A Verification Framework of Spatio-Temporal Consistent Language STeC based on Timed Automata

Yuanrui Zhang
Institute of Software Engineering,
East China Normal University,
Shanghai, China
Polytech Nice Sophia,
University Nice Sophia Antipolis,
Nice, France
Email: yuanrui1990.zhang@gmail.com

Frederic Mallet
Univ. Nice Sophia Antipolis,
CNRS, I3S, UMR 7271,
06900 Sophia Antipolis, France
(NOTE: Member of Aoste team (INRIA/I3S)
Email: Frederic.Mallet@unice.fr

Yixiang Chen
MoE Engineering Research Center for
Software/Hardware Co-design
Technology and Application,
East China Normal University,
Shanghai, China
Email: yxchen@sei.ecnu.edu.cn

***Abstract*—Intelligent Transportation Systems (ITS) are a class of quickly evolving modern safety-critical embedded systems. Dealing with their growing complexity demands a high-level formal modeling language along with adequate verification techniques. STeC has recently been introduced as a process algebra that deals natively with both spatial and temporal properties. Even though STeC has a good abstraction and enough expressive power for modeling ITS in high-level. It does not provide a direct support for verification both in theory and application. We propose to encode STeC models as Timed Automata (abbreviated as TA) and introduce CCSL as its specification language on its abstract level to provide such a support. We illustrate our verification strategy on a simple ITS example.**

I. Introduction

Intelligent Transportation Systems (ITS) are a class of time-sensitive and safety-critical systems. Dealing with their safety issues and their growing complexity on both architecture and communication aspects demands high-level though formal specification models, adequate verification languages along with adequate verification tools. In history, many formal methods have been developed over the last two decades. They are classified according to two categories: untimed and timed formal methods. The former includes many early works in the area of process algebra, automata theory and classic model checking theory. The latter mainly are the timed versions of different classes of formal languages.

In the field of process algebra, CSP ([1], [2]) was firstly proposed by Hoare in 1978 as the first process algebra modeling the communication behaviour in concurrent systems. π-calculus [3], simular to λ-calculus, is a rewriting languages describing and analyzing properties of concurrent computations. Later, Timed-CSP [4] was proposed in 1995 as a timed version of CSP where a new operator 'WAIT' is added.

In automata theory, Robin and Scott initially proposed the concept of finite automata in 1959. Timed Automata (TA) [5] was introduced by Alur and Dill in 1994.

* Yixiang Chen: The Communication author

It has inspired a lot of works as they introduced real-valued clocks thus making 'time' a first-class citizen of the specification languages. Hybrid automata was proposed in 1996 by Henzinger.

LTL (Linear Temporal Language) and CTL (Computational Tree Language) are prominent specification languages proposed in the field of model checking. TCTL was proposed as the timed version of CTL which is used in the verification of Timed Automata. In 2005, the IEEE PSL (Property Specification Language) was proposed.

While in untimed formal models, only the description of functionality of behaviours of system is stressed, in timed formal models, the real-time performance of behaviours is stressed. Each behaviour is required to be fired at the right time, time correctness becomes as important as functional correctness. However, for some classes of systems (especially Intelligent Transportation Systems), behaviours are usually required to be fired not only at the right time but also the right location. So they are not only time-sensitive but spatio-time sensitive. For example, in a 'Railroad Crossing System' (see Fig. 19) that will be introduced later, a train is always required to receive a message from gate at the right location and time (more examples are available in [6]). We believe that such types of systems demand a new formal modeling language that stresses the spatio-temporal consistence of behaviours of systems in its semantics. For this purpose, STeC [6] (Spatio-Temporal Consistence Language) was proposed as a process algebra to build spatio-temporal models.

As computer systems are growing on size and complexity, high-level specification engineering models are becoming more and more important and are widely used through industry, like AADL, SysML and UML (the Unified Modeling Language). Such models should have a proper formal semantics if verification is intended. CCSL [7] (The Clock Constraint Specification Language) was initially proposed as a companion language of UML profile for MARTE, but later had been

developed as an independent formal specification language. CCSL provides formal semantics support for UML models, with its abstract clock models and progressive refinement. In CCSL, logic clock is understood as a sequence of occurences, and the logical, both causal and temporal, relationships between behaviours is described as 'CCSL constraints' that restrict the way the logical clock can tick, i.e., the instants at which the events may occur. Logical clocks are not only good for modeling sequences of occurring events, but also sequences of properly ordered locations (like tracks in ITS). This is this particular property that we further study here.

Formal verification is used in proving the correctness of highly-critical systems with respect to a certain formal specification (or property). Over the past years, many verification techniques have been proposed and developed, along with verification tools. UPPAAL is one of widely accepted verification tools based on Timed Automata theory.

Rigorously speaking, CCSL is not equivalent to pure Timed Automata. In [8], the comparison between CCSL and PSL is analyzed. [9] gives a state-base representation for CCSL. [10] gives an encoding from part of CCSL (called safety CCSL) into Timed Automata. Later we explain why such an encoding is essential for verification purpose.

In this paper, we propose a new verification framework where we use STeC as a formal system-modeling language, and CCSL for the specification of properties of system. We introduce CCSL as a specification language in the domain of STeC. Then we focus on the encoding from STeC into TA, with it we apply UPPAAL as a verification tool on TA to verify properties.

Section 2 gives the academic background that are essential for introducing the contributions of this paper. In Section 3 we propose a verification framework of STeC and CCSL. In Section 4, we introduce STeC traces and introduce CCSL as a specification language in the domain of STeC. In Section 5, we focus on the translation from STeC into Timed Automata for verification purpose. In Section 6, we use a simple example of Intelligent Transportation Systems to illustrate models and verification techniques under the new verification framework. Section 7 summarizes the results.

II. Academic Background

In this section, some basic definitions about STeC, CCSL and TA are introduced. These definitions are necessary for understanding the contribution part. For the original definitions and more details, see [5], [6], [11], [12] and [13].

A. Introduction of STeC

Yixiang Chen introduced the Spatio-Temporal Consistence Language in 2010 [6] and set up its formal semantics with his colleagues in [14]. Unlike the traditional formal modeling languages, it stresses the location and the time of an event of spatio-time sensitive systems, especially ITS. An event (or action) is fired at a given location and a given time. For example, in a Train Scheduling System (see [6], [14]), trains travel from one platform to another with very strict time

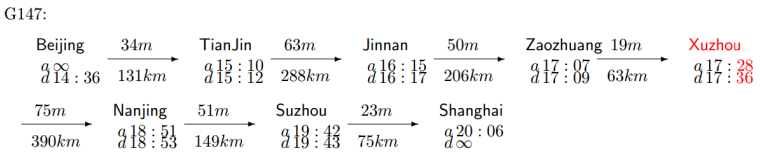

Fig. 1. Scheduling Table of Train 'G147'

restrictions. Fig. 1 shows a scheduling table of train 'G147', where we see that train 'G147' departs from 'Beijing' at 14:36, runs for 131km and arrives at 'Tianjin' at 15:10, stops for 2 minutes and departs at 15:12,.... If a train does not arrive at station at the expected time, then collisions might happen. This characteristic of STeC makes it a powerful and convenient formal modeling language for ITS (for more examples, see [6], [14]). However, STeC does not provide at the moment any support for formal verification.

1) Syntax of STeC: Like other process algebras, in STeC we use 'action' to describe behaviours (or events) of a system, as a set of communicating agents. An 'action' in STeC consists of an action name (also called 'alphabet'), a start location, a start time point, a duration of time and a end location (if an action depends on the position of the Agent). For example, let us consider the following statement:

'the train "G" approaches at location "Lapp" at time t during δ time units'

where 'G' is the name of train, 'Lapp' is a location, t is a time and δ is a duration. It is stated in STeC like this:

$$Approach^{G}_{(Lapp,t)}(\delta)$$

where 'Approach' is an action name (or alphabet), the action 'Approach' is fired at location 'Lapp' and time t, and lasts δ units.

Consider now a second example a bit more complex:

'After receiving Message "Cross" at location "Lpass" and at time t, the train passes the crossroad in d units of time'

This is stated in STeC as follows:

$$\mathbf{Get}_{(Lpass,t)}(Cross); Pass_{(Lpass,t)}(d)$$

(Note: the name of agent 'G' is always omitted.)

Action **Get** is call a 'communication' action, and is aimed at communicating with other agents. ';' is a binary operator called 'sequence', meaning that action **Get** is fired, then action $Pass$ is fired. The composition of actions is an action in STeC. There are many other binary operators in STeC.

Informally, in STeC we call an action 'atomic' if it can not be divided into several 'smaller' actions. In the example above, **Get** and $Pass$ are atomic actions, the composition of them are not atomic.

The formal definition of syntax of STeC is given below in Backus-Naur Form:

$$A ::= \mathbf{Send}^{G\rightharpoonup G'}_{(l,t)}(m) \mid \mathbf{Get}^{G\leftharpoondown G'}_{(l,t)}(m)$$

$$B ::= \alpha_{(l,t)}(l',\delta) \mid \beta_{(l,t)}(\delta)$$

$$P ::= \mathbf{Stop}^G_{(l,t)} \mid \mathbf{Skip}^G_{(l,t)} \mid A \mid B \mid$$

$$P;P \mid P \parallel P \mid []_{i\in I} B_i \to P_i \mid P \trianglerighteq_\delta P \mid$$

$$B \to P \mid P \trianglerighteq ([]_{i\in I} A_i \to P_i)$$

In STeC, there are two types of atomic actions: type A and type B. Type A is for communication purpose. $\mathbf{Send}^{G\to G'}_{(l,t)}(m)$ means that agent G sends a message m to agent G'. $\mathbf{Get}^{G\leftarrow G'}_{(l,t)}(m)$ means that agent G receives a message m from agent G'. Type B are other actions, α and β are to be replaced by user-defined action names (like $Pass$ or $Approach$ in the previous examples). Action $\alpha^G_{(l,t)}(l',\delta)$ means that the action of agent G is fired at location l and time t, after δ time, agent G jumps to l' from l. While $\beta^G_{(l,t)}(\delta)$ stays in location l after the execution.

$\mathbf{Stop}^G_{(l,t)}$, $\mathbf{Skip}^G_{(l,t)}$ are special atomic processes.

';' is the sequence operator, $P;Q$ means that process P is fired then process Q is fired.

'$\parallel$' is the parallel operator, $P \parallel Q$ means that process P and Q are fired concurrently. While $[]_{i\in I} B_i \to P_i$ is a guard choice, meaning that if B_i holds for some i, then P_i is fired. If several B_i holds at the same time, the choice of P_i is a nondeterministic choice.

'$\trianglerighteq$' is the 'interrupt' operator. $P \trianglerighteq_\delta Q$ behaves as P for up to δ time units and then is interrupted by Q. $P \trianglerighteq ([]_{i\in I} A_i \to P_i)$ initially fires P and is interrupted on occurrence of the atomic command A_i and then proceeds like P_i. Here A_i is an action of type A. If several A_i are fired simultaneously, the choice '[]' here is a nondeterministic choice.

For more details about the syntax of STeC, refer to [6] and [14].

2) The Operational Semantics of STeC: A storage σ is introduced in STeC as the storage of messages between agents. Let $\sigma \subseteq \mathbb{M}$, $\mathbb{M}$ is the set of messages of the form $m_{G\rightarrowtail G'}$. Here we use $\mathbf{Sto}$ to represent the set of storages and it is easy to see that: $\mathbf{Sto} \subseteq 2^{\mathbb{M}}$.

Let $\mathbf{Loc}$ be the set of locations. $\mathbb{R}^+$ is the non-negative real set. An environment is a triple (a,u,σ) where σ is a storage, a is a location and u is time. We use the notation $\mathcal{E}$ to represent the set of all the environments: $\mathcal{E} = \mathbf{Loc} \times \mathbb{R}^+ \times \mathbf{Sto}$.

Let $\mathbb{B} = \{B \mid B$ is action type 'B' in STeC $\}$ (see the syntax of STeC). A function $\mathcal{T} : \mathbb{B} \to \{0,1\}$ is defined as the truth valuation function. We have $\mathcal{T}(B) = 1$ (resp. $\mathcal{T}(B) = 0$) if B is true (resp. false).

A configuration of process P is defined as a tuple $\langle P,(a,u,\sigma)\rangle$. Define $\mathbf{Stconf} = \{\langle P,(a,u,\sigma)\rangle \mid P$ is a process and for some $a,u,\sigma\}$ as the set of all configurations in STeC.

The transition relation of STeC $\hookrightarrow_{STeC}$ is a function:

$$\hookrightarrow_{STeC}: \mathbf{Stconf} \to 2^{\mathbf{Stconf}}$$

Any transition relation is denoted as $\langle P,(a,u,\sigma)\rangle \to \langle P',(a',u',\sigma')\rangle$ iff (if and only if) $\hookrightarrow_{STeC} (\langle P,(a,u,\sigma)\rangle) = \langle P',(a',u',\sigma')\rangle$ holds.

τ is a function that maps each process in STeC to the non-negative real set $\mathbb{R}^+$. It computes the duration time for each process. For example, $\tau(\alpha_{(l,t)}(l',\delta)) = \delta$. For more details, see [6].

The operational semantics of STeC is listed as follows. For more details, refer to [6].

- $\frac{a=l,u=t}{\langle \mathbf{Send}^{G\to G'}_{(l,t)}(m),(a,u,\sigma)\rangle \to \langle E,(l,t,\sigma\cup\{m_{G\to G'}\})\rangle}$
- $\frac{a=l,u=t}{\langle \mathbf{Get}^{G\leftarrow G'}_{(l,t)}(m),(a,u,\sigma)\rangle \to \langle E,(l,t,\sigma\setminus\{m_{G\leftarrow G'}\})\rangle}$
- $\frac{a=l,u=t}{\langle \alpha_{(l,t)}(l',\delta),(a,u,\sigma)\rangle \to \langle E,(l',t+\delta,\sigma)\rangle}$
- $\frac{a=l,u=t}{\langle \beta_{(l,t)}(\delta),(a,u,\sigma)\rangle \to \langle E,(l,t+\delta,\sigma)\rangle}$
- $\frac{a=l,u=t}{\langle \mathbf{Stop}^G_{(l,t)},(a,u,\sigma)\rangle \to \langle E,(l,t+1,\sigma)\rangle}$
- $\frac{a=l,u=t}{\langle \mathbf{Skip}^G_{(l,t)},(a,u,\sigma)\rangle \to \langle E,(l,t,\sigma)\rangle}$
- $\frac{\langle P,(a,u,\sigma)\rangle \to \langle P',(a',u',\sigma')\rangle}{\langle P;Q,(a,u,\sigma)\rangle \to \langle P';Q,(a',u',\sigma')\rangle}$
- $\frac{\langle P,(a,u,\sigma)\rangle \to \langle P',(a',u',\sigma_1)\rangle}{\langle P\|Q,(a,u,\sigma)\rangle \to \langle P'\|Q,(a',u',\sigma_1)\rangle}$
- $\frac{\langle Q,(a,u,\sigma)\rangle \to \langle Q',(a',u',\sigma_2)\rangle}{\langle P\|Q,(a,u,\sigma)\rangle \to \langle P'\|Q',(a',u',\sigma_2)\rangle}$
- $\frac{\langle P,(a,u,\sigma)\rangle \to \langle P',(a_1,u_1,\sigma_1)\rangle,\ \langle Q,(a,u,\sigma)\rangle \to \langle Q',(a_2,u_2,\sigma_2)\rangle}{\langle P\|Q,(a,u,\sigma)\rangle \to \langle P'\|Q',(a',u',\sigma_1\uplus\sigma_2)\rangle}$
- $\frac{\mathcal{T}(B_i)=1 \ \ for\ \ i\in I,\ \langle B_i,(a,u,\sigma)\rangle \to \langle E,(a_i,u_i,\sigma_i)\rangle}{\langle []_{i\in I} B_i \to P_i,(a,u,\sigma)\rangle \to \bigcup_{i\in I}\{\langle P_i,(a_i,u_i,\sigma_i)\rangle\}}$
- $\frac{\langle P,(a,u,\sigma)\rangle \to \langle P',(a',u',\sigma')\rangle \ \wedge\ u'\le\delta}{\langle P\trianglerighteq_\delta Q,(a,u,\sigma)\rangle \to \langle P'\trianglerighteq_\delta Q,(a',u',\sigma')\rangle}$
- $\frac{\langle P,(a,u,\sigma)\rangle \to \langle P',(a',u',\sigma')\rangle \ \wedge\ u'>\delta}{\langle P\trianglerighteq_\delta Q,(a,u,\sigma)\rangle \to \langle Q,(a'',\delta,\sigma'')\rangle}$
- $\frac{\mathcal{T}(B)=1,\ \langle B,(a,u,\sigma)\rangle \to \langle E,(a',u',\sigma'\rangle}{\langle B\to P,(a,u,\sigma)\rangle \to \langle P,(a',u',\sigma')\rangle}$
- $\frac{\mathcal{T}(B)=0,\ \langle B,(a,u,\sigma)\rangle \to \langle E,(a',u',\sigma'\rangle}{\langle B\to P,(a,u,\sigma)\rangle \to \langle \mathbf{Stop}^G_{(l,t)},(a',u',\sigma')\rangle}$
- $\frac{\langle A_i,(a,u,\sigma)\rangle \to \langle E,(a',u',\sigma')\rangle,\ \ for\ \ i\in I}{\langle P\trianglerighteq([]_{i\in I} A_i\to P_i),(a,u,\sigma)\rangle \to \bigcup_{i\in I}\{\langle P_i,(a',u',\sigma')\rangle\}}$

B. Introduction of CCSL

The Clock Constraint Specification Language (CCSL) was initially proposed by Charles Andre and Frederic Mallet in 2008 [7]. It has then evolved as a fully-fledged specification language to augment engineering models (UML amongst others) with a formal causal and timed semantics (see [15]). The rest of this section focuses on basic concepts. For more details, see the relative references.

1) Clock: In CCSL, a clock is a ordered sequence of events (or behaviours). The logical relationships between clocks are binary operators between clocks. Here is an example:

'event a is always fired before event b'

which is denoted as $c_a \prec c_b$ in CCSL. Fig. 2 shows an equivalent (infinite) automata of it, which contains an error state. In Fig. 2, the only possible way of getting to the error state is to have a scenario where at a given instant, b has occurred more often than a.

Fig.3 shows a possible trace of $c_a \prec c_b$ that is generated by Timesquare [16]—an analysis framework dedicated to CCSL.

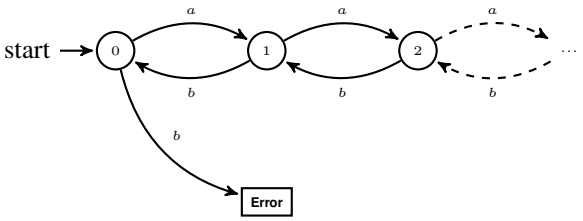

Fig. 2. The automata(infinite) of $c_a \prec c_b$

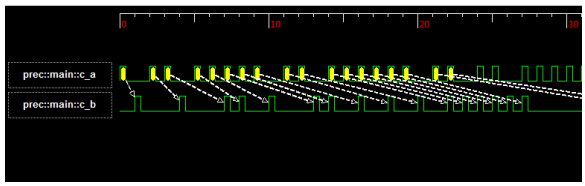

Fig. 3. A possible trace of $c_a \prec c_b$

Another example is the concept of 'subclock' in CCSL:

'event a occurs implies event b occurs'

It means that whenever a is fired, b is fired simultaneously. Fig.4 shows the automata (this time it is finite) of $c_a \subseteq c_b$. From Fig. 4, it is clear to see that event a is never fired alone since c_a is a subclock of c_b. Fig. 5 shows a possible trace.

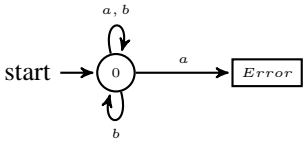

Fig. 4. The automata of $c_a \subseteq c_b$

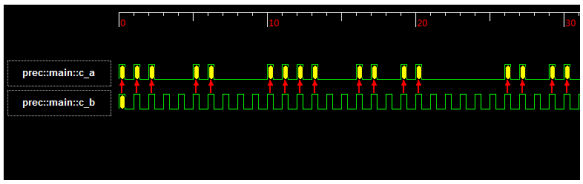

Fig. 5. A possible trace of $c_a \subseteq c_b$

Formally, a Clock in CCSL is a tuple $\langle \mathcal{I}, \prec, \mathcal{D}, \lambda, \mu \rangle$, where $\mathcal{I}$ is a set of instants(possibly infinite), $\prec$ is a quasi-order relation on $\mathcal{I}$, $\mathcal{D}$ is a set of labels, $\lambda : \mathcal{I} \to \mathcal{D}$ is a labelling function, μ is a symbol, standing for a 'unit' of time.

A discrete-time clock in CCSL is a clock with a descrete set of instants $\mathcal{I}$. Since $\mathcal{I}$ is discrete, it can be indexed by natural numbers in a way that respects the ordering on $\mathcal{I}$: set $\mathbb{N}^+ = \mathbb{N} \backslash \{0\}$, $idx : \mathcal{I} \to \mathbb{N}^+$, $\forall i \in \mathcal{I}, idx(i) = k$ if and only if i is the k^{th} instant in $\mathcal{I}$.

Let $c = \langle \mathcal{I}, \prec, \mathcal{D}, \lambda, \mu \rangle$ be a CCSL clock. $c[k]$ denotes the k^{th} element in $\mathcal{I}$, then we have $k = idx(c[k])$. Let $\lambda_i = \lambda(c[i])$.

2) Clock Constraint: In CCSL, we use 'clock' to denote the occurrence of events in a system, binary operations between clocks describe the logical relationship between clocks. They are divided into two basic type of operations, one is called 'Sub Clock' and the other is 'Precedence'. All binary operators in CCSL are derived from these two type of basic operations. For example, binary operations like 'Equality', 'Restriction', 'Discretization', 'Filtering' are derived from 'Sub Clock'. And operations such as 'Speed', 'Alternation', 'Synchronization' are derived from 'Precedence'. For more details, see [7].

'**Sub Clock**' is a binary relation between two clocks which can be defined as this:

Let $c_1 = \langle \mathcal{I}_1, \prec_1, \mathcal{D}_1, \lambda_1, \mu_1 \rangle$, $c_2 = \langle \mathcal{I}_2, \prec_2, \mathcal{D}_2, \lambda_2, \mu_2 \rangle$ be two clocks, we say $c_1 \subseteq c_2$ holds if and only if $\exists h : \mathcal{I}_1 \to \mathcal{I}_2$ such that:

- h is injective.
- h is order preserving: $(\forall i, j \in \mathcal{I}_1)(i \prec_1 j) \to (h(i) \prec_2 h(j))$.
- an instant and its image are coincident: $(\forall i \in \mathcal{I}_1) i = h(i)$.

'**Precedence**' is a binary relation between two clocks which can be defined as follows:

Let $c_1 = \langle \mathcal{I}_1, \prec_1, \mathcal{D}_1, \lambda_1, \mu_1 \rangle$, $c_2 = \langle \mathcal{I}_2, \prec_2, \mathcal{D}_2, \lambda_2, \mu_2 \rangle$ be two clocks, we say 'c_1 precedence' c_2' denoted as $c_1 \prec c_2$ holds if and only if $\exists h : \mathcal{I}_2 \to \mathcal{I}_1$ such that:

- h is injective.
- h is order preserving: $(\forall i, j \in \mathcal{I}_2)(i \prec_2 j) \to (h(i) \prec_1 h(j))$.
- an instant and its image are ordered: $(\forall i \in \mathcal{I}_2)(h(i) \prec i)$.

From these two basic binary operators, we see that the clock relations are only defined on the set of instants $\mathcal{I}$, the domain set $\mathcal{D}$ and function λ are independent from it.

Let $c_A = \langle \mathcal{I}_A, \prec, \mathcal{D}_A, \lambda_A, \mu_A \rangle$ and $c_B = \langle \mathcal{I}_B, \prec, \mathcal{D}_B, \lambda_B, \mu_B \rangle$ be two clocks, $c_A = c_B$ means that the two clocks are 'synchronous', clock c_A ticks whenever clock c_B ticks and vice versa.

c_A **Alternate** c_B means that two clocks ticks 'alternately', in other words, for all $i \in \mathcal{I}_A$, set $k = idx_A(i)$, then we have $c_A[k] \preceq c_B[k] \prec c_A[k+1]$ if c_A and c_B are in the same domain(which means that c_A and c_B is comparable by an operator '$\prec$').

$c_B = c_A$ **delay** n **on** c_C where $n \in \mathbb{N}$ means that clock c_A is delayed n counts of clock c_C comparing with clock c_B, that is to say, after c_A ticks, c_B will tick after n ticks of c_C. Fig. 6 shows the automata of an example. Refer to [7] for formal definition.

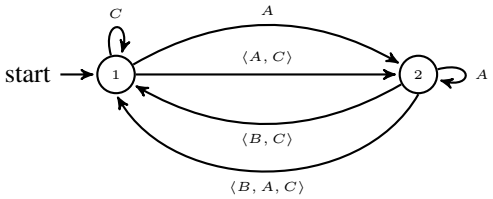

Fig. 6. The automata of $c_B = c_A$ **delay** 1 **on** c_C

$c_B = c_A$ **DiscretizedBy** r where $r \in \mathbb{R}$ creates a discrete-time clock c_B. c_B is a subclock of c_A. We define c_B according to a predicate $P \subseteq \mathcal{I}_A \times \mathcal{I}_B$, which satisfies:

$$(\forall i \in \mathcal{I}_B)(\forall j \in \mathcal{I}_A)(\exists d \in \mathbb{R})P(j,i) = true \text{ iff } \lambda_A(j) = d + (idx_B(i) - 1) \times r$$

c_A is an 'ideal clock', denoted as $c_A =$**IdealClock** iff $\lambda_A : \mathcal{I}_A \to \mathbb{R}^+$ is a bijection.

For the detail of binary operations of CCSL, see [7], and for more other information about CCSL, refer to [8], [9], [11], [15], [17] and [18].

C. Introduction of Timed Automata [5]

Properly speaking, well-defined Timed Automata has the same expressive power as deterministic finite automata (DFA), though it gives 'time' as a particular emphasis since it sets real-valued clocks (not to be confused with the CCSL logical clocks) as first-class citizens. A comprehensive study on the expressiveness of various classes of automata can be found in [19].

1) The Syntax of TA: Let $\mathcal{C}$ be a finite set of non-negative real-valued variables called clocks. The set of guards $G(\mathcal{C})$ is defined by the grammar $g := x \bowtie c \mid g \wedge g$ where $x \in \mathcal{C}$, $c \in \mathbb{N}$ and $\bowtie \in \{<, \leq, >, \geq, ==\}$. A Timed Automata is a tuple $A = (Q, \Sigma, \mathcal{C}, q_0, E, I, AP, L, F)$, where:

- Q is a finite set of states,
- Σ is a finite alphabet,
- $\mathcal{C}$ is a finite set of clocks,
- $q_0 \in Q$ is an initial state,
- $E \subseteq Q \times 2^{\Sigma} \times G(\mathcal{C}) \times 2^{\mathcal{C}} \times Q$ is a finite transition relation,
- $I : Q \to G(\mathcal{C})$ is an invariant-assignment function,
- AP is a finite set of atomic propositions,
- $L : Q \to 2^{AP}$ is a labeling function for the states,
- $F \subseteq Q$ is a set of accepting states.

2) The Operational Semantics of TA: A clock valuation is a function $\nu : \mathcal{C} \to \mathbb{R}$. We define $\nu + r$ as: for each clock $x \in \mathcal{C}$, $(\nu + r)(x) = \nu(x) + r$, where $r \in \mathbb{R}$. If $Y \subseteq \mathcal{C}$ then a valuation $\nu[Y := 0]$ is such that for each clock $x \in \mathcal{C} \backslash Y$, $\nu[Y := 0](x) = \nu(x)$ and for each clock $x \in Y$, $\nu[Y := 0](x) = 0$. The satisfaction relation $\nu \models g$ for $g \in G(\mathcal{C})$ is defined in the natural way.

The operational semantics of a TA $A = (Q, \Sigma, \mathcal{C}, q_0, E, I, AP, L, F)$ is a labeled transition system (LTS):

$$\hookrightarrow_A (q, \nu) : \ (Q \times \mathbb{T}^{\mathcal{C}}) \ \to \ 2^{(Q \times \mathbb{T}^{\mathcal{C}})}$$

where each mapping determines a transition relation $\langle q, \nu \rangle \to \langle q', \nu' \rangle$, which is defined as follows:

$$\hookrightarrow_A (q, \nu) = \begin{cases} \{\langle q, \nu + \alpha \rangle\} & \text{if } \exists \alpha \in \mathbb{R}(\nu + \alpha \models I(q)) \\ \{\langle q', \nu' \rangle \mid q' \in Q'\} & \text{if } \exists \langle q, A, g, Y, q' \rangle \in E \\ & \text{where } \nu \models g, \nu' = \nu[Y := 0] \\ \{\langle q', \nu' \rangle \mid q' \in Q'\} \cup \{\langle q, \nu + \alpha \rangle\} & \text{if both} \end{cases}$$

3) Transition Relation Function: The transition relation function $\psi : Q \to 2^Q$ is a function over the states of TA. We declare that for any $q \in Q$, there exists a value of $\psi(q)$ if and only if the set $\{\langle q, A, g, Y, q' \rangle \mid \langle q, A, g, Y, q' \rangle \in E\} \neq \emptyset$. For each TA there is a conrrespondent transition relation function.

III. The Verification Framework of Spatio-Temporal Models

Classic model checking techniques deals with a formal model $\mathcal{M}$ and specification φ (also called properties). As shown in Fig. 7, a formal model is usually described with a process algebra like CSP, CCS and Timed CSP, while a formal formula is usually described as a type of modal logic such as LTL, CTL, TCTL, or PSL. Verification by observers (An observer is different from the automata of formal models since it usually contains 'accept states' from which if an error path can be reached, we say a counter-example that makes our specification failed is found.) consists in building a semantic model for the model, as a transition system, building accepting transition system for the properties and composing both transition systems. The transition system for the property is called an observer since it does not have any side effects. Valid behaviors of the model should always lead to an acceptance state in the observer. Otherwise a counter example can be extracted by exhibiting the 'invalid' path. Different algorithms are applied for different concrete ways of building the observers. For example, the verification of a LTL specification consists in encoding it as a buchi automata, and combining it with the transition system of model by doing cartesian product to be an observer. While the verification of a CTL specification is based on an algorithm that traverses all states of transition system of model based on the syntax structure of CTL.

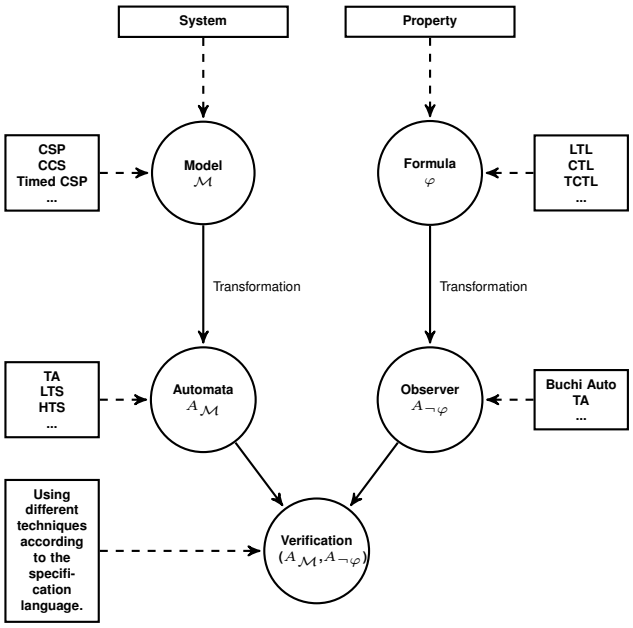

Fig. 7. Classic Verification Framework

Fig. 7 shows the classic verification framework. Our proposed verification Framework (see Fig. 8) follows the same scheme but we use STeC for the specification of the model, since it considers both time and locations as first-class citizens, and CCSL as a specification language. Rather than proposing a direct transformation to transition systems we rather use Timed Automata (TA) as an intermediate formal model. This conveniently allows reusing powerful and widely accepted verification tools like UPPAAL ([20] gives basic verification techniques of TA). CCSL has a different expressive power than classical temporal logics (see [8] for a detailed comparison). It offers a set of pre-defined causal and timed patterns that, in our view, helps abstracting fundamental properties on both time and location without having to deal with the full complexity of temporal logics. [10] describes the family of CCSL specifications that can be encoded as 'pure' Timed Automata. Going through CCSL, compared to describing everything directly as a TA, should alleviate the burden of the design since CCSL provides a library of off-the-shelf often-used property patterns.

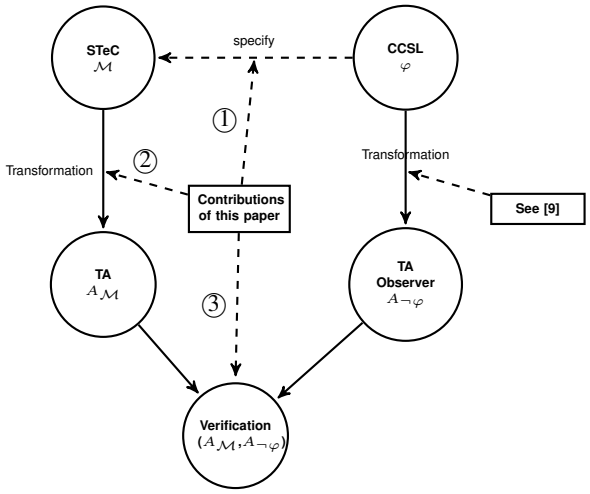

Fig. 8. Proposed Verification Framework

Fig. 8 shows the verification framework we have built in order to verify spatio-temporal specifications of STeC models: i.e., by transforming both STeC and CCSL into Timed Automata(TA). The contributions of this paper are: ① The definition of CCSL in the domain of STeC. ② The encoding from STeC into Timed Automata. ③ The implementation of encoding tool from STeC into TA. We will focus on the first two parts. At last we will illustrate the modeling and verification techniques of the proposed verification framework with a simple ITS example.

For the details of the translation from CCSL into TA, see [10].

IV. The Definition of CCSL in the Domain of STeC

From the previous section, we have known that CCSL, as a specification language that offers logic and timed patterns provides a formal semantic support for the verification of UML. Though CCSL is an independent, formal-defined language, but using it as a specification language in the domain of STeC demands some formal definitions. In this section, we import CCSL into the domain of STeC. The definition of CCSL on STeC is quite trivial, so we focus on the explanation of how a STeC formula 'satisfy' a CCSL formula, which builds a connection between CCSL expressions and STeC formulas. For this purpose we firstly give a definition of traces for STeC in a natural way.

A. Traces on STeC

As usual done for all process algebra, we define the notion of traces in STeC as a sequence of events (or actions) generated by a STeC model(or say a STeC formula). It can be defined inductively on the structure of STeC expressions.

Definition IV.1. *Denote the set of all processes (formulas) of STeC as:*

$$\mathbb{P}_{STeC} = \{\ P \mid\ \ P \text{ is a process in STeC}\}$$

Definition IV.2. *The name of an action or an atomic process defined in Section II is called 'alphabet'. For any atomic process of STeC p, let αp denote its alphabet. Let αP ($P \in \mathbb{P}_{STeC}$) represent the set of alphabets in process P.*

For example, the alphabet of an atomic process $Send_{(Lapp,t)}(Appr)$ is '$Send$', denoted as $\alpha Send_{(Lapp,t)}(Appr) = Send$.

Another example, let $P = Run(\infty); (Send_{(Lapp,t)}(Appr) \parallel Approach_{(Lapp,t)}(\Gamma))$. Then $\alpha P = \{Run, Send, Approach\}$ is the set of alphabets of P.

Definition IV.3. *An element of trace in STeC, called a STeC word, is a quadruple $e = \langle act, l, t, \delta\rangle$, where act is an atomic process in STeC, l is location, t is time and δ is the duration time of the action.*

$\mathbb{E}_{STeC}$ denotes the set of all elements in STeC.

Define the projection of each item in element e as π_{act}, π_l, π_t and π_δ respectively. Generally, let $e = \langle e_1, e_2, ..., e_i, ...\rangle$, we define π_i as the projection on the ith component of e.

For example, $\pi_2(\langle e_1, e_2, e_3\rangle) = e_2$, $\pi_{act}(\langle a_1, l, t, \delta\rangle) = a_1$.

Definition IV.4. $\mathbf{Words}(P)$ *denotes the set of all words of a process P.*

Different from 'actions' in STeC, a 'word' in STeC is a tuple that not only contains the information of the alphabet of an action, but also the location where an action happens, the time at which it occurs and the duration it takes.

For example, let $P = Run(\infty); (\mathbf{Send}_{(Lapp,t,G)}(Appr) \parallel Approach_{(Lapp,t)}(\Gamma))$. Then $\mathbf{Words}(P) = \{\langle Run, l_0, t_0, \infty\rangle, \langle \mathbf{Send}, Lapp, t, 0\rangle, \langle Approach, Lapp, t, \Gamma\rangle\}$.

In this paper, let f is a function, $dom(f)$ denotes the domain of f, $im(f)$ denotes the range of f, and $graph(f)$ denotes the graph of f.

Definition IV.5. *A trace of STeC is a finite or infinite sequence of words which is defined as a partial function $t : \mathbb{N}^+ \rightarrow$*

$\mathbb{E}_{STeC}$ *($\mathbb{N}^+ = \{1,2,3,...\}$ is the set of non-zero natural numbers), where $dom(t) \subseteq \mathbb{N}^+$. It is a partial, injective function and satisfies: for any $n \in \mathbb{N}^+$, if $n \in dom(t)$, then $m \in dom(t)$ for all $m < n$.*

Let t_i be the ith element of trace t, we have $t_i = t(i)$. So sometimes we simply write trace t as $t = t_1 t_2 t_3 ...$, so for any trace t and s, $t = s$ iff $t_i = s_i$ for all $i \in \mathbb{N}^+$.

Denote the set of all traces in STeC as $\mathbb{T}_{STeC}$.

Definition IV.6. *A binary operator $\frown$: $\mathbb{T}_{STeC} \times \mathbb{T}_{STeC} \to \mathbb{T}_{STeC}$ concatenates two traces. Let s, t be any traces in STeC, s is finite. Set $|dom(s)| = m$, then we have a new trace $r = s \frown t$, which satisfies:*

$$for\ all\ x \in \mathbb{N}^+,\quad r(x) = \begin{cases} s(x) & if\ x \le m \\ t(x-m) & otherwise \end{cases}$$

Definition IV.7. *Let $s, t \in \mathbb{T}_{STeC}$, we denote $s \sqsubseteq t$ when s is a sub trace of t. It satisfies:*

1) $im(s) \subseteq im(t)$
2) *for any $t(i), t(j) \in ran(t)$ and $i \le j$, if there exists $m, n \in \mathbb{N}^+$ such that $s(m) = t(i), s(n) = t(j)$, then $m \le n$ holds.*

A restriction on trace is a sub trace of it on some specific alphabet set. We have the following definition.

Definition IV.8. *Let $s, t \in \mathbb{T}_{STeC}$, we say s is a restriction of t on an alphabet set αP if and only if s is a sub trace of t where each element belongs to αP, denoted as $s = t \upharpoonright \alpha P$. It satisfies:*

1) $s \sqsubseteq t$.
2) $im(s) = \{\langle act, l, t, \delta\rangle \mid act \in \alpha P\}$.

After some preparations, we now give a definition of traces for STeC formulas.

Definition IV.9. *A trace of a STeC formula is a function from the set of STeC formulas to a set of STeC traces, denoted as* $\mathbf{Traces} : \mathbb{P}_{STeC} \to \mathbb{T}_{STeC}$. *It is defined according to following lows:*

1) $\mathbf{Traces}(\mathbf{Stop}^G_{(l,t)}) = \emptyset$.
2) $\mathbf{Traces}(\mathbf{Send}^{G \rightarrowtail G'}_{(l,t)}(m)) = \{\langle \mathbf{Send}, l, t, 0\rangle\}$
3) $\mathbf{Traces}(\mathbf{Get}^{G \leftarrowtail G'}_{(l,t)}(m)) = \{\langle \mathbf{Get}, l, t, 0\rangle\}$
4) $\mathbf{Traces}(\alpha^G_{(l,t)}(l', \delta)) = \{\langle \alpha, l, t, \delta\rangle\}$
5) $\mathbf{Traces}(\beta^G_{(l,t)}(\delta)) = \{\langle \beta, l, t, \delta\rangle\}$
6) $\mathbf{Traces}(P;Q) = \{s \frown t \mid s \in \mathbf{Traces}(P) \ \wedge\ t \in \mathbf{Traces}(Q)\}$
7) $\mathbf{Traces}(P \,[]\, Q) = \mathbf{Traces}(P) \cup \mathbf{Traces}(Q)$
8) $\mathbf{Traces}(P \parallel Q) = \{t \mid t \upharpoonright \alpha P \in \mathbf{Traces}(P) \ \wedge\ t \upharpoonright \alpha Q \in \mathbf{Traces}(Q) \ \wedge\ im(t) \subseteq \alpha P \cup \alpha Q\}$
9) $\mathbf{Traces}(P \trianglerighteq_\delta Q) = \{t \mid \exists f \exists g \exists k($
$f \in \mathbf{Traces}(P)$
$\wedge$
$g \in \mathbf{Traces}(Q)$
$\wedge$
$\pi_t(t_i) < \delta$ *and* $\pi_{act}(t_i) = f_i$, *for all* $i \le k$
$\wedge$
$\pi_t(t_i) \ge \delta$ *and* $\pi_{act}(t_i) = g_{i-k}$, *for all* $i > k$
$)\}$
10) $\mathbf{Traces}(P \trianglerighteq ([]_{i\in I} A_i \to P_i)) = \{t \mid \exists f \exists g \exists k \exists i($
$f \in \mathbf{Traces}(P)$
$\wedge$
$g \in \mathbf{Traces}(P_i)$
$\wedge$
$\pi_{act}(t_i) = f_i$, *for all* $i < k$
$\wedge$
$\pi_{act}(t_k) = \alpha A_i$
$\wedge$
$\pi_{act}(t_i) = g_{i-k}$, *for all* $i > k$
$)\}$
11) $\mathbf{Traces}(B \to P) =$
$$\begin{cases} \{t \mid \pi_{act}(t_1) = \alpha B \wedge t_2 t_3 ... \in \mathbf{Traces}(P)\} & if\ \mathcal{T}(B) = 1 \\ \emptyset & otherwise \end{cases}$$

See that $[]_{i\in I} B_i \to P_i$ can be easily considered as a special case of $P[]Q$.

B. The Definition of CCSL for STeC

Next let us define the CCSL in the domain of STeC formally according to the definition of STeC traces.

Definition IV.10. *Let A be a set of alphabets in STeC. $c_A = \langle \mathcal{I}, \prec, \lambda, \mathcal{D}\rangle$ is a clock of A, which satisfies:*

1) *$\mathcal{I}$ is a set of instants, $\prec$ is a quasi-order relation on $\mathcal{I}$ just as defined in the early reference [7]*
2) *$\mathcal{D} = A$*
3) *$\lambda : \mathcal{I} \to \mathcal{D}$ is a labelling function. It is a partial, injective function and satisfies that for any $i \in \mathcal{I}$, if $i \in dom(\lambda)$ then for all $j \prec i$, $j \in dom(\lambda)$. This definition is the same as the definition of λ in [7].*

A clock that satisfies the above rules is said to be in the domain of STeC.

In the definition above, the unit of time μ appearing in the original definition of CCSL in [7] is ignored.

Next we give the definition of the satisfiability of CCSL formula for a STeC formula.

Definition IV.11. *$c_A = \langle \mathcal{I}, \prec, \lambda, \mathcal{D}\rangle$ is a CCSL clock and Exp_c is an expression obtained by combining clocks using binary operators in CCSL. Then for any traces t (of some process) in STeC,*

1) *$t \models c_A$ iff for any $i \in \mathbb{N}^+$, $\pi_{act}((t \upharpoonright A)(i)) = \lambda_i$*
2) *$t \models c_A \subseteq c_B$ iff $t \upharpoonright A \models c_A$, $t \upharpoonright B \models c_B$ and $graph(t \upharpoonright A) \subseteq graph(t \upharpoonright B)$*
3) *$t \models c_A \prec c_B$ iff $t \upharpoonright A \models c_A$, $t \upharpoonright B \models c_B$ and $\pi_t((t \upharpoonright A)(i)) < \pi_t((t \upharpoonright B)(i))$ for any $i \in \mathbb{N}^+$*

For any $P \in \mathbb{P}_{STeC}$, $P \models c_A$ (resp. Exp_c) iff $t \models c_A$ (resp. Exp_c) for all $t \in \mathbf{Traces}(P)$.

V. The Transformation From STeC to TA

In Section III, we choose TA as an intermediate model for the model checking of STeC model because in this way, we can make full use of developed verification tools like UPPAAL. In this section, we mainly focus on the transformation from STeC to TA. We want to build a deductive mapping on the structure of STeC expressions. But firstly, we have to introduce three binary operations over TAs to ease the building of complex TAs by composing simpler ones.

A. New Binary Operators for Timed Automata

Firstly, we introduce some binary operations on TAs, which are essential for the transformation from STeC to TA.

1) STeC-Sequence: We introduce a binary operator over TAs, denoted as $\diamond_{STeC}$. Informally, it 'concatenates' two TAs by adding transitions from one's each final states to the other's initial state. This operator is essential for the sequential composition in STeC (P;Q), see Def. V.5.

Definition V.1. *Let $A = (Q, \Sigma, \mathcal{C}, q_0, E, I, AP, L, F)$ and $A' = (Q', \Sigma', \mathcal{C}', q_0', E', I', AP', L', F')$ be two TAs. We introduce a binary operator over TAs called "STeC-Sequence" and defined as follows:*

$$\diamond_{STeC} : \ \mathbf{TA} \times \mathbf{TA} \ \rightarrow \ \mathbf{TA}$$

where **TA** *represents the set of all TAs. Let $A'' = (Q'', \Sigma'', \mathcal{C}'', q_0'', E'', I'', AP'', L'', F'')$ such that $A'' = A \diamond_{STeC} A'$ and it satisfies:*

- $A'' = (Q \cup Q', \Sigma \cup \Sigma', \mathcal{C} \cup \mathcal{C}' \cup \{y\}, q_0, E'', I'', AP \cup AP', L \cup L', F')$, *where y is a new clock.*
- $E'' = E \cup E' \cup \{\ \langle f, \emptyset, y == 0, \{y\}, i\rangle \mid \ f \in F \wedge i = q_0'\ \}$
- *I'' is defined as follows:*

$$I''(q) = \begin{cases} I(q) \wedge y \leq 0 \quad \text{if } q \in F \\ I(q) \quad \text{if } q \in Q \wedge \{q\} \cap F = \emptyset \\ I'(q) \quad \text{otherwise} \end{cases}$$

2) STeC-Choice: The second introduced binary operator over TAs serves to build the choice operation in STeC. Informally, it combines two TAs by adding a new state, and two transitions from it to the initial states of each TA. It is called 'STeC-Choice' and denoted as $\oplus_{STeC}$.

Definition V.2. *Let $A = (Q, \Sigma, \mathcal{C}, q_0, E, I, AP, L, F)$, $A' = (Q', \Sigma', \mathcal{C}', q_0', E', I', AP', L', F')$ be two TAs. Define*

$$\oplus_{STeC} : \ \mathbf{TA} \times \mathbf{TA} \rightarrow \mathbf{TA}$$

as the "STeC-Choice" where:

- *Let $A'' = A \oplus_{STeC} A'$, then $A'' = (Q \cup Q' \cup \{q_0''\}, \Sigma \cup \Sigma', \mathcal{C} \cup \mathcal{C}' \cup \{y\}, q_0'', E'', I'', AP \cup AP', L \cup L', F \cup F')$, where y is a new clock.*
- $E'' = E \cup E' \cup \{\ \langle q_0'', \emptyset, y = 0, \{y\}, q_0\rangle\ ,\ \langle q_0'', \emptyset, y = 0, \{y\}, q_0'\rangle\}$
- $I'' = I \cup I' \cup \{\langle q_0'', y \leq 0\rangle\}$

3) STeC-Parallel: The third operation is called "STeC Parallel" which serves to build the parallel operation in STeC. Informally, it is like the 'Hand Shaking' operator defined over transition systems, combing two TAs by doing cartesian product. It is denoted as $\otimes_{STeC}$.

Definition V.3. *Knowing $A = (Q, \Sigma, \mathcal{C}, q_0, E, I, AP, L, F)$, $A' = (Q', \Sigma', \mathcal{C}', q_0', E', I', AP', L', F')$ are two TAs. The definition of "STeC-Parallel" is given as follows:*

$$\otimes_{STeC} : \ \mathbf{TA} \times \mathbf{TA} \rightarrow \mathbf{TA}$$

let $A'' = A \otimes_{STeC} A'$, A'' is defined as follows:

- $A'' = (Q \times Q', \Sigma \cup \Sigma', \{x, y\}, \langle q_0, q_0'\rangle, E'', I'', AP \cup AP', L'', F \times F')$
- $E'' = E_1 \cup E_2 \cup E_3 \cup E_4$, *where:*

1) $E_1 = \{\ \langle\langle q_1, q_2\rangle, \{m\}, g_1 \wedge g_2, Y_1 \cup Y_2, \langle q_1', q_2'\rangle\rangle \mid$
$(\langle q_1, \{m!\}, g_1, Y_1, q_1'\rangle \in E \wedge \langle q_2, \{m?\}, g_2, Y_2, q_2'\rangle \in E')$
$\vee$
$(\langle q_1, \{m?\}, g_1, Y_1, q_1'\rangle \in E \wedge \langle q_2, \{m!\}, g_2, Y_2, q_2'\rangle \in E')$
$\}$

2) $E_2 = \{\ \langle\langle q_1, q_2\rangle, A_1 \cup A_2, g_1 \wedge g_2, Y_1 \cup Y_2, \langle q_1', q_2'\rangle\rangle \mid \langle q_1, A_1, g_1, Y_1, q_1'\rangle \in E \wedge \langle q_2, A_2, g_2, Y_2, q_2'\rangle \in E'\ \}$

3) $E_3 = \{\ \langle\langle q_1, q_2\rangle, A_1, g_1, Y_1, \langle q_1', q_2\rangle\rangle \mid \langle q_1, A_1, g_1, Y_1, q_1'\rangle \in E\ \}$

4) $E_4 = \{\ \langle\langle q_1, q_2\rangle, A_2, g_2, Y_2, \langle q_1, q_2'\rangle\rangle \mid \langle q_2, A_2, g_2, Y_2, q_2'\rangle \in E'\ \}$

'm!' and 'm?' are two special alphabets that will be defined in Def. V.5, here we can simply take them as normal alphabet names in TA.

- *L'' is defined as:*

$$L''(\langle q_1, q_2\rangle) = L(q_1) \cup L'(q_2) \text{ for all } \langle q_1, q_2\rangle \in Q \times Q'$$

- *I'' is defined as:*

$$I''(\langle q_1, q_2\rangle) = I(q_1) \ \cup \ I'(q_2) \text{ for all } \langle q_1, q_2\rangle \in Q \times Q'$$

B. From SteC to TA: Induction on the Structure of STeC Specification

After the essential operators have been defined, we can now define a mapping from STeC to TA. We need to define a homomorphism from the set of language STeC to the set of TA inductively.

Definition V.4. *Denote the set of all TAs as:*

$$\mathbb{M}_{TA} = \{\ A \mid A \text{ is a Timed Automata}\}$$

Definition V.5. *Knowing that $\mathbb{P}_{STeC}$ and $\mathbb{M}_{TA}$ are the sets of formulas of STeC and TA respectively. We define a homomorphism $\mathcal{F} : \mathbb{P}_{STeC} \to \mathbb{M}_{TA}$ which can be inductively built according to the following rules:*

i) $\mathcal{F}(\mathbf{Send}^{G \to G'}_{(l,t)}(m)) =$
$(\{q_0, q_1\}, \{m!\}, \{x, y\}, q_0, E, I, AP, L, \{q_1\})$.
$E = \{\langle q_0, \{m!\}, \{x == t\}, \{y\}, q_1\rangle\}$. *$x == t$ is a guard, meaning that only when the global clock x equals t, transition is fired.*
$I = \emptyset$.
$AP = \{l\}$.
$L = \{\langle q_0, l\rangle, \langle q_1, l\rangle\}$.
Fig. 9 shows the graph of the TA.

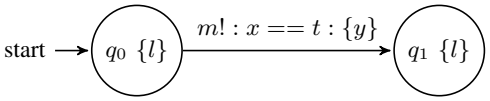

Fig. 9. The TA of $\mathbf{Send}^{G \to G'}_{(l,t)}(m)$

ii) $\mathcal{F}(\mathbf{Get}^{G \leftarrow G'}_{(l,t)}(m)) =$
$(\{q_0, q_1\}, \{m?\}, \{x, y\}, q_0, E, I, AP, L, \{q_1\})$.
$E = \{\langle q_0, \{m?\}, \{x == t\}, \{y\}, q_1\rangle\}$.
$I = \emptyset$.
$AP = \{l\}$.
$L = \{\langle q_0, l\rangle, \langle q_1, l\rangle\}$.
Fig. 10 shows the TA of $\mathbf{Get}^{G \leftarrow G'}_{(l,t)}(m)$.

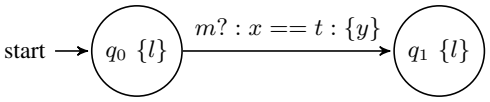

Fig. 10. The TA of $\mathbf{Get}^{G \leftarrow G'}_{(l,t)}(m)$

iii) $\mathcal{F}(\alpha^G_{(l,t)}(l', \delta)) =$
$(\{q_0, q_1\}, \{\alpha\}, \{x, y\}, q_0, E, I, AP, L, \{q_1\})$.
$E = \{\langle q_0, \{\alpha\}, x == t + \delta \wedge y == \delta, \{y\}, q_1\rangle\}$.
$I = \{\langle q_0, y \leq \delta\rangle\}$.
$AP = \{l, l'\}$.
$L = \{\langle l, q_0\rangle, \langle l', q_1\rangle\}$.
Fig. 11 shows the TA of $\alpha^G_{(l,t)}(l', \delta)$.

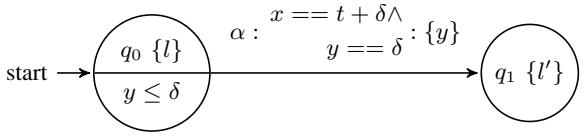

Fig. 11. The TA of $\alpha^G_{(l,t)}(l', \delta)$

iv) $\mathcal{F}(\beta^G_{(l,t)}(\delta)) =$
$(\{q_0\}, x == t + \delta, \{x, y\}, q_0, E, I, AP, L, \{q_0\})$.
$E = \langle q_0, \{\alpha\}, x == t + \delta \wedge y == \delta, \{y\}, q_0\rangle$.
$I = \{\langle q_0, y \leq \delta\rangle\}$.
$AP = \{l\}$.

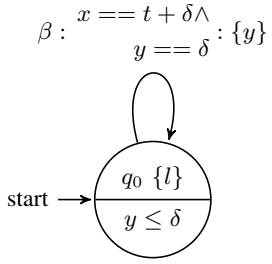

Fig. 12. The TA of $\beta^G_{(l,t)}(\delta)$

$L = \{\langle q_0, l\rangle\}$.
Fig. 12 shows the corresponding TA.

v) For all $R \in \mathbb{P}_{STeC}$, if $R = P \ ; \ Q$, then

$$\mathcal{F}(R) = \mathcal{F}(P; Q) = \mathcal{F}(P) \diamond_{STeC} \mathcal{F}(Q)$$

where $\diamond_{STeC}$ is the 'STeC-Sequence' operator over TAs defined in Def. V.1.
Fig. 13 shows the TA of P and Q respectively. Fig. 14 shows the TA of $P; Q$, where i_P, i_Q are the initial state of $\mathcal{F}(P)$ and $\mathcal{F}(Q)$ respectively. f_P, f_Q are the final state of $\mathcal{F}(P)$ and $\mathcal{F}(Q)$ respectively. More than one final state is allowed.

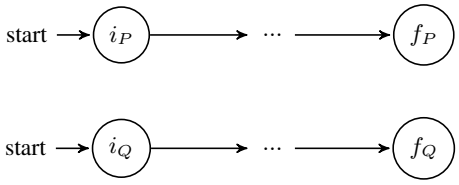

Fig. 13. The TA of P and Q

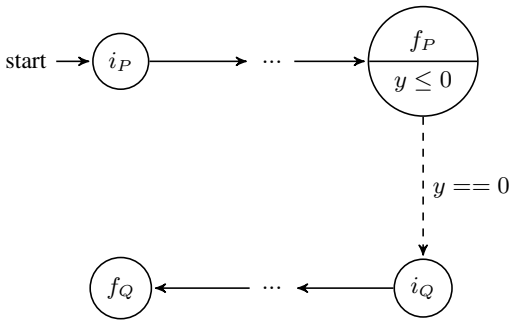

Fig. 14. The TA of $P; Q$

vi)

$$\mathcal{F}(B \to P) = \begin{cases} \mathcal{F}(B) \diamond_{STeC} \mathcal{F}(P) & \text{if } \mathcal{T}(B) = 1 \\ \emptyset & \text{otherwise} \end{cases}$$

vii) For all $R \in \mathbb{P}_{STeC}$, if $R = P \ [] \ Q$, then

$$\mathcal{F}(R) = \mathcal{F}(P \ [] \ Q) = \mathcal{F}(Q) \oplus_{STeC} \mathcal{F}(P)$$

where $\oplus_{STeC}$ is the 'STeC-Choice' operator over TAs defined in Def. V.2.
Fig. 15 shows the TA of $P[]Q$, of which the initial state is the 'new' state.

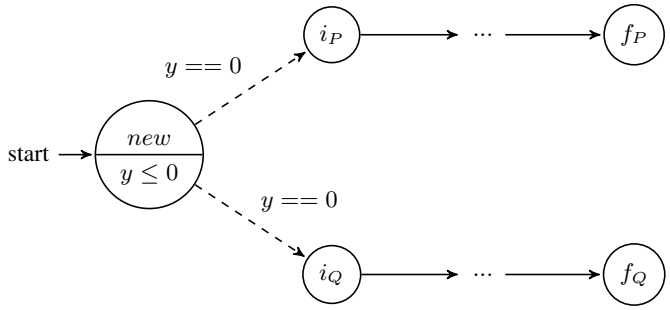

Fig. 15. The TA of $P[]Q$

viii) For all $R \in \mathbb{P}_{STeC}$, if $R = P \parallel Q$, then

$$\mathcal{F}(R) = \mathcal{F}(P \parallel Q) = \mathcal{F}(Q) \otimes_{STeC} \mathcal{F}(P)$$

where $\otimes_{STeC}$ is the 'STeC-Parallel' operator over TAs defined in Def. V.3. And Fig. 16 shows the corresponding TA.

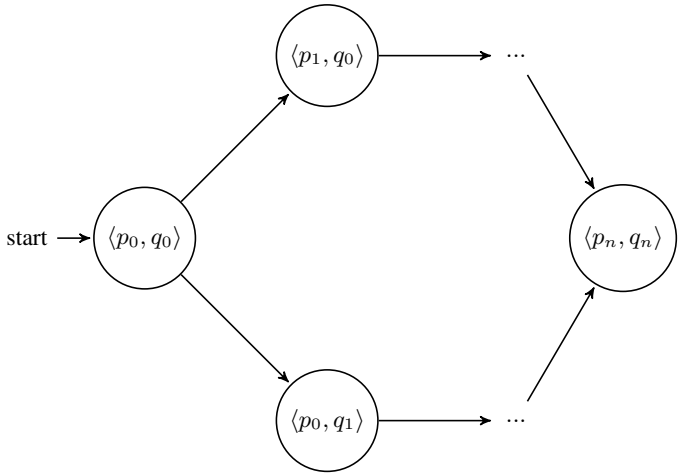

Fig. 16. The TA of $P \parallel Q$

ix) Let $\mathcal{F}(P \trianglerighteq_\delta Q) = (Q, \Sigma, \mathcal{C}, q_0, E, I, AP, L, F) = \mathcal{F}(P) \curvearrowright_\delta \mathcal{F}(Q)$, $\curvearrowright_\delta$ is a binary operator over TAs. Let $\mathcal{F}(P) = (Q_P, \Sigma_P, \mathcal{C}_P, s_P, E_P, I_P, AP_P, L_P, F_P)$ and $\mathcal{F}(Q) = (Q_Q, \Sigma_Q, \mathcal{C}_Q, s_Q, E_Q, I_Q, AP_Q, L_Q, F_Q)$. We define $\curvearrowright_\delta$ as follows:

- $Q = Q_P \cup Q_Q$
- $\Sigma = \Sigma_P \cup \Sigma_Q$
- $\mathcal{C} = \mathcal{C}_P \cup \mathcal{C}_Q \cup \{x\}$. *$x$ is a new clock*
- $q_0 = i_P$ *where i_P is the initial state of $\mathcal{F}(P)$*
- $AP = AP_P \cup AP_Q$
- $L = L_P \cup L_Q$
- $F = F_Q$
- $I = I_P \cup I_Q \cup \{\langle q, x \leq \delta \rangle \mid q \in Q_P\}$
- $E = E_P \cup E_Q \cup \{\langle q, A, g, Y, q' \rangle \mid q \in Q_P \wedge A = \emptyset \wedge g = x \geq \delta \wedge Y = \{y\} \wedge q' = i_Q$ *where i_Q is the initial state of $\mathcal{F}(Q)$.*

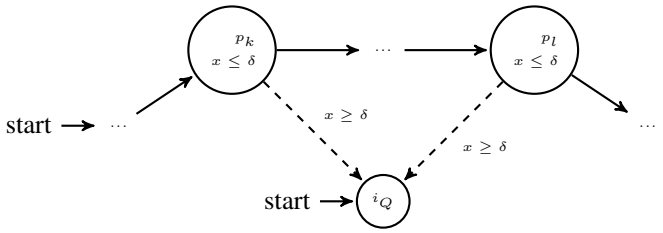

Fig. 17. The TA of $P \trianglerighteq_\delta Q$

Fig. 17 shows the TA of $P \trianglerighteq_\delta Q$ where p_k and p_l are two arbitrary states in $\mathcal{F}(P)$.

x) Let $\mathcal{F}(P) = (Q_P, \Sigma_P, \mathcal{C}_P, s_P, E_P, I_P, AP_P, L_P, F_P)$ and $\mathcal{F}(P_i) = (Q_i, \Sigma_i, \mathcal{C}_i, s_i, E_i, I_i, AP_i, L_i, F_i)$. We define $\mathcal{F}(P \trianglerighteq ([]_{i \in I} A_i \rightarrow P_i)) = (Q, \Sigma, \mathcal{C}, q_0, E, I, AP, L, F)$ as follows:

- $Q = Q_P \cup (\cup_{i \in I} Q_i)$
- $\Sigma = \Sigma_P \cup (\cup_{i \in I} \Sigma_i)$
- $\mathcal{C} = \mathcal{C}_P \cup \mathcal{C}_Q \cup \{x\}$
- $q_0 = i_P$ *where i_P is the initial state of $\mathcal{F}(P)$*
- $AP = AP_P \cup \cup_{i \in I} (AP_i \cup \{m_i*\})$
- $L = L_P \cup (\cup_{i \in I} L_i)$
- $F = \cup_{i \in I} F_i$
- $I = I_P \cup (\cup_{i \in I} I_i)$
- $E = E_P \cup (\cup_{i \in I} E_i) \cup \{\langle q, \{m_i*\}, x == t_{A_i}, \{y\}, q' \rangle \mid q \in Q_P \wedge \exists l_{A_i} \in L(q) \wedge q' = i_{P_i}\}$ *where m_i* is either $m_i!$ or $m_i?$ depending on the type of action A_i (**Send** or **Get**). l_{A_i} and t_{A_i} are the start time and location of action A_i.*

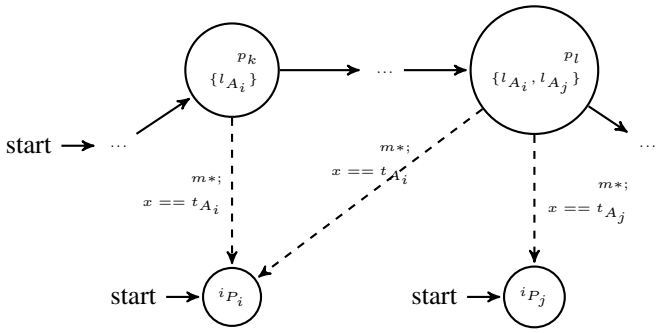

Fig. 18. The TA of $P \trianglerighteq ([]_{i \in I} A_i \rightarrow P_i)$

Fig. 18 shows the combination procedure of $P \trianglerighteq ([]_{i \in I} A_i \rightarrow P_i)$, where p_k and p_l are two arbitrary states in $\mathcal{F}(P)$. i_{P_i}, i_{P_j} are the initial states of any P_i and P_j respectively. From this figure we see that only when location l_{A_i} exists in any state (e.g. p_l) of $\mathcal{F}(P)$, a transition from p_l to i_{P_i} is built (dashed arrow) for any $i \in I$.

Theorem V.1. *The function $\mathcal{F} : \mathbb{P}_{STeC} \to \mathbb{M}_{TA}$ defined in Definition IV.6 is well defined, and it satisfies*

$$dom(\mathcal{F}) = \mathbb{P}_{STeC}$$

Proof: It is obvious according to the definition of $\mathcal{F}$. ■

C. Simplification of Encoded TA From STeC

Last section gives a formal definition of the translation from STeC to TA, alert readers may have noticed that in Def. V.5 the encoded TA may contain some redundant transitions when rule v, vi and vii are applied during the encoding. For example, when using 'STeC-Sequence' operator (rule v) to compose $\mathcal{F}(P)$ and $\mathcal{F}(Q)$ (where $\mathcal{F}(P)$ and $\mathcal{F}(Q)$ are two encoded TAs of P and Q respectively), new transitions from final states of $\mathcal{F}(P)$ to the inital state of $\mathcal{F}(Q)$ (see Fig. 14) are added in the new composed TA. Such transitions are redundant since after state f_P is reached, the transition from it to i_Q is immediately fired. The simplification is to removing all redundant transitions in TA.

We firstly give a formal definition of redundant transitions, then give a formal definition of quotient set of states under a function. At last we give the definition of simplification of TAs.

Definition V.6. *Let $A = (Q, \Sigma, \mathcal{C}, q_0, E, I, AP, L, F)$ be a given Timed Automata. Define $R_A \subseteq E$ as the set of reductant transitions in A, where:*

$$\begin{gathered} R_A = \{\langle q, A, g, Y, q'\rangle \mid \ A = \emptyset \\ \wedge \\ \exists y \ (y \in \mathcal{C} \wedge g = y == 0 \wedge I(q) \models y \leq 0 \wedge y \in Y) \\ \wedge \\ \langle q, A, g, Y, q'\rangle \in E \ \}. \end{gathered}$$

'$\models$' means 'satisfy' that is defined in a natural way shown in Section II. For example, $y \leq 0 \models y \leq 2$, $x \leq 0 \wedge y \leq 0 \models y \leq 0$.

Definition V.7. *Let Q be any set of states of some TA, $\backsim_R \subseteq Q \times Q$ is a relation on Q, then $Q/\backsim_R$ is a quotient set of Q on relation $\backsim_R$ that satisfies:*

$$Q/\backsim_R = \{[q] \mid q \in Q\}$$

where $\backsim_R$ is a equivalence relation. $[q] = \{p \mid \langle p, q\rangle \in \backsim_R\}$ is the equivalence class of q.

Definition V.8. *Given a TA $A = (Q, \Sigma, \mathcal{C}, q_0, E, I, AP, L, F)$, define the simplified TA of A as $\mathbf{Sim}(A) = (Q', \Sigma', \mathcal{C}', q_0', E', I', AP', L', F')$. It satisfies:*

- *$Q' = Q/\backsim_R$, where $\backsim_R \subseteq Q \times Q$ is a relation defined as:*

  $$\begin{gathered} \backsim_R = \{\langle p, q\rangle \mid \ \exists A \exists g \exists Y (\langle p, A, g, Y, q\rangle \in R_A \vee \\ \langle q, A, g, Y, p\rangle \in R_A) \ \} \cup \{\langle p, p\rangle \mid p \in Q\} \end{gathered}$$

  Easy to see that R is a equivalence relation.
- *$\Sigma' = \Sigma$*
- *$\mathcal{C}' = \mathcal{C}$*
- *$q_0' = [q_0]$*
- *$E' = \{\langle [p], A, g, Y, [q]\rangle \mid \langle p, A, g, Y, q\rangle \in E\}$*
- *I' is defined as:*

  $$I'([p]) = \textstyle\bigwedge_{q \backsim_f p} I(q).$$
- *$AP' = AP$*
- *L' is defined as:*

  $$L'([p]) = \textstyle\bigcup_{q \backsim_f p} L(q).$$
- *$F' = F/\backsim_R$*

Informally, Def. V.8 just 'merge' the states among which reductant transitions exists to make a TA more simpler.

VI. An Example of ITS—Railroad Crossing System

In this section, we gives an ITS example showing how to use our proposed verification framework introduced in Section III. As Fig. 19 shows, a 'Railroad Crossing System' is an autonomous ITS system where a behaviour is required to be fired at right location and time. In the next few subsections, we will show how to use STeC to model this system and CCSL to specify some safety properties. And we gives a result of encoded TAs of STeC-model of this system by applying Def. V.5 and a result of corresponding UPPAAL TAs generated by our implemented automatic encoding tool.

In 'Railroad Crossing System' shown in Fig. 19, there are two agents: a train and a gate. The railroad lies in the east-west direction and the road lies in the south-north direction. The scenario of the system is stated as follows:

- When there is no train approaching the gate, gate should be kept open so vehicles can pass the crossing.
- When a train approaches, it sends message 'Approach' to the gate informing that it is approaching. The gate receives the message and the program of gate tells whether to close the gate or keep it open according to the traffic condition of the crossing:
  1. if the gate cannot be closed, the gate sends message 'NonClose' back to the train informing that the train should be stopped. And wait until it can.
  2. if the gate can be closed, sends message 'Close' back to the train telling the train can pass safely.
- The train receives message from gate:
  1. if getting message 'NonClose', it should be stopped and wait message 'Close' for a safe passing.
  2. if getting message 'Close', it can safely pass the crossing.
- After the train safely pass the crossing, it sends message 'Pass' to the gate, the gate should be opened.

We use 'Lapp','Lpass','Lstop' and 'Lleav' to notate the locations at which the train arrive at different time, and 'Open', 'Closed' to notate the locations(states) at which the gate be at different time. The next we use STeC to model the scenario described above (as a comparison, in [21] a similar 'train' system is described using CSP).

A. Modeling ITS Using STeC

We firstly describe train agent and gate agent agent using STeC, and then compose them using parallel operator '||' shown before.

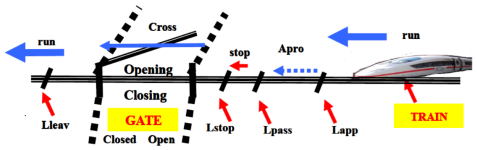

Fig. 19. Railroad Crossing System

1) Train Agent: Let P_{Train} be the name of train agent process and P_{Gate} be the name of gate agent. Let A_{Train}, A_{Gate} be the corresponding encoded TA. Then we have:

$$P_{Train} = Run(\infty); (Send_{(Lapp,t,G)}(Appr) \parallel Approach_{(Lapp,t)}(\Gamma));$$

$$\left\{ \begin{array}{l} Get_{(Lpass,t+\Gamma,G)}(Cross) \rightarrow (Pass_{(Lpass,t+\Gamma)}(\triangle); \\ \quad (Send_{(Lleav,t+\Gamma+\triangle,G)}(Lleav); P_{Train})) \\ [] \\ Get_{(Lpass,t+\Gamma,G)}(Noncross) \rightarrow (Stop_{(Lpass,t+\Gamma)}(\Upsilon); \\ Wait_{(Lstop,t+\Gamma+\Upsilon,G)}(Cross); Pass_{(Lstop,t\prime+\Omega,G)}(\Omega); \\ \quad (Send_{(Lleav,t\prime+\Omega,G)}(Lleav); P_{Train})) \end{array} \right\}$$

The train approaches the crossing at 'Lappr, it sends message 'Appr' to the gate, informing that there is a train coming. After Γ time the train tries to get a message from the gate. If it gets message 'Cross', the gate has been closed and it can pass safely. If it gets the "NonCross" message, then it stops and waits for the gate to be closed. After the train has passed, it sends message 'Leave' to inform the gate.

Applying Def.V.5, we can use the function $\mathcal{F}(P_{Train})$ to transform the model P_{Train} into a TA inductively:

$$\mathcal{F}(P_{Train}) = \mathcal{F}(Run(\infty); P'_{Train})$$

$$= \mathcal{F}(Run(\infty)) \diamond_{STeC} \mathcal{F}(P'_{Train})$$

$$= A_{Run(\infty)} \diamond_{STeC} \mathcal{F}((Send_{(Lapp,t,G)}(Appr) \parallel Approach_{(Laap,t)}(\Gamma));$$

$$\left\{ \begin{array}{l} Get_{(Lpass,t+\Gamma,G)}(Cross) \rightarrow (Pass_{(Lpass,t+\Gamma)}(\triangle); \\ \quad (Send_{(Lleav,t+\Gamma+\triangle,G)}(Lleav); P_{Train})) \\ [] \\ Get_{(Lpass,t+\Gamma,G)}(Noncross) \rightarrow (Stop_{(Lpass,t+\Gamma)}(\Upsilon); \\ Wait_{(Lstop,t+\Gamma+\Upsilon,G)}(Cross); Pass_{(Lstop,t\prime+\Omega,G)}(\Omega); \\ \quad (Send_{(Lleav,t\prime+\Omega,G)}(Lleav); P_{Train})) \end{array} \right\}$$

$$)$$

$$= A_{Run(\infty)} \diamond_{STeC} \mathcal{F}(P_1; P_2) = A_{Run(\infty)} \diamond_{STeC} \mathcal{F}(P_1; P_2)$$

$$= A_{Run(\infty)} \diamond_{STeC} \mathcal{F}(P_1) \diamond_{STeC} \mathcal{F}(P_2) = ... = A_{Train}.$$

where $P_1 = Send_{(Lapp,t,G)}(Appr) \parallel Approach_{(Laap,t)}(\Gamma)$ and

$$P_2 = \left\{ \begin{array}{l} Get_{(Lpass,t+\Gamma,G)}(Cross) \rightarrow (Pass_{(Lpass,t+\Gamma)}(\triangle); \\ \quad (Send_{(Lleav,t+\Gamma+\triangle,G)}(Leave); P_{Train})) \\ [] \\ Get_{(Lpass,t+\Gamma,G)}(Noncross) \rightarrow (Stop_{(Lpass,t+\Gamma)}(\Upsilon); \\ Wait_{(Lstop,t+\Gamma+\Upsilon,G)}(Cross); Pass_{(Lstop,t\prime+\Omega,G)}(\Omega); \\ \quad (Send_{(Leave,t\prime+\Omega,G)}(Lleav); P_{Train})) \end{array} \right.$$

Finally we simplify A_{Train} by applying Def. V.8, and we get $\mathbf{Sim}(A_{Train})$ as the TA of the process P_{Train}. The result of $\mathbf{Sim}(A_{Train})$ is shown in the following figure(Fig. 20).

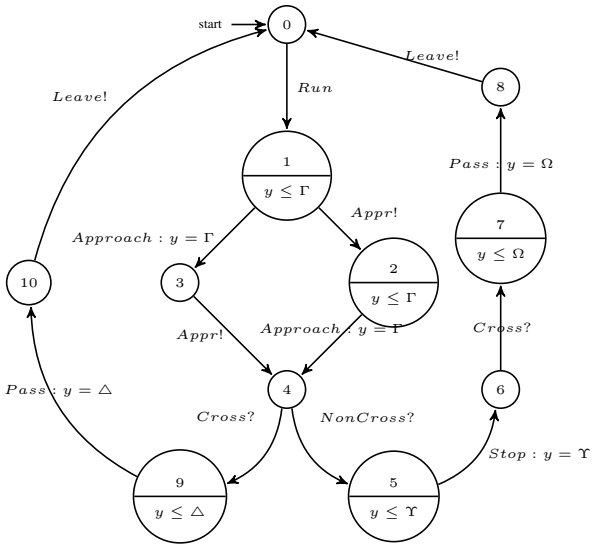

Fig. 20. The Encoded TA (simplified) From Train Agent Process in STeC

Assume that $A_{Train} = (Q, \Sigma, \mathcal{C}, q_0, E, I, AP, L, F)$ is the corresponding TA translated from STeC, the following gives some explanations of A_{Train}:

- $Q = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ is the set of locations of TA.
- $\Sigma = \{Run, Appr!, Cross?, Approach, Pass, Noncross?, Stop, Cross?, Leave!\}$ corresponds to the actions of P_{Train}.
- $\mathcal{C} = \{x, y\}$. Two clocks, one for recording total time from the beginning and one for recording the delay of actions in STeC.
- $E \subseteq Q \times 2^{\Sigma} \times G(\mathcal{C}) \times 2^{\mathcal{C}} \times Q$ is the set of transition relation of A_{Train}. For example, the transition from state 1 to state 3 is a tuple: $\langle 1, \{Approach\}, \{y = \Gamma\}, \{y\}, 3\rangle$, where '$Approach$' is the action name, '$y = \Gamma$' is the guard condition, 'y' is the clock set to be zero after transition is fired.
- $q_0 = 0$ is the initial state.
- $I : Q \rightarrow G(\mathcal{C})$ maps each state to a guard expression. For example, $I(1) = y \leq \Gamma$, so Γ time after being in state 1, the transition is forced to be fired.
- $L : Q \rightarrow 2^{AP}$ is the labelling function. The location in STeC is the atomic proposition in TA. For example, we have $L(10) = \{Lleav\}$, $L(1) = \{Lapp\}$,$L(4) = \{Lpass\}$, $L(0) = \emptyset$, etc.

2) Gate Agent: Following the same procedure, we can get the encoded A_{Gate} from P_{Gate}.

$$P_{Gate} = (Get_{(Open,t,G)}(Appr) \rightarrow Closing_{(Open,t+\theta_0)}(\Pi));$$

$$\left\{ \begin{array}{l} (Closed_{(Closed,t+\theta_0+\Pi)}(1); \\ Send_{(Closed,t+\theta_1,G)}(Cross); (Get_{(Closed,t+\theta_2)}(Leav) \\ \rightarrow Opening_{(Closed,t+\theta_2)}(\zeta))); P_{Gate} \\ [] \\ (Unclosed_{(Unclosed,t+\theta_0+\Pi)}(0); \\ (Send_{(Unclosed,t+\theta_0+\Pi,G)}(NonCross) \parallel \\ Clossing_{(Unclosed,t+\theta_0+\Pi)}(\pi)) \\ ; (Closed_{(Closed,t+\theta 3)}(0) \parallel (Send_{(Closed,t+\theta_3,G)}(\\ Cross)); (Get_{(Closed,t'+\Omega,G)}(Leav) \\ \rightarrow Opening_{(Closed,t'+\theta_4)}(\zeta)); P_{Gate} \end{array} \right\}$$

For the gate agent, it starts with waiting for message 'Appr'. Then the program of gate decides whether close the gate or not. If it is successfully closed, it sends message 'Cross', then waits for message 'Leave' from train to be safe to be opened again. If it is not closed, sends message 'NonClose'.

We can get the simplified version of A_{Gate} as $\mathbf{Sim}(A_{Gate})$, which is shown in Fig. 21 :

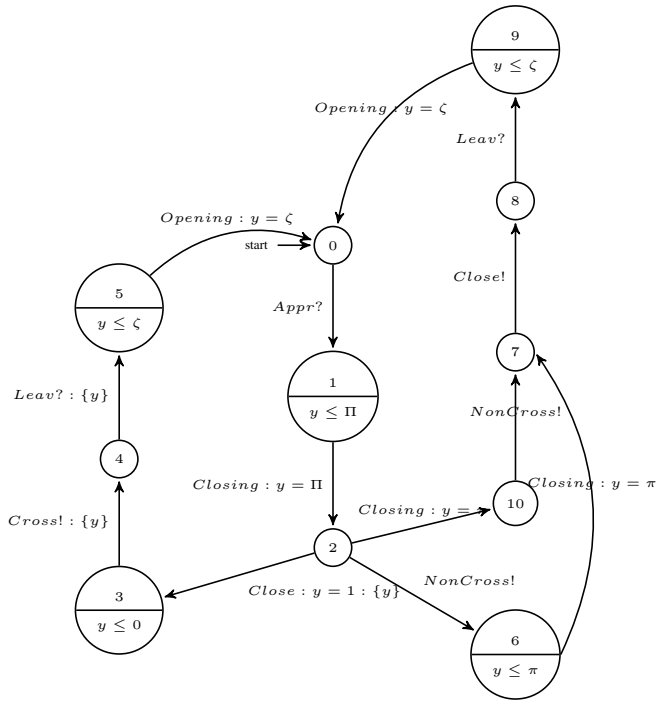

Fig. 21. The Encoded TA (simplified) From Gate Agent Process in STeC

3) Railroad Crossing System: The railroad crossing system as a whole is defined as

$$P_{Sys} = P_{Train} \parallel P_{Gate}$$

The corresponding TA is as follows:

$$A_{Sys} = \mathcal{F}(P_{Sys}) = \mathcal{F}(P_{Train} \parallel P_{Gate}) = A_{Train} \otimes_{STeC} A_{Gate}$$

B. Modeling System in Practice Using STeC

We implemented a tool that takes STeC text as input and output Uppaal Timed Automatas (visit www.uppaal.org for more information about UPPAAL). The next we give the results of encoded TAs of this system by applying our implemented tool. We omit the details of architecture of this tool which is beyond the theme of this paper. Since the implemented encoding algorithm is slightly different from Def.V.5. So the result of encoded TAs(shown in Fig. 35 and Fig. 36) are subtle different from the encoded TAs in Fig. 20 and Fig. 21 which are the direct results from Def.V.5.

In the following we show the STeC code for the 'Railroad Crossing System'. Here is the description of the train agent.

```
//Declarations
TIME t=0.       //time for starting
TIME t_appr=5.      //time spent on approach
TIME t_pass=10.     //time spent on pass
TIME t_stop=8.      //time spent on stop

LOCATION Lpass, Lleav, Lstop, Lapp //location \\
                            of train agent

MESSAGE Cross, Leave, NonCross //messages of \\
                                train agent

CHANNEL Send, Get //communication actions of \\
                    Train agent

ALPHA Pass, Stop, Approach //alpha actions

BETA Run //beta actions

//subprocesses P1 and P2
#define P1 \
  (Get(Lpass)(Cross)->Pass(Lpass)(Lleav,$t_pass)-> \
  Send(Lleav)(Leave))

#define P2 \
  (Get(Lpass)(NonCross)->Stop(Lpass)(Lstop,$t_stop)\
  -> Get(Lstop)(Cross)->Pass(Lstop)(Lleav, $t_pass)\
  -> Send(Lleav)(Leave))

//Train Process
Train=@T(
  Run()();Send(Lapp,$t)(Appr)-> \
  Approach(Lapp)(Lpass,$t_appr)-> \
  (P1 [] P2) -> #T
).
```

In the block of STeC code shown above, we firstly give declarations of different components of STeC language like time, location, message or channel. '**TIME**', '**LOCATION**' and '**MESSAGE**' are the key words of declaration of time, location and message respectively. Two channels '**Send**' and '**Get**' are synchronization actions of type A in STeC declared by key word '**CHANNEL**'. '**ALPHA**' and '**BETA**' represent the 'α' and 'β' type of actions that is defined in Section II.

In the above example, we define 4 time variables: 't', 't_appr', 't_pass' and 't_stop'. 4 locations 'Lpass', 'Lleav', 'Lstop' and 'Lapp'. There are three messages for train agent, that is 'Cross', 'Leave' and 'NonCross'. 'Pass', 'Stop' and

'Approach' are the names of 'α' type of actions. 'Run' is a 'β' action since it does not involve any change of locations. Its 'location' and 'duration' factors being omitted shows that the location and duration of this action are not required.

Two macro 'P1' and 'P2' are subprocesses. The key word '#define' has similar meaning as in C++/C. The macro is substituted for its body during the pre-compilation process.

'Train' is the formula of train process. '@T' is a recursion symbol. It defines a sub formula of 'Train' for which '#T' is substituted at each of its ocurrences. For example, after the subprocess '(P1 [] P2)' is fired, then we replace the symbol '#T' with the sub formula notated by '@T', that is:

```
Run()();Send(Lapp,$t)(Appr)->
Approach(Lapp)($t_appr)-> (P1 [] P2) -> #T
```

So the action 'Run()()' should be fired next.

As the result of STeC tool here we just give the corresponding transformed TA for Train Agent drawn with 'graphviz' tool[1] (Fig. 22), where the red node 'Get_S106' is the initial state and the green node 'F_211' is the final state. The encoded TA of UPPAAL format is shown in Fig. 35 and Fig. 36 in Appendix A.

C. Specifying Properties Using CCSL

As introduced before, we defined CCSL as the specification language in the domain of STeC (see Def. IV.10 and Der. IV.11). Next we specify the properties of 'Railroad Crossing System' using CCSL.

In the 'Railroad Crossing System' introduced above, let us consider two safety properties:

Property 1: When the train passes, the gate must be closed.

We use CCSL to describe the property as follows:

$$c_{close} \prec c_{mayPass} \prec c_{open};$$
$$c_{close} \textbf{ Alternate } c_{open};$$
$$c_{pass} \subseteq c_{mayPass};$$

'$\prec$' means 'precedence' in CCSL, the first formula says the clock 'close' always ticks before the clock 'mayPass' and the clock 'mayPass' always ticks before the clock 'open'. The second formula says that the clock 'close' and the clock 'open' tick alternately. The third formula says that clock 'pass' is a subclock of the clock 'mayPass', which means that whenever the clock 'pass' ticks, the clock 'mayPass' ticks. Consulting the definition of these operators in CCSL(see the previous section of introduction of CCSL or refer to [10] for more details), this CCSL constraint means exactly the **Property 1** mentioned above.

Using Timesquare we have a possible trace satisfying the CCSL constraint shown in Fig. 23.

Considering another property:

[1]'graphviz' is a famous open source visualization tool, visit http://www.graphviz.org/ for more information.

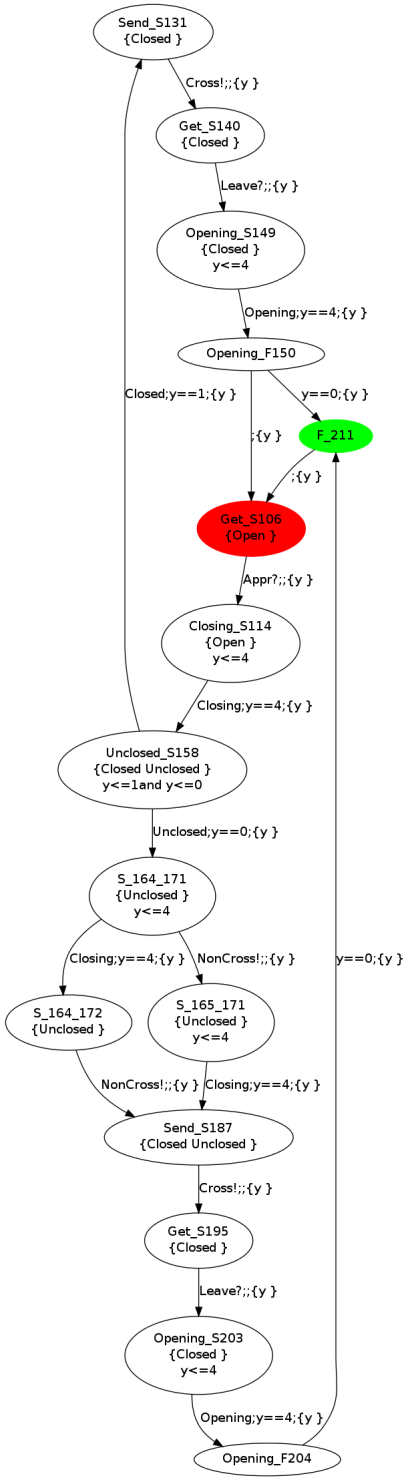

Fig. 22. The TA of Gate Agent Drawn Using graphviz tool

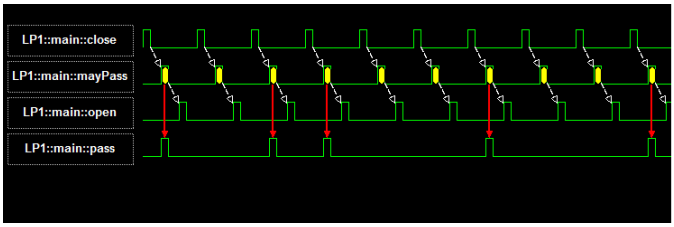

Fig. 23. Simulate result of Property 1

Property 2: Once the train approaches,
the gate shall close in at most 20 seconds.

It contains a time expression '20 seconds'. We use the operator '**DiscretizeBy**' in CCSL to generate a new discrete-time clock c_{time} that ticks every 10s based on an '**IdealClock**'. The expression of **Property 2** is given as follows:

$$c_{appr} \prec c_{close} \prec c_{appr\_delay};$$
$$c_{appr\_delay} = c_{appr} \textbf{ delay } 20 \textbf{ on } c_{time};$$

c_{time} is a 'discrete-time' clock, it is defined as follows:

$$c_{time} = \textbf{IdealClock DiscretizeBy } 1\text{s}$$

The formula above indicates that clock c_{time} 'ticks' every 1s (**IdealClock** can be understood as 'physical clock'), and clock c_{appr_delay} is defined as a clock being fired after 20 'ticks' of c_{time} when c_{appr} 'ticks'. Fig. 39 shows the timed automata of c_{appr_delay}. One defect of it is that it contains too many states, thinking that if we change '20 seconds' into '400 seconds', the resulted TA would contain about 400 states, that is quite large. One alternative way of enhancing it is to let c_{time} 'discretized' by a larger number, say 10s for example, so the delayed counts for c_{appr_delay} becomes smaller. We have the following expressions:

$$c_{appr} \prec c_{close} \prec c_{appr\_delay};$$
$$c_{appr\_delay} = c_{appr} \textbf{ delay } 2 \textbf{ on } c_{time};$$
$$c_{time} = \textbf{IdealClock DiscretizeBy } 10\text{s}$$

Fig. 33 shows its corresponding TA. It contains much less states but it is less 'accurate' than the former one. This is because if c_{appr} 'ticks' between two ticks of c_{time}, the duration until the next tick of c_{appr_delay} is actually less than duration between 2 ticks of c_{time}. Fig. 24 shows one possibility, the duration of first tick of c_{appr} and c_{appr_delay} is less than 20s, while the second duration is equal to 20s.

Now let us give a definition of **IdealClock** by defining the predicate P (Section II gives a definition of **IdealClock**, see [7] for more details).

Let **IdealClock** $= \langle \mathcal{I}_A, \prec, \lambda_A, \mathbb{R}^+, \mu_A \rangle$, $c_{time} = \langle \mathcal{I}_B, \prec, \lambda_B, \mathbb{R}^+, \mu_B \rangle$. Since $\mathcal{I}_B$ is discrete, so we set i^{++} be the successor of i for any $i \in \mathcal{I}_B$. $P \subseteq \mathcal{I}_A \times \mathcal{I}_B$ is defines as:

$$P(j,i) = \begin{cases} true & \text{if } h(i) \prec j \prec h(i^{++}) \\ false & \text{otherwise} \end{cases}$$

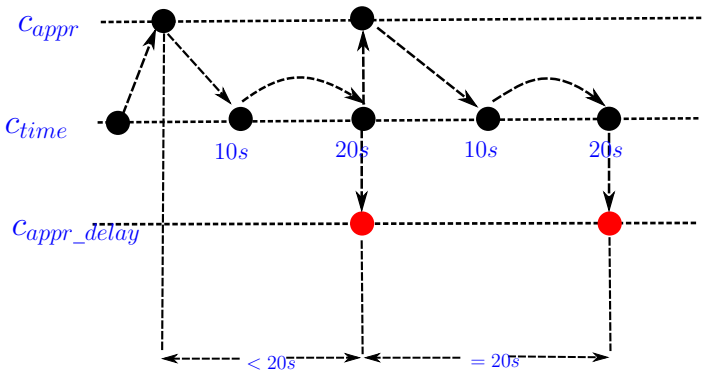

Fig. 24. One condition that the duration between c_{appr} and c_{appr_delay} is less than 20s

where $h : \mathcal{I}_B \to \mathcal{I}_A$ is defined as follows:
$(\forall i \in \mathcal{I}_B)(\exists j \in \mathcal{I}_A)(j = h(i) \wedge \lambda_A(j) = \lambda_B(i))$.
Obviously function $h : \mathcal{I}_B \to \mathcal{I}_A$ is well defined.

D. Verifying Properties on UPPAAL

In this section, we show how to verify the CCSL-descripted properties over STeC model using UPPAAL.

With both STeC system and CCSL properties in TA form that are shown in previous sections, the verification strategy is to compose the system and property by Decare product, and then we proof the satisfaction of system for the property by checking the liveness of CCSL clock.

In our example, we compose the system and property by Decare product in the following way:

$$A_{Sys} = \mathcal{F}(P_{Train} \parallel P_{Gate}) \times A_{CCSL}$$
$$= (A_{Train} \otimes_{STeC} A_{Gate}) \times A_{CCSL}$$

we reach our verification propose by checking if each clock in CCSL satisfies liveness property. Equivalently we check whether each state of A_{CCSL} to satisfy the following CTL property:

$$\mathbf{AG}(\mathbf{AF} \text{ 'clock ticks' }),$$

where 'clock ticks' (in UPPAAL) could be any boolean values indicating that if any CCSL clocks tick. **AG**' and '**AF**' are the operators in CTL. A_{CCSL} is the automata of CCSL property. '$\times$' is the Decare product over TAs. This CTL formula means that 'clock ticks' happens infinitely often, in other words, there is no such a condition under which 'clock ticks' never happens since some point of time.

The encoding from CCSL to TA is introduced in detail in [10], here we only give the results for our example. **Property 1** consists of 4 atomic CCSL expressions, that is:

1. $c_{close} \prec c_{mayPass}$
2. $c_{mayPass} \prec c_{open}$

3. c_{close} **Alternate** c_{open}
4. $c_{pass} \subset c_{mayPass}$

We get the resulted automata of **Property 1** by composing each automata of atomic expression. Fig. 25, Fig. 26, Fig. 27 and Fig. 28 show the automata of each expression. Fig. 29 shows the automata of **Property 1**. Note that we add ϵ-transitions on each state to allow CCSL clock to tick on no condition. For more details about the encoding from CCSL to TA, refer to [10].

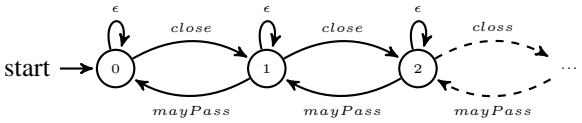

Fig. 25. The automata of $c_{close} \prec c_{mayPass}$

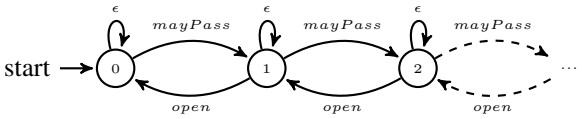

Fig. 26. The automata of $c_{mayPass} \prec c_{open}$

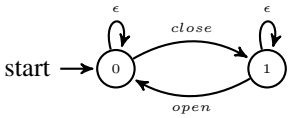

Fig. 27. The automata of c_{close} **Alternate** c_{open}

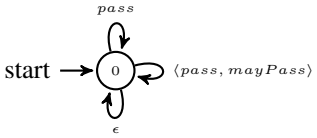

Fig. 28. The automata of c_{close} **Alternate** c_{open}

In practice, we convert **Property 1** into UPPAAL TA, which is shown in Fig. 30. In the UPPAAL TA of **Property 1**, we define each event as boolean variable. In order to synchronize with the Train and Gate agent, we also need to add a function for each event to trigger the boolean variable while the agent triggers the event. Note that in Fig. 30, there is a loop on each state since that we allow each CCSL clock to tick with none boolean variables mimicking the ϵ-transitions in TA shown in Fig. 29. 'x<=ub' is to make sure that each transition will eventually be triggered.

Fig. 37 in Appendix A shows a possible running state diagram of system where three agents run concurrently.

To verify the property we combine the TA of Train-Gate system and the TA of property in UPPAAL and check the liveness of each CCSL clock using CTL formula '**AG AF**

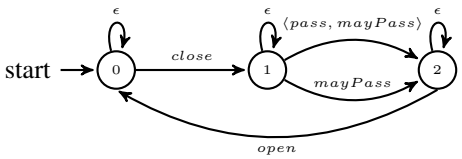

Fig. 29. The automata of **Property 1**

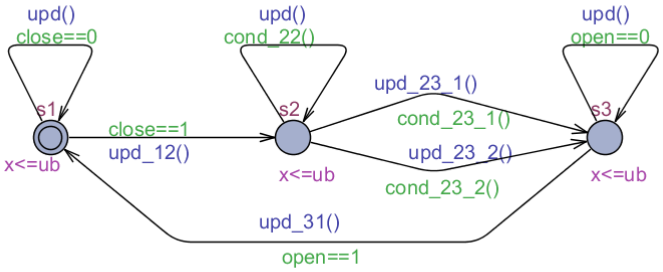

Fig. 30. The UPPAAL TA of **Property 1**

"clock ticks" '. To check **Property 1** we need to check the following CTLs equivalently:

1. **AG AF** ('c_{close} ticks')
2. **AG AF** ('c_{open} ticks')
3. **AG AF** ('$c_{mayPass}$ ticks')
4. **AG** ('train pass' $\rightarrow$ **AF** 'c_{pass} ticks')

In the last statement, we need to consider the pass of the train agent since c_{pass} never ticks if there is no train passing the crossing. Expression 4 means that whenever if the train passes, the clock 'c_{pass}' will eventually tick. Fig. 38 in Appendix A shows a result of the verification of **Property 1**.

The TA of **Property 2** is more complex since it contains time issues. It consists of 4 atomic CCSL expressions:

1. $c_{appr} \prec c_{close}$
2. $c_{close} \prec c_{appr\_delay}$
3. $c_{appr\_delay} = c_{appr}$ **delay** 2 **on** c_{time}
4. $c_{time} =$ **IdealClock DiscretizeBy** 10 s

In [10], the encoding from the expression '**delay on**', '**DiscretizeBy**' and '**IdealClock**' into TA is illustrated in detail. Here we only give the transformed TAs. Fig. 31 and Fig. 32 show the corresponding TA of expression 3 and 4 given above. Fig. 33 gives the TA of **Property 2**. The TAs of expression 1 and 2 are similar to Fig. 25 and Fig. 26, which are omitted here.

In practice, we encoded **Property 2** into UPPAAL TA. Fig. 34 shows its corresponding UPPAAL TA. We check the CTL expressions as follows for the liveness of CCSL clocks:

1. **AG AF** ('c_{appr} ticks')
2. **AG AF** ('c_{close} ticks')
3. **AG AF** ('c_{appr_delay} ticks')

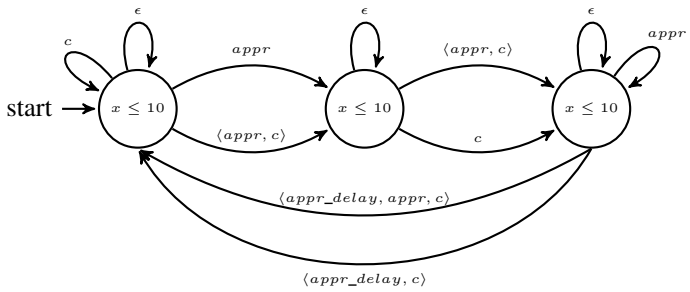

Fig. 31. The TA of $c_{appr\_delay} = c_{appr}$ **delay** 2 **on** c_{time}

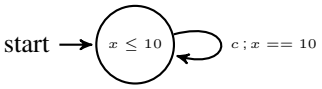

Fig. 32. The TA of c_{time} = **IdealClock DiscretizeBy** 10 s

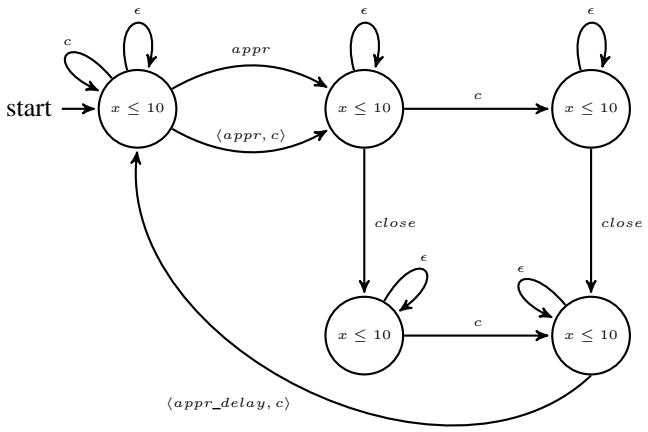

Fig. 33. The TA of **Property 2**

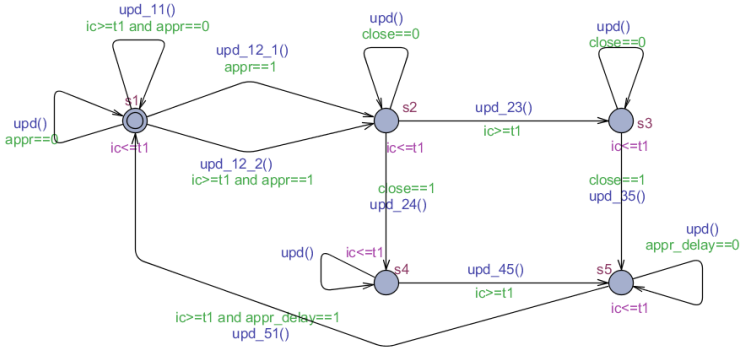

Fig. 34. The UPPAAL TA of **Property 2**

We omit the result of verification of **Property 2**, it is similar to the result of **Property 1**.

VII. Conclusion

In this paper, we proposed a verification framework where we use STeC as a formal modeling language and CCSL as a specification language based on Timed Automatas. We mainly focus on the definition of CCSL in the domain of STeC and the encoding from STeC into TA. We also implement an automatic tool for this transformation and apply UPPAAL as a verification tool to our STeC-models in practice. At last we give a simple example of ITS system to illustrate how to put our proposed verification framework into work both in theory and practice.

As for the future work, in this paper, we give the encoding from STeC to TA, but the equivalence between STeC formula and translated TA has not yet been proofed. Maybe the future work will focus on that.

Acknowledgment

This work is supported by the INRIA Associated Team DAESD between INRIA and ECNU, NSFC (No. 61021004 and No. 61370100) and Shanghai Knowledge Service Platform Project (No. ZF1213).

References

[1] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, pp. 666–677, Aug. 1978.
[2] J. Ouaknine and S. Schneider, "Timed csp: A retrospective," 2006.
[3] R. Milner, *A Calculus of Communicating Systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1982.
[4] S. Schneider, "An operational semantics for timed csp," *Inf. Comput.*, vol. 116, pp. 193–213, Feb. 1995.
[5] R. Alur and D. L. Dill, "A theory of timed automata," *Theor. Comput. Sci.*, vol. 126, pp. 183–235, Apr. 1994.
[6] Y. Chen, "Stec: A location-triggered specification language for real-time systems," in *ISORC Workshops*, pp. 1–6, IEEE, 2012.
[7] C. André and F. Mallet, "Clock constraints in uml/marte ccsl," Research Report RR-6540, 2008.
[8] R. Gascon, F. Mallet, and J. DeAntoni, "Logical time and temporal logics: Comparing uml marte/ccsl and psl," in *TIME* (C. Combi, M. Leucker, and F. Wolter, eds.), pp. 141–148, IEEE, 2011.
[9] F. Mallet, J.-V. Millo, and Y. Romenska, "State-based representation of ccsl operators," Research Report RR-8334, July 2013.
[10] J. Suryadevara, C. Seceleanu, F. Mallet, and P. Pettersson, "Verifying marte/ccsl mode behaviors using uppaal," in *11th International Conference on Software Engineering and Formal Methods*, September 2013.
[11] C. André, "Syntax and semantics of the clock constraint specification language (ccsl)," Research Report RR-6925, 2009.
[12] S. Cattani and M. Kwiatkowska, "A refinement-based process algebra for timed automata," *Formal Aspects of Computing*, vol. 17, no. 2, pp. 138–159, 2005.
[13] P. A. Abdulla, P. Krcál, and W. Yi, "Sampled semantics of timed automata," *Logical Methods in Computer Science*, vol. 6, no. 3, 2010.
[14] H. Wu, Y. Chen, and M. Zhang, "On denotational semantics of spatial-temporal consistency language - stec," in *TASE*, pp. 113–120, IEEE, 2013.
[15] F. Mallet, *Logical Time @ Work for the Modeling and Analysis of Embedded Systems*. Lambert Academic Publisher LAP, 2011. ISBN: 978-3-8433-9388-1.
[16] J. Deantoni and F. Mallet, "Timesquare: Treat your models with logical time," in *TOOLS (50)* (C. A. Furia and S. Nanz, eds.), vol. 7304 of *Lecture Notes in Computer Science*, pp. 34–41, Springer, 2012.
[17] I. Zaretska, G. Zholtkevych, G. Zholtkevych, F. Mallet, and V. Mathematicalmodeling, "Clocks model for specification and analysis of timing in real-time embedded systems."
[18] O. M. Group, *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems Version 1.0*. Object Management Group, 2009.
[19] V. Diekert and P. Gastin, "First-order definable languages," in *Logic*

and Automata: History and Perspectives, Texts in Logic and Games, pp. 261–306, Amsterdam University Press, 2008.

[20] R. Alur and D. Dill, "Automata-theoretic verification of real time systems," 1995.

[21] A. W. Roscoe, "A csp solution to the 'trains' problem," in *The Analysis of Concurrent Systems* (B. T. Denvir, W. T. Harwood, M. I. Jackson, and M. J. Wray, eds.), vol. 207 of *Lecture Notes in Computer Science*, pp. 384–388, Springer, 1983.

APPENDIX A
ADDITIONAL FIGURES PERTAINING TO UPPAAL VERIFICATION

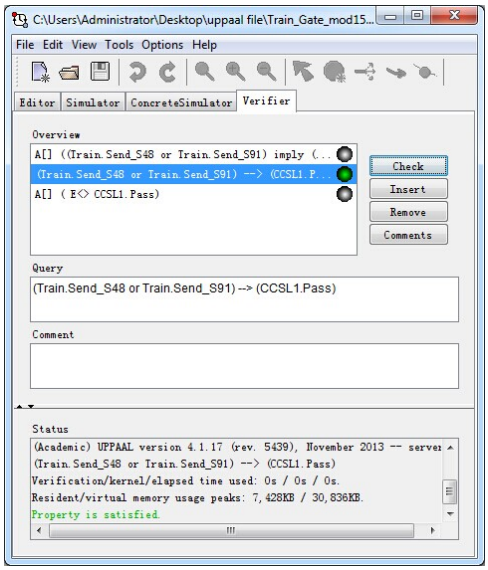

Fig. 38. The result of verification of **Property 1**

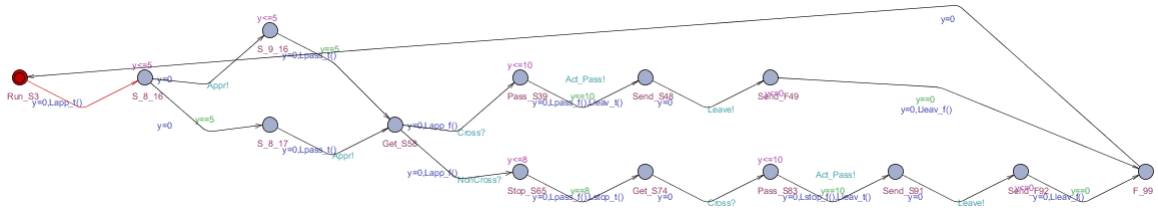

Fig. 35. The Uppaal TA of Train Agent From 'STeC Code'

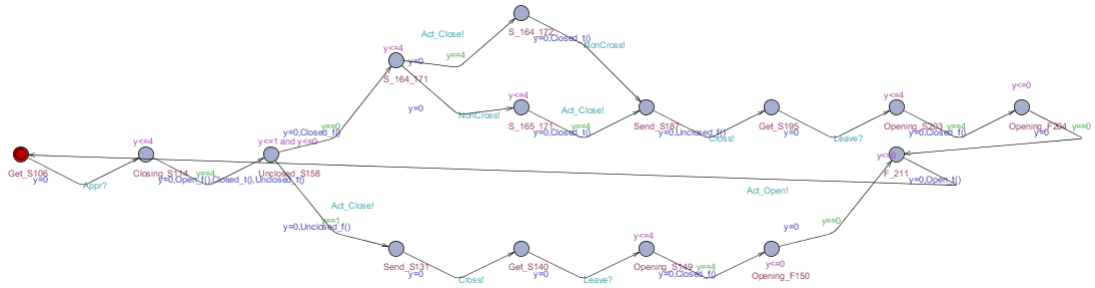

Fig. 36. The Uppaal TA of Gate Agent From 'STeC Code'

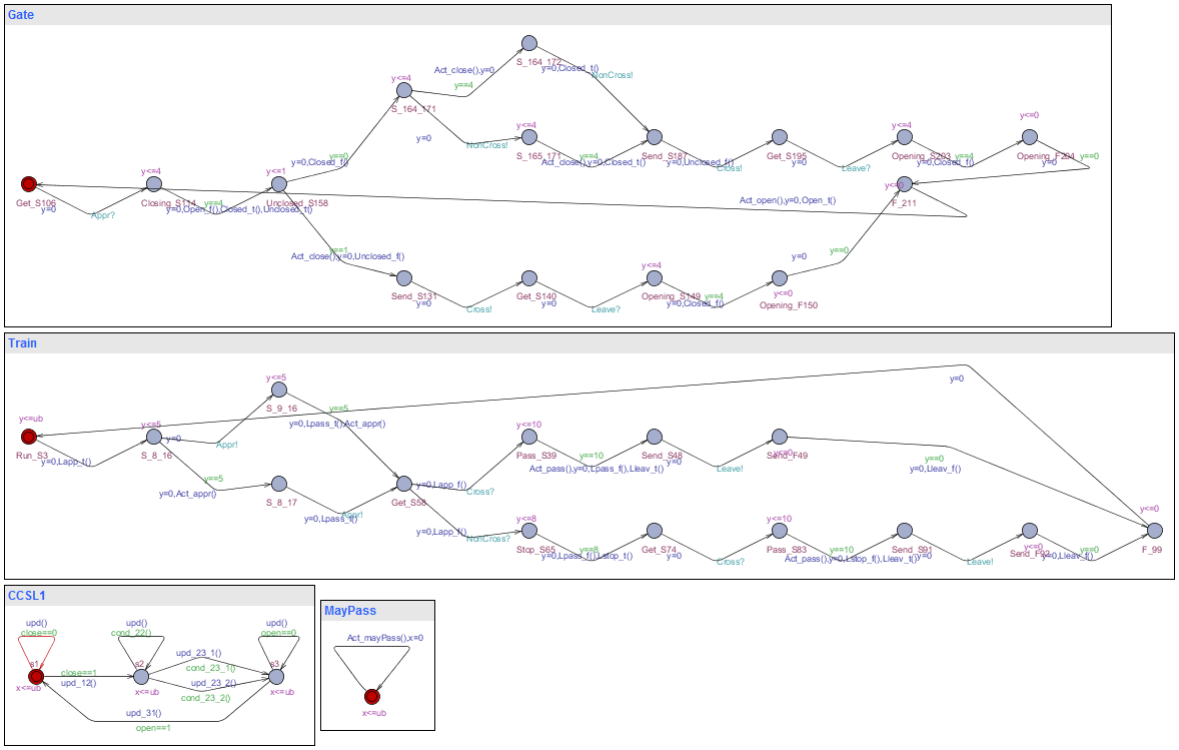

Fig. 37. The running state diagram of A_{Sys} for **Property 1**

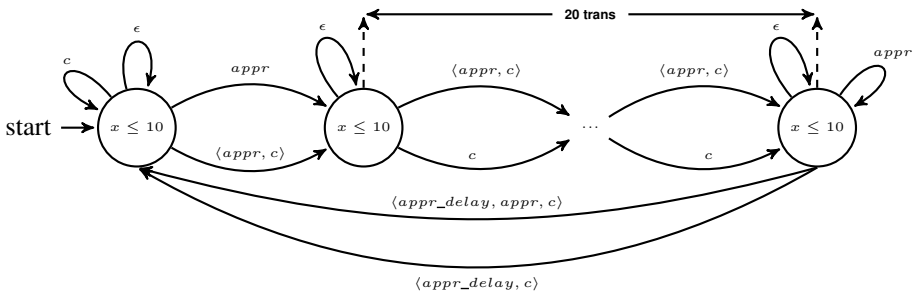

Fig. 39. The TA of $c_{appr\_delay} = c_{appr}$ **delay** 20 **on** c_{time}