STTT manuscript No.
(will be inserted by the editor)

Formal verification and simulation for platform screen doors and collision avoidance in subway control systems

Huixing Fang · Jianqi Shi · Huibiao Zhu · Jian Guo · Kim Guldstrand Larsen · Alexandre David

Received: date / Revised version: date

Abstract For hybrid systems, hybrid automata based tools are capable of verification, while Matlab Simulink/Stateflow is proficient in simulation. We propose a co-verification procedure, in which the verification tool SpaceEx/PHAVer and simulation tool Matlab are integrated to analyze and verify hybrid systems. For the application of this procedure, a Platform Screen Doors System (abbreviated as PSDS, a subsystem of the subway control system), is modeled with hybrid automata and Simulink/Stateflow charts, respectively. The models of PSDS are simulated by Matlab and verified by SpaceEx/PHAVer. The simulation and verification results indicate that the sandwiched situation can be avoided under time interval conditions. We improve the model with four trains and four stations on a subway line, and analyze the urgent control scenario for the safety distance requirement.

In this paper, the Simulink/Stateflow model is a refinement of the SpaceEx/PHAVer model, which is closer to a final implementation. Moreover, the two models are complementary for some features (e.g.,visualization of simulation, correctness proving by verification), stressing different aspects of the overall system and permitting complementary analysis techniques, i.e. verification versus simulation. We conclude that this integration procedure is competent in verifying Subway Control Systems.

Keywords Hybrid Systems · Formal Verification and Simulation · SpaceEx/PHAVer · Matlab Simulink/Stateflow · Subway Control Systems · Feedback-Advancement Verification

H. Fang · J. Shi · H. Zhu · J. Guo (✉)
Shanghai Key Laboratory of Trustworthy Computing, Software Engineering Institute, East China Normal University, Shanghai, China
E-mail: jguo@sei.ecnu.edu.cn

K. G. Larsen, A. David
Department of Computer Science, Aalborg University, Aalborg, Denmark

1 Introduction

In this paper, we focus on the application of formal methods and simulation techniques on subway control systems. We present the case study for the platform screen doors system and analyze the collision avoidance scenario between subway trains. We model the system as a kind of hybrid systems. The concept of hybrid systems arises in embedded control with the interaction between continuous physical behavior and discrete digital controllers.

Hybrid automata have been proposed to model and verify hybrid systems [6, 30]. The verification of reachability problem on hybrid systems is important and challenging [35, 24, 8, 34, 22, 9]. For linear hybrid systems, several algorithms based on model-checking have been developed [6, 42, 7]. The tool HyTech [29] focuses on linear hybrid systems.

Moreover, PHAVer [18, 20] is another tool for the exact verification of hybrid systems with piecewise constant bounds on the derivatives. PHAVer adopts exact and robust arithmetic based on the Parma Polyhedra Library (abbreviated as PPL, [10]). Piecewise affine dynamics with linear hybrid automata are handled by on-the-fly over-approximation. For more complex hybrid systems, in [21], experimental results indicate that SpaceEx can cope with hybrid systems with more than 100 variables which illustrate the scalability of the tool.

Furthermore, simulation based tools are widely used in industry. The most important tool is the Simulink/Stateflow toolset. We can model continuous-time systems in Stateflow charts. Continuous-time modeling allows us to simulate hybrid systems which respond to both continuous and discrete mode changes.

Moreover, we can debug models in Simulink/Stateflow, the debugging supports breakpoints on chart entry, event broadcast, state entry etc. In this paper, we detect the sand-

wich scenario by using the debugging. The advantage of debugging is that you can interact with the model during the analysis, and it is the feature that one can not find in the formal verification tools.

In this paper, we propose the Feedback-Advancement Verification (FAV) procedure to integrate verification with simulation of subway control systems. By using Matlab toolset with Simulink and Stateflow we will benefit from its rich library of primitive components. Moreover, it provides modeling and simulation capabilities for complex systems, such as nonlinear, continuous-time, discrete-time, multi-variable, and multi-rate systems. However, simulation does not guarantee correctness of the systems for unpredicted interactions with the environment. Formal method based tools avoid the flaw of simulation. However, to acquire appropriate analysis commands, it needs to experiment on verification repeatedly, as a result, it is time-consuming for complex systems. Simulation provides visual and concrete perspective which can be used as feedback to support the verification and improve our verification progress. Therefore, it is appropriate to combine simulation with formal verification.

We adopt the tools, i.e., SpaceEx/PHAVer and Matlab Simulink/Stateflow for verification and simulation of Platform Screen Doors System. Firstly, we construct the models in both tools, and then we take verification and simulation on bounded liveness properties, respectively. During the simulation, the sandwiched situation is revealed. This indicates that the passenger would be situated in the dangerous position that is between platform screen doors and train doors, therefore the passenger cannot go back to the platform. The reason for the sandwiched situation is that the time interval between platform screen doors closed and train doors closed is too short to let the passenger return to the platform. We modify the models and simulate this sandwiched situation and then modify the models in SpaceEx/PHAVer and verify the property. As a result, the ultimate models satisfy the time interval constraint and sandwich-free property if the passenger goes back to the platform during the time interval. And, we improve the model with four trains and four stations on a subway line. During the analysis we show that simulation and verification can feedback mutually.

This paper is organized as follows. In Sect. 2, we introduce the hybrid automata briefly, and the train automaton of subway control systems as an example to depict the concepts of hybrid automata. We describe several notations of Matlab Stateflow chart in Sect. 3. In Sect. 4, we propose the Feedback-Advancement Verification procedure which integrates verification with simulation. Sect. 5 describes the requirements in the Platform Screen Doors System. And in Sect. 6, the subway system is modeled with hybrid automata firstly, and then with the Simulink/Stateflow charts. Sect. 7 simulates and verifies these models with Matlab and PHAVer, respectively. We analyze the verification results and modify the models in Sect. 8. The improved models with the collision avoidance scenario are discussed in Sect. 9. During the formal verification, we detect a bug that is hard to be discovered in simulation. In Sect. 10 we discuss other reports on the PSDS and subway control systems and the applications concerning formal methods and simulation techniques. We conclude our procedure and discuss future works in Sect. 11.

2 Fundamentals of Hybrid Automata

In this section, we give a brief introduction about hybrid automata, including linear hybrid automata and affine hybrid automata. We also provide one example about the hybrid automaton of train travelling. Hybrid systems can be modeled as hybrid automata. A hybrid automaton consists of a finite control graph that represents the non-deterministic evolution of real-valued variables over time and discrete switches (transitions) with jump conditions. The nodes of the finite control graph are called control modes, and the edges are called control switches.

2.1 Hybrid Automata Definition

Definition 1 (Hybrid Automata) A hybrid automaton is a tuple $A = \langle X, V, flow, inv, init, E, jump, \Sigma, syn \rangle$ ([7]), where:

1. Variables. $X = \{x_1, x_2, \cdots, x_n\}$ is a finite set of n real-valued variables, where n is the dimension of A.
2. Control modes. V is a finite set of control modes. For each control mode $v \in V$, the initial condition $init(v)$ is a predicate over the variables in X. All initial states must satisfy the initial condition of the control mode.
3. Flow conditions. For each control mode $v \in V$, $flow(v)$ is a predicate over the variables in $X \cup \dot{X}$, where $\dot{X} = \{\dot{x_1}, \dot{x_2}, \cdots, \dot{x_n}\}$ and $\dot{x_i}(1 \leq i \leq n)$ is the first-order derivative of x_i with respect to time. Once the current control mode is v, the variables in X and their time-derivatives satisfy the flow condition $flow(v)$.
4. Invariant conditions. For each control mode $v \in V$, the invariant condition $inv(v)$ for mode v is a predicate over the variables in X. When the control mode is v, $inv(v)$ must be satisfied by the variables in X.
5. Control switches. For source control mode $v \in V$ and target control mode $v' \in V$, if the hybrid automaton A has a transition from v to v', then $(v, v') \in E$ is a control switch, note that E is a multi-set.
6. Jump conditions. For each control switch $e \in E$, there is a jump condition for it. The jump condition $jump(e)$ is a predicate over the variables in $X \cup X'$. For $1 \leq i \leq n$, $x_i \in X$ and $x_i' \in X'$ refer to the value of the variable x_i before and after the control switch, respectively.
7. Events. Σ is a finite set of events. For each control switch $e \in E$, the synchronization label $syn(e)$ represents an

event in Σ. In which, syn is a labeling function, $syn : E \to \Sigma$.

If v is a control mode and $r = (r_1, \cdots, r_n) \in \mathbb{R}^n$ is a value of variables X, then the pair (v, r) is a state of the hybrid automaton A. Since the invariant condition $inv(v)$ is a predicate over the variables in X, therefore the state (v, r) is admissible only if $inv(v)$ is true when the value of the variable x_i is r_i, for $1 \leq i \leq n$. We will illustrate this by an example in the next subsection (Sect. 2.3).

Definition 2 (Parallel Composition of Hybrid Automata) Given two hybrid automata:

$$A_1 = \langle X_1, V_1, flow_1, inv_1, init_1, E_1, jump_1, \Sigma_1, syn_1 \rangle,$$

and

$$A_2 = \langle X_2, V_2, flow_2, inv_2, init_2, E_2, jump_2, \Sigma_2, syn_2 \rangle.$$

Let $A_{1||2} = \langle X, V, flow, inv, init, E, jump, \Sigma, syn \rangle$ be the parallel composition of A_1 and A_2. Thus,

1. $V = V_1 \times V_2$, $X = X_1 \cup X_2$, $syn = syn_1 \cup syn_2$,
2. For $i = 1, 2$, $e_i = (v_i, v_i') \in E_i$, $syn_1(e_1) = syn_2(e_2)$, then $e = ((v_1, v_2), (v_1', v_2')) \in E$ with the jump condition $jump(e) = jump_1(e_1) \wedge jump_2(e_2)$, and synchronization label $syn(e) = syn_1(e_1) = syn_2(e_2)$,
3. If $v_1 \in V_1, v_2 \in V_2, v = (v_1, v_2) \in V$, then the invariant condition $inv(v) = inv_1(v_1) \wedge inv_2(v_2)$,
4. If $syn_1(e_1) \notin \Sigma_2$, then $e = ((v_1, v_2), (v_1', v_2)) \in E$ with $syn(e) = syn_1(e_1)$, $jump(e) = jump_1(e_1)$; it is similar when $syn_2(e_2) \notin \Sigma_1$,
5. $init((v_1, v_2)) = init(v_1) \wedge init(v_2)$ for $v_1 \in V_1, v_2 \in V_2$.

The most important point is that, if the control switches $e_1 \in E_1$ and $e_2 \in E_2$ with the same synchronization label $syn(e) \in \Sigma, e \in E$, then e_1 and e_2 should be synchronized in $A_{1||2}$. The synchronized e_1 and e_2 must occur whenever both $jump_1(e_1)$ and $jump_2(e_2)$ are true. Usually, we use the word 'command' or 'signal' to denote the control switches with the same synchronization label between two hybrid automata.

2.2 The Behavior of Hybrid Automata

The behavior of a hybrid automaton can be described as the composition of discrete and continuous transitions. The readers can refer to [31] for details concerning the semantics of hybrid automata.

Definition 3 (Labeled Transition Systems) A labeled transition system is a tuple $LTS = \langle S, S_{init}, L, \to_l \rangle$.

1. S is a set of states, and $S_{init} \subseteq S$ is a set of initial states.

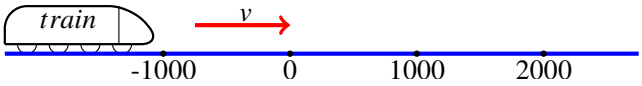


Fig. 1 A train on a line, travelling from left to right. Variable v denotes the velocity of the train. The station is at the position 0

2. L is a set of labels, for each label $l \in L$, the binary relation on the state space S is indicated by $\to_l$. Each triple $p \to_l q$ is called a transition of the system, for p, $q \in S$, and we call q the successor of p.

At this point, we can describe the behavior of a hybrid automaton as a labeled transition system.

Definition 4 (Transition Semantics of Hybrid Automata) The semantics of a hybrid automaton $A = \langle X, V, flow, inv, init, E, jump, \Sigma, syn \rangle$ is the semantics of a labeled transition system $LTS_A = \langle S^A, S^A_{init}, L^A, \to^A_{\alpha|\delta} \rangle$, where:

1. The state space of hybrid automaton A is S^A. The state space and initial states, $S^A, S^A_{init} \subseteq V \times \mathbb{R}^n$. State $(v, r) \in S^A$ iff $inv(v)$ is true when the value of the variables X is r, denoted by $inv(v)[r/X] = true$. Likewise, $(v, r) \in S^A_{init}$ iff $inv(v)[r/X] = true$ and $init(v)[r/X] = true$.
2. The set of labels consists of events and time durations, $L^A = \Sigma \cup \mathbb{R}_{\geq 0}$. For discrete behavior, we have the transition $(v, r) \to^A_\alpha (v', r')$ iff $\exists\, e \in E$, $jump(e)[r, r'/X, X'] = true$, $syn(e) = \alpha$, and the source and target of control switch e in hybrid automaton A is v and v', respectively. For continuous behavior that concerns the time duration $\delta \in \mathbb{R}_{\geq 0}$, the timed transition $(v, r) \to^A_\delta (v, r')$ iff $\exists f : [0, \delta] \to \mathbb{R}^n$, f is a differentiable function, $f(0) = r$ and $f(\delta) = r'$. For all time points $\varepsilon \in (0, \delta)$, $inv(v)[f(\varepsilon)/X] = true$ and $flow(v)[f(\varepsilon), \dot{f}(\varepsilon)/X, \dot{X}] = true$.

2.3 Example: Train Automaton

In this paper, the unit of distance is *metre*, the time unit is *second*, the unit for velocity is m/s, and m/s^2 is for acceleration or deceleration unless otherwise stated. Consider a train in the subway control systems (see Fig. 1). When the distance between train and station is -1000, a sensor signals its approach to the controller (as an arbitrator in the PSDS). Then the train approaches the station at a speed of $v \in [0, 16]$ and the acceleration $\dot{v} = -0.128$, indicating that the train will slow down. And, when the distance is -0.5, the train sends the *near_stop* signal to the controller, and the train will stop. Both distance and speed are reset to zero when the train stops. When the train leaves the station, its speed $v \geq 0$ and acceleration $\dot{v} = 0.128$, the train will speed up. The train is modeled by the automaton shown in Fig. 2.

The set of real-valued variables of the train automaton is $X = \{x, v\}$, where x is the distance between the train and the station, v is the speed of the train.

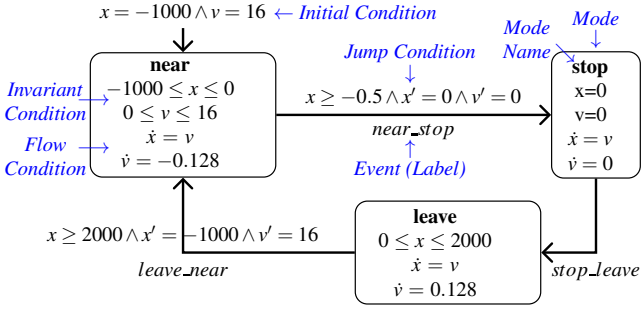


Fig. 2 Train automaton. The initial mode is the *near* mode. The initial distance is -1000 (meters) to the station. The train takes a velocity of 16 (m/s) at the beginning. The resetting of x and v to zero on transition *near_stop* is an abstraction of some underlying controller action that makes the train stop at the right position

The train automaton has three control modes and the set of control modes is $V = \{near, stop, leave\}$. The flow condition of the control mode *near* is $flow(near) = (\dot{x} = v) \wedge (\dot{v} = -0.128)$. The invariant condition of the control mode *near* is $inv(near) = (-1000 \le x \le 0) \wedge (0 \le v \le 16)$. Let $r = (r_1, r_2)$, where $r_1, r_2 \in \mathbb{R}$, r_1 denotes the value of variable x, r_2 denotes the value of variable v. If $r = (-1000, 16)$ and state $(near, r)$ is the initial state of the train automaton, then $(near, r)$ is admissible in mode *near* according to $inv(near)$.

The pair $cs = (stop, leave)$ is a control switch of the train automaton, the control switch $(stop, leave)$ corresponds to the event *stop_leave*. The jump condition (omitted in Fig. 2) of the control switch cs is $x = 0 \wedge v = 0 \wedge x' = 0 \wedge v' = 0$. The pair (p, q) is a *jump* of the train automaton if $p = (stop, r)$ and $q = (leave, r')$, where $r = (0, 0)$ and $r' = (0, 0)$.

2.4 Linear Hybrid Automata and Affine Hybrid Automata

Linear hybrid automata (abbreviated as LHA) [7] are a subclass of hybrid automata. The linear expression, linear constraint and linear predicate are illustrated as follows:

1. Linear expression. A linear expression over a set of variables *Vars* is of the form $s_1x_1 + \cdots + s_nx_n + t$, for $s_1, \cdots, s_n, t \in \mathbb{R}, x_1, \cdots, x_n \in Vars$, $\mathbb{R}$ is the set of real constants.
2. Linear constraint. A linear constraint is of the form $le \bowtie 0$, where le is a linear expression and the sign $\bowtie$ is in $\{<, \le, =\}$.
3. Convex linear predicate. A convex linear predicate is a finite conjunction of linear constraints.
4. Linear predicate. A linear predicate is a finite disjunction of convex linear predicates.

Thus, a hybrid automaton is a linear hybrid automaton if for every control mode $v \in V$, the invariant condition $inv(v)$, and the initial condition $init(v)$ are linear predicates. For every control switch $e \in E$, the jump condition $jump(e)$ is a linear predicate. The flow condition $flow(v)$ is a linear predicate over the variables in $\dot{X}$ only. The formal verification tool HyTech is designed to verify linear hybrid automata.

Affine hybrid automata [16] is a class of hybrid automata. For each control mode $v \in V$, the flow condition $flow(v)$ is an affine dynamics predicate. An affine dynamics predicate over $X \cup \dot{X}$ is the conjunction of the form $\dot{x} = le$, for le is a linear expression.

Comparing with the linear hybrid automaton whose flow condition is over variables in $\dot{X}$ only, the affine hybrid automaton is over variables in $X \cup \dot{X}$. For example, the train automaton in Fig. 2 is an affine hybrid automaton because the predicate $\dot{x} = v$ in the mode *near* is an affine dynamics predicate. The formal verification tool SpaceEx/PHAVer is designed to verify affine hybrid automata.

2.5 Reachability Analysis

Whether a system satisfies the safety property can be determined by the set of reachable states. For a given hybrid automaton A, there is a continuous timed transition $(v, r) \rightarrow_\delta (v, r')$, where (v, r) and (v, r') are two states in the control mode $v \in V$, $\delta \in \mathbb{R}_{\ge 0}$ is a time duration. Let the differentiable function $f : [0, \delta] \rightarrow \mathbb{R}^n$ represents the curve of the flow, thus, during the time duration δ, $f(0) = r$, $f(\delta) = r'$ and the state $(v, f(t))$ is admissible for each $t \in [0, \delta]$. In addition, the flow condition $flow(v)$ is true over the time interval $[0, \delta]$. Furthermore, there is a discrete transition: $(v, r) \rightarrow_\alpha (v', r')$, where the control switch $e = (v, v') \in E$ and $\alpha = syn(e)$ (α is a synchronization label). We can obtain the set of reachable states by repeatedly computing the successors of continuous timed and discrete transitions. Let S be a set of states of A, the continuous timed successors and discrete successors are:

1. Timed successors:

$$Post_t(S) = \{(v, r') | \exists \delta \in \mathbb{R}_{\ge 0}, (v, r) \in S : (v, r) \rightarrow_\delta (v, r')\};$$

2. Discrete successors:

$$Post_d(S) = \{(v', r') | \exists \alpha, (v, r) \in S : (v, r) \rightarrow_\alpha (v', r')\}.$$

The set of reachable states denoted by $Reach(Init)$ is the smallest fixpoint S of the equation,

$$S = Post_t(Post_d(S)), Post_t(Init) \subseteq S.$$

There are also definitions of timed and discrete predecessors. And the set of predecessors, which can reach S is denoted by

$$Reach^{-1}(S).$$

More details can be found in [7].

The process to compute $Reach(Init)$ is used for the forward analysis: starting from the initial states $Init$, iterate

the operator $Post_t$ and $Post_d$, until a fixpoint is reached and then check whether the intersection of the unsafe states and $Reach(Init)$ is empty. In a similar way, the backward analysis: starting from the unsafe states $Unsafe$, iterate the operator Pre_t (timed predecessors) and Pre_d (discrete predecessors), until a fixpoint is reached and then check whether the intersection of the initial states and $Reach^{-1}(Unsafe)$ is empty.

The formal verification tool HyTech implements the forward and backward analysis. But, PHAVer does not implement the backward analysis directly. Nevertheless, to carry out the backward analysis in PHAVer, one uses the reverse command [19] to obtain reverse causality in the automaton and then uses forward analysis.

3 Stateflow Chart Notations

Each state label in the chart of Stateflow starts with the name of the state followed by an optional slash ('/') character. The slash is optimal if the state name is followed by a carriage return. For example, in Fig. 9, the initial state of train chart is the state with name *near*. After the name, the state actions are declared with a keyword label ('en','du',etc.) that identifies the type of action and a colon (':'). State actions are optional as well. There are five types of state actions, described as follows.

1. Entry Action. The keyword label is 'entry' or 'en' for short. In the state *near* of train chart in Fig. 9, the three entry actions are '$distance = -1000$', '$distance\_out = distance$', and '$v = 16$'. This means the value of *distance* is reset to -1000 meters, the value of *distance_out* is reset to the new value of *distance*, and the value of v is reset to 16 m/s, whenever state *near* is entered.
2. During Action. The keyword label is 'during' or 'du' for short. In Fig. 9, state *near* has three during actions, '$distance\_dot = v$' denotes a differential equation that the first derivative of *distance* is v, '$v\_dot = -0.128$' represents the differential equation that the first derivative of v is -0.128 m/s^2, '$distance\_out = distance$' denotes the value of *distance_out* equals *distance*, whenever state *near* is already entered and any event (the sampling of Simulink in our case) occurs.
3. Exit Action. The keyword label is 'exit' or 'ex' for short. If a state is active (entered), but becomes inactive (exited), this action is executed.
4. On Event_Name Action. The keyword label consists of 'on' and *event_name*. If a state is active and its event with *event_name* occurs, the corresponding action is executed. However, this kind of action is not supported in Stateflow chart with continuous variables, for instance, it is not allowed in our case.
5. Bind Action. The keyword label is 'bind'. This action is used to specify the write permission of variables for the state that the action is positioned. It means that the state and its children can change the value of the variables specified in the bind action, other states can read but not write.

Transition Label Notation: A transition in Stateflow chart is a label consists of an event, a condition, a condition action, and/or a transition action. The general form of a transition is:

$$event[condition]\{condition\_action\}/transition\_action$$

In a transition, the event is optional. When an event is specified, the occurrence of the event is the precondition of the transition to be taken. If the event is not specified, the condition of a transition is a Boolean expression which denotes that the transition can be taken when the condition is true. If both event and condition are specified, then the transition can be taken when the event occurs and the condition is true. The condition action is executed whenever the condition is true. An implied condition is true if no explicit condition is specified. The transition action is executed after the transition destination has been determined to be valid. Normally, a transition is valid when the source state of the transition is entered and the event (if specified) occurs and the condition is valid. For detailed notations, readers can refer to the user's guide[1] of Stateflow.

4 Co-Verification Procedure

The formal verification of hybrid system is based on the reachability analysis with hybrid automata. The traditional formal method based tools (e.g., HyTech, PHAVer) have the ability to provide parametric analysis, offer analysis commands for users, and generate error-trace. On the other hand, using Matlab toolset with Simulink and Stateflow, numerous benefits can be achieved by its rich library of primitive components. Simulation data is visualized in the Matlab toolset and we can take advantage of its interactive features. Moreover, it provides modeling and simulation capabilities for complex systems. With simulation, we are not sure about the correctness of the systems because of the unpredicted interactions with the environment.

On the contrary, formal methods based tools avoid the limitation of simulation that cannot guarantee design correctness. However, appropriate analysis commands are not easy to acquire with formal methods tools, it needs to conduct experiment on the verification many times. Hence it is time-consuming for complex hybrid systems.

In our case study, the Stateflow/Simulink model is a refinement of the SpaceEx/PHAVer model, which is closer to

[1] http://www.mathworks.com/help/pdf_doc/stateflow/sf_ug.pdf

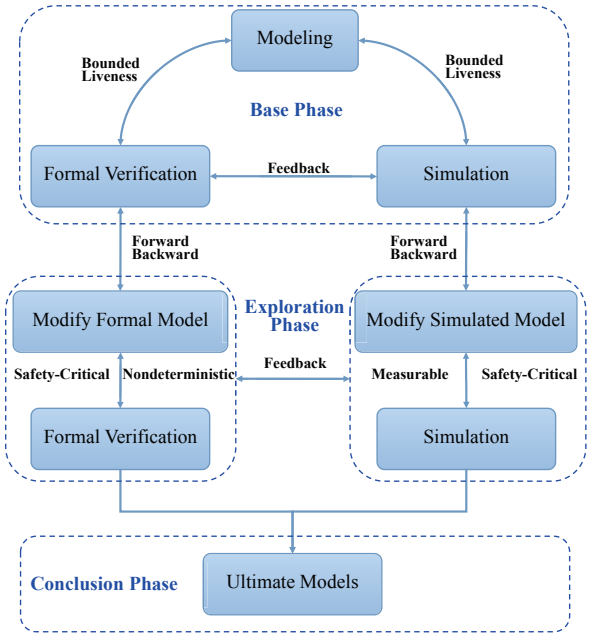


Fig. 3 Feedback-Advancement Verification. FAV consists of three phases, base phase, exploration phase and the conclusion phase

a final implementation, and this could be proved by suitable refinement relations (e.g. timed simulation or timed trace inclusion [31, 38]). Moreover, the two models are complementary for some features (e.g.,visualization of simulation, correctness proving by verification), stressing different aspects of the overall system and permitting complementary analysis techniques, i.e. verification versus simulation.

Therefore, it is feasible to verify and analyze hybrid systems by combining formal verification with simulation. In this paper, we propose the procedure FAV (Fig. 3). This procedure is composed of three phases: base phase, exploration phase and conclusion phase.

1. Base phase: Construct models by formal method (e.g., hybrid automata) and simulation technology (e.g., the Simulink/Stateflow models), respectively. Bounded liveness and non safety-critical properties are checked in this phase. The models are modified according to the results of simulation and verification. As both hybrid automata and Stateflow charts are varieties of finite-state machine, they are consistent with parts of the model structure and syntax, although not in all aspects. For instance, the state in Stateflow is corresponding to the mode in hybrid automata. The flow conditions in the mode of hybrid automata can be described as the during actions in Stateflow state. For example, the flow condition '$\dot{v} = 1$' is consistent with the during action '`du:v_dot=1`'. And the flow condition '$\dot{v} = dv$' with invariant '$-1 \leq dv \leq 0$' can be implemented as during action '`du:v_dot=v_shut`', and the value of `v_shut` can be changed by a manual switch, as we implemented in the Simulink model in Fig. 8. The hybrid automata model in SpaceEx/PHAVer is considered as an abstraction of the Simulink/Stateflow model. Likewise, the model in Simulink/Stateflow is a refinement for the model in SpaceEx/PHAVer. The main objective of this phase is to check whether the models are correct or not in a coarse-grained manner. Therefore, the properties to be checked in this phase are simple (e.g., the train can stop or leave, given the initial states and bounded time) and without concerning the safety. The complex and safety-critical properties are checked in the exploration phase.
2. Exploration phase: In this phase, safety critical properties (e.g., no one is injured in the system) or complex interactive situations of the system (e.g., *Human-Machine Interaction Protocol*) are researched with the combination method of formal verification and simulation. In terms of the conclusion of base phase, if errors or problems occur during verification, then we manually modify formal models and proceed with verification. This process is repeated until the model satisfies the safety properties or the process of verification is failed because of the limitation of tool or resources. On the other hand, if errors or problems occur during simulation, then we modify simulated models and proceed with simulation. Based on the capability of verification tools (e.g. state exploration), some complex critical properties cannot be verified in verification tools. As a result, it shall turn to simulation finally. For simulation, one cannot simulate all the interaction with environment, and it is required to verify the correctness of the new models in verification tools. The revision of models is a conductive modification based on feedbacks between two types of models. For instance, the correction of the controller in Fig. 14 can be illustrated by a conductive graph in Fig. 4. The state with constraint '$z >= 0$' in Fig. 4 represents the state of '`about_to_close2`' in Fig. 14. The edge in the conductive graph denotes the transition with conditions and actions, the actions are executed when the transition is valid according to the conditions. In the conductive graph, the transitions have different priorities, the transition with priority 1 is prior to the one with priority 2. Thus, the transition with priority 1 will be tested at first. The value of z will be reset to 0 and $z << 5$ (much less than 5) when the train doors are not closed. If the train doors are closed, the transition with priority 2 can be tested. As the condition '$z >= 5$' is contained in the transition, the transition will be valid when $z \geq 5$. According to the conductive graph, the priorities for transitions (i.e., control switches) in Fig. 15a are guaranteed by variable c_1, because the condition $c_1 = 0$ is true whenever the transition with label $shut\_closed_1$ is valid. Therefore, the transition with label $close_2$ can only be triggered after the transition with label $shut\_closed_1$ is finished. In addi-

Fig. 4 Conductive graph. Each rectangle denotes one state. The edge between states is considered as a state transition. The constant numbers (e.g., 1 and 2) near transitions denote the priorities for transitions

tion, the transition with label $shut\_closed_1$ sets the value of z to 0, it restricts the time duration to 5 seconds between these two transitions. With the conductive graph the consistent modification can be implemented in the models of Simulink/Stateflow and SpaceEx/PHAVer.

3. Conclusion phase: After the second phase, the final (particular for the properties that had been checked up to this time) model of formal verification or simulation is obtained and can be applied to new developing stages in model-based design.

The exploration phase is used to verify the safety properties and provide a correct model. Moreover, formal verification and simulation could influence each other by feedback.

As the diversities both exit on the expressive power and semantics of the two types of models, maintaining the consistency between them is very challenging for complex hybrid systems. In our procedure, we prefer to do the modeling incrementally. Each modification focuses on a small part of the models, e.g., one mode, a control switch or just one condition expression. The fast and visual reaction of the simulation by Simulink/Stateflow could support this kind of fine tuning on models.

In the following, we utilize our method FAV to model and verify the subway control system. We first illustrate requirements of the subway control system, and then model requirements and verify models in the tools: SpaceEx/PHAVer and Matlab toolset. With simulation, we simulate subway environment and find that the models do not satisfy the safety property. Then we modify models and verify the correctness of the new models with SpaceEx/PHAVer. The visualized simulation results are also given by Matlab toolset.

5 Requirements of Platform Screen Doors in Subway Control Systems

This paper focuses on the Platform Screen Doors System (PSDS). The platform screen doors are widely used at subway stations around the world. These doors screen the platform from the train, and prevent passengers from falling off the platform that is probably due to suicide or the piston effect [12].

The PSDS is a safety-critical system. One small design deficiency in the system may cause serious accident. For example, on 15 July 2007 in Shanghai, China, a man tried to enter a crowded train at the subway station, but failed. When the train doors and platform edge doors closed almost simultaneously, he was sandwiched between closed doors and fell off. Consider the three different schemes of closing train doors and platform screen doors:

1. Simultaneously. The platform screen (or edge) doors and train doors close simultaneously. As we mentioned before, passengers may be sandwiched in the middle and unable to go back to the platform or enter the train. This scheme is not appropriate for the PSDS.
2. Platform Screen Doors First (PSDF). First, we close the platform screen (edge) doors, and then the train doors. This scheme is also improper when the platform screen doors is closed but the train is crowded. Thus, some passengers fail to squash into the train, it needs to open the platform screen doors again.
3. Train Doors First (TDF). First, close the train doors, and then the platform screen doors. Passengers can return to the platform when the train doors are closed. Moreover, there is enough time to go back to the platform before the platform screen doors are closed.

Although it may need other measures to improve safety, we focus on the Platform Screen Doors Systems with the TDF scheme in this paper. The most important requirement is that, before the train starts to leave a station, the passengers should not be sandwiched (i.e., the passengers cannot return back to the platform) between the train doors and screen doors in a period of time after the train doors are closed. Let T_1 be the time when the train doors are closed, T_2 be the time when the screen doors start to close, then the requirement can be declared as a simple formula,

$$T_2 - T_1 \geq C \wedge C > 0$$

where, C is a constant, denoting a time duration, for instance $C = 5$ (seconds) in Sect. 6.

6 Models

We divide the whole system into four parts, train, train doors, platform screen (edge) doors and controller.

6.1 Hybrid Automata Models

Now, we use four automata to formalize the Platform Screen Doors System. The train automaton (Fig. 2) represents the travelling of the train. The train doors automaton (Fig. 5) and platform screen doors automaton (Fig. 6) represent the opening and closing of the train doors and screen doors, respectively. The controller automaton (Fig. 7) represents as an arbitrator between these three automata.

The train doors automaton (see Fig. 5) has four control modes. Modes *open* and *closed* denote the train doors are

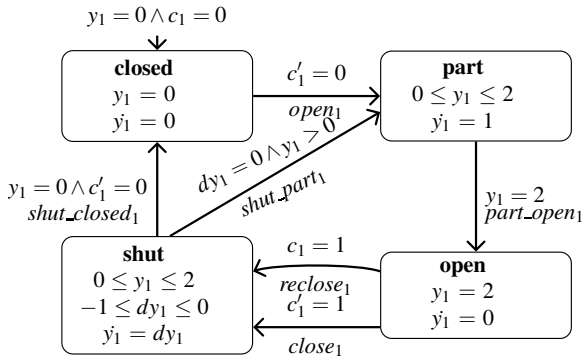


Fig. 5 Train doors automaton. Initially, the train doors are closed. During the mode *part* the doors are opening. Whenever the variable y_1 equals 2 (meters), the doors are opened. The statement '$\dot{y_1} = 1$' means the velocity of opening the door is 1 m/s, i.e., the distance between two door leafs increases at the speed of 1 meter per second

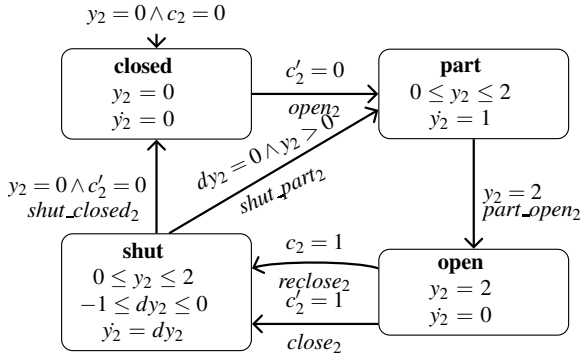


Fig. 6 Platform screen doors automaton. The same as the train doors, the train doors are closed, initially. During the mode *part* the doors are opening. Whenever the variable y_2 equals 2 (meters), the doors are opened. The statement '$\dot{y_2} = 1$' means the velocity of opening the door is 1 m/s

opened and closed, respectively, modes *part* and *shut* denote that the train is opening doors and closing doors, respectively. Each train door consists of two leafs, left door leaf and right door leaf. The variable y_1 represents the distance between the left door leaf and right door leaf. It means that the door is opened when the distance is 2 and then $y_1 = 2$. Similarly, if the door is closed then the distance is 0 and the value of y_1 is 0. Initially, the door is closed, and after receiving an $open_1$ command from the controller, the door will be opened at a rate of 1. When the door is opened and receives a $close_1$ command, it will be closed at a rate of dy_1, where the value of the variable dy_1 ranges between -1 and 0. When the rate is zero but the door is not closed, someone or something is clipped. Thus the door needs to be opened again and then after the door is open, passengers leave the door and the controller also needs to close the door again. The variable c_1 is a guard to re-close the door. We assign 1 to c_1 when the door receives the $close_1$ command. We check the value of c_1 when the door is open. If $c_1 = 1$, close the door again.

The platform screen doors automaton is the same as the train doors automaton except variables. These two automata have similar behavior. They are managed by the controller.

The controller automaton (Fig. 7) plays a role of an arbitrator. It receives signals from train, sends commands to train, train doors and platform screen doors. Initially, the controller is at control mode *idle*. When it receives the signal *near_stop*, the controller enters the mode $about\_to\_open_2$. Five seconds later, it sends command $open_2$ to the platform screen doors and waits 5 seconds. The platform screen doors begin to open when it receives command $open_2$. Then the controller sends command $open_1$ to the train doors. At the same time, the train doors begin to open. Six seconds later, the controller starts to ring (as an alert). After the alert off, the controller waits five seconds and then sends command $close_1$ to train doors. Train doors begin to close. Five seconds later, the controller sends command $close_2$ to platform screen doors if train doors are closed ($c_1 = 0$ denotes train doors are closed). Similarly, five seconds later, it sends command *stop_leave* to train if platform screen doors are closed. After that, the train leaves the station and the controller returns to mode *idle*.

6.2 Simulation Models

In this section, we model our PSDS in Simulink/Stateflow. Our Stateflow chart consists of four subcharts. Fig. 9 represents the Stateflow chart of train. Fig. 10 represents the train doors (the chart of platform screen doors in Fig. 11 is the same as the train doors except the variable names). The controller is shown in Fig. 12.

In Fig. 8, we use two *Manual Switches* to simulate the velocity of closing doors, one for the train doors and another for the platform screen doors, they select values from -1 and 0. Accordingly, *v_shut1* and *v_shut2* are two *input variables* which represent the values of the *Manual Switches* selected. Furthermore, there are three *output variables*, *y1_out*, *y2_out* and *distance_out*. The value of *y1_out* equals *y1* (see Fig. 10) and the value of *y2_out* equals *y2*. *y1_out* and *y2_out* represent the distances between the two sides of doors, train doors and platform screen doors, respectively. The distance between train and station is represented by *distance_out*. We use the *Scope* block to display these three *output variables* with respect to simulation time.

In the train Stateflow chart shown in Fig. 9, *distance* is a *local variable* which denotes the distance between train and station. And, *v* is also a *local variable* which denotes the velocity of the train. The expression *distance_dot* represents the time derivative of the variable *distance*, therefore its value equals the velocity of train which is denoted by *v*. Similarly, the expression *v_dot* represents the time derivative of *v*, thus, *v_dot* denotes the acceleration of the train.

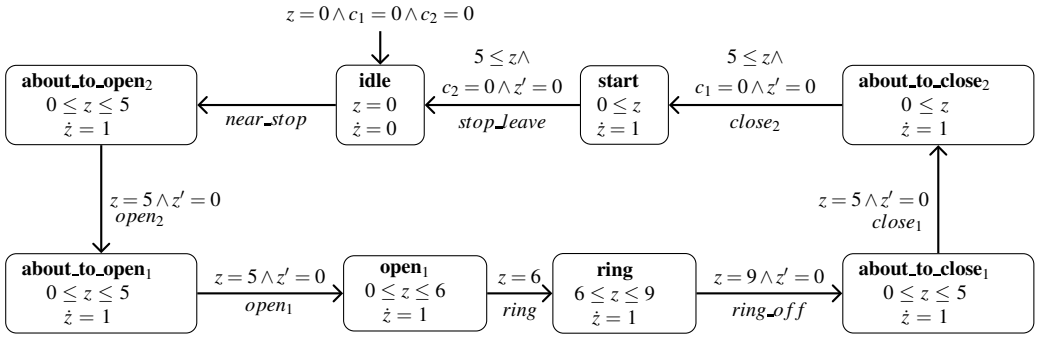


Fig. 7 Controller automaton. The variable z is used as a clock in this automaton. Variable c_1 is shared with train doors automaton, c_2 is shared with platform screen doors automaton. The synchronized label *near_stop* is shared with the train automaton

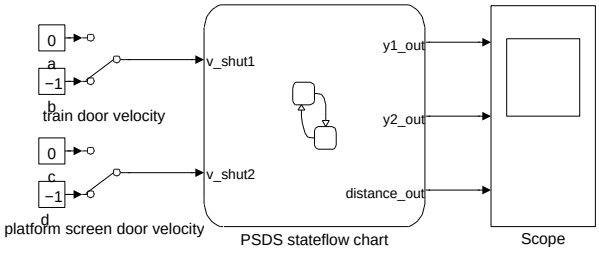


Fig. 8 Simulink model of PSDS. The left part is the Manual Switches, controlling the velocity of train doors and platform screen doors when closing doors. The Stateflow chart is in the middle. Variable *y1_out* (or *y2_out*) is the output variable denoting the state of the train (or platform screen) doors

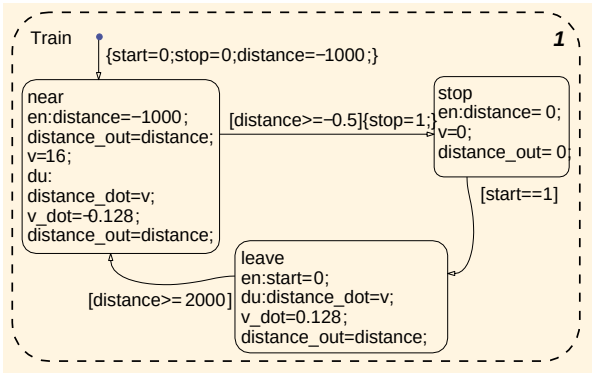


Fig. 9 Train chart. Variables *start* and *stop* are shared between train and controller for the control of the train departure and stopping

In addition, "en:" is the prefix for entry actions which execute when the state is entered, and, "du:" is the prefix for during actions in which the derivatives and updates for variables are defined, the during actions are executed at every time step when the sate is active and no valid transition to another state is available.

In the Stateflow chart, we use *local variables* as shared variables between different Stateflow subcharts. The *local variables* defined in a Stateflow chart can be read and written in the chart's all subcharts. For example, the variable *stop* is used in train chart and controller chart. The condition action $stop = 1$ is executed as soon as the condition $distance >= -0.5$ of the transition from state *near* to *stop* is true. Then in the controller chart, the condition $stop == 1$ is true. Therefore the transition from state *idle* to *about_to_open2* will be active. Similarly, variable *start* is also the shared variable between train and controller. Variable *open1* (*open2*) and *close1* (*close2*) are shared variables between train doors (screen doors) and controller.

The variable *close1* has the similar purpose as c_1 (see Fig. 5) which is used as the guard to re-close train doors. And, the variable *open1* is the guard to open train doors. In the same way, *open2* and *close2* have corresponding purposes, respectively.

Moreover, we use the predicate

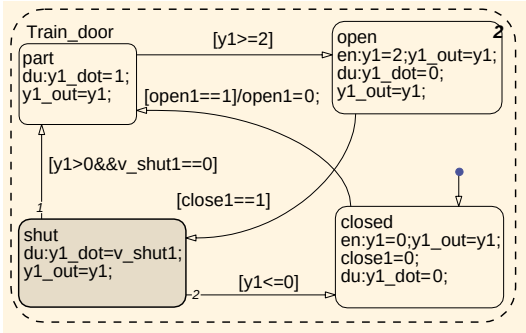


Fig. 10 Train doors chart. Variables *open1* and *close1* are shared between train doors and controller

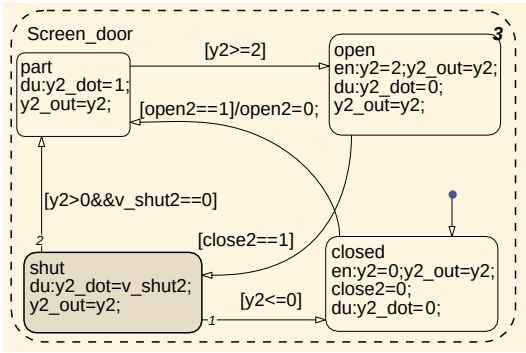


Fig. 11 Platform screen doors chart. Variable *open2* and *close2* are shared between platform screen doors and controller

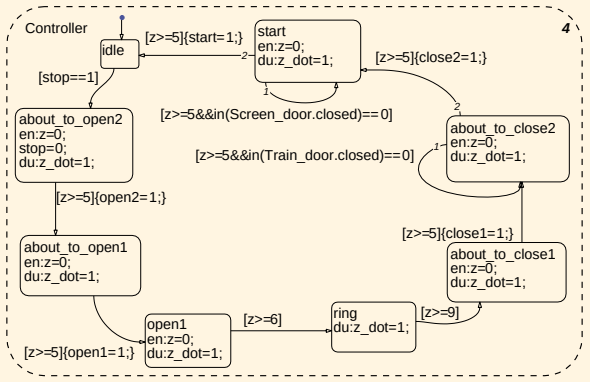


Fig. 12 Controller chart. Here, the variable z is used as a clock. The predicate *in(...)* is employed to check whether the doors are closed

```
in(Train_door.closed) == 0
```

in the controller Stateflow chart, indicating whether the train doors are closed. Here *Train_door* is the name of the train doors chart (see Fig. 10) and *closed* is one of the states of the train doors chart. If the predicate is true, the train doors are not closed. Therefore, a self-loop transition to substates *about_to_close2* and *start* is added (see Fig. 12), and the controller continues waiting until train doors or platform screen doors are closed.

7 Formal Verification and Simulation for PSDS

Given a property P of a hybrid system, formal verification is to answer whether the hybrid system satisfies the property P. For safety-critical systems, we want to check whether a system cannot reach a set of unsafe states S_{unsafe}. Nevertheless, we do not verify the safety property of a system at first. Conversely, first of all, we check the bounded liveness and simple properties of the system, and then the safety property. In the following, we will check four properties of PSDS by formal verification and simulation, respectively.

7.1 Formal Verification

PHAVer is a tool for the verification of hybrid systems with piecewise constant bounds on the derivatives. PHAVer deals with affine dynamics with linear hybrid automata by using an on-the-fly over-approximating algorithm. PHAVer is free from overflow errors by using PPL [10] and GMP (GNU Multiple Precision Arithmetic Library) [23]. In this paper, we adopt the formal verification tool PHAVer to verify hybrid automata models.

The following command statement represents that the four automata are composed using ampersand symbol (`&`):

```
sys = controller & traindoor & screendoor & train.
```

Thus, `sys` is the composition of the four automata which have been described in Sect. 6, representing the PSDS illustrated in this paper. A *symbolic state* is denoted by a combination of a control mode name and a linear formula, for example, `stop & x==0` represents the state when the train is stop and the value of x is zero.

The initial state of our system is represented as follows:

```
1  idle ~ closed ~ closed ~ near &
2  z==0 & x==-1000 & v==16 &  y1==0 &
3  y2==0 & close1_flag==0 & close2_flag==0
```

where mode names of automata in the composition are concatenated with $\sim$ in proper order at line 1. `idle` is the mode in `controller` automaton. And, `closed`, `closed`, `near` are the modes in automata `traindoor`, `screendoor` and `train`, respectively. In addition, `close1_flag` represents the variable c_1 in Fig. 5, and `close2_flag` represents the variable c_2. The following four (bounded) liveness properties are checked for PSDS in our work:

1. **Leaving and Stopping:** At first, we want to analyse whether the train can depart from the station, or whether it can stop at the station. The states that the train can leave and stop are denoted by the following states:

```
1  sys.{
2      $ ~ $ ~ $ ~ leave & True,
3      $ ~ $ ~ $ ~ stop & True
4  }.
```

The expression "`$ ~ $ ~ $ ~ leave`" refers to all the modes compositions except that the train automaton is in mode *leave*, and "`$ ~ $ ~ $ ~ stop`" refers to those in which train automaton is in the mode *stop*. And the identifier `True` refers to all the possible evaluations of variables in automata. The verification of properties in PHAVer is by reachability analysis. The command `sys.reachable` returns the set of states reachable in the automaton `sys` from the initial states. And the command

```
identifier1.intersection_assign(identifier2)
```

intersects the `identifier2` with `identifier1`, then puts the result into `identifier1`, where `identifier1` and `identifier2` represent two sets of states. As a result, the two commands

```
1  sys.reachable
2  intersection_assign(forbidden)
```

can be used to verify whether the unsafe states can be reached from the initial states, where `sys.reachable` represents the states that can be reached by the system (`sys`) and `forbidden` denotes the set of unsafe states. The verification results given by PHAVer support that the above states (leave and stop) are reachable from the initial state.

2. **Ringing:** Both train doors and platform screen doors are opened when the bell is ringing. Consider the following states denoted by:

```
sys.{ ring ~ $ ~ $ ~ $ & True }.
```

The expression "`ring ~ $ ~ $ ~ $`" refers to all the mode compositions in which controller automaton is in the mode *ring*. We can list the intersection of the above states and all the reachable states, then check the modes that the train doors automaton and platform screen doors automaton can be in, provided that the bell is ringing. The intersection is given by PHAVer as follows:

```
controller ~ traindoor ~ screendoor ~ train.{
    ring ~ open ~ open ~ stop &
    6<=z<=9 & x==0 & y2==2  & y1==2 &
    close2_flag==0 & close1_flag==0
}.
```

Where, the `traindoor` and `screendoor` are both in the mode `open`. Therefore, we are sure that both train doors and platform screen doors are open whenever the bell is ringing.

3. **Ordering:** The property declares that, the train doors need to be closed at first, and then the platform screen doors. The screen doors need to keep open when the train doors are being closed. Consider following states denoted by:

```
sys.{ about_to_close2 ~ $ ~ $ ~ $ & True }.
```

Where, the expression "`about_to_close2 ~ $ ~ $ ~ $`" refers to all the mode compositions in which controller automaton is in the mode $about\_to\_close_2$. And, the controller has sent command $close_1$ to train doors when the controller automaton is positioned in the mode $about\_to\_close_2$. We can determine whether the property holds by checking the intersection of the above states and all reachable states. We have the following intersection by PHAVer, and the result states demonstrate that the screen doors are opened when the train doors are closing:

```
controller ~ traindoor ~ screendoor ~ train.{
    about_to_close2 ~ shut ~ open ~ stop &
    -1 <= dy1 <= 0 & 0 <= y1 <= 2 & x == 0 &
    y2 == 2 & close2_flag == 0 &
    close1_flag == 1 & z + y1 >= 2,

    about_to_close2 ~ open ~ open ~ stop &
    x == 0 & y2 == 2 & y1 == 2 &
    close2_flag == 0 & close1_flag == 1 &
    z >= 0,

    about_to_close2 ~ closed ~ open ~ stop &
    x == 0 & y2 == 2 & y1 == 0 &
    close2_flag == 0 & close1_flag == 0 &
    z >= 2,

    about_to_close2 ~ part ~ open ~ stop &
    0 < y1 <= 2 & x == 0 & y2 == 2 &
    close2_flag == 0 & close1_flag == 1 &
    z + y1 >= 2
}.
```

Furthermore, the train doors should be closed when the controller is in mode *start*. And the states the controller is in the mode *start* are denoted by:

```
sys.{ start ~ $ ~ $ ~ $ & True }.
```

In the same way, we can check the intersection of the above states and all reachable states, and then draw the conclusion.

4. **Operation of Doors:** There are no operations of the doors before the train stops or after the train departs from the station. Therefore, the following states should not be reachable:

```
sys.{
    $ ~ open ~ $ ~ $  & x>0,$ ~ part ~ $ ~ $  & x>0,
    $ ~ shut ~ $ ~ $  & x>0,$ ~ open ~ $ ~ $  & x<0,
    $ ~ part ~ $ ~ $  & x<0,$ ~ shut ~ $ ~ $  & x<0
}.
```

Through the formal verification with PHAVer, the above four properties are totally satisfied by our PSDS models.

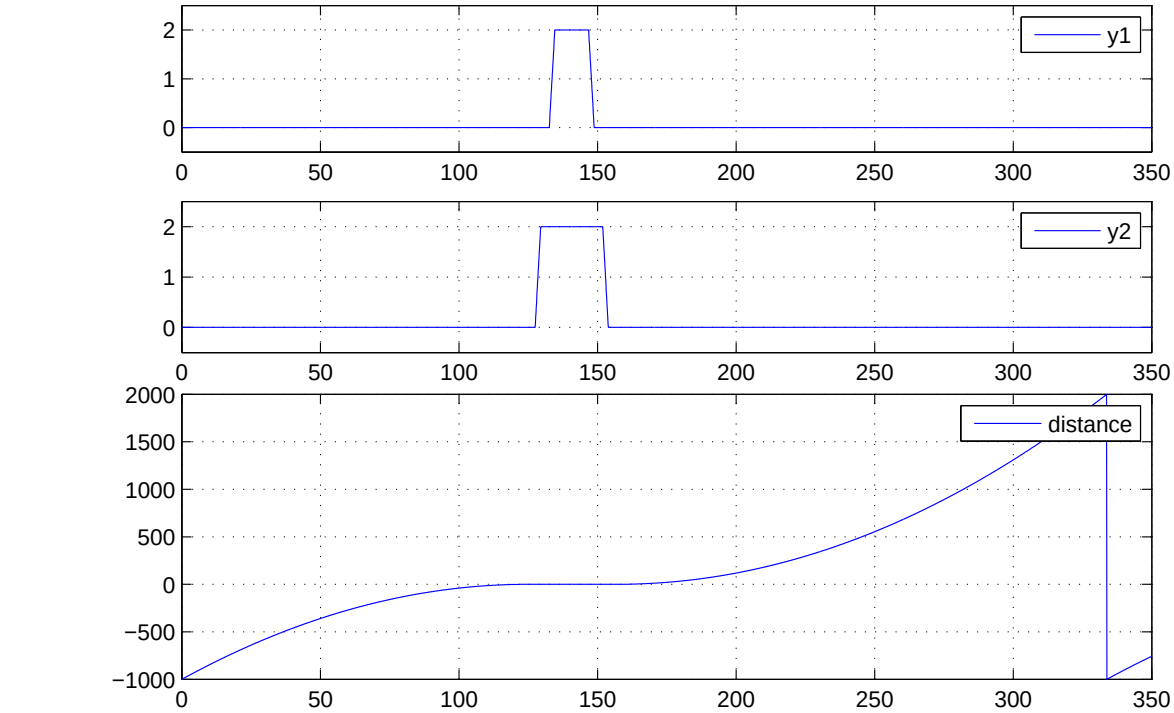
7.2 Simulation

The four (bounded) liveness properties are verified and no errors have been detected in PHAVer. Next, we consider the simulation of our models with the Matlab toolset. As the models have been constructed in Sect. 6, we can analyze the activities of train doors, screen doors and train with the *Scope* block for the result illustration. Fig. 13a displays one normal run of the train from *near* to *leave*.

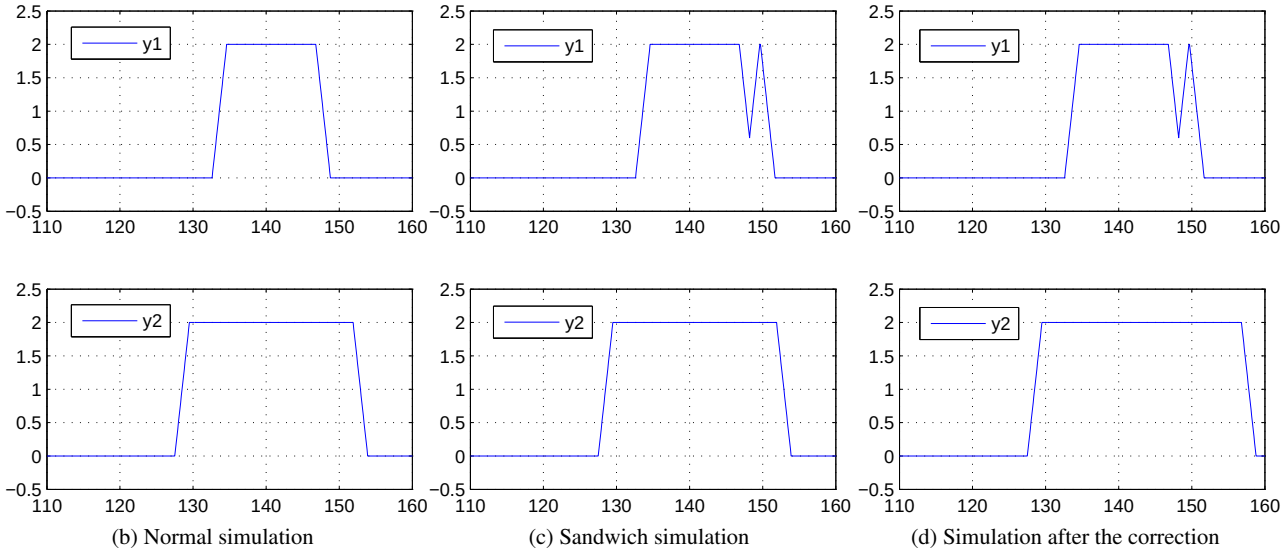
In Fig. 13a, *y1* represents the activities of train doors, *y2* represents the activities of platform screen doors. Here *distance* denotes the activities of the train (i.e., the distance between train and subway station). The *distance* ranges over $[-1000, 2000]$, and the train stops at the station when *distance* equals zero. After the train stops, *y2* increases to 2, which means that the screen doors are opened. And then, *y1* also increases to 2, which means that the train doors are opened. A few seconds later, *y1* decreases to 0, and then *y2* also decreases to 0. It means that the train doors have closed before screen doors begin to close.

In Fig. 13a, we choose the running of the system within about 350 seconds period, in this simulation, the train can leave and stop, thus, the first property *Leaving and Stopping* is satisfied. The variable *y2* equals 2 for several seconds when *y1* decreases from 2 to 0. Thus, the first part of third property *Ordering* is satisfied.

Moreover, the fourth property *Operation of Doors* is also satisfied, because all the variations of *y1* and *y2* are taken place when *distance* equals 0. The other properties can be checked by the analysis of the Stateflow chart with simulation or debugging.



(a) One normal running



(b) Normal simulation (c) Sandwich simulation (d) Simulation after the correction

Fig. 13 Simulation results of PSDS. (a) illustrates an ordinary running of the system. The change of variable *y1* displayed at the top of (a) indicates the state of the train doors. And *y2* is for the screen doors. The number 2 represents that the doors are open, number 0 for closed state. The third part at the bottom of (a) shows the moving of the train. (b) is the detail state for the doors in the normal situation. (c) is for the sandwiched situation. (d) is the result after correction

As mentioned before, we can simulate the situation that someone is sandwiched between the closed doors. The process is described as: first, we start the Stateflow debugger and continue simulating until the variable *y1* begins to decrease. After that, we change the closing speed of train doors from -1 to zero by double-clicking the train door velocity *Manual Switch*, then the train doors re-open and then re-close.

As depicted in Fig. 13c, screen doors begin to close almost at the moment the train doors closed, so the sandwiched situation occurs. We can prolong the closing time of train doors by changing the closing speed for several times to simulate the sandwiched situation.

The normal situation and sandwiched situation are illustrated in Fig. 13b and Fig. 13c, respectively. The good point is that the platform screen doors do not begin to close before

the train doors are closed. But the weak point is that the time interval (denoted by T_{dc}) between the moments when train doors are closed and screen doors begin to close is smaller than it is in the normal situation. Furthermore, the platform screen doors may close immediately when the train doors are closed. Therefore, it is required to set a number of seconds (i.e., T_{dc}) for passengers to return to the platform when train doors are closed. And, the previous PSDS hybrid automata also did not consider this time interval. As a result, we need to modify our models, not only hybrid automata but also the Matlab Simulink/Stateflow charts.

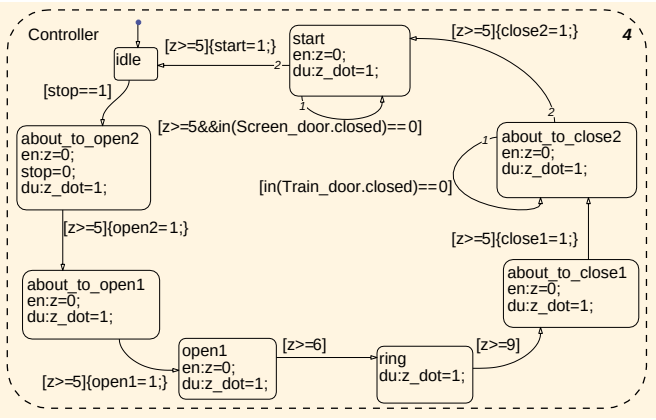


Fig. 14 Correct Controller chart

8 Correction

In this section, we will modify the models of the controller, aiming to avoid the sandwiched situation under proper time interval T_{dc}. Firstly, we do the correction for the simulation models and then the hybrid automata.

8.1 Simulation Correction

For Stateflow chart (see Fig. 12), the self-loop transition of the state *about_to_close*2 contains the transition condition:

```
z>=5 && in(Train_door.closed)==0.
```

where, the predicate $in(.)$ can be evaluated to *true (1)* or *false (0)*. If the train doors are closed, then the predicate $in(Train\_door.closed)$ equals 1, as a result, the boolean expression $in(Train\_door.closed) == 0$ is false.

Let's consider the following situation. If the train doors are closed at the moment $z == 4.9$, then the Stateflow predicate $in(Train\_door.closed) == 0$ is false. Thus, the transition from *about_to_close*2 to *start* will occur when $z == 5$, and the time interval is 0.1s which is smaller than 5s. On the contrary, if we remove $z >= 5$ from ($z >= 5$ && $in(Train\_door.closed) == 0$), then we just check whether the train doors are closed in the transition condition, and the new controller Stateflow chart is in Fig. 14. The new transition condition makes the controller reset the variable z to 0 if the train doors are not closed. Moreover, the variable z is needed to increase from 0 to 5 (will take 5 seconds) after the train doors are closed, and then the controller starts to close platform screen doors. Fig. 13d is the simulation result. Compared with Fig. 13c, the time interval in the new model is much larger.

However, we cannot conclude that the time interval is at least 5 seconds definitely, because we cannot simulate every operation of train doors and controller. We can solve this (i.e., verify time interval property) by formal verification tools like PHAVer in the next subsection.

8.2 Formal Verification Correction

For the controller automaton (see Fig. 7), the clock named t will be placed in the control mode $about\_to\_close_2$. When the controller switches from $about\_to\_close_2$ to *start*, we keep the value of the clock t unchanged, and then we can check this value in the mode *start*. The right time to start this clock is the moment when the train doors are closed. Therefore, we add a self-loop control switch of the mode $about\_to\_close_2$. The jump condition is $(z' = 0 \wedge c_1' = 0 \wedge t' = 0)$, and the synchronization label of the control switch is $shut\_closed_1$ which is the same as the label of the control switch from mode *shut* to *closed* in the train doors automaton (see Fig. 5). Thus the two control switches will be taken simultaneously.

Similarly, we are sure that, for the control of platform screen doors, a self-loop control switch of mode *start* is added with the synchronization label $shut\_closed_2$.

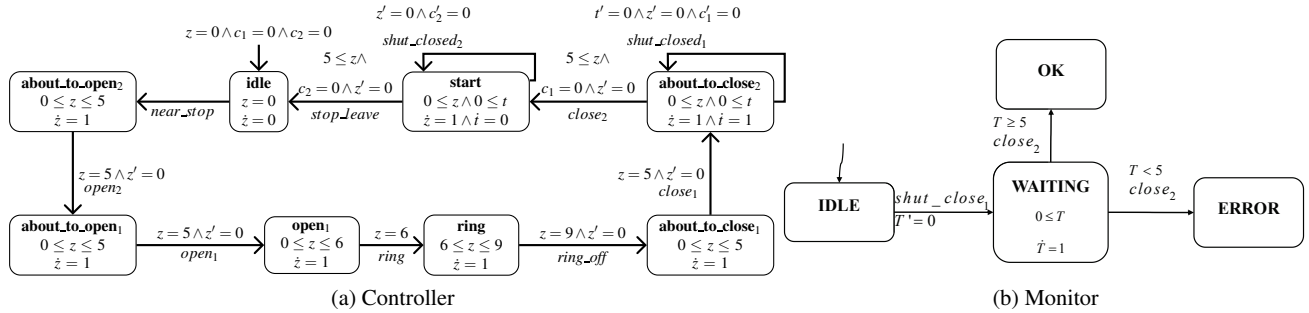
Fig. 15a represents the new controller automaton. And, we integrate the error state into the monitor automaton in Fig. 15b. At this time, we need to verify the time interval property. Consider the states denoted by $sys.\{start \sim \$ \sim \$ \sim \$ \& t < 5\}$. The above states should not be reachable from initial state. This means that the time interval would not be smaller than 5. On the contrary, the following states must be reachable:

```
sys.{ start ~ $ ~ $ ~ $ & t >= 5 }.
```

And, the verification results by PHAVer show that this time interval property is satisfied. And the following states are not reachable:

```
sys = controller & traindoor & screendoor & train & monitor.
sys.{ $ ~ $ ~ $ ~ $ ~ ERROR & True }.
```

The following states given by PHAVer illustrate that, during the mode *start* (the controller prepares to start the train), the train doors are closed, and the screen doors may be in mode *open*, *shut*, *part* and *closed*.



(a) Controller (b) Monitor

Fig. 15 Correct controller automaton and the monitor, the synchronized label $shut\_closed_1$ and the corresponding self-loop transition added on mode $about\_to\_close_2$ are for the goal of time interval property

```
1  controller ~ traindoor ~ screendoor ~ train.{
2      start ~ closed ~ shut ~ stop &
3      -1 <= dy2 <= 0 & 0 <= y2 <= 2 &
4      x == 0 & y1 == 0 &
5      close2_flag == 1 & close1_flag == 0
6      & t >= 5 &  z + y2 >= 2,
7
8      start ~ closed ~ open ~ stop &
9      x == 0 & y2 == 2 & y1 == 0 &
10     close2_flag == 1 & close1_flag == 0 &
11     t >= 5 & z>=0,
12
13     start ~ closed ~ closed ~ stop &
14     x == 0 & y2 == 0 & y1 == 0 &
15     close2_flag == 0 & close1_flag == 0 &
16     t >= 5  & z>=0,
17
18     start ~ closed ~ part ~ stop &
19     0 < y2 <= 2 & x == 0 & y1 == 0 &
20     close2_flag == 1 & close1_flag == 0 &
21     t >= 5 & z + y2 >= 2
22 }.
```

According to the results, the predicate $t \geq 5$ is always true. Which indicates that the time interval property is satisfied.

9 Collision Avoidance in Subway Control Systems

In this section, we improve our models as a subway control system that consists of four trains, four stations, and the platform of each station contains two sides (i.e., platforms), the Side-A and Side-B. The train would stop at Side-A when its direction is Direction-A, and it would stop at Side-B when the direction is Direction-B, as illustrated in Fig. 16.

The first station is Station-1, and the last station is Station-4. At first, the train stops at the Side-A of Station-1, then it leaves for Station-2 and then Station-3, Station-4. When the train departs from Side-A of Station-4, it will change its direction from A to B, and stop at the Side-B of Station-4. Also, the train changes the direction from B to A when it leaves from Side-B of Station-1.

Along the subway line, we use constant numbers to specify the positions of stations. Station-1 is positioned at 0, i.e., it is the reference point among the four stations in the system. Station-2 is positioned at 2284, therefore, the distance between Station-1 and Station-2 is 2284 (meters).

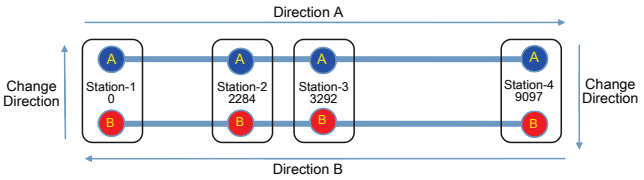


Fig. 16 The subway (metro) line, train running path and stations, Side-A (i.e., platform A) is depicted as a solid blue circle, and Side-B (i.e., platform B) the solid red circle

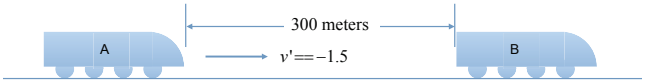


Fig. 17 The urgent distance control for trains

Moreover, each train has one group of train doors described in previous sections. Each side of station has one group of screen doors. As a result, we have eight groups of screen doors in the system.

9.1 Requirement

In this case, we focus on the safety distance between trains. Our objective is to make sure that the train collision is absent in our system. To keep a safety distance, in this case study, we monitor the distance between two trains, the train shall brake when its distance to the train ahead is less than or equal to 300 meters (urgent distance, as illustrated in Fig. 17). The deceleration will be $-1.5m/s^2$ during the braking.

Let Ω be the safety distance between trains, and Φ be the urgent distance, Υ the set of trains. $\forall i, j, i \neq j$, and $T_i, T_j \in \Upsilon$, the requirement is:

$$[D(T_i, T_j) \leq \Phi \implies U(T_i)] \implies [D(T_i, T_j) \geq \Omega]$$

where, D is a function that returns the distance between T_i and T_j if the train T_j is ahead of T_i. $U(T_i)$ is a predicate that checks T_i is in the urgent control state (i.e., breaking or stop). In other words, whenever the distance between two trains is

Table 1 Variables in the Simulink/Stateflow model. The *Output* variable is used to display the simulation result. The *Input* variable is used to receive signal (or just a value) from outside of the Stateflow model. The variable of type *double* can be changed continuously, also double variables can be used to denote floating numbers. The variable of type *int32* is the discrete variable. The type *Inherit* means the type is inherited from the Simulink component that produces the input to Stateflow chart. Local variable and global variable are relative concepts. For example, if Stateflow chart A has one local variable x and four subcharts, then x is a shared variable between the four subcharts. But, if one of the subcharts has a local variable y, then y cannot be read or write by other subcharts

Variable	Scope	Type	Size	Initial Value
v_shutTD	Input	Inherit	[4 1]	
v_shutPSD	Input	Inherit	[4 2]	
openPSD	Local	double	[4 2]	
closePSD	Local	double	[4 2]	
Delay	Local	double	[1 4]	[0 50 100 150]
openTD	Local	double	[4 1]	
closeTD	Local	double	[4 1]	
S	Local	double	[1 4]	[0 2284 3292 9097]
N	Local	int32		4
urstopped	Local	double	[4 1]	
dir	Local	int32	[4 1]	
position	Local	double	[4 1]	
distance_out	Output	double	[4 1]	
y1_out	Output	double	[4 1]	
y2_out	Output	double	[4 2]	

less than or equal to the urgent distance, the train behind is under the urgent control. Then these two trains are safe from collision.

9.2 Simulation of Collision Avoidance Scenario

Since the strong expressive power and fast simulation result feedback of Simulink/Stateflow, we do the simulation before proceeding to the formal verification. Stateflow supports the declaration and invocation of Matlab functions in which we can employ plentiful control structures such as conditional and iterative statements. And, in the model of Simulink/Stateflow, array variables are supported, as a result, we can read and write variables according to an index (Table 1 illustrates some variables and their scopes, types, dimensions at the highest level of our model). This is very convenient for modeling, for instance, in our model, we have four trains, each train has the same chart structure and same variables definitions, it is easy to get and set the their values with an index variable in the control structures. Moreover, the transition label in Stateflow allows non-linear predicates (for instance, we can move the exit action '$maxv = sqrt(0.5 * abs(S[j] - distance))$' in $urgent\_stop$ to the transition from $urgent\_stop$ to $select$ in Fig. 18) and math functions in the condition (i.e., the guard) and action (i.e., the assignment). This expressive power gives the designer a great freedom of choice during the modeling.

In Fig. 18, we improve the train chart with more realistic dynamics and the urgent control reactions. The trains are not allowed to start from the first station simultaneously as we only have one track at each side of one station. As a result, each train has to wait for some time at the beginning. The work flow of one train is depicted as follows: at first, it waits for several time and then stops at the first station. After it picks passengers, the train leaves the first station and goes to Station-2. When the train stops at the final station, all passengers get off the train, then it changes its running direction from A to B, and stops at the Side-B of the final station. The train picks up the passengers at the Side-B of the final station, then it leaves. Similarly, the train also changes the direction when it leaves from the Side-B of the first station.

As illustrated in Fig. 18, the train would be in the stable running state when its velocity reaches 20 (m/s). And, the train would slow down if the distance to the station ahead is less than or equal to 500 meters. During the running, each train will interact with the urgent controller (in Fig. 19). The shared variable $urstop[i]$ indicates whether or not to do the urgent stop. And, $urstart[i]$ represents the restart signal of a train during the urgent control. We put the control logic in the Matlab function $stopDecision$ to decide whether or not to stop a train urgently. In the Matlab function $CalDistance$, the controller calculates the distances between trains and restarts the train if it is safe to start running again.

The simulation results are showed in Fig. 20-21. Fig. 20 illustrates the normal runnings of fours trains, from left to right, the curves represent the positions of Train-0, Train-1, Train-3 and Train-4 over time, respectively. Obviously the urgent distance control was taken (for Train-1, Train-2 and Train-3) before they passed the final station (i.e., the position 9097). And, we can see that Train-0 never goes into the urgent control state as it is the first train that begins to start running in the system, i.e., no train is ahead of Train-0. For Train-1, it has to stop urgently near the second station (i.e., position 2284), because Train-0 is stopped at the second station, and the distance between these two trains is less than 300 meters. Likewise, Train-2 and Train-3 also must urgently stop when the previous trains are stopped urgently. All trains have to change its direction at the final station, and it takes more time for the train during the final station, Train-1 has to stop urgently when Train-0 is at Side-A of the final station. After the final station, there is no urgent control for the four trains as the distances between them are safe enough for their future running. In Fig. 21, the difference for the positions of Train-0 and Train-1 is depicted in the second sub-figure, and the third sub-figure denotes the direction of Train-0, the number 1 for Direction-A, -1 for Direction-B. After Train-0 changes its direction, the position difference

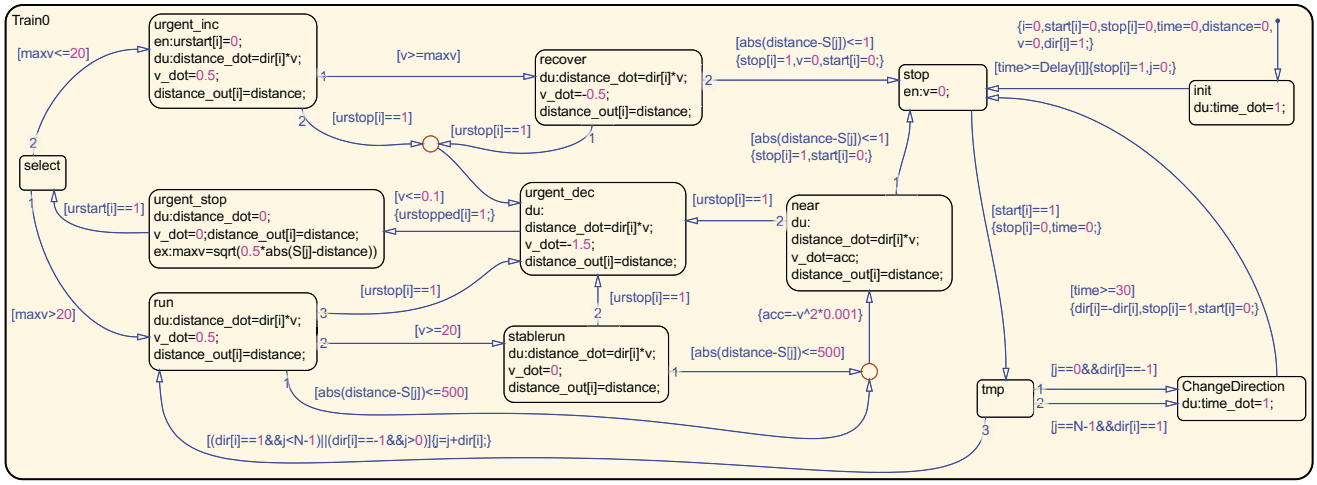


Fig. 18 The improved train chart in Stateflow. The variable j is used for station index, it ranges over $\{0,1,2,3\}$. N is a constant variable that represents the amount of stations, in our case, $N = 4$. The variable i denotes the id of trains, in this statechart, $i = 0$. For other trains, i maybe 1, 2 or 3. The direction of Train-i is represented by the variable $dir[i]$, thus dir is an array. Similarly, the position of station is $S[j]$, for instance, $S[0] = 0$. Initially, before start running, the train has to wait for a period of time, $Delay[i]$ is used for Train-i that after the delay of $Delay[i]$ (seconds) the train can stop at the first station. When the train restarts from urgent stop, we calculate the maximum velocity ($maxv$) of the train according to the distance from the train to the front station and the acceleration ($0.5\ m/s^2$). If $maxv$ is great than 20 m/s, the train will do the normal running (i.e., go to the state run). Otherwise, it will go to the state $urgent\_inc$. The distance between the train and the first station is represented by the variable $distance$. The velocity of the train is denoted by variable v. In addition, the symbol '$*$' used in Stateflow denotes the multiplication operator

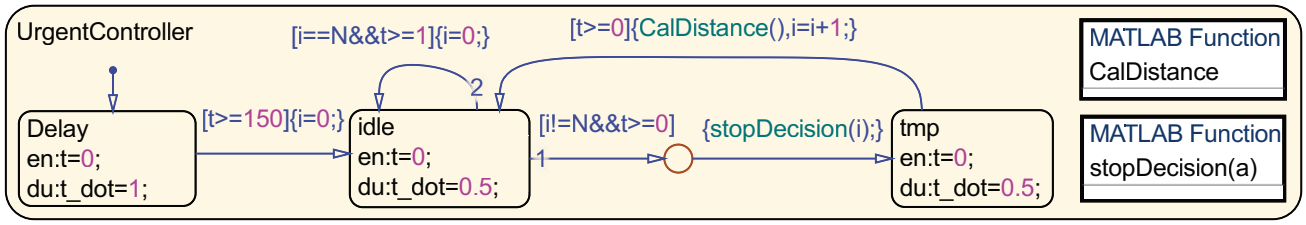


Fig. 19 Urgent controller chart in Stateflow. Variable i is used for the id of trains, in our case, i ranges over $\{0, 1, 2, 3\}$. The amount of trains is represented by variable N, and $N = 4$. The matlab function $stopDecision(a)$, decides whether the train with id a has to stop urgently, for example, it sets the value of $urstop[1]$ to be 1 if Train-1 has to stop urgently. Another function, $CalDistance$ recalculates the distances between trains, and sets the train to restart (urstart[i]=1, the id of this train is 1) if its distance to other trains is a safe one

will be zero, it is safe for these two trains as they are positioned at the different sides of the station or subway line.

9.3 Formal Verification of Collision Avoidance Scenario

For formal verification, we employ SpaceEx ([21]) to model and verify the hybrid automata model. It supports component-based modeling and verification on affine hybrid automata. It is proper to use this tool to handle large-scale hybrid systems with complex dynamics. For more details about the tool SpaceEx, one can refer to the website of the tool [2].

The simulation result in Fig. 20 indicates that the trains have strict running orders. This feedback from simulation can be beneficial to simplify our modeling and verification process with hybrid automata. Now, modeling of two trains in SpaceEx is sufficient to verify the urgent collision avoidance scenario. The train automaton and the urgent controller automaton are illustrated in Fig. 23-24 (see Appendix A). We find one bug that is not detected during the design and simulation (the reachable states are listed as in Fig. 22).

As described in the verification result, the two trains (the trains in the components $trainsys\_1$ and $trainsys\_2$) are positioned at the first station at the same time while the screen doors at the station are closing. $pos1$ and $pos2$ are the positions for these two trains. '$dir1 == 1$' represents that the direction of the first train is in Direction-A, and '$dir2 == 1$' means the second train is also in Direction-A. In Sect. 7, we know that the time for closing the train doors and screen doors may be very long as during the closing we have to reopen and reclose the doors, because of the clamping situation. Therefore, this bug means that the delay of a period of time before the train goes to the first station is not a proper approach to schedule all the trains in our case.

The bug seems very simple, one may believe that people can recognize this at the beginning of the design, and if we monitor the first station carefully, we can remove this

[2] http://spaceex.imag.fr/

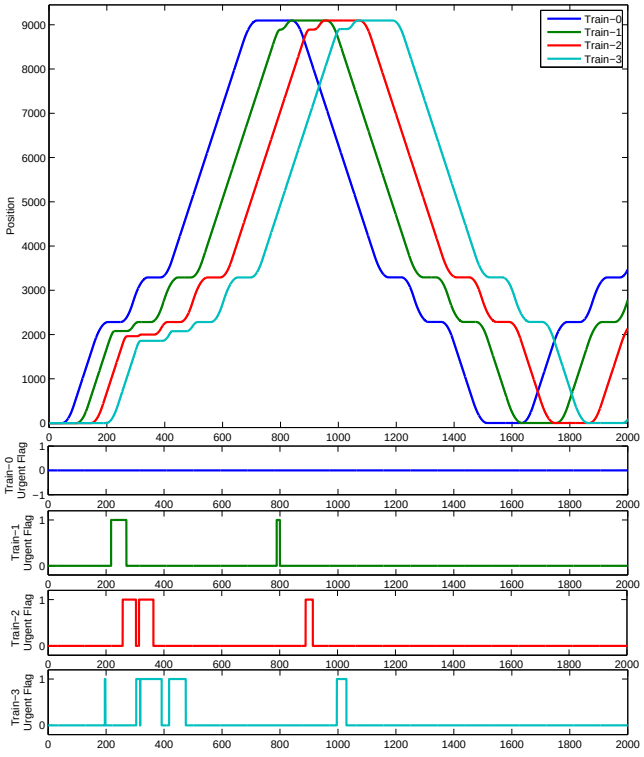


Fig. 20 One normal runnings of four trains for 2000 seconds. The first sub-figure denotes the positions of four trains over time. In addition, four sub-figures represent the urgent control flags for these trains, respectively. If the flag takes value of 1, the train is in urgent control, and 0 otherwise

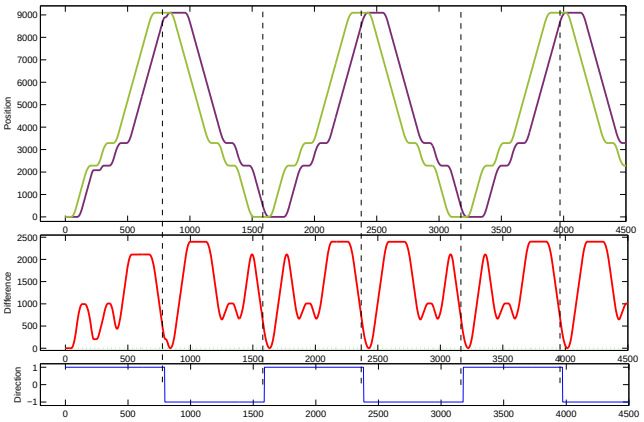


Fig. 21 The distance between Train-0 and Train-1

bug easily. But, a small error may lead to a large discrepancy later on. We treat this "simple" bug as serious as we can. After carefully analyzing for this, we found that the urgent controller also has a similar bug when we observes the final station. In the design of urgent controller, we believe that two trains positioned at the same station with different directions is safe because one train stops at the Side-A and another stops at the Side-B of the station. But, if the train changes its direction from A to B at the final station, then it is very dangerous when there is another train which already stopped at the Side-B of the station. We believe that the latter bug is not as "simple" as the former one. This is one reason why we employ the feedback between simulation and formal verification.

```
1  loc(trainsys_1.train)==stopAtStation & loc(trainsys_1.train)!=init &
2  loc(trainsys_1.traindoors)==closed &
3  loc(trainsys_1.controller)==closeScreenDoors &
4  loc(trainsys_2.train)==stopAtStation & loc(trainsys_2.train)!=init &
5  loc(trainsys_2.traindoors)==closed &
6  loc(trainsys_2.controller)==about_to_open2 &
7  loc(urgentcontrol)==Delay & loc(screendoors_1)==shut &
8  loc(screendoors_2)==closed & loc(screendoors_3)==closed &
9  loc(screendoors_4)==closed & loc(screendoors_5)==closed &
10 loc(screendoors_6)==closed & loc(screendoors_7)==closed &
11 loc(screendoors_8)==closed & screendoors_8.y == 0 &
12 screendoors_7.y == 0 & screendoors_6.y == 0 &
13 screendoors_5.y == 0 & screendoors_4.y == 0 &
14 screendoors_3.y == 0 & screendoors_2.y == 0 &
15 trainsys_1.controller.z+screendoors_1.y == 2 &
16 trainsys_2.controller.z-globalTime == -50 &
17 distance == -1 & urstopped_2 == 0 & urstopped_1 == 0 &
18 trainsys_2.controller.z-t == -50 &
19 trainsys_1.traindoors.y == 0 & trainsys_1.train.S == 0 &
20 trainsys_1.train.time == 0 &  dir1 == 1 &
21 trainsys_1.train.v == 0 & pos1 == 0 &
22 trainsys_1.stationid == 0 &
23 trainsys_2.traindoors.y == 0 & trainsys_2.train.S == 0 &
24 trainsys_2.train.time == 0 & dir2 == 1 &
25 trainsys_2.train.v == 0 &
26 pos2 == 0 & trainsys_2.stationid == 0 &
27 trainsys_2.controller.z-trainsys_1.controller.z <= -1 &
28 trainsys_1.controller.z <= 2 & trainsys_2.controller.z >= 0
```

Fig. 22 The verification result in SpaceEx

10 Related Work

The platform screen doors system is a subsystem of subway control system. The literature [36] reported the work about COPPILOT (PSD controller, [14]) which had been done at ClearSy [13] in France. It applied the B-method ([3]) to the development of platform screen doors systems. More details about B-method can be found at [1], and the 15-year application in industry [37]. The tools and applications of B and Event-B ([2, 47]), for example, the application on metro can be found at the website [15].

The work [36] is an industrial project. And the product Coppilot has been installed in the Paris metro, at several platforms: station Invalides and Saint-Lazare Line 13. The architecture of the work was based on Siemens safety automaton, and ordinary infra-red and radar sensors. It is different from our work, the motivation of our work is to combine the formal verification with simulation and take a try to apply this procedure to the real systems, in addition, our models discussed here are an abstract of the real systems. On the contrary, [36] also contains the process of the code translation from B to LADDER (for the Siemens automaton can be programmed in LADDER).

Another work ([45]) on platform screen doors considered the emergency evacuation in underground railway sta-

tions, studied the evacuation times for variants of passenger loadings on the subway system in Hong Kong.

The characteristic of [45] is the empirical equations and the statistical data for the estimation. Another interesting point is that it also considered the positions of the train doors and platform screen doors when the train stopped. It is not studied in our work as we focus on the time interval property in this paper, and we believe it shall be done in the future work based on formal methods.

The Metrô Rio case study presented in [17] adopted the Matlab Simulink/Stateflow for the development of ATP (Automatic Train Protection) system. It also took the formal verification using Simulink Design Verifier. For the generated code, the tool Polyspace was applied to check the correctness of the code.

The main distinction between our work and the Metrô Rio case study is that it did not consider the connection between formal verification and simulation during development. On the contrary, in our procedure (FAV), there is a feedback relation between formal verification and simulation, this relation is key for the advancement of design and development process.

The ARTEMIS EU-project MBAT [40] studied in [39] elaborated the improving of verification process in driverless metro Systems. It combined model-based static analysis and dynamic testing in the development of the MBAT project in Ansaldo STS. In the CBTC (Communication-Based Train Control) case study, they also studied the "Openning" of the train doors and platform screen doors, but focused on which side of the doors should open at first (or second). This is different from our work demonstrated here, we focus on the "Closing" of the doors.

The common point is that, the proposed Ansaldo STS process workflow in [39] also concerned the feedback between formal verification and the non-formal analysis (e.g., the test).

On the life safety aspect of platform screen doors, the literature [46] investigated the fire and evacuation scenario in a subway train fire. The fire simulation in the work adopted the Fire Dynamics Simulator (FDS V406 [41]) code to predict the smoke spread and analyze the safe time. And, it pointed out that the installation of platform screen doors in the subway station has better advantage for the safety of passengers than that lacks of platform screen doors.

The work [48] studied the requirement analysis for train control systems. It translated the requirement into PSL ([4]) model, and checked the model with the requirement analysis tool RATSY ([11]). Another work on train control systems was proposed in [33], which employed Z ([32]) and Statechart ([27, 28]) for the formal specification and verification. Compared with our work, the above two papers focus on the abstract level of the system and did not consider the continuous behavior of the train in their work. Our work not only considered the discrete transitions between control modes, but also the continuous flow of the system states.

On the timed UML aspect, [43] proposed a method and a tool based on the communicating extended timed automata for simulation and model checking on the UML models. It integrated the real-time feature as UML extensions, and considered timers, clocks, time-related data types, etc. Due to the lack of the description on the continuous behavior (e.g., affine dynamics), it is not applicable for hybrid systems involving linear and non-linear dynamics.

11 Conclusions and Future Work

In this paper, we have presented the models of Platform Screen Doors System (PSDS) which is a subsystem of subway control systems. And, we improved the models with four trains, four stations on a subway line. The formal verification approaches and simulation based techniques have been applied and elaborated in detail. Moreover, we proposed the procedure to integrate the formal verification with industrial simulation. For the verification of PSDS, we combined the formal verification tool SpaceEx/PHAVer with simulation tool Simulink/Stateflow. Benefited from the interactive simulation and the powerful visualization features of Matlab toolset, we have simulated the sandwiched situation by using Simulink blocks in the model, and explored the reaction of the system in the Scope window with respect to simulation time.

To verify the correctness of the correction, we checked the time interval property by the tool SpaceEx/PHAVer. In base phase, we focused on verification and simulation for four fundamental properties, and found the sandwich problem with simulation. Then in the exploration phase, we modified our controller's model and took simulation again, then modified the hybrid automaton of controller and checked the correctness by formal verification with SpaceEx/PHAVer. As a result, we acquired the ultimate models of PSDS for our case. For the collision avoidance scenario, we showed that the simulation result is beneficial for the formal verification. And, we can inspect unpredictable flaws during the verification, which can be feedback to simulation as well.

As future work, we plan to continue the verification on subway control systems. Moreover, the potential work could be to extend the Matlab Simulink/Stateflow toolset with the capability of formal verification. In the literature [26], Hamon and Rushby presented an operational semantics for the Matlab Stateflow. And in [25], Hamon presented a denotational semantics for Stateflow. Although these works are subject to a subset of Stateflow, we could benefit from them to analyze the new features in Stateflow and construct formal tools to improve the verification progress.

Furthermore, in [5], Agrawal, Simon and Karsai presented the algorithm of translation from Simulink/Stateflow

models to hybrid system models in Hybrid System Interchange Format (HSIF [44]). This foundational work is helpful on the formal verification for Simulink/Stateflow models and could be implemented in the Matlab toolset, and it can be adopted in our future work. In addition, the scalability of the integrated notation and method is worth to be discussed in the future work.

Acknowledgements We thank Goran Frehse for his insightful discussion on SpaceEx/PHAVer and hybrid systems. This work was partly supported by the Danish National Research Foundation and the National Natural Science Foundation of China (Grant No. 61361136002) for the Danish-Chinese Center for Cyber Physical Systems. And, also it was supported by National High Technology Research and Development Program of China (No. 2012AA011205), National Natural Science Foundation of China (No. 61321064 and No. 91118008), Shanghai STCSM Project (No. 12511504205), Shanghai Knowledge Service Platform Project (No. ZF1213) and Shanghai Minhang Talent Project.

References

1. Abrial, J.-R.: The B-book: assigning programs to meanings. Cambridge University Press, Cambridge (2005)
2. Abrial, J.-R.: Modeling in Event-B: system and software engineering. Cambridge University Press (2010)
3. Abrial, J.-R., Lee, M., Neilson, D., Scharbach, P., Sørensen, I.: The b-method. In: Proceedings of VDM, *LNCS*, vol. 552, pp. 398–405. Springer-Verlag (1991)
4. Accellera Orgnization: Property specification language reference manual (2003)
5. Agrawal, A., Simon, G., Karsai, G.: Semantic translation of simulink/stateflow models to hybrid automata using graph transformations. Electron. Notes. Theor. Comput. Sci. **109**, 43–56 (2004)
6. Alur, R., Courcoubetis, C., Henzinger, T., Ho, P.: Hybrid automata: An algorithmic approach to the specification and analysis of hybrid systems. In: Hybrid Systems, *LNCS*, vol. 736, pp. 209–229. Springer-Verlag (1993)
7. Alur, R., Henzinger, T., Ho, P.: Automatic symbolic verification of embedded systems. IEEE Trans. Softw. Eng. **22**(3), 181–201 (1996)
8. Asarin, E., Bournez, O., Dang, T., Maler, O.: Approximate reachability analysis of piecewise-linear dynamical systems. In: Proceedings of HSCC, *LNCS*, vol. 1790, pp. 20–31. Springer-Verlag (2000)
9. Asarin, E., Dang, T., Maler, O., Testylier, R.: Using redundant constraints for refinement. In: Proceedings of ATVA, *LNCS*, vol. 6252, pp. 37–51. Springer-Verlag (2010)
10. Bagnara, R., Ricci, E., Zaffanella, E., Hill, P.: Possibly not closed convex polyhedra and the parma polyhedra library. In: Proceedings of SAS, *LNCS*, vol. 2477, pp. 299–315. Springer-Verlag (2002)
11. Bloem, R., Cimatti, A., Greimel, K., Hofferek, G., Könighofer, R., Roveri, M., Schuppan, V., Seeber, R.: Ratsy–a new requirements analysis tool with synthesis. In: Proceedings of CAV, pp. 425–429. Springer-Verlag (2010)
12. Bonnett, C.: Practical Railway Engineering. Imperial College Press (2005)
13. ClearSy: ClearSy–A French SME company. URL http://www.clearsy.com/?lang=en
14. ClearSy: COPPILOT System. URL http://www.coppilot.fr/en/coppilot/
15. ClearSy: Tools and applications at ClearSy. URL http://www.tools.clearsy.com
16. Doyen, L., Henzinger, T., Raskin, J.: Automatic rectangular refinement of affine hybrid systems. In: Proceedings of FORMATS, *LNCS*, vol. 3829, pp. 144–161. Springer-Verlag (2005)
17. Ferrari, A., Fantechi, A., Magnani, G., Grasso, D., Tempestini, M.: The metrô rio case study. Sci. Comput. Program. (2012)
18. Frehse, G.: PHAVer: Algorithmic verification of hybrid systems past HyTech. In: Proceedings of HSCC, *LNCS*, vol. 3414, pp. 258–273. Springer-Verlag (2005)
19. Frehse, G.: Language Overview for PHAVer version 0.35 (2006)
20. Frehse, G.: PHAVer: algorithmic verification of hybrid systems past HyTech. Int. J. Softw. Tools. Technol. Transf. **10**(3), 263–279 (2008)
21. Frehse, G., Le Guernic, C., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: SpaceEx: Scalable verification of hybrid systems. In: Proceedings of CAV, *LNCS*, vol. 6806, pp. 379–395. Springer-Verlag (2011)
22. Girard, A., Le Guernic, C.: Zonotope/hyperplane intersection for hybrid systems reachability analysis. In: Proceedings of HSCC, *LNCS*, vol. 4981, pp. 215–228. Springer-Verlag (2008)
23. Granlund, T., Ryde, K.: The GNU Multiple Precision Arithmetic Library Version 4.0 (2001)
24. Halbwachs, N., Proy, Y., Raymond, P.: Verification of linear hybrid systems by means of convex approximations. In: Proceedings of SAS, *LNCS*, vol. 864, pp. 223–237. Springer-Verlag (1994)
25. Hamon, G.: A denotational semantics for stateflow. In: Proceedings of EMSOFT, pp. 164–172. ACM (2005)
26. Hamon, G., Rushby, J.: An operational semantics for stateflow. Int. J. Softw. Tools. Technol. Transf. **9**(5-6), 447–456 (2007)
27. Harel, D.: Statecharts: A visual formalism for complex systems. Sci. Comput. Program. **8**(3), 231–274 (1987)
28. Harel, D., Naamad, A.: The statemate semantics of statecharts. ACM Trans. Softw. Eng. Methodol. **5**(4), 293–333 (1996)

29. Henzinger, T., Ho, P., Wong-Toi, H.: Hytech: A model checker for hybrid systems. Int. J. Softw. Tools. Technol. Transf. **1**(1-2), 110–122 (1997)
30. Henzinger, T., Kopke, P., Puri, A., Varaiya, P.: What's decidable about hybrid automata? J. Comput. Syst. Sci. **57**(1), 94–124 (1998)
31. Henzinger, T.A.: The theory of hybrid automata. In: Proceedings of LICS, pp. 278–292. IEEE Computer Society (1996)
32. Jacky, J.: The way of Z: practical programming with formal methods. Cambridge University Press (1996)
33. Jo H.-J, Hwang J.-G, Yong Y.-K : Development of formal method application for ensuring safety in train control system (2008). URL http://www.railway-research.org/IMG/pdf/o.3.4.2.3.pdf
34. Kurzhanski, A.B., Varaiya, P.: Ellipsoidal techniques for reachability analysis. In: Proceedings of HSCC, *LNCS*, vol. 1790, pp. 202–214. Springer-Verlag (2000)
35. Le Guernic, C., Girard, A.: Reachability analysis of linear systems using support functions. Nonlinear Anal. Hybrid Syst. **4**(2), 250–262 (2010)
36. Lecomte, T.: Safe and reliable metro platform screen doors control/command systems. In: Proceedings of FM, *LNCS*, vol. 5014, pp. 430–434. Springer-Verlag (2008)
37. Lecomte, T.: Applying a formal method in industry: A 15-year trajectory. In: Proceedings of FMICS, *LNCS*, vol. 5825, pp. 26–34. Springer-Verlag (2009)
38. Lynch, N.A., Vaandrager, F.W.: Forward and backward simulations, ii: Timing-based systems. Inf. Comput. **128**(1), 1–25 (1996)
39. Marrone, S., Nardone, R., Orazzo, A., Petrone, I., Velardi, L.: Improving verification process in driverless metro systems: The mbat project. In: Proceedings of ISoLA, *LNCS*, vol. 7610, pp. 231–245. Springer-Verlag (2012)
40. MBAT Consortium: ARTEMIS Project MBAT. URL http://www.mbat-artemis.eu
41. National Institute of Standards and Technology (NIST): Fire Dynamics Simulator and Smokeview Code. URL http://code.google.com/p/fds-smv/
42. Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: An approach to the description and analysis of hybrid systems. In: Proceedings of Hybrid Systems, *LNCS*, vol. 736, pp. 149–178. Springer-Verlag (1993)
43. Ober, I., Graf, S., Ober, I.: Validating timed uml models by simulation and verification. Int. J. Softw. Tools. Technol. Transf. **8**(2), 128–145 (2006)
44. Pinto, A., Sangiovanni-Vincentelli, A.L., Carloni, L.P., Passerone, R.: Interchange formats for hybrid systems: Review and proposal. In: Proceedings of HSCC, *LNCS*, vol. 3414, pp. 526–541 (2005)
45. Qu, L., Chow, W.: Platform screen doors on emergency evacuation in underground railway stations. Tunn. Undergr. Space Technol. **30**(0), 1–9 (2012)
46. Roh, J.S., Ryou, H.S., Park, W.H., Jang, Y.J.: Cfd simulation and assessment of life safety in a subway train fire. Tunn. Undergr. Space Technol. **24**(4), 447–453 (2009)
47. Su, W., Abrial, J.-R., Zhu, H.: Complementary methodologies for developing hybrid systems with event-b. In: Proceedings of ICFEM, *LNCS*, vol. 7635. Springer-Verlag (2012)
48. Zhao, L., Tang, T., Cheng, R., He, L.: Property based requirements analysis for train control system. J. Comput. Inf. Syst. **9**(3), 915–922 (2013)

A Appendix: Hybrid Automata of Train and Urgent Distance Controller in SpaceEx

Table 2 Variables in Hybird Automata of Train and Urgent Distance Controller

Variable	Meaning
time	Used as a clock to tick time passing in train automaton.
Delay	A constant variable. Used as the initial waiting time for a train, after the delay the train will start to work and stop at the first station.
v	The velocity of a train in our case.
pos	The position of a train in our case.
S	Denotes the position of a subway station. The value of *S* will be reset whenever the train leaves a station.
dir	The running direction of a train. The direction is Direction A (see Fig. 16) when the value of *dir* takes 1, and -1 for Direction B.
j	Represents the identifier of the station that is in the front of the train or the station the train is stopped.
t	Used as a clock to tick time passing in urgent distance controller automaton.
distance	Denotes the distance between two trains.
globalTime	Records the time passing whenever the urgent distance controller is active.
urstopped_1	A flag variable that indicates the behaviors of the first train should be. If its value is 1, then the train has to stop urgently. And, the train can restart to run from urgent stop when the value of *urstopped_1* is 0.
urstopped_2	The same as *urstopped_1*, but *urstopped_2* is used for the second train.

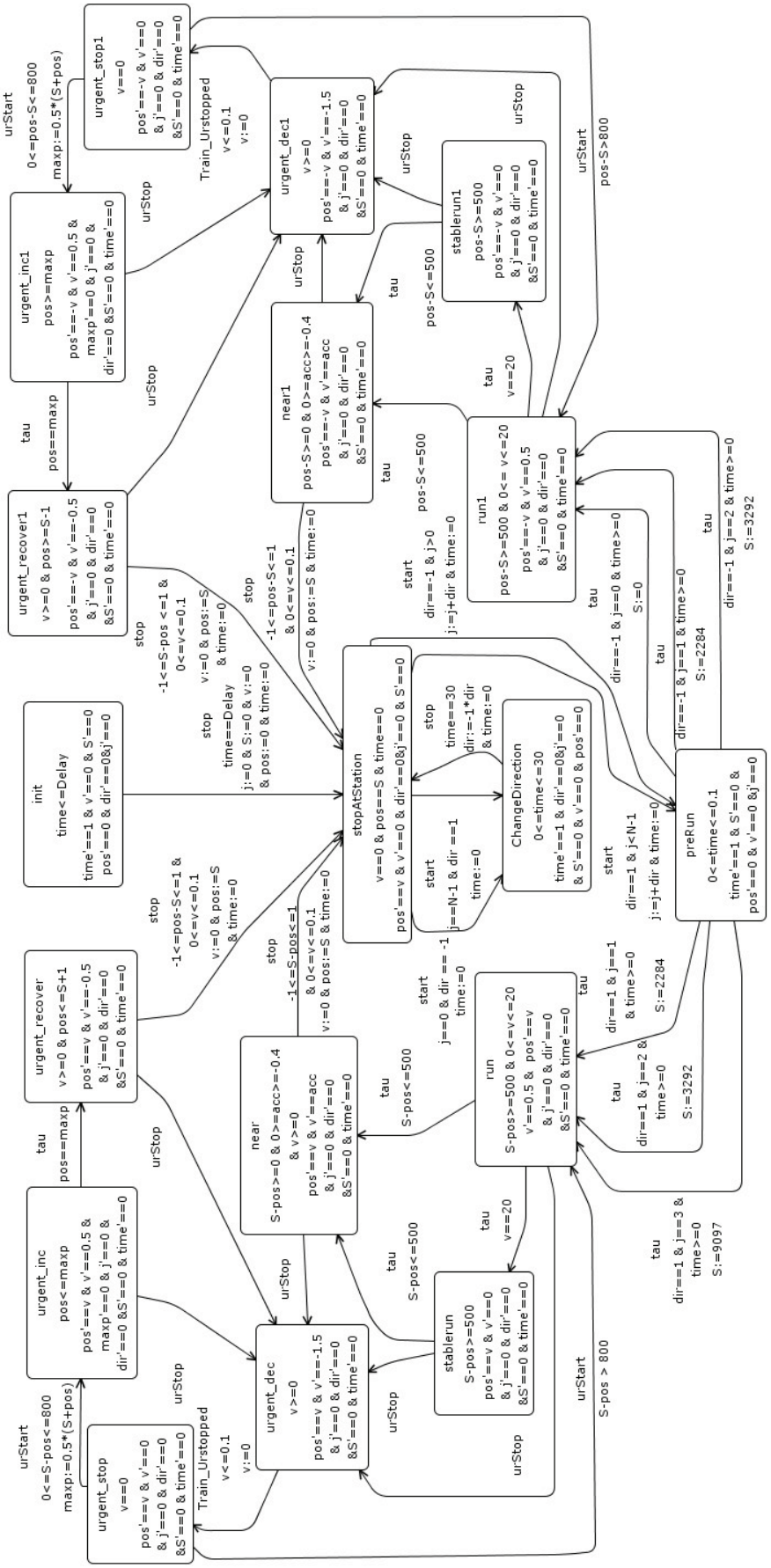


Fig. 23 Train automaton in SpaceEx. The notations of the hybrid automata in SpaceEx are slightly different from that was defined in Definition 1. The primed variable x' in a mode is corresponding to the first-order derivative of x. The symbol '&' is used as a relation operator for logical conjunction. The assignment ':=' in a jump condition is applied for the reset of variables when the control switch is valid. For example, 'j:=0' represents the reset of the value of j to 0. The conditional operators such as '==', '<=' in a jump condition is for the testing of the value of expressions, for instance, '$pos - S <= 500$' is a test of the result for the subtraction of S from pos is less than or equal to 500 meters

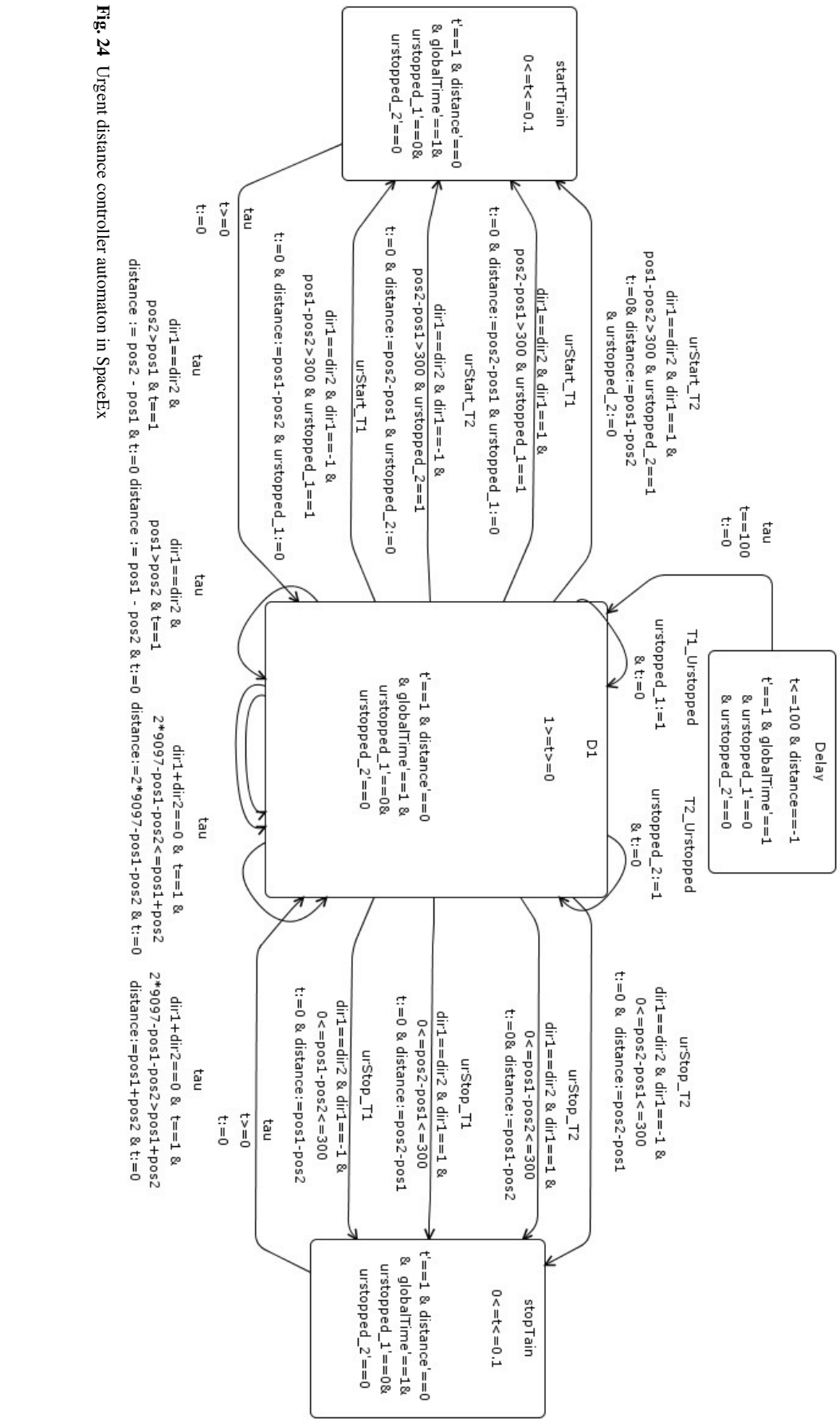


Fig. 24 Urgent distance controller automaton in SpaceEx