))$. By induction hypothesis, we have $val(Str(q)) = val(q)$ and $val(Str(r)) = val(r)$. Therefore we have $val(Str(q \oplus r)) = val(q) \cup val(r) = val(q \oplus r)$. □

Proposition 4.1 says that the traces of a CPM are actually the traces of all strings of a CPM, which means that the set of strings of a CPM actually captures all of its behaviours.

With Definition 4.10, we can have a better understanding of the relation between clock sequence κ and traces of CPM.

Proposition 4.2. *Given a clock sequence κ, Tr^κ (defined in Definition 4.4) is exactly the set of all standard traces in $val(\kappa)$.*

Proposition 4.2 is direct according to Definitions 4.4, 4.5 and 4.10. Here we omit its proof. Proposition 4.2 will play a crucial rule in the proof of Proposition 6.4 in Sect. 6.1.

5. Proof calculus of cDL

In this section, we propose a proof system for cDL, which is the logical framework for analyzing the schedule problem of CCSL specifications. We first briefly introduce the background about sequent and sequent calculus in Sect. 5.1. Then we introduce the proof rules in Sect. 5.2, and discuss the soundness, completeness and decidability of cDL in Sect. 5.3.

5.1. Sequent calculus

Sequent Calculus & Rule In this paper we use Gentzen's sequent [27] as the logical argumentation for the proof calculus of cDL. A sequent has the form:

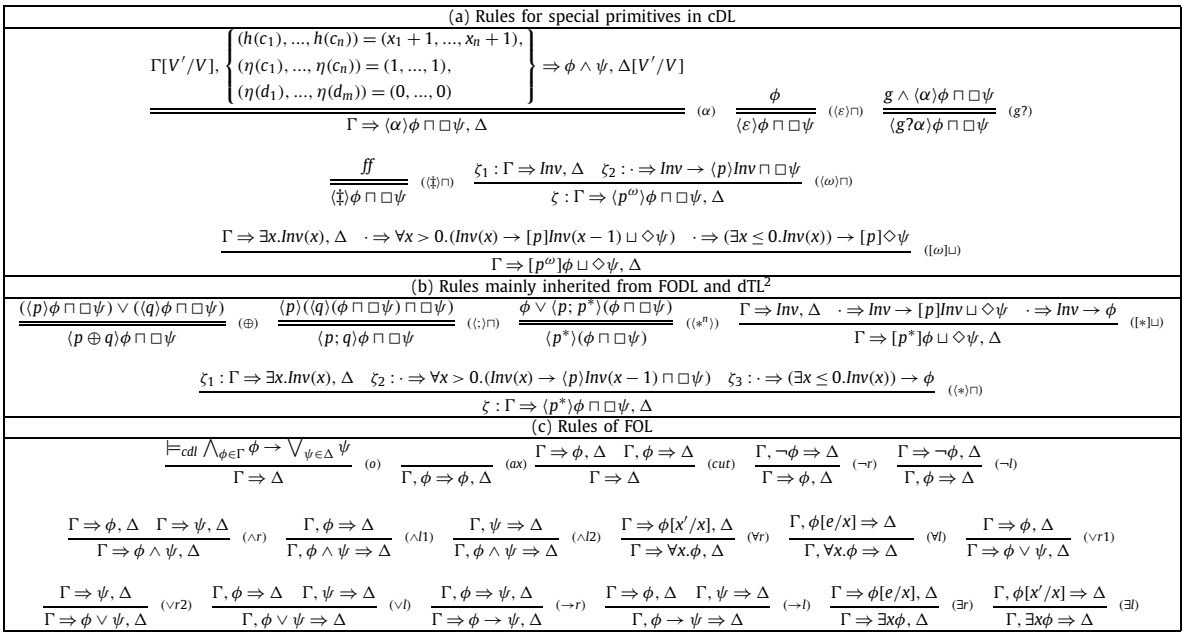


Fig. 10. Proof Calculus of cDL.

$$\Gamma \Rightarrow \Delta =_{df} \bigwedge_{\phi\in\Gamma} \phi \rightarrow \bigvee_{\psi\in\Delta} \psi,$$

where Γ, Δ are two finite multi-sets of logical formulae. A sequent $\Gamma \Rightarrow \Delta$ means that 'if every formula holds in Γ, one can conclude that some formula holds in Δ'. When Γ or Δ is empty, we use $\cdot$ to denote it.

A rule in sequent calculus is of the form:

$$\frac{\Gamma_1 \Rightarrow \Delta_1 \quad ... \quad \Gamma_n \Rightarrow \Delta_n}{\Gamma \Rightarrow \Delta}$$

It means that if $\Gamma_1 \Rightarrow \Delta_1$, ..., $\Gamma_n \Rightarrow \Delta_n$ are all valid, so is $\Gamma \Rightarrow \Delta$. Each sequent $\Gamma_i \Rightarrow \Delta_i$ in the upper part is called a 'premise', while the sequent $\Gamma \Rightarrow \Delta$ in the lower part is called 'conclusion'. We use

$$\frac{\Gamma' \Rightarrow \Delta'}{\Gamma \Rightarrow \Delta}$$

to represent a pair of sequent rules: $\frac{\Gamma' \Rightarrow \Delta'}{\Gamma \Rightarrow \Delta}$ and $\frac{\Gamma \Rightarrow \Delta}{\Gamma' \Rightarrow \Delta'}$, i.e., $\Gamma \Rightarrow \Delta$ is valid iff $\Gamma' \Rightarrow \Delta'$ is valid. Sometimes we write $\frac{\psi}{\phi}$ if for all Γ, Δ, $\frac{\Gamma \Rightarrow \psi, \Delta}{\Gamma \Rightarrow \phi, \Delta}$ holds. It is easy to prove that $\frac{\psi}{\phi}$ just means 'ψ implies ϕ'. We call Γ, Δ the 'context' of the formula ϕ in sequent $\Gamma \Rightarrow \phi, \Delta$ or $\Gamma, \phi \Rightarrow \Delta$.

Node & Proof Tree The derivation of a sequent forms a 'proof tree'. Each sequent in the proof tree is a node, denoted by $\zeta = \langle \nu, \tau \rangle$, where ζ is the node name, ν is a vector of its child nodes, τ is a rule name. In a proof tree,

$$\text{a node } \zeta = \langle ((\zeta_1, ..., \zeta_n), \text{'}(r)\text{'} \rangle \text{ is defined iff there is a derivation } \frac{\zeta_1 : \Gamma_1 \Rightarrow \Delta_1 \quad ... \quad \zeta_n : \Gamma_n \Rightarrow \Delta_n}{\zeta : \Gamma \Rightarrow \Delta}\ (r),$$

where (r) is the name of the rule, $\zeta_1, ..., \zeta_n$ are the child nodes of ζ in sequence from left to right. In a proof tree, we call a node ζ a 'successor' of a node ζ' iff there exist n nodes $(n > 0)$ $\zeta_1, ..., \zeta_n$ s.t. ζ is a child node of ζ_1, ζ_1 is a child node of ζ_2, ..., ζ_n is a child node of ζ'. We call node $\langle \nu, \tau \rangle$ a 'leaf node' if $\nu = \emptyset$. If a leaf node is obtained from a termination rule (rule (o), (ax) introduced below in Fig. 10(c)), we also call it a 'valid node', denoted as $\surd$. We call a proof tree 'a valid proof tree' if all its leaf nodes are valid.

Single-target Sequent/Proof Tree Since the proof system we give in this paper is mainly for decomposing one dynamic formula for a CCSL schedule problem, we restrict ourselves to consider a special type of sequents where there is at most one dynamic formula. We call them 'single-target sequents'. We call a proof tree in which all nodes are single-target sequents a 'single-target proof tree'. Considering this type of sequents and proof trees is enough and does not reduce the ability of sequents for proving all cDL formulae due to the following reasons:

(i) If there exists more than one dynamic formula in a cDL formula ϕ to be proved, since dynamic formulae are atomic formulae in cDL, we can always translate it into a conjunctive/disjunctive normal form like $\phi = \phi_1 \wedge ... \wedge \phi_n$ or $\phi = \phi_1 \vee ... \vee \phi_n$, where each ϕ_i contains only one dynamic formula. Then we can prove ϕ_i one by one through n single-target sequents: $\cdot \Rightarrow \phi_1$, ..., $\cdot \Rightarrow \phi_n$.
(ii) As we will see in Sect. 5.2, every proof tree which starts at a single-target sequent can only have one dynamic formula at all of its nodes because no extra dynamic formulae can be generated by any rules in the proof system of cDL (Fig. 10).

Unless specially mentioned, all sequents and proof trees discussed in this paper are single-targeted.

5.2. Proof rules

Our main contribution in the proof system of cDL is proposing the rules for special primitives: α, $g?\alpha$, ε, p^ω in cDL. They are listed in Fig. 10(a). Other rules in cDL are either directly inherited or can be derived from the proof system of FODL [24], dTL^2 [15] and FOL, which are listed in Fig. 10(b) and (c).

In Fig. 10(a), rule (α) says that under any context Γ, Δ, proving that some trace of α satisfies $\phi \sqcap \Box\psi$ is equivalent to proving that $\phi \wedge \psi$ holds after the execution of α. Variables in V are updated with new values according to α, while their old values are stored in V'. $\alpha = \{c_1, ..., c_n\}$, $\mathcal{C} - \alpha = \{d_1, ..., d_m\}$. $V = (h(c_1), ..., h(c_n), \eta(c_1), ..., \eta(c_n), \eta(d_1), ..., \eta(d_m))$ is a set of variables whose values change as the execution of α. $V' = (x_1, ..., x_n, y_1, ..., y_n, z_1, ..., z_m)$ is a set of new variables (w.r.t. Γ, $\langle\alpha\rangle\phi \sqcap \Box\psi$, Δ) corresponding to V. $\Gamma[V'/V]$ represents the context obtained by doing the substitution $\phi[V'/V]$ for each formula ϕ in Γ, where $\phi[V'/V]$ is the shorthand of $\phi[x_1/h(c_1)]...[x_n/h(c_n)][y_1/\eta(c_1)]...[y_n/\eta(c_n)][z_1/\eta(d_1)]...[z_m/\eta(d_m)]$ replacing variables $h(c_1)...h(c_n)$, $\eta(c_1)...\eta(c_n)$, $\eta(d_1)...\eta(d_m)$ with variables $x_1, ..., x_n$, $y_1, ..., y_n$, $z_1, ..., z_m$ respectively. The vector equation $(x_1, ..., x_n) = (e_1, ..., e_n)$ is a shorthand for equations $x_1 = e_1, ..., x_n = e_n$.

Example 5.1. Consider a sequent $h(c_1) = 0, \eta(c_1) = 0, h(c_2) = 0, \eta(c_2) = 0 \Rightarrow \langle c_1\rangle\Box h(c_1) \geq h(c_2)$, by applying rule ($\alpha$), we obtain the derivation:

$$\frac{x_1 = 0, y_1 = 0, \quad \left\{\begin{array}{c} h(c_1) = x_1 + 1, \eta(c_1) = 1, \\ \eta(c_2) = 0 \end{array}\right\} \Rightarrow h(c_1) \geq h(c_2)}{h(c_2) = 0, z_1 = 0,} \quad {}^{(\alpha)}$$
$$h(c_1) = 0, \eta(c_1) = 0, h(c_2) = 0, \eta(c_2) = 0 \Rightarrow \langle c_1\rangle\Box h(c_1) \geq h(c_2)$$

where x_1, y_1, z_1 are the corresponding new variables of $h(c_1), \eta(c_1), \eta(c_2)$ respectively. They keep the old values of the variables $h(c_1)$, $\eta(c_1)$, $\eta(c_2)$ respectively, while the variables $h(c_1)$, $\eta(c_1)$, $h(c_2)$, $\eta(c_2)$ keep the current values in the premise after the execution of the program c_1. In the derivation above, $h(c_2)$ is kept unchanged.

Rule ($\langle\varepsilon\rangle\sqcap$) holds because we stipulate that the first element of any trace is unrelated to the temporal formula $\Box\psi$ in the definition of $tr \vDash \Box\psi$ (Definition 4.6). Rule ($g?$) moves the guard g out of the dynamic part '$g?\alpha$' as a static formula g. Rule ($\langle\ddagger\rangle\sqcap$) says formula $\langle\ddagger\rangle\phi \sqcap \Box\psi$ is a contradiction, because the program $\ddagger$ will halt and never produce any execution trace. In rules ($\langle\omega\rangle\sqcap$) and ($[\omega]\sqcup$), the state property ϕ is irrelevant since an infinite loop program never terminates. Rule ($\langle\omega\rangle\sqcap$) says that the conclusion holds if we can find an invariant Inv s.t.: (1) Inv holds under the current context Γ, Δ; (2) under any context, if Inv holds, then there exists a trace of p satisfying $\Box\psi$ and after p terminates, Inv holds. Rule ($[\omega]\sqcup$) is similar to ($\langle\omega\rangle\sqcap$), where x indicates the number of repetitions of p before every trace of p satisfying $\Diamond\psi$.

Example 5.2. Fig. 11 gives a graphical illustration of rules ($\langle\omega\rangle\sqcap$), ($[\omega]\sqcup$). The snake arrow indicates a trace. Fig. 11(a) shows that to prove a state (the blue one) satisfies formula $\langle p^\omega\rangle\phi \sqcap \Box\psi$, we firstly show that this state satisfies the invariant Inv, then show that for any state $(s_1, s_2, ..., s_i, ...)$, if it satisfies Inv, then there exists a trace (the red one) satisfying $\Box\psi$ and if it terminates, the terminating state satisfies Inv. If the trace does not terminate, then we have already obtained an infinite trace that satisfies $\Box\psi$. Intuitively, the invariant Inv makes sure that we can always find a segment of trace of p that satisfies $\Box\psi$, by 'concatenating' all these segments, we obtain a trace of p^ω.

Fig. 11(b) shows that to prove the blue state satisfies formula $[p^\omega]\phi \sqcup \Diamond\psi$, firstly we show this state satisfies $\exists x.Inv(x)$, then show that for any state $(s_0,...,s_i, s_{i+1}, ..., s_j, s_{j+1},...)$, if it satisfies $Inv(k)$ for some $k > 0$, then for all traces starting from this state, they either satisfy $\Diamond\psi$ (the red traces), or terminate and satisfy $Inv(k-1)$. The decrease of number k guarantees that there always exists a state (s_j) from which all traces satisfy $\Diamond\psi$.

In Fig. 10(b), rule ($\oplus$) expresses that some trace of $p \oplus q$ satisfies $\phi \sqcap \Box\psi$ iff some trace of p or some trace of q satisfies $\phi \sqcap \Box\psi$. Rule ($\langle;\rangle\sqcap$) means that some trace of $p;q$ satisfies $\phi \sqcap \Box\psi$ iff some trace of p satisfies $\Box\psi$, and if it terminates, there is some following trace of q satisfies $\phi \sqcap \Box\psi$. Rule ($\langle*^n\rangle$) unwinds the loop program into a sequential one. It is based on the equation $p^* \equiv \varepsilon \oplus (p; p^*)$, which means that the traces formed by executing p for $n \geq 0$ times ($val(p^*)$) comprise of the traces formed by executing p for 0 time ($val(\varepsilon)$) and the traces formed by executing p for $n > 0$ times ($val(p; p^*)$). Rules ($[*]\sqcup$) and ($\langle*\rangle\sqcap$) proceed the proof by eliminating the loop operator $*$, they are similar to rules ($\langle\omega\rangle\sqcap$), ($[\omega]\sqcup$) respectively. But since all traces of p^* are finite and will terminate, in rules ($[*]\sqcup$), ($\langle*\rangle\sqcap$), the terminating conditions ('$\cdot \Rightarrow Inv \rightarrow \phi$' and

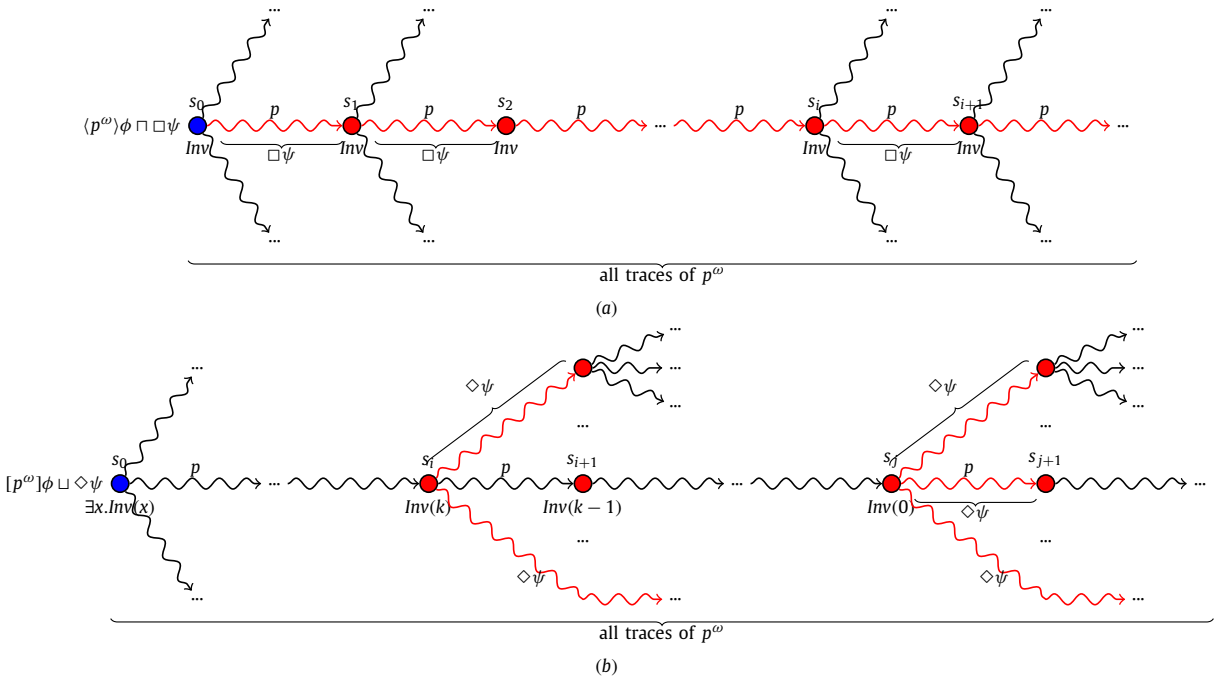


Fig. 11. An illustration of Rule $(\langle\omega\rangle\sqcap)$, $([\omega]\sqcup)$.

'$\cdot \Rightarrow (\exists x \leq 0.Inv(x)) \rightarrow \phi$') for ϕ are considered. Rule $([*]\sqcup)$ says that to prove all traces of p^* either satisfy $\Diamond\psi$ or terminate and satisfy ϕ, we introduce an invariant Inv and prove (1) Inv holds under the current contexts Γ, Δ; (2) under any context, if Inv holds, then all traces of p either satisfy $\Diamond\psi$, or terminate and satisfy Inv; (3) under any context, if Inv holds, then ϕ holds. The explanation of rule $(\langle*\rangle\sqcap)$ is similar to $([*]\sqcup)$, where the decrease of the number x in the invariant makes sure that a finite trace of p^* can be found whose terminating state satisfies ϕ.

In Fig. 10(c), rule (o) is an oracle rule indicating the termination of the proof, where all formulae in Γ, Δ must be QF-FOL formulae. Rule (o) means that to prove the validity of the conclusion, we check the validity of the QF-FOL logical formula in the premise. As indicated in Sect. 1, this process can be handled through an SMT-solving procedure, which is independent from the proof calculus of cDL. (ax) is another termination rule. Other FOL rules are classic and here we omit the details of them. For convenience in the derivation of Sect. 6.4, we also list the FOL rules for connectors $\vee$, $\rightarrow$ and quantifier $\exists$. However, they are not necessary in the proof system and can be derived from other FOL rules.

With the proof calculus of cDL we introduce the derivation of cDL.

Definition 5.1 *(Derivation of cDL).* For any cDL formula ϕ and a multi-set Φ of cDL formulae, we say ϕ is derivable from Φ, denoted by $\Phi \vdash_{cdl} \phi$, iff the sequent $\Phi \Rightarrow \phi$ can be derived to form a valid proof tree according to the rules in Fig. 10. If $\Phi = \emptyset$, we also write $\vdash_{cdl} \phi$.

5.3. Soundness, completeness and decidability of cDL

cDL is sound, as stated in the following theorem.

Theorem 5.1 *(Soundness of cDL). Given a Kripke framework (S, val), for any cDL formula ϕ,*

$$\text{if } \vdash_{cdl} \phi, \text{ then } \models_{cdl} \phi.$$

To prove Theorem 5.1 equals to prove the soundness of each rule in Fig. 10. The soundness of the rules in Fig. 10(b), (c) is directly from the proof calculus of FODL [24] and dTL^2 [15]. The soundness of the rules in Fig. 10(a) can be proved directly according to the semantics of cDL. The only non-trivial cases are rule (α), rule $(\langle\omega\rangle\sqcap)$ and rule $([\omega]\sqcup)$, whose proofs are given in Appendix B.

Generally, like FODL, cDL is not complete due to Gödel's incompleteness theorem [28]. A sub-logic of cDL whose formulae are defined by using all terms and operators in cDL but the operator ω is relatively complete to arithmetic FOL due to the relative completeness of dTL^2 [15]. However, it still remains open whether cDL is relatively complete to arithmetic FOL. The main reason is that for rules $(\langle\omega\rangle\sqcap)$ and $([\omega]\sqcup)$, it is still not clear that for each CPM p, whether there exists an invariant

Inv s.t. falsehood of the premise implies falsehood of the conclusion, which is the key for proving the relative completeness of cDL.

FODL and dTL^2 are generally undecidable, because the process of generating the loop invariant for an arbitrary program model whose domain includes Presburger arithmetic theory is generally undecidable [29]. However, we observe that CPM only contains very simple arithmetic expressions (clock event α) and conditions (clock guard g). It is still not clear for us whether the invariant in cDL is generally decidable or not.

6. Schedulability analysis of CCSL specifications in cDL

In this section we discuss how to analyze the schedule problem of CCSL specifications in the cDL calculus built in previous sections. We first need to encode the schedule problem as a cDL formula, then prove the formula and analyze the proof tree generated through this verification process. As indicated in Fig. 1, the encoding of the schedule problem of a given CCSL specification $SP = \langle \widetilde{Cdf}, \widetilde{Rel} \rangle$ can be accomplished in two steps:

1. Encoding the CCSL specification as ingredients of cDL, which includes two steps:
   (i) Modelling the dynamic behaviour of all clocks $\mathcal{C}(SP)$ as a CPM p_{sp}. This can be done by encoding the synchronous product of the cLTSs of all clock definitions in $\widetilde{Cdf}$ and the cLTSs of all free clocks in $\mathcal{F}(SP)$;
   (ii) Encoding all static clock relations in $\widetilde{Rel}$ as a temporal formula $\Box\psi_{sp}$.
2. Encoding the schedule problem into a cDL formula.

Sect. 6.1, Sect. 6.2 deal with the encoding in step 1.(i) and step 1.(ii) respectively. Sect. 6.3 deals with step 2. In Sect. 6.4, we solve the schedule problem by analyzing its corresponding cDL formula.

6.1. Encoding the behaviour of clocks into cDL

The encoding from cLTS into CPM turns out to be a standard process of encoding Büchi automata into ω-regular languages [26]. From Definition 4.9 in Sect. 4 we see that a CPM is an ω-regular expression that denotes an ω-regular language of *Str*. cLTS can be taken as a special type of Büchi automata that accept clock events or guarded clock events as words and where all states are accepting states. The language accepted by this type of Büchi automata is exactly a subset of the language *Str*. Next we first show how cLTSs can be taken as Büchi automata that accept the language *Str*, then based on this we propose an algorithm (Algorithm 1) for encoding cLTS into CPM.

cLTSs taken as Büchi automata accept a subset of the language *Str*, stated as the following proposition.

Proposition 6.1 *(cLTSs as Büchi automata). Given a cLTS $\mathcal{A} = \langle L, T, l_0, \mathcal{C} \rangle$, we can take it as a Büchi automaton whose locations (i.e. L) are all accepting locations. And for each transition $(l, g?\alpha, l') \in T$ in the Büchi automaton $\mathcal{A}$, $g?\alpha$ is taken as a word rather than a guarded event in the cLTS $\mathcal{A}$. In this Büchi automaton, a string $\rho = a_1 a_2 ... a_n ...$ is accepted by $\mathcal{A}$ iff there exists a path $l_0 \xrightarrow{a_1} l_1 \xrightarrow{a_2} ... \xrightarrow{a_n} l_n ...$ in $\mathcal{A}$. We denote the set of strings accepted by $\mathcal{A}$ as $Str(\mathcal{A})$.*

According to the theory of Büchi automata, the language accepted by $\mathcal{A}$ is an ω-regular language, i.e., $Str(\mathcal{A}) \in Str$.

Proposition 6.1 is direct from the theory of Büchi automata, we omit the proof of it.

According to Definition 4.9 and Proposition 6.1, we propose the encoding from cLTS into CPM in Algorithm 1. Procedure *cLTS_2_CPM* takes a cLTS $\mathcal{A}$ as input and returns the corresponding CPM p as output. It directly follows the process of encoding a Büchi automaton into an ω-regular language. In *cLTS_2_CPM*, given a cLTS $\mathcal{A} = \langle L, T, l_0, \mathcal{C} \rangle$, we use $\mathcal{A}_{l,l'}$ to represent $\mathcal{A}$ as a non-deterministic finite automaton (NFA) with l the initial location and l' the single accepting location, denoted as $\mathcal{A}_{l,l'} = \langle L, T, l, \mathcal{C}, l' \rangle$. Note that $\mathcal{A}_{l,l'}$ and $\mathcal{A}$ share the same form, but differ in the accepting states and the types of accepting traces. Procedure *NFA_2_CPM* is called in procedure *cLTS_2_CPM*, it encodes $\mathcal{A}$, as an NFA $\mathcal{A}_{l,l'}$, into a CPM. Procedure *NFA_2_CPM* directly follows Brzozowski's method [30] for encoding an NFA into a regular language, whose main idea is to encode an NFA as a set of equations on regular expressions, then solve it by using Arden's rule [31]. A finite automaton $\mathcal{A}_{l,l'}$ with n locations can be encoded into exactly n equations following the principle explained in Algorithm 1, where each equation (i) $(1 \leq i \leq n)$ describes the transition relations between a location l_i to other locations (including itself). Using Arden's rule (explained in Proposition 6.2), we can solve the variables l_i one by one and finally obtain l_1 — the corresponding CPM of $\mathcal{A}_{l,l'}$.

Proposition 6.2 *(Arden's rule in CPM). In the regular part of CPM that excludes the infinite loop program of the form p^ω, given any CPMs p, q (where $q \not\equiv \varepsilon$), $X \equiv q^*; p$ is the unique solution of the equation $X \equiv p \oplus q; X$.*

Proposition 6.2 is straightforward since from Definition 4.9 we know that the regular part of CPM is exactly a regular expression.

From Algorithm 1 we observe that each CPM encoded from a cLTS is an infinite program, as stated in the following proposition.

Algorithm 1 Encoding cLTS into CPM.

procedure cLTS_2_CPM($\mathcal{A} = \langle L, T, l_0, \mathcal{C}\rangle$)
  for each $l \in L$ **do** /*compute the CPM for all NFAs $\mathcal{A}_{l_0,l}$ and $\mathcal{A}_{l,l}$*/
    $p_{l_0,l}$:= NFA_2_CPM($\mathcal{A}_{l_0,l}$)
    $p_{l,l}$:= NFA_2_CPM($\mathcal{A}_{l,l}$)
  $p := \oplus_{l\in L}(p_{l_0,l}; p_{l,l}^{\omega})$ /*compute the CPM of $\mathcal{A}^*$*/
  return p
procedure NFA_2_CPM($\mathcal{A}_{l,l'} = \langle L, T, l, \mathcal{C}, l'\rangle$) /* $|L| = n$ */
  Build a set of equations according to $\mathcal{A}_{l,l'}$:

$$l_1 \equiv b_1 \oplus p_{11}; l_1 \oplus p_{12}; l_2 \oplus \ldots \oplus p_{1n}; l_n \quad (1)$$

$$l_2 \equiv b_2 \oplus p_{21}; l_1 \oplus p_{22}; l_2 \oplus \ldots \oplus p_{2n}; l_n \quad (2)$$

$$\ldots$$

$$l_n \equiv b_n \oplus p_{n1}; l_1 \oplus p_{n2}; l_2 \oplus \ldots \oplus p_{nn}; l_n \quad (n)$$

  where $l_1, \ldots, l_n$ are variables, $l_1 = l$ is the initial location of $\mathcal{A}_{l,l'}$. p_{ij} $(1 \le i \le j \le n)$ is of the form $a_1 \oplus \ldots \oplus a_o$, with a_k $(1 \le k \le o)$ in the form of α or $g?\alpha$. In any equation (i) $(1 \le i \le n)$,
  (i) term '$p_{ij}; l_j$' $(1 \le j \le n)$ appears on the right side of (i) iff there exists a compositional transition $[l_i, p_{ij}, l_j]$ in $\mathcal{A}_{l,l'}$;
  (ii) $b_i = \varepsilon$ if $l_i = l'$;
  (iii) $b_i = \ddagger$ if $l_i \neq l'$.
  for each k, $k = n, n-1, \ldots, 2, 1$ **do**
    transform equation (k) into the form $l_k \equiv p \oplus q; l_k$.
    By Proposition 6.2, obtain $l_k \equiv q^*; p$ from $l_k \equiv p \oplus q; l_k$.
    substitute l_k on the right of other equations $(k-1)$, ..., (1) with $q^*; p$.
  return l_1.

Proposition 6.3. *Given a cLTS $\mathcal{A} = \langle L, T, l_0, \mathcal{C}\rangle$, let p = cLTS_2_CPM($\mathcal{A}$), then p is an infinite program, i.e., $val(p) \neq \emptyset$ and all traces of $val(p)$ are infinite traces.*

Proof. According to Algorithm 1, p has the form: $\oplus_{l\in L}(p_{l_0,l}; p_{l,l}^{\omega})$. Therefore the only possibilities that makes p not an infinite program are 1) $p = \ddagger$; 2) $p_{l,l}^{\omega} \equiv \varepsilon$ for some $l \in L$. However, both conditions 1) and 2) are impossible because according to the definition of cLTS (in Sect. 2.3), all locations of cLTS are accepting and for any $l \in L$, $(l, \emptyset, l)$ is a transition of $\mathcal{A}$. So p must be an infinite program. □

Example 6.1. The cLTS of the specification SP_1 (see Sect. 3): $\mathcal{A}_{sp1}$ (Fig. 4(d) in Sect. 2.3), is the synchronous product of the cLTSs of $u_1 \triangleq v_1 \$ 5$ (Fig. 4(a)) and free clocks v_1, v_3 (Fig. 4(b1), (b2)). By Algorithm 1, we can get $p_{sp1} = p_{l_4,l_4}^{\omega} \oplus p_{l_4,l_5}; p_{l_5,l_5}^{\omega}$, where

$$p_{l_4,l_4} = NFA\_2\_CPM(\mathcal{A}_{sp1,l_4,l_4}) = p_2^*,$$
$$p_{l_4,l_5} = NFA\_2\_CPM(\mathcal{A}_{sp1,l_4,l_5}) = p_2^*; p_3; p_4^*,$$
$$p_{l_5,l_5} = NFA\_2\_CPM(\mathcal{A}_{sp1,l_5,l_5}) = p_4^*,$$

p_2, p_3, p_4 are shown in the lower part of Fig. 4. Thus we have

$$p_{sp1} = (p_2^*)^{\omega} \oplus p_2^*; p_3; p_4^*; (p_4^*)^{\omega} \equiv p_2^{\omega} \oplus p_2^*; p_3; p_4^{\omega}.$$

In particular, we give the process of solving the equations in procedure $NFA\_2\_CPM(\mathcal{A}_{sp1,l_4,l_4})$. The set of equations of $\mathcal{A}_{sp1,l_4,l_4}$ is built as follows:

$$l_4 \equiv \varepsilon \oplus p_2; l_4 \oplus p_3; l_5 \quad (1)$$
$$l_5 \equiv \ddagger \oplus p_4; l_5 \quad (2)$$

In (2) by Proposition 6.2 we obtain $l_5 \equiv p_4^*; \ddagger$. Substituting l_5 in (1) and by Proposition 6.2 we obtain

$$l_4 \equiv p_2^*; (\varepsilon \oplus p_3; p_4^*; \ddagger) \equiv p_2^*; \varepsilon \equiv p_2^*.$$

According to Algorithm 1 we can conclude a natural correspondence between the semantics of a cLTS and its corresponding CPM.

Proposition 6.4 *(Relation between cLTS and CPM). Let $\mathcal{A}$ be a cLTS and $p_{\mathcal{A}}$ be its corresponding CPM obtained from Algorithm 1, then*

$$\{tr \mid \text{there is a } \sigma \in Sch(\mathcal{A}) \text{ s.t. } tr \in Tr^{\sigma}\}$$

is the set of all standard traces accepted by $p_{\mathcal{A}}$.

Proposition 6.4 says that the infinite behaviour of a cLTS exactly corresponds to the behaviour of its corresponding CPM obtained from Algorithm 1 in the sense of Definition 4.4. So if we can find a standard trace of a CPM, we then find a schedule of its corresponding cLTS.

The proof of Proposition 6.4 is given in Appendix A

6.2. Encoding clock relations into cDL

Unlike the behaviour of clocks, clock relations can be treated as static properties. As declared in the following proposition, they can be captured as a temporal formula in cDL.

Proposition 6.5 *(Encoding clock relations as temporal formulae). Given a set of clock relations $\widetilde{Rel}$, we can build a temporal formula as: $\psi_{\widetilde{Rel}} ::= \Box \bigwedge \hbar(Rel)$ s.t. $\sigma \vDash_{ccsl} \widetilde{Rel}$ iff $tr \vDash \psi_{\widetilde{Rel}}$ for any σ and any $tr \in Tr^{\sigma}$. $\hbar(Rel)$ is defined as:*

Rel	$\hbar(Rel)$	Rel	$\hbar(Rel)$
$c_1 \preceq c_2$	$h(c_1) \geq h(c_2)$	$c_1 \prec c_2$	$h(c_1) > h(c_2) \vee (h(c_1) = h(c_2) \wedge \eta(c_1) = 0)$
$c_1 \subseteq c_2$	$\eta(c_1) = 1 \rightarrow \eta(c_2) = 1$	$c_1 \# c_2$	$\eta(c_1) = 0 \vee \eta(c_2) = 0$

Proposition 6.5 can be directly proved by Definition 4.4, Definition 4.6 and the semantics of CCSL relations (Fig. 3). Here we omit its proof. Intuitively, for each clock relation $Rel \in \widetilde{Rel}$, $\hbar(Rel)$ exactly corresponds to the condition in the definition of Rel shown in Fig. 3, that the schedule must satisfy at each instant in order to satisfy Rel.

Example 6.2. In the clock relations $\{v_1 \prec v_3, v_3 \preceq u_1\}$ of SP_1, $\hbar(v_1 \prec v_3)$ corresponds to the condition in the definition of $v_1 \prec v_3$ that must be held at any instant $i \in \mathbb{N}^+$ by any schedule σ satisfying $v_1 \prec v_3$, i.e., '$H_\sigma(i, c_1) > H_\sigma(i, c_2) \vee (H_\sigma(i, c_1) = H_\sigma(i, c_2) \wedge c_1 \notin \sigma(i))$' (see Fig. 3).

The clock relations $\{v_1 \prec v_3, v_3 \preceq u_1\}$ of SP_1 can be expressed as

$$\begin{aligned}\Box\psi_{sp1} &= \Box(\hbar(v_1 \prec v_3) \wedge \hbar(v_3 \preceq u_1)) \\ &= \Box((h(v_1) > h(v_3) \vee (h(v_1) = h(v_3) \wedge \eta(v_3) = 0)) \wedge h(v_3) \geq h(u_1)).\end{aligned}$$

6.3. Encoding the schedule problem into cDL

In Sects. 6.1, 6.2, we have encoded the behaviour of all clocks and all clock relations of a specification as ingredients of cDL, the following proposition states that the schedule problem stated in Sect. 2 can be solved by proving a cDL formula.

Theorem 6.1 *(Schedule problem in cDL). Given a CCSL specification $SP = \langle \widetilde{Cdf}, \widetilde{Rel} \rangle$, the schedule problem of SP (stated in Sect. 2) holds iff cDL formula*

$$\phi_{SP} = I \rightarrow \langle p_{sp} \rangle (tt \sqcap \Box(\psi_{sp} \wedge \psi_{\emptyset}))$$

is valid, where p_{sp}, ψ_{sp} are obtained from SP through Proposition 6.4 and Proposition 6.5, $I = \bigwedge_{c \in \mathcal{C}(SP)} (h(c) = 0 \wedge \eta(c) = 0)$, $\psi_{\emptyset} = \bigvee_{c \in \mathcal{C}(SP)} \eta(c) = 1$.

I represents the initial environment of clock-related variables. We set all clock-related variables to 0, indicating at the beginning no clock ticks. This corresponds to $H_\sigma(0, c) = 0$ for any schedule σ and clock c. Since all traces of p_{sp} are infinite, the state property is set to tt. The formula $\psi_{\emptyset}$ means 'at least one clock ticks at any instant', which is used to avoid the schedules that contains $\emptyset$ at any instant (as explained in Sect. 2).

Proof of Proposition 6.1. Assuming that I holds initially, there exists an (infinite) trace tr of p_{sp} satisfying '$tt \sqcap \Box(\psi_{sp} \wedge \psi_{\emptyset})$', iff tr satisfies $\Box(\psi_{sp} \wedge \psi_{\emptyset})$, iff tr satisfies $\Box\psi_{sp}$ and in tr each state contains at least one variable $\eta(c)$ satisfying $\eta(c) = 1$. According to Propositions 6.4, 6.5, there exists a schedule σ of the corresponding cLTS of SP that satisfies $tr \in Tr^{\sigma}$ and $\sigma \vDash_{ccsl} \widetilde{Rel}$. By Proposition 2.1, we know $\sigma \vDash_{ccsl} \widetilde{Cdf}$, hence $\sigma \vDash_{ccsl} SP$. □

Example 6.3. We consider the schedule problem of SP_1. According to Theorem 6.1, it can be expressed as a cDL formula:

$$\phi_{sp1} = I_{sp1} \rightarrow \langle p_{sp1} \rangle tt \sqcap \Box(\psi_{sp1} \wedge \psi_{\emptyset}),$$

where $I_{sp1} = \bigwedge_{c \in \{v_1, v_3, u_1\}} h(c) = 0 \wedge \eta(c) = 0$, $\psi_{\emptyset} = \bigvee_{c \in \{v_1, v_3, u_1\}} \eta(c) = 1$. p_{sp1}, ψ_{sp1} were given in Examples 6.1, 6.2 respectively.

Algorithm 2 Generating a schedule from proof tree.

```
1: procedure GEN_SCH(ζ0 = ⟨ν0, τ0⟩, κ0) /* ζ0 is a valid proof tree */
2:     Ξ := {ζ0} /*nodes remaining to be analyzed*/
3:     κ := κ0 /*initialize κ*/
4:     while Ξ ≠ ∅ do
5:         take a ζ = ⟨ν, τ⟩ in Ξ and remove it from Ξ
6:         if ν = ∅ then continue /*a leaf node*/
7:         if τ ='(α)' then
8:             put all nodes of ν in Ξ
9:             α := T(ζ) /*α is the 'target event' of rule (α)*/
10:            κ := κα
11:            continue
12:        else if τ ='(⟨∗⟩⊓)' then /*set ν = (ζ1, ζ2, ζ3)*/
13:            put ζ3 in Ξ
14:            κ′ := GEN_SCH(ζ2, λ)
15:            κ := κ(κ′)^k /*k is a witness of '∃x.Inv(x)' in ζ1*/
16:            continue
17:        else if τ ='(⟨ω⟩⊓)' then /*set ν = (ζ1, ζ2)*/
18:            κ′ := GEN_SCH(ζ2, λ)
19:            κ := κ(κ′)^ω
20:            continue
21:        else /*where τ can be other cases, they do not contribute to the body of the schedule*/
22:            put all nodes of ν in Ξ
23:            continue
24:    return κ
```

6.4. Solving the schedule problem

In Theorem 6.1, if ϕ_{SP} is valid, the derivation of ϕ_{SP} actually provides a hint of what the schedules of *SP* may look like. Essentially, the valid proof tree of ϕ_{SP} itself can be seen as a special 'transition system' that captures the behaviour of all schedules of *SP*. By analyzing this proof tree, one can generate a possible schedule of *SP* expressed in the form of $\kappa_1\kappa_2^{\omega}$.

The generation procedure is described as Algorithm 2, where procedure *Gen_Sch* takes a valid proof tree ζ_0 and a clock sequence κ_0 as inputs, and outputs a sequence κ. In Algorithm 2 we use ':=' to mean the assignment and '=' to represent the logical equality. Starting from the root node ζ_0, procedure *Gen_Sch* traverses each node of the tree and consecutively updates the sequence κ according to the rule at each node. Only 3 rules cause the change of the sequence κ: (α), $(\langle*\rangle\sqcap)$ and $(\langle\omega\rangle\sqcap)$. In the analysis we only consider formulae of the forms defined in Definition 4.2 so we do not need to consider rules $([\omega]\sqcup)$, $([*]\sqcup)$. At line 10, sequence κ is appended with an event α if rule (α) is applied at the current node. $T(\zeta)$ returns the target event of rule (α) (if rule (α) is applied to this node). E.g., for node 6 of the proof tree 1 in Fig. 12, we have $T(6) = \{v_1\}$. At line 15, the sequence κ is appended with a sequence $(\kappa')^k$, where κ' is computed by another invocation of procedure *Gen_Sch*. k is a value manually found to make the formula $\exists x.Inv(x)$ valid. The nodes $\zeta_1, \zeta_2, \zeta_3$ correspond to the child nodes of rule $(\langle*\rangle\sqcap)$ respectively (see rule $(\langle*\rangle\sqcap)$ in Fig. 10(b)). At line 19, sequence κ is appended with an infinite sequence $(\kappa')^{\omega}$, where κ' is computed by another invocation of procedure *Gen_Sch*. The nodes ζ_1, ζ_2 correspond to the child nodes of rule $(\langle\omega\rangle\sqcap)$ respectively (see rule $(\langle\omega\rangle\sqcap)$ in Fig. 10(a)).

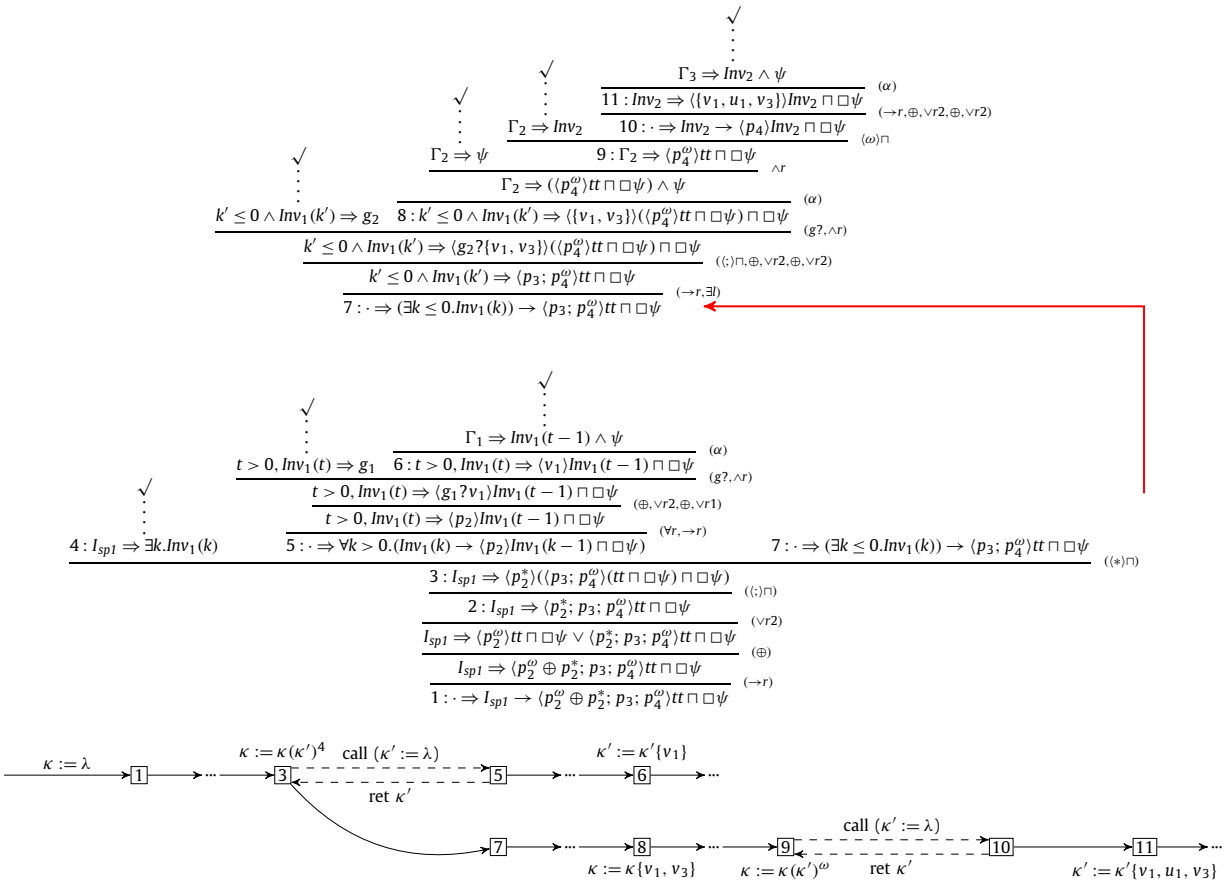
Theorem 6.2 *(Schedule generation in cDL). Given a CCSL specification* $SP = \langle \widetilde{Cdf}, \widetilde{Rel}\rangle$ *and the cDL formula* $\phi_{SP} = I \rightarrow \langle p_{sp}\rangle(tt \sqcap \Box(\psi_{sp} \wedge \psi_{\emptyset}))$ *of the schedule problem of SP, assume* ϕ_{SP} *is valid and generates a proof tree* ζ_0*, let* σ_0 *be the schedule generated by procedure Gen_Sch*$(\zeta_0, \emptyset)$ *given in Algorithm 2, then*

σ_0 *is a schedule of SP.*

The proof of Theorem 6.2 is given in Appendix A.

Example 6.4. Fig. 12 shows the derivation of the formula ϕ_{sp1} in Example 6.3. In order to save space, we use a derivation with multiple rules: $\dfrac{\zeta_2}{\zeta_1}\ {}^{(r_1,r_2,\ldots,r_n)}$ to mean that there are n $(n \geq 1)$ derivation steps between node ζ_1 and node ζ_2, and the names of the inference rules being applied by each derivation step are listed on the right in order. E.g., the derivation from node 1 to node 2 can be abbreviated as a derivation with multiple rules:

$$\frac{2 : I_{sp1} \Rightarrow \langle p_2^*; p_3; p_4^{\omega}\rangle tt \sqcap \Box\psi}{1 : \cdot \Rightarrow I_{sp1} \rightarrow \langle p_2^{\omega} \oplus p_2^*; p_3; p_4^{\omega}\rangle tt \sqcap \Box\psi}\ {}^{(\rightarrow r, \oplus, \vee r2)}$$



$p_2 = \emptyset \oplus g_1?\{v_1\} \oplus g_1?\{v_1, v_3\}$	$p_3 = \emptyset \oplus g_2?\{v_1\} \oplus g_2?\{v_1, v_3\}$
$p_4 = \emptyset \oplus \{v_1, u_1\} \oplus \{v_1, u_1, v_3\}$	$g_1 = h(v_1, u_1) < 4$
$g_2 = h(v_1, u_1) = 4$	$\psi = \psi_{sp1} \wedge \psi_{\emptyset}$
$Inv_1(k) = (h(v_1, u_1) = 4 - k) \wedge (h(v_1, v_3) \geq 4 - k) \wedge h(v_3, u_1) \geq 0 \wedge 0 \leq k \leq 4$	$Inv_2 = h(v_1, v_3) > 0 \wedge h(v_3, u_1) \geq 0$
$\Gamma_1 = \{t > 0, Inv_1(t)\}[x_1, x_2, x_3, x_4/h(v_1), \eta(v_1), \eta(v_3), \eta(u_1)] \cup \{h(v_1) = x_1 + 1, \eta(v_1) = 1, \eta(v_3) = 0, \eta(u_1) = 0\}$	
$\Gamma_2 = (k' \leq 0 \wedge Inv_1(k'))[y_1, y_2, y_3, y_4, y_5/h(v_1), h(v_3), \eta(v_1), \eta(v_3), \eta(u_1)] \cup \{h(v_1) = y_1 + 1, h(v_3) = y_2 + 1, \eta(v_1) = 1, \eta(v_3) = 1, \eta(u_1) = 0\}$	
$\Gamma_3 = Inv_2[z_1, z_2, z_3, z_4, z_5, z_6/h(v_1), h(v_3), h(u_1), \eta(v_1), \eta(v_3), \eta(u_1)] \cup \{h(v_1) = z_1 + 1, h(v_3) = z_2 + 1, h(u_1) = z_3 + 1, \eta(v_1) = 1, \eta(v_3) = 1, \eta(u_1) = 1\}$	

Fig. 12. Derivation of formula $I_{sp1} \rightarrow \langle p_{sp1} \rangle tt \sqcap \Box(\psi_{sp1} \wedge \psi_{\emptyset})$.

Due to the limit of space, we move the derivation of node 7 to the top. We use $\overset{\surd}{\vdots}\ \zeta$ to denote the branch where from node ζ, all derivations will end in valid nodes $\surd$. The lower part of Fig. 12 gives the detailed content of each node, where $p_1 - p_4$, g_1, g_2 have been declared in Fig. 4. $x_1, ..., x_4$, $y_1, ..., y_5$, $z_1, ..., z_6$ are new variables with respect to their contexts.

The derivation starts from the root 1, and stops if (i) all leaf nodes are valid nodes ($\surd$), or (ii) one of the leaf nodes is not valid. The derivation of the formula ϕ_{sp1} is a valid proof tree, with all leaf nodes are valid. The whole proof procedure is semi-automatic, while the only places that need manual intervention are nodes 3 and 9, where two invariants Inv_1, Inv_2 need to be determined.

By calling procedure $Gen\_Sch(1, \lambda)$ (where λ is a special string defined in Definition 4.8), finally we can generate a schedule

$$\sigma = v_1^4\{v_1, v_3\}\{v_1, v_3, u_1\}^{\omega}$$

of SP_1. The path below the derivation in Fig. 12 shows the process of node enumeration in procedure *Gen_Sch*, where variables κ, κ' denote the corresponding variables in Algorithm 2, 'call/ret' means the calling/return of a new procedure *Gen_Sch*. The value of the variable κ is initiated to λ before dealing with node 1. The value of the variable κ is updated after dealing with nodes 3, 6, 8, 9 and 11 in procedure *Gen_Sch*. E.g., after dealing with node 3, the variable κ is appended by $(\kappa')^4$ where the variable $\kappa' = \{v_1\}$ is computed from the branch $5 \rightarrow ... \rightarrow 6 \rightarrow ...$, while the witness of $\exists k.Inv_1(k)$ at

Table 1
The definition of formula ϕ_{sp1} and its sequent in Coq.

```
1  (* Example in the paper*)
2  (* clocks v1, u1 and v3*)
3  Definition v1 : CName := (clk 1).
4  Definition u1 : CName := (clk 2).
5  Definition v3 : CName := (clk 3).
6
7  (* program p *)
8  (* g1, g2*)
9  Definition g1 : g_exp := D(v1, u1) < 4.
10 Definition g2 : g_exp := D(v1, u1) == 4.
11
12 (* p2, p3, p4*)
13 Definition p2 : CPM_exp := @ nil U g1?{v1} U g1?{v1|v3}.
14 Definition p3 : CPM_exp := @ nil U g2?{v1} U g2?{v1|v3}.
15 Definition p4 : CPM_exp := @ nil U @ {v1|u1} U @ {v1|u1|v3}.
16 (* p *)
17 Definition p : CPM_exp := p2^w U (p2^* ; p3 ; p4^w).
18
19 (* initial condition I *)
20 Definition I : cDL_exp := (h(v1) =' 0 /\' y(v1) =' 0) /\'
21 (h(u1) =' 0 /\' y(u1) =' 0) /\' (h(v3) =' 0 /\' y(v3) =' 0).
22
23 (* formula psi *)
24 (* formula expressing the specification *)
25 Definition psi_sp1 := (h(v1) >' h(v3) \/' (h(v1) =' h(v3) /\' y(v3) ='
       0)) /\' h(v3) >=' h(u1).
26 (* formula for expressing 'at least one clock ticks at each instant of
       a schedule' *)
27 Definition psi_es := y(v1) =' 1 \/' y(u1) =' 1 \/' y(v3) =' 1.
28 (* psi *)
29 Definition Psi := psi_sp1 /\' psi_es.
30
31 (* Other components of the sequent *)
32 Definition Gamma0 : Gamma := nil.
33 Definition Delta0 : Delta := nil .
34 Definition V0 : list Var := nil .
35 Definition C0 : list CName := v1 :: u1 :: v3 :: nil .
36 Definition ntC0 : list CName := C0.
37
38 Theorem Example : <| Gamma0 , empty ==>
39 exp ( I —>' << p >> (tt' , Psi) ) , Delta0 // V0, C0, ntC0 |>.
```

node 4 is 4. At node 8, the variable κ is appended by an event $\{v_1, v_3\}$ due to the derivation of rule (α). Similar analysis can be given for nodes 6, 9 and 11.

7. Mechanization of cDL

In order to show the potential applicability of our method, we have mechanized the cDL calculus in Coq [18] — a theorem prover based on a type of higher-order typed $\lambda-$calculus called 'calculus of inductive constructions' (CIC). We use CIC to define the syntax of cDL and to define the sequents of cDL as the propositions of CIC in an inductive way based on the proof system given in Fig. 10. Each rule in the proof system of cDL is treated as an 'axiom' in Coq. To prove a cDL formula, which is expressed as a $\lambda-$expression in CIC, we deduce it by applying these axioms in Coq. The axioms transform the $\lambda-$expression into a quantifier-free arithmetical proposition in CIC, which can then be either proved in Coq or solved in an SMT-procedure using tools like Z3. When the proposition is a linear integer arithmetic logic formula, it can also be solved by the 'Omega' solver embedded in Coq [18].

In this paper, we only show how a cDL formula can be expressed and verified in Coq through an example. More details about our implementation can be found online[2] (where the example given below can also be found at the end of the code file). We take the formula $\phi_{sp1} = I_{sp1} \to \langle p_{sp1}\rangle tt \sqcap \Box(\psi_{sp1} \wedge \psi_{\emptyset})$ (in Example 6.3) as an example. Table 1 shows the definition of the formula ϕ_{sp1} and the sequent $\cdot \Rightarrow \phi_{sp1}$ in Coq.

In Coq, CPM and cDL formulae are defined as the inductive types CPM_exp and cDL_exp respectively. Clock names and variables are defined as the types CName and Var. The CPM p_{sp1} is defined as p at line 17, where p2, p3 and p4 define the programs p_2, p_3 and p_4 (shown in Fig. 4) respectively, the guards g_1 and g_2, the variables v_1, u_1 and v_3 are defined as g1, g2, v1, u1 and v3 respectively. The symbols ;, U, ^w and ^* define the operators ;, $\oplus$, ω and $*$ respectively. {a1 | a2 | ... | an} denotes a list of elements a1, ..., an (with nil denoting the empty list) in Coq. {c1 | c2 | ... | cn} with c1, ..., cn : CName defines the set of clocks $\{c_1, c_2, ..., c_n\}$. @{c1 | c2 | ... | cn} denotes the clock event $\{c_1, c_2, ..., c_n\}$ in CPM. g?{c1 | c2 | ... | cn} defines the guarded clock event $g?\{c_1, ..., c_n\}$, with g : g_exp defining the guard g. The guard expression is defined as the type g_exp, where D(c1, c2) represents $h(c_1, c_2)$ — the distance between $h(c_1)$ and $h(c_2)$ (in Sect. 2.3). The initial condition I_{sp1} and the formula $\psi = \psi_{sp1} \wedge \psi_{\emptyset}$ are defined as I and Psi at line 20 and line 29 respectively, where h(c), y(c) represent clock-related variables $h(c), \eta(c)$ respectively. The symbols ~', /\', \/', —>' denote the logic connectives $\neg, \wedge, \vee, \to$ respectively, while the symbols <=', <', >', >=', =' represent the operators $\leq, <, >, \geq, =$ in expression e.

The sequent $\cdot \Rightarrow \phi_{sp1}$ is defined as the theorem Example in Coq (at line 38). A sequent is defined as a 4-tuple <| G, pls1 ==> pls2, D |>, where G and D represent the contexts Γ and Δ respectively, pls1, pls2 are places for formulae that are targeted at current sequent, they can be either empty (represented by empty) or contain a target formula ϕ (expressed as exp phi). The formula ϕ_{sp1} is defined as I —>' << p >> (tt', Psi), where << p >> (phi, psi) represents the dynamic formula $\langle p\rangle\phi \sqcap \psi$. V0, C0, ntC0 are auxiliary variables used in the derivation procedure.

We prove the theorem Example in Coq by applying the axioms defined as the rules of the proof system of cDL (Table 10). Table 2 shows the procedure of proving theorem Example in Coq, where the correspondence relations between some axioms and rules are declared in the lower part of the table. Compared to other rules, rule (α) is special in the implementation. In Coq we use 3 axioms to split the simultaneous occurrences of clocks in a clock event, which is captured by the single rule (α) in the proof system of cDL. Axiom r_Alpha_c deals with the occurrence of each clock, axiom r_Alpha_idle2 deals with

[2] https://github.com/antitaboo/cDL-for-SA-of-CCSL/.

Table 2
The procedure of proving Theorem Example.

```
1  Proof.
2  apply r_impR. (* rule −> r *)
3  apply r_Choice. (* rule choice *)
4  apply r_orR_2. (* rule \/ r 2 *)
5  apply r_Seq. (* rule sequence *)
6  apply r_Star_dia; exists inv1; split ; [| split ]. (* rule <<star>> *)
7   − (*@*) apply r_extR. exists 4. simpl. split . (* rule ext r *)
8    + (*@*) intuition . + apply r_o. cbn. intros . omega. (* rule o *)
9    − apply r_allR. apply r_impR; apply r_impR. cbn. (* rule all r, rule
          −> r *)
10     apply r_Choice. apply r_orR_2. apply r_Choice. apply r_orR_1. (*
           rule cho, \/ r 2, cho, \/ r 2 *)
11     apply r_Guard. apply r_andR; split. (* rule g?, /\ r *)
12     + (*@*) cbv. apply r_o;cbn. intros . omega. (* rule o *)
13     + cbv. apply r_Alpha_c. apply r_Alpha_idle2; apply r_Alpha_idle2;
           apply r_Alpha_idle1; cbv. (* rule alpha *)
14       (*@*) apply r_o. cbn. intros . omega. (* rule o *)
15   − apply r_impR. apply r_placeL_add. apply r_extL. cbn. (* rule −> r,
         ext l *)
16     apply r_Seq. apply r_Choice. apply r_orR_2. apply r_Choice. apply
         r_orR_2. (* rule sequence, cho, \/ r 2, cho, \/ r 2 *)
17     apply r_Guard. apply r_andR; split. (* rule g?, /\ r *)
18      * (*@*) apply r_o. cbn. intros . omega. (* rule o *)
19      * apply r_Alpha_c; apply r_Alpha_c; apply r_Alpha_idle2; apply
           r_Alpha_idle1. cbn. (* rule alpha *)
20        apply r_andR; split . Focus 2. (* rule /\ r *)
21        { (*@*) apply r_o. cbn. intros . omega. } Unfocus. (* rule o *)
22        { apply r_Omega_dia; exists inv2; split . (* rule <<omega>> *)
23       − (*@*) apply r_o. cbn. intros. omega. (* rule o *)
24       − apply r_impR. apply r_Choice. apply r_orR_2. apply r_Choice.
           apply r_orR_2. (* rule −> r, cho, \/ r 2, cho, \/ r 2 *)
25         apply r_Alpha_c; apply r_Alpha_c; apply r_Alpha_c; apply
           r_Alpha_idle1; cbv. (* rule alpha *)
26        (*@*) apply r_o. cbn. intros . omega. (* rule alpha *)
27       }
28 Qed.
```

Axioms in Coq	Rules in cCDL	Axioms in Coq	Rules in cCDL	Axioms in Coq	Rules in cCDL	Axioms in Coq	Rules in cCDL
r_Alpha_c, r_Alpha_idle1, r_Alpha_idel2	(α)	r_Seq	$(\langle;\rangle\sqcap)$	r_Guard	$(g?)$	r_Choice	$(\oplus)$
r_Star_dia	$(\langle*\rangle\sqcap)$	r_Omega_dia	$(\langle\omega\rangle\sqcap)$	r_o	(o)	r_andR	$(\wedge r)$
r_allR	$(\forall r)$	r_orR_1	$(\vee r1)$	r_orR_2	$(\vee r2)$	r_impR	$(\rightarrow r)$
r_extR	$(\exists r)$	r_extL	$(\exists l)$				

the behaviour of the clocks that do not occur, while axiom r_Alpha_idle1 ends the whole procedure if the behaviours of all clocks have been dealt with. We will not give the details of all axioms here, interested readers can refer to the code online for more details.

The whole procedure of the proof in Table 2 corresponds to the derivation process of formula ϕ_{sp1} given in Fig. 12, with each line corresponding to exactly one derivation step in Fig. 12 ('(*@*)' indicating the lines whose corresponding derivation is omitted in Fig. 12). For example, line 6 corresponds to the derivation from node 3 to nodes 4, 5, 7 in Fig. 12 where rule $(\langle*\rangle\sqcap)$ is applied. Line 7, 9 and 15 correspond to the derivation from node 4, 5 and 7 of Fig. 12 respectively. For each derivation in Fig. 12, several tactics may need to be applied. E.g., at line 7, which corresponds to the derivation from node 4 (not shown in Fig. 12), after applying axiom r_extR, we need to find a witness of k by using tactic exists, and then split the conjunction with the tactic split. Any detailed introduction of these tactics is beyond the scope of this paper, one can refer to [18] for more details.

The whole procedure of the proof terminates successfully with the key word 'Qed' (at line 28). All leaf nodes of the proof tree end by using the tactic omega to solve the obtained quantifier-free linear integer arithmetic propositions in CIC (at lines 8, 12, 14, 18, 21, 23, 26). However, this is not the general case, sometimes manual work might be needed for the arithmetic propositions obtained at the leaf nodes might be non-linear. A more powerful SMT-solving tool (such as Z3) could be used as a back-end tool to deal with these propositions in an efficient way.

Currently, this proof has been made entirely manually. But we can design a tactical in Coq to automatize most of these selections of tactics. And that could be one of our future work.

Currently we do not realize the schedule generation algorithm (Algorithm 2). It can be considered either implemented in Coq or implemented as an independent tool that receives valid proof trees produced by Coq as inputs. More work will be put on that in the future.

8. Related work and discussion

8.1. Schedulability analysis of CCSL specifications

In CCSL, the state space of the cLTS of a CCSL specification depends on $h(c_1, c_2)$ — the difference of the number of ticks of two clocks c_1 and c_2. If all such differences are bounded, then this cLTS has a finite number of states. A CCSL specification is often called 'safe' [13] if its corresponding cLTS is finite.

Previous approaches for solving the schedule problem of CCSL specifications are mainly based on model checking and SMT-solving techniques.

In the model-checking-based approach [10,11], the schedule problem of a CCSL specification can be translated into the reachability problem of finite state automata. A CCSL specification is transformed into a finite automaton, then reachability

analysis is made on its states. [10] proposes to encode a CCSL specification into a finite SPIN automaton, whose validity can be checked by the model checking tool SPIN [32]. For unsafe specifications, a bound needs to be set to acquire finite state automata, so only approximate checking is possible. E.g., in some previous works like [12,20], relation $c_1 \prec_n c_2$ has been adopted to bound the unsafe specification $c_1 \prec c_2$, where n is a bound to limit the gap between the distance of the ticks between c_1 and c_2. In [11], a rewriting system is built for CCSL and the behaviour of CCSL clocks is explored by the rewriting tool Maude [33]. By rewriting CCSL expressions, any specification can be expanded into a (possibly infinite) transition system and bounded model checking is then applied.

In the SMT-solving-based approach [9,12], a CCSL specification is directly encoded into an FOL formula with quantifiers according to the semantics of CCSL shown in Fig. 3. This FOL formula captures the conditions that should hold at each instant for the specification (e.g., the formula ϕ' in Fig. 1). To solve the FOL formula, a bound needs to be set in order to eliminate all quantifiers, and this bound indicates the number of instants at which the conditions are checked in an SMT-solving procedure. E.g., after setting a bound n for formula ϕ' in Fig. 1, it becomes $\bigwedge_{1\leq i\leq n} H_\sigma(i, c_1) \geq H_\sigma(i, c_2)$ and we can solve it in an SMT-solving procedure. By solving the FOL formula, a bounded schedule is obtained. [9] proposed a sufficient (but not necessary) condition to check if a periodic schedule can be obtained from this bounded schedule. In this approach, whether a periodic schedule can be found depends on the bound set for solving the FOL formula: If the bound is proper, a schedule can be found; If the bound is too small, it might happen that no schedules can be found, but we still do not know whether there exists a schedule or not.

Our approach to schedulability analysis of CCSL specifications is based on theorem proving. Compared to the previous approaches, our approach is not limited to special types of CCSL specifications and does not depend on the bound that is set for approximate checking. Different from the SMT-solving-based approach, which directly encodes a specification into an FOL formula with quantifiers, our approach encodes a specification into a dynamic cDL formula. And by proving this dynamic formula or its negation in cDL calculus, we can know exactly whether there exists a schedule or not and can generate one if it exists. The proving process decomposes the dynamic formula into QF-FOL formulae according to the syntactic structure of the CPM so that a bound can be avoided.

Our approach can be considered as a complement to the previous approaches. When no schedule can be found after even setting a large bound, our approach can be adopted to try to give a formal proof of whether a schedule exists or not. The structure of the CPM that reveals the behaviour of clocks of the CCSL specification can provide some hints on why a schedule can or cannot exist, so as to help people have a better understanding of the specification. On the other side, when problems are hard or fail to be proved by our approach, we can try the previous approaches to perform an approximate checking with a proper bound.

The main disadvantage of our approach is that the verification procedure of cDL formulae is generally undecidable, which means that we have to manually search loop invariants in the verification procedure. However, considering the simplicity of CPM and the peculiarity of CCSL relations, it is possible to propose an automatic algorithm for generating loop invariants for CPMs and logical formulae with particular shapes, which could be one of our future work.

8.2. Dynamic logic

Dynamic logic [34] is a formalism for modelling and reasoning about program specifications. Classical dynamic logics like FODL [14] only capture state properties of programs. Process logic [35] firstly introduces temporal logic in dynamic logic to express temporal properties of programs, where formulae of the form '$\langle p\rangle\Diamond\psi$' were introduced. Later [36] proposes a dynamic logic with modalities where formulae of the form '$\ll p \gg \psi$' were introduced to mean the same as $\langle p\rangle\Diamond\psi$ in process logic. Process logic (similar for the dynamic logic with modalities) can only prove formulae of the form $\langle p\rangle\Diamond\psi$, for formulae of the form '$\langle p\rangle\Box\psi$', no inference rules were proposed for the sequential program $p = q; r$. dTL^2 [15] introduces the normalized trace formulae of the form '$\langle p\rangle\phi \sqcap \Box\psi$' and their related rules to solve this problem.

The syntax and semantics of cDL are largely based on and extended from FODL and dTL^2. In cDL we add the infinite loop operator ω to capture the infinite clock behaviour in cLTS. The operator ω is necessary for capturing the schedule problem of CCSL and its effect cannot be subsumed by the finite loop operator $*$. In fact, formula $\langle p^\omega\rangle\Box\psi$ and formula $\langle p^*\rangle\Box\psi$ have different meanings (the former is stronger than the latter), and so do their rules. Because of this, it is clear that cDL has a different expressiveness from dTL^2. More analysis will focus on the theory of cDL in the future.

9. Conclusion and future work

In this paper, we proposed a theorem-proving approach to schedulability analysis of CCSL specifications. To this end, we propose a variation of dynamic logic cDL and its proof calculus, based on which we analyze the schedule problem in CCSL. Based on FODL, cDL inherits the normalized trace formulae from dTL^2 and introduces clock events and the infinite loop operator ω as an extension in order to capture the cLTS models of CCSL specifications. With cDL, the schedule problem can be expressed as a cDL formula and so as can be checked by verifying this formula in a semi-automatic way combining theorem proving and SMT-solving. We also propose an algorithm for generating a schedule by analyzing the valid proof tree generalized by a verification procedure. To show the potential applicability of our method we mechanize cDL in Coq. Through an example, all these points are clearly illustrated.

The future work may focus on the following aspects: (1) Improving the implementation of cDL and evaluating its applicability in practice; (2) Analyzing the relative completeness of cDL and the decidability of the loop invariants in CPM; (3) Proposing a general methodology for schedulability analysis of RTESs based on dynamic logic. As a general dynamic logic, the application of cDL should not be limited to CCSL specifications. We believe it could also be used for schedulability analysis of other specifications of synchronous systems, whose behaviour can be captured as a program model in cDL.

CRediT authorship contribution statement

Yuanrui Zhang: Conceptualization, Methodology, Software, Writing - original draft, Writing - review & editing. **Frédéric Mallet:** Conceptualization, Supervision, Writing - review & editing. **Huibiao Zhu:** Writing - review & editing. **Yixiang Chen:** Writing - review & editing. **Bo Liu:** Writing - review & editing. **Zhiming Liu:** Funding acquisition, Supervision, Writing - review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

We thank all the anonymous reviewers including those who reviewed the conference version published at TASE 2019 for their valuable comments on this work. This work was partly supported by the Capacity Development Grant of Southwest University (No. SWU116007), the NSFC (Key Project) (No. 61732019), the NSFC (Regular Project) (No. 61672435), the NSFC-RS Project (No. 61811530327), the Special Foundation for Basic Science and Frontier Technology Research Program of Chongqing (No. cstc2017jcyjAX0295), the National Key Research and Development Program of China (No. 2018YFB2101300), and the National Natural Science Foundation of China (No. 61872145).

Appendix A. Proof of Proposition 6.4 and Theorem 6.2

Proof of Proposition 6.4. Given a cLTS $\mathcal{A} = \langle L, T, l_0, C\rangle$, let $p = cLTS\_2\_CPM(\mathcal{A})$, first we show that $Str(\mathcal{A}) = Str(p)$, i.e., the language accepted by $\mathcal{A}$ (as a Büchi automaton) is exactly the language of p (as an ω-regular expression). From Proposition 6.1, $\mathcal{A}$ can be taken as a Büchi automaton that accepts a subset of the ω-regular language Str. According to the theory of transforming Büchi automata into ω-regular language [26], we have $Str(\mathcal{A}) = \bigcup_{l\in L} Str(\mathcal{A}_{l_0,l})Str(\mathcal{A}_{l,l})^{\omega}$, where L is the set of all accepting states in the Büchi automaton $\mathcal{A}$. For any $\mathcal{A}_{l,l'}$ and $p_{l,l'} = NFA\_2\_CPM(\mathcal{A}_{l,l'})$, since procedure NFA_2_CPM is a standard process of transforming a finite automaton into a regular expression and the regular part of CPM (that excludes the infinite loop program of the form p^{ω}) is a regular language, we have $Str(\mathcal{A}_{l,l'}) = Str(p_{l,l'})$. Therefore $Str(\mathcal{A}) = \bigcup_{l\in L} Str(p_{l_0,l})Str(p_{l,l})^{\omega}$. By Definition 4.9 we have $Str(\mathcal{A}) = Str(\oplus_{l\in L} p_{l_0,l}; p_{l,l}^{\omega}) = Str(p)$.

Now we get back to Proposition 6.4. For any schedule $\sigma = \alpha_1\alpha_2...\alpha_n...$, by the definition of $Sch(\mathcal{A})$ (in Sect. 2.3), we have $\sigma \in Sch(\mathcal{A})$ iff there exists a path $l_0 \xrightarrow{a_1} l_1 \xrightarrow{a_2} l_2 \xrightarrow{a_3} ... \xrightarrow{a_{n-1}} l_n \xrightarrow{a_n} ...$ in $\mathcal{A}$, where a_i ($i \in \mathbb{N}^+$) can be α_i or $g_i?\alpha_i$. Then we have there is a string $\rho = a_1a_2...a_n...$ accepted by the Büchi automaton $\mathcal{A}$, i.e. $\rho \in Str(\mathcal{A})$. Since $Str(\mathcal{A}) = Str(p)$, $\rho \in Str(p)$. By Proposition 4.2 we know that Tr^{σ} is the set of all standard traces in $val(\alpha_1\alpha_2...\alpha_n...)$. Moreover, by the fact that $\sigma \in Sch(\mathcal{A})$, it is easy to see that every standard trace in $val(\alpha_1\alpha_2...\alpha_n...)$ is a standard trace in $val(\rho)$, because every guard g_i of α_i (suppose $a_i = g_i?\alpha_i$) is true in $\mathcal{A}$. Hence every trace of Tr^{σ} is a standard trace in $val(\rho)$.

On the other side, for any standard trace $tr \in val(p)$, from Proposition 6.3 tr must be infinite. By Proposition 4.1, there must exist a string $\rho' = a'_1a'_2...a'_n... \in Str(p)$ s.t. $tr \in val(\rho')$. Since $Str(p) = Str(\mathcal{A})$, $\rho' \in Str(\mathcal{A})$. Because $val(\rho') \neq \emptyset$, so there exists a path $l_0 \xrightarrow{a'_1} l_1 \xrightarrow{a'_2} l_2 \xrightarrow{a'_3} ... \xrightarrow{a'_{n-1}} l_n \xrightarrow{a'_n} ...$ in the cLTS $\mathcal{A}$. Let σ' be the schedule obtained by removing all guards in ρ', we have $\sigma' \in Sch(\mathcal{A})$. □

In the proof of Theorem 6.2 given below, we use $\dfrac{\zeta_2}{\zeta_1}\ {}^{(r_1,r_2,...,r_n)}$ to denote a derivation with multiple rules: there are n ($n \geq 1$) derivation steps between node ζ_1 and ζ_2, and the names of the inference rules applied by each derivation step are listed on the right in order.

Proof of Theorem 6.2. For any string ρ, let $\kappa(\rho)$ be the clock sequence obtained from ρ by removing all guards in the clock events of ρ.

In the valid proof tree of the sequent $\cdot \Rightarrow \phi_{SP}$, for any node ζ with a sequent of the form $\Gamma \Rightarrow \langle p\rangle\phi \sqcap \Box\psi, \Delta$ (where $\langle p\rangle\phi \sqcap \Box\psi$ is the target formula), let $\zeta(p)$ be the proof tree which starts at node ζ as the root node, but ends at a node where program p has just been eliminated in formula $\langle p\rangle\phi \sqcap \Box\psi$. Let κ be the clock sequence generated by analyzing the proof tree $\zeta(p)$ through procedure Gen_Sch in Algorithm 2, we prove:

(a) there exists a string ρ s.t. $\rho \in Str(p)$ and $\kappa(\rho) = \kappa$;
(b) under the context Γ (i.e., provided that all formulae in Γ are true), there exists a trace $tr \in val(\rho)$ s.t. $tr \vDash \Box\psi$ and $tr_e \in val(\phi)$ if tr_e exists.

We prove (a), (b) by induction on the structure of p.

For the base case, according to Proposition 6.3, there is only one situation when $p = a$ where a is of the form α or $g?\alpha$. We take $p = g?\alpha$ as an example, the case for $p = \alpha$ is similar. The proof tree $\zeta(p)$ is in the form of:

$$\dfrac{\dfrac{\dfrac{\dots}{\Gamma \Rightarrow g, \Delta} \quad \dfrac{(1) : \dots}{\Gamma \Rightarrow \langle\alpha\rangle\phi \sqcap \Box\psi, \Delta}\ {\scriptstyle (\alpha)}}{\zeta : \Gamma \Rightarrow \langle g?\alpha\rangle\phi \sqcap \Box\psi, \Delta}\ {\scriptstyle (g?, \wedge r)}}{\dots}$$

which ends at node (1) because at this node program p has just been eliminated. According to procedure *Gen_Sch* of Algorithm 2, we know $\kappa = \alpha$. Let $\rho = g?\alpha$, clearly we have $\rho \in Str(p)$ and $\kappa(\rho) = \kappa$. Since $\rho = p$, obviously under the context Γ there is a trace $tr \in val(\rho)$ satisfying $tr \vDash \Box\psi$ and $tr_e \in val(\phi)$ if tr_e exists.

For the induction step, we have 4 cases:

(i) $p = q; r$. The proof tree $\zeta(q; r)$ is in the form shown as follows:

$$\dfrac{\dfrac{\dfrac{\dots}{\zeta_2 : \Gamma' \Rightarrow \langle r\rangle\phi \sqcap \Box\psi, \Delta'}}{\vdots}}{\dfrac{\zeta_1 : \Gamma \Rightarrow \langle q\rangle(\langle r\rangle\phi \sqcap \Box\psi), \Delta}{\zeta : \Gamma \Rightarrow \langle q; r\rangle\phi \sqcap \Box\psi, \Delta}\ {\scriptstyle (\langle;\rangle\sqcap)}}{\dots}$$

According to procedure *Gen_Sch* in Algorithm 2, we can split κ into two parts: $\kappa = \kappa_1\kappa_2$, where κ_1 is the clock sequence generated in procedure *Gen_Sch* from node ζ_1 to node ζ_2, and κ_2 is the clock sequence generated in procedure *Gen_Sch* after node ζ_2. Since κ_1 can be seen as the sequence generated by analyzing the proof tree $\zeta_1(q)$ which ends at node ζ_2 where program q has just been eliminated, and κ_2 can be seen as the sequence generated by analyzing the proof tree $\zeta_2(r)$, by inductive hypothesis there exist $\rho_1 \in Str(q)$ and $\rho_2 \in Str(r)$ s.t. $\kappa(\rho_1) = \kappa_1$, $\kappa(\rho_2) = \kappa_2$. So $\rho_1\rho_2 \in Str(q)Str(r) = Str(q; r) = Str(p)$. Since $\kappa = \kappa_1\kappa_2$, $\kappa(\rho_1\rho_2) = \kappa$. Again by inductive hypothesis there are traces $tr_1 \in val(\rho_1)$, $tr_2 \in val(\rho_2)$ satisfying: 1) under the context Γ, $tr_1 \vDash \Box\psi$ and $tr_{1,e} \in val(\langle r\rangle\phi)$ if $tr_{1,e}$ exists; 2) under the context Γ', $tr_2 \vDash \Box\psi$ and $tr_{2,e} \in val(\phi)$ if $tr_{2,e}$ exists. So it is not hard to see that the trace $tr = tr_1 \circ tr_2 \in val(\rho_1) \circ val(\rho_2) = val(\rho_1\rho_2)$ satisfies that $tr \vDash \Box\psi$ and $tr_e \in val(\phi)$ if tr_e exists.

(ii) $p = q \oplus r$. The proof tree $\zeta(q \oplus r)$ can be in the form shown as follows:

$$\dfrac{\dfrac{\dfrac{\dots}{\zeta_1 : \Gamma \Rightarrow \langle q\rangle\phi \sqcap \Box\psi, \Delta}}{\zeta : \Gamma \Rightarrow \langle q \oplus r\rangle\phi \sqcap \Box\psi, \Delta}\ {\scriptstyle (\oplus, \vee r1)}}{\dots}$$

According to Algorithm 2 κ is actually generated by analyzing the proof tree $\zeta_1(q)$ in procedure *Gen_Sch*. By hypothesis analysis, we know there exists a string $\rho \in Str(q)$ s.t. $\kappa(\rho) = \kappa$, and under the context Γ there is a trace $tr \in val(\rho)$ satisfying that $tr \vDash \Box\psi$ and $tr_e \in val(\phi)$ if tr_e exists. We observe that $\rho \in Str(q) \subseteq Str(q) \cup Str(r) = Str(q \oplus r) = Str(p)$, so the result is directly obtained.

(iii) $p = q^*$. The proof tree $\zeta(q^*)$ is of the form:

$$\dfrac{\dfrac{\dots}{\zeta_1 : \Gamma \Rightarrow \exists x.Inv(x), \Delta} \quad \dfrac{\dfrac{\dfrac{\dots}{\zeta_4 : Inv(t) \Rightarrow \langle q\rangle Inv(t-1) \sqcap \Box\psi}}{\zeta_2 : \cdot \Rightarrow \forall x > 0.(Inv(x) \rightarrow \langle q\rangle Inv(x-1) \sqcap \Box\psi)}\ {\scriptstyle (\forall r, \rightarrow r)} \quad \dfrac{\dots}{\zeta_3 : \cdot \Rightarrow (\exists x \leq 0.Inv(x)) \rightarrow \phi}}{\zeta : \Gamma \Rightarrow \langle q^*\rangle\phi \sqcap \Box\psi, \Delta}\ {\scriptstyle (\langle*\rangle\sqcap)}}{\dots}$$

which ends at node ζ_3 where program q^* has just been eliminated. According to Algorithm 2, $\kappa = \kappa_1^k$ where κ_1 is the clock sequence generated in procedure *Gen_Sch* by analyzing the proof tree ζ_4, k is a witness making $Inv(k)$ valid at node ζ_1. By inductive hypothesis, there exists $\rho_1 \in Str(q)$ s.t. $\kappa(\rho_1) = \kappa_1$ and under the context in which $Inv(t)$ $(t \geq 1)$ holds, there is a trace $tr_t \in val(\rho_1)$ satisfying that $tr_{t,e} \vDash \Box\psi$ and $tr_{t,e} \in val(Inv(t-1))$ if $tr_{t,e}$ exists. So by letting $t = k, k-1, \dots, 1$, we can obtain k such traces tr_k,...,tr_1. Let $\rho = \rho_1^k$, clearly $\rho \in Str(q)^k \subseteq Str(q)^* = Str(q^*) = Str(p)$ and $\kappa(\rho) = \kappa^k(\rho_1) = \kappa_1^k = \kappa$. Let $tr = tr_k \circ tr_{k-1} \circ \dots \circ tr_1$, it is easy to see that $tr \in val^k(\rho_1) = val(\rho_1^4) = val(\rho)$. By the soundness of rule $(\langle*\rangle\sqcap)$, it is not hard to see that under the context Γ, $tr \vDash \Box\psi$ and $tr_e \in val(\phi)$ if tr_e exists.

(iv) $p = q^{\omega}$. The proof tree $\zeta(q^{\omega})$ is of the form:

$$\dfrac{\dfrac{\dots}{\zeta_1 : \Gamma \Rightarrow Inv, \Delta} \quad \dfrac{\zeta_3 : Inv \Rightarrow \langle q \rangle Inv \sqcap \Box\psi}{\zeta_2 : \cdot \Rightarrow Inv \rightarrow \langle q \rangle Inv \sqcap \Box\psi}\ (\rightarrow r)}{\zeta : \Gamma \Rightarrow \langle q^{\omega} \rangle \phi \sqcap \Box\psi, \Delta}\ (\langle\omega\rangle\sqcap)$$

According to Algorithm 2, $\kappa = \kappa_1^{\omega}$ where κ_1 is the clock sequence generated by analyzing the proof tree $\zeta_2(q)$ in procedure *Gen_Sch*. By inductive hypothesis, there exists $\rho_1 \in Str(q)$ s.t. $\kappa(\rho_1) = \kappa_1$ and under the context Inv, there is a trace $tr_1 \in val(\rho_1)$ satisfying that $tr_1 \vDash \Box\psi$ and $tr_{1,e} \in val(Inv)$ if $tr_{1,e}$ exists. Let $\mathcal{A}_{sp}$ be the corresponding cLTS of p_{sp}, since $Str(p_{sp}) = Str(\mathcal{A}_{sp})$, we have $\sigma_0 \in Str(\mathcal{A}_{sp})$. So $\sigma_0 \in Sch(\mathcal{A}_{sp})$, i.e., σ_0 is a schedule of SP. Let $\rho = \rho_1^{\omega}$, clearly we have $\rho \in Str(q)^{\omega} = Str(q^{\omega}) = Str(p)$ and $\kappa(\rho) = \kappa^{\omega}(\rho_1) = \kappa_1^{\omega} = \kappa$. Let $tr = \underbrace{tr_1 \circ tr_1 \circ \dots}_{\infty}$, we have $tr \in \underbrace{val(\rho_1) \circ val(\rho_1) \circ \dots}_{\infty} = val(\rho_1^{\omega}) = val(\rho)$. By the soundness of rule $(\langle\omega\rangle\sqcap)$, it is easy to see that under the context Γ, $tr \vDash \Box\psi$ and $tr_e \in val(\phi)$ if tr_e exists.

Now we focus on the proof tree of the sequent $\cdot \Rightarrow \phi_{SP}$ itself, it is in the form shown as follows:

$$\dfrac{\dfrac{\dots}{\zeta_0 : I \Rightarrow \langle p_{sp} \rangle (tt \sqcap \Box(\psi_{sp} \wedge \psi_{\emptyset}))}}{\cdot \Rightarrow I \rightarrow \langle p_{sp} \rangle (tt \sqcap \Box(\psi_{sp} \wedge \psi_{\emptyset}))}\ (\rightarrow r)$$

σ_0 in fact can be seen as the sequence generated by analyzing the proof tree $\zeta_0(p_{sp})$ in procedure *Gen_Sch* of Algorithm 2, therefore we have:

(1) there exists a string ρ_0 s.t. $\rho_0 \in Str(p_{sp})$ and $\kappa(\rho_0) = \sigma_0$;
(2) under the context I, there exists a trace $tr_0 \in val(\rho_0)$ s.t. $tr_0 \vDash \Box(\psi_{sp} \wedge \psi_{\emptyset})$ and $tr_{0,e} \in val(tt)$ if $tr_{0,e}$ exists.

On the one hand, let $\mathcal{A}_{sp}$ be the corresponding cLTS of p_{sp}, in the proof of Proposition 6.4 we have shown that $Str(\mathcal{A}_{sp}) = Str(p_{sp})$. Since $\rho_0 \in Str(p_{sp})$, $\rho_0 \in Str(\mathcal{A}_{sp})$. By $\kappa(\rho_0) = \sigma_0$ and the fact that ζ_0 is a valid tree, we can get $\sigma_0 \in Sch(\mathcal{A})$. So by Proposition 2.1 there is $\sigma_0 \vDash_{ccsl} \widetilde{Cdf}$.

On the other hand, since $\kappa(\rho_0) = \sigma_0$, $val(\rho_0) \subseteq val(\sigma_0)$. So we have $tr_0 \in val(\sigma_0)$. From the fact that tr_0 is a standard trace in $val(\sigma_0)$ that satisfies (2), in fact we can get that all standard traces of $val(\sigma_0)$ satisfy (2). This is because of the following two reasons: 1) for any standard trace $tr \in val(\sigma_0)$, all clock-related variables in tr can be complete determined by σ_0; 2) according to the construction of ψ_{sp} and $\psi_{\emptyset}$ in Theorem 6.1, $\psi_{sp}, \psi_{\emptyset}$ only contain clock-related variables. By Proposition 4.2 we know all traces in Tr^{σ_0} satisfy (2). By Proposition 6.5, we obtain $\sigma_0 \vDash_{ccsl} \widetilde{Rel}$.

$\sigma_0 \vDash_{ccsl} SP$ since $\sigma_0 \vDash_{ccsl} \widetilde{Cdf}$ and $\sigma_0 \vDash_{ccsl} \widetilde{Rel}$. That is, σ_0 is a schedule of SP. □

Appendix B. Proof of soundness of cDL system

According to the analysis in Sect. 5.3, to prove Theorem 5.1, here we only need to give the proof of the soundness of rules (α), $(\langle\omega\rangle\sqcap)$ and rule $([\omega]\sqcup)$. The soundness of rule $\dfrac{\Gamma_1 \Rightarrow \Delta_1 \quad \dots \quad \Gamma_n \Rightarrow \Delta_n}{\Gamma \Rightarrow \Delta}$ means that if $\vDash_{cdl} \bigwedge_{\phi\in\Gamma_1} \phi \rightarrow \bigvee_{\psi\in\Delta_1} \psi$, ..., $\vDash_{cdl} \bigwedge_{\phi\in\Gamma_n} \phi \rightarrow \bigvee_{\psi\in\Delta_n} \psi$ hold, then $\vDash_{cdl} \bigwedge_{\phi\in\Gamma} \phi \rightarrow \bigvee_{\psi\in\Delta} \psi$ holds.

Proof of the soundness of rule (α)**.** By the definition of the rule of the form: $\dfrac{\Gamma' \Rightarrow \Delta'}{\Gamma \Rightarrow \Delta}$ in Sect. 5.1, we need to prove the following two propositions:

(i) If $\forall s \in S$, $s \vDash_{cdl} (\bigwedge_{\phi\in\Gamma} \phi[V'/V] \wedge P) \rightarrow ((\phi \wedge \psi) \vee \bigvee_{\phi\in\Delta} \phi[V'/V])$, then $\forall s \in S$, $s \vDash_{cdl} \bigwedge_{\phi\in\Gamma} \phi \rightarrow (\langle\alpha\rangle\phi \sqcap \Box\psi \vee \bigvee_{\phi\in\Delta} \phi)$.
(ii) If $\forall s \in S$, $s \vDash_{cdl} \bigwedge_{\phi\in\Gamma} \phi \rightarrow (\langle\alpha\rangle\phi \sqcap \Box\psi \vee \bigvee_{\phi\in\Delta} \phi)$, then $s \vDash_{cdl} (\bigwedge_{\phi\in\Gamma} \phi[V'/V] \wedge P) \rightarrow ((\phi \wedge \psi) \vee \bigvee_{\phi\in\Delta} \phi[V'/V])$.

In rule (α), recall that we have assumed $\alpha = \{c_1, \dots, c_n\}$, $\mathcal{C} - \alpha = \{d_1, \dots, d_m\}$ (see Sect. 5.2). And $V' = (x_1, \dots, x_n, y_1, \dots, y_n, z_1, \dots, z_m)$ is a set of new variables (w.r.t. Γ, $\langle\alpha\rangle\phi \sqcap \Box\psi$, Δ) corresponding to $V = (h(c_1), \dots, h(c_n), \eta(c_1), \dots, \eta(c_n), \eta(d_1), \dots, \eta(d_m))$. $P = \bigwedge_{1\le i\le n} h(c_i) = x_i + 1 \wedge \bigwedge_{1\le i\le n} \eta(c_i) = 1 \wedge \bigwedge_{1\le i\le m} \eta(d_i) = 0$.

For (i), for any $s \in S$, if $s \vDash_{cdl} \bigwedge_{\phi\in\Gamma} \phi$, we show that $s \vDash_{cdl} \langle\alpha\rangle\phi \sqcap \Box\psi \vee \bigvee_{\phi\in\Delta} \phi$. Construct an s' such that

$$\begin{cases} s'(h(c_i)) = s(h(c_i)) + 1, & \text{for any } 1 \le i \le n \\ s'(\eta(c_i)) = 1, & \text{for any } 1 \le i \le n \\ s'(\eta(d_i)) = 0, & \text{for any } 1 \le i \le m \\ s'(z) = s(z), & \text{for each new variable } z' \in V' \\ s'(y) = s(y), & \text{for other variable } y \notin V \end{cases} . \tag{B.1}$$

Since the variables in V' are all new variables w.r.t. the contexts Γ, Δ, we can assume that for any $z' \in V'$, $s(z') = s'(z')$. Because if it is not the case actually we can consider another state s'', whose value differs state s only on those variables in V' (which satisfies $s''(z') = s'(z')$ for any $z' \in V'$). Since $s \models_{cdl} \bigwedge_{\phi\in\Gamma}\phi$, obviously $s'' \models_{cdl} \bigwedge_{\phi\in\Gamma}\phi$. And if we can prove $s'' \models_{cdl} \langle\alpha\rangle\phi \sqcap \Box\psi \vee \bigvee_{\phi\in\Delta}\phi$, then we also have $s \models_{cdl} \langle\alpha\rangle\phi \sqcap \Box\psi \vee \bigvee_{\phi\in\Delta}\phi$.

Since $s \models_{cdl} \bigwedge_{\phi\in\Gamma}\phi$, from (B.1) we get $s' \models_{cdl} \bigwedge_{\phi\in\Gamma}\phi[V'/V] \wedge P$. So $s' \models_{cdl} (\phi\wedge\psi) \vee \bigvee_{\phi\in\Delta}\phi[V'/V]$ from the assumption of (i). If $s' \models_{cdl} \bigvee_{\phi\in\Delta}\phi[V'/V]$, obviously $s \models_{cdl} \bigvee_{\phi\in\Delta}\phi$ by the definition of s' in (B.1). Hence $s \models_{cdl} \langle\alpha\rangle\phi \sqcap \Box\psi \vee \bigvee_{\phi\in\Delta}\phi$. If $s' \models_{cdl} \phi\wedge\psi$, from the construction of s' easy to see that $ss' \in val(\alpha)$. According to the semantics of $\langle\alpha\rangle\phi \sqcap \Box\psi$ (Definition 4.6) there is $s \models_{cdl} \langle\alpha\rangle\phi \sqcap \Box\psi$. Therefore $s \models_{cdl} \langle\alpha\rangle\phi \sqcap \Box\psi \vee \bigvee_{\phi\in\Delta}\phi$.

For (ii), for any $s \in S$, if $s \models_{cdl} \bigwedge_{\phi\in\Gamma}\phi[V'/V] \wedge P$, we need to show $s \models_{cdl} (\phi\wedge\psi) \vee \bigvee_{\phi\in\Delta}\phi[V'/V]$. Construct an s' such that

$$\begin{cases} s'(h(c_i)) = s(x_i), & \text{for any } 1 \le i \le n \\ s'(\eta(c_i)) = s(y_i), & \text{for any } 1 \le i \le n \\ s'(\eta(d_i)) = s(z_i), & \text{for any } 1 \le i \le m \\ s'(z) = s(z), & \text{for each new variable } z' \in V' \\ s'(y) = s(y), & \text{for other variable } y \notin V \end{cases} . \tag{B.2}$$

Since $s \models_{cdl} \bigwedge_{\phi\in\Gamma}\phi[V'/V]$, from (B.2) we get $s' \models_{cdl} \bigwedge_{\phi\in\Gamma}\phi$. So $s' \models_{cdl} \langle\alpha\rangle \sqcap \Box\psi \vee \bigvee_{\phi\in\Delta}\phi$ from the assumption of (*ii*). If $s' \models_{cdl} \bigvee_{\phi\in\Delta}\phi$, obviously $s \models_{cdl} \bigvee_{\phi\in\Delta}\phi[V'/V]$ from the definition of s' in (B.2). If $s' \models_{cdl} \langle\alpha\rangle \sqcap \Box\psi$, since $s \models_{cdl} P$, from the construction of s' easy to see that $s's \in val(\alpha)$. Again, according to the semantics of $\langle\alpha\rangle \sqcap \Box\psi$ we have $s \models_{cdl} \phi\wedge\psi$. Both situations above conclude that $s \models_{cdl} (\phi\wedge\psi) \vee \bigvee_{\phi\in\Delta}\phi[V'/V]$. □

Proof of the soundness of rule ($\langle\omega\rangle\sqcap$)**.** We need to prove that for any Γ, Δ,

$$\text{if } \bigwedge_{\phi\in\Gamma}\phi \to (Inv \vee \bigvee_{\phi\in\Delta}\phi) \text{ and } Inv \to \langle p\rangle Inv \sqcap \Box\psi \text{ hold, then } \bigwedge_{\phi\in\Gamma}\phi \to (\langle p^\omega\rangle\phi \sqcap \Box\psi \vee \bigvee_{\phi\in\Delta}\phi) \text{ holds.} \tag{B.3}$$

Due to the arbitrariness of Γ, Δ, we can assume $\bigwedge_{\phi\in\Gamma}\phi$ holds and $\bigvee_{\phi\in\Delta}\phi$ does not hold. Otherwise the proposition above holds obviously. Thus it equals to prove that

$$\text{if } Inv \text{ and } Inv \to \langle p\rangle Inv \sqcap \Box\psi \text{ hold, then } \langle p^\omega\rangle\phi \sqcap \Box\psi \text{ holds.} \tag{B.4}$$

In fact we only need to prove $\langle p^\omega\rangle\Box\psi$ since all traces of p^ω are infinite (see Definition 4.6). According to the assumption of proposition (B.4), the infinite trace tr of p^ω that satisfies $\Box\psi$ can be constructed as one of the following two forms:

(i) $tr = tr_1 \circ tr_2 \circ \ldots \circ tr_n \circ \ldots$, where each tr_i $(i \ge 1)$ is a finite trace of p, and it satisfies $tr_{i,b} \models_{cdl} Inv$, $tr_{i,e} \models_{cdl} Inv$ and $tr_i \models \Box\psi$.
(ii) $tr = tr_1 \circ tr_2 \circ \ldots \circ tr_m$, where each tr_i $(1 \le i < m)$ is finite, tr_m is infinite. Each tr_i satisfies $tr_i \in val(p)$, $tr_{i,b} \models_{cdl} Inv$, $tr_{i,e} \models_{cdl} Inv$ and $tr_i \models \Box\psi$, while tr_m satisfies $tr_m \in val(p)$, $tr_{m,b} \models_{cdl} Inv$, $tr_m \models \Box\psi$.

Suppose $tr \not\models \Box\psi$, by Definition 4.6 there exists an $i \ge 0$ s.t. $tr(i) \models_{cdl} \neg\psi$. No matter what forms trace tr is in, there must exist an $N \ge 1$ s.t. trace tr_N contains state $tr(i)$. But this contradicts the fact that $tr_N \models \Box\psi$. Therefore $tr \models \Box\psi$, so $\langle p^\omega\rangle\phi \sqcap \Box\psi$ holds. □

The soundness of rule ($[\omega]\sqcup$) can be proved in a similar way as rule ($\langle\omega\rangle\sqcap$).

Proof of the soundness of rule ($[\omega]\sqcup$)**.** Similar to the proof of rule ($\langle\omega\rangle\sqcap$), due to the arbitrariness of Γ and Δ, we only need to prove that

$$\text{if } \exists x.Inv(x), \forall x > 0.(Inv(x) \to [p]Inv(x-1) \sqcup \Diamond\psi) \text{ and } (\exists x \le 0.Inv(x)) \to [p]\Diamond\psi \text{ hold, then } [p^\omega]\phi \sqcup \Diamond\psi \text{ holds.}$$

We only need to prove $[p^\omega]\Diamond\psi$ since all traces of p^ω are infinite. Set k the number that makes $Inv(k)$ hold. If $k \le 0$, then we get $[p]\Diamond\psi$ holds. So all traces of p satisfy $\Diamond\psi$. Since any trace of p^ω is either a trace of p or must have a prefix in p, we have $[p^\omega]\Diamond\psi$ holds. So $[p^\omega]\phi \sqcup \Diamond\psi$ holds. If $k > 0$, we now show that $[p^{k+1}]\Diamond\psi$ holds. Actually, any trace tr of p^{k+1} must be of the form:

(1) $tr = tr_1 \circ tr_2 \circ \ldots \circ tr_{k+1}$, where tr_i $(1 \le i \le k+1)$ is finite. Each tr_i satisfies $tr_i \in val(p)$. $tr_{1,b} \models_{cdl} Inv(k)$. For any $1 \le j \le k$, either $tr_j \models \Diamond\psi$, or $tr_{j,e} \models_{cdl} Inv(k-j)$, $tr_{j+1,b} \models_{cdl} Inv(k-j)$ holds. And $tr_{k+1} \models \Diamond\psi$.
(2) $tr = tr_1 \circ tr_2 \circ \ldots \circ tr_m$, where $1 \le m \le k+1$, $tr_1, \ldots, tr_{m-1}$ is finite, tr_m is infinite. Each tr_i $(1 \le i \le m)$ satisfies $tr_i \in val(p)$. $tr_{1,b} \models_{cdl} Inv(k)$. For any $1 \le j \le m-1$, either $tr_j \models \Diamond\psi$, or $tr_{j,e} \models_{cdl} Inv(k-j)$, $tr_{j+1,b} \models_{cdl} Inv(k-j)$ holds. And $tr_m \models \Diamond\psi$.

From both forms we have $tr \vDash \Diamond\psi$ since either tr_{k+1} or tr_m is a suffix of it. Hence $[p^{k+1}]\Diamond\psi$. Because any trace of p^ω is either a trace of p^{k+1} or must contain a prefix in p^{k+1}, $[p^\omega]\Diamond\psi$ holds. □

References

[1] Y. Zhang, F. Mallet, H. Zhu, Y. Chen, A logical approach for the schedulability analysis of ccsl, in: 2019 International Symposium on Theoretical Aspects of Software Engineering (TASE), 2019, pp. 25–32.
[2] F. Mallet, Clock constraint specification language: specifying clock constraints with UML/MARTE, Innov. Syst. Softw. Eng. 4 (3) (2008) 309–314.
[3] OMG, UML profile for MARTE: modeling and analysis of real-time embedded systems, Tech. Rep., OMG, formal/11-06-02, June 2011.
[4] C. André, Syntax and Semantics of the Clock Constraint Specification Language (CCSL), Research Report RR-6925, INRIA, 2009.
[5] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Commun. ACM 21 (7) (1978) 558–565.
[6] J. Peters, R. Wille, N. Przigoda, U. Kühne, R. Drechsler, A generic representation of CCSL time constraints for UML/MARTE models, in: DAC, ACM, 2015, pp. 122:1–122:6.
[7] E.-Y. Kang, P.-Y. Schobbens, Schedulability analysis support for automotive systems: from requirement to implementation, in: SAC, ACM, 2014, pp. 1080–1085.
[8] H. Yu, J.-P. Talpin, L. Besnard, T. Gautier, H. Marchand, P.L. Guernic, Polychronous controller synthesis from MARTE/CCSL timing specifications, in: MEMOCODE, IEEE, 2011, pp. 21–30.
[9] M. Zhang, F. Mallet, H. Zhu, An SMT-based approach to the formal analysis of MARTE/CCSL, in: ICFEM '16, Springer, 2016, pp. 433–449.
[10] L. Yin, J. Liu, Z. Ding, F. Mallet, R. de Simone, Schedulability analysis with ccsl specifications, in: APSEC (1), IEEE Computer Society, 2013, pp. 414–421, 978-1-4799-2143-0.
[11] M. Zhang, F. Dai, F. Mallet, Periodic scheduling for MARTE/CCSL: theory and practice, Sci. Comput. Program. 154 (2018) 42–60.
[12] M. Zhang, Y. Ying, Towards SMT-based LTL model checking of clock constraint specification language for real-time and embedded systems, in: LCTES '17, ACM, 2017, pp. 61–70.
[13] F. Mallet, J.-V. Millo, R. de Simone, Safe CCSL specifications and marked graphs, in: MEMOCODE, IEEE, 2013, pp. 157–166.
[14] D. Harel, First-Order Dynamic Logic, LNCS, vol. 68, Springer, 1979.
[15] J.-B. Jeannin, A. Platzer, dTL2: differential temporal dynamic logic with nested temporalities for hybrid systems, in: IJCAR, in: Lecture Notes in Computer Science, vol. 8562, Springer, 2014, pp. 292–306.
[16] C. Barrett, P. Fontaine, C. Tinelli, The SMT-LIB Standard: Version 2.6, Tech. Rep, Department of Computer Science, The University of Iowa, 2017, available at www.SMT-LIB.org.
[17] T. Nipkow, L.C. Paulson, M. Wenzel, Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283, Springer, 2002.
[18] Y. Bertot, P. Castéran, Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions, Texts in Theoretical Computer Science, An EATCS Series, Springer, 2004.
[19] Y. Zhang, H. Wu, Y. Chen, F. Mallet, Embedding CCSL into dynamic logic: a logical approach for the verification of CCSL specifications, in: FTSCS 2018, Gold Coast, Australia, 2018.
[20] F. Mallet, R. de Simone, Correctness issues on MARTE/CCSL constraints, Sci. Comput. Program. 106 (2015) 78–92.
[21] F. Mallet, J.-V. Millo, Boundness issues in ccsl specifications, in: Formal Methods and Software Engineering, Springer, Berlin, Heidelberg, 2013, pp. 20–35.
[22] F. Mallet, Automatic generation of observers from MARTE/CCSL, in: RSP, IEEE, 2012, pp. 86–92.
[23] V.R. Pratt, Semantical considerations on Floyd-Hoare logic, in: FOCS, IEEE Computer Society, 1976, pp. 109–121.
[24] D. Harel, D. Kozen, J. Tiuryn, Dynamic logic, SIGACT News 32 (1) (2001) 66–69.
[25] A. Platzer, A temporal dynamic logic for verifying hybrid system invariants, in: LFCS '07, Springer, 2007, pp. 457–471.
[26] W. Thomas, Automata on Infinite Objects, MIT Press, Cambridge, MA, USA, 1991, pp. 133–191.
[27] G. Gentzen, Untersuchungen über das logische Schließen, Ph.D. thesis, NA, Göttingen, 1934.
[28] K. Gödel, Über formal unentscheidbare Sätze der Principia mathematica und verwandter Systeme, Monatshefte Math. Phys. 38 (1) (1931) 173–198.
[29] A. Blass, Y. Gurevich, Inadequacy of computable loop invariants, ACM Trans. Comput. Log. 2 (1) (2001) 1–11.
[30] J.A. Brzozowski, Derivatives of regular expressions, J. ACM 11 (4) (1964) 481–494.
[31] D.N. Arden, Delayed-logic and finite-state machines, in: SWCT (FOCS), IEEE Computer Society, 1961, pp. 133–151.
[32] G.J. Holzmann, The model checker spin, IEEE Trans. Softw. Eng. 23 (5) (1997) 279–295.
[33] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, All About Maude – A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, Lecture Notes in Computer Science, vol. 4350, Springer, 2007.
[34] V.R. Pratt, Semantical consideration on Floyd-Hoare logic, in: 17th Annual Symposium on Foundations of Computer Science (sfcs 1976), 1976, pp. 109–121.
[35] D. Harel, D. Kozen, R. Parikh, Process logic: expressiveness, decidability, completeness, J. Comput. Syst. Sci. 25 (2) (1982) 144–170.
[36] B. Beckert, S. Schlager, A sequent calculus for first-order dynamic logic with trace modalities, in: R. Goré, A. Leitsch, T. Nipkow (Eds.), Automated Reasoning, Springer, Berlin, Heidelberg, 2001, pp. 626–641.