University of Huddersfield Repository

Zhao, Lin, Thulasiraman, Krishnaiyan, Ru, Niu and Ge, Xiaocheng

Failure Propagation Modeling and Analysis Via System Interfaces

Original Citation

Zhao, Lin, Thulasiraman, Krishnaiyan, Ru, Niu and Ge, Xiaocheng (2016) Failure Propagation Modeling and Analysis Via System Interfaces. Mathematical Problems in Engineering. ISSN 1024-123X

This version is available at http://eprints.hud.ac.uk/28927/

The University Repository is a digital collection of the research output of the University, available on Open Access. Copyright and Moral Rights for the items on this site are retained by the individual author and/or other copyright owners. Users may access full items free of charge; copies of full text items generally can be reproduced, displayed or performed and given to third parties in any format or medium for personal research or study, educational or not-for-profit purposes without prior permission or charge, provided:

- The authors, title and full bibliographic details is credited in any copy;
- A hyperlink and/or URL is included for the original metadata page; and
- The content is not changed in any way.

For more information, including our policy and submission procedure, please contact the Repository Team at: E.mailbox@hud.ac.uk.

http://eprints.hud.ac.uk/
Using System Interfaces for Failure Propagation Modeling and Analysis

Lin Zhao\textsuperscript{a}, Xiaocheng Ge\textsuperscript{b}, Rui Niu\textsuperscript{a}, Haifeng Wang\textsuperscript{c}

\textsuperscript{a}State Key Laboratory of Rail Traffic Control and Safety, Beijing Jiaotong University, Beijing 100044, China
\textsuperscript{b}Computer Science Department, University of York, York YO10 5GH, UK
\textsuperscript{c}School of Electronic and Information Engineering, Beijing Jiaotong University, Beijing 100044, China

Abstract

XXX XXX XX XXX. In this paper we propose a novel approach for safety analysis based on system interface models, which are shared for both system and safety engineering. 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 can be provided for much of the failure analysis. We particularly focus on fault modeling and on how to compute minimal cut sets. In addition, we explore state space reconstruction strategy and bounded searching technique to reduce the number of states that need to be analyzed, which will remarkably improve the efficiency of cut sets searching algorithm.

Keywords: Interface, Safety

1. Introduction

Due to the manual, informal, and error prone nature of the traditional safety analysis process, the use of formal models and analysis techniques as an aid to support safety-related activates in the development process has attracted increasing interest. Model-based safety analysis, where the analysis is carried out on formal system models that take into account system behavior 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 whilst further improving efficiency and quality of safety analysis process.

Email address: lzhao3@bjtu.edu.cn (Lin Zhao)
There are several reasons why choosing interface model as a central model for safety analysis. First, interface information could be abstracted from the existing system design models conveniently. And meanwhile, they are easily comprehensible to the safety engineers. Those are helpful to the tight integration of the systems and safety engineering processes. Second, interfaces are often much simpler than the corresponding implementations, which helps to combat the state space explosion problem in the following automatic analysis. We also propose a state-space reconstruction algorithm that will greatly minimize the scale of state space and thus improve the efficiency of search-based minimal cut sets generation.

The paper is structured as follows. In the next section, we introduce interface automaton as a formal model for safety analysis. In Section 3, we . 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 in the last section.

2. Interfaces and Fault Modeling

2.1. Definitions and Notation

Interface automata give a formal and abstract description 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 they occurring 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 $P = (\mathcal{V}_P, \mathcal{V}_P^{\text{init}}, I_P, O_P, H_P, T_P)$, where

- $\mathcal{V}_P$ is a finite set of states,
- $\mathcal{V}_P^{\text{init}} \subseteq \mathcal{V}_P$ is a set of initial states,
- $I_P, O_P, H_P$ are mutually disjoint sets of input, output and internal actions. We denote by $\mathcal{A}_P = I_P \cup O_P \cup H_P$ the set of all actions,
- $T_P \subseteq \mathcal{V}_P \times \mathcal{A}_P \times \mathcal{V}_P$ is a transition relation.

A trace on interface automaton is an alternating sequence consisting of states and actions. For example, $p_0, a_0, p_1, a_1, ..., a_{k-1}, p_k$ where $p_i \in \mathcal{V}_P$ and $a_j \in \mathcal{A}_P$ ( $i \in \{0, ..., k\}$ and $j \in \{0, ..., k - 1\}$ ). If an action $a \in I_P$ (resp. $a \in O_P$, $a \in H_P$), then $(v, a, v') \in T_P$ is called an input (resp. output, internal) transition. We denote by $T_P^I$ (resp. $T_P^O$, $T_P^H$) the set of input (resp. output, internal) transitions. An action $a \in \mathcal{A}_P$ is enabled at a state $v \in \mathcal{V}_P$ if there is a transition $(v, a, v') \in T_P$ for some $v' \in \mathcal{V}_P$. We denote by $I_P(v)$, $O_P(v)$, $H_P(v)$
the subsets of input, output and internal actions that are enabled at the state $v$.

We illustrate the basic feathers 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 $\text{Approach}$ and $\text{Exit}$) and the gate (via the actions $\text{Lower}$ and $\text{Raise}$ by which the controller commands the gate to close or to open). When everything is ready, a signal $\text{Enter}$ is sent to authorize the entrance of train.

In the graphic representation, each automaton is enclosed in a box, whose ports correspond to the input and output actions. The symbol $?$ 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.

The parallel composition of interface automata shows how they all relate and work together. Two interface automata $P$ and $Q$ are composable if $\mathcal{I}_P \cap \mathcal{I}_Q = \emptyset = \mathcal{O}_P \cap \mathcal{O}_Q$, i.e. 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

![Image of interface models of railroad crossing control system](image-url)
The parallel composition of interface models is shown in Figure 2a. Here, we have only depicted the reachable states of the composition. The automaton $T_{rain} \times \text{Controller} \times \text{Gate}$ in Figure 2b, where all the actions have been hidden as internal ones after synchronization, describes the system function in an orderly and concise manner.

### 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 of 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 independent...
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 hidden as internal actions. We denote by $E_{bf}$ and $E_{pf}$, 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.

Solid lines in the figure depict the nominal system interfaces, while those dash lines show the fault behaviors of each components. Based on the real system interfaces, this kind of extended is easy to perform, 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), i.e., 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 $P = \langle V_P, V_P^{init}, I_P, O_P, H_P, T_P \rangle$ be an interface automaton, $E_{bf}$ be the set of basic faults, and top-level event $TLE \subseteq V_P$. $cs \subseteq E_{bf}$ is a cut set of $TLE$ if there exists a trace $t = p_0, a_0, p_1, a_1, ..., a_{k-1}, p_k$ on $P$ satisfying:

Figure 3: Basic faults and propagating faults in interface automata
Typically, each cut set is associated with a trace witnessing the occurrence of the TLE. We use \( CS_{P,TLE}^P \) 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 \( CS_{P,TLE}^P \) be the set of all cut sets on automaton \( P \) with respect to \( TLE \). We have the set of all minimal cut sets of \( TLE \) on automaton \( P \) as follows:

\[
MCS_{P,TLE}^P = \{ cs \in CS_{P,TLE}^P \mid \forall cs' \in CS_{P,TLE}^P \ (cs' \subseteq cs \rightarrow cs' = cs) \}.
\]

Based on the previous definitions, the computation of minimal cut sets is to find out all traces leading to the TLE, i.e., all cut sets \( CS_{P,TLE}^P \), and then minimize this set.

### 3.1. State Space Reconstruction

Several automatic analysis techniques for minimal cut sets generation have been developed on a variety of models, e.g., 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, non-contributing 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 developed a procedure of state space reduction.

To reconstruct the state space of the given interface models, our approach is to cluster states that are non-contributing 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, minimal in some sense, and keeps all necessary information about faults propagation.

**Definition 5. (State Space Partition)** Let \( P \) be an interface automaton, \( \mathcal{E}^{bf} \) be the set of basic faults, and top-level event \( TLE \subseteq V_P \). The set of states \( V_P \) consists of three disjoint parts, denoted by \( SafetyArea^P \), \( TriggeringArea^P \), and \( HazardCore^P \), where

- \( SafetyArea^P \) is a forward closure \( U \subseteq V_P \) such that: (1) \( V_P^{\text{init}} \subseteq U \) and (2) if \( u \in U \) and \( (u,a,u') \in (T_O^P \cup T_H^P) \), then \( u' \in U \);

- \( HazardCore^P \) is a backward closure \( U \subseteq V_P \) such that: (1) \( TLE \subseteq U \) and (2) if \( u \in U \) and \( (u',a,u) \in (T_O^P \cup T_H^P) \), then \( u' \in U \);
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, i.e., the occurrence of any basic fault.
TriggeringArea is a complement set containing all states of $V_P$ 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 automa-
ton $P$, 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 got a new state space for
further analysis.

**Theorem 1.** 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 $cs$. 

Figure 4: The reconstruction of system state space
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 \( DP \) from \( \forall V^{Init} \) to \( TLE \) satisfying \( \forall a \in cs \rightarrow a \in t' \). By applying step (3) and (4) of Definition 6 at both ends of this trace respectively, we get a new path \( t = Init, a_m, p_{m+1}, \ldots, p_n, a_n, Top \). Obviously, \( t \) is in \( Reduced(DP) \). 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 search-based minimal cut sets generation is to find out all combinations of basic faults that contributing to the top-level event in \( DP \). Theorem 1 shows that \( DP \) and \( Reduced(DP) \) 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:

- Failure of the sensors (actions \( S_1.F \) and \( S_2.F \)) which will prevent to send signals (actions \( Approach \) and \( Exit \)) when the train is approaching or exiting.

- Failure of the brake (action \( Bra.F \)) which will lead to non-authorized entering of the cross, i.e., bypassing action \( Enter \).

- Failure of the barrier (action \( Stuck \)) which result 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, reflecting both normal interactions and fault propagation.

Figure 5: The extended interface models of railroad crossing control
The safety goal of this system is clear: it must never happen that the train is on the crossing (at state $t_2$) and the crossing is not secured (at states $g_0$ or $g_3$), which is a top-level event in fault tree analysis.

$RC = Train \times Controller \times Gate$ is the parallel composition of those extended interface models. There are 30 states and 63 transitions in the state space $DG_{RC}$. By using the state space reduction technique in Definition 6, we can obtain a reduced state space $Reduced(DG_{RC})$, 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, i.e., $Init$ 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 visited.
2. When the top element of the stack is target node $T_{op}$, a path is successfully found. Record this path, pop out $T_{op}$ and set it unvisited.

3. For the current top of the stack $u$, find its successor who is unvisited and push this node in the stack. If no such successor existing, pop out $u$, 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 $T_{op}$ constitute the body of this tree. As an illustration, consider the searching of Figure 6b. The tree structure in Figure 7a depicts all simple paths between $Init$ and $T_{op}$ generated by the above algorithm. Since a cut set only consists of basic faults, we got the following four cut sets from this tree by remove extra actions and duplicate paths:

$$\{\{Stuck, Bra_F\}$$
$$\{\{Stuck, S1_F, Bra_F\}\}$$
$$\{\{S1_F, Bra_F\}\}$$
$$\{\{S1_F, Stuck, Bra_F\}\}$$

After finding all possible cut sets in $Reduced(DG_{P})$, it is easy to identify those minimal ones. According to Definition 4, given two cut sets $cs_1$ and $cs_2$, if $cs_1 \subseteq cs_2$, $cs_2$ 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 $T_{op}$ 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 length are bounded by some integer $k$. This problem can be efficiently solved in $Reduced(\text{DG}_P)$ via a breadth-first search with bound $k$. The result is a set of cut sets whose length are no more than $k$, denoted by $Ksets$, and can therefore be used for guiding the branches pruning during the rest searching. Figure 7 shows a very distinct optimization effect on the graph $Reduced(\text{DG}_RC)$. Firstly, we perform a bounded breadth-first search (let $k = 1$) which will find all cut sets in Figure 6a quickly, that is, $Ksets = \{\{\text{Stuck}\}, \{\text{Bra}_F\}\}$.

Figure 7a shows the paths generated by the naive algorithm, while Figure 7b represents the results of the optimized algorithm which uses $Ksets$ to prune superfluous paths. The comparison indicates that over half of the total vertexes 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 $Reduced(\text{DG}_P)$, a set of basic faults $BasicFaults$, and bounded searching result $Ksets$. The outputs $MCSList$ 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 $Reduced(\text{DG})$ 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 $Reduced(\text{DG}_RC)$ in Figure 6, computing $Ksets = \{\{\text{Stuck}\}, \{\text{Bra}_F\}\}$ with $k = 1$ firstly, then using Algorithm 1 to perform a full search in state space and we get all minimal cut sets $MCSList = \{\{\text{Stuck}\}, \{\text{Bra}_F\}\}$. This result shows that the necessary and sufficient conditions for the occurrence of top-level event (i.e., a train is on the crossing, meanwhile the crossing is not secured) are: the barrier is stuck or the brake fails, while sensors failure will not consequentially lead to dangerous situation.

In this approach, the choosing of parameter $k$ in the bounded searching is relatively flexible, depending on the size of $Reduced(\text{DG})$. 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 a appropriate value for $k$
Algorithm 1: Generation of minimal cut sets

\textbf{Input:} Reduced(DG), BasicFaults, \textit{K}sets

\textbf{Output:} MCSList is a list of minimal cut sets.

1. Push Reduced(DG).\textit{Init} in stack \textit{S} and set it \textit{visited};
2. \textit{MCSList} := \textit{K}sets;
3. \textit{cs} := Null;
4. \textbf{while} \textit{S} is not empty \textbf{do}
5. \hspace{1em} \textit{v} := \textit{S}.\textit{top}(); // Get the top element of stack \textit{S}
6. \hspace{2em} \textbf{if} There exist a vertex \textit{u} satisfying \((\textit{v}, \textit{b}, \textit{u}) \in \text{Reduced}(DG) \land \textit{u} \text{ is unvisited}) \textbf{then}
   7. \hspace{3em} \textbf{if} \textit{b} \in BasicFaults \textbf{then}
      8. \hspace{4em} Add basic fault \textit{b} into the set \textit{cs};
   9. \hspace{3em} \textbf{if} \textit{cs} \notin MCSList \textbf{then}
      10. \hspace{4em} Push \textit{u} in stack \textit{S};
    11. \hspace{3em} \textbf{else}
    12. \hspace{4em} Remove a from \textit{cs};
    13. \hspace{3em} Set \textit{u} \textit{visited};
   14. \hspace{1em} \textbf{else}
    15. \hspace{2em} Pop \textit{v} from stack \textit{S} and set \textit{v} as well as its successors \textit{unvisited};
    16. \hspace{2em} Update(\textit{cs});
   17. \hspace{2em} \textbf{if} the current top element of the stack is Reduced(DG).\textit{Top} \textbf{then}
    18. \hspace{3em} Add a copy of \textit{cs} to the end of MCSList;
    19. \hspace{3em} Pop out Reduced(DG).\textit{Top} and set it \textit{unvisited};
    20. \hspace{3em} Update(\textit{cs});

21. Filter(MCSList);
22. Return(MCSList);

is key to the solution. For large models, we recommend a relatively small \textit{k} for the bounded searching firstly. If Algorithm 1 can not terminate within a reasonable amount of time, gradually increasing \textit{k} until 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 transfer fuel to the collector tank via valves A and B as required. In 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 non-return 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 over pressure protection; in the event of attempting to pump into a closed or blocked pipe, the over pressure 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 $FSS = P_{Left} \times P_{Bottom} \times P_{Right}$ 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:

**SR 1**: *Simultaneous loss of fuel supply from the left and right main tank shall 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 $TLE = p_2 \ast q_2$, where $\ast$ is
uses as a wildcard to substitute for any state of automaton $P_{\text{Bottom}}$. The set of all basic faults could be obtained from Table 1, i.e.,

$$E_{\text{bf}} = \{ A_F, B_F, C_F, D_F, X_F, Y_F, Z_F, W_F, \text{Empty}_P, \text{Empty}_Q, \text{Empty}_R \}.$$

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

In our case, we choose $k = 4$ for the bounded breadth-first search which returns a few cut sets $K_{\text{sets}}$ as a key parameter for the further computation, where $K_{\text{sets}} = \{ \{ P_F, Q_F \}, \{ P_F, X_F \}, \{ P_F, B_F \}, \{ W_F, Q_F \}, \{ W_F, X_F \}, \{ W_F, B_F \}, \{ A_F, Q_F \}, \{ A_F, X_F \}, \{ A_F, B_F \} \}$. Then, Algorithm 1 is performed with this $K_{\text{sets}}$ and its successful termination returns the following
Table 1: Parameters of the fuel supply system interface models

<table>
<thead>
<tr>
<th>Name</th>
<th>Action</th>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_F, B_F$</td>
<td>input</td>
<td>basic fault</td>
<td>Unintended shutdown of valves $A$ or $B$</td>
</tr>
<tr>
<td>$C_F, D_F$</td>
<td>input</td>
<td>basic fault</td>
<td>Fail to open valves $C$ or $D$</td>
</tr>
<tr>
<td>$W_F, X_F, Y_F, Z_F$</td>
<td>input</td>
<td>basic fault</td>
<td>Clogging of valves $W$, $X$, $Y$ or $Z$</td>
</tr>
<tr>
<td>$P_F, Q_F, R_F$</td>
<td>output</td>
<td>propagating fault</td>
<td>Fuel supplied successfully by pump $P$ or $Q$</td>
</tr>
<tr>
<td>No output $P$, No output $Q$</td>
<td>output</td>
<td>propagating fault</td>
<td>No Fuel supplied by pump $P$ or $Q$</td>
</tr>
<tr>
<td>Empty $P$, Empty $Q$, Empty $R$</td>
<td>input</td>
<td>basic fault</td>
<td>Left, right or reserve tank is empty</td>
</tr>
<tr>
<td>input $P R$, input $Q R$</td>
<td>output &amp; output</td>
<td>propagating fault</td>
<td>Fuel supplied successfully from reserve tank to the left or right one</td>
</tr>
<tr>
<td>No input $P R$, No input $Q R$</td>
<td>input &amp; output</td>
<td>propagating fault</td>
<td>No Fuel supplied from reserve tank to the left or right one</td>
</tr>
</tbody>
</table>

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 2**: 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 $SR_2$, 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 inputs. Using the composition of $FSS$ and $Env$, we can quickly locate all dangerous states by $TLE = *** e_1$ in the new interface model $FSS_2 = FSS \times Env$, which consist of 132 states and 536 transitions. In a similar way, the state space can be reduced to $Reduced(DG_{FSS_2})$, 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 indicates that not only the state space reduction effect is satisfactory, but also the improved searching algorithm has more quick constringency speed than naive exhaustive searching. Both of $FSS$ and $FSS_2$ can benefit from state space reconstruction, but the impact is more pronounced on $FSS_2$ where nearly eighty percent of the states and seventy percent of the transition 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 of unnecessary branches for the further exploration. Essentially, these two technical measures contribute to the efficiency improvement by narrowing down the search region from different perspectives.

<table>
<thead>
<tr>
<th></th>
<th>$FSS + SR_1$</th>
<th>$FSS_2 + SR_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>States in $DG$</td>
<td>140</td>
<td>132</td>
</tr>
<tr>
<td>States in $Reduced(DG)$</td>
<td>88</td>
<td>26</td>
</tr>
<tr>
<td>Transitions in $DG$</td>
<td>560</td>
<td>536</td>
</tr>
<tr>
<td>Transitions in $Reduced(DG)$</td>
<td>442</td>
<td>176</td>
</tr>
<tr>
<td>Paths generated by naive searching</td>
<td>59721</td>
<td>603</td>
</tr>
<tr>
<td>Paths generated by Algorithm 1</td>
<td>35875</td>
<td>422</td>
</tr>
<tr>
<td>Minimal cut sets</td>
<td>39</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 2: The comparison of experimental results
5. Conclusions and Future Directions

Acknowledgments

This work was partially supported by the National Basic Research Program of China (No.2014CB340703), and the Foundation of State Key Laboratory of Rail Traffic Control and Safety (No.RCS2012ZT006).