Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2016, Article ID 8593612, 11 pages
http://dx.doi.org/10.1155/2016/8593612

Research Article

Failure Propagation Modeling and Analysis via System Interfaces

Lin Zhao,[1] Krishnaiyan Thulasiraman,[2] Xiaocheng Ge,[3] and Ru Niu[1]

[1] *State Key Laboratory of Rail Traffic Control and Safety, Beijing Jiaotong University, Beijing 100044, China*
[2] *School of Computer Science, University of Oklahoma, Norman, OK 73019, USA*
[3] *Institute of Railway Research, University of Huddersfield, Huddersfield HD1 3DH, UK*

Correspondence should be addressed to Lin Zhao; lzhao3@bjtu.edu.cn

Received 12 January 2016; Revised 30 March 2016; Accepted 5 April 2016

Academic Editor: Egidijus R. Vaidogas

Safety-critical systems must be shown to be acceptably safe to deploy and use in their operational environment. One of the key concerns of developing safety-critical systems is to understand how the system behaves in the presence of failures, regardless of whether that failure is triggered by the external environment or caused by internal errors. Safety assessment at the early stages of system development involves analysis of potential failures and their consequences. Increasingly, for complex systems, model-based safety assessment is becoming more widely used. In this paper we propose an approach for safety analysis based on system interface models. By extending interaction models on the system interface level with failure modes as well as relevant portions of the physical system to be controlled, automated support could be provided for much of the failure analysis. We focus on fault modeling and on how to compute minimal cut sets. Particularly, we explore state space reconstruction strategy and bounded searching technique to reduce the number of states that need to be analyzed, which remarkably improves the efficiency of cut sets searching algorithm.

1. Introduction

Our society is relying more and more on the safety of a number of computer-based systems, for example, the control system of managing air traffic or operating a nuclear power plant. These systems are usually called safety-critical systems, which are a class of engineered systems that may pose catastrophic risks to its operators, the public, and the environment. The development of these systems demands a rigorous process of system engineering to ensure that safety risks of the system, even if some of its components fail, are mitigated to an acceptable level. System safety analysis techniques are well established and are used extensively during the design of safety-critical systems.

The size, scale, heterogeneity, and distributed nature of current (and likely future) systems make them difficult to verify and to analyze, particularly for nonfunctional properties including availability, performance, and security, as well as safety. Due to the manual, informal, and error-prone nature of the traditional safety analysis process, the use of models and automatic analysis techniques as an aid to support safety-related activities in the development process has attracted increasing interest. Model-based safety analysis (MBSA), where the analysis is carried out on formal system models that take into account system behaviors in the presence of faults, has been proposed to address some of the issues specific to safety assessment. Recent work in this area has demonstrated some advantages of this methodology over traditional approaches, for example, the capability of automatic generation of safety artifacts, and shown that it is a promising way to reduce costs while further improving efficiency and quality of safety analysis process.

The existing approaches to MBSA, for example, ESACS/ISAAC [1, 2], AltaRica [3–5], Failure Propagation and Transformation Notation (FPTN) [6, 7], Hierarchically Performed Hazard Origin and Propagation Studies (HiP-HOPS) [8], and the AADL with its error annex [9], can be classified into two groups: (a) failure logic based or (b) system states based. Original MBSA techniques, such as FPTN and HiP-HOPS, have sought to unify classical safety analysis methods such as Fault Tree Analysis (FTA) and Failure Modes and Effects Analysis (FMEA) and to provide a formalism for capturing a single authoritative safety model of the system. These approaches emphasized the model of failure propagation logic. The second group of MBSA approaches addresses the analysis of the transition of system states [10–12], in order to identify the routes that a system transits from a safe state to a hazardous state. Since these search-based techniques normally require exhaustive enumeration of all reachable states, they do not fully exploit the advantage of the internal structure of the state space and domain knowledge of safety analysis.

Safety is clearly an emergent property of a system that can only be determined in the context of the whole. As an emergent property, safety arises only when the system components interact with each other in an environment. Such property is controlled or enforced by a set of constraints related to behaviors of system components. Accidents often result from interactions among components that violate these constraints. In general, the term interaction is conceptually simple; it is a kind of action that occurs as two or more objects have an effect upon one another. In practice, interactions among the components dramatically increase the complexity of the overall system. It is intuitively obvious that growing interaction complexity poses a great challenge to engineer safety of the system. In some cases, although hazard identification and safety assessment had been undertaken for system components, the hazards could be missed apparently at least in part because they arise out of the complex and indirect interactions in a complex system, especially when the components of the system are independently developed or operated. The new challenge to MBSA due to the complexity of a system is that it is very hard to analyze all possible dysfunctional interactions in the system so that its hazardous states which reflect the effects of dysfunctional interactions and inadequate enforcement of safety constraints can be identified.

Using interface models to capture these interactions would offer twofold benefits. Interface information could be abstracted from the existing system design models conveniently. This is helpful to the tight integration of the systems and safety engineering processes. Furthermore, interface models are often more abstract and contain much less corresponding implementation details, which help to combat the state space explosion problem in the following automatic analysis.

In this paper, we propose an approach of model-based safety analysis which utilizes extended interface automata [13] to model the nominal behaviors as well as fault behaviors of the system. To avoid the exploration of the entire reachability set, we present a structural analysis strategy, which takes into account the inner structure of state space. This has made possible development of efficient algorithms for the purpose of safety analysis. By applying state space reduction and heuristic search, a much smaller reachable space needs to be explored and thus the efficiency of proposed minimal cut sets algorithm has been improved.

The rest of the paper is organized as follows. In Section 2, we introduce interface automaton as a formal model for safety analysis. In Section 3, we show how to use domain knowledge for efficient state space reduction and minimal cut sets generation. Section 4 mainly demonstrates our approach on a small yet realistic safety-related example where minimal cut sets are generated and analyzed. Conclusions and outlooks for future work are presented in Section 5.

2. Interfaces and Fault Modeling

2.1. Definitions and Notation. Interface automata give a formal and abstract description of the interactions between components and the environment. This formalism captures the temporal aspects of component interfaces, including input assumptions and output guarantees, in terms of I/O actions and the order in which they occur in automata. Input assumptions describe the possible behaviors of the component's external environment, while output guarantees describe the possible behaviors of the component itself.

Definition 1 (interface automata). An interface automaton is defined as a tuple $P = \langle \mathscr{V}_P, \mathscr{V}_P^{\text{Init}}, \mathscr{I}_P, \mathscr{O}_P, \mathscr{H}_P, \mathscr{T}_P \rangle$, where

(i) $\mathscr{V}_P$ is a finite set of states,

(ii) $\mathscr{V}_P^{\text{Init}} \subseteq \mathscr{V}_P$ is a set of initial states,

(iii) $\mathscr{I}_P$, $\mathscr{O}_P$, and $\mathscr{H}_P$ are mutually disjoint sets of input, output, and internal actions; one denotes by $\mathscr{A}_P = \mathscr{I}_P \cup \mathscr{O}_P \cup \mathscr{H}_P$ the set of all actions,

(iv) $\mathscr{T}_P \subseteq \mathscr{V}_P \times \mathscr{A}_P \times \mathscr{V}_P$ is a transition relation.

A trace on interface automaton is an alternating sequence consisting of states and actions, such as $p_0, a_0, p_1, a_1, \ldots, a_{k-1}, p_k$, where $p_i \in \mathscr{V}_P$ and $a_j \in \mathscr{A}_P$ ($i \in \{0, \ldots, k\}$ and $j \in \{0, \ldots, k-1\}$). If an action $a \in \mathscr{I}_P$ (resp., $a \in \mathscr{O}_P$, $a \in \mathscr{H}_P$), then $(v, a, v') \in \mathscr{T}_P$ is called an input (resp., output, internal) transition. We denote by $\mathscr{T}_P^{\mathscr{I}}$ (resp., $\mathscr{T}_P^{\mathscr{O}}$, $\mathscr{T}_P^{\mathscr{H}}$) the set of input (resp., output, internal) transitions. An action $a \in \mathscr{A}_P$ is enabled at a state $v \in \mathscr{V}_P$ if there is a transition $(v, a, v') \in \mathscr{T}_P$ for some $v' \in \mathscr{V}_P$. We denote by $\mathscr{I}_P(v)$, $\mathscr{O}_P(v)$, and $\mathscr{H}_P(v)$ the subsets of input, output, and internal actions that are enabled at the state v.

We illustrate the basic features of interface automata by applying them to the modeling of a railroad crossing control system. Figure 1 depicts the interfaces of three components modeling the train, controller, and gate, respectively. Two sensors are used to detect the approach and exit of the train. The state changes of the controller stand for handshaking with the train (via the actions *Approach* and *Exit*) and the gate (via the actions *Lower* and *Raise* by which the controller commands the gate to close or to open). When everything is ready, a signal *Enter* is sent to authorize the entrance of the train.

In the graphic representation, each automaton is enclosed in a box, whose ports correspond to the input and output actions. The symbols ? and ! are appended to the name of the action to denote that the action is an input and output action, respectively. An arrow without source denotes the initial state of the automaton.

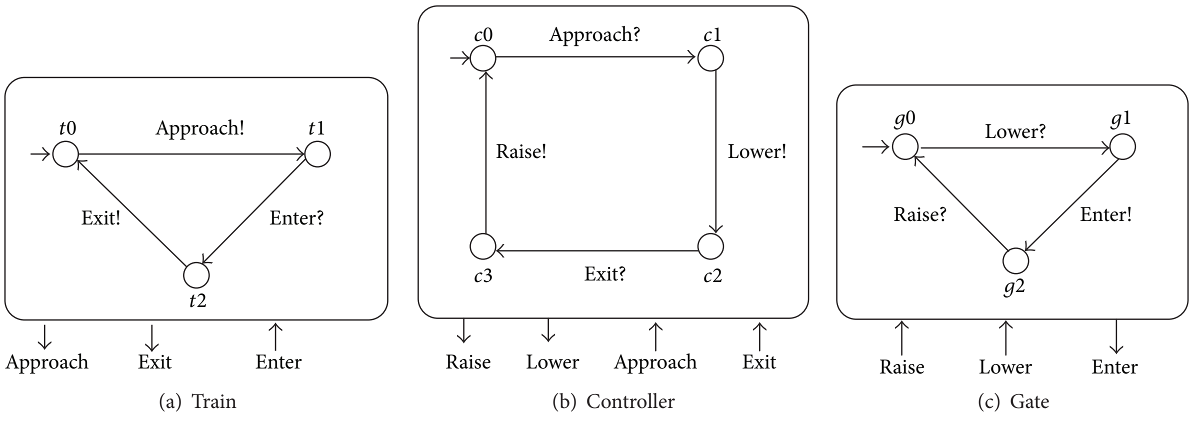

FIGURE 1: The interface models of railroad crossing control system.

The parallel composition of interface automata shows how they all relate and work together. In Alfaro and Henzinger's original paper of interface automata [13], providing a particular form of parallel composition mainly aimed to analyze the compatibility of components. In this paper, the compatibility is not our concern. Therefore, we abandon this kind of parallel composition, using a more traditional one which is common in automaton theory. Two interface automata P and Q are composable if $\mathcal{I}_P \cap \mathcal{I}_Q = \varnothing = \mathcal{O}_P \cap \mathcal{O}_Q$; that is, they have neither common inputs nor common outputs. We let $\text{shared}(P, Q) = \mathcal{A}_P \cap \mathcal{A}_Q$. In a composition, the two automata will synchronize on all common actions and asynchronously interleave all other actions.

Definition 2 (parallel composition). If P and Q are composable interface automata, the parallel composition $P \times Q$ is the interface automaton defined by

$$\begin{aligned}
&\mathcal{V}_{P\times Q} = \mathcal{V}_P \times \mathcal{V}_Q,\\
&\mathcal{V}^{\text{Init}}_{P\times Q} = \mathcal{V}^{\text{Init}}_P \times \mathcal{V}^{\text{Init}}_Q,\\
&\mathcal{I}_{P\times Q} = \left(\mathcal{I}_P \cup \mathcal{I}_Q\right) \setminus \text{shared}(P, Q),\\
&\mathcal{O}_{P\times Q} = \left(\mathcal{O}_P \cup \mathcal{O}_Q\right) \setminus \text{shared}(P, Q),\\
&\mathcal{H}_{P\times Q} = \left(\mathcal{H}_P \cup \mathcal{H}_Q\right) \setminus \text{shared}(P, Q),\\
&\mathcal{T}_{P\times Q} = \left\{\left((v,u), a, \left(v', u\right)\right) \mid \left(v, a, v'\right) \in \mathcal{T}_P \wedge a\right.\\
&\quad \left.\notin \text{shared}(P, Q) \wedge u \in \mathcal{V}_Q\right\}\\
&\quad \cup \left\{\left((v,u), a, \left(v, u'\right)\right) \mid \left(u, a, u'\right) \in \mathcal{T}_Q \wedge a\right.\\
&\quad \left.\notin \text{shared}(P, Q) \wedge v \in \mathcal{V}_P\right\}\\
&\quad \cup \left\{\left((v,u), a, \left(v', u'\right)\right) \mid \left(v, a, v'\right) \in \mathcal{T}_P \wedge \left(u, a, u'\right)\right.\\
&\quad \left.\in \mathcal{T}_Q \wedge a \in \text{shared}(P, Q)\right\}.
\end{aligned} \tag{1}$$

The parallel composition of *train* and *controller* is shown in Figure 2(a). Here, we have only depicted the reachable states of the composition. The automaton *train* × *controller* × *gate* in Figure 2(b), where all the actions have been hidden as internal ones after synchronization, describes the system function in an orderly and concise manner.

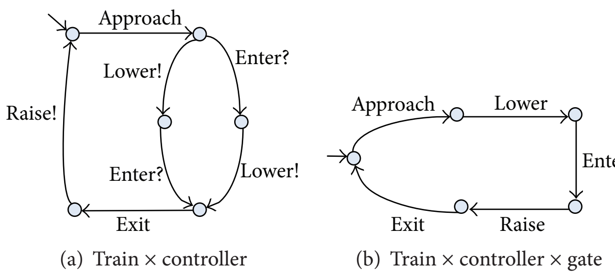

FIGURE 2: The parallel composition of interface models.

2.2. Fault Propagation Modeling on Interface Automata. For model-based safety analysis, failure modes must be explicitly modeled. Our approach to modeling fault behaviors is to specify them using the interface automata notation itself. The incorporation of the fault behaviors directly on the system interface models will promote ease of specification of complex fault behaviors for both the system design and safety engineers, allowing them to create simple but realistic models for precise safety analysis.

Fault modeling is aiming to specify the direct effects of failure modes. In our approach, this is done via importing new actions, states, and transitions to the existing models. There are two types of faults in interface automata: basic faults and propagating faults. Basic faults differ from propagating faults in their activation condition. Basic faults are intrinsic to a component and originate within the component boundary. Their activation occurs independently of other component failures and can be modeled using an independent input action.

The faults that get activated by interaction or interference due to error propagation are considered as propagating faults. In interface automata, propagating faults will be synchronized during the composition of two components and then

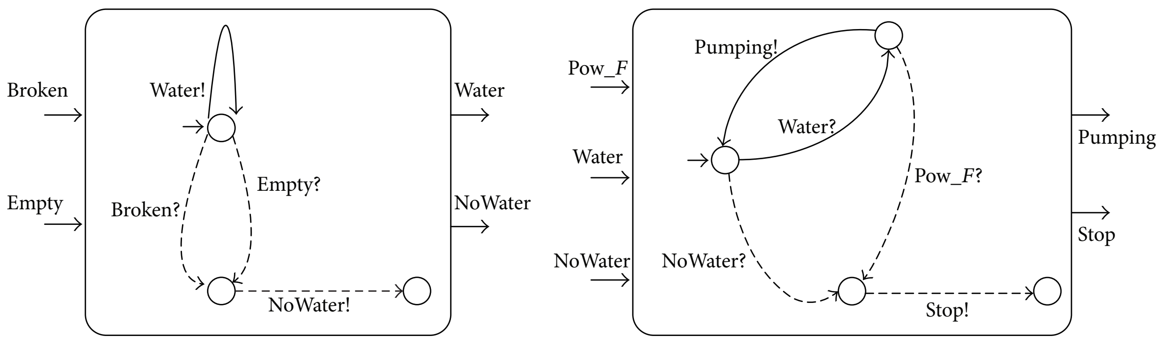

Figure 3: Basic faults and propagating faults in interface automata.

hidden as internal actions. We denote by $\mathscr{E}^{\mathrm{bf}}$ and $\mathscr{E}^{\mathrm{pf}}$, respectively, the mutually disjoint sets of basic faults and propagating faults.

Consider the cooling water supply system in Figure 3. This system consists of an electric pump and a water tank. Two components synchronize on action *Water*, which means there is water in the tank and the pump will start working (action *Pumping*). However, the tank may be broken or empty, denoted by input actions *Broken* and *Empty*. Here, *Broken* and *Empty* are basic faults since they originate within the tank component. *NoWater* is defined as a propagating fault to model the failure propagation from water tank to the pump. Also, there are other propagating faults, like power failure (action *Pow_F*) and the stop of pump (action *Stop*), between pump and other devices not listed in this example.

Our extension towards interface automata lies on two aspects. Firstly, as shown in Figure 3, the extended definition of interface automata could be regarded as a 7-tuple $P = \langle \mathscr{V}_P, \mathscr{V}_P^{\mathrm{Init}}, \mathscr{I}_P, \mathscr{O}_P, \mathscr{H}_P, \mathscr{T}_P, \mathscr{E}^{\mathrm{bf}}, \mathscr{E}^{\mathrm{pf}} \rangle$. Since $\mathscr{E}^{\mathrm{bf}}$ and $\mathscr{E}^{\mathrm{pf}}$ also have input or output attributes, they do not need special treatment during the composition. Besides, solid lines in the figure depict the nominal system interfaces, while the dash lines show the fault behaviors of each of the components. Based on the real system interfaces, this kind of extension is easy to perform and easy to understand and provides useful system insights and shared formal models between the design and safety analysis stages.

3. Algorithms Assist in Failure Analysis

Minimal cut set is the combination of basic faults which can guarantee occurrence of a top-level event (TLE), that is, a set of undesired states, but only has the minimum number of these faults. The key problem investigated in this paper is how to efficiently produce minimal cut sets through exhaustive state space exploration. We first give the following definitions of (minimal) cut sets.

Definition 3 (cut set). Let $P = \langle \mathscr{V}_P, \mathscr{V}_P^{\mathrm{Init}}, \mathscr{I}_P, \mathscr{O}_P, \mathscr{H}_P, \mathscr{T}_P \rangle$ be an interface automaton, $\mathscr{E}^{\mathrm{bf}}$ the set of basic faults, and top-level event TLE $\subseteq \mathscr{V}_P$. cs $\subseteq \mathscr{E}^{\mathrm{bf}}$ is a cut set of TLE if there exists a trace $t = p_0, a_0, p_1, a_1, \ldots, a_{k-1}, p_k$ on P satisfying the following:

(1) $p_0 \in \mathscr{V}_P^{\mathrm{Init}}$, $p_i \notin \mathscr{V}_P^{\mathrm{Init}}$ $(i \in \{1, \ldots, k\})$, and $p_k \in$ TLE;

(2) $\forall i \in \{0, \ldots, k-1\}$ $(a_i \in \mathscr{E}^{\mathrm{bf}} \rightarrow a_i \in \mathrm{cs})$;

(3) $\forall a \in \mathrm{cs} \rightarrow a \in t$.

Intuitively, a cut set is a combination of some basic faults which can lead to the occurrence of the given top-level event, that is, the set of all basic faults contained in a trace from initial states $\mathscr{V}_P^{\mathrm{Init}}$ to top-level event (TLE) is a cut set with respect to TLE. We use $\mathrm{CS}_{\mathrm{TLE}}^{P}$ to represent all cut sets on automaton P with respect to TLE. Minimal cut sets are formally defined as follows.

Definition 4 (minimal cut sets). Let $\mathrm{CS}_{\mathrm{TLE}}^{P}$ be the set of all cut sets on automaton P with respect to TLE. One has the set of all minimal cut sets of TLE on automaton P as follows:

$$\mathrm{MCS}_{\mathrm{TLE}}^{P} = \left\{ \mathrm{cs} \in \mathrm{CS}_{\mathrm{TLE}}^{P} \mid \forall \mathrm{cs}' \in \mathrm{CS}_{\mathrm{TLE}}^{P} \left(\mathrm{cs}' \subseteq \mathrm{cs} \longrightarrow \mathrm{cs}' = \mathrm{cs} \right) \right\}. \quad (2)$$

Based on the previous definitions, the computation of minimal cut sets is to find out all traces leading to the TLE, that is, all cut sets $\mathrm{CS}_{\mathrm{TLE}}^{P}$, and then minimize these sets.

3.1. State Space Reconstruction. Several automatic analysis techniques for minimal cut sets generation have been developed on a variety of models, for example, Petri net, finite state machine, NuSMV model, and AltaRica model. The main difficulty in this kind of search-based minimal cut sets generation is state space reduction, because in general the complexities of searching algorithms depend on the size of the state space.

We observed that, for safety analysis on interface models, only those actions that contribute to the occurrence of the predefined TLE need to be analyzed. During the state exploration, noncontributing actions could be ruled out as far as possible. This means that a majority of transitions relevant to internal and output actions could be peripheral to our core searching algorithm. Based on this observation, we develop a procedure of state space reduction.

To reconstruct the state space of the given interface models, our approach is to cluster states that are noncontributing to the occurrence of TLE into equivalent classes and eliminate relevant transitions. The numbers of states and transitions are reduced using a restricted forward and backward reachability analysis from initial states and TLE, respectively. The result is a representation of the state space that is compact and minimal in some sense and keeps all necessary information about faults propagation.

Definition 5 (state space partition). Let P be an interface automaton, $\mathscr{E}^{\mathrm{bf}}$ the set of basic faults, and top-level event TLE $\subseteq \mathscr{V}_P$. The set of states $\mathscr{V}_P$ consists of three disjoint parts, denoted by SafetyArea^P, TriggeringArea^P, and HazardCore^P, where

(i) SafetyArea^P is a forward closure $U \subseteq \mathscr{V}_P$ such that (1) $\mathscr{V}_P^{\mathrm{Init}} \subseteq U$ and (2) if $u \in U$ and $(u, a, u') \in (\mathscr{T}_P^{\mathscr{O}} \cup \mathscr{T}_P^{\mathscr{H}})$, then $u' \in U$;

(ii) HazardCore^P is a backward closure $U \subseteq \mathscr{V}_P$ such that (1) TLE $\subseteq U$ and (2) if $u \in U$ and $(u', a, u) \in (\mathscr{T}_P^{\mathscr{O}} \cup \mathscr{T}_P^{\mathscr{H}})$, then $u' \in U$;

(iii) TriggeringArea^P $= \mathscr{V}_P \setminus$ (SafetyArea^P $\cup$ Hazard Core^P).

Definition 5 divides the state space of an interface automaton into three separate areas based on top-level event (TLE). Intuitively, the set SafetyArea contains the reachable states of an automaton from the initial states by taking only internal or output transitions. In this area, the current running of the system is safe and there is no occurring of any basic faults. HazardCore consists of all states that can reach TLE through a series of continuous internal or output transitions. States within the scope of HazardCore could evolve into TLE without any external stimulation, that is, the occurrence of any basic fault. TriggeringArea is a complement set containing all states of $\mathscr{V}_P$ excepting those in SafetyArea $\cup$ HazardCore. According to this definition, all of the basic faults are contained in the TriggeringArea, which is the focus of our state exploration algorithm for minimal cut sets generation.

We use a directed graph DG_P, consisting of vertices and labeled edges, to denote the underlying transition diagram of an interface automaton P.

Definition 6 (state space reconstruction). Given an interface automaton P, *Reduced*(DG_P) is the reduced form of its original transition diagram by applying the following steps on DG_P:

(1) remove all transitions from TriggeringArea^P to SafetyArea^P;

(2) remove all transitions from HazardCore^P to SafetyArea^P or TriggeringArea^P;

(3) combine all states of SafetyArea^P into a new state Init;

(4) combine all states of HazardCore^P into a new state Top.

Definition 6 and Figure 4 show the process of our state space reduction strategy. All states in the set *SafetyArea* are merged into a new state *Init*. All states in the set *HazardCore* are combined into state *Top*. After removing all reverse transitions on the fault propagation path, we get a new state space for further analysis.

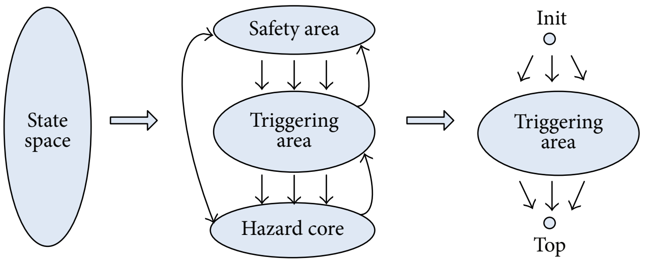

FIGURE 4: The reconstruction of system state space.

Theorem 7. *Let P be an interface automaton. If cs is a cut set with respect to top-level event (TLE), then there exists a trace t in Reduced*(DG_P) *from Init to Top, containing all elements of cs.*

Proof. Since the set of basic faults cs is a cut set of TLE, according to Definition 3, there is a trace $t' = p_0, a_0, p_1, a_1, \ldots, a_{k-1}, p_k$ in DG_P from $\mathscr{V}_P^{\mathrm{Init}}$ to TLE satisfying $\forall a \in \mathrm{cs} \rightarrow a \in t'$. By applying steps (3) and (4) of Definition 6 at both ends of this trace, respectively, we get a new path $t = \mathrm{Init}, a_m, p_{m+1}, \ldots, p_n, a_n, \mathrm{Top}$. Obviously, t is in *Reduced*(DG_P). During this process, only those internal and output transitions in SafetyArea^P and HazardCore^P are removed. Because all basic faults are defined as input actions, hence no basic fault is eliminated from t'; that is, trace t still contains all elements of cs. □

The essence of the search-based minimal cut sets generation is to find out all combinations of basic faults that contributed to the top-level event in DG_P. Theorem 7 shows that DG_P and *Reduced*(DG_P) are equivalent for this purpose, whereas the latter contains far fewer states and transitions. We use an example to illustrate the effectiveness of this approach. Reconsider the previous railroad crossing control system in Figure 1, which is in an ideal world where no errors occur. The next step is to extend these models such that failure modes are also correctly described. The following three failure modes are taken into account in this example:

(i) Failure of the sensors (actions *S1_F* and *S2_F*) which will prevent sending signals (actions *Approach* and *Exit*) when the train is approaching or exiting.

(ii) Failure of the brake (action Bra_*F*) which will lead to nonauthorized entering of the cross, that is, bypassing action *Enter*.

(iii) Failure of the barrier (action *Stuck*) which results in the barrier being stuck at any location; a new state g_3 is added to represent the stuck state of the barrier.

These failure modes are integrated into the formal interface models, as shown in Figure 5. This model extension provides us a failure propagation map on nominal system model,

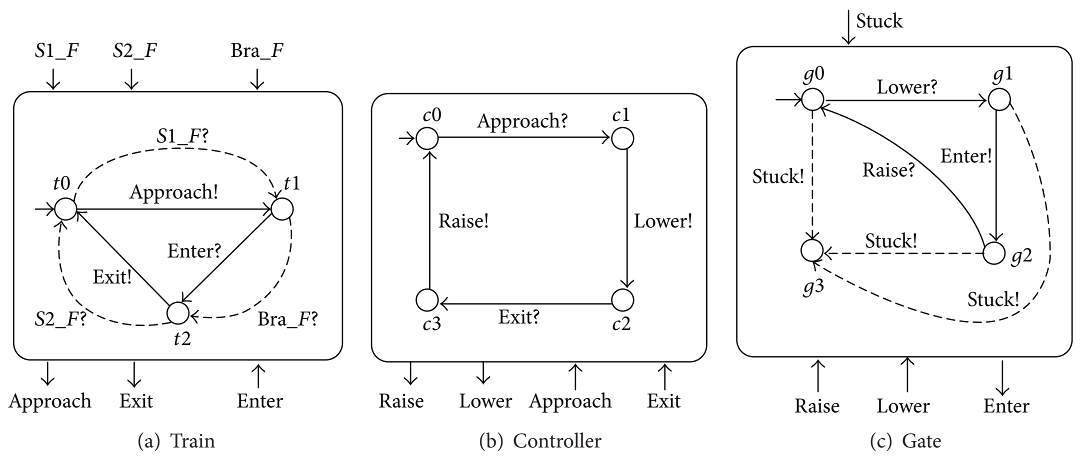

Figure 5: The extended interface models of railroad crossing control.

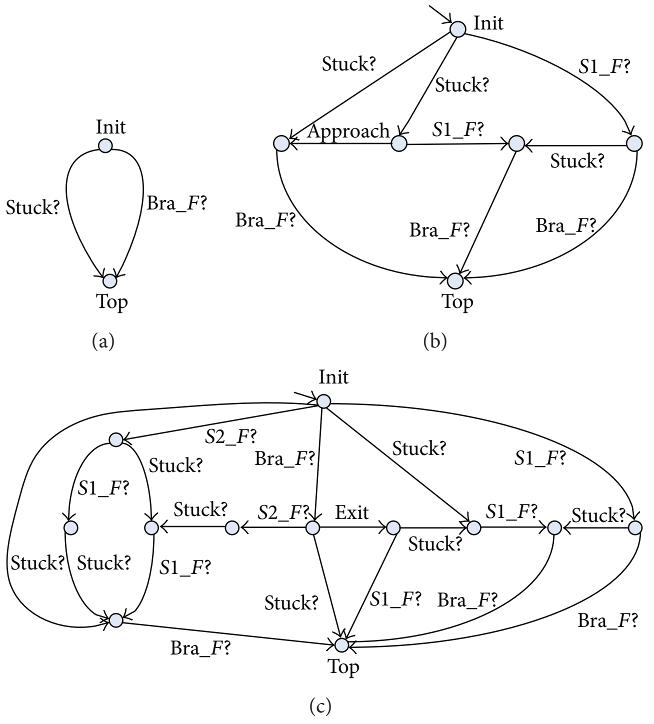

Figure 6: The reshaped state space *Reduced*($\mathrm{DG}_{\mathrm{RC}}$).

reflecting both normal interactions and fault propagation. The safety goal of this system is clear:

SR 1: it must never happen that the train is on the crossing (at state t_2) and the crossing is not secured (at state g_0 or g_3).

RC = $train \times controller \times gate$ is the parallel composition of those extended interface models. There are 30 states and 63 transitions in the state space $\mathrm{DG}_{\mathrm{RC}}$. By using the state space reduction technique in Definition 6, we can obtain a reduced state space *Reduced*($\mathrm{DG}_{\mathrm{RC}}$), as shown in Figure 6, which only contains 17 states and 31 transitions. For brevity, we use 3 separate subgraphs to represent the entire state space, while these subgraphs have the common endpoints *Init* and *Top*.

3.2. Minimal Cut Sets Generation. Here, we discuss the basic searching algorithm for cut sets generation using forward reachability analysis. The first step is to find all possible simple paths (paths without cycles) between two vertices, that is, *Init* and *Top*, in the graph *Reduced*(DG_P). To solve this problem, the traditional depth-first search algorithm could be adjusted in the following manner:

(1) Start at source vertex *Init* and perform a depth-first walk. All nodes on the path are pushed in a stack and set as *visited*.

(2) When the top element of the stack is target node *Top*, a path is successfully found. Record this path, pop out *Top*, and set it as *unvisited*.

(3) For the current top of the stack u, find its successor that is *unvisited* and push this node in the stack. If no such successor exists, pop out u and set u and its successors *unvisited*.

(4) Go back to step (2) until the stack is empty.

To better visualize this process, one can think of a search tree rooted at the vertex *Init*, and all simple paths leading to node *Top* constitute the body of this tree. As an illustration, consider the searching of Figure 6(b). The tree structure in Figure 7(a) depicts all simple paths between *Init* and *Top* generated by the above algorithm. Since a cut set only consists of basic faults, we get the following four cut sets from this tree by removing extra actions and duplicate paths:

$$\begin{gathered}\{\text{Stuck}, \text{Bra\_}F\}, \\ \{\text{Stuck}, S1\_F, \text{Bra\_}F\}, \\ \{S1\_F, \text{Bra\_}F\}, \\ \{S1\_F, \text{Stuck}, \text{Bra\_}F\}.\end{gathered} \tag{3}$$

After finding all possible cut sets in *Reduced*(DG_P), it is easy to identify those minimal ones. According to

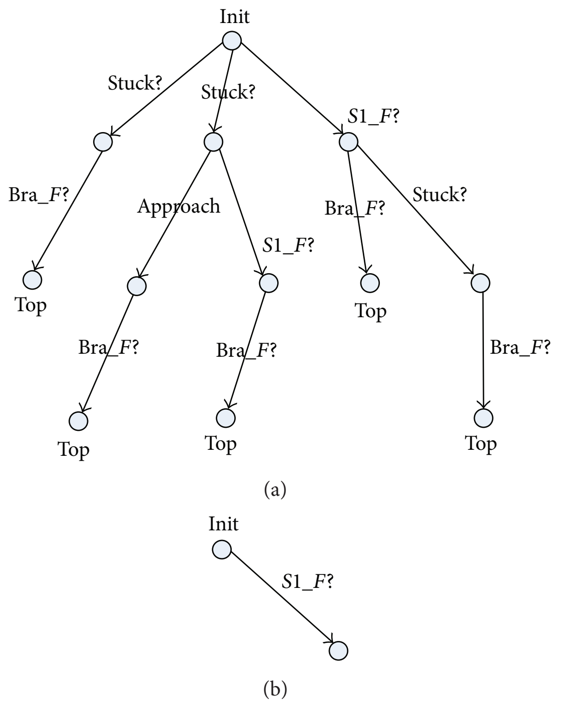

Figure 7: The comparison of different search strategies.

Definition 4, given two cut sets cs_1 and cs_2, if $\text{cs}_1 \subset \text{cs}_2$, cs_2 must not be a minimal cut set. This fact could be used to design a simple filter through pairwise comparison of these cut sets. It is worth noting that sorting this set of cut sets by size in advance will make the comparison more efficient. So far, we have presented a simple search-based algorithm for minimal cut sets generation.

Unfortunately, this naive algorithm has a major drawback: it needs to traverse all possible simple paths between *Init* and *Top* during the searching. However, from a practical perspective, some branches of the search tree could be pruned. Since the ultimate goal is to get minimal cut sets, according to Definition 4, no further extension of the current path is necessary if it contains a cut set which has been found in the previous exploration. Using this observation, we present a heuristic searching strategy based on a bounded breadth-first search to improve the performance of state space exploration.

The basic idea is to search for all simple paths between *Init* and *Top* whose lengths are bounded by some integer k. This problem can be efficiently solved in $\textit{Reduced}(\text{DG}_P)$ via a breadth-first search with bound k. The result is a set of cut sets whose lengths are no more than k, denoted by Ksets, and can therefore be used for guiding the branch pruning during the rest searching. Figure 7 shows a very distinct optimization effect on the graph $\textit{Reduced}(\text{DG}_{\text{RC}})$. Firstly, we perform a bounded breadth-first search (let $k = 1$) on $\textit{Reduced}(\text{DG}_P)$. Via a breadth-first search with depth 1, the searching process will find two traces from *Init* to *Top*, which are exactly shown as Figure 6(a). Thus, we get

$$K\text{sets} = \{\{\text{Stuck}\}, \{\text{Bra\_}F\}\}. \qquad (4)$$

We then use part of the state space (i.e., Figure 6(b)) to demonstrate the branches pruning effect of Ksets. Figure 7(a) shows the paths generated by the naive algorithm, while Figure 7(b) represents the results of the optimized algorithm which uses Ksets to prune superfluous paths. The comparison indicates that over half of the total vertices and edges are ruled out of the searching. In order to tail off the search space and boost converging rate of the algorithm, unnecessary branches are trimmed in terms of the results of the k bounded searching.

Algorithm 1 implements the above discussed techniques, which takes as input a directed graph $\textit{Reduced}(\text{DG}_P)$, a set of basic faults *BasicFaults*, and bounded searching result Ksets. The output *MCSList* returns all minimal cut sets as a list, which will be initialized with Ksets. The set cs is used to record all basic faults in the current path, and it will be added to the end of *MCSList* once the node *Top* in *Reduced*(DG) is reached. All nodes on the current searching path are pushed into a stack S. If the top element of the stack is node *Top* or cs $\in$ *MCSList*, the algorithm will backtrack to continue the search for a new path by popping out the old stack top and trying out the unvisited neighbor of the new stack top. This procedure is repeated over and over until S gets empty. Procedure *Filter*(*MCSList*) carries out the pairwise comparison of all elements in *MCSList* to get those minimal cut sets, as we mentioned before.

For $\textit{Reduced}(\text{DG}_{\text{RC}})$ in Figure 6, computing $K\text{sets} = \{\{\text{Stuck}\}, \{\text{Bra\_}F\}\}$ with $k = 1$ firstly and then using Algorithm 1 to perform a full search in state space, we get all minimal cut sets $\textit{MCSList} = \{\{\text{Stuck}\}, \{\text{Bra\_}F\}\}$. This result shows that the necessary and sufficient conditions for the occurrence of top-level event (i.e., a train is on the crossing, while the crossing is not secured) are as follows: the barrier is stuck or the brake fails, while sensors failure will not consequentially lead to a dangerous situation.

In this approach, the choosing of parameter k in the bounded searching is relatively flexible, depending on the size of *Reduced*(DG). The role of Ksets will gradually change with the increasing of k. Obviously, if the value of k is big enough, all simple paths between *Init* and *Top* will be found in a breadth-first way with low efficiency. Therefore, providing an appropriate value for k is key to the solution. For large models, we recommend a relatively small k for the bounded searching firstly. If Algorithm 1 can not terminate within a reasonable amount of time, gradually increase k until an adequate number of searching branches have been cut down so that the algorithm gets terminated.

4. Fuel Supply System Example

In this section, we exemplify our approach with a fuel supply system for small aircraft. Figure 8 is the schematic diagram of this system. The engine is supplied with fuel pumped at high pressure from a collector tank—a small tank located close to the engine. This demonstration is not concerned with the high-pressure fuel system. The main fuel storage in the aircraft is in the left and right main tanks. Each tank contains a low-pressure pump (P and Q in the diagram) which transfers fuel to the collector tank via valves A and B as required. In

```
Input: Reduced(DG), BasicFaults, Ksets
Output: MCSList is a list of minimal cut sets.
(1) Push Reduced(DG).Init in stack S and set it visited;
(2) MCSList ≔ Ksets;
(3) cs ≔ Null;
(4) while S is not empty do
(5)     v ≔ S.top(); // Get the top element of stack S
(6)     if There exist a vertex u satisfying ((v, b, u) ∈ Reduced(DG) ∧ u is unvisited) then
(7)         if b ∈ BasicFaults then
(8)             Add basic fault b into the set cs;
(9)         if cs ∉ MCSList then
(10)            Push u in stack S;
(11)        else
(12)            Remove a from cs;
(13)        Set u visited;
(14)    else
(15)        Pop v from stack S and set v as well as its successors unvisited;
(16)        Update(cs);
(17)    if the current top element of the stack is Reduced(DG).Top then
(18)        Add a copy of cs to the end of MCSList;
(19)        Pop out Reduced(DG).Top and set it unvisited;
(20)        Update(cs);
(21) Filter(MCSList);
(22) Return(MCSList);
```

Algorithm 1: Generation of minimal cut sets.

flight, valves A and B are normally left open. The aircraft also has a smaller reserve tank, also fitted with its own low-pressure pump R. All pumps are protected by nonreturn valves W, X, Y, and Z. Valves C and D (normally closed and opened when required) allow fuel to be transferred from the reserve to either wing tank as necessary. The pumps have built-in overpressure protection; in the event of attempting to pump into a closed or blocked pipe, the overpressure relief system will simply return fuel to the tank.

We model this system at interface level by three components interacting with each other, as shown in Figure 9. The automaton at the top left, denoted by P_{Left}, describes the interface behavior of the left tank, pump P, and valves W and A. The top right model P_{Right} consists of the right tank, pump Q, and valves X and B. The reserve tank, pump R, and valves C, D, Y, and Z are modeled as P_{Bottom} at the bottom. The solid part of the figure depicts the nominal interactions among these components.

In order to analyze the system behavior in presence of faults, we would like to extend this nominal system model with the given failure modes. In Table 1, we list all faults under consideration, defined as input or output actions, and their intuitive meaning. Those dash lines as well as the new added actions in Figure 9 demonstrate our model extension. The parallel composition FSS = $P_{\text{Left}} \times P_{\text{Bottom}} \times P_{\text{Right}}$ describes the behavior of this fuel supply system in the presence of faults. There are 140 states and 560 transitions in the state space DG_{FSS}.

For this example, assume that the safety requirement is as follows:

SR 2: simultaneous loss of fuel supply from the left and right main tank will not occur.

That is to say, it must never happen that P_{Left} is at the state p_2 and at the same time P_{Right} at q_2. Thus, the top-level event is TLE = $p_2 * q_2$, where $*$ is used as a wildcard to substitute for any state of automaton P_{Bottom}. The set of all basic faults could be obtained from Table 1, that is,

$$\mathscr{E}^{\text{bf}} = \{A\_F, B\_F, C\_F, D\_F, X\_F, Y\_F, Z\_F, W\_F, \\ Empty\_P, Empty\_Q, Empty\_R\}. \quad (5)$$

According to Definition 5, DG_{FSS} could be divided into three parts: $\text{SafetyArea}^{\text{FSS}}$, $\text{TriggeringArea}^{\text{FSS}}$, and $\text{HazardCore}^{\text{FSS}}$, while $\text{TriggeringArea}^{\text{FSS}}$ is the focus of our attention. Using the reconstruction approach given in Definition 6, we get a reduced state space *Reduced*(DG_{FSS}). In contrast, the new state space only contains 88 states and 442 transitions.

Table 1: Parameters of the fuel supply system interface models.

Name	Action	Type	Meaning
A_F, B_F	Input	Basic fault	Unintended shutdown of valve A or B
C_F, D_F	Input	Basic fault	Failure to open valve C or D
W_F, X_F, Y_F, Z_F	Input	Basic fault	Clogging of valve W, X, Y, or Z
P_F, Q_F, R_F	Input	Basic fault	Pump failure in P, Q, or R
OutputP, *OutputQ*	Output	Propagating fault	Fuel supplied successfully by pump P or Q
No_outputP, *No_outputQ*	Output	Propagating fault	No Fuel supplied by pump P or Q
Empty_P, *Empty_Q*, *Empty_R*	Input	Basic fault	Left, right, or reserve tank is empty
InputPR, *InputQR*	Input & output	Propagating fault	Fuel supplied successfully from reserve tank to the left or right one
No_inputPR, *No_inputQR*	Input & output	Propagating fault	No Fuel supplied from reserve tank to the left or right one

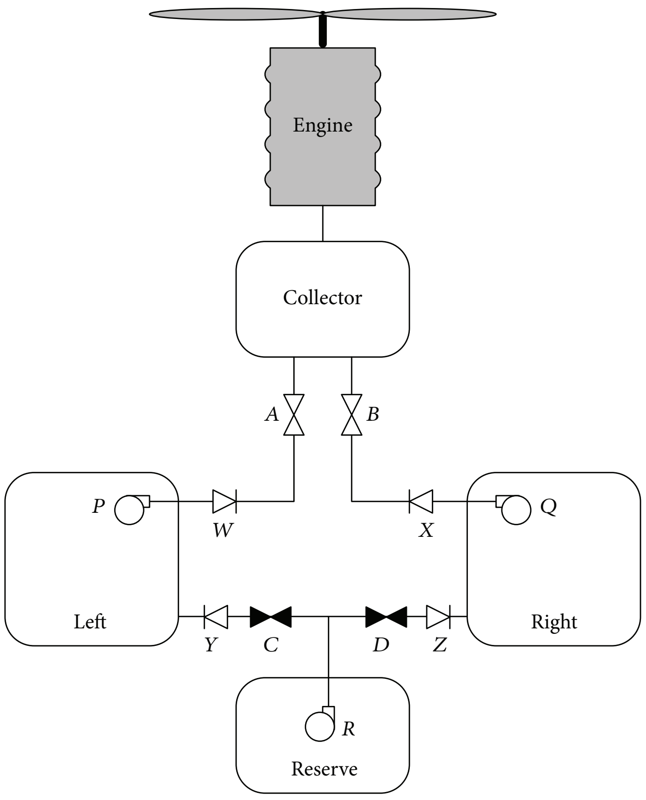

Figure 8: The fuel supply system for a small aircraft.

In our case, we choose $k = 4$ for the bounded breadth-first search which returns a few cut sets *K*sets as a key parameter for the further computation, where

$$\begin{aligned} K\text{sets} = \{&\{P\_F, Q\_F\}, \{P\_F, X\_F\}, \{P\_F, B\_F\}, \\ &\{W\_F, Q\_F\}, \{W\_F, X\_F\}, \{W\_F, B\_F\}, \{A\_F, Q\_F\}, \\ &\{A\_F, X\_F\}, \{A\_F, B\_F\}\}. \end{aligned} \quad (6)$$

Then, Algorithm 1 is performed with *K*sets and its successful termination returns the following *MCSList*, including 39 minimal cut sets:

$\{P\_F, Q\_F\}$ $\{P\_F, X\_F\}$ $\{P\_F, B\_F\}$ $\{W\_F, Q\_F\}$ $\{W\_F, X\_F\}$

$\{W\_F, B\_F\}$ $\{A\_F, Q\_F\}$ $\{A\_F, X\_F\}$ $\{A\_F, B\_F\}$ $\{P\_F, R\_F, EmptyQ\}$

$\{P\_F, EmptyR, EmptyQ\}$ $\{P\_F, D\_F, EmptyQ\}$ $\{P\_F, Z\_F, EmptyQ\}$

$\{W\_F, R\_F, EmptyQ\}$ $\{W\_F, EmptyR, EmptyQ\}$ $\{W\_F, D\_F, EmptyQ\}$

$\{W\_F, Z\_F, EmptyQ\}$ $\{A\_F, R\_F, EmptyQ\}$ $\{A\_F, EmptyR, EmptyQ\}$

$\{A\_F, D\_F, EmptyQ\}$ $\{A\_F, Z\_F, EmptyQ\}$ $\{EmptyP, R\_F, Q\_F\}$

$\{EmptyP, R\_F, X\_F\}$ $\{EmptyP, R\_F, B\_F\}$ $\{EmptyP, R\_F, EmptyQ\}$

$\{EmptyP, EmptyR, Q\_F\}$ $\{EmptyP, EmptyR, X\_F\}$ $\{EmptyP, EmptyR, B\_F\}$

$\{EmptyP, Y\_F, B\_F\}$ $\{EmptyP, C\_F, Q\_F\}$ $\{EmptyP, C\_F, X\_F\}$

$\{EmptyP, C\_F, B\_F\}$ $\{EmptyP, Y\_F, Q\_F\}$ $\{EmptyP, Y\_F, X\_F\}$

$\{EmptyP, EmptyR, EmptyQ\}$ $\{EmptyP, C\_F, D\_F, EmptyQ\}$

$\{EmptyP, C\_F, Z\_F, EmptyQ\}$ $\{EmptyP, D\_F, EmptyQ, Y\_F\}$

$\{EmptyP, Y\_F, Z\_F, EmptyQ\}$

Additionally, this kind of automatic analysis on interface models provides a convenient way to explore the feasibility of different architectures and design choices. For instance, consider the following safety requirement:

SR 3: any loss of fuel supply from the left or right main tank is not allowed.

An interface automaton implicitly represents both assumptions about the environment and guarantees about the specified component. One interesting note about this formalism is that while the environment changed, it would exhibit different system behavior. The environment could also be modeled explicitly by another interface automaton. For safety requirement SR 3, any occurrence of output actions *No_outputP* or *No_outputQ* will lead to danger. The automaton in Figure 10 provides such an environment by accepting these actions as legal inputs. Using the composition

Table 2: The comparison of experimental results.

	RC + SR 1	FSS + SR 2	FSS2 + SR 3
States in DG	30	140	132
States in *Reduced*(DG)	16	88	26
Transitions in DG	63	560	536
Transitions in *Reduced*(DG)	31	442	176
Paths generated by naive searching	19	59721	603
Paths generated by Algorithm 1	8	34875	316
Minimal cut sets	2	39	14

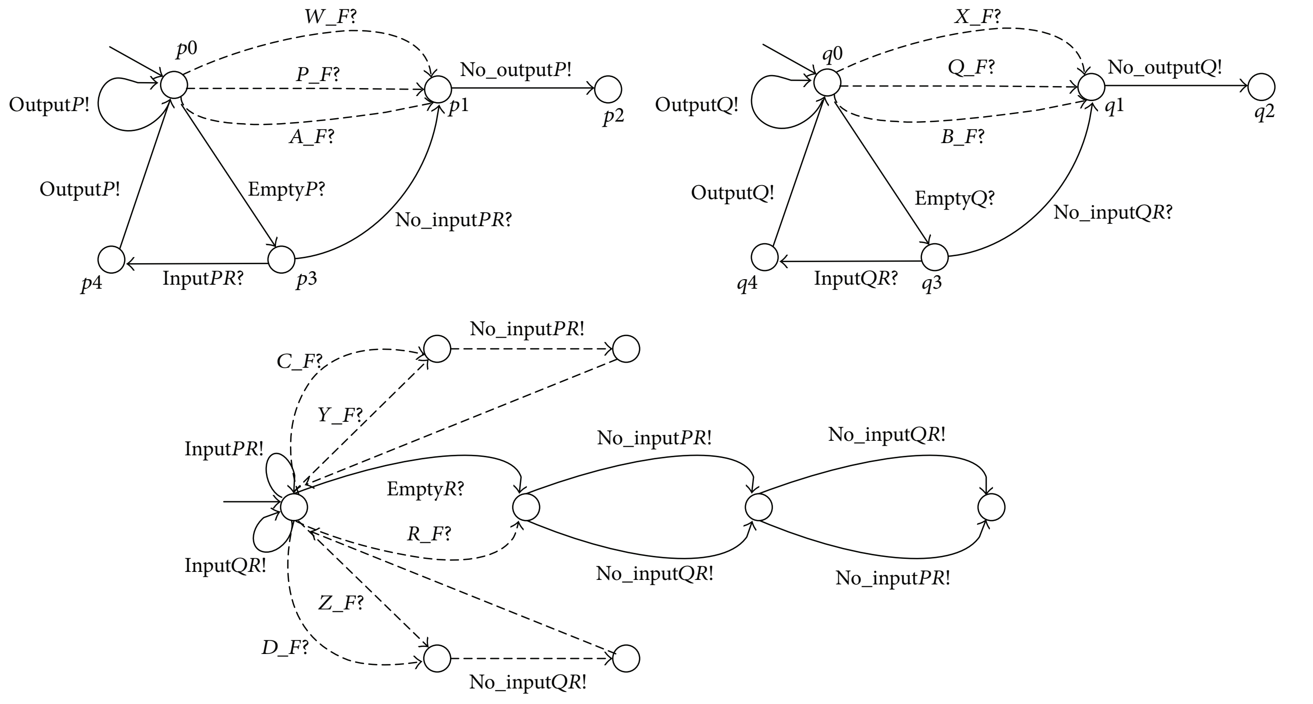

Figure 9: The interface models of fuel supply system.

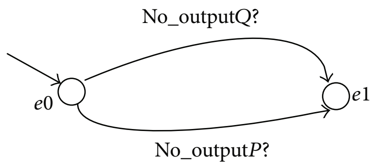

Figure 10: An environment model of fuel supply system.

of FSS and Env, we can quickly locate all dangerous states by TLE $= * * *\ \ e_1$ in the new interface model FSS2 = FSS×Env, which consist of 132 states and 536 transitions. In a similar way, the state space can be reduced to *Reduced*($\mathrm{DG}_{\mathrm{FSS2}}$), containing only 26 states and 176 transitions, and then the corresponding minimal cut sets are generated as follows (with the choice of parameter $k = 2$):

$\{P\_F\}$ $\{W\_F\}$ $\{A\_F\}$ $\{Q\_F\}$ $\{X\_F\}$ $\{B\_F\}$ $\{EmptyP, R\_F\}$

$\{EmptyP, EmptyR\}$ $\{EmptyP, C\_F\}$ $\{EmptyP, Y\_F\}$ $\{R\_F, EmptyQ\}$

$\{EmptyR, EmptyQ\}$ $\{D\_F, EmptyQ\}$ $\{Z\_F, EmptyQ\}$

The above demonstration and experimental results shown in Table 2 indicate that not only is the state space reduction effect satisfactory, but also the improved searching algorithm has more quick convergence speed than naive exhaustive searching. Both of these models can benefit from state space reconstruction, but the impact is more pronounced on FSS2 where nearly eighty percent of the states and seventy percent of the transitions are eliminated. The number of simple paths generated by the searching process is considered as an index of efficiency. The naive searching strategy will deliver all corresponding simple paths in the state space, while Algorithm 1 discards some unnecessary branches for further exploration. Essentially, these two technical measures contribute to the efficiency improvement by narrowing down the search region from different perspectives.

5. Conclusions and Future Directions

Safety analysis is indispensable for ensuring the system safety but is very time-consuming and error-prone. The approach presented in this paper has shown that applying interface automata and restricted reachability analysis techniques to faults modeling and minimal cut sets generation yields a promising means to improve automation and reduce the effort of the analysis. We believe that it is in these areas that formal methods could be most effectively used to aid in safety analysis.

Although several search-based failure analysis approaches have been proposed, this is the first reported application of using the characteristics of faults propagation and cut sets to speed up the search process in a reduced state space (i.e., via our state space reconstruction and guided searching strategy). Besides taking full advantage of domain knowledge, a further very important merit is the flexibility of this approach. With different setting of environment parameters, interface automata could exhibit different behavior which provides an efficient way for engineers to explore different design choices, as we have shown in the case study.

However, we note that directly adding complex fault behaviors to nominal system model tends to severely clutter the model with failure information. This added complexity typically obscures the actual nonfailure system functionality which will make model evolution and maintenance difficult. Without favorable tool supporting, manual incorporation of the fault behaviors may also lead to error-prone extension of the nominal model. Therefore, it is crucial to separately specify fault behavior with a formal notation (e.g., FPTN) and provide a merge mechanism for automatic model extension.

In this paper, we concentrated our attention on fault propagation analysis, but the other important features such as failure ordering analysis and fault tree generation and optimization were not discussed. We will work on those issues and larger case studies aimed at analyzing the scalability of this approach will be the emphasis of our further work. Furthermore, we find some useful characteristics of our state space partition strategy, such as the identification of HazardCore and TriggeringArea, can provide valuable information for runtime monitoring, failure forecast, and fault diagnosis. That might be an interesting attempt to explore the role of model-based safety analysis in system verification and validation.

Competing Interests

The authors declare that they have no competing interests.

Acknowledgments

This work was partially supported by the National Basic Research Program of China (no. 2014CB340703), the Foundation of State Key Laboratory of Rail Traffic Control and Safety (no. RCS2015ZT002), and the Fundamental Research Funds for the Central Universities (no. 2015JBM113).

References

[1] M. Bozzano, A. Villafiorita, O. Åkerlund et al., "ESACS: an integrated methodology for design and safety analysis of complex systems," in *Proceedings of the European Safety and Reliability Conference*, pp. 237–245, 2003.

[2] O. Akerlund, P. Bieber, E. Boede et al., "ISAAC, a framework for integrated safety analysis of functional, geometrical and human aspects," in *Proceedings of 3rd European Congress on Embedded Real Time Systems (ERTS '06)*, pp. 109–120, Toulouse, France, January 2006.

[3] A. Arnold, G. Point, A. Griffault, and A. Rauzy, "The AltaRica formalism for describing concurrent systems," *Fundamenta Informaticae*, vol. 40, no. 2-3, pp. 109–124, 1999.

[4] M. Boiteau, Y. Dutuit, A. Rauzy, and J.-P. Signoret, "The AltaRica data-flow language in use: modeling of production availability of a multi-state system," *Reliability Engineering and System Safety*, vol. 91, no. 7, pp. 747–755, 2006.

[5] P. Bieber, C. Castel, and C. Seguin, "Combination of fault tree analysis and model checking for safety assessment of complex system," in *Proceedings of the European Dependable Computing Conference*, pp. 19–31, Toulouse, France, 2002.

[6] P. Fenelon and J. A. McDermid, "New directions in software safety: causal modelling as an aid to integration," Tech. Rep., High Integrity Systems Engineering Group, Department of Computer Science, University of York, 1992.

[7] P. Fenelon and J. A. McDermid, "An integrated tool set for software safety analysis," *The Journal of Systems and Software*, vol. 21, no. 3, pp. 279–290, 1993.

[8] Y. Papadopoulos and M. Maruhn, "Model-based synthesis of fault trees from Matlab-Simulink models," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN '01)*, pp. 77–82, Goteborg, Sweden, July 2001.

[9] P. Feiler and A. Rugina, "Dependability modeling with the architecture analysis & design language (AADL)," Tech. Rep., Software Engineering Institute, Carnegie Mellon University (SEI/CMU), 2007.

[10] J. Bowen and V. Stavridou, "Safety-critical systems, formal methods and standards," *Software Engineering Journal*, vol. 8, no. 4, p. 189, 1993.

[11] A. Rauzy, "Mode automata and their compilation into fault trees," *Journal of Logic and Algebraic Programming*, vol. 78, no. 1, pp. 1–12, 2002.

[12] M. Bozzano and A. Villafiorita, "Improving system reliability via model checking: the FSAP/NuSMV-SA safety analysis platform," in *Computer Safety, Reliability, and Security*, S. Anderson, M. Felici, and B. Littlewood, Eds., vol. 2788 of *Lecture Notes in Computer Science*, pp. 49–62, 2003.

[13] L. D. Alfaro and T. A. Henzinger, "Interface automata," in *Proceedings of ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 109–120, September 2001.