Verifying Chinese Train Control System Under a Combined Scenario by Theorem Proving

Liang Zou[1], Jidong Lv[2], Shuling Wang[1], Naijun Zhan[1], Tao Tang[2], Lei Yuan[2], and Yu Liu[2]

[1] State Key Lab. of Comput. Sci., Institute of Software, Chinese Academy of Sciences
[2] State Key Lab. of Rail Traffic Control and Safety, Beijing Jiaotong University

Abstract. In this paper, we investigate how to formalize and verify the System Requirements Specification (SRS) of Chinese Train Control System Level 3 (CTCS-3), which includes a set of basic operational scenarios that cooperate with each other to achieve the desired behavior of trains. It is absolutely necessary to prove that the cooperation of basic scenarios indeed completes the required behavior. As a case study, a combined scenario with several basic scenarios integrated is studied in this paper. We model each scenario as a Hybrid CSP (HCSP) process, and specify its properties using Hybrid Hoare Logic (HHL). Given such an annotated HCSP model, the deductive verification of conformance of the model to the properties is then carried out. For the purpose, we implement a theorem prover of HHL in Isabelle/HOL, with which the process including modelling and verification of annotated HCSP models can be mechanized. In particular, we provide a machine-checked proof for the combined scenario, with the result indicating a design error in SRS of CTCS-3.

Keywords: Chinese Train Control System, Hybrid System, Specification and Verification, Theorem Proving

1 Introduction

The System Requirements Specification (SRS) of Chinese Train Control System Level 3 (CTCS-3) [16] is a standard specification for supervising train movements to ensure the high reliability, safety and efficiency of high-speed trains in China. CTCS-3 includes 14 basic operational scenarios, each of which with different system components involved, and the cooperations among these scenarios to achieve the desired behavior of trains. One important problem in this area is to formalise and verify the specifications for the scenarios of CTCS-3, both separately and integrally, to guarantee the correctness.

Due to continuous character of train movement and discrete interactions between different system components, we model each scenario of CTCS-3 as a hybrid system. Hybrid system seamlessly combines the models for discrete controllers and for dynamic systems represented by differential or algebraic equations. In this paper, we adopt Hybrid CSP (HCSP) [2, 20] as the formal modelling language for hybrid systems. As an extension of CSP, HCSP introduces real-time constructs and differential equations for continuous evolution; and being a process algebra, it provides standard means for constructing complex systems out of simpler ones, which facilitates compositionality.

By using HCSP, each basic scenario of CTCS-3 is formalized as an HCSP process, which is usually a parallel composition of sub-processes corresponding to different components involved in this scenario. A combined scenario integrates several basic scenarios that occur in a same situation, and the HCSP process for it can be constructed from the processes corresponding to each separate scenario. However, the combination of scenarios may not preserve correctness because of complex interactions between these scenarios, thus as a remedy, to verify correctness of combined scenarios is very necessary.

In this paper, we consider one combined scenario that integrates several basic scenarios including *movement authority*, *level transition* and *mode transition*. We model the combined scenario using HCSP, and then formulate the property to be proved using Hybrid Hoare logic (HHL) proposed in [6]. HHL is defined especially for reasoning about HCSP processes, including first-order logic to specify pre/post-conditions which describe the properties related to discrete jumps, and duration calculus (DC) [18, 17] to record execution history that specifies continuous properties of systems, and a set of axioms and inference rules to axiomatize each construct of HCSP. Finally, we prove the negation of the property that a train eventually passes through the location at which a level transition and a mode transition take place simultaneously. This result reflects some design error in SRS of CTCS-3.

In order to provide a machine-checked proof for the negation of the property, we implement a theorem prover for HHL in proof assistant Isabelle/HOL. The implementation includes the encodings of HCSP language and HHL proof system, including both syntax and semantics (or inference rules for HHL instead). It is built from scratch, i.e., defining the datatype for expressions from the bottom most, in the style of deep embedding. Therefore, we can make full use of inductive structure of assertions and thus reduce the size of verification conditions generated.

1.1 Related Work

There have been a number of abstract models and specification languages proposed for formalizing and verifying hybrid systems. The most popular is hybrid automata [1, 8, 4], with real-time temporal logics [8, 9] interpreted on their behaviors as specification languages. However, analogous to state machines, hybrid automata provides little support for structured description and composition. The approach most closely related to ours is the work by Platzer [11], where hybrid programs and the related differential dynamic logic for the deductive verification of hybrid systems are proposed. As a case study of the approach, the safety and liveness of movement authority scenario of European Train Control System was proved [12]. However, hybrid programs do not support parallelism and communication.

For mechanization of HCSP verification, the encodings of the assertion languages of HHL especially DC are most essential. The first attempt at encoding DC in a theorem prover was done in PVS [14], where shallow embedding is adopted thus reasoning is done directly in high-order meta-logic of PVS. Later, the work in both [3] and [13] considers the encoding of DC in Isabelle/HOL in deep embedding style, and our encoding combines their approaches.

For formal modelling and verification of scenarios of train control systems, most of existing work only consider single scenario so far, e.g. in [12, 5], some separate scenarios of European Train Control System are considered.

1.2 Structure of the paper

We give a brief introduction of HCSP and HHL in Sec. 2, and then introduce a combined scenario of CTCS-3 and its formal model in HCSP in Sec. 3. We present the mechanization of HCSP specifications in Isabelle/HOL in Sec. 4, based on which verify the combined scenario via interactive theorem proving in Sec.5. Finally, the paper concludes and discusses the future work.

2 Preliminaries

This section introduces briefly the modelling language and specification language for hybrid systems that we adopt in the paper.

2.1 Hybrid CSP Language

HCSP [2, 20] is a formal language for describing hybrid systems, which is an extension of CSP by introducing timing constructs, interrupts, and differential equations for representing continuous evolution. Exchanging data among processes are described solely by communications, and no shared variable is allowed between processes in parallel, so each program variable is local to the respective sequential component. The syntax of HCSP is given as follows:

$$\begin{aligned} P &::= \text{skip} \mid x := e \mid ch?x \mid ch!e \mid P; Q \mid B \rightarrow P \mid P \sqcup Q \mid P^* \\ &\quad\mid \langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle \mid \langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle \trianglerighteq [\!]_{i \in I} (io_i \rightarrow Q_i) \\ S &::= P \mid S \| S \end{aligned}$$

Here P, Q, Q_i, S are HCSP processes, x and s stand for process variables, ch for channel name, io_i for a communication event (either $ch?x$ or $ch!e$), B and e for boolean and arithmetic expressions, and d for a non-negative real constant, respectively.

The intended meaning of the individual constructs is explained as follows:

- skip terminates immediately having no effect on variables; and $x := e$ assigns the value of expression e to x and then terminates.
- $ch?x$ receives a value along channel ch and assigns it to x, and $ch!e$ sends the value of e along ch. A communication takes place as soon as both the sending and the receiving parties are ready, and may cause one side to wait.

- The sequential composition $P; Q$ behaves as P first, and if it terminates, as Q afterwards.
- The conditional $B \to P$ behaves as P if B is true, otherwise it terminates immediately.
- The internal choice $P \sqcup Q$ behaves as either P or Q, and the choice is made randomly by the system.
- The repetition P^* executes P for some finite number of times.
- $\langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle$ is the continuous evolution statement (hereafter shortly *continuous*). It forces the vector s of real variables to evolve continuously according to the differential equations $\mathcal{F}$ as long as the boolean expression B, which defines the *domain of* s, holds, and terminates when B turns false.
- $\langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle \trianglerighteq []_{i \in I}(io_i \to Q_i)$ behaves like the continuous $\langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle$, except that it is preempted as soon as one of the communications io_i takes place. That is followed by the respective Q_i. Notice that, if the continuous terminates before a communication from among $\{io_i\}_{i \in I}$ occurs, then the process terminates immediately without waiting for communication.
- $S_1 \| S_2$ behaves as if S_1 and S_2 run independently except that all communications along the common channels connecting S_1 and S_2 are to be synchronized. S_1 and S_2 in parallel can neither share variables, nor input nor output channels.

The basic constructs of HCSP are expressive enough to define a number of constructs known in process calculi. For instances, the **stop** and *external choice* in timed CSP can be respectively defined as

$$\begin{aligned} \textbf{stop} &\;\widehat{=}\; \langle \dot{t} = 1 \& \text{True} \rangle, \text{ and} \\ []_{i \in I}(io_i \to Q_i) &\;\widehat{=}\; \textbf{stop} \trianglerighteq []_{i \in I}(io_i \to Q_i); \end{aligned}$$

and especially, the timeout $\langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle \trianglerighteq_d Q$ can be defined by

$$t := 0; \langle F(\dot{s}, s) = 0 \wedge \dot{t} = 1 \& t < d \wedge B \rangle; t \geq d \to Q,$$

which behaves like the continuous $\langle \mathcal{F}(\dot{s}, s) = 0 \& B \rangle$, if the continuous terminates before d time units, otherwise, after d time units of evolution according to $\mathcal{F}$, it moves on to execute Q. Based on timeout, the wait statement can be defined as wait $d \;\widehat{=}\; \langle \dot{t} = 1 \rangle \trianglerighteq_d$ skip.

Super-Dense Computation For HCSP, we adopt the notion of super-dense computation [8] to assume that digital control does not consume time compared to continuous evolution of environment. Discrete processes such as skip, assignment, as well as the evaluation of boolean expressions in $B \to P$, take no time to complete. Thus at a time point, multiple discrete processes may occur. Because of synchronization, the input or output process may cause to wait for the compatible party being available, but as soon as both parties become ready, a communication will occur and complete immediately.

2.2 Hybrid Hoare Logic

HHL [6] is an extension of Hoare logic for specifying and reasoning about HCSP processes. In HHL, each specification for a sequential process P takes the form $\{Pre\}P\{Post; HF\}$, where $Pre, Post$ represent pre-/post-conditions, expressed by first-order logic, to specify discrete properties of variables held at starting and termination of the execution of P; and HF history formula, expressed by DC [18, 17], to record the execution history of P, including real-time and continuous properties. The effect of discrete processes will be specified by the pre-/post-conditions, but not be recorded in the history. The specification for a parallel process is then defined by assigning to each sequential component of it the respective pre-/post-conditions and history formula, that is

$$\{Pre_1, Pre_2\}P_1\|P_2\{Post_1, Post_2; HF_1, HF_2\}$$

In HHL, each of HCSP constructs is axiomatized by a set of axioms and inferences rules, which constitute a basis for implementing the verification condition generator for reasoning about HCSP specifications in Sec.4. The full explanation of HHL can be found in [6].

DC is a real extension of Interval Temporal Logic (ITL) [10] for specifying and reasoning about real-time systems. Like ITL, the only modality in DC is the chop $^\frown$ to divide a considered interval into two consecutive sub-intervals such that its first operand is satisfied on the first sub-interval, while the second operand is satisfied on the second sub-interval. Besides, DC extends ITL by introducing durations of state expressions $\int S$, and the temporal variable ℓ to denote the length of the considered interval, i.e. $\int 1$. Here, we will adopt the notion of point formula introduced in [19], denoted by $\lceil S \rceil^0$, to mean that S holds at the considered point interval. Then the formula $\lceil S \rceil$ is defined as $\neg(\ell > 0^\frown\lceil \neg S \rceil^{0\frown}\ell > 0)$, meaning that the state expression S holds at each point of the considered reference interval.

3 A Combined Scenario of CTCS-3 and Its HCSP Model

A train at CTCS-3 applies for movement authorities (MAs) from Radio Block Center via GSM-Railway and is guaranteed to move safely in high speed within its MA. CTCS-2 is a backup system of CTCS-3, under which a train applies for MAs from Train Control Center via train circuit and balise instead. There are 9 main operating modes in CTCS-3, among which the Full Supervision and Calling On modes will be involved in the combined scenario studied in this paper. During Full Supervision mode, a train needs to know the complete information including its MA, line data, train data and so on; while during Calling On mode, the on-board equipment of the train cannot confirm cleared routes, thus a train is required to move under constant speed 40km/h.

The operating behavior of CTCS-3 is specified by 14 basic scenarios, all of which cooperate with each other to constitute normal functionality of train control system. The combined scenario considered here integrates the Movement

Authority and Level Transition scenarios of CTCS-3, plus a special Mode Transition scenario.

For modeling a scenario, we model each component involved in it as an HCSP process and then combine different parts by parallel composition to form the model of the scenario. In particular, the train participates in each scenario, and the HCSP model corresponding to the train under different scenarios has a very unified structure. Let s be trajectory, v velocity, a acceleration, t clock time of a train respectively, then we have the following general model for the train:

$$Train \,\widehat{=}\, \begin{pmatrix} \langle \dot{s} = v, \dot{v} = a, \dot{t} = 1 \& \ B \rangle \trianglerighteq [\!]_{i \in I}(io_i \to P_{comp_i}); \\ Q_{comp} \end{pmatrix}^*$$

where P_{comp_i} and Q_{comp} are discrete computation that takes no time to complete. The train process proceeds as follows: at first the train moves continuously at velocity v and acceleration a, as soon as domain B is violated, or a communication among $\{io_i\}_{i \in I}$ between the train and another component of CTCS-3 takes place, then the train movement is interrupted and shifted to Q_{comp}, or P_{comp_i} respectively; after the discrete computation is done, the train repeats the above process, indicated by * in the model. For each specific scenario, the domain B, communications io_i, and computation P_{comp_i} and Q_{comp} can be instantiated correspondingly. We assume the acceleration a is always in the range $[-b, A]$.

In the rest of this section, we will first model three basic scenarios separately, and then construct a combined scenario from them.

3.1 Movement Authority Scenario

Among all the scenarios, MA is the most basic one and crucial to prohibit trains from colliding with each other. Before moving, the train applies for MA from Radio Block Center (RBC, in CTCS-3) or Train Control Center (in CTCS-2), and if it succeeds, it gets the permission to move but only within the MA it owns. An MA is composed of a sequence of segments. Each segment is represented as a tuple $(v_1, v_2, e, mode)$, where v_1 and v_2 represent the speed limits of emergency brake pattern and normal brake pattern by which the train must implement emergency brake and normal brake (thus v_1 is always greater than v_2), e the end point of the segment, and *mode* the operating mode of the train in the segment. We introduce some operations on MAs and segments. Given a non-empty MA α, we define $hd(\alpha)$ to return the first segment of α, and $tl(\alpha)$ the rest sequence after removing the first segment; and given a segment seg, we define $seg.v_1$ to access the element v_1 of seg, and similarly to other elements.

Given an MA, we can calculate its static speed profile and dynamic speed profile respectively. As an illustration, Fig. 1 presents an MA with three segments, separated by points s_1, s_2, and s_3. In the particular case, we assume s_3 the end of the MA, thus the train is required to fully stop at s_3 if the MA is not extended. The static speed profile corresponds to two step functions formed by the two speed limits (i.e. v_1 and v_2) of each segment; and for any segment seg, the dynamic speed profile is calculated down to the higher speed limit of next

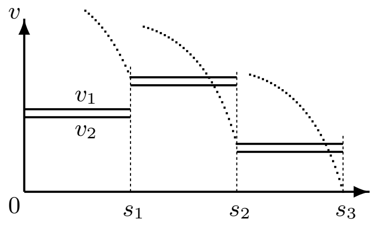

Fig. 1. Static and dynamic speed profiles

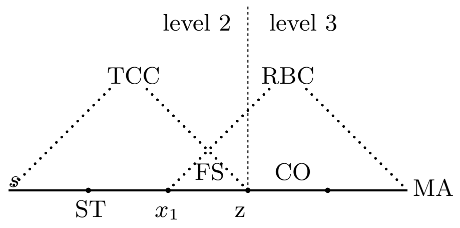

Fig. 2. Level and mode transition

segment taking into account the train's maximum deceleration (i.e. constant b), and corresponds to the curved function $v^2 + 2b\,s < next(seg).v_1^2 + 2b\,seg.e$, where $next(seg)$ represents the next segment following seg in the considered MA. The train will never be allowed to run beyond the static and dynamic speed profiles.

By instantiation to the general model, we get the model for a train under MA scenario. Let B_0 represent the general restriction that the train always moves forward, i.e. $v \geq 0$, or otherwise, the train has already stopped deceleration (denoted by $a \geq 0$). If B_0 fails to hold, the acceleration a needs to be set by a non-negative value in $[0, A]$. Notice that we add T_{delay} to clock t to guarantee that the interrupt B_0 can at most occur once every T_{delay} time units, to avoid Zeno behavior. This is in accordance with the real system to check the condition periodically. We adopt this approach several times.

Let B_1 denote the case when the speed is less than the lower limit v_2, or otherwise the train has already started to decelerate; and B_2 the case when the speed is less than the higher limit v_1 and not exceeding the dynamic speed profile, or otherwise the train has already started an emergency brake, respectively. When B_1 or B_2 is violated, the acceleration a will be assigned to be negative or maximum deceleration b respectively, as shown in $Q1_{comp}$ below. For future use, we denote the formula for specifying dynamic speed profile, i.e. $\forall seg : MA\,.\,v^2 + 2b\,s < next(seg).v_1^2 + 2b\,seg.e$, by DSP_Form.

Let B_7 represent that the train moves within the first segment of current MA. Whenever it is violated, i.e. $s > hd(MA).e$, the train will apply for extension of MA from TCC and RBC respectively. Define $rMA2$ and $rMA3$ to represent the MAs allocated by TCC and RBC respectively, then normally the MA of train will be defined as the minimum of the two. As defined in $Q1_{comp}$, the application procedure behaves as follows: the train first sends the value of $\neg B_7$ to both TCC and RBC; if $\neg B_7$ is true, it sends the end of authorities (defined by $getEoA$) of $rMA2$ and $rMA3$ to TCC and RBC, and then receives the new extended authorities (defined by $setMA2$, $setMA3$) for $rMA2$ and $rMA3$ from TCC and RBC respectively; and finally the MA will be updated correspondingly (defined by $comb$).

$$
\begin{array}{ll}
B_0 & \widehat{=}\ (v \geq 0 \vee a \geq 0 \vee t < Temp + T_{delay}) \\
B_1 & \widehat{=}\ (\forall seg : MA \,.\, v < seg.v_2) \vee a < 0 \vee t < Temp' + T_{delay} \\
B_2 & \widehat{=}\ (\forall seg : MA \,.\, v < seg.v_1 \wedge v^2 + 2b\, s < next(seg).v_1^2 + 2b\, seg.e) \vee a = -b \\
B_7 & \widehat{=}\ (s <= hd(MA).e) \\
Q1_{comp} & \widehat{=}\ \neg B_0 \rightarrow (Temp := t; \sqcup_{\{0<=c<=A\}} a := c); \\
& \quad \neg B_1 \rightarrow (Temp' := t; \sqcup_{\{-b<=c<0\}} a := c); \\
& \quad \neg B_2 \rightarrow a := -b; \\
& \quad CH_{b2}!\neg B_7; CH_{b3}!\neg B_7; \\
& \quad \neg B_7 \rightarrow (CH_{eoa2}!getEoA(rMA2); ch_{ma2}?rMA2; \\
& \qquad\qquad CH_{eoa3}!getEoA(rMA3); ch_{ma3}?rMA3; \\
& \qquad\qquad MA := comb(rMA2, rMA3)) \\
TCC & \widehat{=}\ CH_{b2}?b2; b2 \rightarrow (CH_{eoa2}?eoa2; ch_{ma2}!setMA2(eoa2)) \\
RBC_{ma} & \widehat{=}\ CH_{b3}?b3; b3 \rightarrow (CH_{eoa3}?eoa3; ch_{ma3}!setMA3(eoa3))
\end{array}
$$

3.2 Level Transition

When a train moves under CTCS-2, then whenever passing a balise, which is assumed to be equally distributed every δ meters along the track, the train can apply for upgrade to CTCS-3 when necessary. Let B_3 represent the negative of the case when the train is at level 2 and passing a balise. When B_3 is violated, then as specified in $Q2_{comp}$, the following computation will take place: first, the train sends a level upgrade application signal to RBC; as soon as RBC receives the application, it sends back the package (b, x_1, x_2) to the train, where b represents whether RBC approves the application, x_1 the location for starting level upgrade, and x_2 the location for completing level upgrade; if RBC approves the level upgrade (i.e. b is true), the train enters level 2.5 and meanwhile passes the balise. Notice that level 2.5 does not actually exist, but is used only for modelling the middle stage between level 2 and level 3, during which the train will be supervised by both CTCS-2 and CTCS-3. Finally, as soon as the train at level 2.5 reaches location x_2 (the negative denoted by B_4), the level will be set to 3, specified in $Q3_{comp}$. RBC_{lu} defines the process for RBC under the level transition scenario.

$$
\begin{array}{ll}
B_3 & \widehat{=}\ level \neq 2 \ \vee \ s \neq n * \delta \\
B_4 & \widehat{=}\ level \neq 2.5 \ \vee \ s \leq LU.x_2 \\
Q2_{comp} & \widehat{=}\ \neg B_3 \rightarrow (CH_{LUA}!; CH_{LU}?LU; LU.b \rightarrow level = 2.5; n = n + 1); \\
Q3_{comp} & \widehat{=}\ \neg B_4 \rightarrow level := 3 \\
RBC_{lu} & \widehat{=}\ CH_{LUA}?; \sqcup_{b_{LU} \in \{true, false\}} CH_{LU}!(b, x_1, x_2)
\end{array}
$$

3.3 Mode Transition

When a train moves under CTCS-2, it will always check whether its operating mode is equal to the mode of current segment, i.e. $hd(MA).mode$. We denote this condition by B_5, and as soon as it is violated, the train will update its mode to be consistent with *mode* of the segment, specified in $Q4_{comp}$.

$$\begin{array}{ll} B_5 & \widehat{=}\ mode = hd(MA).mode \\ Q4_{comp} & \widehat{=}\ \neg B_5 \rightarrow mode := hd(MA).mode \end{array}$$

We consider the mode transition from Full Supervision (FS) to Calling On (CO) under CTCS-3, which is a little complicated. In the MA application stage, RBC can only grant the train the MAs before the CO segment. The train needs to ask the permission of the driver before moving into a CO segment at level 3. To reflect this situation in modelling, we initialize both the speed limits for CO segments to be 0, and as a result, if the train fails to get the permission from the driver, it must stop before the CO segment; but if the train gets the driver's permission, the speed limits of the CO segments will be reset to be positive.

Let B_6 denote the negation of the case when the train is at level 3, and it moves to 300 meters far from the end of current segment, and the mode of next segment is CO. As soon as B_6 is violated, then as specified in $Q5_{comp}$, the following computation will take place: first, the train will report the status to the driver and ask for permission to enter next CO segment via communications; if the driver sends true, the speed limits of next CO segment will be reset to be $40km/h$ and $50km/h$ respectively (abstracted away by function $coma(MA)$). As a consequence, the train is able to enter next CO segment at a positive speed successfully. $Driver_{mc}$ defines the process for the driver under the mode transition scenario.

$$\begin{array}{lll} B_6 & \widehat{=} & level \neq 3 \vee CO \neq hd(tl(MA)).mode \vee hd(MA).e - s > 300 \\ & & \vee t < Temp + T_{delay} \\ Q5_{comp} & \widehat{=} & CH_{win}!\neg B_6; \neg B_6 \rightarrow Temp := t; CH_{DC}?b_{rConf}; b_{rConf} \rightarrow coma(MA) \\ Driver_{mc} & \widehat{=} & CH_{win}?b_{win}; b_{win} \rightarrow \sqcup_{b_{sConf} \in \{true, false\}} CH_{DC}!b_{sConf} \end{array}$$

3.4 Combined Scenario and Model

We combine the scenarios introduced above, but with the following assumptions for the occurring context:

- The train moves inside an MA it owns;
- There are two adjacent segments in the MA, divided by point z. The train is supervised by CTCS-2 to the left of z and by CTCS-3 to the right, and meanwhile, it is operated by mode FS to the left of z and by mode CO to the right. Thus the locations for mode transition and for level transition are coincident. As the starting point of a CO segment, both speed limits for location z are initialized to 0 by RBC;
- The train has already got the permission for level transition from RBC which sends $(true, x_1, z)$.

Please see Fig. 2 for an illustration.

The model of the combined scenario can then be constructed from the models of all the basic scenarios contained in it. The construction takes the following steps: firstly, decompose the process for each basic scenario to a set of sub-processes corresponding to different system components that are involved in the scenario (usually by removing parallel composition on top); secondly, as a component may participate in different basic scenarios, re-construct the process for it based on the sub-processes corresponding to it under these scenarios (usually by conjunction of continuous domain constraints and sequential composition of discrete computation actions); lastly, combine the new obtained processes for all the components via parallel composition. According to this construction process, we get the following HCSP model for the combined scenario:

$$\begin{array}{ll} System & \widehat{=}\, Train^* \parallel Driver^*_{mc} \parallel RBC^*_{lu} \parallel RBC^*_{ma} \parallel TCC^* \\ Train & \widehat{=}\, \langle \dot{s} = v, \dot{v} = a, \dot{t} = 1\&\ B_0 \wedge B_1 \wedge B_2 \wedge B_3 \wedge B_4 \wedge B_5 \wedge B_6 \wedge B_7\rangle; P_{train} \\ P_{train} & \widehat{=}\, Q1_{comp}; Q2_{comp}; Q3_{comp}; Q4_{comp}; Q5_{comp} \end{array}$$

According to SRS of CTCS-3, we hope to prove that the combined scenario satisfies a liveness property, i.e., the train can eventually pass through the location for level transition and mode transition. Our work applies deductive verification method for verifying HCSP models. First, the requirements to be proved are specified using HHL assertions as annotations in HCSP model, and then based on the proof system of HHL, the annotated HCSP model is reduced to a set of logical formulas whose validity implies the conformance of the model with respect to the requirements. This process can be mechanized in proof assistant, which will be the main content of the rest.

4 Isabelle Implementation

In this section, we aim to check if an HCSP process is correct with respect to a specification written in HHL, by providing a machine-checkable proof in Isabelle/HOL. For this purpose, we need to encode HCSP including both its syntax and semantics, and moreover, the axioms and inference rules of HHL. We adopt the deep embedding approach [15] here, which represents the abstract syntax for both HCSP and assertions by new datatypes, and then defines the semantic functions that assign meanings to each construct of the datatypes. It allows us to quantify over the syntactic structures of processes and assertions, and furthermore, make full use of deductive systems for reasoning about assertions written in FOL and DC.

The full repository including all the mechanization code related to Sec.4 and Sec.5 can be found at `https://github.com/liangdezou/HHL_prover`, and we present part of them here. We start from encoding the bottom construct, i.e. expressions, that are represented as a datatype `exp`:

datatype exp = RVar *string* | SVar *string* | BVar *string* | Real *real*
| String *string* | Bool *bool* | exp + exp | exp − exp | exp ∗ exp

An expression can be a variable, that can be of three types, `RVar x` for real variable, `SVar x` and `BVar x` for string and boolean variables; a constant, that can be also of the three types, e.g. `Real 1.0`, `String ''CO''` and `Bool True`; an arithmetic expression constructed from operators $+, -, *$. Based on expressions, we can define the assertion languages and the process language HCSP respectively.

4.1 Assertion Language

There are two assertion logics in HHL: FOL and DC, where the former is used for specifying the pre-/post-conditions and the latter for the execution history of a process respectively. The encodings for both logics consist of two parts: syntax and deductive systems. We will encode the deductive systems in Gentzen's sequent calculus style, which applies backward search to conduct proofs and thus is more widely used in interactive and automated reasoning. A sequent is written as $\Gamma \vdash \Delta$, where both Γ and Δ are sequences of logical formulas, meaning that when all the formulas in Γ are true, then at least one formula in Δ will be true. We will implement a sequent as a truth proposition. The sequent calculus deductive system of a logic is composed of a set of sequent rules, each of which is a relation between a (possibly empty) sequence of sequents and a single sequent. In what follows, we consider to encode FOL and DC respectively.

First-Order Logic The FOL formulas are constructed from expressions by using relational operators from the very beginning, and can be represented by the following datatype `fform`:

datatype fform = [True] | [False] | exp [=] exp | exp [<] exp
| [¬] fform | fform [∨] fform | [∀] *string* fform

The other logical connectives including [∧], [→], and [∃] can be derived as normal. For quantified formula [∀]*string* `fform`, the name represented by a string corresponds to a real variable occurring in `fform`. We only consider the quantification over real variables here, but it can be extended to variables of other types (e.g. string and bool) without any essential difficulty. Notice that we add brackets to wrap up the logical constructors in order to avoid the name conflicts between `fform` and the FOL system of Isabelle library. But in sequel, we will remove brackets for readability when there is no confusion in context; and moreover, in order to distinguish between FOL formulas and Isabelle meta-logic formulas, we will use $\Rightarrow$, & and | to represent implication, conjunction and disjunction in Isabelle meta-logic.

Now we need to define the sequent calculus style deductive system for `fform`. The Isabelle library includes an implementation of the sequent calculus of classical FOL with equation, based upon system LK that was originally introduced by Gentzen. Our encoding of the sequent calculus for `fform` is built from it directly, but with an extension for dealing with the atomic arithmetic formulas that are defined in `fform`. We define an equivalent relation between the validity of formulas of `fform` and of *bool*, the built-in type of Isabelle logical formulas, represented as follows:

formT (f :: fform) ⇔ ⊢ f

where the function `formT` transforms a formula of type `fform` to a corresponding formula of *bool*. This approach enables us to prove atomic formulas `f` of `fform` by applying the built-in arithmetic solvers of Isabelle and proving `formT (f)` instead.

Duration Calculus Encoding DC into different proof assistants has been studied, such as [14] in PVS, and [3, 13] in Isabelle/HOL. DC can be considered as an extension of Interval Temporal Logic (ITL) by introducing state durations (here point formulas instead), while ITL an extension of FOL with the introducing of temporal variables and chop modality by regarding intervals instead of points as worlds. Therefore, both [3] and [13] apply an incremental approach to encode ITL on top of an FOL sequent calculus system, and then DC on top of ITL. We will follow a different approach here, to represent DC formulas as a datatype, as a result, the proving of DC formulas can be done by inductive reasoning on the structures of the formulas.

The datatype `dform` encodes the history formulas *HF*:

datatype dform = [[True]] | [[False]] | dexp[[=]]dexp | dexp[[<]]dexp
| [[¬]]dform | dform[[∨]]dform |[[∀]] *string* dform | pf fform | dform⌢dform

We will get rid of double brackets for readability if without confusion in context. The datatype `dexp` defines expressions that are dependent on intervals. As seen from *HF*, it includes the only temporal variable ℓ for representing the length of the interval, and real constants. Given a state formula `S` of type `fform`, `pf S` encodes the point formula $\lceil S \rceil^0$, and furthermore, the following `high S` encodes formula $\lceil S \rceil$:

high :: fform ⇒ dform
high S ≡ ¬ (True ⌢pf (¬S)⌢ ℓ > Real 0)

The chop modality ⌢ can be encoded as well.

To establish the sequent calculus style deductive system for `dform`, we first define the deductive system for the first-order logic constructors of `dform`, which can be taken directly from the one built for `fform` above, and then define the deductive system related to the new added modalities for DC, i.e. ℓ, ⌢ and `pf`.

For ℓ and ⌢, we encode the deductive system of ITL from [17], which is presented in Hilbert style. Thus, we need to transform the deductive system to sequent calculus style, and it is not so natural to do. We borrow the idea from [13] that for each modality, define both the left and right introducing rules, e.g., the following implementation

LI : \$H, P ⊢ \$E ⇒ \$H, P⌢(ℓ = Real 0) ⊢ \$E
RI : \$H ⊢ P, \$E ⇒ \$H ⊢ P⌢(ℓ = Real 0), \$E

where `$H`, `$E` represent arbitrary sequences of logical formulas of type `dform`, encodes the axiom of ITL: $P \leftrightarrow P^\frown(\ell = 0)$. In the same way, for point formula `pf`, we encode the deductive system of DC defined in [17] in sequent calculus style, e.g., the following implementation

PFRI : $H ⊢ (pf S_1 ⌢ pf S_2), $E ⇒ $H ⊢ pf ($S_1 \wedge S_2$), $E

encodes the axiom of DC: $\lceil S_1 \rceil^0 {}^\frown \lceil S_2 \rceil^0 \rightarrow \lceil S_1 \wedge S_2 \rceil^0$.

4.2 HCSP Syntax

We represent HCSP processes as a datatype `proc`, and each construct of HCSP can be encoded as a construct in datatype `proc` correspondingly. Most of the encoding is directly a syntactic translation, but with the following exceptions:

- In the deductive verification of HCSP process, the role of differential equation is reflected by an differential invariant with respect to the property to be verified, which can be automatically discovered in polynomial cases. So in `proc`, instead of differential equation, we use differential invariant to describe the underlying continuous, and for aiding verification, we also add execution time range of the continuous. Thus, we encode continuous of form $\langle \mathcal{F}(\dot{s}, s) = 0\&B \rangle$ as `<Inv&B> : Rg`, where `Inv` represents the differential invariant of the continuous, `B` the domain constraint, and `Rg` the range of execution time, of the continuous respectively; and `Inv`, `B` are implemented as formulas of type `fform`, while `Rg` of type `dform`.
- For sequential composition, we encode $P; Q$ as `P; mid; Q`, where `P` and `Q` represent the encodings of P and Q respectively, and `mid` is added to represent the intermediate assertions between P and Q. This is requisite for reducing proof of sequential composition to the ones of its components, and commonly used in theorem proving.
- For parallel composition, we remove the syntax restriction that it can only occur in the outmost scope, thus it is encoded with the same datatype `proc` as other constructs.

4.3 Verification Condition

Based on the inference rules of HHL, we implement the verification condition generator for reasoning about HCSP specifications. The inference rules encoded here are slightly different from those presented in [6], in the sense that we remove the point formulas for specifying discrete changes in history formulas and use $\ell = 0$ instead. This will not affect the expressiveness and soundness of HHL.

In deep embedding, the effects of assignments are expressed at the level of formulas by substitution. We implement a map as a list of pairs `(exp * exp) list`, and then given a map σ and a formula `p` of type `fform`, we define function `substF(`σ`, p)` to substitute expressions occurring in `p` according to the map σ. Based on this definition, we have the following axiom for assignment `e:=f`:

axioms Assignment :
  ⊢ (p → substF ([(e, f)], q)) ∧ (ℓ = Real 0 →G) ⇒ {p} (e :=f) {q; G}

According to the rule of assignment, the weakest precondition of `e := f` with respect to postcondition `q` is `substF ([(e, f)], q)`, and on the other hand, the strongest history formula for assignment is `ℓ= Real 0`, indicating that as a discrete action, assignment takes no time. Therefore, `{p} (e :=f) {q;G}` holds, if `p` implies the weakest precondition, and moreover, `G` is implied by the strongest history formula.

For continuous `<Inv & B> : Rg`, we assume that the precondition can be separated into two conjunctive parts: `Init` referring to initial state of continuous variables, and `p` referring to other distinct variables that keep unchanged during continuous evolution. With respect to precondition `Init∧p`, according to the rule of continuous, when it terminates (i.e. `B` is violated), the precondition `p` not relative to initial state, the closures of `Inv` and of `¬B` hold in postcondition; moreover, there are two cases for the history formula: the continuous terminates immediately, represented by `ℓ= Real 0`, or otherwise, throughout the continuous evolution, `p`, `Inv` and `B` hold everywhere except for the endpoint, represented by `high (Inv∧p∧B)`, where both cases satisfy `Rg`.

axioms Continuous : ⊢(Init → Inv) ∧ ((p ∧ close(Inv) ∧ close(¬B)) →q)
  ∧ ((((ℓ = Real 0) ∨ (high (Inv ∧ p ∧ B))) ∧ Rg) → G)
  ⇒ {Init ∧ p} <Inv & B> : Rg {q; G}

where function `close` returns closure of corresponding formulas. The above axiom says that `{Init∧p} <Inv & B> : Rg {q;G}` holds, if the initial state satisfies invariant `Inv`, and furthermore, both `q` and `G` are implied by the postcondition and the history formula of the continuous with respect to `Init∧p` respectively.

For sequential composition, the intermediate assertions need to be annotated (i.e., `(m, H)` below) to refer to the postcondition and the history formula of the first component. Therefore, the specification `{p} P;(m, H);Q {q;H⌢G}` holds, if both `{p} P {m;H}` and `{m} Q {q;G}` hold, as indicated by the following axiom.

axioms Sequence : {p} P {m; H}; {m} Q {q; G} ⇒{p} P; (m, H); Q {q; H⌢G}

The following axiom deals with communication `P1; ch!e || P2;ch?x`, where `P1` and `P2` stand for sequential processes. Let `p1` and `p2` be the preconditions for the sequential components respectively, and `(q1, H1)`, `(q2, H2)` the intermediate assertions specifying the postconditions and history formulas for `P1` and `P2` respectively. `r1` and `G1` represent the postcondition and history formula for the left sequential component ended with `ch!e`, while `r2` and `G2` for the right component ended with `ch?x`. `Rg` stands for the execution time range of the whole parallel composition.

axioms Communication :
  {p1, p2} P1 || P2 {q1, q2; H1, H2};
  ⊢ (q1 → r1) ∧ (q2 →substF ([(x, e)] , r2));
  ⊢ (H1 ⌢ high (q1)) →G1) ∧ (H2 ⌢ high (q2)) →G2);
  ⊢ (((H1 ⌢ high (q1)) ∧ H2) ∨ ((H2 ⌢ high (q2)) ∧ H1)) →Rg;
    ⇒ {p1, p2} ((P1; (q1, H1); ch !! e) || (P2; (q2, H2); ch ?? x))
        {r1, r2; G1 ∧ Rg, G2 ∧ Rg}

As shown above, to prove the final specification, the following steps need to be checked: first, the corresponding specification with intermediate assertions as postconditions and history formulas holds for `P1 || P2`; second, after the communication is done, for the sending party, `q1` is preserved, while for the receiving party, `x` is assigned to be `e`. Thus, `r1` must be implied by `q1`, and `q2` implies the weakest precondition of the communicating assignment with respect to `r2`, i.e. `substF ([(x, e)], r2)`; third, for the communication to take place, one party may need to wait for the other party to be ready, in case that `P1` and `P2` do not terminate simultaneously. The left sequential component will result in history formula $\texttt{H1}\frown\texttt{high (q1)}$, in which `high (q1)` indicates that during waiting time, the postcondition of `P1` is preserved, and similarly for the right component. Thus, `G1` and `G2` must be implied by them respectively; and finally, for both cases when one party is waiting for the other, the conjunction of their history formulas must satisfy the execution time `Rg`.

For repetition, we have the following implementation:

axioms Repetition :
{p1, p2} P || Q {p1, p2; H1, H2}; $\vdash$(H1 $\frown$ H1 $\rightarrow$H1) $\wedge$ (H2 $\frown$ H2 $\rightarrow$H2)
$\Rightarrow$ {p1, p2} P^* || Q^* {p1, p2; H1 $\vee$ (ℓ = Real 0), H2 $\vee$ (ℓ = Real 0)}

The above axiom says that the final specification for $\texttt{P}^*\texttt{||}\ \texttt{Q}^*$ holds, if the same specification holds for one round of execution, i.e. `P || Q`, and moreover, `H` is idempotent with respect to chop modality. The formula ℓ`= Real 0` indicates that the repetition iterates zero time.

4.4 Soundness

First, we define the operational semantics of HCSP by function `evalP` (only the case for sequential processes is presented here):

consts evalP :: proc $\Rightarrow$ cstate $\Rightarrow$ *real* $\Rightarrow$ proc $*$ cstate $*$ *real*

where `cstate` is of the form *real* $\Rightarrow$ `state list`. Each state of type `state` assigns respective values to process variables; and each element of type `cstate`, called by a *behavior*, associates a sequence of states to each time point. A behavior defines the execution history of a process, and is able to reflect super-dense computation by recording all the discrete changes in the sequence of states respectively at a time point. Given a process `P`, an initial behavior `f`, an initial time `a`, the transition `evalP (P,`f`, a) = (P',`f`', b)` represents that executing from behavior `f` at time `a`, `P` evolves to `P'` and ends at behavior `f'` and time `b`.

We then define the validity of a specification `{p} P {q;H}` with respect to the operational semantics, as follows:

definition Valid :: fform $\Rightarrow$ proc $\Rightarrow$ fform $\Rightarrow$ fform $\Rightarrow$ *bool*
where Valid (p, P, q, H) = $\forall$ f d f' d'. evalP (P, f, d) = (Skip, f', d') $\Rightarrow$
evalF (f, p, d) $\Rightarrow$ (evalF (f', q, d') & ievalF (f', H, d, d'))

which says that, given a process `P`, for any initial behavior `f` and initial time `d`, if `P` terminates at behavior `f'` and time `d'`, and if the precondition `p` holds under the initial state, i.e. the last element in state list `f(d)` (represented by `evalF (f, p, d)`), then the postcondition `q` will hold under the final state, i.e. the last element in state list `f'(d')` (represented by `evalF (f', q, d')`), and the history formula will hold under `f'` between `d` and `d'` (represented by `ievalF (f', H, d, d')`).

Based on the above definitions, we have proved the soundness of the proof system in Isabelle/HOL, i.e. all the inference rules of the proof system are valid.

5 Proof of the Combined Scenario

Under the given assumptions in Section 3.4, we need to check whether the combined scenario (i.e. model *System*) satisfies a liveness property, i.e., the train will eventually move beyond location x_2 for both level transition and mode transition. In this section, instead of proving the liveness property directly, we provide a machine-checked proof for negation of the livness, which says, after moving for any arbitrary time, the train will always stay before location x_2. We start from encoding the model *System* and the negation property first.

According to HCSP syntax implemented by `proc`, most encoding of model *System* is a direct translation, except for continuous and sequential composition. Firstly, the continuous of *System* needs to be represented in the form of differential invariants. According to the differential invariant generation method proposed in [7], the differential invariant $(a = -b) \to \mathit{DSP\_Form}$ is calculated for the continuous, indicating that when the train brakes with maximum deceleration b, it will never exceed the dynamic speed profile. Obviously it is a complement to the domain constraint B_2, saying that the train will never exceed the dynamic speed profile except for the case of emergency brake. We adopt the conjunction of these two formulas, that results in *DSP_Form*, as the final invariant for the continuous. Thus we represent the continuous as `<Inv&B> : Rg`, where `Inv` and `B` correspond to encodings of *DSP_Form* and the domain constraints respectively, and `Rg` is `True`, specifying the executing time of the continuous; Secondly, the intermediate formulas for all sequential composition are added. We finally get the encoding of *System*, represented by `System` below.

Now it is turn to encode the negation property, specified by pre/post-conditions, and history formula. The precondition is separated into two parts depending on whether it is relative to initial values, shown by `Init` and `Pre` below:

definition Init :: form **where** Init ≡ (x2 − s > Real 300)
definition Pre :: form **where**
Pre ≡ (level = Real 2.5) ∧ (fst (snd (snd (hd (MA)))) = x2)
    ∧ (snd (snd (snd (hd (MA)))) = String “FS”)
    ∧ (snd (snd (snd (hd (tl (MA))))) = String “CO”)
    ∧ (fst (hd (MA)) = Real 0) ∧ (fst (snd (hd (MA))) = Real 0)

The `Init` represents that the initial position of the train (i.e. `s`) is more than 300 meters away from `x2`. The `Pre` indicates the following aspects: the train moves at level `2.5`, i.e. in process of level transition from CTCS-2 to CTCS-3; the end of current segment is `x2`; the mode of the train in current segment is `‘‘FS’’`; the mode of the train in next segment is `‘‘CO’’`; and at the end of current segment, both speed limits are initialized to be `0`. Notice that for any segment *seg*, $seg.v_1$ is implemented as `fst (seg)`, and $seg.v_2$ as `fst (snd (seg))`, and so on.

We then get a specification corresponding to the negation property, with the postcondition and history formula for the train to indicate that the train will never pass through location x_2:

theorem System_proof : {Init ∧ Pre, True, True, True, True} System
{Pre ∧ s <= x2, True, True, True, True;
(ℓ = Real 0) ∨ (high (Pre ∧ s <= x2)), True, True, True, True}

In Isabelle/HOL, we have proved this specification as a theorem. From this fact, we know that the model `System` for level transition and mode transition fails to conform to the liveness property. This reflects some design flaw for the specifications of related scenarios in CTCS-3.

6 Conclusion and Future Work

In this paper, we have studied the formalization and verification of the scenarios defined in SRS of CTCS-3, by using HCSP and HHL as the modelling and specification languages respectively. We consider a combination of several basic scenarios, which is expected to conform to a liveness property according to SRS of CTCS. Especially, we have shown in the case study how to construct the model for the combined scenario from the separate ones corresponding to basic scenarios involved in it. The modelling technique can be applied to train control systems in general, especially for other combined scenarios of CTCS-3. For tool support, we have implemented a theorem prover in Isabelle/HOL for verifying HCSP models annotated with HHL assertions, within which we have proved the violation of the combined scenario with respect to the liveness property.

Future Work First of all, the case study in this paper is only a first step towards the formal checking of the correctness of SRS of CTCS-3, and we will study the whole SRS of CTCS-3 in forthcoming research. Second, the automation of the theorem proving implementation of HCSP in Isabelle/HOL is not considered currently, which needs the incorporation of existing tools (e.g. automatic SMT solvers) for arithmetic and the decision procedure implementation for subset of DC (i.e. one assertion language included in HHL). Finally, because of non-compositionality of HHL proposed in [6], the proof system is incomplete to prove all HCSP processes. These three aspects constitute our main future research.

Acknowledgements. The work has been supported mainly by projects NSFC-91118007, NSFC-6110006, and National Science and Technology Major Project of China (Grant No. 2012ZX01039-004).

References

1. R. Alur, C. Courcoubetis, T. A. Henzinger, and P. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems, LNCS 736*, pages 209–229. Springer-Verlag, 1992.

2. J. He. From CSP to hybrid systems. In *A Classical Mind, Essays in Honour of C.A.R. Hoare*, pages 171–189. Prentice Hall International (UK) Ltd., 1994.
3. S. T. Heilmann. *Proof Support for Duration Calculus.* PhD thesis, Technical University of Denmark, 1999.
4. T. A. Henzinger. The theory of hybrid automata. In *LICS'96*, pages 278–292. IEEE Computer Society, 1996.
5. J. Hoenicke and E. Olderog. CSP-OZ-DC: A combination of specification techniques for processes, data and time. *Nord. J. Comput.*, 9(4):301–334, 2002.
6. J. Liu, J. Lv, Z. Quan, N. Zhan, H. Zhao, C. Zhou, and L. Zou. A calculus for hybrid CSP. In *APLAS'10*, pages 1–15. Springer, 2010.
7. J. Liu, N. Zhan, and H. Zhao. Computing semi-algebraic invariants for polynomial dynamical systems. In *EMSOFT '11*, pages 97–106. ACM, 2011.
8. Z. Manna and A. Pnueli. Verifying hybrid systems. In *Hybrid Systems, LNCS 736*, pages 4–35. Springer, 1993.
9. Z. Manna and H. Sipma. Deductive verification of hybrid systems using STeP. In *HSCC'98, LNCS 1386*, pages 305–318. Springer, 1998.
10. B. C. Moszkowski and Z. Manna. Reasoning in interval temporal logic. In *Logic of Programs*, pages 371–382. Springer, 1983.
11. A. Platzer. Differential dynamic logic for hybrid systems. *Journal of Automated Reasoning*, 41(2):143–189, 2008.
12. A. Platzer and J. Quesel. European train control system: A case study in formal verification. In *ICFEM*, pages 246–265. Springer, 2009.
13. T. M. Rasmussen. *Interval Logic - Proof Theory and Theorem Proving.* PhD thesis, Technical University of Denmark, 2002.
14. J. U. Skakkebaek and N. Shankar. Towards a duration calculus proof assistant in PVS. In *FTRTFT'94*, pages 660–679. Springer, 1994.
15. M. Wildmoser and T. Nipkow. Certifying machine code safety: Shallow versus deep embedding. In *TPHOLs 2004*, pages 305–320. Springer, 2004.
16. S. Zhang. *CTCS-3 Technology Specification.* China Railway Publishing House, 2008.
17. C. Zhou and M. R. Hansen. *Duration Calculus: A Formal Approach to Real-Time Systems.* Series: Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2004.
18. C. Zhou, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, **40**(5):269–276, 1991.
19. C. Zhou and X. Li. A mean-value duration calculus. In *A Classical Mind, Essays in Honour of C.A.R. Hoare*, pages 432–451. Prentice-Hall International, 1994.
20. C. Zhou, J. Wang, and A. P. Ravn. A formal description of hybrid systems. In *Hybrid systems, LNCS 1066*, pages 511–530. Springer, 1996.