ICS 2002

Workshop on Databases and Software Engineering

Title :

A Dual Language Approach to Software Formal Specifications and Safety Analysis

Abstract

In this paper we present a systematic approach to apply Statecharts modeling and analysis to safety-critical systems. Procedures are devised to first convert Startchart specifications to fault trees for hazard analysis, and then analyze the constructed fault trees to generate accident sequences and express them in UML sequence diagrams. Thus, incorrect or hazardous states and scenarios can be identified so as to assist the designer to modify the system. Our systematic approach makes the conventional subjective fault tree construction objective and repeatable. Thus, safety analysis using formal specifications can be done automatically. Furthermore, we convert the statechart specifications into temporal logic for safety or correctness proof. With the dual specification languages, our method takes the advantages of statecharts' visual understandability and temporal logic's proof clarity. A railroad-crossing case is given to demonstrate the feasibility and effectiveness of our method.

Keywords: statechart, temporal logic, safety analysis, fault tree analysis, sequence diagram.

Authors: Chin-Feng Fan[1], Chia-Ho Sun[1], and Swu Yih[2]
Affiliation: [1]Computer Engineering & Science Dept., Yuan-Ze University, Taiwan
[2] I&C Dept., Nuclear Energy Research, Lung-Tang, Taiwan

Address: 135 Far East Road, Chung-Li, Taiwan 320
E-mail: csfanc@saturn.yzu.edu.tw
Fax : (03)4638850
Contact: Chin-Feng Fan

A Dual Language Approach to Software Formal Specifications and Safety Analysis*

Chin-Feng Fan[1], Chia-Ho Sun[1], and Swu Yih[2]

[1] Computer Engineering & Science Dept.
Yuan-Ze University, Chung-Li, Taiwan
[2] I&C Dept., Nuclear Energy Research, Lung-Tang, Taiwan

Abstract

In this paper we present a systematic approach to apply Statecharts modeling and analysis to safety-critical systems. Procedures are devised to first convert Startchart specifications to fault trees for hazard analysis, and then analyze the constructed fault trees to generate accident sequences and express them in UML sequence diagrams. Thus, incorrect or hazardous states and scenarios can be identified so as to assist the designer to modify the system. Our systematic approach makes the conventional subjective fault tree construction objective and repeatable. Thus, safety analysis using formal specifications can be done automatically. Furthermore, we convert the statechart specifications into temporal logic for safety or correctness proof. With the dual specification languages, our method takes the advantages of statecharts' visual understandability and temporal logic's proof clarity. A railroad-crossing case is given to demonstrate the feasibility and effectiveness of our method.

Keywords: statechart, temporal logic, safety analysis, fault tree analysis, sequence diagram.

1. Introduction

Formal specifications are important to safety-critical software since formal specifications facilitate safety analysis and verification. A dual language approach originally refers to one language in the computer executable programming language; the second language is the specification (or assertion) language. Manna and Pnueli [10] extended the dual–language approach to incorporate fair transition systems and temporal logic framework together in specification. This paper extends the dual language concept to include both a model-based and a property-based specification

* This research is supported in part by Atomic Energy Research and National Science Council under the grant number NSC91-2623-7-155-001.

language; however, these two languages are used for different purposes. We utilize the visual understanding of Statecharts, a model-based language, to facilitate subsequent hazard analysis; while, we utilize the ease of verification of temporal logic, a property-based language, to perform correctness and safety proof. Systematic conversion from statecharts to temporal logic is also presented. We apply fault trees to safety analysis of a safety-critical system. However, different from the current subjective fault tree construction, we propose an algorithmic approach to construct fault trees from statechart specifications. To obtain better understanding of the potential accident sequences, we then convert the constructed fault tree into UML sequence diagrams so as to facilitate design modification. Once the correct statecharts are converted to temporal logic, safety or correctness verification can then be performed. Thus, our approach consists of the following steps:

1. Use statecharts to model the system.
2. Construct fault tree based on state chart specifications
3. Express identified accident sequences in UML sequence diagrams.
4. Modify system design if needed
5. Convert the correct statecharts into temporal logic specifications
6. Verification of temporal logic specifications.

The major virtue of our method is that most of these steps can be carried out systematically and objectively.

2. Background

Dual languages in this paper refer to statechart and temporal logic specifications. Thus, we will first briefly introduce them.

The statechart was introduced by Harel [4]. It is an extension of conventional finite-state machines. The Statechart adds structures, hierarchies, and concurrency to the conventional finite-state machines so that it is appropriate to represent behavior of complex control systems. A statechart includes states, super-states, concurrent states, and events as well as transitions. A transition can be labeled as α [C]/ β where α is a trigger event, and C is a condition that guards the transition from being taken unless it is true when α occurs, and β is an action that is carried out if the transition is taken. Also, at the entrance and exit points of a state, actions can be performed; this will be denoted as “entering/action” and “exiting/action” in side the state box. Statecharts use broadcast fashion for message passing. However, in this paper for clarity’s sake, we use channels to pass messages. Send and Receive are assumed to be two predefined events with the format “Send or Receive (channel, message)” .

Besides the conventional propositional connectives and quantifiers, Temporal Logic [3,9, 10, 11] introduces special temporal operators including the following:

[ ] P：In all future states, the state predicate P is true.

◇ P：In some future state, the state predicate P is true.

( )P：In the next state, the state predicate P is true.

[－]P：In all the past states, the state predicate P is true.

＜－＞P：In some past state, the state predicate P is true.

（－）P：In the previous state, P is true.

S：Since operator

U、 *u* ：until operator

W：unless、waiting for

For safety analysis, we use fault trees. Fault tree analysis (FTA) is widely used in aerospace, nuclear and electronics industries for system safety analysis. FTA is a means for analyzing causes of hazards. It uses logic AND, OR gates to describe the combinations of individual faults that can constitute a hazardous event. A square box in a fault tree indicates an event resulting from a combination of events through a logic gate. A circle indicates a terminal fault event. A diamond denotes an event that is not developed further. This is shown in Fig. 1.There are several procedures proposed to automatically synthesize a fault tree, but only for systems consisting purely of hardware elements. Software fault trees at instruction level were proposed by Nancy Leveson [7]; they can be partially generated automatically. However, for software specification and design level fault analysis, or analysis of systems with both hardware and software [1,2], current fault tree construction cannot be fully automatic. These fault trees are usually constructed subjectively; i.e., different persons may construct different fault trees for the same top event. Thus, a systematic construction method is urgently needed.

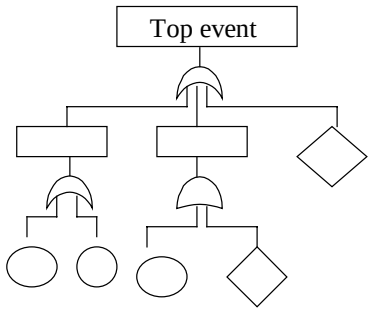

Fig. 1. A fault tree

In conventional safety analysis, once fault trees have been constructed, event

trees [12] can be built to enumerate all possible combination of event sequences. Yet event trees may have the exponential explosive problem. We may be only interested in some fault sequences, and moreover, we may be particularly interacted in the interaction between subsystems. Thus, a UML sequence diagram can express such accident sequences in a much clearer fashion than an event tree. In our method, UML sequence diagrams for potential faulty scenarios are generated from the constructed fault trees to assist designers for system modification.

3. Our systematic approach

Our approach consists of the steps shown in Fig. 2. Steps 2, 3, and 5 can be done systematically. Thus, details of these steps are described below.

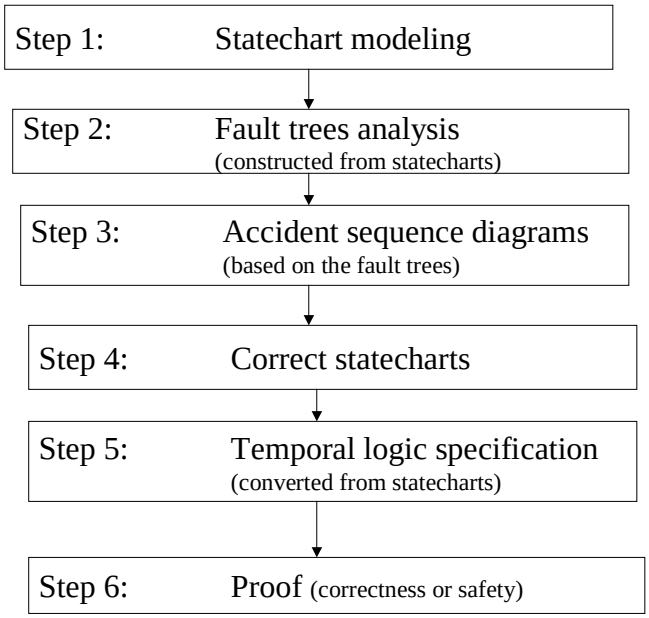

Fig.2. Our approach steps

3.1. Construct a fault tree based on statechart specifications

To construct a fault tree, an undesired top event (a hazardous situation) should be identified first. Then, the relevant subsystems, states, and transitions leading to the concerned states are then located. Basically, the process of generating lower level causes in a fault tree is a tracing loop -- tracing from the related transition backwards in the statecharts to its action, trigger events and conditions. The loop will terminate

when it reaches a terminal event, which is hardware failure, an external event, or when there are no more new nodes. The tracing process is shown in Fig. 3. We trace from the abnormal events concerning action A backwards, along the path from transition *t1* to *t2*; all the triggering events and conditions as well as action in the traced path may be abnormal. With dedicated channels, the tracing is easier. There are two kinds of abnormality that we deal with in our fault trees. They are Case 1: "what should have happened, but it hasn't happened", and Case 2: "what shouldn't have happened, but it has happened". The former is omission errors; the latter is commission errors. For each omission error, we consider the following causes:

1. the trigger event did not occur or condition was not true,
2. the trigger event and condition were true, but transition action failed
3. the trigger event occurred but was not received.

For commission errors, we consider the following cases:

1. the trigger event and condition abnormally occurred or were true
2. the trigger event and condition did not happen, but the action has abnormally been taken.

For simplicity's sake, we use "triggering event" representing "triggering events and their related conditions". Fig. 4 is the conceptual diagram of the constructed fault trees. Each undesired event is a root of a fault tree. The algorithm constructing a fault tree based on statecharts is given below:

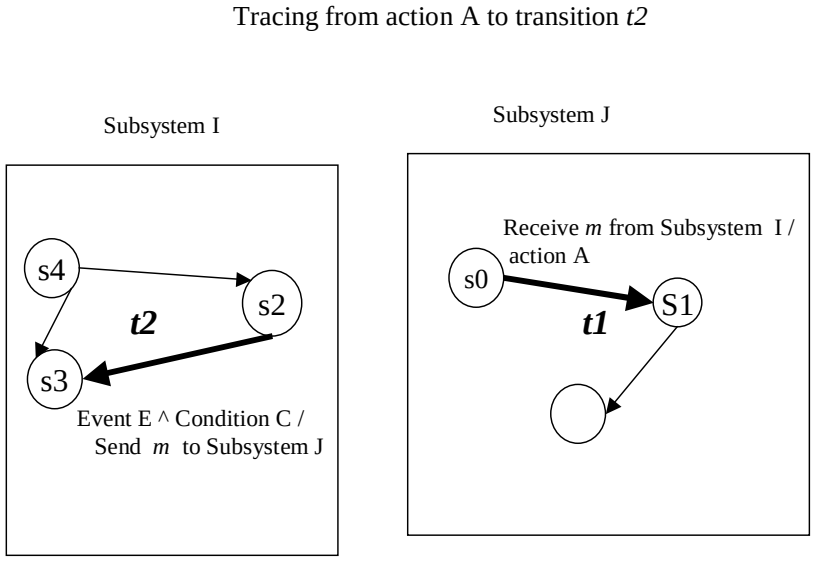

Fig. 3 Tracing in a statechart

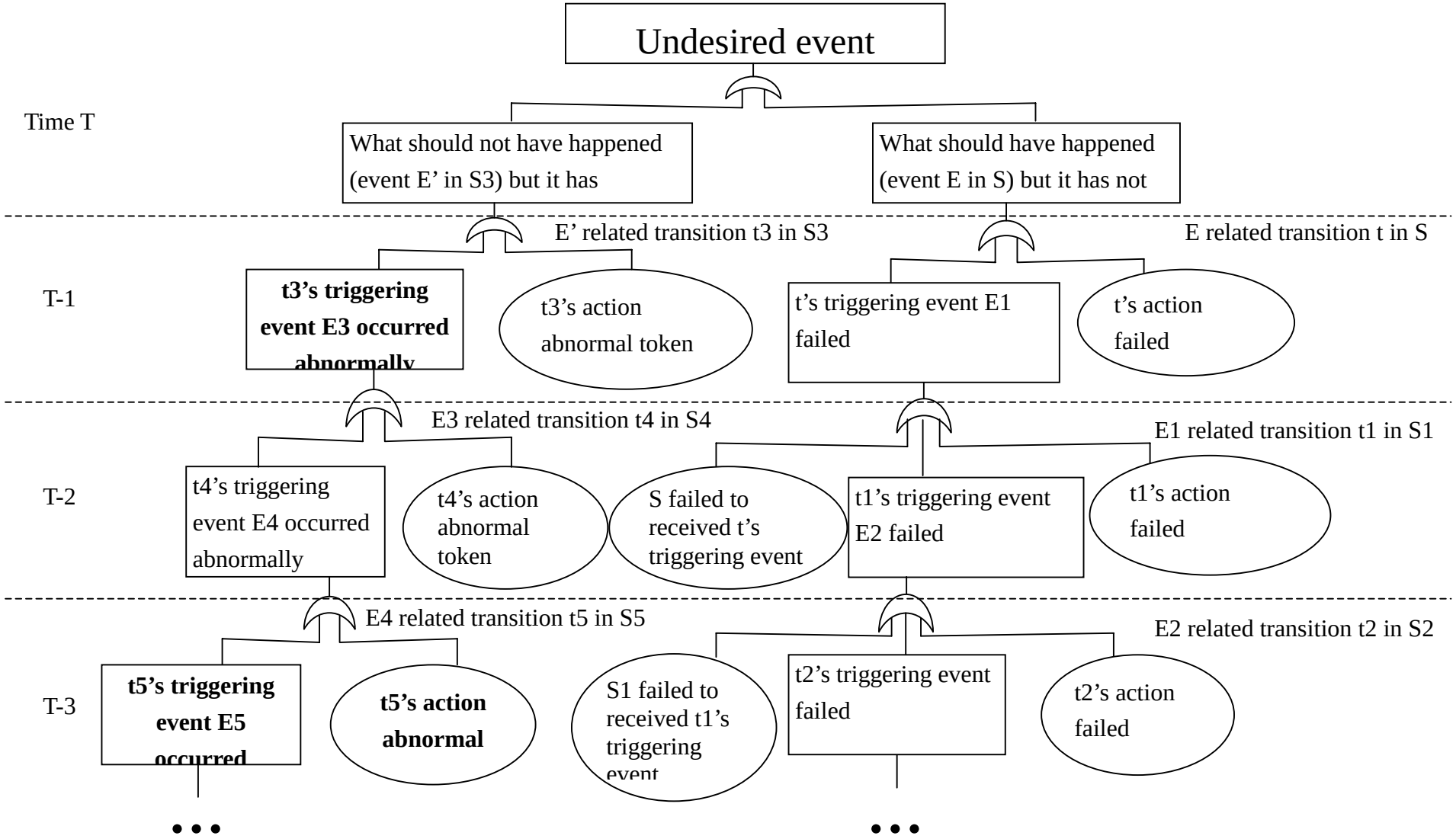

Fig. 4 Fault tree generated from statechart specifications

1. For each undesired event Do
2. Begin
   identify its top level causes
   Ci = (assumption A, erroneous event E) at Time T including
   Case 1="what should have happened but it has not happened"
   Case 2="what shouldn't have happened but it has happened"
3. Trace back in statecharts and identify the subsystem S, its state, and transition t related to E
4. Assign next level OR causes to be:
   Case 1: (t's action failed) OR (t's triggering event failed)
   Case 2: (t's action abnormally taken)
   OR (t's triggering event abnormally occurred)
5. FOR each non-terminated cause C LOOP
   // assume C related to transition t0 in subsystem S0
6. Trace the statecharts back to t0's triggering
   subsystem S1 and related transition t1
7. Generate the fault tree's next level OR causes to be:
   Case 1: (S0 failed to receive triggering message)
   OR (t1's action failed)
   OR (t1's triggering event failed)
   Case 2: (t1's action abnormally taken)
   OR (t1's triggering event abnormally occurred)
8. Until the cause is hardware failure or an external event or no more new causes
9. END FOR

In the above construction, we considered both hardware and software failure, including sensor errors, device failure, message passing , message receiving problems, and software problems. Our fault trees may not ensure completeness in all situations; however, the generation process is systematic and automatic.

3.2. Construct UML Sequence diagram from fault trees

To further express the accident sequences and subsystem interaction identified by the fault trees, UML sequence diagrams are then generated. However, sequence diagrams of normal scenarios are first built as basic templates. Then, the abnormal parts of the accident scenarios will be composed from the above fault trees and incorporated into their related basic templates. Currently, we only consider scenarios with a single failure. Fig. 5 demonstrates how to construct a faulty portion of a sequence diagram from a fault tree. In a fault tree, leaves conflicting with the top

event are trimmed first. The generation of faulty sequences starts from the bottom level. We only consider single-failure cases for the time being; thus, one leaf is for one faulty scenario. Note that our fault trees have two major branches (omission and commission errors). For each branch, each of our fault trees has at most one non-terminated node at each level. First, a sequence diagram for a normal scenario is generated as a template. A faulty scenario is composed by negating all the lower level non-terminated causes in the fault tree until the concerned leaf; then the path from this leaf up to the root will be taken. Take Fig. 5 for example, our algorithm may generate the abnormal scenario for node A by considering A^B^C^E^F; for node X, we consider the path $\neg$A^ $\neg$B^X^C^E^F; for node D, we consider the path $\neg$A^ $\neg$B^$\neg$C^D^E^F. These faulty paths are thus generated and then incorporated into normal sequence diagram templates to compose complete accident sequences.

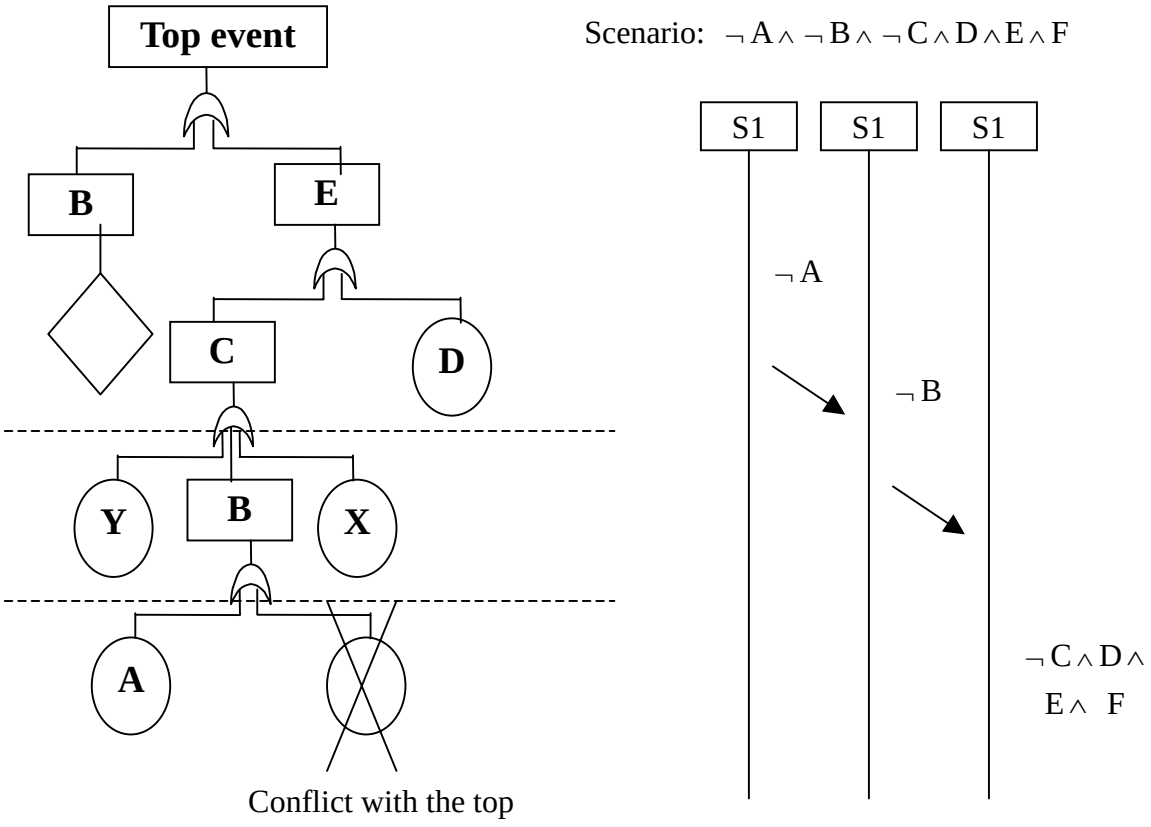

Fig. 5 Sequence diagram generation

3.3 Conversion from Statecharts to Temporal Logic

Notations are easier to manipulate in proof process than graphs. Thus, we may systematically convert Statecharts to temporal logic to facilitate the verification

process. To enhance temporal logic's readability, we categorize temporal logic rules into the following five groups: General Rules, Backward Rules, Forward Rules, Transition Rules, and Temporal Constraint Rules. The following notations are used:

State: (subsystem = state)
Transition: (subsystem: state1 → state 2)
Event : Each event has a system-wise unique name.
Channel: from subsystem 1 to subsystem 2 defined as (subsystem1_subsystem2)
Message passing : Send (channel, message), Receive(channel, message).

The five groups of temporal rules are explained below:

1. *General Rules*： Define all subsystems' possible states.
2. *Backward Rules*： Describe each state's preceding states.
3. *Forward Rules*：Describe each state's next states.
4. *Transition Rules*：For each transition, describe its trigger event, condition and its action. For example, Subsystem A changes from State a to State b, its transition rule is defined as:
   (A=a) ∧(trigger event) ∧ (guard condition)⇒ ( )((A：a →b) ∧ (output action))
   Also, at a state's entering or exiting points, action may be taken:
   Subsystem = entering (State) ⇒ ( ) action
   Subsystem = exiting (State) ⇒ ( ) action
5. *Constraint Rules*： Extra constraints or properties, if needed, can be added.
   Also, to expedite proof, preconditions or constraints for some concerned transitions can be traced back to its triggering transition or conditions.

Besides the possible extra properties added in category 5, the above conversion can be generated systematically from statecharts. Without those extra properties, these temporal rules are actually equivalent to the original statecharts. However, the rule format facilitates and expedites verification process.

Take Fig. 3 for example, part of the generated rules is as follows:

1. General rule: [ ](I=S2 || S3 || S4)
2. Backward rule: I=S3 => (-) (I=S2 || S3 ||S4)
3. Forward rule: I=S2 => ( ) (I=S2 || S3)
4. Transition rule: (I= S2) ∧ E ∧ C => (I: S2-> S3) ∧ SEND(I_J, m)
5. Constraint rule: (J: S0->S1) =><->(I: S2→ S3)

3. Case Study: railroad crossing example

We use a railroad crossing example as our case study to demonstrate how to use our proposed method. There are three subsystems in this example: a train, a controller, and a gate. The train has traveling, approach, ingate, and exit states; the controller keeps in working state; while the gate's state is either up or down. When the train approaches the intersection, it sends a message to inform the controller; controller then commands the gate to lower it. When the train exits, it will again send a message to inform the controller, which then issues a command to lift the gate. The original statecharts are shown in Fig. 6. In the example, trigger events include trainapproach, ingate, leave, send, and receive.

To check whether the current system design is correct or safe; a fault tree following the algorithm in Section 3.1. is constructed. It is shown in Fig. 7 with the top event "train is ingate and the gate is up". The next level reasons include two branches: Case 1: the gate was not put down; and Case 2: the gate was lifted up. Each node in the fault tree is expressed as 2-tuple, (subsystem: trigger message, action). Also, to simplify the expression, messages are indicated as "sender.message" in the diagram. The numbers shown in the un-trimmed leaves are the case numbers used in faulty sequence diagrams (Fig. 9) later.

The sequence diagram of a normal scenario is shown in Fig. 8. There are 8 untrimmed leaves in the built fault tree; thus, eight potential faulty partial scenarios may be generated and put in the context of the normal sequence diagram. These faulty single-failure scenarios are shown in Fig. 9. Among these 8 cases, diagrams labeled as Fig. 9-1 and Fig.9-6 are due to sensor problems, Fig. 9-3 and Fig. 9-7 are caused by controller software problems, Fig. 9-5 and Fig. 9-8 are due to gate hardware malfunction, and Fig. 9-2 and Fig.9-4 are due to message receiving problems.

There are many possible ways to modify this system to ensure its safety. Hardware design diversity or redundancy and software n-version programming are among such possible solutions. However, if we only consider software controller

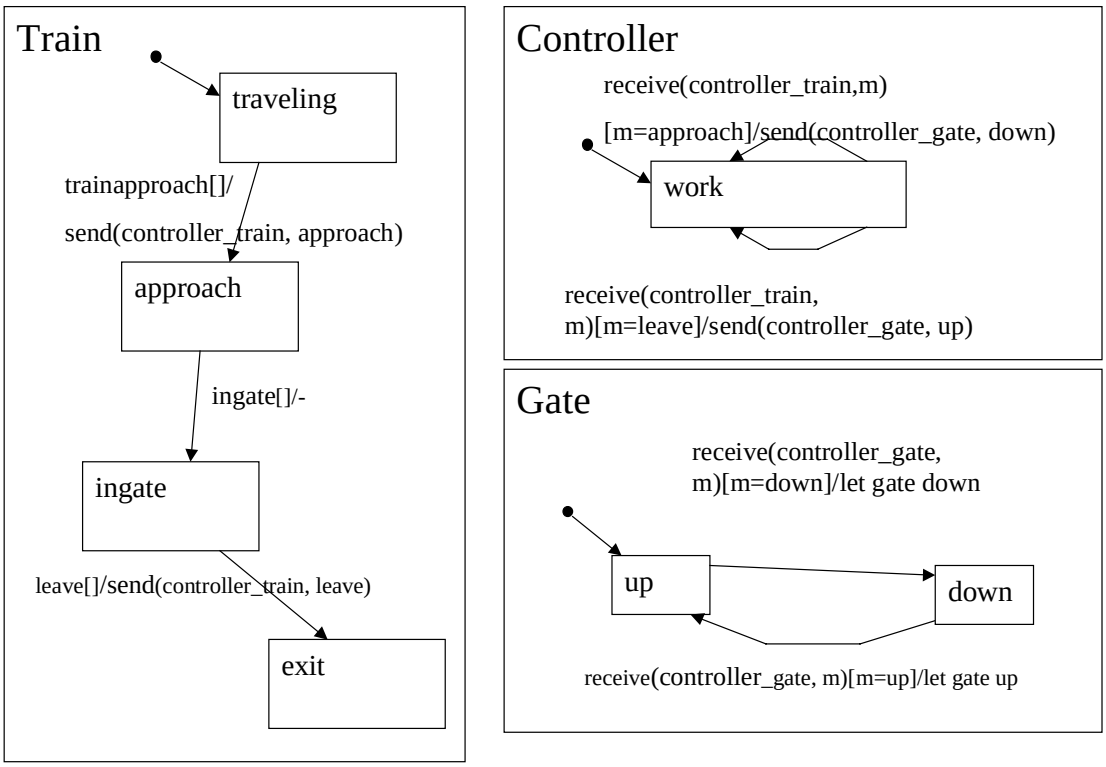

Fig. 6. Original statecharts

problems, we may solve controller's problems by using hardware interlock to ensure that the train is ingate if and only if the gate is down. The modified statecharts are shown in Fig. 10. Now the train has a new state, the interlock state. Immediately after the gate is entering the down state, a message will be sent to the train; this will trigger the train to change from the interlock state to the ingate state. Thus, assuming hardware has no problem, this solution solve the concerned risk. The sequence diagram for this modified version is shown in Fig. 11.

To facilitate proof, we then convert the modified startchart specifications into temporal logic. These rules are shown in Fig. 12. In this case, no extra properties are added into the temporal rules. Thus, these temporal rules are equivalent to statecharts. Note that the redundant backward and forward rules as well as constraint rules are designed to expedite the proof process. Assume that hardware systems are correct, we would like to prove that the controller functions correctly. That is, we prove that the situation (train=ingate $\wedge$ gate=up) will not hold. The proof using backward search is shown in Fig. 13.

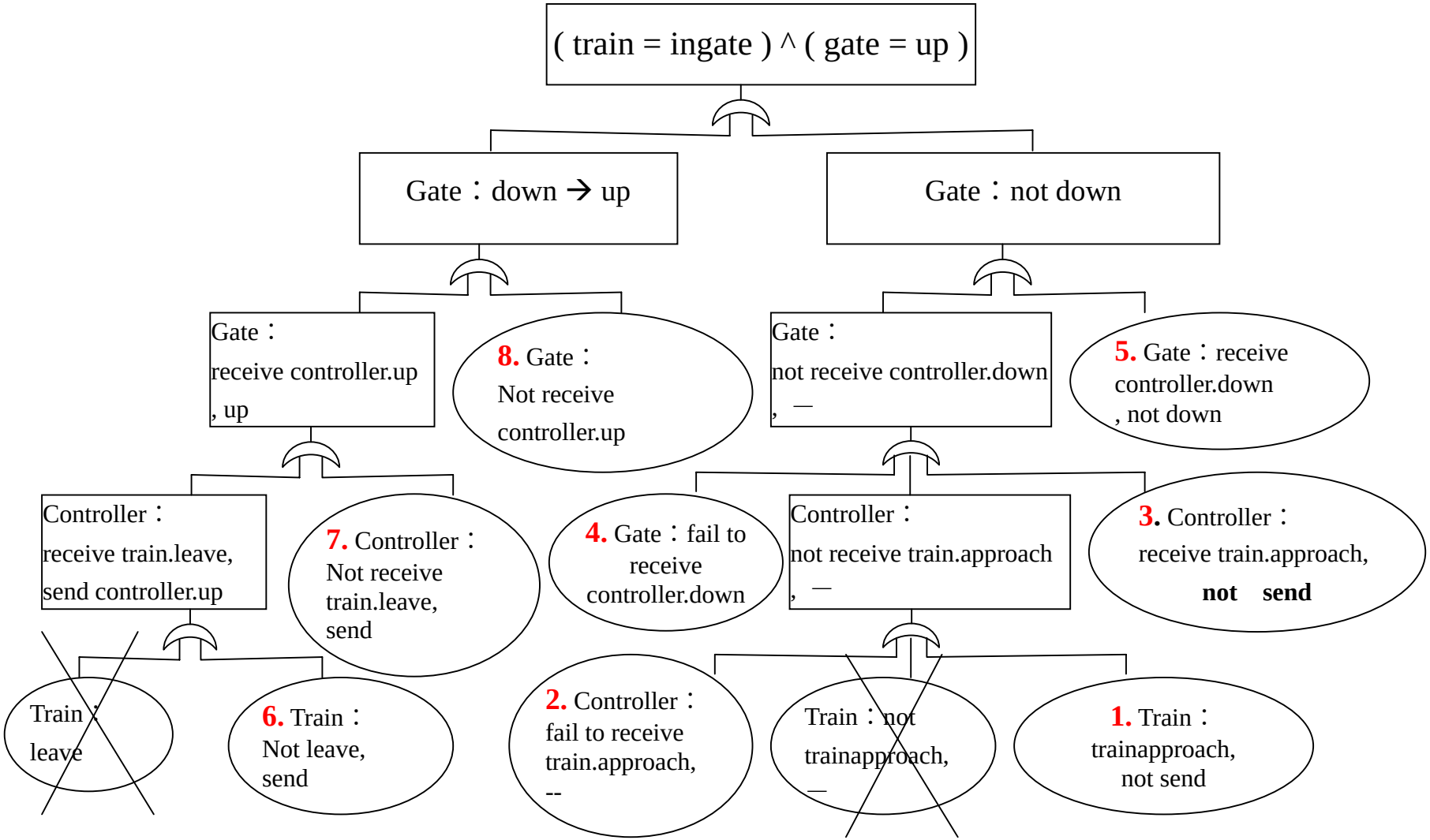

Fig. 7. The Fault tree systematically generated from statechart specifications

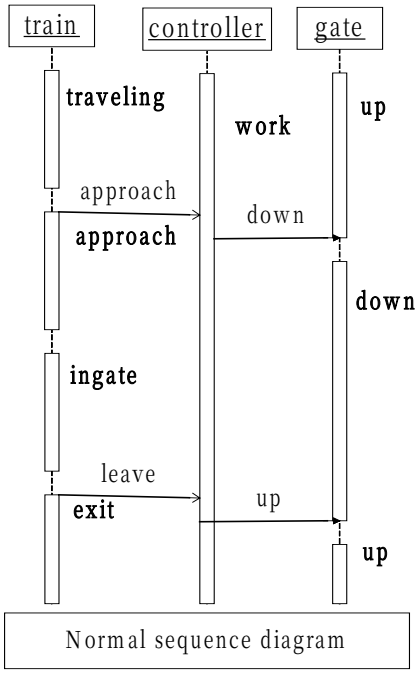

Fig. 8 Sequence diagram template

5. Conclusion

We have presented a systematic construction method to convert a formal specification to fault trees and then generate UML sequence diagrams for accident scenarios identified by the fault trees. This systematic approach makes the conventional subjective fault tree construction objective and repeatable. UML sequence diagrams are more understandable than conventional event trees in expressing interaction among subsystems and devices. Thus, safety analysis using formal specifications can be automatically performed. Moreover, to obtain the advantages of both model-based and property-based specifications, we have developed a systematic way to convert statechart specifications to temporal logic specifications so as to facilitate verification process. A railroad crossing case using the proposed steps to ensure its software safety has been reported. It demonstrated the feasibility and effectiveness of our method. The virtue of our approach lies in that it is procedural and thus repeatable.

Our method can be applied to a statecharts with multiple layers. We have also applied the same approach to a digital reactor protection system to verify its safety. This case study will be reported shortly.

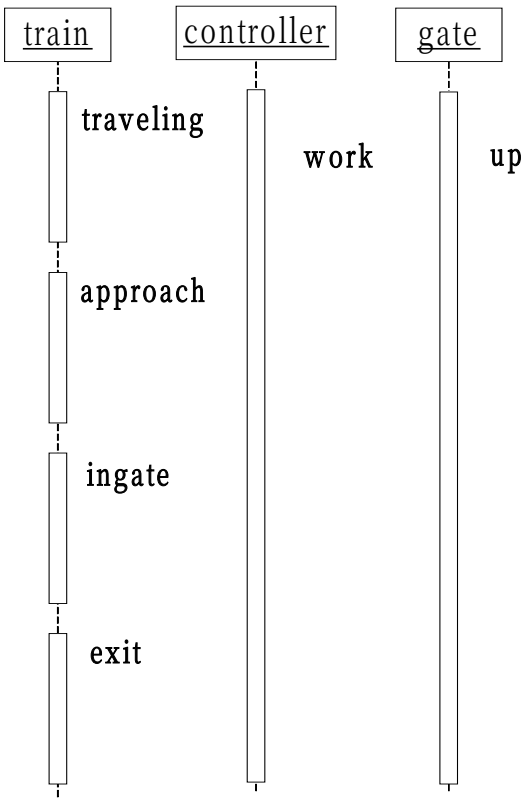

Fig. 9-1: Sensor errors

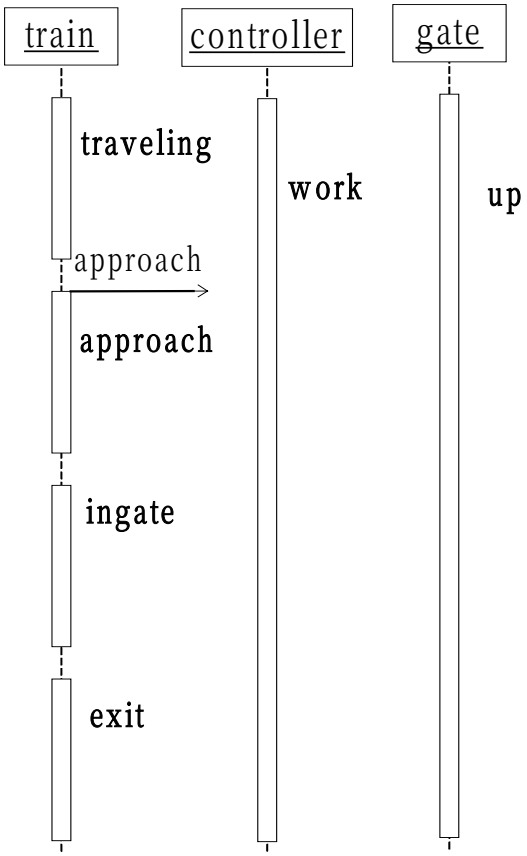

Fig. 9-2: Message receiving errors

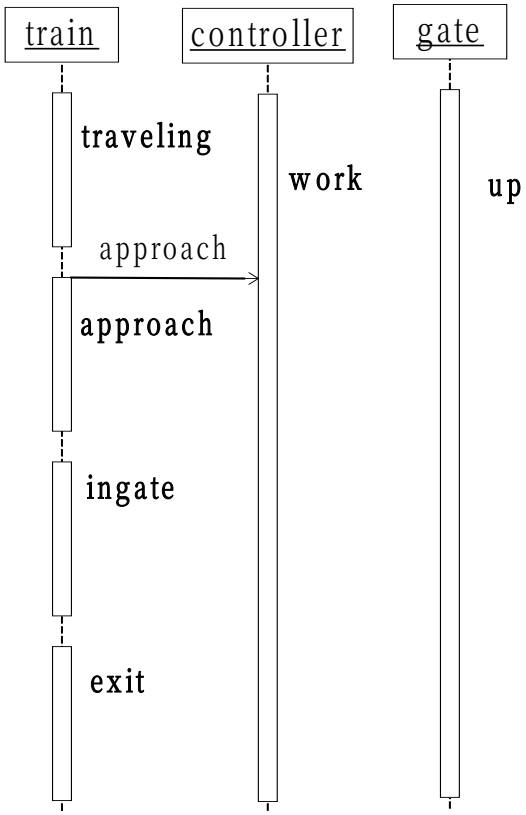

Fig. 9-3: Controller software errors

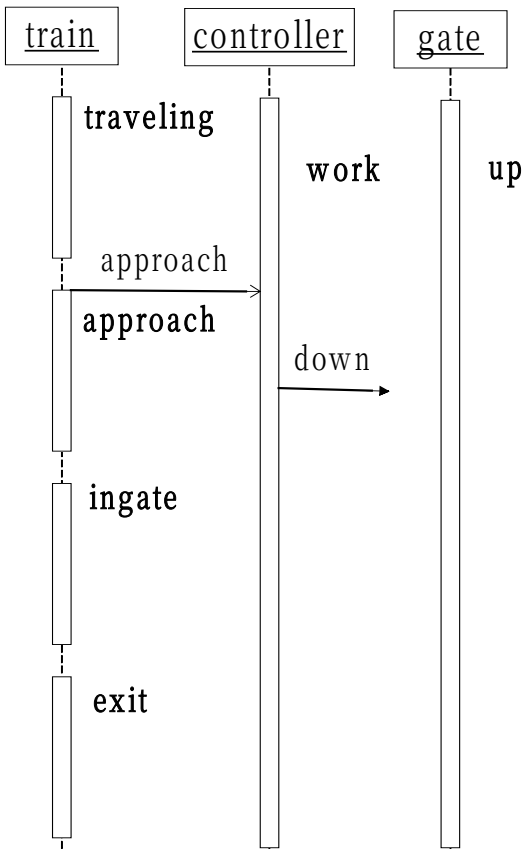

Fig. 9-4: Message receiving error

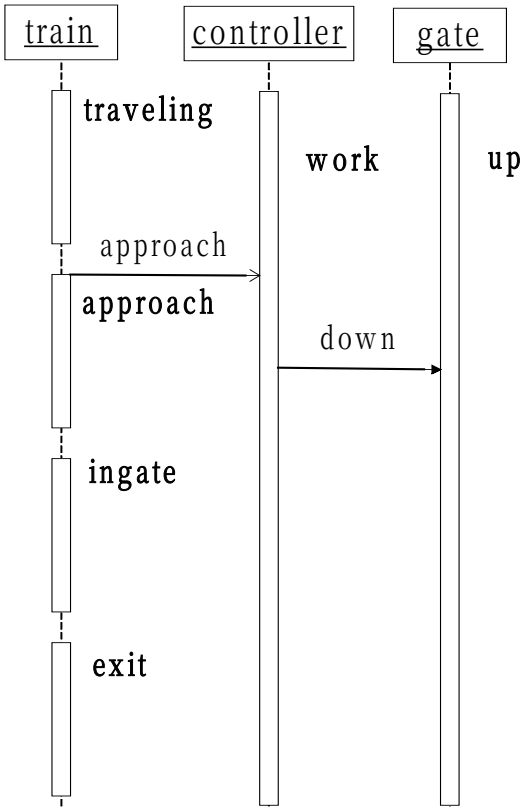

Fig.9-5: Gate hardware errors

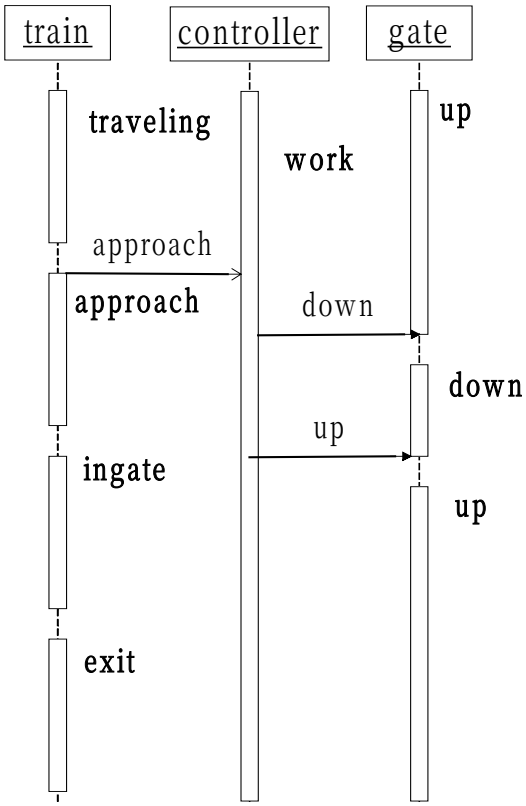

Fig.9-7: Controller software errors

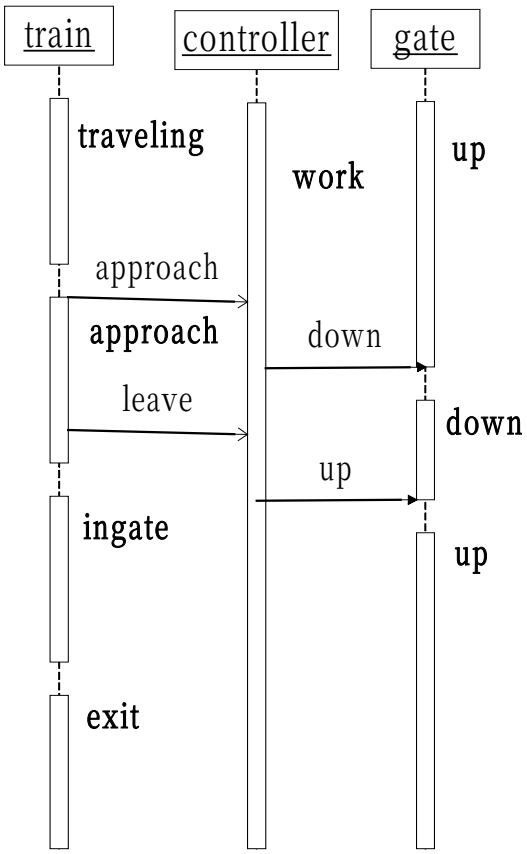

Fig. 9-6: Sensor errors

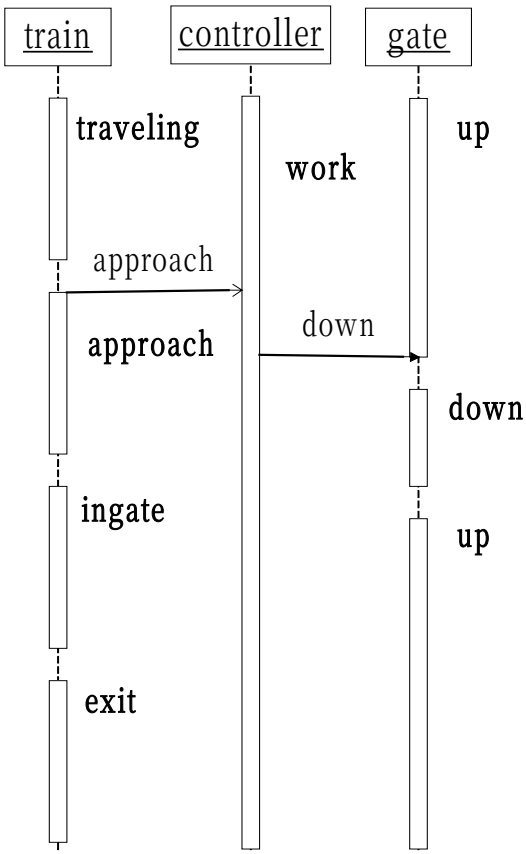

Fig. 9-8 :Gate hardware errors

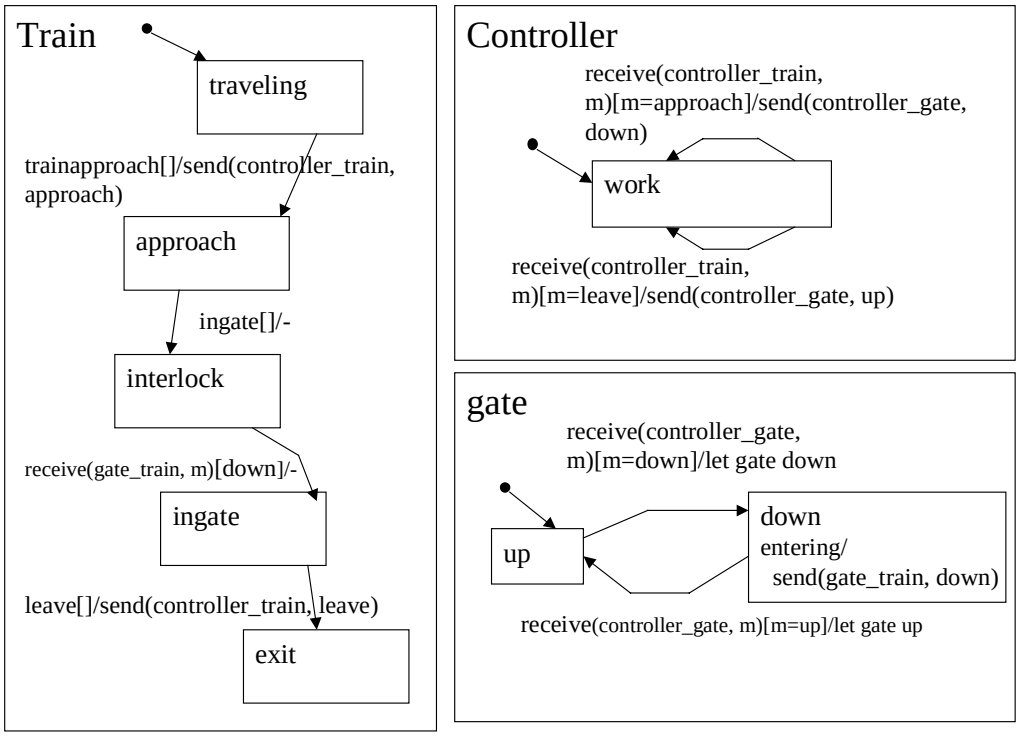

Fig. 10. Modified system design

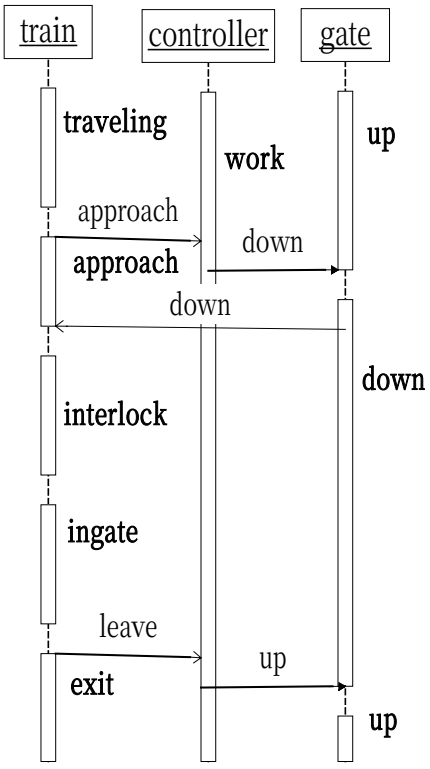

Fig. 11. Modified sequence diagram

Single Train General Rules：

1. [ ](train=traveling || approach || interlock || ingate || exit)
2. [ ](gate=up || down)
3. [ ](controller=work)

Single Train Backward Rules：

1. train=traveling ⇒ (-) (train=traveling)
2. train=approach ⇒ (-) (train=approach || traveling)
3. train=interlock ⇒ (-) (train=interlock || approach)
4. train=ingate ⇒ (-) (train=ingate || interlock)
5. train=exit ⇒ (-) (train=exit || ingate)
6. gate=up ⇒ (-) (gate=up || down)
7. gate=down ⇒ (-) (gate=down || up)
8. controller=work ⇒ (-) (controller=work)

Single Train Forward Rules：

1. train=traveling ⇒ ( )(train=traveling || approach)
2. train=approach ⇒ ( )(train=approach || interlock)
3. train=interlock ⇒ ( )(train=interlock || ingate)
4. train=ingate ⇒ ( )(train=ingate || exit)
5. train=exit ⇒ ( )(train=exit)
6. gate=up ⇒ ( )(gate=up || down)
7. gate=down ⇒ ( )(gate=down || up)
8. controller=work ⇒ ( )(controller=work)

Single Train Transition Rules：

1. (train=traveling) ∧ trainapproach ⇒
   ( ) (train：traveling→approach) ∧ (SEND(controller_train, approach))
2. (train=approach) ∧ interlock ⇒ ( )(train：approach→interlock)
3. (train=interlock) ∧ RECEIVE(gate_train, m) ∧ (m=down) ⇒
   ( ) (train：interlock→ingate)
4. (train=ingate) ∧ leave ⇒ ( )(train：ingate→exit) ∧ (SEND(controller_train, leave))
5. (gate=up) ∧ RECEIVE(controller_gate, m) ∧ (m=down) ⇒
   ( )(gate：up→down) ∧ (let gate down)
6. (gate=down) ∧ RECEIVE(controller_gate, m) ∧ (m=up) ⇒
   ( ) (gate：down→up) ∧ (let gate up)

7. (controller=work) ∧ RECEIVE(controller_train, m) ∧ (m=approach) ⇒ ( )(controller：work→work) ∧ SEND(controller_gate, down)
8. (controller=work) ∧ RECEIVE(controller_train, m) ∧ (m=leave) ⇒ ( )(controller：work→work) ∧ SEND(controller_gate, up)
9. (gate= entering(down)) ⇒ ( )SEND(gate_train, down)

Single Train Constraint Rules：

1. (gate：up→down) ⇒ <->(train：traveling→approach)
1. (gate：down→up) ⇒ <->(train：ingate→exit)
2. (train：interlock→ingate) ⇒ <->(gate：up→down)

Fig. 12. Converted temporal logic rules

Proof:

Prove P= (train=ingate) ∧ (gate=up) is true.

We use ***backward search.***

Considering the previous state may be as follows:

Case I: (train=interlock) ∧ (gate=up)

Case II: (train=ingate) ∧ (gate=down)

Case I ：

(1) (train=interlock) ∧ (gate=up) P & Backward Rule 4
(2) train: interlock → ingate P& (1)
(3) <->(gate：up→down) (2) & Constraint Rule 3
(4) (gate：up→down) ∧ <>(gate：down→up) (3) & P (gate=up)
(5) <->(train：traveling→approach) ∧ <->(train：ingate→exit) (4) & Constraints Rules 1 & 2

Yet, (5) conflicts with P since the single train has not exited yet

Case II:

(1) (train=ingate) ∧ (gate=down) P & Backward rule 6
(2) gate: down→ up previous transition. (1) & P
(3) <->(train：ingate→exit) (2) & Constraint Rule 2

Yet, (3) conflicts with P where (train=ingate)

Thus, both cases are not true. Thus, P cannot be true.

Fig. 13. Correctness proof

References

[1] C, Fan and S. Yih, “Frame-based safety analysis approach for decision-based errors,” *Reliability Eng. & System Safety*, 55, pp. 243-256, 1997.

[2] C. Fan and W. Chen, “Accident sequence analysis of human-computer interface design,” *Reliability Eng. & System Safety*, 67, pp29-40, 2000.

[3] A. Galton (editor), *Temporal Logics and their Applications*, Academic Press, 1987.

[4] D. Harel, “Statecharts: a visual formalization for complex systems,” *Sci. Comput. Program*, Vol. 8, pp. 231-274 , 1987.

[5] N. M. Heimdahl and N. Leveson, “Completeness and Consistency in Hierarchical State-Based Requirements,” *IEEE Transactions on Software Engineering*, Vol. 22, No. 6, pp. 363-377, June 1996.

[6] F. Kroger, *Temporal Logic of Programs*, Springer-Verlag, 1986.

[7] N. Leveson and J. Stolzy, “Safety Analysis Using Petri Nets,” *IEEE Transactions on Software Engineering*, Vol. SE-13, NO. 3, pp. 386-397. 1987.

[8] N. Leveson, et.al, Safety verification of Ada Programs using Software fault trees, *IEEE Software*, 8(7), pp. 48-59, July 1991.

[9] J. Ostroff, *Temporal Logic for Real-Time Systems*, Research Studies Press Ltd., 1989.

[10] A . Pnueli, “The temporal logic of programs,” Proc. of the 18the IEEE Annual Symposium on th Foundations of Computer Science, pp. 46-57, Nov. 1977.

[11] N. Rescher and A. Urquhart, *Temporal Logic*, Springer-Verlag, Library of Exact Philosophy, 1971.

[12] WASH-1400, NUREG 75/014, Reactor safety study, US NRC, Oct. 1974.