Available online at www.ijpe-online.com

International Journal of Performability Engineering

Vol. 13, No. 6, October 2017, pp. 897-912
DOI: 10.23940/ijpe.17.06.p11.897912

Predicting Accidents in Interlocking Systems: An SHA Model-Based Approach

Yan Wang, Wen Zhong, Xiaohong Chen*, Jing Liu

Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China

Abstract

In recent days, rail transit accidents happen from time to time, but the causes are difficult to be found. According to the stochastic and real-time characteristics of equipment faults, three layer models based on stochastic hybrid automata (SHA) are proposed for interlocking systems. The three layer models consist of a system model, a monitoring model and a fault prediction model. The accidents caused by the equipment faults are predicted by simulating these models together on UPPAAL-SMC platform. The main contributions of this paper include: (1) extracting model patterns for interlocking systems (2) presenting a pattern-based system model generation process and an automatic generation method of monitoring model based on time constraints and (3) defining the accidents prediction model of collision accidents to predict the accidents and monitoring accident causes through model simulation.

Keywords: interlocking systems; model monitoring; UPPAAL-SMC; stochastic hybrid automata (SHA); accident prediction

(Submitted on July 25, 2017; Revised on August 30, 2017; Accepted on September 15, 2017)

(This paper was presented at the Third International Symposium on System and Software Reliability.)

1. Introduction

Rail transit systems are safety critical systems. Its safety is of great importance. However, in recent years, China's railway accidents happen frequently. For example, July 23, 2011 Yong Wen line "7.23" major railway traffic accident. This is a serious accident caused by the serious design flaws of the equipment in the control center, inadequate checks on the road, and ineffective emergency response after equipment failure caused by lightning. The accident led to two trains rear-ends collision, and caused 40 deaths, 172 injuries, and the direct economic losses of 19371.65 million [1].

There are many reasons for various accidents. The reason may be a failure caused by one equipment fault, or a failure caused by multiple faults, or multiple failures occurred at the same time. So it is a hot topic to find the causes of the accident from the already happened accidents.

There are many ways to analyze the causes of accidents, which are mainly divided into analytical methods and model checking based methods. The analytical methods mainly include reliability block diagram method [2,6], failure mode consequence analysis method FMEA [5], fault tree analysis method FTA [8,19] and so on. The model checking based approaches mainly use counterexamples in model checking [14,15].

However, of these two methods, the analytical method belongs to the static analysis without paying attention to time constraint; it is not applicable to the dynamic fault analysis and fault prediction of the system. And model checking approaches are confronted with state explosion problem. In the railway transit transportation field, the equipment fault is an important cause of accidents. For example, the communication between the circuit and the train control center caused by the lightning in Yong Wen line is out of order, and the signal of the track section signal is abnormal. The occurrence of these

* Corresponding Author.
E-mail address: xhchen@sei.ecnu.edu.cn.

faults is unpredictable, with a strong stochastic. In addition, due to the strong real-time characteristics of the rail transit system, the time constraints must be taken into account when modeling.

SHA [3] is an important model of modeling stochastic and time, and has been widely used in various fields such as electromechanical systems, computer simulation, automata and so on [2]. Therefore, this paper proposes to model the equipment faults of the rail transit system using SHA, and construct three layer models including rail transit system model, the monitoring model and the accident prediction model using NSHA. At the model level, the accidents caused by faults are simulated by UPPAAL-SMC [4]. In order to facilitate the construction of rail transit system model, we extract the pattern of the system model for the interlocking system. Using the pattern, you only need to fill in the appropriate parameters to customize the specific system model. In addition, we also give the controller automatic generation algorithm according to its time constraints, and define the prediction model according to the common collisions in the accident of the rail transit system. Through simulation, we predict the possible consequences, analyze its causes and issue appropriate alerts according to different circumstances.

The rest of this paper is organized as follows. Section 2 introduces the notations of SHA, NSHA, and the interlocking systems. Section 3 presents how to extract system model patterns. Section 4 describes how to get the three models. Section 5 gives an example of how to construct models and make predictions. After the introduction of related works in Section 6, Section 7 concludes the paper.

2. Preliminaries

2.1. Introduction to SHA and NSHA

SHA is a timed automata whose clock rates can be different at different locations (i.e., states). Figure 1 shows a graphical representation of an NSHA consisting of two SHA, i.e., A and B. The SHA A has four states A_0, A_1, A_2 and A_3, and has a clock variable *x*. In A_0, x'==0 means that the value of *x* has no change. The label "1" near location A0 indicates that the delay of SHA A at location A0 follows an exponential distribution with λ=1. SHA supports the nondeterministic execution. For example, when SHA A exits the location A0, it has two choices of the target location, i.e., A_1 and A_2.

SHA uses the dashed arrow line labeled with the weight information to denote the possibility of the entering a target location. For example, the probability of going to location$A_1$from A_0 is 3/(3+2), while the probability of going to location A_2 from A_0 is 2/(3+2). To conduct synchronization between SHA in an NSHA, SHA communicates with each other via broadcast channels and shared variables. As an example shown in Figure 1, the two SHA communicate with each other using the channel a. After the synchronization using the send and receive operation [9], both SHA will go to their next locations (i.e, A_3 and B_1 in this example) simultaneously.

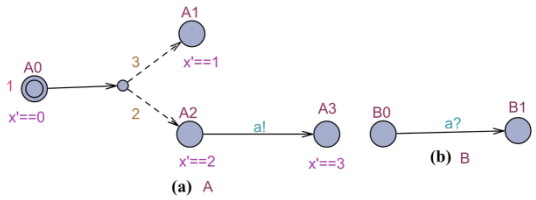


Figure 1. An NSHA, (A||B)

2.2. Interlocking systems

The purpose of railway interlocking system is to control points and signal lights to prevent trains from collisions and derailments [11], while at the same time, allowing its movement. To be more specific, the functions of railway interlocking system are monitoring the occupancy status of the individual track section by track circuits, controlling the positions of points, and sending signals to inform drivers whether they are allowed to enter the route or not. The processing flow of the interlocking system is in https://github.com/wymgal/IS.git. Generally, the physical domain of an interlocking system consists of the following entities:

Tracks: The railway tracks are divided into sections, and each section is associated with a track circuit for detecting whether it is occupied by a train or not.

Points: Track sections are joined by points that can guide trains into different directions depending on the positions of the points. A point can be in position normal or reverse, as well as unlocked to show that the tracks are unconnected at the crossing. A train can pass by a point that has been locked and the point will be occupied.

Signal Lights: Signal Lights are placed between track sections. They can be in color red or green to indicate proceed or stop respectively.

Routes: Routes are defined by interlocking tables, which are created when a railway yard is designed. Each route consists of sequentially-connected track sections according to layout topology. Only if the route has been established, a train will be authorized to enter the route.

Figure 2 displays the track layout diagram of a sample railway yard. The track layout diagram outlines the geographical arrangement of the tracks and track-side equipment. From the diagram, we can see that the station has two switches (named SW1 and SW2), seven signals (named S1-7) and six track sections (named T1-6).

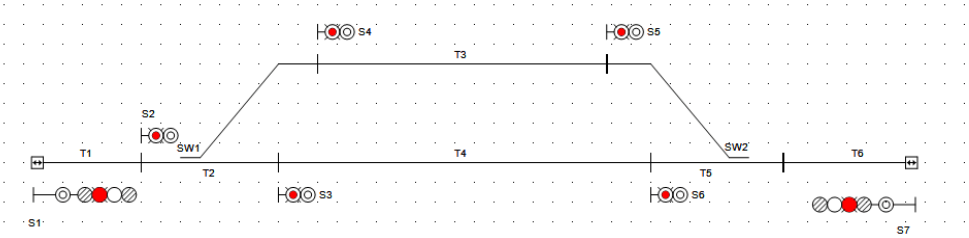


Figure 2. An example of track layout diagram

Apart from the track layout diagram shown in Figure 2, a railway yard still needs *interlocking tables* to specify the train routes of the station. It describes how these components are topologically configured, including the conditions for locking and releasing the train route and for when the entry signals of the route is set to show proceed or stop.

3. System Model Patterns

According to the processing flow of interlocking systems, we get its context diagram, as shown in Figure 3. The system contains the *environment* and the *controller*. The environment consists of five entities, namely, *Train*, *Signal light*, *Point*, *Track* and *Interlocking Table*. The *controller* and the *environment entities* interact with each other. The interactions are the shared message between the controller and the environment entities. From Figure 3, there are 18 interactions. For example, the train entity sends "trainEnter", "trainLeave", and "request" messages to the controller.

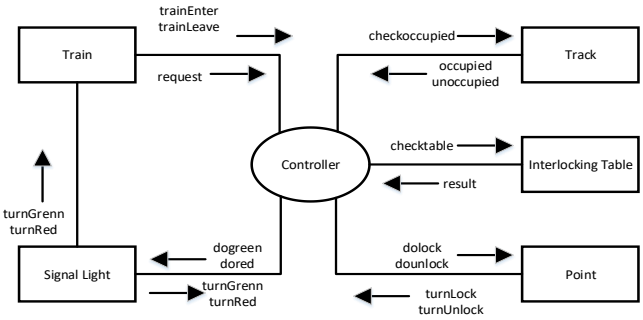


Figure 3. Context diagram of the interlocking system

3.1. Environment entity patterns

Firstly, we give a 4-step process to obtain the SHA of each environment entity. The four steps are extracting the processing flow of the entity, constructing automata, modeling faults, and adding the time constraints.

Step 1: Extracting the processing flow of the entity

The process description related to the entity is found in the system processing flow, including all the behaviors related to the entity. The process can be written in the form of activity diagram.

Step 2: Constructing automata

This step is to transform the activity diagram to an automata. Each time the activity diagram sends or receives a message (action), the entity's automata moves from one state to another state. Therefore, an action is transformed into a state and a transition in an automata. The transition is action. The specific transformation process is shown in Algorithm 1.

Algorithm 1: activity diagram to automata transformation algorithm
Input:An activity diagram Ad=<P0,P,T>
Output:An automata Am=<L,E>
Procedure

```
1.    Read activity diagram Ad;
2.    S=P0
3.    While(S){
4. Find the node of T.source=S;
5.      if (S.type==activity){
6.            Define a location l in an automaton,  L=L ∪{l};
7.            Define a edge e  in an automaton, E=E ∪{e};
8.            if (S.action==send)
9.                  e.message = S.message+"!";
10.             else
11.                 e.message==S.message+"?";
12.   }
13.   else if (S.type==decision){
14.           Define a l in an automaton,  L=L ∪{l};
15.           Search T where T.source=S
16.           for(int i=0;i<T.size;i++){
17.             if (S.action==send)
18.                 e.message = S.message+"!";
19.             else
20.                 e.message==S.message+"?";
21.           }
22.   }
23.     else if(S.type==merge){
24.           Find the last unhandled node connecting the merge node;
25.           The transition  points to the state of the next node of the merge l;
26.   }
27.   else{
28.           Define a location l in an automaton,  L=L ∪{l};
29.   }
30.     S=T.target;
31. }
```

Step 3: Modeling faults

In environmental entities, it is possible that the occurrence of an abnormal event can lead to a fault, and an abnormal event can be represented by the probability. Therefore, we find all the abnormal events, and use stochastic probability events to express them. Different events are simulated with different probabilities. Based on step-2 automata, stochastic probability events are added to model faults.

Step 4: Adding time constraints

This step is to add time constraints on the results of step 3. Firstly, extract the time constraints of entities. This is domain knowledge. Then, define the corresponding clock variables. The representation of the clock constraint in the automata is the time between one message and another message; that is, in the automata, the initial value of clock variable on the "update" of previous message transition is 0, and the inequality of the clock variable is defined in the "guard" of the transition of the subsequent message.

Next, we will present the detailed process for each entity.

Train:

1) Extracting the processing flow of the entity

According to the system processing flow, we obtain the processing flow of the train entity, and represent them in ang h activity diagram, as shown in Figure 4.

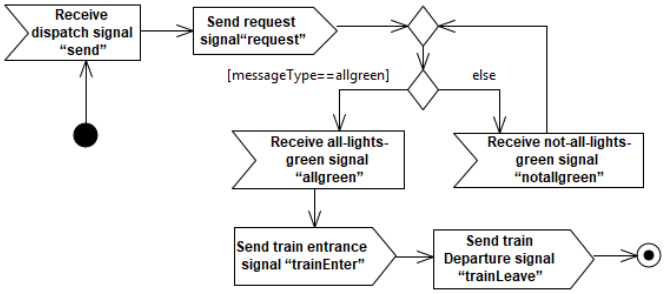


Figure 4. Context diagram of the interlocking system

2) Constructing automata

According to the algorithm 1, the automata of the train entity is obtained, which contains 7 states and 6 transitions, where *state 1* and *state 7* are the initial state and the final state respectively. The messages on the transition are: "send?", "request!""notallgreen?", "allgreen?", "trainEnter!", and "trainLeave!".

3) Modeling faults

A fault may occur during the train running, that is, between sending "trainEnter" message and sending "trainLeave" message. Add an error state to the automata. Message sent from "trainEnter" is transferred to the error state with the probability of m% and transferred to the starting point of "trainLeave" message with the probability of n%. In the probabilities, m + n = 100, m and n are real numbers, and are decided by domain experts.

4) Adding the time constraints

The time constraints obtained from domain experts are as follows: The not-all-lights-green signal "notallgreen" is received within specified time. The all-lights-green "allgreen" is received within specified time. The clock variable x is defined to indicate the waiting time for the signal light, that is, the time from sending "request" message to receiving "notallgreen" message or "allgreen" message. Therefore, the "update" initial value of x for the "request" transition is 0, and the inequality "x<z" (z is a constant) is used as the "guard" for transition "notallgreen" or "allgreen". The SHA of the train is thus obtained, as shown in Figure 5(a).

Signal Light:

We divide the activities involved in *Signal Light* into two parts. One is *SSignalLight*, which is responsible for setting the status of the signal light. The other part is *RSignalLight*, which is responsible for inquiring the status of the signal light.

$$Signal\ Light = SSignalLight \parallel RSignalLight$$

(1) *SSignalLight*

The modeling process of *SSignalLight* is as follows:

1) Extracting the processing flow of the entity

According to the system processing flow, we get the processing flow of the *SSignalLight*, and represent them in an activity diagram as shown in https://github.com/wymgal/IS.git.

2) Constructing automata

According to the algorithm 1, the automata of the *SSignalLight* is obtained as shown in https://github.com/wymgal/IS.git, which contains 4 states and 4 transitions, where *state 1* is the initial state. The transitions are: "dogreen?", "turnGreen!", "dored?" and "turnRed!".

3) Modeling faults

A fault may occur during the change of the signal lights' states, that is, between receiving "dogreen" message and sending "turnGreen" message or between receiving "dored" message and sending "turnRed" message. Add an error state to the automata. Message sent from "dogreen" is transferred to the error state with the probability of m% and transferred to the starting point of "turnGreen" message with the probability of n%. Similarly, we can model faults in the situation where the signal light changes from green to red.

4) Adding time constraints

The time constraints obtained are as follows: There is a certain delay in the change of *Signal Light* condition. According to the time constraints, the local clock *a* is given to indicate the delay time of signal light changing from red to green, and the local clock *b* indicates the delay time of signal changing from green to red.

The clock a represents the time from receiving "dogreen" message to the next state *GREEN*. Therefore, the "update" initial value of for the "dogreen" transition is 0, and the inequality "a>1" is used as the "guard" for the transition. Using the same method, we can define the clock *b* in this automata. The SHA of the train is thus obtained, as shown in Figure 5(b).

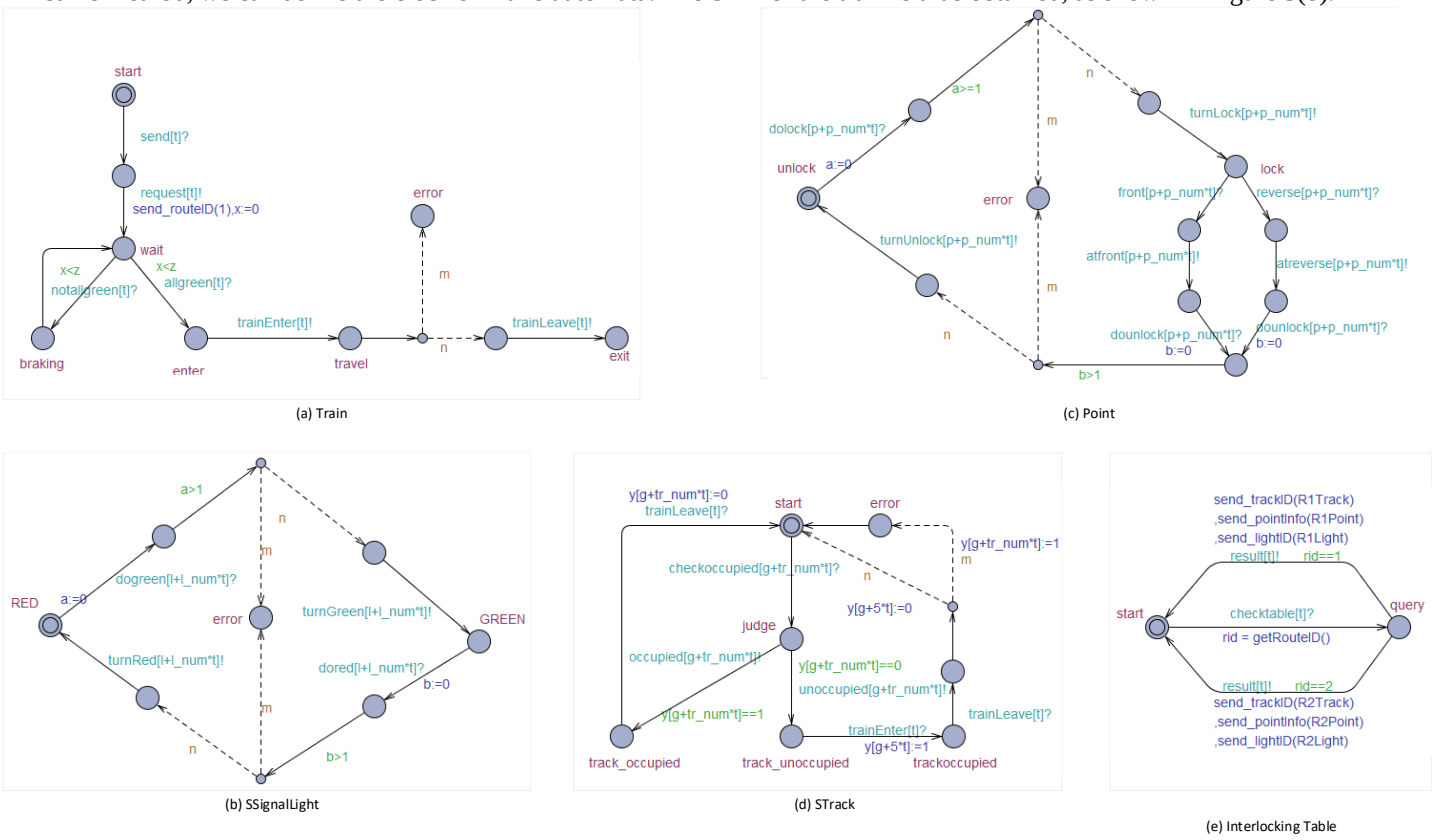


Figure 5. SHA models of environment entities

(2) *RSignalLight*

In order to get a set of single light states, we start with a single light. According to the system processing flow, we can get the processing flow of the *RSignalLight*. According to it, we build an automata model for one signal light as shown in Figure 6. There are 5 locations in the automata. They are start location, normal location *p1*, normal location *p2*, normal location *p3* and normal location *p4*.

There are 10 transitions in the automata. From the start location to the end location, draw a transition whose message is "turnGreen?" and make the green identifier "isLightGreen" be true. Similarly, draw a transition whose message is "turnRed?" and make the red identifier "isLightRed" true. From *start* location to *p2*, draw a transition and determine the value of "isLightGreen" on its "guard". If the "isLightGreen" is true, the message of this transition is "allgreen!". From *p2* to

start location, draw a transition without message so that the automata can repeatedly determine the value of "isLightGreen". Similarly, we can get the rest of the transitions.

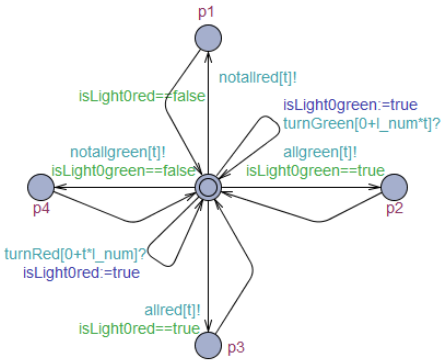


Figure 6. Automata of RSignalLight

For a group of lights, the modeling process is based on the situation of one signal light. Add the corresponding different signal lights' green identifier *isLightGreen* and the red light identifier *isLightRed*. For *N* signal lights, there should be *n* isLightGreen[0,1,...,n-1] and isLightRed[0,1,...,n-1]. Add n transitions with the message of "turnGreen[n-1]?" from the initial state to the itself. Make isLightRed[n-1]=1. The judgement condition of transition "allgreen!" is: isLightGreen[0]==1&&isLightGreen[1]==1&&...&&isLightGreen[n-1]==1. Similarly, we change the judgement condition of transition "notallGreen!", transition "allred!" and transition "notallred". The automata for a group of signal lights is in https://github.com/wymgal/IS.git.

In this paper, the error of *RSignalLight* is not considered, as well as the time constraints of the *RSignalLight*.

Point:

1) Extracting the processing flow of the entity

According to the system processing flow, we can get the processing flow of the point entity, and represent them in an activity diagram, as shown in https://github.com/wymgal/IS.git.

2) Constructing automata

According to the algorithm 1, the automaton of the point is obtained as shown in https://github.com/wymgal/IS.git, which contains 8 states and 9 transitions, where *state 1* is the initial state. The messages on the transition are: "dolock?", "turnLock!", "front?", "reverse?", "atfront?", "atreverse?", "dounlock?", "dounlock?", and "turnUnlock!".

3) Modeling faults

A fault may occur during the point changing from the locked state to the unlocked state or from the unlocked state to the locked state, that is, between receiving "dolock" message and sending "turnLock" message or between receiving "dounlock?" message and sending "turnUnlock" message. Add an error state to the automata. Message sent from "dolock" is transferred to the error state with the probability of m% and transferred to the starting point of "turnLock" message with the probability of n%. Similarly, we can model faults in the situation where the point changing from unlocked state to the locked state.

4) Adding time constraints

The time constraints for point entity is that there is a certain delay in the change of point condition. The local clock *a* is defined to indicate that the delay time of the point changing from unlocked state to locked state, and the local clock *b* indicates the delay time of the point changing from locked state to unlocked state. That is, clock *a* is the time from receiving "dolock" message to the next state lock. Therefore, the "update" initial value for the "dolock?" transition is 0, and the inequality "a>1" is used as the "guard" for the transition. Similarly, we can define the clock *b*. The SHA of the point is thus obtained, as shown in Figure 5(c).

Track:

We divide the activities involved in Track into two parts. One is *STrack*, which is responsible for setting the status of the Track. The other part is *RTrack*, which is responsible for inquiring the status of the *Track*. So we get:

$$Track = STrack \parallel RTrack$$

(1) *STrack*

1) Extracting the processing flow of the entity

According to the system processing flow, the processing flow of the *STrack* entity is extracted and represented by activity diagrams as shown in https://github.com/wymgal/IS.git.

2) Constructing automata

According to the algorithm 1, the automata of the *STrack* is obtained as shown in https://github.com/wymgal/IS.git, which contains 5 states and 6 transitions. The messages on the transition are: "checkoccupied?", "occupied!", "unoccupied?", "trainEnter?", "trainLeave?", and "trainLeave?".

3) Modeling faults & Adding time constraints

A fault may occur during the process of setting the track state, that is, between receiving "trainEnter" message and receiving "trainLeave" message. Add an error state to the automata. Message sent from "trainEnter" is transferred to the error state with the probability of m% and transferred to the starting point of "trainLeave" message with the probability of n%. The method of adding transition with a probability is consistent with the case of the train entity.

Without taking into account the time constraints of track entity, we do not do time constraints extraction.

(2) *RTrack*

RTrack is divided into two steps. One is for a track. We build an SHA model for a track, as shown in https://github.com/wymgal/IS.git . There are 3 locations in the automata and 6 transitions. The value of "isTrackun" is true. From *start* location to *p1*, draw a transition and determine the value of "isTrackoc" on its "guard". If the "isTrackoc" is true, the message of this transition is "trackoccupied!". From *p1* to *start* location, draw a transition without message to make the automata can repeatly determine the value of "isTrackoc". Similarly, we can get the rest of the transitions.

The other one is for a group of tracks. The processing is similar with *RSignalLight*. We only need to change the judgment conditions and transitional messages.

Interlocking Table

1) Extracting the processing flow of the entity

According to the system processing flow, the processing flow of the Interlocking Table entity is obtained and represented by an activity diagram in https://github.com/wymgal/IS.git.

2) Constructing automata

According to the algorithm 1, the automata of the interlocking table is obtained as shown in https://github.com/wymgal/IS.git, which contains 2 states and 2 transitions, where *state 1* is the initial state. The messages on the transition are: "checktable?" and "result!". If there are *n* query results, there are *n* transitions with the "result!" message, and all the transitions point to the start location.

This paper does not consider the error situation and time constraints of the interlocking table entity, so we do not do fault modeling.

3.2. Controller Pattern

The controller is divided into two parts. One is the center which is responsible for controlling all the tracks, points, signal lights, trains and interlocking tables. The other part called *submodels* is in charge of controlling each track, point, signal light and train respectively. So we define:

$$Controller = Center \parallel Submodules$$
$$Submodules = CTrack \parallel CPoint \parallel CSignalLight \parallel Dispatcher$$

The *CTrack* is responsible for sending "checkoccupied" message to each track, and checking the occupancy situation of each track. After receiving the instruction of *Center*, the *CPoint* sends "dolock" and "dounlock" message to each point. After

receiving instruction from Center, the *CSignalLight* sends "dogreen" and "dored" message to each signal light. The *Dispatcher* is responsible for sending dispatching instructions to control different trains entering the track at different time.

1) Extracting controller related processing scenarios

According to the system processing flow, the processing scenarios of the *Center*, the *CTrack*, the *CPoint*, the *CSignalLight*, and the *Dispatcher* are extracted as expressed in https://github.com/wymgal/IS.git.

2) Constructing automata

a. Constructing *Center* automata:

According to the algorithm 1, the automata of the *Center* is obtained as shown in Figure 7, which contains 15 states and 16 transitions, where *state 1* is the initial state. The message on the transition are:"request?", "checktable!", "result?", "checktrack!", "trackoccupied?", "allunoccupied?", "lockpoint!", "finishmove?", "dolightgreen!", "allgreen?", "trainEnter?", "trainLeave?", "lockpoint!", "allunlock?", "dolightred!", and "allred?".

In the global declaration, we define 9 functions required by *Center* to interact with each entity. They are send_*routeID*, *getRouteID*, *send_trackID*, *getTrackID*, *send_pointInfo*, *getPointInfo*, *send_lightID*, and *getLightID*. Their functions are the meaning of their name. Due to limited space, the exact definition is shown in https://github.com/wymgal/IS.git.

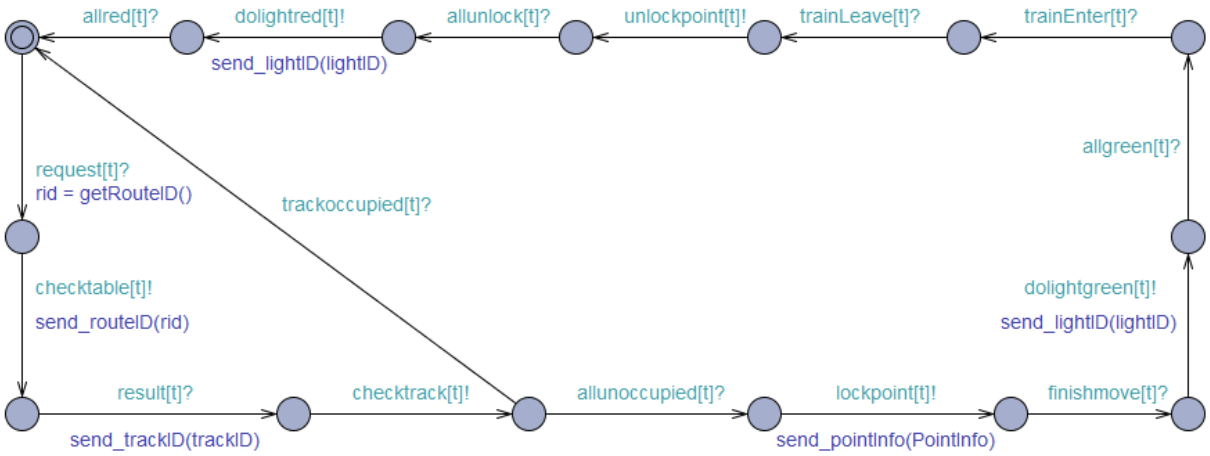


Figure 7. The Center pattern

b. Constructing *CTrack*:

Building *CTrack* is divided into controlling one track and a group of tracks. For controlling one track, starting with an initial state, draw a transition with the message of "checktrack?", which points to the next state. Draw a state and a transition from the current state to the next state with the message of "checkoccupied!". The last "checkoccupied!" transition points to the initial state.

For a group of tracks, give n transitions for N tracks, i.e., n "checkoccupied[0,1,...,n-1]" from the current state to the next state. The last "checkoccupied" points to the initial state.

c. Constructing *CPoint*:

According to the processing flow of *CPoint*, we build an automata model for *CPoint*.

Draw a transition from the initial state to the next state with the message of "lockpoint?". Draw a state and a transition with message of "dolock!" from current state to the next state. Draw a state and a transition with message of "turnLock?" from current state to current state. On this transition, the point lock identifier "isPointlock" assigned to 1. Draw another transition on the last state, and define the judgment condition for the point lock identifier "isPointlock == 1" on the transition.

The processing of moving points is similar with locking points. We just need to change the messages. There is a transition from the last state to the initial state.

For a group of points, the processing method is the same with *CTrack*. We only need to change the judgement conditions and messages.

d. Constructing *CSignalLight*:

The processing process of *CSignalLight* is the same with *CTrack*. We only need to change the messages. For example, message "checktrack?" changes to "dolightgreen?". Message "checkoccupied!" changes to "dogreen!".

e. Constructing *Dispatcher*:

Draw a transition from initial state to the next state with the message of "send[0]!". Draw a state and a transition from current state to the next state with the message of "trainEnter[0]?". Draw a state and a transition from current state to the next state with message of "send[1]!". If there are n trains, we add n transitions with message of "send[1,2,..., n-1]!" and add n transitions with the message of "trainEnter[0,1,2,...,n-1]?".

4. Three Models Construction Process

4.1. System model construction

This system model is constructed in 3 major steps:

1) Defining the System

This step consists of 4 sub-steps:

a. Declare system models in model declarations:

This sub-step is to declare all the models in the system. Through analysis, we define that the system is composed by 13 models, so we make the following declaration:

system Train, Track, Light, Point, Center, Dispatcher, CSignalLight, CPoint, CTrack, RSignalLight, RTrack, InterlockTable, Monitor ;

b. Instantiate different entities in global declaration:

This sub-step is to get model instantiations. Firstly, we get the number of trains, signals, tracks and points from the interlocking table. Suppose *Nt*, *Nl*, *Nr* and *Np* are the number of trains, signals, tracks and points respectively. The models could be instantiated by the following declarations:

```
const int TRAINS=Nt; typedef int[0,TRAINS-1] train_t;
const int LIGHTS =Nl; typedef int[0,LIGHTS-1] light_l;
const int TRACKS = Nr; typedef int[0,TRACKS-1] track_t;
const int POINTS=Np; typedef int[0,POINTS-1] point_p;
```

c. Building templates for the various environmental entity models

Reuse patterns of Train, Signal, Point, Track and Interlocking Table in Section 3. The parameters in each template can be modified according to specific situations. According to the number of entities, the *RSignalLight* and *RTrack* model can be modified.

d. Reusing controller template

Reuse the Center pattern in Section 3, create the SHA model of the center, and modify the SHA models of the controller *submodules*.

2) Defining system interactions

The interactions of the system are achieved by the communication between the entities in the context diagram. So defining the system interactions is to define all the messages in the communication between entities. Define all messages in the global declaration. For example, Chan green[Nt*Nl]. According to the interlocking table, we can know that one train needs *Nl* signal lights, that is, needs *Nl* green signals. So for *Nt* trains, there should be *Nt*Nl* green signals. The other messages are declared in https://github.com/wymgal/IS.git.

3) Declaring system variables

The global variables required by the system are actually shared information between models. They should include the track occupancy identifier, and the number of instantiations of each entity in the global declaration. In addition, the variables used by the functions of Center should be declared too. The exact declaration is as follows.

```
int y[Nt*Nr] ={0,0,...,0}// the track occupancy identifier
const int l_num=Nl, p_num=Np, tr_num=Nr; // the number of instantiations of each entity
int route_id, trackID[Nr], PointInfo[Np][Np], lightID[Nl];   // the variables used by the functions
```

4.2. Monitor model

We construct the monitor model in two steps:

1) Extracting time constraints in the center model

Referring to the related data of the interlocking system, the time constraints of interactive behaviors in the interlocking system are extracted. These constraints can be expressed as <message, message, constraint expression>, such as <checktrack, trackoccupied, <= 4s>, indicating that the time difference between message "checktrack" and "trackoccupied" is less than 4s.

2) Generating Monitor

This step is to generate the Monitor using an algorithm. The basic idea is to monitor the time constraints related events in the Center. Suppose there are n time constraints, and the m events they involved is $Events=\{event_1, event_2,\dots, event_m\}$. (1) From the start node V_0 of the Center model, access the event $event_i$ which connects to the point. (2) If the eventi belongs to the *Events*, define a state and a transition from current state to the next state with the message of "$event_i$?". If the $event_i$ does not belong to Events, continue to access the next node. (3) Repeat step 2 until all nodes and edges are accessed. The detailed process is shown in Algorithm 2.

```
Algorithm2: Monitor generation algorithm
Input:An Center Model Ca=<P0,CL,CE>, Constraints[]={startEvent,endEvent,constraintInfo}
Output:An Monitor Model<L,E>
Procedure
1.    Read Center Model Automata Ca;
2.    Define a e0 in an automaton , E=E ∪{e0};
3.    Define a clock variable x in an automaton ;
4.    S=P0
5.    While(S){
6.      Find the node of CE.source=S;
7.      if (CE.message in Constraints[]){
8.            Define a l in an automaton,  L=L ∪{l};
9.            Define a e in an automaton, E=E ∪{e};
10.           Assign x to 0 on e;
11.           e.message = S.message+"?";
12.           if(CE.message.type==start)  {
13.                 Define a e in an automaton, E=E ∪{e};
14.                 e.source=next S;
15.                 e.target=e0;
16. Define the expression of x "x>constraintInfo.number" on e;
17.           }
18.    }
19.    S=CE.target;
20.  }
```

4.3. Prediction model

The primary risks of the rail transit system for accidents include derailment and collision [13]. This paper only considers the situation of collision. The basic idea is to use the time interval between the two trains entering same track to judge whether they will collide or break down, which leads to low efficiency. So we consider two situations.

Situation 1: Two trains enter the same track at different time, if the time interval is less than the specified time, a collision accident may occur.

Situation 2: The first train has entered the track and the next train is waiting. In order to avoid the collision accident, how long it will take for the next train entering the track.

We need to calculate the minimum waiting time T_{min} in the first situation and the maximum waiting time T_{max} in the second situation. Assume that the distance between the two trains is S:

Situation 1: the two trains have the same speed *V* at the beginning, but the previous train is slowing down and the next train is speeding up. Their acceleration are *-a* and *a* and the number of *a* is the maximum acceleration of the train.

By the acceleration formula [12] (1), we can get $T_{min}=\sqrt{\frac{S}{a}}$.

$$S = V * T_{\min} + \frac{1}{2} a T_{\min}^2 - (V * T_{\min} - \frac{1}{2} a T_{\min}^2) \quad (1)$$

Situation 2: the previous train is speeding up with an acceleration *a*, and the initial speed is 0. It will run with a constant speed after the speed reaches *V*, and the number of *a* is the maximum acceleration of the train.

By the acceleration formula (2), we can get $T_{max}=\frac{V}{a}+\frac{S-\frac{V^2}{2a}}{V}$. To avoid collision, the next train waits T_{max} minutes at most.

$$S = \frac{1}{2} a t_1^2 + V * t_2, \quad t_1 = \frac{V - 0}{a} \quad (2)$$

Based on this, we define a prediction model as follows:

Suppose there are two trains, train1 and train2. We choose the time duration between them to predict their relations.Duration=Train2.entertime-Train1.entertime

- *If Duration <=T_{min}, there may be a collision, it should issue an early warning.*
- *If Duration >T_{max}, there may be a failure waiting.*

5. Case Study

In this paper, we still use the case whose track layout diagram is shown in Figure 2. Table 1 shows its interlocking table. There are two routes, Route1 and Route2, 5 signal lights S1, S2, S4, S5, and S7 on the Route1, and 5 lights S1, S2, S3, S6, and S7 on the Route2. Two points SW1 and SW2, and five tracks T1, T2, T3, T4, and T5 are included.

5.1. System model construction

1) Defining the system

Firstly declare the same 13 models as listed in Section IV. From Table 1, we get the numbers of the trains, signal lights, tracks and points, which are 2, 5, 5, 2 respectively. It means Nt=2, Nl=5, Nr=5, Np=2. Put them into the model declaration to declare the models of the system. Then instantiate different entities in global declaration:

```
const int TRAINS=2;    const int LIGHTS =5;  const int TRACKS = 5;  const int POINTS=2;
```

Table 1. Interlocking table of Figure 1

Route			Signals		Points			Track
ID	From	To	Green	Red	Open		Close	
					Up	Down		
R1	S1	S7	S1,S2,S4,S5,S7	S3,S6	SW1,SW2			T1,T2,T3,T5,T6
R2	S1	S7	S1,S2,S3,S6,S7	S4,S5		SW1,SW2		T1,T2,T4,T5,T6

We reuse the patterns of *Train*, *Signal Light*, *Point*, *Track* and *Interlocking Table* in the Section 3, and modify *RSignalLight* and *RTrack* according to their numbers 5 and 5. We modify the number of transitions and judging conditions, and get these two models. *RSignalLight* model is shown in Figure 8 while *RTrack* model is listed in https://github.com/wymgal/IS.git. Finally construct the controller model. We reuse the Center pattern in Section V, and

build *CTrack*, *CPoint*, *CSignalLight* according to 5 tracks, 2 points and 5 signal lights. Reuse the *Dispatcher* of Section V. We add two clock variables to Dispatcher, clock variable m starts timing when the train0 enter the track, and the clock variable n starts timing when the train1 enter the track.

The *CTrack* is shown in Figure 9(a) while the others are listed in https://github.com/wymgal/IS.git.

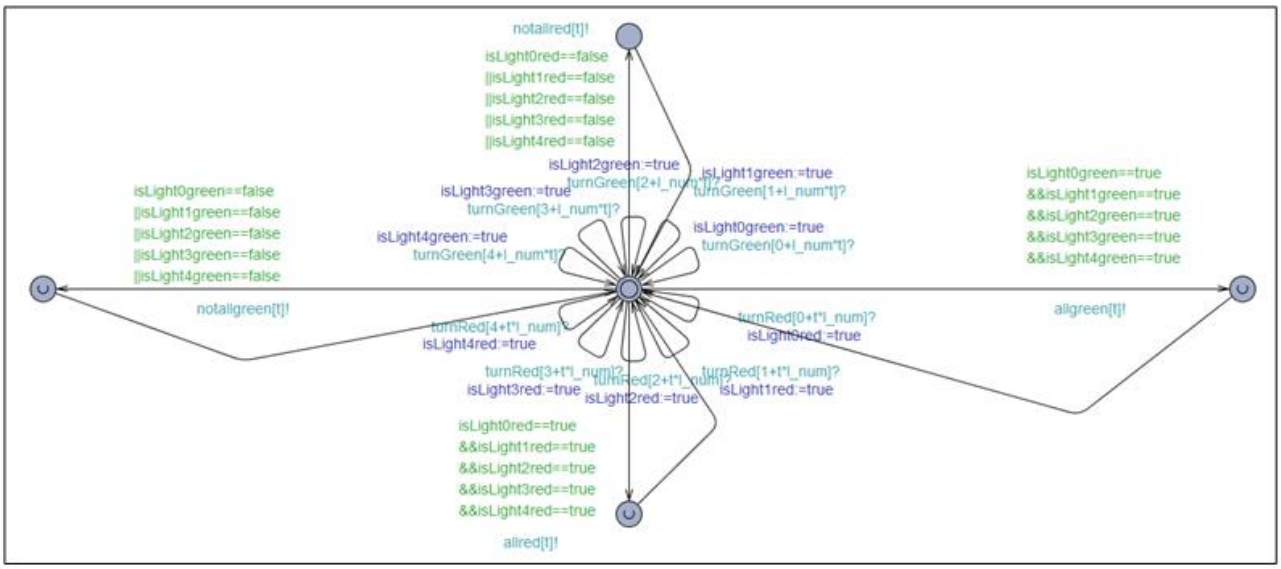
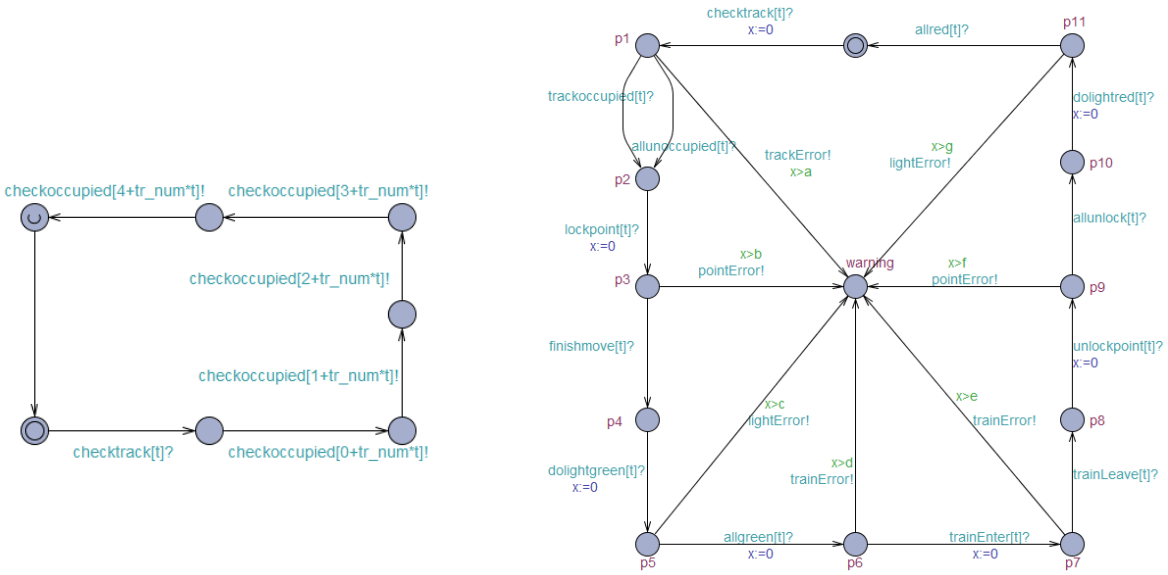


Figure 8. SHA of SSignal Light in the example



(a)*CTrack* (b) Monitor

Figure 9. SHA of *CTrack* andMonitor

2) Defining system interactions and declaring system variables

Put *Nt*=2,*Nl*=5, *Nr*=5,*Np*=2 into the global declaration：

```
int y[10] ={0,0,0,0,0,0,0,0,0,0};   // The track occupancy identifier is occupied
const int l_num=5, p_num=2,tr_num=5; // The numbers of each entity
int route_id,  trackID[5],  PointInfo[2][2],  lightID[5];   // Declare the number of instantiations of each entity:
```

5.2. Monitor model construction

1) Extracting time constraints in center model

According to domain knowledge, we get the following time constraints:

<checktrack,trackoccupied,<=4s>, checkreack,allunoccupied,<=4s>,
<lockpoint,finishmove,<=4s>, <dolightgreen,allgreen,<=4s>,
<allgreen,trainEnter,<=10s>, <trainEnter,trainLeave,<=200s>,
<unlockpoint,allunlocked,<=4s>, <dolightred,allred,<=4s>.

From these 8 constraints and related14 events, we record them as: *Events*={checktrack, trackoccupied, allunoccupied, lockpoint, finishmove, dolightgreen, allgreen, trainEnter, trainLeave, unlockpoint, allunlocked, dolightred, dolightred, allred}

2) Generating Monitor model

According to the monitor generation algorithm 2, a monitor model could be obtained automatically as shown Figure 9(b). It has 12 states and 20 transitions including 7 transitions of warning state. So we add warning messages to the 7 transitions.

5.3. Prediction model construction & prediction

From domain knowledge [16,18], we get the acceleration of the train is always between 1 and 2 m/s2. The speed is always between 80~350km/h that is 22~97m/s, and the length of a track is between 500 and 1000m. So in this case, we specified the distance between the two trains to be 1000m, the acceleration to be 1m/s2, and the speed to be 50m/s.

Input these data into the two situations predefined in the prediction model, we can get T_{min}≈31.6s, which means the time for two trains to enter the same track is at least 31.6s, and T_{max}=45s, which means the next train's waiting time is 45s at most after the previous train entering the track.

Table 2. Failure condition table of experimental results

ID	Train1.entertime	Train2.entertime	x	Monitor warning	Result	Cause
1	13.736	280.866	5.809	trackError!	low efficiency	Don't meet the time constraints <checkreack,allunoccupied,<=4s>.
2	328.029	2872．028	18.382	trackError!	collision accident	Don't meet the time constraints <checktrack,trackoccupied,<=4s>.
3	96.616	99862.396	16.428	pointError!	low efficiency	Don't meet the time constraints <lockpoint,finishmove,<=4s>.
4	273.927	97497.397	159.368	lightError!	low efficiency	Don't meet the constraint <dolightgreen,allgreen,<=4s>.
5	17.806	3183.947	3142.304	trainError!	low efficiency	Don't meet the constraint <allgreen,trainEnter,<=10s>.
6	330.489	3986.472	3510.631	trainError!	low efficiency	Don't meet the constraint <trainEnter,trainLeave,<=200s>.
7	230.983	4694.566	3511.963	pointError!	collision accident	Don't meet the constraint <unlockpoint,allunlocked,<=4s>.
8	362.215	1468.994	1175.120	lightError!	collision accident	Don't meet the constraint <dolightred,allred,<=4s>.
9	398.783	420.381	\	error!	collision accident	The time difference between the two trains enter the same track is less than T_{min}, so the two trains are likely to collide.

After getting these 3 models, we simulate them in UPPAAL-SMC. After 5,000 simulations, we get 9 kinds of situations listed in Table 2. The monitor gives 6 alarms. In these results, 4 kinds of situations may lead to collision accidents. The causes should be seriously paid attention to.

6. Related Work

There are many ways to analyze the causes of accidents, which are mainly divided into analytical method and model-checking based method.

The analytic method mainly includes failure mode and effects analysis (FMEA) and fault tree analysis method (FTA) [7]. FMEA firstly models the possible faults and types of the components and elements in the system, then identifies the effects of various types of faults on adjacent parts or elements and the eventual impact on the system, and finally proposes

measures to avoid or minimize these effects [16,17]. Similarly, FTA firstly models the system fault through the analysis of various factors [10]. The fault tree is constructed to determine the possible combinations and probability of the system faults. These methods both belong to static analysis without paying attention to time. However, the faults in the railway transit systems may relate to real-time property, their occurrences may have temporal relations, and the occurrences of accidents in the railway transit systems rely on speed, distance and time. So the traditional analytic methods are not very suitable for railway transit systems. Instead, our approach dynamically monitors the interactions between system components and their time constraints, and can therefore catch the faults caused by other faults.

The model checking based method firstly models a system model, and checks it against given properties in CTL or LTL by exhaustive search [14,15]. The causes can be found by the counter-examples. But that approach cannot be used in large scale project due to the state space explosion problem. Our approach using the Statistical Model Checking (SMC) is a simulation based approach. So the size of the project will not affect the effect of this method. In contrast to the whole system model in the traditional model checking methods, our three-layer model is more easy to find the reason.

7. Conclusions

Accidents cause is of great important for the safety of railway transit. This paper proposes a model monitoring method that is based on SHA to detect the fault and predict accident in the interlocking system. Major contributions include: (1) the template pattern in the interlocking system model is extracted and non-professionals can reuse the pattern according to the system parameters of the different systems to quickly customize the specific system model and (2) a three-layer model framework is proposed, which includes system model, control model and monitoring model. The system model can be reused to construct a detailed model of different systems. The monitoring model can monitor different entities in the system model and it can be automatically generated according to the corresponding algorithm by introducing the corresponding time constraint.

Acknowledgements

This paper is partially supported by the projects funded by the NSFC Key Project 61332008, NSFC 61572195 and 61472140, and SHEITC160306.

References

1. "7.23" Yong Wen line special major railway traffic accident investigation report [EB/0L].(2011-12-25) [2013-02-10]. http://www/gov.cn/gzdt/2011-12/29/content_2032986.html.
2. A. Bemporad and S. D. Cairano, "Optimal Control of Discrete Hybrid Stochastic Automata. Hybrid Systems: Computation and Control," Springer Berlin Heidelberg, 2005
3. L. Bortolussi and A. Policriti, "Stochastic Programs and Hybrid Automata for (Biological) Modeling," *Conference on Computability in Europe: Mathematical Theory and Computational Practice*, Vol.5635, pp.37-48, Springer-Verlag, 2009
4. P. Bulychev, A. David, K. G. Larsen, M. Mikučionis, D. B. Poulsen, and A. Legay, et al. "Uppaal-smc: Statistical Model Checking for Priced Timed Automata," *Electronic Proceedings in Theoretical Computer Science*, 85, 2012
5. G. Carmignani, "An Integrated Structural Framework to Cost-based Fmeca: the Priority-cost Fmeca," *Reliability Engineering & System Safety*, 94(4), 861-871, 2009
6. G. Chen, Z. Yang, J. Zhao, and Z. Fei, "Dynamic Bayesian Networks Method of Safety Analysis Based on Reliability Block Diagram," in *Proceedings of IEEE International Conference on Reliability, Maintainability and Safety* pp.1047-1051, 2015
7. M. L. Chiozza and C. Ponzetti, "Fmea: a Model for Reducing Medical Errors," *Clinica chimica acta; international journal of clinical chemistry*, 404(1), 75, 2009
8. S. K. Chen, B. H. Mao, H. E. Tian, J. F. Liu, and H. D. Liu, "Reliability Evaluation of Railway Traction Power Systems by Fault Tree Analysis," *Journal of the China Railway Society*, 28(6), 123-129, 2006
9. G. B. A. David and K. G. Larsen, "A Tutorial on Uppaal 4.0," *Department of Computer Science*, 4(12), 200—236, 2006
10. X. Feng and X. F. Wang, "Analysis on Reliability and Performance of Computer-based Interlocking System with the Dynamic Fault Tree Method," *Journal of the China Railway Society*, 33(12), 78-82, 2011
11. V. Hartonas-Garmhausen, A. Cimatti, E. Clarke, and F. Giunchiglia, "Verification of a Safety-critical Railway Interlocking System with Real-time Constraints," *Science of Computer Programming*, 36(1), 53-64, 2000
12. U. H. D. Ing, R. W. D. Ing, and J. H. D. Ing, "Acceleration and Braking Control for Train with Logic Control for Even force and Minimum Braking Distance," DE2522958, 1976
13. H. Jin, Zhang, and Wu. "Urban Rail Transit Risk Management System Establishiment," *Urban Rapid Rail Transit*, 2010
14. Y. Jiang, B. Cukic, and Y. Ma, "Techniques for Evaluating Fault Prediction Models," *Empirical Software Engineering*, 13(5), 561-595, 2008
15. T. M. Khoshgoftaar, and N. Seliya, "Tree-based Software Quality Estimation Models for Fault Prediction," in *Proceedings of the 8th IEEE Symposium on Software Metrics*, pp.203-214, 2002.

16. X. Y. Lei, H. Huang, M. T. Sun, and Q. J. Liu. "Dynamic Property Real-time Monitoring System of Railway Turnout," CN 104787085 A, 2015
17. Q. Z. Pang, Y. F. Li, F. Lei, and L. U. Shun Qing, "Discussion on the Safety Assessment for the Manufacture of Oxygen Enterprise," *Journal of Safety Science & Technology*, 2007
18. C. Woo-Jin, K. Dong-Whe, and S. C. Yang, "Verification of the Validity for the Speed-limit Regulation on the Turnout System," 8(1), 2005
19. D. R. Zeng, H. E. Zheng-You, and Y. U. Min, "Reliability Analysis of Metro's Traction Substation," *Journal of the China Railway Society*, 30(4), 22-27, 2008