SCIENCE CHINA
Information Sciences Manuscript

• RESEARCH PAPER •

January 2015, Vol. 58 xxxxxx:1–xxxxxx:26
doi: xxxxxxxxxxxxxx

Behavior Modeling and Verification of Movement Authority Scenario of Chinese Train Control System using AADL

AHMAD Ehsan[1,2], DONG Yunwei[1], LARSON Brian R.[3], LV JiDong[4], TANG Tao[4] & ZHAN NaiJun[2]*

[1]*School of Computer Science, Northwestern Polytechnical University, Xi'an,* 710072*, China;*
[2]*State Key Lab. of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing,* 100190*, China;*
[3]*Computing and Information Systems, Kansas State University, Manhattan, KS* 66506*, USA;*
[4]*State Key Lab. of Rail Traffic Control and Safety, Beijing Jiaotong University, Beijing,* 100044*, China*

Received January 1, 2015; accepted January 1, 2015; published online January 1, 2015

Abstract Train control systems like most digital controllers are, by definition, hybrid systems as they interact with or try to control some aspects of the physical world. Detailed behavior modeling with constraints specification and formal verification, required for reliability prediction, is a great challenge for hybrid system designers. Train control systems further intensify this challenge with extensive interaction between computing units, and their physical environment, and their mutual dependence on each other. In this paper, we investigate behavior modeling and formal verification of Chinese Train Control System Level 3 (CTCS-3) using Architectural Analysis & Design Language (AADL) to cope with this challenge. AADL is an architecture description language for embedded systems and is based on model-based engineering paradigm. Along with structural modeling of embedded systems using the core language constructs, AADL also provides support for language extension through annex sublanguage. In system requirements specification document, the behavior of the CTCS-3 is specified as a set of basic operation scenarios that cooperate with each other to achieve safe and secure functionality of trains. Movement Authority scenario, explored in this paper, is considered as a basic and most crucial scenario to prohibit trains from colliding with each other. The detailed discrete behavior of control system is modeled and verified using the Behavior Language for Embedded Systems with Software (BLESS) annex sublanguage of AADL, and the continuous behavior of train with the cyber-physical interaction (communication between train, and control system) is modeled using the Hybrid annex sublanguage. The behavior of the movement authority scenario at system level is verified using the Hybrid Hoare Logic (HHL) theorem prover. Behavior constraints are specified as assertions using first-order logic formulas augmented with a simple temporal operator.

Keywords AADL, Behavior modeling, BLESS annex, CTCS, Hybrid annex, Train control system

Citation AHMAD, DONG, LARSON, LV, TANG, ZHAN. Behavior Modeling and Verification of Movement Authority Scenario of Chinese Train Control System using AADL. Sci China Inf Sci, 2015, 58: xxxxxx(26), doi: xxxxxxxxxxxxxx

* Corresponding author (email: znj@ios.ac.cn)

1 Introduction

Train control system with stringent reliability, safety, and performance requirements, is the core element of any railway operations management system. Significant increase in train traffic, and modern infrastructure strategies to accommodate high levels of traffic while reducing system costs require a standardized and dependable train control system. The System Requirements Specification (SRS) document [9] of Chinese Train Control System Level 3 (CTCS-3) provides standardized specification of the control system architecture, and its behavior. The behavior of the control system is described by 14 basic operation scenarios consisting of different architectural components, and communication among these components.

The CTCS-3 is a *hybrid system* because it interacts with and tries to control some aspect of the physical world—which is the movement of train in this case. In a typical hybrid system, the control system, consisting of embedded computing units with the discrete behavior, interacts with its physical environment with the continuous behavior, to monitor and control those physical quantities necessary to ensure correct system functionality. The behavior of a hybrid system is determined by not only the continuous evolution of controlled variables in the physical environment, but also the communication events (between the control system and its physical environment) which drive transitions among evolutions. Evolution in continuous domain causes jumps in discrete domain, which in turn promotes continuous evolution by (re)setting appropriate controlled variables. Continuous evolutions, discrete jumps, and communication events along with behavioral constraints are considered the essential elements of detailed design, and together pose a great challenge to correct hybrid system modeling and verification. For train control systems, extensive interaction between its computing units (Radio Block Center, Vital Computer, Driver Machine Interface, etc.), and their environment (including the dynamics of train), and their mutual dependence on each other further intensify this challenge.

Architectural Analysis & Design Language (AADL) is an SAE International standard and is based on architecture-centric model-based engineering approach for the design of embedded systems [11]. AADL provides support for structural modeling of embedded computing units (control software, and the platform on which it runs), and does not define detailed behavior modeling required for extensive formal analysis needed for dependability prediction. It has been successfully applied in highly safety-critical domains like medical and aerospace. Along with structural modeling of embedded systems using the core language constructs, AADL also provides support for extension through annex sublanguages.

To equip AADL for detailed behavior modeling and verification of hybrid systems, the Behavior Language for Embedded Systems with Software (BLESS) and Hybrid annexes have been introduced [3, 7]. BLESS uses a state transition system with guards and actions to model the discrete behavior of a control system. Hybrid annex uses process algebra notations to model the continuous behavior, and interaction between computing units, and their physical environment. Both of these annexes support behavior constraints specification through first-order logic formulas augmented with a simple temporal operator, as assertions. For discrete behavior verification, BLESS provides a proof tool for automatic generation of proof obligations, and formal proofs with human guidance using axioms and inference rules. Hybrid annex uses Hybrid Hoare Logic (HHL) Prover, an interactive theorem prover, to verify AADL models annotated with the Hybrid annex specifications.

In this paper we present the results of an effort focused on modeling and analysis of the operation scenarios of the CTCS-3 with AADL. We consider the *movement authority* scenario which is the most basic one, and is crucial to prohibit trains from colliding with each other. The detailed discrete behavior of the control system is modeled and verified using the BLESS annex while the continuous behavior of train is modeled using the Hybrid annex, and furthermore, the system level behavior verification is performed using the HHL Prover. The behavior constraints are specified as assertions.

1.1 Contributions

Our first contribution is that we show how safety-critical hybrid systems can be modeled and verified in an *integrated* development environment which consists of AADL, and the BLESS and Hybrid annexes, and where the artificial separation between discrete and continuous domains has been erased. AADL support integration through an effective mechanism for component contract specification based on interfaces and interactions, and through well defined semantics for extensive formal analysis at different architecture levels. This integration not only supports requirement identification for both discrete and continuous variables, but can also facilitate to assess correct operations of the physical portion of a hybrid system through several dependability related analysis, and the certification of systems level behavior correctness.

Our second contribution is that we characterize detailed behavior modeling, and certification of three important system level properties of the movement authority scenario of the CTCS-3. Keeping in view the essential hybrid system design elements, we identify all the operational constraints, realize the discrete behavior modeling of the control system along with the continuous behavior modeling of its physical environment with the cyber-physical interaction, and verify the operational safety of trains, under the movement authority scenario. The cyber-physical interaction, a major design challenge for hybrid systems, is modeled as communication events performed along data ports specified in type classifiers of AADL components. These communication events realize communication interrupts to preempt continuous evolution of controlled variables, modeling the physical dynamics of a train, according to the newly devised control strategy by the control system.

1.2 Outline

Section 2 briefly introduces HCSP, and AADL with the BLESS, and Hybrid annexes. Section 3 presents an overview of the movement authority scenario of the CTCS-3 by highlighting its operation with important behavior constraints. Section 4 describes interface, and behavior modeling of the components involved in the scenario. Section 5 discusses formal verification of the control system with discrete behavior using the BLESS proof tool, and Section 6 describes system level behavior verification of the movement authority scenario using the HHL Prover. Section 7 summarizes the related work while Section 8 summarizes this paper.

2 Preliminaries

This section presents an overview of HCSP by highlighting primitive language constructs. AADL is also introduced with emphases on its abilities for architectural and behavioral modeling of embedded systems. Behavioral modeling capabilities of the AADL are described through the BLESS, and Hybrid annexes.

2.1 Overview of HCSP

Hybrid Communication Sequential Processes (HCSP) is an extension of Hoare's Communicating Sequential Processes (CSP) for modeling and verifying hybrid systems [10,19]. In HCSP, differential equations are introduced to model continuous evolution of the physical environment along with interrupts, so both the discrete, and continuous behaviors are still modeled as *processes*. A hybrid system in HCSP, is a parallel composition of networked sequential processes interacting through dedicated channels, or a repetition of a sub-system. The concurrent processes can only interact through communication, and no shared variables are allowed. The set of variables is denoted by $\mathcal{V} = \{x,\ y,\ z,...\}$ and the set of channels is denoted by $\Sigma = \{ch_1, ch_2, ch_3, ...\}$. The processes of HCSP are constructed as follows:

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

Here P, Q, and Q_i represent sequential processes, whereas S stands for a (sub)system, $ch, ch_i \in \Sigma$ are communication channels, while $ch_i{*}$ is a communication event which can be either an input event $ch?x$ or an output event $ch!e$, B, and e are the boolean, and arithmetic expressions, respectively, and d is a non-negative real constant.

Process **skip** terminates immediately without updating variables, and process $x := e$ assigns the value of expression e to variable x and then terminates. Process *wait* d keeps idle for d time units without changing the variables. Interaction between processes is based on two types of communication events: $ch!e$ sends the value of e along channel ch and $ch?x$ assigns the value received along channel ch to variable x. Communication takes place when both the source, and the destination processes are ready.

HCSP supports both sequential and concurrent composition. A sequentially composed process $P; Q$ behaves as P first, and if it terminates, as Q afterwards. The alternative process $B \to P$ behaves as P only if B is true and terminates otherwise. Internal choice between processes P and Q, denoted as $P \sqcup Q$ is resolved by the process itself. Communication controlled external choice $[]_{i\in I}(ch_i* \to Q_i)$ specifies that as soon as one of the communications ch_i* takes place, the process starts behaving as process Q_i. The repetition P^* executes P for an arbitrary finite number of times, and the choice of the number of times is non-deterministic.

Continuous evolution is specified as $\langle \mathcal{F}(\dot{s}, s) = 0\&B\rangle$. The real variable s evolve continuously according to differential equations $\mathcal{F}$ as long as the boolean expression B is true. B defines the domain of s. Interruption of the continuous evolution due to B (as soon as it becomes false) is known as *Boundary Interrupt*. The continuous evolution can also be preempted due to the following interrupts:

- *Timeout Interrupt*: $\langle \mathcal{F}(\dot{s}, s) = 0\&B\rangle \trianglerighteq_d Q$ behaves like $\langle \mathcal{F}(\dot{s}, s) = 0\&B\rangle$, if the continuous evolution terminates before d time units. Otherwise, after d time units of evolution according to $\mathcal{F}$, it behaves as Q.
- *Communication Interrupt*: $\langle \mathcal{F}(\dot{s}, s) = 0\&B\rangle \trianglerighteq []_{i\in I}(ch_i* \to Q_i)$ behaves like $\langle \mathcal{F}(\dot{s}, s) = 0\&B\rangle$, except that the continuous evolution is preempted whenever one of the communications ch_i* takes place, which is followed by respective Q_i.

Finally, S defines a HCSP system on the top level. A parallel composition $S_1\|S_2$ behaves as if S_1 and S_2 run independently except that they need to synchronize along the common communication channels. Formal syntax, and operational semantics of HCSP are further detailed in [4].

The Hybrid Hoare Logic, specification logic for reasoning about HCSP behavior, is introduced in Section 6.1.

2.2 Overview of AADL

AADL supports dependability prediction through *analyzable* architecture development. Due to its extensive support for modeling (abstraction, reusability, composition, etc.), and its substantial analysis capabilities, AADL is a strong candidate for modeling safety-critical systems and has been used in domains like medical, and aerospace engineering. System Architecture Virtual Integration (SAVI), an important collaborative project for designing complex distributed aerospace systems, has selected AADL as its architecture description language [12]. AADL supports *virtual integration*, the essence of the SAVI project, through component contract specification.

An embedded system architecture in AADL, consists of connected components specified by their *type* and *implementation* classifiers for both the application software, and its execution platform. Components are connected through externally visible interfaces called *ports* specified in the *features* section of the type classifier. Port communication is typed and directional, and is used to transmit and receive data, control, and data with control through *data* ports, *event* ports, and *event data* ports, respectively. Internal structure of a particular component is realized by specifying its subcomponents and the connections between them in its implementation classifier.

The application software is modeled primarily with *process*, *data*, *subprogram*, and *thread* components. Process components model the protected memory space shared among thread subcomponents that represent sequential control flow. Data components model types, persistent values, or parameters of a subprogram. Subprogram components model computation.

The execution platform is modeled primarily with *processor*, *memory*, *bus*, and *device* components. The hardware that executes the software is modeled with processor components. Memory components model data storage. Device components model other physical entities like actuators and sensors, or custom logic. Bus components model the physical connection between execution platform components. AADL allows binding of logical components and connections to physical components.

AADL also provides *system* components to model the composition of logical and physical components, and *abstract* components to model interfaces without further elaboration.

The core AADL standard only provides support for structural modeling of embedded computing systems and nothing related to the detailed behavior of the software, and the physical environment which is controlled by the software can be modeled. The BLESS, and Hybrid annexes were created to precisely model the behavior of the control system, and the physical envrionmt. AADL together with the BLESS, and Hybrid sublanguages, is capable of fully modeling the hybrid systems in an integrated manner. The detailed discrete behavior of the computing units is modeled using the BLESS annex, and the continuous behavior of the physical environment and the cyber-physical interaction is modeled using the Hybrid annex.

2.2.1 *BLESS annex*

Behavior Language for Embedded Systems with Software (BLESS) was created to specify behavior of component interfaces, define formal semantics for component implementations, and provide tool support for reasoning about the compliance of behaviors to component's specifications. A BLESS subclause annotates component classifiers with state machines using guards and actions to model the detailed discrete behavior. BLESS subclauses have sections for `variables`, `states`, and `transitions` to specify local variables, behavior states and transitions among states. A state can be labeled as `initial`, `complete`, or `final` while the unlabeled states are considered as *execute* states. A complete behavior starts from an initial state, suspends at complete states until next dispatch and terminates at a final state while checking guards and performing actions upon each transition. Execute states are transients such that the delay between dispatch and suspension is negligible.

Sections `assert`, and `invariant` are used to specify assertions, and the predicates that must hold throughout the behavior model. BLESS also provides a tool for automatic generation of proof obligations based on first-order logic formulas specified as *Assertions* 1). Proof obligations are then solved using interactive theorem proving to produce formal proof as a list of theorems, each of which is axiomatic, given, or derived from earlier theorems in the list by sound inference rules. Formal syntax, grammar, and details on proof tool of BLESS are presented in [13].

The use of the BLESS annex for behavior modeling is presented in Section 4.2 while the use of the BLESS proof tool for formal verification of the *Controller* component (described in Section 3.1.3) of the CTCS-3 is illustrated in Section 5.

2.2.2 *Hybrid annex*

Hybrid annex is inspired by HCSP with the added intention of impartially supporting other continuous behavior modeling tools and methodologies. Hybrid annex subclauses can be attached to either AADL device component implementations to model the continuous behavior of sensors and actuators, or to abstract component implementations to model the continuous behavior of the physical environment.

A Hybrid annex subclause contains six sections, each of which is dedicated to specifying a particular aspect of a continuous behavior model. Sections `assert` and `invariant` are used to specify assertions and predicates that must hold throughout the continuous behavior model's lifetime. Sections `variables` and `constants` are used to specify local variables and constant values. The continuous behavior with differential equations, and communication and time interrupts is specified in the `behavior` section as parallel composition of sequentially executing processes communicating through channels specified in the `channels` section. Each Hybrid annex construct used in this paper is fully explained in [7].

The use of the Hybrid annex for the continuous behavior modeling of the CTCS-3 is further explained in Section 4.3. AADL models annotated Hybrid annex specifications are formally verified using the HHL Prover. System level behavior verification of the movement authority scenario using the HHL Prover is explained in Section 6.

1) The capital 'A' proper noun signifies temporal logic formulas used by BLESS.

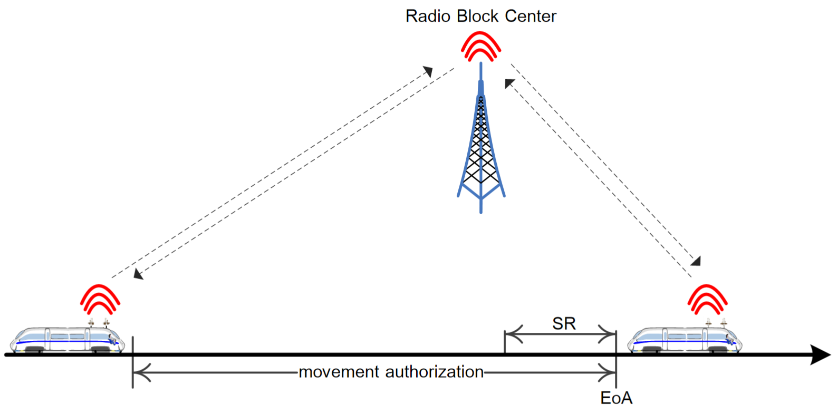

Figure 1 Trains coordination with dynamic movement authorizations in CTCS-3

3 Movement authority scenario of CTCS-3

Due to the nature of required functions, diversity of the operation scenarios, and the components used for each of these functions, the CTCS-3 is partly on the trackside and partly on board the trains, thus defining two subsystems: *on-board* subsystem, and *trackside* subsystem. On-board subsystem supervises movement of the train on the bases of information exchanged with the trackside subsystem, and the train to which it belongs.

The behavior of the CTCS-3 is defined in terms of 14 basic operation scenarios, all of which cooperate with each other to describe proper functionality of the train control system. The scenario considered in this paper is the Movement Authority (MA) scenario.

3.1 Involved components

The MA scenario involves three principal components: Radio Block Center, Train, and Controller.

3.1.1 *Radio Block Center*

The Radio Block Center (RBC) is a computer-based system that exchanges information with a train on basis of the data received from the trackside, and on-board subsystems. The main purpose of the RBC is to provide/extend movement authorities to allow safe movement of trains. As depicted in Figure 1, dynamic movement authority is assigned to a particular train depending on current track situation and movement of other trains within the region of responsibility of a particular RBC. A movement authority is further subdivided into several distance segments (16 segments at most), each of which is of 2 kilometers in length. Train can apply for movement authority at the start, or well before the end of movement authority (EoA), shown as *SR* in Figure 1.

3.1.2 *Train*

In the MA scenario, movement of the train, based on the continuous evolution of it's position and it's velocity, is represented by the *Train* [2] component. It transmits current position and velocity of the train and receives the computed acceleration to follow the velocity constrains in each segment of a particular movement authority. The control system (part of the On-board system) samples velocity, and position of the train periodically (usually after every 200 milliseconds) to compute new acceleration on the basis of current position and velocity of the train, and the information received form the RBC.

2) In the rest of the paper, "Train" is used to refer the modeling component while "train" is used to refer the physical train on the track.

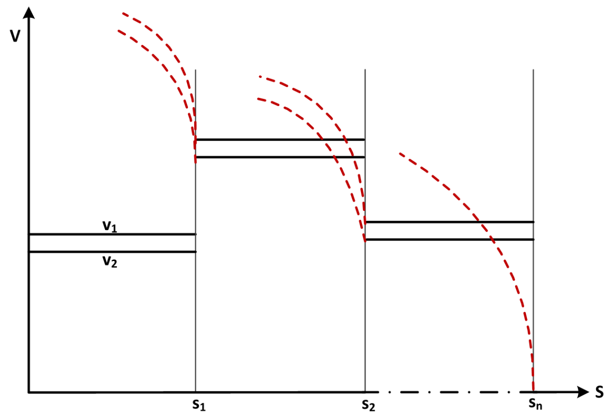

Figure 2 Static and dynamic speed profiles in a movement authority

3.1.3 *Controller*

In the MA scenario, part of the on-board subsystem responsible for determining train's location relative to the location that the train has permission to move, is represented by the *Controller* component. Main functionality of the Controller is to control the velocity, by setting the acceleration, of the train such that its movement is restricted to its movement authority. Depending on the current velocity, and position of the train, the Controller adjusts acceleration so that the train never runs beyond the *static* and *dynamic* speed profiles of a particular segment of the current movement authority and stops safely before the EoA, if the movement authority is not extended in time.

3.2 The MA scenario

According to the SRS document [9], in the CTCS-3, the train applies for movement authority from the RBC and if granted, it gets permission to move within the movement authority it owns. A movement authority is composed of a sequence of segments where each segment has: two speed limits (v_1 and v_2), a segment end point e, and a *mode* to represent operation mode of the train. Speed limits v_1 and v_2 (where $v_1 \geqslant v_2$) represent the constraints for the train to apply emergency, and normal service brake, respectively.

Operation of the Train and the Controller components, in the aforementioned scenario, forms a classical hybrid system with a control-feedback loop, where the Controller regulates velocity of the Train by adjusting acceleration or deceleration (with discrete actions) on the basis of current velocity and position (with continuous evolutions) of the train and the information exchanged with the RBC for movement authority extension. Once the acceleration has been regulated by the Controller, train continues moving according to the differential equations $\dot{s} = v$ and $\dot{v} = a$, where s is the position of the train while v and a denote its velocity and acceleration, respectively.

Figure 2 depicts a movement authority with segments s_1, s_2,...,s_n. The train must apply for new movement authority as it reaches to a specific distance (SR) from the EoA, and is required to fully stop at the end of the segment s_n if its movement authority is not extended. For a given movement authority, static and dynamic speed profiles are calculated for each segment. The static speed profiles are the regions formed by the speed limits as $v \leqslant seg.v_1$ and $v \leqslant seg.v_2$ corresponding to a two step function. The dynamic speed profiles (shown as dotted lines) are calculated for both v_1 and v_2 using train's maximum deceleration b and its position s according to equation $v_i^2 + 2bs \leqslant next(seg).v_i^2 + 2b(seg.e)$. The $next(seg)$ represents the segment next to the current segment. The speed of the train is constrained to static and dynamic profiles of the current segment.

When the extension of the movement authority has been granted, and the rear end of the train has passed the Balise (a device on the track, part of the trackside system) of the last segment of the current movement authority, that part of track is now available to be granted to another train as part of its movement authority by the RBC.

3.3 Behavior constraints

For safe movement of the train under the MA scenario, behavior of the involved components is restricted by following constraints.

Always moving forward: One of the important general restriction is that the train always moves forward with $v \geqslant 0$, or otherwise it has already stopped decelerating represented as $a \geqslant 0$, where $a \in [0, A]$, in which A is the maximum acceleration. We represent this restriction as the following boolean expression:

$$B_0 \mathrel{\widehat{=}} (v \geqslant 0 \vee a \geqslant 0 \vee Temp < t < Temp + T_{delay})$$

Notice that we add T_{delay} to clock t (the system time) to guarantee checking the condition is conducted every T_{delay} time units, to avoid *Zeno* behaviour. This is accordance with the real system to check the condition periodically with the sampling time 200 milliseconds.

Service brake intervention: First of all, the static speed profile of service brake intervention can not be violated, *i.e.*, $v < seg.v_2$, otherwise a service brake should be taken, *i.e.*, the acceleration a is set to a negative value. In addition, the condition is checked every T_{delay} time units also. Likewise, the dynamic speed profile of service brake intervention should be followed, otherwise, a fully brake should be taken, *i.e.*, $a = -b$. These two conditions are represented by B_1 and B_2 below, respectively.

$$\begin{aligned} B_1 &\mathrel{\widehat{=}} (\forall seg : v < seg.v_2) \vee a < 0 \vee Temp < t < Temp + T_{delay}) \\ B_2 &\mathrel{\widehat{=}} (\forall seg : v < seg.v_2) \wedge v^2 + 2bs < next(seg).v_2^2 + 2b\ seg.e) \vee a < 0 \end{aligned}$$

Emergency brake intervention: Any violation to the static and/or dynamic speed profile of emergency brake intervention results in a full brake, which is specified by the following two formulas:

$$\begin{aligned} B_3 &\mathrel{\widehat{=}} \forall seg : v > seg.v_1 \Rightarrow a = -b \\ B_4 &\mathrel{\widehat{=}} (\forall seg : v < seg.v_1) \wedge v^2 + 2es < next(seg).v_1^2 + 2e\ seg.e) \vee a = -b \end{aligned}$$

Movement authority extension: A train can move safely to a specific distance in the region of its movement authority. It needs to apply for extension of the current movement authority when it is at least *SR* distance far from the EoA (see Figure 1) and must stop before the EoA, if movement authority is not extended. Following boolean expression represents this restriction, where, s is the current position of the train.

$$B_5 \mathrel{\widehat{=}} (s >= EoA - SR)$$

The details for the modeling and verification of the MA scenario, described in this section, are presented in the rest of the paper. Behavior restriction B_0 is specified by modeling the Controller as an AADL `Periodic` component and assigning the value `200 ms` to its `Period` property, in the next section. Behavior restrictions B_1, B_2, B_3, and B_4 are specified as the BLESS Assertions `<<SBL()>>`, `<<DSPV2()>>`, `<<EBL()>>` and `<<DSPV1()>>`, respectively, in Section 5.1, and are discussed in detail in relevant sections. Behavior constraint B_5 is a transitory restriction and is specified in the action step of a particular transition `T3_Move_Check` in Listing 4, and is discussed in Section 4.2.

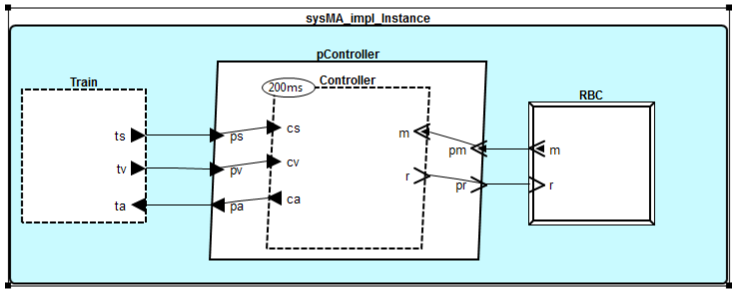

Figure 3 AADL model of MA scenario using graphical notations

4 Behavior modeling using AADL

This section describes component interface modeling using core AADL language constructs, the discrete behavior modeling with the BLESS annex, and the continuous behavior modeling with the Hybrid annex. As depicted in Figure 3, the RBC is modeled as an AADL device component `RBC`[3)], Train is modeled as an AADL abstract component `Train`, and the Controller is modeled as an AADL thread component `Controller` within a process component `pController`. Implementation classifier of `pController` contains `Controller` (as a subcomponent), and the connections between them. As process `pController` is only used to represent the protected memory space for thread `Controller` and its behavior depends upon the underlying thread (*i.e.*, `Controller`), type and implementation classifiers of `pController` are not specified here.

The `sysMA` models the whole MA scenario as an AADL system component and contains all other components (`RBC`, `pController`, and `Train`) as subcomponents. Subcomponents are connected though appropriate connections between their interfaces defined in the type classifier of each component.

4.1 Component interface modeling

4.1.1 *Radio block center*

The type classifier of Listing 1 declares interface of the `RBC` component. The `r` in event port is used to get request for the movement authority extension, and out event data port `m` is used to transmit newly computed movement authority. The `Controller` component is linked to `r`, and `m` with appropriate connections.

Listing 1 RBC component type

```
device RBC
  features
     r : in event port; --receives movement authority request
     m : out event data port CTCS_Types::MovementAuthority; --sends movement authority
end RBC;
```

We assume correct functionality of the RBC in the MA scenario and only focus on its communication behavior, implementation classifier of the `RBC` component is not presented here.

4.1.2 *Train*

Listing 2 specifies type classifier of the `Train` component with two out data ports `ts`, and `tv`, and one in data port `ta`. Out data ports `ts`, and `tv` are used to transmit train's position and velocity while `ta` is used to receive new acceleration value. The `Controller` is linked to `ts`, `tv`, and `ta` with appropriate connections.

The corresponding implementation classifier of the `Train` component is described in Listing 5, and is explained in Section 4.3, in detail.

3) Expressions written in `Typewriter Font Family` represent code snippet of the AADL model.

Listing 2 Train component type

```
abstract Train
  features
    ta : in data port CTCS_Types::Acceleration; --receives new acceleration
    ts : out data port CTCS_Types::Position; --sends current position
    tv : out data port CTC_Types::Velocity; --sends current velocity
end Train;
```

4.1.3 *Controller*

Type classifier of the `Controller`, in Listing 3, specifies interface of the Controller responsible for computation of new acceleration, and communication among the components involved in the MA scenario. The `Controller` receives the current position, and velocity of the train through in data ports `cs`, and `cv`, respectively, and transmits new computed acceleration to the train through out data port `ca`. Out event port `r` is used to send request for new movement authority when the train is at the start position, or for extension if the train is already moving. New movement authority is received at `m` in event data port.

Periodic execution of the `Controller` is specified by declaring `Dispatch_Protocol` property as `Periodic` in the `properties` section. The value of the property `Period` (*i.e.*, `200 ms`) specifies sampling period as described by the behavior constraint B_0.

Listing 3 Controller component type

```
thread Controller
  features
     r  : out event port; --sends movement authority request
     m  : in event data port CTCS_Types::MovementAuthority; --receives movement authority
    cs  : in data port CTCS_Types::Position; --receives train position
    cv  : in data port CTC_Types::Velocity; --receives train velocity
    ca  : out data port CTCS_Types::Acceleration; --sends new acceleration

 properties
    Dispatch_Protocol => Periodic;
    Period => 200 ms;
end Controller;
```

The corresponding implementation classifier for the `Controller` type is given in Listing 4, and is detailed in Section 4.2.

AADL Data components `MovementAuthority`, `Position`, `Velocity`, and `Acceleration` are defined in package `CTCS_Types` to specify the ranges of possible values, the variables of these types can take on, and their units of measure.

4.2 Discrete behavior modeling

What follows are the details of behavior modeling of the `Controller` component while its verification is detailed in Section 5. Listing 4 contains a BLESS state machine (without the BLESS Assertions) that captures the behavior of the `Controller` component in its implementation classifier. In the `variables` section, `iMA`, initialized with `null`, represents current movement authority while variables `iSeg`, and `nSeg`, both initialized with `null`, represent the current, and the next segment, respectively.

Based on the structure of movement authority and its segments, AADL Data component `Segment` is declared as of type *record* with elements `v1`, and `v2` for velocities, `e` for end of the segment, and `m` for mode of the train in a particular segment. Data component `MASegments` is declared as an array of type `Segment`. Data component `MovementAuthority` is then declared as of type *record* with two elements: `Segments` of type `MASegments`, and `EoA` of type `Position`. Data components `MovementAuthority`, `MASegments`, and `Segment` are defined in package `CTCS_Type`. Variable `i` is used for array indexing and is initialized with `0`.

Listing 4   Controller component implementation with BLESS annex

```
thread implementation Controller.impl
annex BLESS{**
 variables
    i   : Base_Types::Integer:=0; --segment index
    b   : CTCS_Types::Deceleration; --maximum deceleration as used in Assertions
    v   : CTCS_Types::Velocity; --train velocity
    s   : CTCS_Types::Position; --train position
    xl  : CTCS_Types::Acceleration; --calculated acceleration
   eoa  : CTCS_Types::Position; --End of Authority
   iMA  : CTCS_Types::MovementAuthority:=null; --movement authority
   iSeg, nSeg : CTCS_Types::Segment:=null; -- segments

 states
    READY : initial state;
    GMA   : complete state;
    CMA   : state;
    RETRY : state;
    MFR   : complete state;
    CMF   : state;
    SBI   : complete state;
    CSB   : state;
    EBI   : complete state;
    STOP  : final state;

 transitions
    T0_go: READY -[]-> GMA { r! };
    T1_MA_Check: GMA -[on dispatch]-> CMA { m?(iMA) ; eoa:=iMA.EoA };
    T2_MA_Ok: CMA -[not (iMA=null)]-> MFR {};
    T3_Move_Check: MFR -[on dispatch]-> CMF { { cs?(s) & cv?(v) };
                    if ( s>=(eoa-SR) )~>
                        r!
                    [] (not (s>=(eoa-SR)) )~>
                        skip
                    fi  };
    T4_SBI_Point: CMF -[not ((s=CTCS_Properties::start) or ((v < iSeg.v2))) or
                    not ((s=CTCS_Properties::start) or  ((((v**2) + (2*b*s)) <
                    (iMA[nSeg.v2] + (2*b*iSeg.e))))))]-> SBI
                    { ca!(CTCS_Properties::SB_Rate) };

    T5_Move_Ok: CMF -[((s=CTCS_Properties::start) or ((v < iSeg.v2)))
                    or  (((v**2) + (2*b*s)) < (iMA[nSeg.v2] + (2*b*iSeg.e)))]-> MFR
                    { ca!(xl); };
    T6_SBI_Check: SBI -[on dispatch]-> CSB { {cs?(s) & cv?(v)} };
    T7_SBI_Out: CSB-[ ((s=CTCS_Properties::start) or ((v < iSeg.v2))) and
                    ((s=CTCS_Properties::start) or
                    ((((v**2) + (2*b*s)) < (iMA[nSeg.v2] + (2*b*iSeg.e))))) ]-> MFR
                    { ca!(xl) };

    T8_SBI_Ok: CSB -[not (((s=CTCS_Properties::start) or ((v < iSeg.v2))) and
                    ((s=CTCS_Properties::start) or
                    ((((v**2) + (2*b*s)) < (iMA[nSeg.v2] + (2*b*iSeg.e)))))) ]-> SBI {};

    T9_MA_NotOk: CMA-[iMA=null]-> RETRY {};
    T10_MA_Retry: RETRY -[]-> GMA {};
    T11_EBI_Point: CMF -[not (v < iSeg.v1) or not ((((v**2) + (2*b*s)) <
                    (iMA[nSeg.v1] + (2*b*iSeg.e)))) ]-> EBI {};
    T12_Stop: EBI -[v=0]-> STOP {};
    T13_EBI_Point: SBI -[not (v < iSeg.v1) or not ((((v**2) +
                    (2*b*s)) < (iMA[nSeg.v1] + (2*b*iSeg.e))))]-> EBI {};
  **};

end Controller.impl;
```

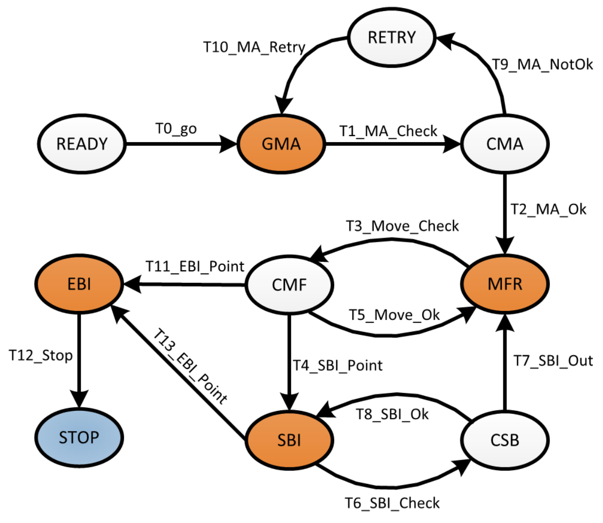

Figure 4 Controller behavior state machine

Variables s, v, and x1 are declared to store the position, velocity, and newly computed acceleration of the train. Variable b represents the maximum deceleration rate when emergency brake is applied while variable eoa is used to store the value of the EoA. Data components MovementAuthority, MASegments, and Segment are defined in package CTCS_Type. Variable i is used for array indexing and is initialized with 0. Variables s, v, and x1 are declared to store the position, velocity, and newly computed acceleration of the train. Variable b represents the maximum deceleration rate when emergency brake is applied while variable eoa is used to store the value of the EoA.

The states section of BLESS specification for the Controller component contains ten behavior states with READY as initial state; GMA (Get Movement Authority), MFR (Move Forward), SBI (Service Brake Intervention), and EBI (Emergency Brake Intervention) as complete states; RETRY, CMA (Check Movement Authority), CMF (Check Move Forward), and CSB (Check Service Brake) as execute states; and STOP as a final state. Transitions are specified in the transitions section and are of the form:

```
transition_name:
  source state(s) -[condition]-> destination state {action};
```

Transitions are named appropriately (*e.g.*, T2_MA_Ok), and each transition has one or more source states (*e.g.*, the CMA state of T2_MA_Ok), a single target state (*e.g.*, the MFR state of T2_MA_Ok), a guard condition (*e.g.*, not (iMA=null) of T2_MA_Ok), and a (possibly empty) set of actions to execute (*e.g.*, cv?(v) &cs?(s) of T6_SBI_Check). Only finite number of execute states are allowed to be passed before entering a complete or final state. Transitions leaving complete states may have on dispatch as guard conditions evaluated by the run time system (*e.g.*, T1_MA_Check, T3_Move_Check, etc.). Transitions leaving execute states may have a boolean expression as transition condition. Transition actions may contain sequential actions separated by ; operator or actions to be executed in any order separated by & operator, as cs?(s) &cv?(v) is specified in T6_SBI_Check.

Depicted in Figure 4, behavior of the Controller component is as follows. At the start when the train is ready to move, in the READY state, it applies for new movement authority along out event port r through T0_go. Controller then waits for the new movement authority in GMA state, received along in event data port m and if granted, moves to the CMA state through T1_MA_Check. If the newly received movement authority is not null, the train is allowed to move forward (in the MFR state) through T2_MA_Ok, otherwise transition T9_MA_NotOk is executed to the RETRY state, indicating that movement authority is not granted and is to be applied for again after a certain period.

When the train is moving forward, after every 200 milliseconds transition T3_Move_Check is executed to check the current position to request for movement authority extension. The train must apply for new movement authority as it reaches to a specific distance (SR) from the EoA, as depicted in Figure 1. Request for movement authority extension is applied to RBC through port r.

If current velocity of the train (received along `cv` in data port) is greater than service brake limit of the current segment or dynamic speed profile for v_2, transition `T4_SBI_Point` is performed and the `Controller` moves to the `SBI` state. If velocity of the train is less than service brake limit and the dynamic speed profile, transition `T5_Move_Ok` is performed and the train is allowed to move forward for the next 200 milliseconds.

Once the service brake is applied and the `Controller` is in the `SBI` state, on each dispatch through `T6_SBI_Check`, the current velocity of the train is checked against the service brake limit and the dynamic speed profile for v_2 to execute the appropriate transition (`T7_SBI_Out` or `T8_SBI_Ok`), and the newly computed acceleration or deceleration values are transmitted along out data port `ca`. If the service brake fails in the `SBI` state and the velocity of the train exceeds the emergency brake limit (v_1), or dynamic speed profile for v_1, transition `T13_EBI_Point` is preformed indicating application of the emergency brake.

If service brake is failed in the `CMF` state and current velocity of the train is greater than the emergency brake limit (v_1) or dynamic speed profile for v_1, emergency brake is applied and the `Controller` is moved to the `EBI` state by performing `T11_EBI_Point` transition. When the velocity of the train becomes zero, `Controller` is moved to the final state the `STOP` by performing transition `T12_Stop`.

In order to restart, driver has to reset certain parameters (not explained here) to enter to the `READY` state again and apply for new movement authority.

4.3 Continuous behavior modeling

The implementation classifier of Listing 5 specifies detailed continuous behavior of the `Train` component in the MA scenario. The `variables` section contains local variables in the scope of current the Hybrid annex subclause. Variable `s` represents position of the train and variable `v` represents velocity of the train. Variable `a` represents acceleration of the train while variable `t` represents the current time of the train. A data type is assigned to each variable, by a classifier reference to an appropriate AADL data component. The referenced external data component `Position`, `Velocity`, `Acceleration`, and `Time` are defined within the scope of a package `CTCS_Types` that has been imported using the AADL `with` clause.

Listing 5 Train component implementation with Hybrid annex

```
abstract implementation Train.impl
annex hybrid {**
 variables
   s : CTCS_Types::Position -- current position
   v : CTCS_Types::Velocity -- current velocity
   a : CTCS_Types::Acceleration --current acceleration
   t : CTCS_Types::Time      --train clock time

 behavior
   Train ::= ' DT 1 s = v ' & ' DT 1 v = a ' & ' DT 1 t = 1 '
              [[> ts!(s), tv!(v), ta?(a) ]]> Continue
   Continue ::= skip
   RunningTrain ::= s:=0 & v:= 0 & t:= 0 & REPEAT(Train)
**};
end Train.impl;
```

The `behavior` section of the Hybrid annex subclause uses process algebra notations to model the continuous behavior, and the cyber-physical interaction with other AADL components. The continuous behavior of the train is modeled by a repeating `Train` process as the `RunningTrain` process of the `behavior` section in Listing 5. Movement of the train is modeled as continuous evolution of the position `s` following the differential equation $\dot{s} = v$ (where $\dot{s}$ is the time-derivative of s) and the continuous evolution of its velocity `v` following the differential equation $\dot{v} = a$ (where $\dot{v}$ is the time-derivative of v and a is the acceleration). This continuous movement of the train is specified as `'DT 1 s = v' & 'DT 1 v = a'`. Here, `'DT 1 s = v'` depicts rate of change of `s` with respect to time and `'DT 1 v = a'` depicts rate of change of `v` with respect to time while `1` following the `'DT'` is the order of the derivative. Continuous evolution of train clock time `t` is specified as `'DT 1 t = 1'`. Variables `s`, `v`, and `t` are initialized with `0`. The ampersand (`&`) acts as a separator having no semantics.

The continuous behavior of the `Train` can be preempted by its communication with the `Controller` component. This cyber-physical interaction is modeled as `[[> ts!(s), tv!(v), ta?(a) ]]>` depicting the communication events performed along the data ports `ts`, `tv`, or `ta` specified in the type classifier of Listing 2. The `ts!(s)` and `tv!(v)` represent transmission of value of variables `s` and `v`, and are considered as output communication events. Event `ta?(a)` is an input communication event as it specifies receiving of the value of variable `a` along data port `ta`. Process `Continue` models the subsequent behavior, the successful termination of an iteration, described with a primitive process `skip`.

5 Discrete behavior verification

This section presents specification of behavior constraints B_1, B_2, B_3, and B_4, and formal verification of the `Contorller` component. Modeling of the behavior constrains as Assertions, generation of the proof obligation based on these Assertions, and discharging the proof obligations to produce formal proof as a list of theorems, is discussed below.

5.1 Constraints specification using BLESS assertions

In BLESS, behavior constraints are specified in the `assert` section either for later inclusion as terms in the `invariant` section, thereby making it more concise, or for expressing conditions on states, state actions, and transitions. An Assertion is a first-order logic formula enclosed between `<<` and `>>`. The `invariant` section may contain a single Assertion that always hold true for all aspects of the component's behavior.

Listing 6 depicts the Assertions, and invariant used to specify behavior constraints of the `Controller` component. Variables used, are declared in the `variables` section of the BLESS subclause and are detailed in Section 4.2. The first Assertion `<<SBL: :(s=CTCS_Properties::start) or (v < iSeg.v2)>>` is a disjunction of `(s=CTCS_Properties::start)`, and `(v < iSeg.v2)`. Data component `start` represents the starting position of the train and is declared in package `CTCS_Properties`. Term `(v < iSeg.v2)` specifies behavior constraint B_1 and `SBL` is the label of the Assertion.

Listing 6 Behavior constraints as BLESS Assertions

```
assert
   <<SBL: :(s=CTCS_Properties::start) or (v < iSeg.v2)>> -- for behavior constraint B1
   <<DSPV2: : (s=CTCS_Properties::start) or (((v**2) + (2*b*s)) <
              (iMA[nSeg.v2] + (2*b*iSeg.e))) >> -- for behavior constraint B2
   <<EBL: :(v < iSeg.v1)>> -- for behavior constraint B3
   <<DSPV1: :(((v**2) + (2*b*s)) < (iMA[nSeg.v1] + (2*b*iSeg.e))) >> -- for behavior
                                                                      -- constraint B4
   <<Acceleration: :=   -- for out data port ca
       (not SBL() or not DSPV2()) -> CTCS_Properties::SB_Rate,
       (not EBL() or not DSPV1()) -> -b,
       (SBL() and DSPV2()) -> x1 >>

invariant
 <<true>>
```

The second Assertion with label `DSPV2` specifies behavior constraint B_2. It is a disjunction of two terms: `(s=CTCS_Properties::start)`, and `(((v**2) + (2*b*s)) < (iMA[nSeg.v2] + (2*b*iSeg.e)))`. Third Assertion `<<EBL: :(v < iSeg.v1)>>` specifies behavior constraints B_3.

The Assertion with label `Acceleration` has a special structure and is used as port Assertion. It specifies the predicates (on the left side of the `->`) that must be true when a particular value (on the right side of the `->`) is transmitted along the port. The predicate `SBL() and DSPV2()` must hold when newly computed acceleration `x1` is sent while the predicate `not SBL() or not DSPV2()` must hold when the service brake is applied and `SB_Rate` is transmitted along the port. Incase of emergency brake intervention, the maximum deceleration `-b` is transmitted, predicate `not EBL() or not DSPV1()` must hold. Assertions `SBL()`, `DSPV2()`, `EBL()`, and `DSPV1()` are defined in the same `assert` section. Variable `x1` is declared in the `variables` section while data components `SB_Rate` is declared in the scope of package `CTCS_Properties` that has been imported using the AADL `with` clause.

The `invariant` section in Listing 6 contains a predicate `<<true>>` stating that behavior of the component is always true if all the predicates on the states, transitions, and actions always hold true.

5.2 Adding BLESS assertions

For extensive behavior analysis, BLESS supports *Port*, *State*, and *Action Step* Assertions. Assertions are also included as terms in the `invariant` section to define a predicate that must hold throughout the model's execution lifetime as explained in the previous subsection. Below we explain each of these Assertions in relation with the `Controller` component.

Listing 7 Controller component type with BLESS Assertions

```
thread Controller
...
     r : out event port
          { BLESS::Assertion=> "<<(s=CTCS_Properties::start) or (s>=(eoa-SR))>>"; };
     m : in event data port CTCS_Types::MovementAuthority
          { BLESS::Assertion=> "<<:=IMA>>"; };
    cs : in data port CTCS_Types::Position
          { BLESS::Assertion=> "<<:=POSITION>>"; };
    cv : in data port CTCS_Types::Velocity
          { BLESS::Assertion=> "<<:=VELOCITY>>"; };
    ca : out data port  CTCS_Types::Acceleration
          { BLESS::Assertion=> "<<:=Acceleration()>>"; };
...
 end Controller;
```

5.2.1 *Port assertions*

AADL supports modeling of the *assume-guarantee* relationship between components with an effective mechanism for component contract specification based on interface and interactions. A port Assertion defines a predicate that must be true when an event or data is sent or received on the port. Port Assertions are specified using `BLESS::Assertion=>` as the prefix, followed by the predicate enclosed in `"<<` and `>>"`.

Listing 7 depicts the `Controller` type classifier annotated with the BLESS Assertions. The Assertions `<<:=POSITION>>`, and `<<:=VELOCITY>>` attached to in data ports `cs`, and `cv`, respectively, represent values transmitted by the `Train` component. As communication infrastructure between the `Train` and the `Controller` is abstracted with simple AADL connections (see Figure 3) so these two Assertions represent the assumption that correct values of velocity and position are received on appropriate ports *i.e.*, `cs` and `cv`.

Assertion `<<(s=CTCS_Properties::start) or (s>=(eoa-SR))>>`, attached to port `r` states that movement authority request can either be sent at the start or when the train is at least `SR` meters far from the end of authority (EoA).

Assertion `<<:=IMA>>` attached to the port `m` represents that movement authority received on port `m` is equal to the `IMA`. As the internal structure of the *RBC* component is not explored as part of this study, we assume that the `IMA` is either `null` if movement authority is not extended, or has a proper constant value (represented as `IMA`). Assertion `<<:=Acceleration()>>` attached to the port `ca` has a special structure defined in the `assert` section, as explained in the previous subsection.

5.2.2 *State assertions*

The predicate defined as state Assertion must hold true while the `Controller` is in that state. Listing 8 shows the Assertions attached to each `Controller` state. Assertion `<<(i=0) and (s=CTCS_Properties::start)>>`

Listing 8 Controller component states with BLESS Assertions

```
thread implementation Controller.impl
 annex BLESS{**
 ... --variables section is not shown
  states
    READY : initial state <<(i=0) and (s=CTCS_Properties::start)>> ;
    GMA   : complete state  <<(i=0) and (s=CTCS_Properties::start)>> ;
    CMA   : state <<(i=0) and (s=CTCS_Properties::start)>>;
    RETRY : state <<(i=0) and (s=CTCS_Properties::start)>>;
    MFR   : complete state << SBL() and DSPV2()>>;
    CMF   : state <<SBL() and DSPV2()>>;
    SBI   : complete state << not SBL() or not DSPV2()>>;
    CSB   : state <<not SBL() or not DSPV2()>>;
    EBI   : complete state <<not EBL() or not DSPV1()>>;
    STOP  : final state <<(not EBL() or not DSPV1()) and v=0>>;
  ... --transitions section is not shown
 **};
end Controller.impl;
```

attached to the `READY`, `GMA`, `CMA`, and `RETRY` states specifies that value of the index variable `i` must be `0`, and the train is at the start position, whenever the `Controller` is in any of these three states. If the `Controller` is either in the `MFR` state, or in the `CMF` state, the Assertion `<<SBL() and DSPV2()>>` must hold true.

The definition of the Assertions `SBL()`, and `DSPV2()` in the `assert` section specify behavior constraints for service brake limit and dynamic speed profiles for v_2 (B_1, and B_2). So, the conjunction of these two Assertions attached to any state (when the train is moving forward) means that velocity of the train must be less than the service brake limit and dynamic speed profile for v_2, of a particular segment. Once the service brake is applied and the train is either in the `SBI` state, or in the `CSB` state, Assertion `<<not SBL() or not DSPV2()>>` must hold, which means that the train keeps decelerating until its velocity is less than the service brake limit, and the dynamic speed profile for v_2, and can accelerate afterwards.

The predicate `not EBL() and not DSPV1()` attached to the states `EBI`, and `STOP` specify behavior constraints B_4, representing emergency brake intervention. If service brake is failed, and the velocity of the train is greater than or equal to the v_1, or dynamic speed profile for v_1, then the `Controller` enters to the `EBI` state indicating the application of the emergency brake. The `Controller` ultimately enters the `STOP` state when the train is fully stopped and its velocity is zero. This is specified in Assertion `not EBL() and not DSPV1() and v=0`, attached to the `STOP` state.

5.2.3 *Action step assertions*

An action step Assertion defines the predicate that must hold true, at the point where the Assertion is inserted, during execution of transition's action steps. Two sample transitions annotated with action step Assertions are shown in Listing 9, covering all the important concepts required to be explained, while the rest of the transitions (as part of the complete AADL model) are available at [14].

Assertion `<<(i=0) and (s=CTCS_Properties::start)>>` inserted before and after action step `r!` states the predicate that must hold true before and after an output communication event on port `r` is preformed. Entering a complete state which suspends thread (the `Controller` component) until next dispatch, a simultaneous assignment is used before the Assertion in transitions where the destination state is a complete state. Temporal operator `'` used in the simultaneous assignment `(i',s':=i,s)` and the Assertion `<<(i'=0) and (s'=CTCS_Properties::start)>>` means value of the expression in the next clock cycle (or thread period). The `;` operator after `r!` depicts the sequential composition of the next action step, which is the simultaneous assignment in this case.

Transition `T3_Move_Check` specifies the last behavior constraint B_5. When the train is moving forward, a request for movement authority extension can only be applied if the train is at least *SR* meters far from the EoA. As the source state `MFR` is a complete state, on every dispatch, the current position, and velocity of the train are received on in data ports specified by communication events `cs?(s)` and `cv?(v)`.

Listing 9 Controller component transitions with BLESS Assertions

```
thread implementation Controller.impl
 annex BLESS{**
  ... --variables and states sections are not shown
  transitions
    T0_go: READY -[]-> GMA
            {
            <<(i=0) and (s=CTCS_Properties::start)>>
            r!;
            <<(i=0) and (s=CTCS_Properties::start)>>
            (i',s':=i,s)
            <<(i'=0) and (s'=CTCS_Properties::start)>>
            };
   ...
   T3_Move_Check: MFR -[on dispatch]-> CMF
            {
            { <<SBL() and DSPV2()>> cs?(s) <<s=POSITION>> & cv?(v)
            <<SBL() and DSPV2() and v=VELOCITY>> };
            <<SBL() and DSPV2()>>
            if ( s>=(eoa-SR) )~>
            <<SBL() and DSPV2() and (s>=(e-SR))>>
                r!
            <<SBL() and DSPV2()>>
            [] (not (s>=(eoa-SR)) )~>
            <<SBL() and DSPV2()>>
                skip
            <<SBL() and DSPV2()>>
            fi  <<SBL() and DSPV2()>>
        };
  ...   -- Transitions T1_MA_Check, T2_MA_Ok, T4_SBI_Point, T5_Move_Ok,
        -- T6_SBI_Check, T7_SBI_Out, T8_SBI_Ok, T9_MA_NotOk,
        -- T10_MA_Retry, T11_EBI_Point, T12_Stop,
        -- and T13_EBI_Point are not shown
  ...
 **};
end Controller.impl;
```

Assertion `<<SBL() and DSPV2()>>` before and after the communication events is inserted to ensure that the previous and the new velocity, and position of the train are less than service brake limit and dynamic speed profile for v_2. Communication events can execute concurrently as specified by the `&` operator. Assertions `<<s=POSITION>>` after `cs?(s)`, and `<<s=VELOCITY>>` after `cv?(v)` specify the assumption discussed in Section 5.2.1.

Transition `T3_Move_Check` contains an *alternative formula* and the Assertion `<<SBL() and DSPV2()>>`, used before the keyword `if` and after the keyword `fi`, states what must be true before and after the execution of this formula. Condition expression for each alternative is parenthesized with `~>` as postfix while `[]` acts as *else if* in a typical *if* - *else if* statement. Assertion inserted before and after each action step in an alternate specifies what must be true before and after an action is preformed in a particular alternate. The first alternative `s>=(eoa-SR)` states, that if the current position of the train is at least *SR* meters from the EoA, then a request for movement authority extension is sent along port `r`. The second alternative is true if the current position of the train `s` is less than `eoa-SR`. Action step `skip` indicates the successful termination, without doing anything.

The transitions not discussed here have similar structures, as discussed above for two sample transitions, for action step Assertion inclusion. Complete AADL model of the system with the BLESS, and Hybrid annexes specifications is available at [14].

5.3 Proof obligations

BLESS proof obligations are of the form `<<P>> S <<Q>>`, traditionally known as *Hoare Triple*, where `P` and `Q` are the precondition and postcondition, respectively, and `S` is an action. It states that if `P` is true before `S` is executed then executing `S` establishes `Q`. Both `P` and `Q` are the BLESS Assertions, and the `S` can be a simple action or may further be composed of `<<P>> S <<Q>>`.

Formal verification of the behavior of the `Controller`, modeled as an AADL thread component requires to solve all the proof obligations generated based on the Assertions explained in the previous subsection. Below we present an examples of different kinds of proof obligation generated for states and transitions of the `Controller` component. Complete set of all proof obligations generated for the `Controller` component, is available at [14].

Complete states imply invariant: Due to the fact that entering a complete state suspends execution of the thread until the next dispatch, each complete state's Assertion must imply the invariant. The proof obligation generated by the BLESS proof tool, for the complete state `MFR` is:

```
[serial 1004]: MA::Controller.impl
P [121] <<SBL() and DSPV2()>>
S [104]->
Q [104] <<true>>
  What for: <<M(MFR)>> -> <<I>>
  from invariant I when complete state MFR has Assertion <<M(MFR)>> in its definition.
```

Here, values between "`[`" and "`]`" are line numbers in the AADL model and "`what for:`" part explains the reason of the proof obligation. Proof obligations for other complete states `GMA`, `SBI`, and `EBI` are generated in the same way.

Execute states have enabled outgoing transition: For every execute state, there must always be at least one enabled outgoing transition with a true execute condition. In case of more than one enabled transitions, this choice is nondeterministic. To ensure this, each execute state's Assertion must imply the disjunction of outgoing transition condition. Proof obligation for execute state `CSB` is:

```
[serial 1007]: MA::Controller.impl
P [124] <<not SBL() or not DSPV2()>>
S [124]->
Q [124] <<(((s = CTCS_Properties::start) or ((v < iSeg.v2)))
  and
  ((s = CTCS_Properties::start)
  or ((((v**2)+(2*b*s)) < (iMA[nSeg.v2]+(2*b*iSeg.e))))))
  or (not (((s = CTCS_Properties::start) or ((v < iSeg.v2)))
  and
  ((s = CTCS_Properties::start)
  or ((((v**2)+(2*b*s)) < (iMA[nSeg.v2]+(2*b*iSeg.e)))))))>>
  What for: Serban's Theorem:  disjunction of execute conditions leaving
  execution state CSB, <<M(CSB)>> -> <<e1 or e2 or . . . en>>
```

Proof obligations for other execute states `CMA`, `RETRY`, and `CMF` are generated in the same way.

Port communication: For every out event port communication action, its precondition must imply the port's Assertion. Below is the proof obligation generated for out event port `r`.

```
[serial 1053]: MA::Controller.impl
P [130] <<CTCS_Properties::start = s and i = 0>>
S [131]->
Q [1] <<(s = CTCS_Properties::start) or (s >= (eoa-SR))>>
  What for: applied port output <<pre>> -> <<M(r)>> [serial 1036]>
```

Table 1 States proof obligations and theorems

State	Proof Obligation	Proof Theorems
GMA	1003	1
MFR	1004	2
SBI	1005	3
EBI	1006	4
CSB	1007	5—6
CMF	1008	7—9
RETRY	1009	10
CMA	1010	11—12

The precondition `<<CTCS_Properties::start = s and i = 0>>` is the Assertion inserted right before the communication event `r!` (in transition `TO_go`) while the Assertion on port `r` specified with its definition as *Assertion property*, `<<(s = CTCS_Properties::start) or (s >= (eoa-SR))>>`, becomes the postcondition. Proof obligations for all other ports are generated in the same way.

5.4 Discharging proof obligations

Proof obligations generated from Assertions are transformed into simpler ones by applying inference rule provided by the BLESS proof tool. The BLESS proof tool is implemented as a plug-in to Open-Source AADL Tool Environment version 2 (OSATE 2), the development environment for AADL modeling [15].

Listing 10 Sample proof theorems generated by the BLESS proof tool

```
Theorem (2)                                  [serial 1004]
P [120] <<SBL() and DSPV2()>>
S [102] ->
Q [102] <<true>>
by True Conclusion Schema (tc): P->true
...
Theorem (6)                                  [serial 1007]
P [124] <<not SBL() or not DSPV2()>>
S [124] ->
Q [124] <<(((s = CTCS_Properties::start) or ((v < iSeg.v2)))
  and
  ((s = CTCS_Properties::start)
  or ((((v**2)+(2*b*s)) < (iMA[nSeg.v2]+(2*b*iSeg.e))))))
or (not (((s = CTCS_Properties::start) or ((v < iSeg.v2)))
  and
  ((s = CTCS_Properties::start)
  or ((((v**2)+(2*b*s)) < (iMA[nSeg.v2]+(2*b*iSeg.e)))))))>>
by Law of Excluded Middle: P or not P is tautology
and theorem 5:
Theorem (5) [serial 1025] used for:
  Law of Excluded Middle: P or not P is tautology [serial 1007]
...
Theorem (307)                                  [serial 1002]
P [115] << >>
S [102] ->
Q [115] <<Controller.impl proof obligations>>
by Initial Thread Obligations
and theorems 1 2 3 4 6 9 10 12 31 44 69 114 141 166 185 215 234 242 251 280 282 306:
```

The proof tool produces the formal proof of the behavior correctness as a list of theorems when all the proof obligations are solved. Each theorem is either axiomatic, or derived from earlier theorems by applying inference rules. Like other interactive theorem provers, the BLESS proof tool requires human intervention to select proof tactics. When proof obligations cannot be further simplified, or are obviously false, errors in either the program or its proof outline are indicated. Line numbers on precondition, postcondition, and action indicate where the error most likely resides. BLESS also provides proof derivation tree to trace unsolved proof obligations back to the initial proof obligation to help the programmer understand where the error lies. Listing 10 contains some of the sample theorems.

Table 2 Transitions proof obligations and theorems

Transitions	Proof Obligation	Proof Theorems
T0_go	1011	13—31
T1_MA_Check	1012	32—44
T2_MA_Ok	1013	45—69
T3_Move_Check	1014	70—114
T4_SBI_Point	1015	115—141
T5_Move_Ok	1016	142—166
T6_SBI_Check	1017	167—185
T7_SBI_Out	1018	186—215
T8_SBI_Ok	1019	216—234
T9_MA_NotOk	1020	235—242
T10_MA_Retry	1021	243—251
T11_EBI_Point	1022	252—280
T12_Stop	1023	281—282
T13_EBI_Point	1024	283—306

Sample Theorem (2) is used to discharge the example proof obligation discussed for *Complete states imply invariant*, and Theorem (6) with Theorem (5) is used to discharge the proof obligation discussed for *Execute states have enabled outgoing transitions*.

In all, complete proof of the `Controller` component requires 307 theorems. Table 1 presents proof obligations along with proof theorems generated for each state. For state `GMA` proof obligation 1003 is generated and proof theorem 1 is used to discharge this proof obligation while proof obligation 1008 is generated for `CMF` state and proof theorems 7—9 are used to discharge this obligation. Proof obligations and theorems generated for other states are described in the same way.

Table 2 contains proof obligations and proof theorems generated for each transition. For `T0_go`, proof obligation 1011 is generated and proof theorems 13—31 are used to discharge this obligation. For `T7_SBI_out`, proof obligation 1017 is generated and theorems 186–215 are used to discharge this obligation. Proof obligations and theorems generated for other transitions are described in the same way.

Based on the behavior correctness proof generated by the BLESS proof tool, we can claim that the properties required to be proved for the system level behavior verification of the `sysMA` component are proved for the `Controller` component.

Complete set of all proof theorems in relation with proof obligations is available at [14].

6 System level behavior verification

In the previous section we have proved behavior correctness of the `Controller` component using the BLESS proof tool. This section is focused on behavior verification of the whole `sysMA` as depicted in Figure 3.

In [8], we have illustrated the application of the HHL Prover for verification of hybrid systems modeled using AADL, and the Hybrid annex through a benchmark hybrid system—water level control system. Here, we apply the same approach but with the detailed discrete behavior modeling using the BLESS annex.

6.1 Hybrid hoare logic

AADL models annotated with the Hybrid annex specification, for the continuous behavior modeling, formally verified using Hybrid Hoare Logic (HHL) Prover. In [2], classical Hoare Logic is extended to Hybrid Hoare Logic for hybrid systems, by adding history formulas to describe continuous properties that hold true throughout the execution of a continuous processes.

History formulas, in HHL, are specified using the Duration Calculus (DC), which is a real arithmetic extension of the Interval Temporal Logic (ITL) for describing and reasoning about real-time systems. The mainly used assertion $\lceil S \rceil$, where S is a state formula, means that S holds everywhere inside the considered interval. In the HHL, specification for a sequential process P is of the form $\{Pre\}\, P\, \{Post; HF\}$, where $Pre, Post$ represent precondition, and postcondition, respectively and are expressed by first-order logic to specify properties of variables held at starting and termination of the execution of P. HF is a history formula, expressed using the DC, to record the execution history of P, including its real-time, and continuous properties. The specification for a parallel process is then defined by assigning to each sequential component the respective precondition, postcondition, and history formula, that is

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

.

Each Hybrid annex construct, based on HCSP, is axiomatized by a set of axioms, and inference rules.

6.2 Corresponding HCSP processes

For verification, the HHL Prover requires the behavior to be modeled as interacting HCSP processes. Behavior of the `sysMA` can be modeled as a parallel composition of three processes; *Controller*, *Train*, and *RBC* representing `Controller`, `Train`,a `RBC` AADL components respectively.

$$sysMA \triangleq Controller \parallel Train \parallel RBC$$

We assume the correct functionality of the RBC in the MA scenario and only focus on its communication behavior, no further details are described for the corresponding *RBC* process.

As precise behavior of the `Controller` component has already been modeled (as explained in Section 4.2) and verified using the BLESS annex (as explained in Section 5), here, the corresponding *Controller* process only represents the abstracted functionality of the component. Currently this refinement is done manually with expert domain knowledge. Automatic refinement of complex BLESS behaviors into HCSP processes to incorporate corresponding the Hybrid annex specifications, is a topic of future research. Below is the definition of the *Controller* process:

$$\begin{aligned}
Controller &\triangleq (Ch_s?cs; Ch_v?cv; (cv \geqslant seg.v2 \vee not\ DSPV2) \rightarrow\ Ch_a!SB\_Rate;\\
&\quad (cv \geqslant seg.v1 \vee not\ DSPV1) \rightarrow\ Ch_a!-b;\\
&\quad wait\ Period; Temp := Temp + Period\ )^*\\
DSPV2 &\triangleq seg.v_2^2 + 2bs \leqslant next(seg).v_2^2 + 2bseg.e\\
DSPV1 &\triangleq seg.v_1^2 + 2bs \leqslant next(seg).v_1^2 + 2bseg.e
\end{aligned}$$

Here, $Ch_s?cs$, and $Ch_v?cv$ are the communication events along channels Ch_s and Ch_v representing the connections between the `Controller`, and `Train` component. Variables cs and cv representing ports of the `Controller` component. *DSPV2* and *DSPV1* represent Assertions `DSPV2()` and `DSPV1()`, respectively while $cv \geqslant seg.v2$ and $cv \geqslant seg.v1$ represent Assertions `SBL()` and `EBL()`, respectively. *SB_Rate*, and $-b$ are the deceleration rates for service and emergency brake intervention. The value of the *Period* is 200 milliseconds.

The repeated behavior of the *Controller* process is as follows. In every dispatch, it receives current position and velocity of the train along channels Ch_s and Ch_v and transmits either *SB_Rate* along channel Ch_a if the $cv \geqslant seg.v2 \vee not\ DSPV2$ is violated, or $-b$ along channel Ch_a if $cv \geqslant seg.v1 \vee not\ DSPV1$ is violated. It then waits for the next dispatch as specified by *wait Period*. Variable *Temp* represents current time period.

The continuous behavior of the `Train` component modeled using Hybrid annex, as explained in Section 4.3, is presented by the following HCSP process *Train*.

$$Train \triangleq \langle \dot{ts} = cv, \dot{tv} = ta, \dot{t} = 1\rangle \trianglerighteq (Ch_s!ts; Ch_v!tv) \rightarrow Ch_a?a$$

As Hybrid annex is inspired by HCSP so each notation of Hybrid annex automatically corresponds to a respective HCSP notation. This refinement is the deliberate similarity of the Hybrid annex grammar to HCSP, allowing a simple and direct refinement. Here, $\dot{ts}$, $\dot{tv}$, and $\dot{t}$ represent the continuous evolution of position, velocity and time of the train. The continuous evolution is preempted as soon as a communication interrupt occurs along Ch_s, or Ch_v. Afterwards, process *Train* is ready to receive newly computed acceleration value from the *Controller* process along channel Ch_a.

6.3 Properties to be proved

Behavior verification of the `sysMA`, represented by HCSP process *sysMA*, can be realized by verifying the following system level properties (although the behavior verification of the `Controller` assures these properties, but for the `Controller` component only). Specification of all the properties follows the HHL syntax.

6.3.1 *End of authority (EoA)*

As explained in Section 3.2, for every movement authority the *RBC* also transmits the End of Authority (EoA). The EoA is equal to the end of the last segment. Below is the property to be verified for this behavior.

$$\{ts = 0 \wedge tv = 0 \wedge t = 0 \wedge next(seg).v_1 = 0 \wedge next(seg).v_2 = 0 \wedge ea = seg.e, True\}sysMA$$
$$\{True, True; \lceil ts \leqslant EoA \wedge ts = EoA \rightarrow tv = 0\rceil, True\}$$

where, $\{ts = 0 \wedge tv = 0 \wedge t = 0 \wedge next(seg).v_1 = 0 \wedge next(seg).v_2 = 0 \wedge ea = Seg.e, True\}$ is the precondition specifying the initial position of the train, $\lceil ts \leqslant EoA \wedge ts = EoA \rightarrow tv = 0\rceil$ is the property to be proved, and *True* at the end of the specification is a history formula. The property states that position of the train must always be less the EoA and if it is equal to EoA the train must stop. History formula specified as *True*, means that the property must always hold and $tv = 0$ represents a stopped train.

6.3.2 *Service brake intervention (SBI)*

Below is the property of the *sysMA* to be verified to ensure the SBI.

$$\{ts = 0 \wedge tv = 0 \wedge t = 0, True\}sysMA\{True, True; \lceil (tv \geqslant seg.v2 \vee not\ DSPV2) \wedge$$
$$t \geqslant (Temp + Period) \rightarrow ta = SB\_Rate\rceil, True\}$$

Predicates of the precondition $\{ts = 0 \wedge tv = 0 \wedge t = 0, True\}$ are the same as for the EoA property. The property to be proved is $\lceil (tv \geqslant seg.v2 \vee not\ DSPV2) \wedge t \geqslant (Temp + Period) \rightarrow ta = SB_{Rate}\rceil$ stating that whenever the velocity of the train is greater than or equal to the SBI limit of the current segment, or the *DSPV2* does not hold, the acceleration of the train is set to *SB_ Rate*.

6.3.3 *Emergency Brake Intervention (EBI)*

The last property to be verified is related to EBI and is specified as under:

$$\{ts = 0 \wedge tv = 0 \wedge t = 0, True\}sysMA\{True, True; \lceil (tv \geqslant seg.v1 \vee not\ DSPV1) \wedge$$
$$t \geqslant (Temp + Period) \rightarrow ta = -b\rceil, True\}$$

The precondition is the same as for the SBI property. The property to be proved is $\lceil (tv \geqslant seg.v1 \vee not\ DSPV1) \wedge t \geqslant (Temp + Period) \rightarrow ta = -b\rceil$. It states whenever the velocity of the train is greater than or equal to the EBI limit of the current segment, or the *DSPV1* does not hold, the acceleration of the train is set to $-b$.

6.4 Verification using HHL prover

An interactive theorem prover for HHL (the HHL Prover) has been implemented in proof assistant Isabelle/HOL. The tool can be downloaded from `lcs.ios.ac.cn/~znj/HHLProver`. Given the annotated HCSP processes with the HHL specifications, verification conditions are generated based on the HHL proof system. Thus, properties to be proved are reduced to a set of (logically equivalent) verification conditions, each of which is either a first-order logic formula, or a DC formula. The logical formulas are then proved by applying the inference rules defined using the HHL. Further details on axioms and inference rules of the HHL Prover are presented in [2].

Following code snippet (written in Isabelle/HOL) shows the HCSP process annotated with HHL specifications and the lemma to be proved for system level behavior verification of the `sysMA` process corresponding to all three properties presented above: EoA, SBI, and EBI.

```
theory sysMA_Proof
    imports HHL
begin
  definition ts :: “exp” where ″ts == RVar “ts” ″
  definition tv :: “exp” where ″tv == RVar “tv” ″
  definition ta :: “exp” where ″ta == RVar “ta” ″
  definition t :: “exp” where ″t == RVar “t” ″
...
  lemma Goal :
  ″{v21[=](Real 0)[&]v22[=](Real 0)[&]eoa[=]e, ts[=](Real 0)[&]tv[=](Real 0)[&]t[=](Real 0)}
  sysMA
  {WTrue, WTrue; WTrue, high
  (((cv[⩾]v12[|][∼]DSPV2)[&](t[>](temp[+](Real period)))[→]ta[<](Real 0))
  [&]((cv[⩾]v11[|][∼]DSPV1)[&](t[>](temp[+](Real period)))[→]ta[=](Real − b))
  [&](ts[⩽]eoa[&](ts[=]eoa[→]tv[=](Real 0))))}″
...
  apply (rule conjR)
  apply (rule Trans)    apply (simp add : v21_def v22_def eoa_def e_def ts_def
                                tv_def t_def DSPV_def equal_less_def)
  apply (rule conjR)
  apply (simp add : DSPV_def)
...
end
```

Statements starting with *definition* shows the sample variable definition while *Goal* represents the lemma to be proved. Here, variables v_{11}, v_{12}, v_{21}, and v_{22} are declared for velocities v_1 and v_2 of the current and the next segment in a particular movement authority. Variables *eoa*, and *e* depict the EoA and the end of the last segment respectively. Variables *cv*, and *ca* represent velocity, and acceleration for the `Controller` component while variables *tv*, *ta*, and *ts* represent the velocity, acceleration and position of the `Train` component. Logical and relational operators are specified between quare brackets [ ]. *WTrue* is used as precondition and defines what should be true at the start. *Real* represents the data type real, and -*b* is the maximum deceleration in case of emergency brake intervention. *DSPV*1, and *DSPV*2 are the behavior constraints while the history formula *high* states the properties must hold throughout the execution of the process.

The above specified lemma *Goal* is proved using axioms and inference rules with the HHL Prover, thus depicting the system level behavior correctness proof of the `sysMA`. Statements starting with *apply* are some of the applied sample rules. Complete HCSP model annotated with HHL specifications and required definitions, and the complete proof of the model is available at [14].

7 Related work

Modeling and verification of hybrid systems has been explored a great deal. Hybrid automata [16, 20] with temporal constraints, specified using temporal logic [17, 18], is the most popular modeling formalism. However, analogous to state machines, hybrid automata provides little support for structured and compositional specification. As an alternative approach, Platzer proposed hybrid programs, and related differential dynamic logic for compositional modeling, and deductive verification of hybrid systems [21]. HCSP [10, 19], was proposed for process algebra based specification and verification of hybrid systems. It has been used in this paper to formalize the Hybrid annex specifications and to facilitate system level verification, because of its exclusive support for modeling of parallelism, and extensive communication among components, that occur ubiquitously in AADL models.

Train control systems have been investigated for reliability prediction, test case generation, and formal verification but in this section we summarize the literature focused on behavior modeling and verification, that is most closely related to the work presented in this paper. Verification of the CTCS-3 under a combined scenario composed of the MA, Level Transition, and Mode Transition scenarios is presented in [6]. Behavior of each component involved in a particular scenario is modeled as an HCSP process which are then combined by parallel composition to form the model of the scenario. The model of the combined scenario is specified as a parallel composition of all three basic scenarios, and verified using the HHL Prover [2]. Modeling of the CTCS-3 using Simulink/Stateflow is presented as a case study in [1] to explain the formal verification of Simulink diagrams. An extension of the work is also presented in [27] to avoid the inherent incompleteness of simulation by translating Simulink/Stateflow diagram to HCSP processes for formal analysis and verification. In [23], movement authority of the European Train Control System (ETCS) is discussed as a *hybrid program* to identify the relevant safety constraints on both the continuous and discrete parameters. Different safety and liveness properties of the MA scenario are then verified using the KeYmaera tool based on differential dynamic logic, to ensure the controllability and reactivity of the train.

A topology-based method for modeling of train control systems is proposed in [24] to describe the physical characteristics of the railway network to interpret and verify the train control logic. In this approach, a topological space is constructed for movement authority calculation based on fundamental principles, and computational model of the control system. Topological space is simulated for performance analysis. In [26], a formal method for component integration of communication-based train control systems for safety analysis, is proposed. They performed a case study on zone calculator using the SCADE suite with built-in Design Verifier. MARTE statecharts augmented with dense clock information for the continuous behavior modeling, called the hybrid MARTE statecharts, are introduced by Liu, et al., in [22]. They used hybrid automata semantics to verify hybrid systems modeled using hybrid MARTE statecharts. To address the issue of state space explosion, a model checking approach based on *time-bound short-run behavior* has been proposed in [25].

The above mentioned studies are centered on system realization while our work in this paper is focused on system design. We have captured the design concerns with the architectural, and behavioral modeling notations. The architecture is specified using core AADL constructs and defines the software and the hardware components. The behavior is specified using the BLESS and Hybrid annexes, that characterize how the system processes, and transmits the data to its environment. For example, to compute new acceleration based on the information received from the RBC, and the train, and the send it to the train. Our model is more close to system implementation, and the ultimate code production.

Zhang, at el., has used AADL to model the CTCS-3 based on the communication among visualized subsystems [5]. Movement of the train is modeled using Modelica which is then transformed into AADL property sets. Their approach seems too general to be used for detailed behavior (continuous and discrete) modeling and nothing has been provided to model the communication between the train and the control system. Compared to [5], behavior modeling and verification of the CTCS-3 presented in this paper, is more expressive in its ability to specify the primitives of hybrid system models, *e.g.*, variables with data types, constants with measuring units and detailed behavior modeling of the control system and the train. It also provides extensive support for the cyber-physical interaction modeling with timed, and communication interrupts, an essential element of the CTCS-3 modeling that is not provided for, to such an extent, by the related efforts. Exclusive support for behavior constraints specification and the definition of component invariants using Assertions is a novel feature of our work.

8 Conclusion and future work

In System Requirements Specification document, behavior of the Chinese Train Control System Level 3 (CTCS-3) is specified as a set of operation scenarios that cooperate with each other to achieve desired functionality of the train. Movement Authority scenario, one of the basic and most crucial to prohibit trains from colliding with each other, is investigated and modeled using AADL. The structure of the system is modeled using the core AADL constructs. The discrete behavior of the control system is modeled and verified using the BLESS annex which produced the formal proof as a list of 307 theorems. The continuous behavior of the train and the cyber-physical interaction (communication between the train and the controller) is modeled using the Hybrid annex and the system level behavior is verified using an interactive theorem prover—the HHL Prover. Behavior constraints are specified using first-order logic formulas as Assertions.

To further evaluate the performance and scalability of our modeling and verification approach, we aim to model the CTCS-3 under combined scenarios by integrating several operation scenarios and verify the compositional behavior of the complete on-board system. Adding fault behavior, and fully analyzing the on-board system of the CTCS-3 for safety perspective, is also an important planned work.

Acknowledgements

The first and sixth authors are supported by National Basic Research Program of China (973 Program) under grant No. 2014CB340701, by NSFC under grants 91118007 and 91418204, and by the CAS/SAFEA International Partnership Program for Creative Research Teams. The second author is supported by the National Infrastructure Software Plan under Grant No.2012ZX01041-002-003. The fourth and fifth authors are supported by the National Basic Research Program of China (973 Program) under grant No. 2014CB340703, and NSFC under grant No.61304185. We would like to thank Mr. Liang Zou for assisting in system level correctness proof using the HHL Prover. We also thank the anonymous reviewers for their constructive comments on previous version of this paper.

References

1 Zou L, Zhan N, Wang S, Franzle M, Qin S. Verifying simulink diagrams via a hybrid hoare logic prover. In: Proceedings of the 11th ACM International Conference on Embedded Software. Montreal, 2013. 1–13

2 Liu J, Lv J, Quan Z, Zhan N, Zhao N, Zhou C, Zou L. A calculus for hybrid CSP. In: Proceedings of the 8th Asian Symposium on Programming Languages and Systems. Shanghai, 2010. 1–15

3 Larson R B, Chalin P, Hatcliff J. BLESS: formal specification and verification of behaviors for embedded systems with software. In: Proceedings of the 5th International Symposium, NASA Formal Methods. California, 2013. 276–290

4 Zhan N, Wang S, Zhao H. Formal modelling, analysis and verification of hybrid systems. In: Proceedings of the 10th International Colloquium on Theoretical Aspects of Computing. Shanghai, 2013. 276–290

5 Zhang L, Xu B. Specification of communication based train control system using AADL. In: Proceedings of the 4th International Conference on Mobile, Ubiquitous, and Intelligent Computing . Gwangju, 2013. 63–68

6 Zou L, Lv J, Wang S, Zhan N, Tang T, Yuan L, Liu Yu. Verifying chinese train control system under a combined scenario by theorem proving. In: Proceedings of the 5th International Conference on Verified Software: Theories, Tools and Experiments. California, 2013. 262–280

7 Ahmad E, Larson R B, Barrett C S, Zhan N, Dong Y. Hybrid annex: an AADL extention for continuous behavior and cyber-physical interaction modeling. In: Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology, Portland. 2014. 29–38

8 Ahmad E, Dong Y, Wang S, Zhan N, Zou L. Adding formal meanings to AADL with hybrid annex. In: Formal Aspects of Component Software, LNCS 8997, Springer. 2014. 228-247

9 Zhang S. CTCS-3 technology specification. Beijing: China Railway Publishing House, 2000.

10 He J. From CSP to hybrid systems. A classical mind, W. Roscoe (Ed.). Hertfordshire: Prentice Hall International (UK) Ltd. 1994. 171–189

11 SAE International. Architecture Analysis & Design Language (AADL). SAE AS5506 Rev. B, 2012.

12 Feiler P, Jorgen H, Niz D, Wrage L. System architecture virtual integration: an industrial case study, Technical Report CMU/SEI-2009-TR-017, 2009.

13 BLESS language reference manual, http://www.santoslab.org/pub/bless/docs/BLESS_Language_Reference_Manual.pdf, 2014.

14 https://github.com/ehah/SCIS2014, 2014.

15 Osate 2 web site, https://wiki.sei.cmu.edu/aadl/index.php/Osate_2, 2014.

16 Alur R, Courcoubetis C, Henzinger A T, Ho P. Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In: Hybrid Systems. LNCS 736, Springer. 1993. 209–229

17 Moszkowski B C, Manna Z. Reasoning in interval temporal logic. Logic of Programs. Springer. 1983, 371–382

18 Manna Z, Pnueli A. Verifying hybrid systems. In: Hybrid Systems, LNCS 736, Springer. 1993. 4–35

19 Zhou C, Wang J, Ravn P A. A formal description of hybrid systems. In: Hybrid Systems III. LNCS 1066, Springer. 1996. 511–530

20 Henzinger A T. The Theory of Hybrid Automata. In: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science, Washington DC, IEEE Computer Society, 1996. 278–2092

21 Platzer A. Differential dynamic logic for hybrid systems. Journal of Automated Reasoning. Springer. 2008, 41(2): 143–189

22 Liu J, Liu Z, He J, Mallet F., Ding Z. Hybrid MARTE statecharts. Frontiers of Computer Science. SP Higher Education Press. 2013, 7(1): 95–108

23 Platzer A, David Q. European train control system: a sase study in formal verification. In: Proceedings of the 11th Internatinal Conference on Formal Engineering Methods, Rio de Janerio, LNCS 5885, 2009. 246–265

24 Wang H, Schmid F, Chen L, Roberts C, Xu T. A topology-based model for railway train control systems. IEEE Transactions on Intelligent Transportation Systems. IEEE Computer Society. 2013, 14(2): 819–827

25 Bu L, Wang Q, Chen, X, Wang L, Zhang T, Zhao J, Li X. Toward online hybrid systems model checking of cyber-physical systems' time-bounded short-run behavior. ACM SIGBED Review. ACM. 2011, 8(2): 7–10

26 Wang H, Tang T. On integrating component into safety critical system. In: Proceedings of the International Conference on Information Engineering and Computer Science, Wuhan, 2009. 1–4

27 Guo D, Lv J, Wang S, Tang T, Zhan N, Zhou D, Zou L. Formal analysis and verification of chinese train control system. In press. Science China Information Sciences, 2014, 44: 1-? (in Chinese)