A verification framework for spatio-temporal consistency language with CCSL as a specification language

Yuanrui ZHANG¹, Frédéric MALLET (✉², Yixiang CHEN (✉)¹

1 MoE Engineering Research Center for Software/Hardware Co-design Technology and Application, East China Normal University, Shanghai 200062, China
2 University Nice Sophia Antipolis, I3S, UMR 7271 CNRS, INRIA, 06900 Sophia Antipolis, France

© Higher Education Press and Springer-Verlag GmbH Germany, part of Springer Nature 2018

Abstract The spatio-temporal consistency language (STeC) is a high-level modeling language that deals natively with spatio-temporal behaviour, i.e., behaviour relating to certain locations and time. Such restriction by both locations and time is of first importance for some types of real-time systems. CCSL is a formal specification language based on logical clocks. It is used to describe some crucial safety properties for real-time systems, due to its powerful expressiveness of logical and chronometric time constraints. We consider a novel verification framework combining STeC and CCSL, with the advantages of addressing spatio-temporal consistency of system behaviour and easily expressing some crucial time constraints. We propose a theory combining these two languages and a method verifying CCSL properties in STeC models. We adopt UPPAAL as the model checking tool and give a simple example to illustrate how to carry out verification in our framework.

Keywords spatio-temporal consistency, real-time systems, spatio-temporal systems, high-level modelling language, clock constraint specification, model checking, verification framework

1 Introduction

Real-time systems are widespread nowadays and are quickly evolving. To achieve strong results on the correctness of system, it often requires, from the beginning of system design, modeling formalism to describe systems at an abstract level and corresponding verification techniques to reason about abstract system behaviour. In some sorts of real-time systems like mobile distributed systems or Cyber Physical Systems (CPSs), agents’ behaviour often relate not only to real time, but also to physical or logical locations. For example, in Intelligent Railroad Crossing System (IRCS) (Fig. 1), the “smart” train should inform the “smart” gate that it is coming when it arrives at the location Appr at time t. Such constraint — that an event must be triggered at explicit time and location is called “spatio-temporal consistency” [1]. At high-level of system design, a formalism, where such constraints can be easily expressed, is required to help engineers design more reliable programs when it comes to refinements. Right specification languages are also needed to give an easy expression of some important properties that are of interest.

Fig. 1 Intelligent railroad crossing system

Spatio-temporal consistency language (STeC) [1, 2] is a modeling formalism proposed for describing spatio-temporal behaviour of real-time systems. It is a process algebra-like
modeling language, and looks like an extension of CSP [3] and CCS [4], with location taken as a primitive in its syntax [1]. It aims at modeling system at high level. Not like other process algebras extended with location or time, like timed CSP [5], timed CCS [6], ambient calculus [7], π-calculus [8], etc, STeC addresses a stronger constraint between time and location—spatio-temporal consistency, rather than only considering one of them alone. For example, in timed CCS an event can be triggered at an explicit time, while in ambient calculus an event can be triggered at a specific location (ambience). In STeC, an event can only be triggered at some explicit time and location.

Clock Constraint Specification Language (CCSL) [9] is a specification language based on logical clocks [10]. It was initially proposed as a companion language for the modeling language MARTE [11], and now has been fully developed as an independent language. Compared with traditional specification languages like LTL, CTL, TCTL [12] and PSL [13], logical clock provides an intuitive way to express event sequences and its chronometric attributes. The logical and chronometric constraints between events can be easily expressed by the relationships between logical clocks. Comparing to traditional specification languages, CCSL alleviates the burden for specifying some crucial safety properties, since it provides a library of off-the-shelf often-used property patterns. A comparison of expressiveness between CCSL and PSL is analyzed in [14].

In order to give a verification support for STeC language, and inspired by the MARTE/CCSL framework [15], in this paper we proposed a STeC/CCSL verification framework for modeling and verifying spatio-temporal systems—a special type of real-time systems where agents’ behaviour is related not only to real time, but also to locations. Compared to other frameworks for real-time systems, STeC/CCSL framework focuses on capturing system behaviour at high level, with the advantage of addressing the spatio-temporal consistency of system behaviour at the syntax level. And it supports specifying and verifying logical and chronometrical timing constraints (and possibly some carefully selected spatial constraints in the future) in CCSL style, which could be complex or difficult to be expressed by other traditional specification languages like LTL, CTL.

In STeC/CCSL framework, in order to connect STeC and CCSL, we propose a linking theory and a model checking framework to verify CCSL specifications. We formalize some concepts in CCSL and build an observational mapping between CCSL clocks and STeC events. To carry out model checking, we propose theory and algorithms to translate STeC and CCSL into their equivalent Timed Automata (TA).

We make a general analysis for the computation complexity of the whole verification process, and show that our framework runs ‘nearly as fast as’ the traditional model checking process, despite the translation process. At last, in order to show that our proposed verification framework is applicable, we propose the translation into UPPAAL [16] TA, a model checking tool widely used in academia and industry.

Technically, our work can be seen as a combination and extension of [17, 18], where the verification aspect of CCSL and STeC has been explored separately. In [18], we introduced TA semantics of STeC. The translation from STeC into TA was given inductively based on the syntax structure of STeC. In this paper we take a more general approach by dividing STeC configurations into “regions”. In [17], a translation strategy was proposed to encode CCSL into TA and two examples were given to illustrate the model checking scheme in UPPAAL. There the TA of CCSL is based on untimed synchronous model cLTS [19], which is not suitable for our framework since STeC is an asynchronus language. We make some improvements by translating it into asynchronous TA models. And we give a full translation from CCSL into UPPAAL TA. Our proposed model checking framework is partially based on the idea of the verification scheme in [17], but is quite different as we consider the combination with STeC and the CCSL generators.

This paper is organized as follows. Section 2 introduces some backgrounds about STeC, CCSL and TA. In Section 3, we introduce a spatio-temporal system—intelligent railroad crossing system (IRCS) as an example and model it using STeC. In Section 4 we present the main contribution — we propose the linking theory that connects STeC and CCSL and the model checking framework. In Section 5, we carry out model checking in UPPAAL, based on the translation into TA. In Section 6 we verify three IRCS safety properties in UPPAAL as an example. Section 7 concludes our work and discusses possible future works. Section 8 compares with some similar verification frameworks proposed in recent years.

2 Preliminaries

Given any two sets A, B, \( A - B = \{a \mid a \in A \land a \notin B\} \) is the difference set of A and B. \( A \cup B = \{(a, i) \mid (a \in A \land i = 0) \lor (a \in B \land i = 1)\} \) is the disjoint union of A and B. \( \mathbb{N}^+ \) denotes the set of non-negative real numbers, \( \mathbb{N}^+ \) denotes the set of natural numbers (excluding 0).
Given a function \( f : A \rightarrow B \), \( \text{dom}(f) = \{a \mid a \in A \land f(a) \in B\} \) denotes its domain, and \( \text{cod}(f) = \{b \mid b \in B \land \exists a. (f(a) = b)\} \) denotes its codomain. A function is partial iff \( \text{dom}(f) \subset A \). A function \( f : \mathbb{N}^+ \rightarrow B \) is down-closed iff \( i \in \text{dom}(f) \) implies \( \forall j < i. (j \in \text{dom}(f)) \). A function is finite iff \( \text{dom}(f) \) is a finite set.

A partial order relation \( \leq A \times A \) is a reflexive, antisymmetric and transitive relation defined on \( A \). A strict partial order relation \( \ll A \times A \) is transitive, but neither reflexive or antisymmetric. A total (strict) partial order \( \leq (<) \) is a (strict) partial order such that for all \( x, y \in A \), \( x \leq y \) \((<) \) or \( y \leq x \) \((<) \) holds.

In the remaining of this section, we introduce the basic concepts of STeC, CCSL and TA.

2.1 Introduction of STeC

We restate the syntax and operational semantics of STeC in this section. The version of STeC we give is based on [1], but differs in some details. The main difference is that we distinguish two types of parallel compositions between processes: the interleaving \( \| \) between processes in agents, and the concurrency \( \triangledown \triangledown \) between agents. In operational semantics, we split original STeC transitions into two kinds: time-only transitions and process transitions. This would ease the way to translate STeC into TA (Section 4.6.2). The expressiveness of this version remains the same as in [1].

2.1.1 Syntax

An agent \( Ag \) in STeC is defined based on the following rules in Backus-Naur form:

\[
A :: \text{Send}_{\text{Ag}}^C(i, m) \mid \text{Get}_{\text{Ag}}^C(i, m), \\
B :: \alpha_{\text{Ag}}(l, \delta) \mid \beta_{\text{Ag}}(\delta), \\
AT :: E \mid \text{Stop}_{\text{Ag}}(l) \mid \text{Skip}_{\text{Ag}}(l) \mid A \mid B, \\
Ag :: AT \mid Ag; Ag \mid Ag\| Ag \mid Ag \parallel Ag \mid Ag \geq_s Ag \mid Ag \geq (\|_{\text{st}} Ag_i \rightarrow Ag_i).
\]

A system \( P \) can be an agent, or the composition of several agents running in parallel:

\[
P :: = Ag \mid P \Rightarrow P.
\]

In STeC, symbol \( P, Q, R \) range over STeC processes. The set of event names \( \text{Alp} \) is ranged over by \( a, b, c \). The set of location names \( \text{Loc} \) is ranged over by \( l \). The set of message names \( \text{Msg} \) is ranged over by \( m \). Symbol \( A_{\text{STeC}}, B_{\text{STeC}}, AT_{\text{STeC}} \) denote the set of processes of type \( A, B \) and \( AT \) respectively, and symbol \( P_{\text{STeC}} \) denotes the set of all processes. Symbol \( t \) and \( \delta \) denote the time that ranges over \( \mathbb{R}^+ \).

We now give an intuitive explanation of each STeC sentence.

In atomic processes \( AT \), processes of type \( A \) are communicating channels between agents. \( \text{Send}_{\text{Ag}}^C(i, m) \) means that an agent sends message \( m \) through channel \( C \). \( \text{Get}_{\text{Ag}}^C(i, m) \) means that an agent receives message \( m \) through channel \( C \). \( \alpha_{\text{Ag}}(l, \delta) \) means that an event starts at location \( l \) and time \( t \), and consumes \( \delta \) time to terminate. By executing it, the agent moves from location \( l \) to \( l' \). \( \beta_{\text{Ag}}(\delta) \) means that an event starts at location \( l \) and time \( t \), and takes \( \delta \) time to terminate. During its execution the agent stays in \( l \). \( \text{Stop}_{\text{Ag}}(l) \) means that an agent at \( (l, t) \) does not terminate, it does nothing but consumes time. \( \text{Skip}_{\text{Ag}}(l) \) means that the program terminates successfully. \( E \) indicates the ending of a process, it does nothing and does not consume time.

In an agent \( Ag \), \( ; \) is the sequence operator, \( P; Q \) means that the process first behaves like \( P \), then behaves like \( Q \). \( \| \) is the nondeterministic choice, \( P\| Q \) means that either \( P \) proceeds or \( Q \). \( \triangledown \triangledown \) is the interleave between processes in an agent. \( P \triangledown Q \) means that agent \( P \) and \( Q \) proceed in parallel without communications. \( \geq_s \) is the interrupt operator. \( P \geq_s Q \) behaves as \( P \) for up to \( \delta \) time and then is interrupted by \( Q \). \( P \geq (\|_{\text{st}} A_i \rightarrow P_i) \) initially behaves as \( P \) and is interrupted by event \( A_i \in A_{\text{STeC}} \) and then behaves like \( P_i \). \( I \) is a finite index set. If several \( A_i \) occur simultaneously, the choice \( \triangledown \triangledown \) turns out to be a nondeterministic choice. \( P \Rightarrow Q \) means that agent \( P \) and \( Q \) are running in parallel with communications.

2.1.2 Operational semantics

An environment in STeC is a pair \( (l, t) \). \( E \) denotes the set of environments, \( E \subseteq \text{Loc} \times \mathbb{R}^+ \). A configuration of process \( P \) is a triple \( (P, l, t) \), where \( (l, t) \in E \). We use \( \text{Stconf} \) to denote the set of configurations. Transition relation \( \rightarrow \subseteq \text{Stconf} \times \text{Stconf} \) is defined as: \( (P, l, t) \rightarrow (P', l, t') \) iff \( ((P', l, t'), (P, l, t)) \in \rightarrow \).

There are two types of transitions in the operational semantics of STeC: time-only transitions and process transitions. During time-only transitions, the process remains unchanged while time progresses. During process transitions, the process emits an event instantly. The transition rules for atomic processes are given in Table 1. Time-only transitions are denoted as \( \Rightarrow \). Symbol \( e \notin \text{Alp} \) means ‘there is no events triggered during transitions’. Process transitions are denoted by \( \Rightarrow_b \), with \( b \in \text{Alp} \). We follow the convention in automata theory that using \( C^\prime m/Cm \) to express input/output communication events. Note that there is no transition rules for process \( E \), because \( E \) indicates the ending of a process. So semantically, we can take any processes for example \( E; P, E \| E, \)
which do not proceed as the equivalent processes of
process \( E \), denoted by \( E \parallel E = E \parallel E = \cdots = E \).

We define \( \to^* \) as the transitive closure of \( \to \). For any
\( A \subseteq \text{Alp} \), \( (P, l, t) \xrightarrow{\Delta E} (P_n, l_n, t_n) \) means \( (P, l, t) \xrightarrow{\mu_1} (P_1, l_1, t_1) \)
\( \xrightarrow{\mu_2} \cdots \xrightarrow{\mu_n} (P_n, l_n, t_n) \) where \( n \geq 0 \), \( \mu_i \in \text{Alp} \cup \{e\} \),
\( A = \{ a \mid a \in \{ \mu_0, \mu_1, \ldots, \mu_n \} - \{e\} \} \). We write \( (P, l, t) \xrightarrow{\mu_1} \cdots \xrightarrow{\mu_n} (P', l', t') \) to mean that there exists a
\( \langle P''', l'', t'' \rangle \) such that \( \langle P', l', t' \rangle \)

Table 1 Transition rules for atomic processes

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>( \text{Send}^c_{l}(m), l \xrightarrow{\mu} (E, l) )</td>
</tr>
<tr>
<td>2.</td>
<td>( \text{Get}^c_{l}(m), l \xrightarrow{\mu} (E, l) )</td>
</tr>
<tr>
<td>3.</td>
<td>( \nu \in \mathbb{R}^+, s \geq s + d )</td>
</tr>
<tr>
<td>5.</td>
<td>( \text{Stop}<em>{l}(s), s \xrightarrow{\mu} \text{Stop}</em>{l}(s + d) )</td>
</tr>
<tr>
<td>6.</td>
<td>( \text{Skip}_{l}, l \xrightarrow{\mu} (E, l) )</td>
</tr>
</tbody>
</table>

Table 2 lists the transition rules for compositional processes, where \( \mu \in \text{Alp} \cup \{ e \} \). Rule 2 indicates that the choice operator \( \parallel \) in STeC is an internal choice. In Rule 3, \( \langle l_1, l_2 \rangle \) is

Table 2 Transition rules for compositional processes

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>( (P_1, \mu) \xrightarrow{\mu} (P''', l'', t'') )</td>
</tr>
<tr>
<td>2.</td>
<td>( (P_1, l_1, t_1) \xrightarrow{\mu} (P''', l'', t'') )</td>
</tr>
<tr>
<td>3.</td>
<td>( (P_1, l_1, t_1) \xrightarrow{\mu} (P''', l'', t'') )</td>
</tr>
<tr>
<td>4.</td>
<td>( (P_1, l_1, t_1) \xrightarrow{\mu} (P''', l'', t'') )</td>
</tr>
<tr>
<td>5.</td>
<td>( (P_1, l_1, t_1) \xrightarrow{\mu} (P''', l'', t'') )</td>
</tr>
<tr>
<td>6.</td>
<td>( (P_1, l_1, t_1) \xrightarrow{\mu} (P''', l'', t'') )</td>
</tr>
</tbody>
</table>

Table 3 Translation for primitive constraints

<table>
<thead>
<tr>
<th>spec</th>
<th>TA</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c_1 \subseteq c_b )</td>
<td>Start ( 0 )</td>
</tr>
<tr>
<td>( c_2 \subseteq c_b )</td>
<td>Start ( 0 )</td>
</tr>
<tr>
<td>( c_3 \subseteq c_b )</td>
<td>Start ( 0 )</td>
</tr>
</tbody>
</table>

2.2 Introduction of CCSL

In this section we restate the basic definitions of clock, clock
constraint, and clock specification. The version we propose
is based on [9,17,19–21], and the definitions follow the style
of [9]. We distinguish the definition of “Delay On” for dis-
crete and dense clocks, since we adopt different translating
approaches for two different types of clocks (see Table 3).

2.2.1 Clock

A CCSL clock consists of an ordered sequence of ticks (also
called “instants”). Each instant can be assigned to a name by
a labelling function. Formally, a clock is a tuple \( \langle I, <, D, \lambda \rangle \), where

1) \( I \) is a set of instants (possibly infinite).
2) \(<\) is a strict total partial order relation on \( I \).
3) \( D \) is a set of labels.
4) \( \lambda : I \rightarrow D \) is a labelling function, assigning each instant of a clock with a name.

An ideal physical clock \( \text{IdealClk} = \langle I_0, <_0, D_0, \lambda_0 \rangle \) is the only dense clock discussed in this paper. \( \lambda_0 : I_0 \rightarrow D_0 \) is a bijection and \( D_0 = \mathbb{R}^+ \).

In a discrete-time clock \( I \) can be indexed by natural numbers in a way that respects the ordering on \( I \). A function \( idx : I \rightarrow \mathbb{N}^* \) is defined as: \( \forall i \in I, idx(i) = k \) iff \( i \) is the \( k^{th} \) instant in \( I \). Let \( i_0 \) be the first element in \( I \), i.e., for any \( x \in I \), there is \( i_0 < x \lor x = i_0 \). For a discrete clock, \( P(i) \) (with \( i \) not the first element in \( I \)) is the unique immediate predecessor of \( i \). We have: \( idx(P(i)) = idx(i) - 1 \). \( S(i) \) is the unique immediate successor of \( i \). We have: \( idx(S(i)) = idx(i) + 1 \).

Let \( c = \langle I, <, D, \lambda \rangle \) be a CCSL clock. \( c[k] \) denotes the \( k^{th} \) element in \( I \), we have \( k = idx(c[k]) \). Without special mentioning all clocks appearing in this paper are discrete clocks.

For convenience, we define projections of items for a clock \( c = \langle I, <, D, \lambda \rangle \): \( \pi_1(c) = I \), \( \pi_2(c) = < \), \( \pi_3(c) = D \), and \( \pi_4(c) = \lambda \).

### 2.2.2 Primitive clock constraints

There are different versions of constraints given in [9, 17, 19–21]. We only introduce a subset of clock constraints which are relevant to the time constraints discussed in this paper. Still, our theory proposed in this paper can be easily extended to the rest of them.

We first define “time structure” where instants from different clocks can be compared [9]. A time structure is a tuple \( \langle I, <, \equiv, H \rangle \), where \( I \) is a set of instants of all clocks, \(<\) is a strictly partial order relation between instants, and \( \equiv \) is a co-incidence relation, indicating the simultaneous occurrence of instants in different clocks, \( H \) is a set of constraint mappings in \( I \) (defined explicitly later). From \(<\) we derive a partial order relation \( \leq \equiv \subset \cup \equiv \), it means that \( a \leq b \iff a < b \lor a \equiv b \).

Let \( C \) be a set of clocks, a time structure \( \langle I, <, \equiv, H \rangle \) (over \( C \)) is well-structured if it satisfies the following properties:

1) \(<\) is a partial order relation in \( TS \).
2) \( I = \bigcup_{c \in C} I_c \), i.e., \( I \) contains instants of all clocks in \( C \).
3) \( \forall c_a = \langle I_a, <_a, D_a, \lambda_a \rangle \in C \), \( \forall i, j \in I_a \), if \( i <_a j \), then \( i < j \), i.e., partial order \(<\) must be consistent with \(<_a\) in each clock \( c_a \in C \).
4) \( \forall c_a = \langle I_a, <_a, D_a, \lambda_a \rangle \), \( c_b = \langle I_b, <_b, D_b, \lambda_b \rangle \in C \), \( \forall i_a, j_a \in I_a, i_b < j_b \in I_b \), if \( i_a \equiv i_b \land j_a \equiv j_b \land i_a <_a j_a \), then \( i_b <_b j_b \), i.e., \( \equiv \) preserves the order in each clock.

Clock constraints is derived by defining different sets of well-structured time structures. Let \( c_a = \langle I_a, <_a, D_a, \lambda_a \rangle \), \( c_b = \langle I_b, <_b, D_b, \lambda_b \rangle \), \( c_d = \langle I_d, <_d, D_d, \lambda_d \rangle \) be any three clocks. The definitions of clock constraints are given as follows:

Sub clock — constraint \( Cn := c_a \sqsubseteq c_b \) describes that clock \( c_a \) can tick only if clock \( c_b \) ticks. Formally it is defined by a (possibly infinite) set of time structures \( TS = \{TS_i\}_{i \in I} \) (I is a countable index set). Each \( TS_i = \langle I_i, <_i, \equiv, \{h_i\} \rangle \) is well structured and \( I_i = I_a \sqcup I_b \). The constraint mapping \( h_i : I_a \rightarrow I_b \) satisfies: 1) \( h_i \) is injective. 2) \( \forall x \in I_a, (x \equiv h(x)) \).

Figure 2 shows a time structure for the constraint \( c_a \sqsubseteq c_b \), where only finite number of ticks of clocks are drawn (the same for all figures below). Each dot lying on a long arrow indicates a tick of corresponding clock. The long arrow itself means the order \( (<_a, <_b) \) defined in each clock. The dashed arrow indicates the mapping \( h \) and the dash line represents the “inter-clock” relation \( \equiv \), which means that two ticks occur simultaneously.

Note that usually there is more than one time structures for a given constraint, see Fig. 3 for example as another time structure for \( c_a \sqsubseteq c_b \) (with two more ticks of \( c_b \) ahead of \( c_a \)). Each time structure indicates a possible behaviour of how clocks should tick to satisfy the constraint.

### Fig. 2 A time structure of \( c_a \sqsubseteq c_b \)

### Fig. 3 Another time structure of \( c_a \sqsubseteq c_b \)

Strict precedence — the constraint \( Cn := c_a \prec c_b \) describes that clock \( c_a \) always ticks before clock \( c_b \). It is defined over a set of well-structured time structures \( TS = \{TS_i\}_{i \in I} \). In each \( TS_i = \langle I_i, <_i, \equiv, \{h_i\} \rangle \) where \( I_i = I_a \sqcup I_b \), \( h_i : I_b \rightarrow I_a \) satisfies:

1) \( h_i \) is injective.
2) \( \forall x, y \in I_b, (x <_b y \Rightarrow h(x) <_a h(y)) \).
3) \( \forall x \in I_b, (h(x) <_i x) \).

Figure 4 gives a time structure of \( c_a < c_b \).

Precedence — the constraint \( Cn := c_a \leq c_b \) describes that clock \( c_a \) always ticks ‘not-after’ clock \( c_b \). In each \( TS_i = (I_i, <_i, \equiv, [h_i]) \in TS \) where \( I_i = I_a \cup I_b \), \( h_i : I_b \rightarrow I_a \) satisfies:

1) \( h_i \) is injective.
2) \( \forall x, y \in I_b, (x <_b y \Rightarrow h(x) <_a h(y)) \).
3) \( \forall x \in I_b, (h(x) <_i x) \).

Figure 5 gives a time structure of \( c_a \leq c_b \), where the dash-and-dot line indicates the \( \equiv \) relations between clocks.

\[
\begin{align*}
\text{Figure 5} \quad &\text{A time structure of } c_a \leq c_b \\
\end{align*}
\]

Alternation — the constraint \( Cn := c_a \text{ alternatesWith } c_b \) describes that two clocks ticks alternately. It is defined over \( TS \). In each \( TS_i = (I_i, <_i, \equiv, [h_i]) \in TS \) where \( I_i = I_a \cup I_b \), \( h_i : I_a \rightarrow I_b \) satisfies:

1) \( h_i \) is a bijection.
2) \( \forall x \in I_a, (x <_i h(x) \land h(x) <_i S(x)) \).
3) \( \forall x, y \in I_a, (x <_a y \Rightarrow h(x) <_b h(y)) \).

Figure 6 gives an example of the time structure.

\[
\begin{align*}
\text{Figure 6} \quad &\text{A time structure of } c_a \text{ alternateWith } c_b \\
\end{align*}
\]

Delay on — the constraint \( Cn := c_b = c_a \text{ delay } n \text{ on } c_d \), where \( n \in \mathbb{N}, c_d \) is a discrete clock, means that clock \( c_b \) is delayed \( n \) ticks of clock \( c_d \) for \( c_a \). In other words, if \( c_d \) ticks, then \( c_b \) ticks synchronously with the \( n \)th following ticks of \( c_d \). Formally, in each \( TS_i = (I_i, <_i, \equiv, [h_i^1, h_i^2]) \in TS \) where \( I_i = I_a \cup I_b \cup I_d \), \( h_i^1 : I_b \rightarrow I_a \) and \( h_i^2 : I_b \rightarrow I_d \) satisfy:

1) \( h_i^1, h_i^2 \) are both injective.
2) \( \forall x \in I_b, (x \equiv_1 h_i^2(x) \land h_i^1(x) \leq_i P^{n-1}(h_i^2(x)) \land P^n(h_i^2(x)) <_i h_i^1(x)) \).

Figure 7 gives an example of the time structure, where \( n \) indicates that there are \( n \) ticks between those two ticks (including them).

\[
\begin{align*}
\text{Figure 7} \quad &\text{A time structure of } c_b = c_a \text{ delay } n \text{ on } c_d \\
\end{align*}
\]

Delay On (for Dense Clock) — the constraint \( Cn := c_b = c_a \text{ delay } r \text{ on } \text{IdealClk} \) where \( r \in \mathbb{R}^+ \) is defined over \( TS \). In each \( TS_i = (I_i, <_i, \equiv, [h_i^1, h_i^2]) \in TS \) where \( I_i = I_a \cup I_b \cup I_0 \), \( h_i^1 : I_b \rightarrow I_a \) and \( h_i^2 : I_b \rightarrow I_0 \) satisfies:

1) \( h_i^1, h_i^2 \) are both injective.
2) \( \forall x \in I_b, \exists k \in I_0, (x \equiv_1 h_i^2(x) \land h_i^1(x) \equiv_1 k \wedge \lambda_0(h_i^2(x)) = \lambda_0(k) = r) \).

Figure 8 gives an example of the time structure, where we use a short line to indicate a tick of \( \text{IdealClk} \) to stress that it is a dense clock (i.e., it ticks everywhere on the long arrow \( <_0 \)).

\[
\begin{align*}
\text{Figure 8} \quad &\text{A time structure of } c_b = c_a \text{ delay } r \text{ on } \text{IdealClk} \\
\end{align*}
\]

In this paper we use \( Cn \) to range over all clock constraints. We often write \( TS_{Cn} \) as the corresponding set of time structures of a given constraint \( Cn \). We use \( C_{CCSL} \) to denote the set of all clock constraints.

2.2.3 CCSL specification, safe specification

A CCSL specification \( \text{spec} \subseteq C_{CCSL} \) is a finite set of constraints. We use \( \text{spec} \) to range over all specifications. Sometimes we also call a specification “a set of constraints” or just “constraints” instead.

A CCSL specification can be equivalently understood as a type of labelled transition system (LTS) [19]. Some of primitive constraints, called unsafe constraints, correspond to LTSs with infinite states (e.g., \( c_a < c_b \) in Table 3). Such LTSs are
not allowed when we analyze the model checking aspects of CCSSL. However, a specification containing one or several unsafe constraints does not have to be unsafe. An example is the specification of Property 1 in Section 6. If a specification corresponds to an LTS with finite states, we call it a safe specification.

When dealing with model checking aspects of CCSSL, we always consider safe specifications. Mallet et al. [22] introduces an efficient algorithm to check whether a specification is safe or not using directed graph.

2.3 Introduction to timed automata [23]

2.3.1 Definition

Let $C$ be a finite set of non-negative real-valued variables—clocks. Guard expression is defined by the grammar $g := \text{true} | \text{false} | c \star n | g \land g$ where $c \in C$, $n \in \mathbb{N} (\mathbb{N} = \mathbb{N}^+ \cup \{0\})$ and $\star \in \{<,\leq,>,\geq,=\}$. $G(C)$ denotes the set of guards related to $C$. A timed automaton is a tuple $A = (Q, \Sigma, C, i, Ed, I, AP, L, F)$, where:

1. $Q$ is a set of states.
2. $\Sigma$ is a set alphabets.
3. $C$ is a finite set of clocks.
4. $i \in Q$ is the initial location.
5. $Ed \subseteq Q \times \Sigma \times G(C) \times 2^C \times Q$ is a transition.
6. $I : Q \rightarrow G(C)$ is an invariant-assignment function.
7. $AP$ is a set of atomic propositions.
8. $L : Q \rightarrow AP$ is a labeling function.
9. $F \subseteq Q$ is a set of accepting states.

2.3.2 States and transitions

A configuration in TA is a pair $\langle q, v \rangle$ where $q \in Q$ is a state and $v : C \rightarrow \mathbb{R}^+$ is a valuation of clocks. For $r \in \mathbb{R}^+$, we define $v + r$ as: for each clock $c \in C$, $(v + r)(x) = v(x) + r$.

If $Y \subseteq C$ then a valuation $v[Y := 0]$ is such that for each clock $x \in C - Y$, $v[Y := 0](x) = v(x)$ and for each clock $x \in Y$, $v[Y := 0](x) = 0$. A satisfaction relation $v \models g$ holds iff the valuation $v$ makes guard $g$ true. Given a TA $A = (Q, \Sigma, C, i, Ed, I, AP, L, F)$, the transition function between states is a relation $\rightarrow_{\text{TA}} \subseteq (Q \times \mathbb{R}_+^C) \times (Q \times \mathbb{R}_+^C)$, defined as:

$$\rightarrow_{\text{TA}} (q, v) = \begin{cases} (q, v + a), & \text{if } \forall \beta \leq \alpha \in \mathbb{R}^+, v + \beta \models I(q), \\ (q', v'), & \text{if } \langle q, a, g, Y, v' \rangle \in Ed, a \in \Sigma, \\ & v \models g, v' = v[Y := 0]. \end{cases}$$

The first transition above is the time-only transition, we write it as $\langle q, v \rangle \rightarrow_{\text{TA}} (q, v + \alpha)$. The second transition is written as $\langle q, v \rangle \rightarrow_{\text{TA}} (q', v')$, $\rightarrow_{\text{TA}}^*$ is the transitive closure of $\rightarrow_{\text{TA}}$, defined as: $\langle q, v \rangle \rightarrow_{\text{TA}} (q_n, v_n)$ if $\langle q, v \rangle \rightarrow_{\text{TA}}^m \langle q_1, v_1 \rangle \rightarrow_{\text{TA}}^n \langle q_2, v_2 \rangle \rightarrow_{\text{TA}}^\star \cdots \rightarrow_{\text{TA}}^\star \langle q_{n-1}, v_{n-1} \rangle \rightarrow_{\text{TA}}^\star \langle q_n, v_n \rangle$ where $n > 0$, $\mu_i \in \Sigma \cup \{\varepsilon\}$, $i = 1, 2, \ldots, n$, $A = \{a | a \in [\mu_1, \mu_2, \ldots, \mu_n] - \{\varepsilon\}\}$. $\langle q, v \rangle \rightarrow_{\text{TA}}^\star (q', v')$ means that there exists a $(q'', v'')$ such that $\langle q, v \rangle \rightarrow_{\text{TA}}^\star (q'', v'') \rightarrow_{\text{TA}} (q', v')$.

In a TA $A$, the initial configuration, denoted as $i_{\text{TA}} = \langle i, v'_A \rangle$ where $i$ is the initial state, satisfies $\emptyset \models I(q, v). (q, v) \rightarrow_{\text{TA}}^\star (i, v'_A)$. The set of all its configurations of $A$, denoted as $\text{Conf}_{\text{TA}}(A)$, is given by $\text{Conf}_{\text{TA}}(A) = \{(q, v) | i_{\text{TA}} \rightarrow_{\text{TA}}^\star (q, v)\}$.

2.3.3 TA traces

A time word is a pair $\langle a, t \rangle$ where $a \in \Sigma$ and $t \in \mathbb{R}^+$. We use $\mathbb{E}_{\text{TA}}$ to denote the set of all time words. A finite TA trace $tr : \mathbb{N}^+ \rightarrow \mathbb{E}_{\text{TA}}$ is a partial, down-closed, finite function. A trace $tr = \langle (a_1, t_1) \langle a_2, t_2 \cdots (a_n, t_n) \rangle$ is in $\mathbb{E}_{\text{TA}}$ iff there are transitions $(i, v'_A) \rightarrow_{\text{TA}}^\star (q_1, v_1) \rightarrow_{\text{TA}}^\star \cdots \rightarrow_{\text{TA}}^\star (q_n, v_n)$ in $A$ such that $q_n \in F$. $t_i$ here indicates the elapsing time of transition $\mu_i$ from the beginning (where the time equals 0).

The set of all finite traces of $A$ is denoted by $\text{Traces}_{\text{TA}}(A)$.

3 Intelligent railroad crossing system

In this section we analyze a simple spatio-temporal system—IRCS and model it using STeC. In IRCS (Fig. 1) there are two agents: a “smart” train and a “smart” gate. The “smart” gate is located at the crossing where a railroad lies in east-west direction and a road lies in south-north direction. The “smart” train communicates with the “smart” gate dynamically when it tries to pass the crossing. Each “Lapp”, “Lpass”, “Lstop”, “Lleave” on the track indicates the logical locations in which only proper behaviour can be performed by agent train at the right time. The states of agent gate (“Lclose”, “Lopen”) are also considered as logical locations, in which only proper behaviour can be performed by the gate at the right time.

The scenario of IRCS is as follows:

Train—

1. The train approaches the crossing at location “Lapp”, taking 20s, and arrives at location “Lpass”, during which it sends message “Appr” to the gate.

2. At “Lpass” the train waits for gate’s messages, if it receives message “NonCross”, then stops after 30s and reaches location “Lstop”. If it receives “Cross”, it keeps running, passing the crossing within 60s, and finally
reaches “Lleave”.
3) At “Lstop” the train waits for message “Cross”, after receiving it, it restarts and passes the crossing.
4) After passing the crossing, the train sends “Leave” to the gate at location “Lleave”.

Gate—
1) At the beginning, the gate is assumed to be open, waiting for messages from the train.
2) After the gate receives message “Appr” from the train, it tries to close itself in 10s.
3) The gate spends 1s for judging if itself is successfully closed. If so, the gate sends message “Cross”. If not, the gate sends message “NonCross”.
4) After receiving message “Leave” from the train, the gate opens itself for 10s.

Agent train can be described by STeC as follows:

\[
T = \text{Send}_{Lapp,i}(Appr); \text{Approach}_{Lapp,i}(Lpass, 20); \text{Send}_{Lapp,i}(Lpass, 20) \\
\{ \\
\text{Get}_{Lpass,i+20}(Cross) \rightarrow \text{Pass}_{Lpass,i+20}(Lleave, 60); \\
\text{Send}_{Lpass,i+20}(Lleave, 50); \text{Skip}_{Lleave,i+50}(Lleave, 60); \text{Send}_{Lleave,i+110}(Lleave); \\
\text{Get}_{Lpass,i+20}(NonCross) \rightarrow (\text{Stop}_{Lpass,i+20}(Lstop, 30); \text{Wait}_{Lpass,i+20}(Cross)); \\
\text{Pass}_{Lstop,i+50}(Lleave, 60); \text{Send}_{Lleave,i+110}(Lleave); \\
\text{Skip}_{Lleave,i+110}(Lleave, 60); \\
\}
\]

where Wait is a process, defined as:

\[
\text{Wait}_{Lapp,i}(m) = \text{Stop}_{Lapp,i}(m) \sqsubseteq (\text{Get}_{Lapp,i}(m) \rightarrow \text{Skip}_{Lapp,i}(m)).
\]

It describes an agent waits for a message \( m \) at \((l, t)\) until it receives the message at location \( l' \) and time \( t' \).

Agent gate can be described by STeC as follows:

\[
G = \text{Wait}_{Lopen,i}(Appr); \text{Closing}_{Lopen,i}(10); \\
\{ \\
\text{Closed}_{Lclose,i+10}(1); \text{Stop}_{Lclose,i+11}; \\
\text{Send}_{Lopen,i+20}(Cross); \text{Wait}_{Lopen,i+20,Lclose,i+10}(Lleave); \\
\text{Open}_{Lopen,i+10}(Lopen, 10); \text{Skip}_{Lopen,i+90}(Lopen, 10); \\
\}
\]

By composing them together by parallel operator we have:

\[
P_{IRCS} = T \Rightarrow G.
\]

Initially, we set \( t = 300 \) to be the starting point of this system and set \( t' = 0 \).

## 4 The verification of STeC/CCSL framework

This section is the main contribution of this paper. We build the STeC/CCSL framework by proposing a theory linking them and a model checking scheme for verifying CCSL properties in STeC models. Since CCSL constraint is not a CTL-like language and its semantics does not support tree-like searching mechanism, adding that we have to consider—special clock (introduced below) as a companion in model checking STeC processes, we can not directly apply the conventional model checking algorithm [12] in our purpose. Instead, we propose an alternative approach by taking a CCSL specification as an observer for STeC model, rather than an assertion. In this way, as we will see soon, essentially the satisfaction of a CCSL specification can be transformed into a classical model checking problem with a simple CTL formula to be checked.

In order to reason about STeC and CCSL models, we first introduce the notions of STeC traces and traces in CCSL specifications (Sections 4.1–4.3). Then in Section 4.4 we connect STeC and CCSL by defining how a STeC process satisfy a CCSL specification in a natural way. In Sections 4.5 and 4.6 we propose the theory and algorithms to translate both STeC and CCSL into TA, in order to carry out the model checking. Finally we propose a model checking framework for STeC and CCSL in Section 4.7. Section 4.8 analyzes the time complexity of our model checking algorithm. We show that the translation process does not increase the complexity of the model checking procedure. In Section 4.9, we consider the model checking framework in which the model can contains infinite traces. With it we can extend STeC/CCSL framework to a general one with any modelling languages with infinite traces. Similar linking theory can be built while the same model checking scheme can be applied to it.

### 4.1 Traces in STeC

Since there is no recursion rules in the syntax of STeC, so in this paper, it is enough for us to consider only finite traces. However, as pointed out in Section 4.9, our theory proposed in this section can be easily extended to the case for other modelling languages with recursion rules (which thus con-
tain infinite traces).

**Definition 4.1 (Initial configuration, configurations)**
The initial configuration of a process $P$, denoted as $ic_P = (P, i_P, r_P)$, is defined as follows:

1) $ic_{Send\{l\}(m)} = (Send_{\{l\}}(m), l, t),$
   $ic_{Get\{l\}(m)} = (Get_{\{l\}}(m), l, t),$
   $ic_{a_0\{l\}(\delta)} = (a_{\{l\}}(l', \delta), l, t),$
   $ic_{\delta\{l\}(\delta)} = (\delta_{\{l\}}(\delta), l, t),$
   $ic_{Stop\{l\}} = (Stop_{\{l\}}, l, t),$
   $ic_{Skip\{l\}} = (Skip_{\{l\}}, l, t).$

2) For any $P = P_1 * P_2$ where $\ast \in \{;\},$ the configurations
   $ic_{P_1} = (P_1, l, t),$ then $ic_{P} = (P, l, t).$

The set of all configurations of $P$ is defined as:

$$Conf_{STeC}(P) = \{ (Q, l, t) \mid ic_P \xrightarrow{A}^*(Q, l, t) \}.$$

**Definition 4.2 (STeC traces)**
A word in STeC is a quadruplet $e = (a, l, t, \delta)$, where $a \in Alp$ is an alphabet, $l$ is a location, $t$ is a time point and $\delta$ is the duration of an event. The projection of each item in word $e$ is defined as $a_{act}$, $a_{l}$, $a_{\delta}$ and $a_{\epsilon}$ respectively. $a_{act}(a, l, t, \delta) = a$, $a_{l}(a, l, t, \delta) = l$, $a_{\delta}(a, l, t, \delta) = \delta$, $a_{\epsilon}(a, l, t, \delta) = \epsilon$. $\mathbb{E}_{STeC}$ denotes the set of all words in STeC.

A trace in STeC is a finite sequence of STeC words which is defined as a partial, down-closed, finite function $tr : \mathbb{N}^+ \rightarrow \mathbb{E}_{STeC}$. Given a process $P \in \mathbb{P}_{STeC}$, a trace

$$tr = \langle a_1, l_1, t_1, \delta_1 \rangle \cdots \langle a_n, l_n, t_n, \delta_n \rangle,$$

is a trace of $P$ iff there are transitions:

$$\langle P, i_P, r_P \rangle \xrightarrow{a_1} \langle P_1, l_1, t_1 \rangle \xrightarrow{a_2} \cdots \xrightarrow{a_n} \langle P_n, l_n, t_n \rangle,$$

such that $t_i - t_{i-1} = \delta_i$ for $i = 2, 3, \ldots, n$, $P_n = E$.

The set of all finite traces of $P$ is denoted as $Traces_{STeC}(P)$.

As an example, from agent gate $G$ (in Section 3) we have one trace $tr \in Traces_{STeC}(G)$:

$$tr = \langle C?Appr, Lopen, t, t - t' \rangle (Closing, Lclose, t + 10, 10)$$
$$\langle Closed, Lclose, t + 11, 1 \rangle (C!Cross, Lclose, t + 20, 0)$$
$$\langle C?Leave, Lclose, t + 80, 0 \rangle (Skip, Lclose, t + 80, 0)$$
$$\langle Open, Lopen, t + 90, 10 \rangle (Skip, Lopen, t + 90, 0).$$

4.2 Assumptions in CCSL clock and STeC model

From the definition of CCSL clock in Section 2.2.1 we see that a clock is assumed to make only one observation at a time for the same event. In other word, each clock ticks only once at a time. This stipulation is mainly for two reasons: 1) At the beginning, CCSL aims at describing logical and chronometric relations between events of real-time embedded systems, in which it is unnecessary to consider that two same events occur at a time (for example, in hardware systems, signals are only expected to occur once at a time). 2) From the view of observers, when two same events occur simultaneously, it is natural to think that actually there is no way to tell them apart.

In our case in STeC models, it is meaningful to consider that an event occurs more than once at a time. For example, in reality, it is not possible that a system can send two same messages (to the same channel) at a single time point (e.g., $P = Send_{\{l\}}(m_1); Send_{\{l\}}(m_1)$). So we keep this assumption made for CCSL clock and only consider those STeC processes in which an event can only be triggered once at a time. We call them “normal processes”, as defined below.

**Definition 4.3 (Normal STeC process)** A normal STeC process $P$ is a process such that for any $tr \in Traces_{STeC}(P)$, it satisfies

$$\mathcal{A}(a, l, t, \delta), \langle a', l', t', \delta' \rangle \in cod(tr). (a = a' \wedge t = t').$$

In the rest of this paper, all STeC processes we discuss are normal processes.

For the same reason, we shall only consider “normal traces” in TA.

**Definition 4.4 (Normal TA traces)** Given a TA $A$, the set of finite normal traces, denoted by $Traces_{TA}^N(A)$, is a subset of finite traces of $A$ that satisfies for any $tr \in Traces_{TA}^N(A)$,

$$\mathcal{A}(a, t), \langle a', t' \rangle \in cod(tr). (a = a' \wedge t = t').$$

4.3 Traces of a time structure, $\mathcal{T}S$ of a specification

We introduce the notion of traces in a time structure, and the notion of time structures of a specification. They are helpful for us to characterize STeC traces in time structures, and to describe the relationship between CCSL TA and specifications latter.

**Definition 4.5 (Traces of a time structure)** A finite trace\(^{13}\) $tr : \mathbb{N}^+ \rightarrow I$ of a time structure $TS = (I, <, \equiv, H)$ is a partial, down-closed, finite function which satisfies that

1) $tr(1) = c_1[1]$ for some $c_i$ in $TS$.
2) For any $i, j \in \mathbb{N}^+$, $i < j$ implies $tr(i) \leq tr(j)$.

\(^{13}\) As the same reason discussed in Section 4.2, we only consider finite traces.
2) For any $i \in \mathbb{N}^+$, if there is a $k \in \mathcal{I}$ such that $tr(i) \leq k \leq tr(i + 1)$, then $k = tr(i)$ or $k = tr(i + 1)$.

Figure 9 gives an example of TS trace. The dashed arrow indicates the partial order between trace elements (square dots).

![Fig. 9 An example of TS trace](image)

The notion of time structures for a specification underlies in the definition of specifications [9]. We formalize it here.

**Definition 4.6 (Time structure of specification)** Given two time structures $TS_1 = \langle \mathcal{I}_1, <_{1}, \equiv_1, H_1 \rangle$, $TS_2 = \langle \mathcal{I}_2, <_{2}, \equiv_2, H_2 \rangle$, $TS_1 \cup TS_2$ computes the union of the two time structures pointwisely:

$TS_1 \cup TS_2 = \langle \mathcal{I}_1 \cup \mathcal{I}_2, <_1 \cup <_2, \equiv_1 \cup \equiv_2, H_1 \cup H_2 \rangle$.

Let $spec = \{Cn_1, Cn_2, \ldots, Cn_k\}$ be a specification, we can give its time structures $TS_{spec}$. Each $TS \in TS_{spec}$ is given by selecting $TS_1 \in TS_{Cn_1}, TS_2 \in TS_{Cn_2}, \ldots, TS_k \in TS_{Cn_k}$ such that:

$TS = TS_1 \cup TS_2 \cup \cdots \cup TS_k$,

is well-structured.

![Fig. 10 A time structure of $\{c_a < c_b < c_d, c_a \text{ alternatesWith } c_d\}$](image)

As an example, Fig. 10 shows a time structure of constraints $spec = \{c_a < c_b < c_d, c_a \text{ alternatesWith } c_d\}$, with $h_1, h_2, h_3$ the well-selected constraint mappings in $TS_1, TS_2, TS_3$.

### 4.4 Satisfaction of a CCSL specification

Intuitively, each tick of a CCSL clock corresponds to an occurrence of one or more STeC events. The satisfaction of a CCSL specification by a STeC process can thus be defined as the behaviour of the process match each constraint of the specification, i.e., all traces of the process must obey the constraints of the specification.

In all discussions below, the clock we are actually interested in is the one that keeps track of only one event.

**Definition 4.7 (Event clock)** A single-labelled clock $c_a = \langle \mathcal{I}_a, <_a, D_a, \lambda_a \rangle$ associated to a label set $\Sigma$ is a discrete clock with

$D_a = \{a\}, a \in \Sigma$,

i.e., a clock with only one label.

To associate with the notion of events in STeC, without ambiguity, we also call label $a$ an event, $\Sigma$ an event set, $c_a$ an event clock.

In the rest of this paper, unless we specially point out, all discrete clocks we discuss are event clocks.

The next two definitions about traces will easy our illustrations of Definition 4.11 coming soon.

**Definition 4.8 (Characteristic function)** Given a trace $tr : \mathbb{N}^+ \rightarrow \Sigma$, $\Sigma$ is a set of labels. A characteristic function of trace $tr$ for a given set $A \subseteq \Sigma$, denoted by $\phi_A^r : \text{dom}(tr) \rightarrow \{0, 1\}$, is defined as

$\phi_A^r(i) = \begin{cases} 
1, & \text{if } tr(i) \in A, \\
0, & \text{otherwise}.
\end{cases}$

**Definition 4.9 (Sub trace)** Given a trace $tr$ and an event set $A \subseteq \Sigma$, a sub trace of $tr$, denoted by $tr|_A : \mathbb{N}^+ \rightarrow \Sigma$, it is defined as:

$tr|_A(i) = tr(j)$,

provided that

$j = min\{k \mid tr(k) \in A \land \sum_{l=1}^{k} \phi_A^r(l) = i\}$

exists.

Intuitively, the $ith$ element of $tr|_A$ is the element of $tr$ whose event name is in $A$ and is the $ith$ such element appeared in $tr$.

It is clear to check that the sub trace $tr|_A : \mathbb{N}^+ \rightarrow \Sigma$ built by Definition 4.9 is a partial, down-closed function.

To link CCSL constraints to STeC processes, the crucial idea is to build a mapping from STeC events to CCSL clocks. We next introduce the notion of observational events and clocks.

**Definition 4.10 (Observational event, observational clock)** For a STeC event $a \in \text{Alp}$, $\overline{a}$ denotes the event in CCSL constraints that observes $a$. We call it an observational counterpart, or a synchronizing counterpart of $a$. 
Call an event set
\[ S = \{a \mid a \in S\}, \]
the observational event set of \( S \).

For an event clock \( c_{\pi} \) in \( S \) with \( a \in \text{Alp} \), we call \( c_{\pi} \) an observational clock of \( a \).

Call the mapping \( (\cdot) : a \mapsto \pi \) the observational mapping.

Note that the observational mapping is not necessarily an injection, an example is Property 3 (Section 6), where the mapping is defined as \( (\cdot) = \{\text{Appr} \mapsto \text{appr}, \text{Closed} \mapsto \text{close}\} \).

We introduce a function \( \mathcal{C}(\mathcal{C}_n) \) to return the set of all clocks appeared in \( \mathcal{C}_n \). For instance, \( \mathcal{C}(c_a < c_b) = \{c_a, c_b\} \). A function \( \mathcal{D}(\mathcal{C}_n) \) returns the set of all labels appeared in \( \mathcal{C}_n \). It is defined as \( \mathcal{D}(\mathcal{C}_n) = \{a \mid a \in \pi_{STeC}\} \). For example, \( \mathcal{D}(c_a < c_b) = \{a, b\} \). For a specification \( \text{spec} \), we have \( \mathcal{C}(\text{spec}) = \bigcup_{\mathcal{C}_n \in \text{spec}} \mathcal{C}(\mathcal{C}_n) \) and \( \mathcal{D}(\text{spec}) = \bigcup_{\mathcal{C}_n \in \text{spec}} \mathcal{D}(\mathcal{C}(\mathcal{C}_n)) \) to compute the set of clocks and events appeared in it.

The next definition declares the link between CCSL and STeC.

**Definition 4.11 (Satisfaction of CCSL specification)** Let \( P \) be a STeC process, \( \text{spec} \) be a CCSL specification defined over \( TS_{\text{spec}} \), let

\[ (\cdot) : S \rightarrow \mathcal{D}(\text{spec}), \]

be an observational mapping, \( S \subseteq \text{At}(P) \). Given a trace \( tr \in \text{Traces}_{\text{STeC}}(P) \), an injective mapping

\[ f : \text{dom}(tr|_S) \rightarrow \bigcup_{a \in \mathcal{D}(\text{spec})} I_a, \]

is defined as

\[ f(i) = c_{\pi}[j], \]

where \( x_i = \pi_{\text{act}}(tr|_S(i)), j = \sum_{k=1}^{i} \phi_{x_i}(k), c_{\pi} \) is an observational clock of \( x_i \).

Say \( tr \) satisfies \( \text{spec} \) with respect to \( (\cdot) \), denoted as \( tr \models_{(\cdot)} \text{spec} \), iff there exists a time structure

\[ TS = \langle \bigcup_{a \in \mathcal{D}(\text{spec})} I_a, \preceq, H \rangle \in TS_{\text{spec}}, \]

such that the mapping \( f \) satisfies for any \( i, j \in \text{dom}(tr|_S) \) with \( \pi_{\text{act}}(tr|_S(i)) \neq \pi_{\text{act}}(tr|_S(j)) \).

1) \( \pi_{\text{act}}(tr|_S(i)) < \pi_{\text{act}}(tr|_S(j)) \) implies \( f(i) < f(j) \).
2) \( \pi_{\text{act}}(tr|_S(i)) = \pi_{\text{act}}(tr|_S(j)) \) implies \( f(i) \equiv f(j) \).

In fact, \( f \) is a trace of such a \( TS \).

If \( tr \models_{(\cdot)} \text{spec} \) for all \( tr \in \text{Traces}_{\text{STeC}}(P) \), we say the process \( P \) satisfies \( \text{spec} \) with respect to \( (\cdot) \), written as

\[ P \models_{(\cdot)} \text{spec}. \]

Figure 11 shows a clear picture of how we build the trace \( f \) in \( TS \), where \( (\cdot) \) is an injection that maps \( x \) to its observational event \( \pi (x \in \{a, b, c\}) \) in \( TS \). The \( i \)-th tick of the corresponding clock. The sequence (made of square dots) is exactly the trace \( f \).

![Fig. 11 Relation between \( tr|_S \) and \( TS \)](image-url)

**Definition 4.11** essentially links an occurrence of a STeC event to a tick of CCSL clock by the mapping \( (\cdot) \) and \( f \). It is easy to prove that \( f \) is an injection and is a trace of \( TS \). The time structure \( TS \) plays two roles: on one hand, it preserves the order from the STeC trace to its corresponding observational trace \( f \) in \( TS \) (shown as the conditions “1)”, “2)” above). On the other hand, \( TS \) is itself well-structured over \( \text{spec} \) (because \( TS \in TS_{\text{spec}} \)), which means that the \( f \) satisfies the specification \( \text{spec} \).

Note that in the above definition, we do not assume that \( \text{spec} \) only contains observational clocks. It may contain other clocks, we call them generated clocks, which will be introduced next.

### 4.5 Generated clocks

Sometimes when we specify properties, we need to produce an auxiliary clock (which does not observe any STeC events) in order to give a correct description. We call such clock a generated clock [24]. For example in Property 2 (Section 6), we wish to express that if the train stops, it must stop 50s after the train approaches. For this property, it is wrong to use the specification

\[ c_{\text{stop}} = c_{\text{appe}} \text{ delay 50 on IdealClk}. \]
since the train may not stop after it approaches the gate according to the behaviour of $P_{RCS}$ given in Section 3. The solution is to introduce a new clock named $c_{\text{mayStop}}$, to record the time points at which $c_{\text{stop}}$ may occur. Thus the correct specification results in
\[
\{c_{\text{mayStop}} = c_{\text{appr}} \text{ delay 50 on IdealCk}, c_{\text{Stop}} \subseteq c_{\text{mayStop}}\},
\]
which means, if the train stops, it must stop at the time points 50s after the train approaches. Here the generated clock $c_{\text{mayStop}}$ ticks restrictively by the two primitive constraints above.

**Definition 4.12 (Generated event, generated clock)** An event $a \notin \mathbb{Alp}$ is called a generated event. An event clock $c_a = (I_a, <_a, D_a, \Lambda_a)$ in a specification is called a generated clock.

A set of generated events is denoted by $\Sigma_G$.

Apart from these definitions, we give an illustration for the “observational relationship” between STeC models and CCSL constraints, see Fig. 12.

**Fig. 12** Relations between a STeC model and CCSL constraints

### 4.6 Timed model for CCSL and STeC

As indicated at the beginning of Section 4, an alternative approach is to take CCSL constraints as observers for STeC processes. In order to implement such an observational relationship indicated in Fig. 12 so that model checking can be carried out, we need to translate both STeC and CCSL to LTSs.

For CCSL, a translation into TA [17] is based on Clock Labelled Transition System (cLTS) in [19], which is an untimed synchronous model where if two ticks occur simultaneously, it is expressed as a set of labels. For example, constraint
\[
\{\pi \subseteq \underline{\pi}, \bar{\pi}\}
\]
is expressed as a cLTS in Fig. 13. If $\pi$ and $\bar{\pi}$ tick simultaneously, the cLTS emits a set $\{[\pi, \bar{\pi}]\}$.

**Fig. 13** cLTS of $\pi \subseteq \underline{\pi}$

However, cLTS is not suitable for asynchronous languages where only one occurrence of an event is expected at a time. If two events occur at the same time, it causes nondeterministic choice which one comes first. Since STeC is an asynchronous language, when dealing with synchronization with STeC models, we need to consider the synchronization between $a$ and $\pi$, $b$ and $\bar{\pi}$ separately, rather than $[a, b]$ and $[\pi, \bar{\pi}]$ as a whole. In addition, cLTS does not support time issues, thus in which we can not express constraints relating to dense clocks, e.g., the $\text{Delay On}$ operator for dense clock.

For this reason, based on [17] we consider an asynchronous TA model for CCSL. In TA, we turn a set of synchronous events in cLTS into several asynchronous events that occur at the same physical time. For example, if two ticks $\pi$, $\bar{\pi}$ happen at the same time, we introduce two transitions indicating the occurrences of $\pi$ and $\bar{\pi}$ separately, but with no time consuming between them (as in Table 3 shows). As a result our timed model of CCSL not only can reflect the logical aspect of clocks, but also the chronometrical aspect of them.

For STeC, the operational semantics is itself time sensitive. So we propose a translation from STeC into its equivalent TA.

We now implement the observations from CCSL to STeC echoing Fig. 12. It turns out to be the synchronized product of timed automata originally owed to [23]. We modify the synchronizing condition there in order to adapt it to the observational mappings defined in Definition 4.10.

**Definition 4.13 (Synchronized product of TA)** Given two TA $A_1 = (Q_1, S_1 \cup O, C_1, i_1, Ed_1, I_1, AP_1, L_1, F_1)$, $A_2 = (Q_2, S_2 \cup \overline{O}, C_2, i_2, Ed_2, I_2, AP_2, L_2, F_2)$, where $\overline{O}$ is the observable event set of $O$ by some mapping $\overline{\cdot}$. The synchronized product
\[
A_1 \times A_2 = (Q, S, C, i, Ed, I, AP, L, F),
\]
is defined as:

1) $Q \subseteq Q_1 \times Q_2$ contains states of the form $(q_1, q_2)$ where $q_1 \in Q_1$ and $q_2 \in Q_2$.

2) $S = S_1 \cup S_2 \cup \{\tau\}$, $\tau$ is an internal event, $\tau \notin S_1 \cup S_2$.

3) $C = C_1 \cup C_2$.

4) $i = \langle i_1, i_2 \rangle$.

5) $Ed = Ed_a \cup Ed_b \cup Ed_c$ where
\[
Ed_a = \{(q_1, q_2), s_1, g_1, Y_1, (q'_1, q_2) \mid s_1 \in S_1 \land q_1, s_1, g_1, Y_1, q'_1 \in Ed_1\}.
\]
\[
Ed_b = \{(q_1, q_2), s_2, g_2, Y_2, (q_1, q_2') \mid s_2 \in S_2 \land q_2, s_2, g_2, Y_2, q_2' \in Ed_2\}.
\]
Edc = \{ (\langle q_1, q_2 \rangle, \tau, g_1 \land g_2, Y_1 \cup Y_2, \langle q_1', q_2' \rangle) \mid
\}
\begin{align*}
o \in O \land \overline{\sigma} \in \overline{O} \land \langle q_1, o, g_1, Y_1, q_1' \rangle \in Ed_1 \Land 
\langle q_2, \overline{\sigma}, g_2, Y_2, q_2' \rangle \in Ed_2\}.
\end{align*}

6) For each \( \langle q_1, q_2 \rangle \in Q, I((q_1, q_2)) = I_1(q_1) \land I_2(q_2) \).
7) \( AP = AP_1 \cup AP_2 \).
8) For each \( \langle q_1, q_2 \rangle \in Q, L((q_1, q_2)) = L(q_1) \lor L(q_2) \).
9) \( F \) contains state \( \langle q_1, q_2 \rangle \) where \( q_1 \in F_1 \) and \( q_2 \in F_2 \).

4.6.1 TA for CCSL specification

The timed semantics of primitive constraints in TA we give is similar to the untimed one in cLTS in [19], except for the primitives that concerns time issues, such as “sub-clock” and “delay on” for dense clock.

The translation from primitive constraints into TA is given in Table 3, where all states of each TA are accepting states, \( t, c \) are clocks. Each transition is labelled by three parts: an event, a guard and a set of reset clocks. For example, in the TA of \( c_b = c_t Delay n On c_f, \) the condition “\( b, c = n - 1, t := 0' \)” means that the event \( b \) is triggered when \( c = n - 1, t \) is reset to zero. The constraints drawn beside nodes are invariants. e.g., in the TA of \( c_a \subseteq c_b \) the invariant of state 2 is \( t \leq 0 \).

The TA of a specification is the composition of the TA of its primitive constraints. The composition of cLTS is given in [19], based on which we give a version for TA.

Definition 4.14 (Composition for CCSL TA) Given two TA \( A_1 = (Q_1, S_1, C_1, i_1, Ed_1, I_1, L_1, F_1) \) and \( A_2 = (Q_2, S_2, C_2, i_2, Ed_2, I_2, AP_2, L_2, F_2) \) of primitive constraints, let

\[ S_3 = S_1 \cap S_2, \]

be the common events of \( A_1 \) and \( A_2 \). The composition for CCSL TA, denoted by \( A_1 \otimes A_2 \), is defined just the same as \( A_1 \times A_2 \) in Definition 4.13, except that

\[ S = S_1 \cup S_2, \]

and

\[ Ed = Ed_a \cup Ed_b \cup Ed_c, \]

where

\[ Ed_a = \{ (\langle q_1, q_2 \rangle, s_1, g_1, Y_1, \langle q_1', q_2' \rangle) \mid s_1 \in S_1 \Land \langle q_1, s_1, g_1, Y_1, q_1' \rangle \in Ed_1 \}, \]

\[ Ed_b = \{ (\langle q_1, q_2 \rangle, s_2, g_2, Y_2, \langle q_1', q_2' \rangle) \mid s_2 \in S_2 \Land \langle q_2, s_2, g_2, Y_2, q_2' \rangle \in Ed_2 \}, \]

\[ Ed_c = \{ (\langle q_1, q_2 \rangle, s_3, g_1 \land g_2, Y_1 \cup Y_2, \langle q_1', q_2' \rangle) \mid s_3 \in S_3 \Land \langle q_1, s_3, g_1, Y_1, q_1' \rangle \in Ed_1 \Land \langle q_2, s_3, g_2, Y_2, q_2' \rangle \in Ed_2 \}. \]

The synchronized transition here (in \( Ed_c \)) is different from that in Definition 4.13. The same clock events in two CCSL TA must be synchronized when making the composition.

With Table 3 and Definition 4.14, we give the translation from CCSL to TA.

Definition 4.15 (Translation from CCSL to TA) The TA semantics of a CCSL specification, denoted as \( \llbracket spec \rrbracket_{CCSL} \), is defined as

\[ \llbracket spec \rrbracket_{CCSL} = \prod_{Cn \in \text{spec}} Cn_{CCSL}, \]

where each \( Cn_{CCSL} \) is defined according to Table 3.

As an example, Fig. 14 gives the translated TA of specification \( spec_2 \) of Property 2 in Section 6, where \( c_1, c_2 \) corresponds to the clock in \( \llbracket c_{mayStop} = c_{app} \land \text{delay on} 50 \lor \text{IdealClock} \rrbracket_{CCSL} \) and \( \llbracket c_{stop} = \underline{c_{mayStop}} \rrbracket_{CCSL} \) respectively. Events \( \overline{X} \) expresses the observational events in CCSL.

![Fig. 14 The encoded TA of spec2](image-url)

The following theorem says, in some sense, we can think a CCSL specification is equivalent to its encoded TA.

Theorem 4.1 Given a specification \( spec \) defined over \( TS_{spec} \), A surjective function

\[ f : \text{Traces}_{TA}^N(\llbracket spec \rrbracket_{CCSL}) \rightarrow \bigcup_{TS \in TS_{spec}} \text{Traces}_{TS}(TS), \]

is a “forget” function that forgets the time information about a trace. It is defined that for any trace

\[ tr = \langle a_1, t_1 \rangle \langle a_2, t_2 \rangle \cdots \langle a_n, t_n \rangle, \]

\[ f(tr) = a_1 a_2 \cdots a_n : \mathbb{N}^* \rightarrow \{ a_1, a_2, \ldots, a_n \}, \]

is a trace. Define the set

\[ f^{-1}(tr) \triangleright TS = \{ tr' \mid tr' \in f^{-1}(tr) \land f(tr') \in \text{Traces}_{TS}(TS) \}. \]

We claim that \( \llbracket spec \rrbracket_{CCSL} \) coincides with \( spec \) in the sense that \( f \) satisfies
1) If $tr \in \text{Traces}_{TA}^N[\text{spec}_{\text{CCSL}}]$, then we can find a time structure $TS$ in which $f(tr) \in \text{Traces}_{TS}(TS)$, and
- for any $(a_i, t_i), (a_j, t_j)$ with $t_i < t_j$ in $\text{spec}_{\text{CCSL}}$, then $a_i \subseteq a_j$ in $TS$,
- for any $(a_i, t_i), (a_j, t_j)$ with $t_i = t_j$ in $\text{spec}_{\text{CCSL}}$, then $a_i \equiv a_j$ in $TS$.

2) If $tr \in \text{Traces}_{TS}(TS)$ for some $TS \in \mathcal{T}_S^{\text{spec}}$, then for all $tr' \in f^{-1}(tr) \triangleright TS$, $tr' \in \text{Traces}_{TA}^N[\text{spec}_{\text{CCSL}}]$.

**Proof** It can be proved by the induction based on the structure of $\text{spec}$. For primitive constraints, we just give an example: $c_a \subseteq c_b$. Others are similar.

(Base case, $\Rightarrow$). Let $tr = \langle a_1, t_1 \rangle \langle a_2, t_2 \rangle \cdots \langle a_n, t_n \rangle$ is a normal TA trace in $\text{spec}_{\text{CCSL}}$, and $f(tr) = a_1 a_2 \cdots a_n$, we build a $TS = (I, <, \equiv, [h])$ following several steps:

1) First, let $I = I_a \cup I_b$. In $TS$ we add relations to $<$ such that $TS$ is well-structured in the sense of $c_a$ and $c_b$:
   - whenever $x <_a y$ in $c_a$, we add $x < y$ in $<$,
   - whenever $x <_b y$ in $c_b$, we add $x < y$ in $<$.  

2) Make $f(tr)$ become a trace in $TS$. To realize it, first we define the mapping
   $$
g : \text{dom}(f(tr)) \to I_a \cup I_b,$$

   as
   $$g(i) = c_a[j],$$

   where $j = \sum_{k=1}^{i} \phi^{(tr)}(k), a_i \in \{a, b\}$, i.e., mapping $a_i$ to the $j$th element of $c_a$, $j$ is the occurrence times in $f(tr)$ before $a_i$. Then we enrich relations $<$ and $\equiv$ as:
   - for $(a_i, t_i), (a_j, t_j)$ with $t_i < t_j$, we define $g(i) < g(j)$ in $TS$,
   - for $(a_i, t_i), (a_j, t_j)$ with $t_i = t_j$, we define $g(i) \equiv g(j)$ in $TS$. (Note that $a_i, a_j$ here must be in different clocks.)

3) The constraint mapping $h : I_a \to I_b$ can be built as:
   - whenever $x \equiv y, x \in I_a$ and $y \in I_b$, we have $h(x) = y$. (Note that given such $x$, there must be just one such $y$ with $x \equiv y$.)

   Easy to check that such a $TS$ we built is well-structured and $c_a \subseteq c_b$ is defined over it.

(Base case, $\Leftarrow$). If $tr = a_1 a_2 \cdots a_n \in \text{Traces}_{TS}(TS)$ for some $TS \in \mathcal{T}_S_{c_a \subseteq c_b}$, according to definition of $f$, it is easy to check that any normal traces in $f^{-1}(tr) \triangleright TS$ is in $\text{TA } \parallel c_a \subseteq c_b \parallel_{\text{CCSL}}$.

(Inductive step, $\Rightarrow$). Now suppose $\text{spec} = \text{spec}_1 \cup \text{spec}_2$, and the theorem holds for any constraints in $\text{spec}_1$ and $\text{spec}_2$ respectively.

If $tr \in \text{Traces}_{TA}^N[\text{spec}_{\text{CCSL}}]$, since

$$\text{spec}_{\text{CCSL}} = \text{spec}_1_{\text{CCSL}} \otimes \text{spec}_2_{\text{CCSL}},$$

we consider $tr|_{\text{Ds}^{\text{spec}}_1} \in \text{Traces}_{TA}^N[\text{spec}_1_{\text{CCSL}}]$ and $tr|_{\text{Ds}^{\text{spec}}_2} \in \text{Traces}_{TA}^N[\text{spec}_2_{\text{CCSL}}]$. By the assumption we know there exists $TS_1 = \langle I_1, <, \equiv, H_1 \rangle, TS_2 = \langle I_2, <, \equiv, H_2 \rangle$ such that $f(tr|_{\text{Ds}^{\text{spec}}_1})$ is a trace in $TS_1$ and $f(tr|_{\text{Ds}^{\text{spec}}_2})$ is a trace in $TS_2$, and $TS_1, TS_2$ are well-structured with $\text{spec}_1, \text{spec}_2$ defined over them respectively. Consider

$$TS = TS_1 \cup TS_2,$$

easy to check that $TS$ must be a well-structured time structure with $\text{spec}$ defined over it.

Figure 15 gives a picture of the relation between $TS, TS_1$ and $TS_2$, where the dashed square indicates the common part of events in $I_1$ and $I_2$, the solid squares indicate the events of $I_1$ and $I_2$ respectively. From it we can see that in order to make $tr$ become a trace in $TS$, the only enrichment we need to make is the relations (like $r_1$) that neither belongs to $TS_1$ or $TS_2$ (while relations like $r_2, r_3, r_4$ have already existed in either $TS_1$ or $TS_2$). So for any $(x, t_x), (y, t_y)$ in $tr$ with $x \in I_1 - I_2, y \in I_2 - I_1$, we have:

1) if $t_x < t_y$, we add $x < y$ in $TS$,
2) if $t_x = t_y$, we add $x \equiv y$ in $TS$.

We write the time structure after the enrichment as $TS'$. It is not hard to show that $TS'$ is still well-structured. This is because from Fig. 15 one can easily see that any triangle relations (like $r_1, r_2, r_3$) is impossible to be derived as a contradiction in $TS'$. For example, if $x < z$ in trace $tr|_{\text{Ds}^{\text{spec}}_1}$, $z \equiv y$ in trace $tr|_{\text{Ds}^{\text{spec}}_2}$, then there must be $x < y$ in trace $tr$. And $x < y$ does not affect any constraint mappings since all of them do not concern $x$ and $y$ at the same time.

![Fig 15 Relation between $TS, TS_1$ and $TS_2$](image-url)
Traces_{TS}(TS_1) and \(tr_{I_2} \in \text{Traces}_{TS}(TS_2)\), with \(TS_1, TS_2\) well-structured and \(spec_1, spec_2\) defined over them respectively. Thus by assumption for any \(tr_1 \in f^{-1}(tr_{I_2}) \triangleright TS_1\), \(tr_2 \in f^{-1}(tr_{I_2}) \triangleright TS_2\), there are \(tr_1 \in \text{Trace}^N_{TA}([spec_1]_{CCSL})\) and \(tr_2 \in \text{Trace}^N_{TA}([spec_2]_{CCSL})\). Since for any \(tr' \in f^{-1}(tr) \triangleright TS\), we can find such \(tr_1, tr_2\) with \(tr'I_{I_2} = tr_1\) and \(tr'_{I_2} = tr_2\), from the definition of \(f\), easy to see that \(tr' \in \text{Trace}^N_{TA}([spec]_{CCSL})\).

Note that \(f(tr)\) might be a trace in more than one time structures. That is why we consider \(f^{-1}(tr_{I_2}) \triangleright TS\) instead of \(f^{-1}(tr)\). Function \(f\) is surjective because TA traces contain time information that indicates at which time an event occurs. However in the time structure \(TS\), though it has time issues (like Sub Clock or Delay On for dense clock), only the relationship between clocks is reflected.

### 4.6.2 TA for STeC process

Intuitively, according to the operational semantics of STeC, each STeC configuration can be seen as a configuration in a TA, each time-only transition between configurations can be seen as a time-only transition between TA configurations, each process transition can be seen as a transition between TA states. In this section we formalize this intuition by first introducing the notion of “STeC configuration region” where there are only time-only transitions between two configurations, then we correspond each region to a state in a TA. In this way the process transitions correspond exactly to the transitions between states.

One thing should be noticed is that the location in STeC turns out to be an atomic proposition in the state of TA (as shown in Definition 4.18). The encoded TA does not lose the capability to express the “locations” in STeC. In fact, as indicated at the beginning of Section 2.1, the location in STeC is only used for checking the spatio-temporal consistencies at the semantics level. It does not give any more information about the system. What we lose in TA is the restriction of locations: in STeC, the restriction is guaranteed by its semantics, while in TA, such restriction is missing, since we can not build a transition whose firing relies on the satisfaction of propositions. However, the lost of restrictions does not affect that we use TA as models in model checking with CCSL, since CCSL does not include any constraints that concern locations in STeC. It is efficient to require that the expressive power of a STeC process and its encoded TA are the same in the sense of Theorem 4.2.

**Definition 4.16 (Configuration region)** A STeC configuration region is a set of configurations, denoted by \([P, l, t]\) where \(P\) is a process and \(l\) is a location. It is defined as

\[
[P, l, t] = \{ (P, l, t') \mid (P, l, t') \in \text{Conf}_{STeC}(P) \land t' \geq t\},
\]

such that for any \((P, l, t_1), (P, l, t_2) \in [P, l, t]\) with \(t_2 > t_1(\geq t)\), we have

\[
(P, l, t_1) \xrightarrow{\text{reg}} (P, l, t_2),
\]

i.e., there are only time-only transitions between \((P, l, t_1)\) and \((P, l, t_2)\).

Use \(\text{reg}\) to range over all regions.

The only existence of time-only transitions between \((P, l, t_1)\) and \((P, l, t_2)\) can be easily proved according to the operational semantics in Table 1, 2.

**Definition 4.17 (Configuration Region Transition System (CRTS))** Based on Definition 4.16 we can define a transition system, called Configuration Region Transition System (CRTS), in which the states are configuration regions in the form of \([P, l, t]\), transitions between regions are in the form of \([P, l, t] \xrightarrow{a} [P', l', t]\) where \(a \in \text{Alp}\).

Given a STeC process \(P\), the CRTS respected to \(P\) can be built according to the following rules:

1) For the initial configuration \((P, l_p, t'_p)\) given in advance, we have a state \([P, l_p, t'_p]\).

2) For any \((P, l, t) \in [P, l, t]\), if there exists a process transition \((P, l, t) \xrightarrow{a} (P', l', t)\), then we have a state \([P', l', t]\), and a transition \([P, l, t] \xrightarrow{a} [P', l', t]\).

The corresponding CRTS of \(P\) is denoted as \(\text{CRTS}_P\).

Now we are ready to translate STeC into TA. The only difference between TA and CRTS is that we need to use TA clocks to express time issues instead of the symbol \(t\).

**Definition 4.18 (Translation from STeC to TA)** Given a STeC process \(P\) and its corresponding CRTS \(\text{CRTS}_P\), the TA semantics of \(P\), denoted by \([P]_{\text{TA} \text{- STeC}}\), is a TA

\[
(Q, \Sigma, C, i, Ed, I, AP, L, F),
\]

constructed according to following rules:

1) \(Q\) is the set of all regions in \(\text{CRTS}_P\).

2) \(\Sigma = \text{At}(P)\).

3) For each transition \(\text{reg}_i \xrightarrow{a} \text{reg}_j\) in \(\text{CRTS}_P\), we have a clock denoted as \(c(\text{reg}_i, a, \text{reg}_j)\) in \(C\).

4) \(i = [P, l_p, t'_p]\) where \((P, l_p, t'_p)\) is the initial configuration given in advance.
5) For any state \([P, l, t] \in Q, L(P, l, t) = l\).
6) \(F\) has states of the form \([E, l, t]\).
7) For each transition \([P, l, t] \xrightarrow{a} [P', l', t']\) in \(CRT_{SP}\), we have a transition \(\langle [P, l, t], a, g, Y, [P', l', t'] \rangle\) in \(Ed\) where
   a. \(g := c([P, l, t], a, [P', l', t']) = t' - t\).
   b. \(I([P, l, t])\) is in the form of \([\cdots \land c([P, l, t], a, [P', l', t']) \leq t' - t \land \cdots]\).
   c. \(Y\) consists of all clocks in the form of \(c([P', l', t'], a, \text{reg})\) where \(\text{reg} \in Q\) is an arbitrary state.
8) For any \(\text{reg} \in Q\), \(I(\text{reg})\) is constructed as in 7), i.e., the conjunction of the invariants of all transitions from \(\text{reg}\).

Consider the earlier example \(PI_{RCS} = T \Rightarrow G\), we can compute both \(\|T\|_{STeC}\) and \(\|G\|_{STeC}\) shown in Figs. 16 and 17 respectively.

A \(STeC\) process is equivalent to its encoded TA in the following sense.

**Theorem 4.2** Given a \(STeC\) process \(P\). A bijective function 

\[ f : Traces_{TA}(\|P\|_{STeC}) \rightarrow Traces_{STeC}(P), \]

is defined that for any trace 

\[ tr = \langle a_1, t_1 \rangle \langle a_2, t_2 \rangle \cdots \langle a_n, t_n \rangle \in Traces_{TA}(\|P\|_{STeC}), \]

along with the transitions 

\[ [P, l_p, t_p] \xrightarrow{a_1} [P, l_1, t_1] \xrightarrow{a_2} \cdots \xrightarrow{a_n} [P_n, l_n, t_n], \]

we have 

\[ f(tr) = \langle a_1, l_1, t_1, t_1 - t_p \rangle \langle a_2, l_2, t_2 - t_1 \rangle \cdots \langle a_n, l_n, t_n - t_{n-1} \rangle, \]

in \(Traces_{STeC}(P)\). \(\|P\|_{STeC}\) coincides with \(P\) in the sense that \(f\) satisfies:

\[ tr \in Traces_{TA}(\|P\|_{STeC}) \text{ iff } f(tr) \in Traces_{STeC}(P). \]

**Proof** From the constructions of \(CRTs\) and TA of \(STeC\), it can be easily proved by the induction on the length of \(STeC\) processes.

By Definition 4.18 we can translate a process \(P = P_1 \parallel P_2 \parallel \cdots \parallel P_n\) into a TA by dividing \(P\) into configuration regions. It is useful to know that our translation preserves the parallel composition \(\parallel\) in the sense of Proposition 4.1. When we use model checking tools like UPPAAL to build our models, we actually do not need to translate the whole \(P\), but each component \(P_1, P_2, \ldots, P_n\). Since these tools take lazy-evaluation strategy to deal with parallel composition.

**Definition 4.19 (Composition for \(STeC\) TA)** Given two TA \(A_1 = (Q_1, S_1 \cup Ch_1, C_1, l_1, Ed_1, l_1, L_1, F_1)\), \(A_2 = (Q_2, S_2 \cup Ch_2, C_2, l_2, Ed_2, l_2, AP_2, L_2, F_2)\) of \(STeC\) processes, \(Ch_1, Ch_2\) are set of communicating events of the form \(C \cdot m\) between \(A_1\) and \(A_2\), where \(\cdot \in \{!, ?\}\). The composition of \(STeC\) TA, denoted as \(A_1 \odot A_2\), is defined just as the same as \(A_1 \times A_2\) in Definition 4.13, except that 

\[ S = S_1 \cup S_2 \cup Ch, \]

where elements in \(Ch\) are of the form \(C.m\) with \(C.m \in (C?m), C\cdot m \in Ch_1\) and \(Ch_2\) respectively.

\[ Ed = Ed_a \cup Ed_b \cup Ed_c, \]

where

\[ Ed_a = \{\langle q_1, q_2, s_1, g_1, Y_1, \langle q_1', q_2' \rangle \rangle | s_1 \in S_1 - Ch_1 \land \langle q_1, s_1, g_1, Y_1, \langle q_1' \rangle \rangle \in Ed_1\}; \]
\[ Ed_b = \{\langle q_1, q_2, s_2, g_2, Y_2, \langle q_1, q_2' \rangle \rangle | s_2 \in S_2 - Ch_2 \land \langle q_2, s_2, g_2, Y_2, \langle q_2' \rangle \rangle \in Ed_2\}; \]
\[ Ed_c = \{\langle q_1, q_2, C.m, g_1 \land g_2, Y_1 \cup Y_2, \langle q_1', q_2' \rangle \rangle | (C \cdot m \in Ch_1 \land C \cdot m \in Ch_2) \land \langle q_1, C \cdot m, g_1, Y_1, \langle q_1' \rangle \rangle \in Ed_1 \land \langle q_2, C \cdot m, g_2, Y_2, \langle q_2' \rangle \rangle \in Ed_2\}. \]

where the pair \((\cdot, \cdot)\) could be \((!, !)\) or \((?, ?)\).
Proposition 4.1 (Preservation of translation) The translation \([\mathcal{L}]_{\text{STeC}}\) preserves operation \(\otimes\) with respect to \(\otimes\), that is, given a process \(P = P_1 \otimes P_2 \otimes \cdots \otimes P_n\), there is

\[
[\mathcal{L}]_{\text{STeC}} = [P]_{\text{STeC}} \otimes [P]_{\text{STeC}}.
\]

This property about STeC translation will be useful in Section 5, where we translate each agent \(P_i\) into UPPAAL TA, rather than \(P\).

4.6.3 Algorithm translating STeC and CCSL into TA

Based on Definitions 4.14–4.18, we propose Algorithms 1–4, as a general implementation to translate STeC and CCSL into their corresponding TA. They form the base for implementing STeC/CCSL framework in practice and adopting the existing verification methodologies such as UPPAAL, which will be discussed in the following sections.

To avoid the detail of the implementation we only give a general description. We choose mathematical data structures and operations rather than the concrete ones. For example, we use mathematical set and set operations such as union \(A \cup B\). To save pages sometimes we use English sentence to express one of series of operations in real programs. Like in line 4 of Algorithm 1. The key words follow the traditional meanings in programming languages, like \textbf{if then} \textbf{else}, \textbf{for do}, etc. \textbf{procedure} just means function, and \textbf{let} is the declaration of local variables.

---

## Algorithm 1: Translation from CCSL into TA

```plaintext
1: procedure CCSL2_TA(spec)
2:     let \(R\) be an init empty TA
3:     for each \(Cn\) in spec do
4:         directly get the TA \(R_{Cn}\) of \(Cn\) based on Table 3
5: Compute \(R := \bigotimes_{\text{Cn in spec}} R_{Cn}\)
6: return \(R\)
```

## Algorithm 2: Translation from STeC into TA

```plaintext
1: procedure STeC2_TA(P)
2:     let \(A := (Q, E, C, i, Ed, I, AP, L, F)\) be an empty TA
3:     let \(i := \{P, \bar{P}, \bar{I}, \bar{F}\}\) be the init state
4: Build_TA(A, i)
5: return \(A\)
```

## Algorithm 3: Function \text{trans\_atom}

```plaintext
1: procedure Trans_Atom([P, I, t], T)
2:     if \(P = Q, R\) then
3:         let \(T' := \text{Trans}([Q, I, t])\)
4:         for each \([Q, I, t] \rightarrow [Q', I', t'] in T'\) do
5:             if \(a = \text{Skip}\) then
6:                 \(T := T \cup \{[P, I, t] \rightarrow [R, I', t']\}\)
7:             else
8:                 \(T := T \cup \{[P, I, t] \rightarrow [Q', I', t']\}\)
9:         else if \(P = Q \parallel R\) then
10:            let \(T' := \text{Trans}([Q, I, t]) \cup \text{Trans}([R, I, t])\)
11:            for each \([Q, I, t] \rightarrow [Q', I', t'] in T'\) do
12:                \(T := T \cup \{[P, I, t] \rightarrow [Q', I', t']\}\)
13:            for each \([R, I, t] \rightarrow [R', I', t'] in T'\) do
14:                \(T := T \cup \{[P, I, t] \rightarrow [R', I', t']\}\)
15:            else if \(P = Q \otimes R\) then
16:                let \(T' := \text{Trans}([Q, I, t]) \otimes \text{Trans}([R, I, t])\)
17:                for each \([Q, I, t] \rightarrow [Q', I', t'] in T'\) do
18:                    \(T := T \cup \{[P, I, t] \rightarrow [Q', I', t']\}\)
19:                for each \([R, I, t] \rightarrow [R', I', t'] in T'\) do
20:                    \(T := T \cup \{[P, I, t] \rightarrow [Q \parallel R', I', t']\}\)
21:            else if \(P = Q \otimes R\) then
22:                let \(T' := \text{Trans}([Q, I, t]) \otimes \text{Trans}([R, I, t]), \text{flag}=0\)
```

---
Follows the algorithm in [19]. Here we omit the
returns a TA. Build_TA is for building the TA step by step
based on De

By now we have all things needed to propose our model
framework of STeC and CCSL

Theorem 4.3 (Model checking CCSL constraints in a

Algorithm 1 translates CCSL to TA, the computation of \( \otimes \)
(at line 5) follows the algorithm in [19]. Here we omit the
detail. Algorithm 2 contains three main procedures, the main
procedure is STeC_2_TA(P), it accepts a STeC process P and
returns a TA. Build_TA is for building the TA step by step
based on configuration region \([P, l, i]\). The whole process is
based on Definition 4.18. Procedure Trans computes the
transitions of region \([P, l, i]\), based on the operational semantic
rules in Tables 1 and 2. Algorithms 3 and 4 give the detail for
the cases when P is an atomic or a compositional process.

4.7 Model checking framework of STeC and CCSL

By now we have all things needed to propose our model
checking algorithm to verify CCSL properties in a STeC
model. Echoing the beginning of Section 4, we propose an
algorithm to check if \( P \models spec \) given a process P, an obsen-

**Theorem 4.3 (Model checking CCSL constraints in a

**STeC model** Let P be a STeC process, \( spec \) be a CCSL
specification, and

\[ \overline{\overline{\text{!}}} : S \rightarrow \mathcal{D}(spec), \]

be an observational mapping where \( S \subseteq \text{At}(P) \). Let

\[ M = [\{P\}]_{\text{STeC}} \times [\{spec\}]_{\text{CCSL}}, \]

be the synchronized product of the STeC model and the
CCSL specification. Then we have

\[ P \models spec \text{ iff } M \models \text{AFF}_M, \]

where \( F_M \) is the set of accepting states of \( M \). The CTL for-
formula \( \text{AFF}_M \) means that “every path (starting from the initial
state) will finally reach a state in \( F_M \).”

\( \text{AF} \) is a CTL operator. \( \text{AF} \) means that “for all paths, fi-
nally…” Refer to [25] for more details.

The form \( M \models \text{AFF}_M \) is decidable by the conventional
model checking algorithm for CTL when \( M \) is a finite-state
TA. So we often require that \( spec \) is a safe specification (as
indicated in Section 2.2.3).

Figure 18 gives an intuitive explanation of how our scheme
works. Echoing Fig. 12, we now get a better picture of how
CCSL clock can observe STeC events by taking synchro-
nized product (\( \otimes \)) in their TA models. STeC events synchro-
nize with their observational counterparts in a TA of primi-
tive constraints. The shared clocks between different primitive
constraints are synchronized by taking the compositional
product (\( \otimes \)) between CCSL TA.

![Fig. 18 A detail picture in M](image-url)

We now give a proof of Theorem 4.3.

**Proof (Proof of theorem 4.3) \( \Rightarrow \).** If \( P \models spec \), for any
\( \tau r \in \text{Traces}_{\text{STeC}}(P) \), from Definition 4.11, the mapping

\[ f : \text{dom}(\tau r|_S) \rightarrow \bigcup_{a \in D_{\text{spec}}} I_a, \]

is a trace in some selected \( TS \in \mathcal{T}S_{\text{spec}} \). According to The-
orem 4.2 and Theorem 4.1, there is a corresponding trace,
denoted by \( \overline{tr} \), of \( tr \) in \( \text{Traces}_{TA}(\{P\}_{STeC}) \), and a set of corresponding traces, denoted as \( \overline{f} \), of \( f \) in \( \text{Traces}_{TA}(\{\text{spec}\}_{CCSL}) \). When we consider trace \( tr \) running in \( \{P\}_{STeC} \times \{\text{spec}\}_{CCSL} \), there must exist a trace \( tr' \) in the TA \( \{\text{spec}\}_{CCSL} \) that can make synchronization with \( tr \). Easy to see that \( tr' \) must be in \( \overline{f} \) so it is in the TA \( \{\text{spec}\}_{CCSL} \). Therefore \( M \models AFF_M \) (\( \equiv \)). On the other direction, consider a trace \( tr \in \text{Traces}_{TA}(M) \). Since \( M \models AFF_M \) so we have \( tr_s \in \text{Traces}_{TA}(\{P\}_{STeC}) \) and \( tr_{\{\text{spec}\}} \in \text{Traces}_{TA}(\{\text{spec}\}_{CCSL}) \).

From Theorem 4.2 and Theorem 4.1, there is a corresponding trace, denoted by \( tr_{\{\text{spec}\}} \) of \( tr_{\{\text{spec}\}} \) in a some \( TS \in TS_{\text{spec}} \), and a corresponding trace, denoted by \( tr_s \) of \( tr_s \) in \( \text{Traces}_{STeC}(P) \).

Compare \( tr_{\{\text{spec}\}} \) and \( tr_s \) with \( tr_{\{\text{spec}\}} \) and \( tr_s \) we soon find out \( tr_{\{\text{spec}\}} \) is exactly the mapping

\[
tr_{\{\text{spec}\}} : \text{dom}(tr_s) \rightarrow \bigcup_{a \in \{\text{spec}\}} I_a
\]

we require. So \( P \models \sigma \).

4.8 Complexity analysis of STeC/CCSL framework

We give a general analysis of the computation complexity of STeC/CCSL framework. We prove that the verification on our proposed framework is “almost” as efficient as the traditional model checking techniques, since it can be proved that the complexity of the translation process stays below the bound of the complexity of model checking algorithm itself.

Given any STeC process \( P \) and specification \( \text{spec} \), let \( n(\text{spec}) \) be the number of constraints in \( \text{spec} \). Let \( St(P) \) be the set of configuration regions of \( P \) and \( Tr(P) \) be the set of transitions between regions of \( P \). Let \( pt(P) \) be the set of transitions that start from \( P \). Let \( l(P) \) be the length of formula \( P \), defined as: \( l(P) = 1 \) where \( a \in \mathcal{A}_{STeC} \); \( l(Q \ast R) = 1 + l(Q) + l(R) \) where \( * \in \{,] , [\}, \\}, \{\} \}. \) Let \( sub(P) \) be the set of all subformulas of \( P \), defined as: \( sub(a) = \{a\} \) where \( a \in \mathcal{A}_{STeC} \); \( sub(Q \ast R) = \{Q \ast R\} \cup sub(Q) \cup sub(R) \) where \( * \in \{,] , [\}, \\}, \{\} \} \}

For \( \text{CCSL}_{TA}(\text{spec}) \), we can compute the complexity of the composition \( \otimes \) of all TA \( R_{Ca} \) based on Algorithm 2 in [19] (page 8), where each \( R_{Ca} \) can be converted into a finite automata associated with state invariants. The worst case is \( O(n^{(s(\text{spec}))}) \), where \( t \) is the maximal number of transitions in converted \( R_{Ca} \). It is a constant as \( R_{Ca} \) has fixed number of states and transitions according to Table 3. Thus the worst case for \( \text{CCSL}_{TA}(\text{spec}) \) is \( O(n(\text{spec}) + pt(\text{spec})) = O(n(\text{spec})) \).

The complexity of \( \text{STeC}_{TA}(\text{spec}) \) is a bit complex. Let the computation time of \( \text{Build}_{TA}(A, \{P, l, t\}) \) and \( \text{Trans}([P, l, t]) \) be \( B(P^2) \) and \( T(P) \) respectively. Let \( P = P_1 \bowtie P_2 \bowtie \cdots \bowtie P_n \), \( pt = \max(pt(P_1), pt(P_2), \ldots, pt(P_n)) \), \( Tr = \text{max}(Tr(P_1), Tr(P_2), \ldots, Tr(P_n)) \). \( B(P) \) can be computed as:

\[
B(P) = T(P) + |pt(P)| + \Sigma_{P \rightarrow P'} B(P')
\]

Easy to see that \( \Sigma_{P \rightarrow P'} |pt(P')| = |Tr(P)| \). For the first sum \( \Sigma_{P \rightarrow P'} T(P') \), we consider two conditions of \( T(P) \). If \( P = Q \bowtie R \) where \( * \neq \bowtie \), according to Algorithm 4, the worst condition is \( Q \parallel R \), thus we have:

\[
T(P) = T(Q \parallel R) = T(Q) + T(R) + |pt(Q)| + |pt(R)|.
\]

If \( P = Q \bowtie R \), we have:

\[
T(P) = T(Q) + T(R) + |pt(Q)| \cdot |pt(R)|.
\]

so in \( B(P), \Sigma_{P \rightarrow P'} T(P') \) are divided into two parts:

\[
\Sigma_{P \rightarrow P'} T(P') = \Sigma_{P \bowtie Q \bowtie R} T(P_1) + \Sigma_{P \bowtie Q \bowtie R} T(P_2)
\]

\[
\leq \frac{n(n - 1)}{2} \cdot pt^2 + \Sigma_{P \bowtie Q \bowtie R} T(P_2) + \Sigma_{i \in 1, \ldots, n} T(P_i)
\]

\[
= \frac{n(n - 1)}{2} \cdot pt^2 + \Sigma_{P \bowtie Q \bowtie R} T(P').
\]

In \( \Sigma_{P \bowtie Q \bowtie R} T(P') \), the same \( P' \) only needs to be computed for once. We expand \( \Sigma_{P \bowtie Q \bowtie R} T(P') \) and get:

\[
\Sigma_{P \bowtie Q \bowtie R} T(P') = \Sigma_{S \in sub(P), S \bowtie Q \bowtie R} T(S)
\]

\[
= \Sigma_{S \in sub(P), S \bowtie Q \bowtie R} T(S) + \Sigma_{S \in sub(P), S \bowtie Q \bowtie R} |pt(S)|
\]

\[
\leq \Sigma_{S \in sub(P), S \bowtie Q \bowtie R} |pt(S)|
\]

Note that here \( \Sigma_{S \in sub(P), S \bowtie Q \bowtie R} |pt(S)| \leq \Sigma_{P \rightarrow P'} |pt(S)| \) for \( i \in 1, \ldots, n \). Usually, we can think \( |pt(S)| \leq |St(P)| \). In the worst case, \( Tr(P) = Tr^t \). Thus

\[
\frac{n(n - 1)}{2} \cdot pt^2 + n \cdot Tr \leq n^2 \cdot Tr^t + n \cdot Tr
\]

\[O(Tr^t) = O(Tr(P)).\]

So we have

\[
B(P) \in O(2 \cdot |Tr(P)| + |St(P)|) = O(|Tr(P)| + |St(P)|).
\]

So the upper bound of \( \text{STeC}_{TA}(\text{spec}) \) is \( O(|Tr(P)| + |St(P)|) \).

According to [12] we know that the complexity of our model checking algorithm for STeC/CCSL framework is \( O([AFF_M] \cdot |St(P)| \cdot |CCSL_{TA}(\text{spec})| + |Tr(P)| \cdot |pt(\text{spec})|) \), where let \( c \) be the maximum number of states of \( R_{Ca} \). Thus the total complexity of STeC/CCSL framework is the complexity of the...
translations adding the complexity of model checking, which is:

\[ O(|\text{spec}|) + |\text{fr}| + |\text{lst}| + |\text{AFF}_M| \cdot (|\text{lst}| + |\text{fr}|) = O(|\text{lst}| + |\text{fr}| + |\text{fr}| + |\text{fr}|). \]

The complexity of translation procedures stay within the bound of the complexity of the traditional model checking algorithm.

4.9 Model checking framework for infinite traces

In previous sections we only deal with processes whose traces are finite, due to the fact that STeC has no recursions defined in its syntax. However, our theory also can be applied to models with infinite traces, with only a few modifications to make.

In TA theory, according to [23], an infinite trace

\[ tr = \langle a_1, t_1 \rangle \langle a_2, t_2 \rangle \cdots \langle a_n, t_n \rangle \cdots \]

is in TA if \( A = (Q, \Sigma, C, i, Ed, I, AP, L, F) \) if there is a transition

\[ \langle i, v_i \rangle \xrightarrow{a_1, t_1} \langle q_1, v_1 \rangle \xrightarrow{a_2, t_2} \cdots \xrightarrow{a_n, t_n} \langle q_n, v_n \rangle \cdots \]

in \( A \) such that for any \( N \in \mathbb{N}^+ \), there exists a \( i \geq N \) such that \( q_i \in F \).

In a time structure \( TS \), an infinite trace is defined just as in Section 2.2.3.

Suppose that we have a normal process \( P \) of some modelling language called \( PA \), with the set of finite or infinite traces \( \text{Traces}_{PA}(P) \), and somehow we can translate it into its equivalent TA \( [P]_{PA} \) and propose a new definition and theorem similar to Definition 4.18 and Theorem 4.2. Then all the other definitions and theorems of our theory proposed above still work in the case of infinite traces by just replacing \( \text{Traces}_{TA}^U \), \( \text{Traces}_{TS} \) and \( \text{Traces}_{STeC} \) with their counterparts for infinite traces, except for the Theorem 4.3, where the model checking scheme combining PA and CCSL can be dealt with the satisfaction

\[ M = [P]_{PA} \times [\text{spec}]_{CCSL} \models \text{AG(AFF}_M). \]

\( \text{AG(AFF}_M) \) means that for each path in \( M \), it will eventually pass the state in \( F_M \) for infinitely many times.

In some model checking tools, like UPPAAL, the CTL formula where the path quantifiers are nested is not allowed (e.g., \( \text{AG(AFF}_M) \)). To solve this problem we can take an alternative approach by checking

\[ \text{AG(}\neg\text{deadlock in } M). \]

which means that “for every path, and every state in each path, there is no deadlock in \( M \).” \text{deadlock} is a special expression in UPPAAL, to judge if in a state, deadlock can happen. Though they are not equivalent, but in most cases\(^3\), we can use the latter in place of the former. And this is the approach used in [17] for checking safety properties in UPPAAL. However, this formula actually can not replace the formula \( \text{AFF}_M \) in the case of finite traces, since in a TA where there is no infinite trace there is always a deadlock. This is also why we do not take this approach for our purpose in this paper.

5 Model checking in UPPAAL

In this section, we apply our model checking scheme of STeC/CCSL in UPPAAL.

UPPAAL is a verification tool based on TA theory, its modelling language, as an extension of TA with discrete data, offers additional features such as bounded integer variables and urgency states. A UPPAAL system is a network of TA which run concurrently with handshakes of each other through channels. UPPAAL TA offers many variable types besides clocks and channels, like bounded integers, booleans, etc. Channels are for synchronizing parallel automata. Guards have the same meaning as in TA. There are update functions on edges of UPPAAL TA, which update the values of variables like clocks and integers. Behrmann [16] gives a full guide.

From the theory proposed last section, given a STeC process \( P \), a CCSL specification \( \text{spec} \) and a mapping \( \langle \cdot \rangle \), we can check \( P \models UPL \text{ spec} \) in UPPAAL following several steps listed below:

1. Translate \( \text{spec} \) into UPPAAL TA, denoted by \( \langle \text{spec} \rangle_{UPL} \), based on Definition 4.15.
2. Let \( P = P_1 \parallel P_2 \parallel \cdots \parallel P_n \). Translate each \( P_i \) into UPPAAL TA, denoted by \( \langle P_i \rangle_{UPL} \), based on Theorem 4.2, Definition 4.19 and Proposition 4.1.
3. \( \langle \text{spec} \rangle_{UPL} \) and each \( \langle P_i \rangle_{UPL} \) form a network of UPPAAL TA in UPPAAL, for this network, we check

\[ \langle P_1 \rangle_{UPL}, \ldots, \langle P_n \rangle_{UPL}, \langle \text{spec} \rangle_{UPL} \models \bigwedge_{i=1}^{n} A \leftrightarrow P_i. \text{Accept}, \]

based on Theorem 4.3.

Here “\( A \leftrightarrow \)” is a CTL operator in UPPAAL, it has the same meaning as \( \text{AF} \) in Theorem 4.3. “\( P_i.\text{Accept} \)” corresponds the accepting states in \( \langle P_i \rangle_{UPL} \). Since in \( \langle \text{spec} \rangle_{UPL} \) all states are accepting states, so actually we do not need to consider them in the specification.

\(^{3}\) To be exact, when there is no deadlock in \( [P]_{PA} \)
From the list above we see that we translate spec into \(\llbracket spec\rrbracket_{UPL}\) as one UPPAAL TA. Readers might wonder why we do not consider translating each \(Cn \in spec\) instead, and let UPPAAL do the lazy-evaluation of the composition at runtime, just like what we do for STeC process? The answer is, \(\llbracket Cn\rrbracket_{CCSL}\) might be a TA with infinite states, though we can make the composition \(\otimes\) theoretically to get a finite TA of spec by lazy-evaluation [19], we can not express an infinite TA in UPPAAL!

When we carry out the translation we need to make some modifications since the UPPAAL TA is different from the general TA model. One of the main differences is that in UPPAAL TA, there is only one type of channel, so we must find a way to distinguish the composition \(\otimes\) between STeC TA and the composition \(\times\) between a STeC TA and a CCSL observer. We list the modifications as below:

1) For each event \(C_n! (C_m?)\) in \(\llbracket P\rrbracket_{STeC}\), we have the channel \(C_m! (C_m?)\) in \(\llbracket P\rrbracket_{UPL}\).
2) For each event \(a\) (not in the form of \(C_m!\)) in \(\llbracket P\rrbracket_{STeC}\), and its corresponding observational event \(\pi\) in \(\llbracket spec\rrbracket_{CCSL}\), we have a pair \(t_a!, t_a?\) in \(\llbracket P\rrbracket_{UPL}\) and \(\llbracket spec\rrbracket_{UPL}\) respectively.
3) If an event \(C_m!\) in \(\llbracket P\rrbracket_{STeC}\) is observed by CCSL constraints, in all \(\llbracket P\rrbracket_{UPL}\) that contains \(C_m!\), we add an urgent transition right after it with a channel \(t_m!\) being triggered on it. And we have a channel \(t_m?\) in \(\llbracket spec\rrbracket_{UPL}\).
4) If an event \(a\) is either an unobservable event in \(\llbracket P\rrbracket_{STeC}\) or a generated event in \(\llbracket spec\rrbracket_{CCSL}\), no labels assigned to their corresponding transitions in UPPAAL TA.

Table 4 gives a graphical illustration for clause 1), 2), 3), 4) given above respectively. “\(\otimes\)” means the urgent state in UPPAAL, a type of state from which any transitions must immediately happen without consuming time.

<table>
<thead>
<tr>
<th>Clause</th>
<th>Transitions in STeC/CCSL</th>
<th>Transitions in UPPAAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1)</td>
<td>(C_m!) (\otimes) (C_m?) (\otimes) (C_m!) (\otimes) (C_m?)</td>
<td></td>
</tr>
<tr>
<td>2)</td>
<td>(a) (\otimes) (a) (\otimes) (a) (\otimes) (a) (\otimes) (a)</td>
<td></td>
</tr>
<tr>
<td>3)</td>
<td>(C_m!) (\otimes) (C_m?) (\otimes) (C_m!) (\otimes) (C_m?)</td>
<td></td>
</tr>
<tr>
<td>4)</td>
<td>(a) (\otimes) (a) (\otimes) (a) (\otimes) (a) (\otimes) (a)</td>
<td></td>
</tr>
</tbody>
</table>

From Theorem 4.3 in fact \(\llbracket spec\rrbracket_{CCSL}\) can only observe the interaction \(C_m!\) in \(\llbracket P\rrbracket_{STeC}\), rather than \(C_m!\) in some \(\llbracket P\rrbracket_{STeC}\). But in 3) above, \(\llbracket spec\rrbracket_{UPL}\) observes the channel \(C_m!\) instead, since we consider the network \(\{\llbracket P_i\rrbracket_{UPL}\}_{i=1,...,n}\) in UPPAAL, not \(\llbracket P\rrbracket_{UPL}\) as a whole, so there is no \(C_m!\) in UPPAAL TA. These two means are equivalent because for CCSL, observing \(C_m!\) or \(C_m!\) makes no difference.

### 6 Case study—verifying IRCS system in UPPAAL

We apply the techniques introduced in previous sections to the verification of IRCS system (in Section 3) in UPPAAL.

We consider three safety properties.

**Property 1**: The train always passes after the gate is closed and before the gate is opened.

Its CCSL specification is as follows:

\[\text{spec}_1 = \{c_{\text{close}} < c_{\text{pass}}, c_{\text{pass}} < c_{\text{open}},\]
\[c_{\text{close}} \text{ alternatesWith } c_{\text{open}}\}\]

with the observational mapping \(\{\} = \{\text{Close} \leftrightarrow \text{close}, \text{Pass} \leftrightarrow \text{pass}, \text{Open} \leftrightarrow \text{open}\}\).

**Property 2**: If the train stops, it must stop at a time point that is 50s later since the train approaches.

Its CCSL specification is as follows:

\[\text{spec}_2 = \{c_{\text{mayStop}} = c_{\text{appr delay}} 50 \text{ on IdealClk},\]
\[c_{\text{stop}} < c_{\text{mayStop}}\}\]

with the observational mapping \(\{\} = \{\text{CAppr} \leftrightarrow \text{appr, Stop} \leftrightarrow \text{stop}\}\). \(c_{\text{mayStop}}\) is a generated clock.

**Property 3**: Once the train approaches, the gate shall close in less than 21 seconds.

Its CCSL specification is as follows:

\[\text{spec}_3 = \{c_{\text{appr delay}} = c_{\text{appr delay}} 21 \text{ on IdealClk},\]
\[c_{\text{appr}} < c_{\text{close}}, c_{\text{close}} < c_{\text{appr delay}},\]
\[c_{\text{appr alternatesWith } c_{\text{appr delay}}}\}\]

with the observational mapping \(\{\} = \{\text{CAppr} \leftrightarrow \text{appr, Close} \leftrightarrow \text{close, Closed} \leftrightarrow \text{close}\}\). \(c_{\text{appr delay}}\) is a generated clock. Note that in this specification event \(\text{Close, Closed}\) are both observed by the clock \(c_{\text{close}}\).

The three specifications can be translated into UPPAAL TA as in the last section shows. Figs. 19–21 show the UPPAAL TA of \(\text{spec}_1, \text{spec}_2\) and \(\text{spec}_3\) respectively. The UPPAAL TA of the process \(T\) and \(G\) are different since we have a different mapping \(\{\}\) for each property. Figure 22 gives an example of train agent \(\llbracket T\rrbracket_{UPL}\) for Property 1.
We verify if these properties hold in UPPAAL by checking two CTL formulas in each case:

\[ A \leftrightarrow T.A 
\]

\[ A \leftrightarrow G.A 
\]

The final results are listed in Table 5, from which we can see that Property 3 does not hold since if the gate is open, it will close itself exactly 21 seconds after the train approaches.

### 7 Conclusion and future work

In this paper, we propose a STeC/CCSL framework for spatio-temporal systems and explore its verification techniques. We build a theory to connect STeC and CCSL in their theories and propose a model checking framework for verifying CCSL properties in STeC models. We propose the theory and algorithms to translate STeC and CCSL into TA and analyze their time complexities. At last we propose the translation from STeC/CCSL into UPPAAL TA and as a case study we show how to verify the IRCS system in UPPAAL.

<table>
<thead>
<tr>
<th>Property</th>
<th>CTL formula</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property 1</td>
<td>( A \leftrightarrow T.A )</td>
<td>Yes</td>
</tr>
<tr>
<td>Property 1</td>
<td>( A \leftrightarrow G.A )</td>
<td>Yes</td>
</tr>
<tr>
<td>Property 2</td>
<td>( A \leftrightarrow T.A )</td>
<td>Yes</td>
</tr>
<tr>
<td>Property 2</td>
<td>( A \leftrightarrow G.A )</td>
<td>Yes</td>
</tr>
<tr>
<td>Property 3</td>
<td>( A \leftrightarrow T.A )</td>
<td>No</td>
</tr>
<tr>
<td>Property 3</td>
<td>( A \leftrightarrow G.A )</td>
<td>No</td>
</tr>
</tbody>
</table>

For the future work, we mainly have a prospect in two directions:

1) In this paper, we use CCSL to express properties concerning time, but not locations. For some system where spaces are of first important, for example, in a track station system, the tracks can be in four directions, they can cross each other forming a complex track network. In such a system, to give a description of relationships between the locations of trains is important, we may need a kind of primitive language to describe geographical information, not just simple relations like “precedence” and “subset”. So our future work may focus on the extension of CCSL to describe geographical relationships of spaces.

2) Since our language is for CPSs, a kind of systems that very much rely on its environment, we may want to consider the uncertainty of environment in systems. For example, in our case study, the traffic condition on the crossing is an uncertain factor of our system, but we ignore it by simply using the internal choice “[[ ]]”. The next step might focus on the extension of our model to a stochastic model (to extend both STeC and CCSL to a probabilistic version), where uncertain factors can be considered. For the verification of such models in practice, we may need the “probabilistic” version of UPPAAL tool—UPPAAL SMC.

### 8 Related works

Several approaches have been proposed combining CCSL as
a specification language with modelling languages in verification of real-time systems.

MARTE [11], as an extension of UML [26], is a general model-based language for modeling and analysis of real-time and embedded systems. The modelling parts provide support required from specification to detailed design of real-time and embedded features. The analyzing parts offer facilities to annotate models with information required to perform specific analysis. CCSL has recently been introduced into MARTE for testing and run-time verification of crucial safety properties of models [15]. However, MARTE is designed as an unified modeling language for developing large systems in practice. Due to the large size and complexity of models it is hard (and sometimes unrealistic) to directly apply model checking techniques with CCSL. To do it people need to extract behaviour of concern from MARTE models and encode it into a more abstract formal language such as CCS or CSP. Contrary to MARTE, STeC is a formal language and aims at modelling systems at a high-level. Its simpler semantics make it easier for composing several distributed agents and analyze the interactive behaviour between them. A trade-off approach may be using STeC to capture an abstract specification while using MARTE to handle the detail designs.

Timed-pNets [27] is a communication behavioural semantic model for distributed systems. Its model is a tree style structure with leaves as distributed agents expressed as LTSs and nodes as communication channels between agents. It supports a hierarchical design pattern either in a top-down or a bottom-up fashion. In Timed-pNets CCSL has been adopted for giving specification of causality relations between events on each leave/node. The correctness of specification on each leave/node can be checked by running simulations on an automated tool called timed square [28]. The verification technique used there is based on simulations, rather than our approach which is based on model checking. But we believe our proposed model checking framework about STeC and CCSL can be easily adapted to Timed-pNets.

He [29] proposed a clock-based modeling language for hybrid systems. As examples, it was used for modeling several CPSs including IRCS discussed in this paper [30]. This model is based on a hybrid clock theory that can be seen as an extension to CCSL with several clock operators for characterizing dynamic features of hybrid systems. It is a novel approach using clocks and clock relations to model systems, which makes it easy to extract the specification of the control component from the specification of the total system and the desire behaviour of the physical component. Nevertheless, using clock expressions as a formal model leads to the lost of ability to express the compositionality of systems, which is provided by the combinators of formal languages like STeC. It still remains to be seen that such models can be equipped with a strong verification support like automatic proving or model checking. Comparing with our approach, we prefer to use clock theory to express safety properties, rather than modeling the whole system.

André and Mallet [31] investigated simulating and checking an Esterel [32] program against a CCSL specification. CCSL constraints were encoded into Esterel code as an observer of an Esterel program, and model checking takes place in Esterel Studio. This approach is much similar to ours. The only difference is that an Esterel observer for a given clock relation programs the negation of the primitive constraint associated with this relation in order to check possible violations. While in this paper we encode a clock relation into an observer that just behaves like it. The violation checking is by the CTL formula we give in Theorem 4.3. Compared with Esterel, which is a synchronous language for reactive systems, we focus on the verification of CCSL in an asynchronous language—STeC, whose synchronization mechanism between events is different from Esterel.

Another work linking CCSL and an asynchronous language was proposed in [33], where a CCSL specification was encoded into a Promela model. Promela [34] is an asynchronous modelling language for modelling and verifying concurrent processes (e.g., distributed systems). Not like STeC, in Promela both synchronous and asynchronous communication via message can be defined. Promela supports formal verifications by model checker SPIN. By encoding CCSL into a Promela model, CCSL specification can thus be checked by SPIN. However since Promela/SPIN is an untimed verification framework so only logical time issues in CCSL have been taken cared. In this paper we stress both logical and chronometric time issues in CCSL and encode it into a more refined TA model. The TA model equipped with real-time information allows us to specify physical time constraints which is important in analyzing high-level abstract models of CPSs.

Acknowledgements This work was supported by the National Natural Science Foundation of China (Grant Nos. 61370100, 61321064), Shanghai Knowledge Service Platform Project (ZP1213), Shanghai Municipal Science and Technology Commission Project (14511100400) and Defense Industrial Technology Development Program JCKY (2016212B004-2).

Specially thank Professor Hengyang Wu, who gave us many usable proposals and found out many syntax errors in this paper. Also thank all reviewers for their time to carefully read this paper and give their valuable questions and suggestions.
References


Yuanrui Zhang is a Phd student in the School of Computer Science and Software Engineering, East China Normal University, China. He received his BS degree in pure and applied mathematics, and his MS degree in computer science. His current research interests are verification of real-time systems, interactive proving theory and its
application, formal modelling and verification of cyber-physical systems. Now he is working on verification of CCSL specifications using logical approach.

Frédéric Mallet is a full professor in the Informatics Department, University of Nice Sophia Antipolis, France. He is also a member of the KAIROS team-project, a joint team between the I3S laboratory (UMR CNRS) and the INRIA research center Sophia-Antipolis Méditerranée. His current research interests focus on modelling, simulation and verification of real-time and embedded systems, model-driven engineering, parallel and distributed computing, computer architecture, modelling and verification of cyber-physical systems. Professor Mallet is one of co-inventors of CCSL language and a contributor to Time Square, a simulation tool for CCSL. He was deeply involved as a voting member of MARTE RTF for the definition of the Time and allocation sub-profiles.

Yixiang Chen is a full professor in the School of Computer Science and Software Engineering, East China Normal University, China. Where he is coordinating trustworthy software, Internet of things and Human-Cyber-Physical System related research activities. Professor Chen is the director of the MoE Engineering Research Center for Software/Hardware Co-design Technology and Application. He is a Vice-Chairman of Technical Committee for Embedded System China Computer Federation.