Modeling of Interlocking Systems based on Patterns

Wang Yan, Zhong Wen, Xiaohong Chen*, Dehui Du

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

*corresponding author, xhchen@sei.ecnu.edu.cn

Abstract—**The equipment faults of the interlocking system in rail transit system has occurred frequently, and these faults can cause serious accidents. The modeling of the interlocking system must aim at the stochasticity and real-time characteristics of equipment faults. Therefore, this paper proposes to model the interlocking system using stochastic hybrid automata. In order to improve model efficiency, we try to extract the pattern of the interlocking system model, and reuses these patterns in system modeling. The main contributions include: (1) Based on the business analysis of the interlocking system, 12 model patterns of the interlocking system are extracted; and (2) the modeling process of the interlocking system based on patterns reuse is given to guide system modeling. Finally, a case study is presented to illustrate the feasibility and effectiveness of our approach.**

Keywords-Interlocking system; Pattern reuse; Stochastic hybrid automata; Modeling; UPPAAL-SMC

I. INTRODUCTION

The safety of rail transit systems is of great importance. However, in recent years, railway accidents happen from time to time. For example, July 23, 2011 Yong Wen line "7.23" major railway traffic accident. This is a serious accident caused by the design flaws of the equipment in the control center, inadequate checks on the road, and ineffective emergency response after equipment failure which caused by lightning. The accident led to a rear-end train collision, and caused 40 deaths, 172 injuries, and the direct economic losses of 19371.65 million RMB [1].

The interlocking system is one of the core subsystems in the rail transit system [2]. It has SIL4-level safety requirements, complex logic, and high requirements for real-time performance [3] [4]. It can endanger the safety of the vehicle when a failure occurs. Therefore, modeling and analyzing the interlocking system become very important and is one of the key methods to ensure system safety.

The interlocking system is composed by various equipment. The main cause of most accidents in interlocking system is equipment failure which is stochastic. For example, lightning strike causes the short circuit of track and leads to an accident. Therefore, we should consider the stochastic characteristic of equipment faults when modeling. Stochastic Hybrid Automata (SHA) [5] offers stochasticity and time modeling, and has been widely used in various fields such as electromechanical systems, computer simulation, automata and so on [6]. In this paper, we take the time constraints into consideration and propose to use SHA to construct the system model for verification. Considering the modeling and verification, we choose the platform UPPAAL-SMC [7] for modeling and analysis.

In order to facilitate the construction of rail transit system model, we extract the patterns of the system model for the interlocking system. Using these patterns, one only needs to fill in the appropriate parameters to customize the specific system model. The final customized system model is verified.

The rest of this paper is organized as follows. Section II presents the framework of our approach; Section III gives the method for constructing the system model patterns and Section IV clarifies pattern based system model generation method; a case study is given in section V and the related work follows in section VI. Finally, Section VII concludes the paper.

II. FRAMEWORK OF OUR APPROACH

The purpose of railway interlocking system is to control points and signal lights to prevent trains from collisions and derailments [8], while allowing its movement. The processing flow of the interlocking system is on https://github.com/wymgal/IS.git (for simplicity, we will refer it as our website in the following). Generally speaking, the physical domain of an interlocking system consists of 5 entities, *tracks*, *points*, *signal lights*, *routes*, and *interlocking table*. The *tracks* are divided into sections, and each section is associated with a circuit for detecting whether it is occupied or not. Track sections are joined by *points* which 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. *Signal lights* are placed between track sections and use red or green color to indicate proceed or stop signal respectively. *Routes* are established for authorizing a train to enter. It is often defined by *interlocking table*, which includes 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 signal.

According to the above descriptions, we get a context diagram of interlocking system, as shown in Figure 1. The system contains six entities, *Train*, *Signal Light*, *Point*, *Track*, *Interlocking Table* and the *Controller*. Except *Controller*, we name the other five entities the *environment* of the interlocking system. The *environment* interacts with the *Controller*. The interactions are the shared message between the *Controller* and the *environment* entities.

Based on the context diagram, we give a framework of our

10.18293/SEKE2018-138

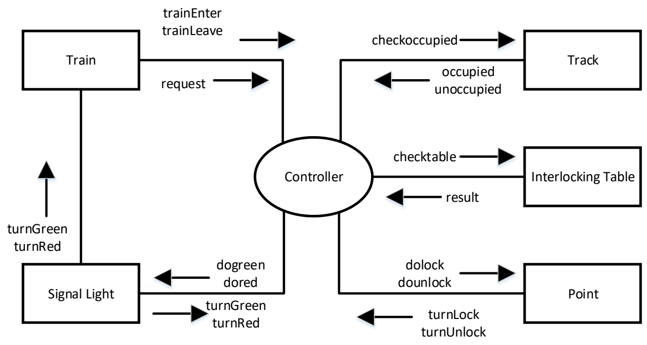


Fig.1 Context diagram of the interlocking system

approach as shown in Figure 2. Firstly, we extract the system pattern from the domain knowledge of the interlocking system. The system pattern consists of two parts: the environment patterns and the controller patterns. Based on the extracted system model, a system model is generated in combination with a model parameter table. The generated system model is simulated on the UPPAAL-SMC platform to verify the properties of the system.

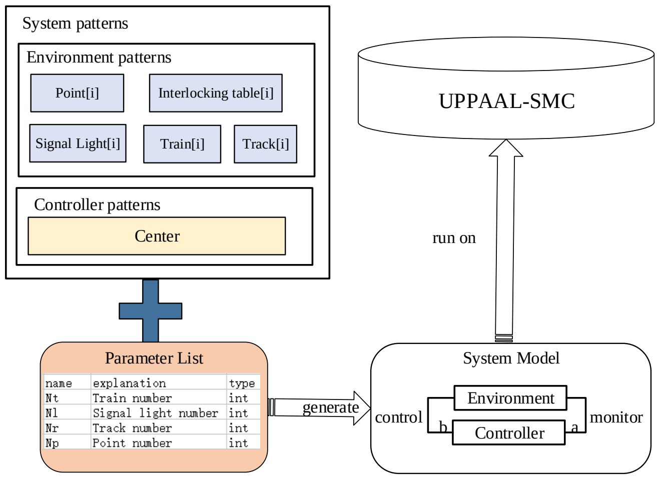


Fig.2 Framework of our approach

The environment patterns include the models of each environmental entity of the interlocking system, such as *Train*, *Point*, *Interlocking Table*, *Signal Light*, and *Track*. The controller patterns are also defined. Model parameter list is the key to realize the reusability which is given by domain experts. Using parameters in the parameter list, the extracted system patterns can be instantiated to generate a specific system model.

III. Interlocking System Model Patterns

A. *Constructing process*

Firstly, we give a 3-step process to obtain the SHA of each system entity. The three steps are constructing the basic automata, modeling faults, and adding time constraints.

Step 1: Constructing the basic automata

The process description related to the entity is found according to the system processing flow and system context diagram, including all the behaviors related to the entity.

We give a guideline to get a basic automata of each entity. Each time the entity sends or receives a message (action), the entity's automata moves from one state to another state. Therefore, an action of each entity is transformed into a state and a transition in a basic automata. The transition is an action.

Step 2: Modeling faults

In the 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 performed with different probabilities. Based on the basic automata, stochastic probability events are added to model faults.

Step 3: Adding time constraints

This step is to add time constraints on the results of step 2. Firstly, the time constraints of entities are extracted from the domain experts and expressed as <*message1*, *message2*, *<=n* time unit>. Then, the corresponding clock variable *x* are defined. The representation of the clock constraint in the automata is the time between *message1* and *message2*, that is, in the automata, the initial value of clock variable *x* on the "update" of *message1* is 0, and the inequality *x*<=*n* of the clock variable is defined in the "guard" of the *message2*.

B. *Environment entity patterns*

a) *Train Pattern*

We obtain the processing flow of the train entity from the system processing flow. When the train enters the track, it sends a request signal to the *controller* and waits for the signals of signal lights. If the train is accepted within the stipulated time, it enters the track. If rejected, it stops and waits. According to the guideline, a basic automata of the train entity is obtained.

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%, where, m + n = 100, m and n are real numbers, and are decided by domain experts.

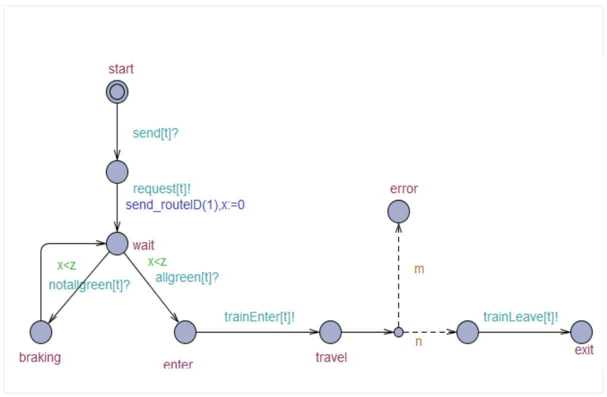
The time constraints are obtained from domain experts as follows. The not-all-lights-green signal "notallgreen" is received within specified time. The all-lights-green "allgreen" is received within specified time. The clock variable *x* is defined to indicate the waiting time for the signal light, that is, the time from sending "request" message to receiving "notallgreen" message or "allgreen" message. Therefore, the initial value of *x* on the "update" of the "request" transition is 0, and the inequality "*x*<z" (z is a constant) is used as the "guard" of transition "notallgreen" or "allgreen". Therefore, the SHA pattern of the train is obtained, as shown in Figure 3(a).

b) *Signal Light Pattern*

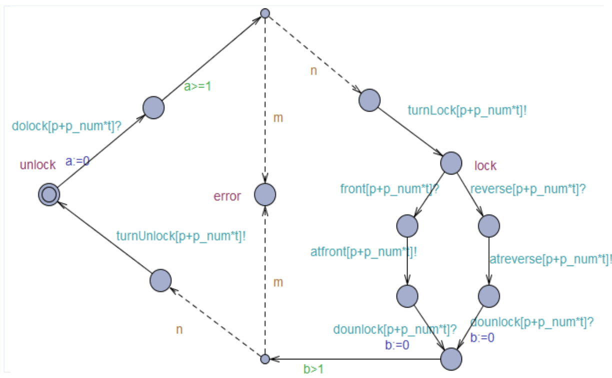
We divide the activities involved in *Signal Light* into two parts. One is *SSignalLight*, which is responsible for setting the status of the signal light. The other part is *RSignalLight*, which is responsible for inquiring the status of the signal light.

Signal Light = SSignalLight||RSignalLight

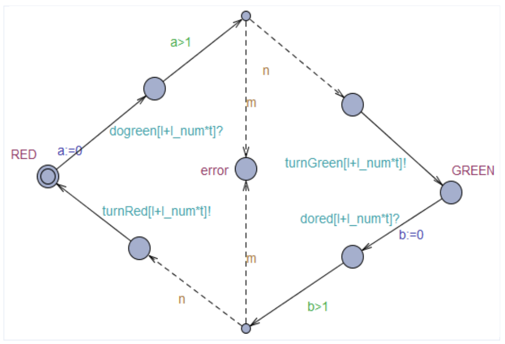
SSignalLight: We get the processing flow of the *SSignalLight* from the system processing flow. The initial state of the *Signal Light* is red. After receiving the commands of the *controller*, the signal light changes its state. According to the guideline, the



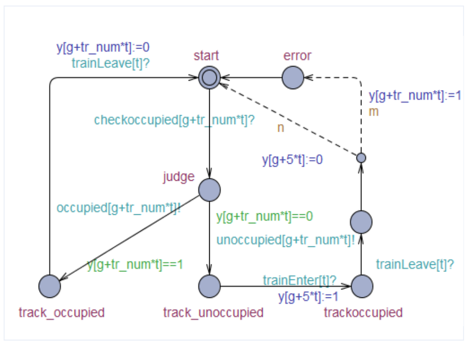
(a) Train



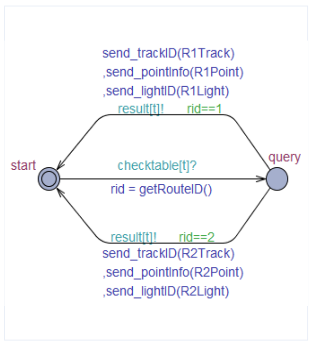
(c) Point



(b) SSignalLight



(d) STrack



(e) Interlocking Table

Fig.3 SHA patterns of environment entities

basic automata of the *SSignalLight* is obtained (see our website). 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.

The time constraints are obtained are as follows. There is a certain delay in the state change of Signal Light. A local clock *a* is given to indicate the delay time of signal light changing from red to green, and a clock *b* indicates the delay time of signal changing from green to red. The initial value on the "update" of the "dogreen" transition is 0, and the inequality "a>1" is used as the "guard" of the transition. Similarly, we can define the clock *b* in this automata. The SHA of the *SSignalLight* is thus obtained, as shown in Figure 3(b).

RSignalLight: In order to get a set of single light states, we first model one single light. According to the system processing flow, we can get the processing flow of the *RSignalLight*. According to the guideline, we build an automata for one signal light as shown in Figure 4.

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*

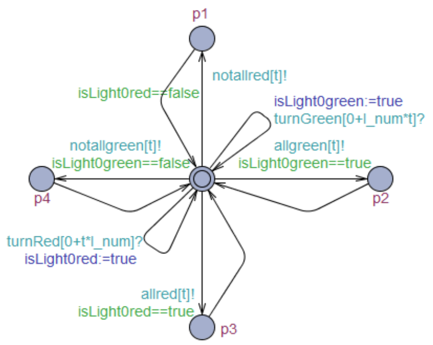


Fig.4 SHA pattern of RSignalLight for querying single signal light

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 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 our website. We do not consider the error situation and the time constraints of *RSignalLight*.

c) Point Pattern

We can get the processing flow of the point entity from the system processing flow. The initial state of the *Point* is unlocked. When the point receives commands from the *controller*, the point changes its state. There is a certain delay in the state change of the point entity. According to the guideline, a basic automata of the point is obtained as shown in our website.

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 basic 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 locked state to the unlocked state.

The time constraint for point entity is that there is a certain delay in the state change of the point. 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 initial value of *a* on the "update" of the "dolock" transition is 0, and the inequality "*a*>1" is used as the "guard" of the transition. Similarly, we can define the clock *b*. The SHA of the point is thus obtained, as shown in Figure 3 (c).

d) Track Pattern

We divide 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||RTrack**

STrack: The processing flow of the *STrack* is extracted from the system processing flow. After receiving commands from the *controller*, the track check whether it is occupied and return the results to the *controller*. According to the guideline, a basic automata is obtained (see our website).

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 similar with the case of the train entity. Without taking the time constraints of track entity into account, we do not extract time constraints. The SHA of *STrack* is obtained by the above steps, as shown in Figure 3 (d).

RTrack: It is finished in two steps. One is for one track. We build an SHA model for one track, as shown in our website. 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.

e) Interlocking Table Pattern

According to the system processing flow, the processing flow of the Interlocking Table entity is obtained. After receiving the query command from the controller, the interlocking table queries the related information and returns results. According to the guideline, the automata of the interlocking table is obtained as shown in Figure 3(e). We do not consider the error situation and time constraints of the interlocking table entity.

C. 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 the interlocking table. The other part called *Submodules*, which is in charge of controlling each track, point, signal light and train respectively. So we define:

Controller=Center||Submodules

Submodules=CTrack||CPoint||CSignalLight||Dispatcher

Center: The processing scenario of the *Center* is extracted as expressed in our website. According to the guideline, the basic automata of the *Center* is obtained as shown in Figure 5.

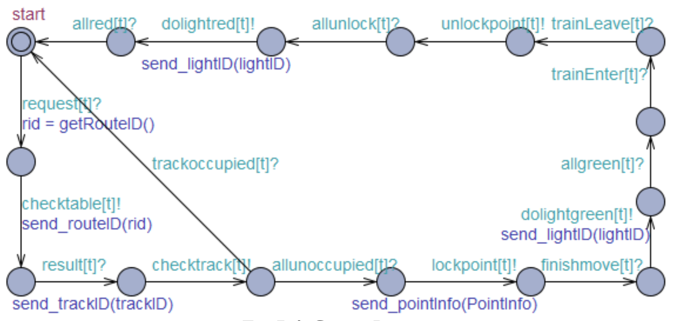


Fig.5 A Center Pattern

In the global declaration, we define 8 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 our website.

CTrack, ***CPoint***, and ***CSignalLight***: 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 construction process of *CTrack*, *CPoint*, *CSignalLight* are similar to the construction process of *RSignalLight*. Similarly, we can get the *CTrack*, *CPoint* and *CSignalLight* (see our website).

Dispatcher: It is responsible for sending dispatching instructions to control different trains entering the track at different time. How many trains to be sent is decided by detailed number. So we cannot give a graph pattern here. But the basic sentence can be recording as: for each train *i*, we add a state with the message of "send[i-1]!" and a transition with the massage of "trainEnter[i-1]?".

IV. A SYSTEM MODELING METHOD BASED ON PATTERNS

We give a 5-step process to model the system.

Step 1: Declaring all the models in the system Through analysis, we define that the system is composed by 12 models, so we make the following declaration:

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

Step 2: Setting model instantiations 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;
```

Step 3: Reusing patterns The parameters in each pattern can be modified according to specific situations. According to the number of entities, the *RSignalLight* and *RTrack* model can be instantiated. Reuse the *Center* pattern, create the SHA model of the *Center*, and instantiate the SHA models of the *Controller* submodules.

Step 4: Defining the system interactions The interactions 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 our website.

Step 5: Declaring system variables The global variables in 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, t_num=Nt;
                                    // the number of instantiations of each entity
int route_id, trackID[Nr], PointInfo[Np][Np], lightID[Nl];
                                    // the variables used by the functions
```

V. CASE STUDY

In this paper, we use a case which interlocking table is shown in Table I. 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.

A. Defining the system

According to the process, we declare the 12 models as listed in Section IV. From Table I, 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:

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

Reuse the patterns of *Train*, *Signal Light*, *Point*, *Track* and *Interlocking Table*, 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 and *RTrack* model are in our website. Finally the *Controller* model is constructed. We reuse the *Center* pattern, and build *CTrack*, *CPoint*, *CSignalLight* according to their numbers. Reuse the *Dispatcher*. 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* model is shown in our website, and the rest of the *Controller* submodules are displayed in our website. After this, put *Nt*=2, *Nl*=5, *Nr*=5, *Np*=2 into the global declaration as follows. Finally the system is built.

```
int y[10] ={0,0,0,0,0,0,0,0,0,0};
const int l_num=5,p_num=2,tr_num=5, t_num=2;
int route_id,  trackID[5],  PointInfo[2][2],  lightID[5];
```

B. Simulation and verification

UPPAAL uses BNF syntax to describe the security requirements of the system, and the modeler can verify the related properties of the system according to the different design requirements. This paper only considers the part design requirements, including the system model is not deadlock (1), the *Signal Light* model can enter the green light state (2), and the Monitor model can detect the error of the system into the warning state (3). They are represented as follows:

```
A[] not deadlock  (1)
E<> Light.GREEN (2)
E<> Monitor.warning (3)
```

The simulation model of the system in the UPPAAL platform, after repeated simulation and observation, meet the above three design requirements: the preliminary determination of each state of the model is deadlock and reachable; signal light can enter the green state; the monitor can determine the corresponding error and enter warning state.

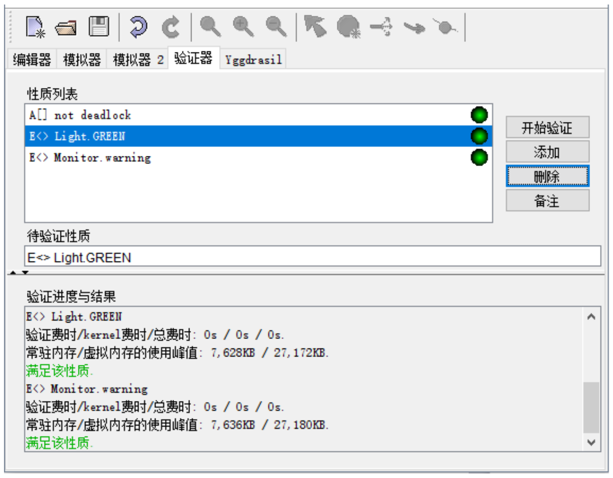
As a result, the model meets the requirements of the system, the expected security requirements, and ensures the security and correctness of the model. The following results are obtained, as shown in Figure 6.

VI. RELATED WORK

In order to ensure the correctness and safety of the system, there are many works for modeling the real time and fault stochastic characteristics. Formal methods are widely used in modelling and analysis [10,11], such as Timed Automata [12], Petri net [13], Z language [14] and so on. For example, Wang uses the time automata theory to model and verify the railway station signal interlocking system and the interlocking route control process [15]. Hei et al. use Petri net to model and verify the distributed control interlocking system [16]. Tiejiang Wang describes the security requirements of computer interlocking software in Z language [17].

TABLE I. INTERLOCKING TABLE

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



性质列表

A[] not deadlock ●
E<> Light.GREEN ●
E<> Monitor.warning ●

待验证性质

E<> Light.GREEN

验证进度与结果

```
E<> Light.GREEN
验证费时/kernel费时/总费时: 0s / 0s / 0s.
常驻内存/虚拟内存的使用峰值: 7,628KB / 27,172KB.
满足该性质.
E<> Monitor.warning
验证费时/kernel费时/总费时: 0s / 0s / 0s.
常驻内存/虚拟内存的使用峰值: 7,636KB / 27,180KB.
满足该性质.
```

Fig.6 Verification results

However, these formal methods have shortcomings for modeling the interlocking system. Although timed automata consider the time requirement of the system with real-time characteristics, it has no stochastic and cannot model the complex system's stochastic faults. The Petri net and Z languages do not consider the time and stochasticity of the system when modeling. Compared with these methods, our approach of using stochastic hybrid automata can consider the stochastic and real time characteristics of interlocking system, which is more suitable for interlocking system modeling.

Another related work is pattern based modeling. Although the UPPAAL-SMC provides the train gate example, but it does not have patterns. There are many efforts in pattern based modeling. For example, Wu et al. propose a traffic pattern modeling approach for the urban intersection[18]. Zhang et al. propose an observer-pattern modeling method to eliminate the time-variance effect for two-stage boost inverter [19]. However, these pattern-based system modeling work is rarely related to the interlocking system. In addition, many modeling languages used in them do not pay particular attention to time constraints and stochasticity.

VII. CONCLUSION AND FUTURE WORK

As one of the core systems of rail transportation system, the interlocking system ensures the safety of trains. Based on the SHA, this paper presents the modeling of the interlocking system using patterns. The modeled system could be analyzed using simulation and verification technology. The main contributions of this paper include:

(1) The 12 model patterns for interlocking systems are extracted covering 6 entities consisting of train, signal light, point, track, interlocking table and controller;

(2) An approach for reusing these patterns to construct an exact interlocking system model is proposed. Using this approach, novices of SHA could be quickly build a system for further analysis.

The next step work is to consider more fault types and apply this model to accident prediction.

ACKNOWLEDGMENTS

This work was supported by the Natural Science Foundation of China (No. 61472140) and Defense Industrial Technology Development Program (JCKY2016212B004-2).

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] Baofeng Xie. Status and development of computer interlocking system of station[J]. Transportation system engineering and information, 2004, 4(4):86-90.

[3] EN501 28C. Railway application-SoRware for railway control and protectionsystem[J]. 2000.

[4] Krishna,C . M . Real Time Systems[M] // Real-Time Systems. McGraw-Hill Higher Education,1 996：3-5.

[5] Bortolussi L, Policriti A. Stochastic Programs and Hybrid Automata for (Biological) Modeling[C]// Conference on Computability in Europe: Mathematical Theory and Computational Practice. Springer-Verlag, 2009:37-48.

[6] Bemporad A, Cairano S D. Optimal Control of Discrete Hybrid Stochastic Automata[J]. IEEE Transactions on Automatic Control, 2005, 56(6):1307-1321.

[7] Bulychev P, David A, Larsen K G, et al. UPPAAL-SMC: Statistical Model Checking for Priced Timed Automata[J]. Electronic Proceedings in Theoretical Computer Science, 2012, 85.

[8] Hartonas-Garmhausen V, Cimatti A, Clarke E, et al. Verification of a safety-critical railway interlocking system with real-time constraints[J]. Science of Computer Programming, 2000, 36(1):53-64.

[9] Zengming Yu, Zhengdong Liu. The change of interlocking function in communication based train control system[J]. Railway Operation Technology,2011, 17(4):13-15.

[10] Amir Pnueli. Formal Verification: All Qucstiom and Some Answers.CS Leading Teachers Course, WIS, May 9,1999.

[11] Wing J M. A specifier’S introduction to formal methods [J]. Computer, 1990, 23(23)：8-22.

[12] Alur R. Timed Automata[J]. Lecture Notes in Computer Science, 1999, 126(94):183--235.

[13] Yang yang, Ming Pan, Meifang He. Formal specification process of the interlocking software with Petri nets[J]. China Railway Sciences,2002,23(3):49-54.

[14] Tiantian Tang. Research on Application of Software Reliability Design Method in Computer Interlocking System[D]. Hefei Polytechnic University,2004.

[15] Guanning Wang. Modeling and Verification of Interlocking Route Control Process Based on UPPAAL[D]. Beijing Jiaotong University,2009.

[16] Hei X, Takahashi S, Hideo N. Toward developing a Decentralized Railway Signalling System Using Petri Nets[C]// Robotics, Automation and Mechatronics, 2008 IEEE Conference on. IEEE, 2008:851-855.

[17] Tiejiang Wang, Meng Li. Z specification for computer interlocking software [J]. ChinaRailway Society, 2003, 25(4):62-66.

[18] Wu C E, Yang W Y, Ting H C, et al. Traffic pattern modeling, trajectory classification and vehicle tracking within urban intersections[C]// International Smart Cities Conference. 2017:1-6.

[19] Zhang H, Li W, Ding H, et al. Observer-Pattern Modeling and Nonlinear Modal Analysis of Two-stage Boost Inverter[J]. IEEE Transactions on Power Electronics, 2017, PP(99):1-1.