Fail-Safe Test Generation in Safety Critical Systems

Anneliese Andrews  
Department of Computer Science  
University of Denver  
Email: andrews@cs.du.edu

Salwa Elakeili  
Department of Computer Science  
University of Denver  
Email: salwamoh2000@gmail.com

Salah Boukhris  
Department of Computer Science  
University of Denver  
Email: salah_salih@yahoo.com

This paper provides a technique for testing proper failure mitigation in safety-critical systems. Unlike other approaches which integrate behavioral and failure models, and then generate tests from the integrated model, we build failure-mitigation tests from an existing behavioral test suite, using an explicit mitigation model for which we generate mitigation paths which are then woven at selected failure points into the original test suite to create failure-mitigation tests.


I. INTRODUCTION

According to Knight [1] safety-critical systems (SCSs) are those systems whose failure could result in loss of life, damage to the environment or significant property damage. There are many well-known examples in application areas such as medical devices, aircraft flight control, weapons and nuclear systems. Many modern information systems are becoming safety-critical because financial loss and even loss of life can result from their failure. Our aim is to develop a systematic model-based testing (MBT) approach to test fail-safe behavior in SCSs that leverages a behavioral test suite, covers diverse types of failures, improves scalability issues and avoids state space explosion [2] by exploiting behavioral test suites instead of integrating a behavioral model (BM) with a fault model (FM) such as [3],[4],[5] - which have only been used for safety analysis, not for test generation. In order to develop an MBT technique it is also necessary to provide models for mitigation requirements for each failure type, testing criteria for the models, techniques for generating test cases through these mitigation models, and rules for weaving mitigation test paths into behavioral tests at selected failure points. Selecting failure points also needs systematic rules based on testing criteria.

This paper uses Fault Trees (FT) as a Fault Model and Extended Finite State Machines (EFSMs) as a Behavioral Model. Mitigation models are also suggested. Their types are inspired by exception handling patterns developed for process modeling, e. g. [6]. The goal is to provide an MBT technique for testing fail-safe behavior alongside behavioral testing that is flexible, systematic, scalable, and shows potential of being extendable to other types of behavioral models like UML.

Unlike safety analysis, our focus is on testing proper mitigation where it is required. Systematic mitigation testing is essential to reduce risks and possibly severe consequences of mitigation defects in SCSs.

The remainder of this paper is organized as follows. Section II gives some background and related work about model based testing, fault modeling and analysis, integration of safety analysis techniques, behavior models, and mitigation modeling. Section III explains our approach. We illustrate it using the controller of a Railroad Crossing Control system (RCCS) in Section IV. Section V draws conclusions.

II. BACKGROUND AND RELATED WORK

A. Model Based Testing (MBT)

According to Nguyen et al.[7] MBT is an approach to generate test cases using a model of the system under test (SUT). The model provides an abstract view of the SUT by focusing on specific aspects. Utting et al. [8] provide a survey on MBT. They define six dimensions of MBT approaches (a taxonomy): model scope, characteristics, paradigm, test selection criteria, test generation technology and test execution. Dias-Neto et al.[9] characterize 219 MBT techniques, discuss approaches supporting the selection of MBT techniques for software projects, risk factors that may influence the use of these techniques in industry and their mitigation. In [8], the authors classify MBT notations as State_Based, History_Based, Functional, Operational, Stochastic, and Transition_based. Transition_based notations are graphical node-and-arc notations that focus on defining the transitions between states of the system such as variants of finite state machines (FSMs), EFSMs, and communicating extended finite state machines (CEFSMs)). Examples of transition_based notations also include UML behavioral models (like activity diagrams, sequence and interaction diagrams), UML state charts, and Simulink Stateflow charts[8]. Nguyen et al. [7] introduce an approach that integrates model-based and combinatorial testing to generate test cases. Their approach starts with a finite state model and applies MBT to generate test cases. Each path is transformed into classification trees. Executable test cases are generated from these trees using t-way combinatorial criteria. A post optimization algorithm reduces the number of test cases and guarantees the combinatorial criteria of choice on the entire set of test paths are fulfilled.

B. Fault Modeling and Analysis

To make systems low risk and fail-safe, software for safety critical systems (SCSs) must deal with the hazards
identified by safety analysis. There are over 100 different hazard analysis techniques in existence. The most common analysis methods for SCSs are Preliminary Hazard List Analysis (PHL), Preliminary Hazard Analysis (PHA), Subsystem Hazard Analysis (SSHA), System Hazard Analysis (SHA), Fault Tree Analysis (FTA), Event Tree Analysis (ETA), Failure Mode and Effects Analysis (FMEA), Fault Hazard Analysis (FHA), Functional Hazard Analysis (FuHA), Sneak Circuit Analysis (SCA), Petri Net Analysis (PNA), Markov Analysis, Hazard and Operability Analysis (HAZOP), Cause Sequence Analysis, and Common Cause Failure Analysis [10]. These techniques aid in the detection of safety flaws, design errors, and weaknesses of technical systems. FTA is a top-down deductive analysis technique used to detect the specific causes of possible hazards [11][12]. The top event in a fault tree is the system hazard. FTA works downward from the top event to determine potential causes of a hazard. It uses boolean logic to represent these combinations of individual faults that can lead to the top event [12]. FTA is a qualitative model which discloses the possible combinations of identified basic events sufficient to cause the hazard. FMEA is a bottom-up method of analyzing and evaluating safety problems in a system. The FMEA technique consists of identifying and listing all possible failure modes, evaluating effects on the whole system for each failure mode, and identifying all potential causes that may lead to each failure mode [13]. Ramaiah et al.[14] apply FMEA and FTA to the software functions of a prototype SCS- Railroad Crossing Control System (RCCS) to identify possible hazardous software faults. Tribble et al.[11] conduct a comprehensive software safety analysis on a Flight Guidance System (FGS) and Vertical Navigation (VNAV) system using model checking and theorem proving to verify the presence of safety properties in the model.

C. Integration of Safety Analysis Techniques and Behavior Models

Several studies have tried to bridge the gap between fault tree and system modeling. Ariss et al.[3] introduce an approach that integrates fault trees and statecharts via a set of transformation steps that maintain semantics of both models. A set of conversion rules that transform gates of fault trees into statechart notation is presented. The integrated model shows how systems behave when a failure occurs. It aids in the identification of system constraints in order to mitigate failures or correct functional and safety specifications. Thus safety analysis is included into the software design process at an early stage. Kim et al.[4] present rules and algorithms to bridge the gap between hazard analysis and system specification by transforming hazards from FTs to a state machine diagram. The algorithms aid the engineer to develop the primary events of the FT by matching them with elements of the state machine diagram, provide transformation rules, and deal with implicit transitions of the state machine diagram. The integrated model focuses on the causes of the hazard and shows direct paths to the causes which helps to identify test scenarios.

The authors of [3] and [4] both propose an approach to integrate fault trees and statecharts. They differ in how integration is done. In addition, in [3] the authors consider FT notations involving time or counters. In [4] each transformed state machine diagram captures both explicit and implicit causes that trigger a hazard with respect to normal behavior.

Kaiser et al.[5] provide an approach to integrate behavioral states and events into State/Event Fault Trees (SEFTs). Temporal order of events can be expressed by SEFTs, as are gates with memory (e.g. priority AND). The component concept developed for component fault trees(CFTs) has been further developed for SEFTs: each system may be decomposed into subcomponents. Components are transformed into Deterministic and Stochastic Petri Nets (DSPNs) for quantitative probabilistic analysis.

Reza et al. [15] propose the use of FTA and Petri nets (PNs) to improve the safety of a system by performing both forward and backward reachability analyses related to undesired and desired behaviors. They map a fault tree to a PN using a set of mapping rules (FT-PN). Miguel et al. [16] introduce a model driven development approach and the use of a platform-independent language to bridge the gap between safety analyses (FTA and failure mode effects and criticality analysis) and software development languages such as UML.

In [17], the authors employ UML stereotypes in conjunction with component-based software development techniques, integrating safety related elements into UML, to handle safety analysis during architecture design. Lu et al. [17] show that an adoption of HazOp and FTA to UML component models may provide a useful technique for hazard analysis.

Iwu et al. [18] illustrate an approach to developing SCSs by integrating Practical Formal Specifications (PFSs) into the UML with various forms of safety analysis. These techniques show how to combine safety models with functional modeling for safety analysis purposes only, but not for testing. Since some fault models are far less formal than the behavioral models, compatibility issues can arise. They do not consider mitigation modeling in a systematic way.

D. Mitigation Modeling

Safety critical systems (SCSs) have requirements that mandate that safety faults have to be identified, removed and mitigated. Mitigating failures allows a system to continue operations at a reduced level rather than failing completely. Since our objective is MBT for SCS, we need to provide test ready models for mitigation behaviors. Many mitigations follow a common pattern, like a safety-shutdown, trying alternatives, and omitting functionality that has become dangerous, etc. While no work on mitigation models for SCSs exists, exception handling patterns have been defined for process modeling. Lerner et al.[6] identify several, like presenting other alternatives, inserting behavior or skipping some tasks or aborting the current processing. They focus on the composition of the exception handling tasks with normal tasks to identify higher level patterns. Exception handling is a common approach to fault tolerance in software systems. Avizienis et al.[19] illustrate a taxonomy of error handling and fault handling
(fault tolerance) techniques such as rollback, rollforward, and compensation. Ye et al. [20] describe a systematic approach to allow the selection of suitable mitigation strategies (to make the system fail safe) according to a taxonomy of component failure types. With their approach, an untrustworthy component can be used within a critical application with increased confidence. Donmez et al. [21] introduce a taxonomy of mitigation strategies for driver distraction as a framework to address the driver distraction problem systematically. Subramanian et al. [22] present some patterns of safety fault mitigation in medical devices which propose appropriate fault mitigation processes and techniques for diverse safety situations. None of these address testing proper mitigation via MBT.

III. APPROACH

A. Test Generation Process

Our goal is to provide an MBT approach to test proper mitigation of safety failures in SCSs. Because of some scalability and complexity issues in integrating behavioral and fault models, we decided to not integrate behavioral, fault, and mitigation models. This also has the advantages that one can generate a behavioral, functional test suite as usual, using a behavioral model of choice, such as a UML sequence diagram, activity diagram, EFMS, etc. Any of a member of graph-based testing criteria from [23] can be used. We then use a fault model and fault-coverage criteria to determine what fault is to be injected at which point in the behavioral test suite. Mitigation models describe mitigation patterns associated with a fault. Mitigation test criteria describe required coverage. Mitigation test paths are then generated and woven into the behavioral test similar to aspect-oriented modeling [24]. Weaving rules describe how a mitigation test path is woven into the original behavioral test. The test generation process is illustrated in Figure 1. The safety critical testing process then has the following phases:

- Construct a behavioral test suite BT from the behavior model BM, using behavior test criteria BC.
- Construct mitigation test suites MT from mitigation models MM, using mitigation criteria MC.
- Select positions of failure (p) in test suite (BT), and type of failure (e) (failure scenarios). Select (p,e) using failure coverage criteria FC.
- Construct a safety mitigation test suite SMT using the behavioral test suite (BT), point of failure (p), type of failure (e) and mitigation test suite (MT) according to weaving rules (WR).

We describe each phase in the following subsections.

B. Phase 1: Behavioral Model BM and Behavioral Test BT

Although a wide range of behavioral models exist, we illustrate our approach using EFMS. EFMSs have been widely used in areas ranging from aircraft, train control, and medical applications [25]. An EFMS is defined as [26]:

\[ E = (S, X, E_v, V) \]

where S is a set of states, X is a set of transitions, Ev is a set of events, and V is a store represented by a set of variables. Transitions have a source state \( source(x) \in S \), a target state \( target(x) \in S \) and a label \( lbl(x) \). Transition labels are of the form \( e_i[g] \) where \( e_i \in Ev \), g is a guard, i.e. a condition that guards the transition from being taken when \( e_i \) is true, and a is a sequence of actions (we assume a standard expression language including assignments). All parts of a label are optional. Test criteria such as edge-coverage, prime-path coverage, etc. [23] can be defined. Using any of a number of test path generation techniques, test paths can then be generated that fulfill these coverage criteria. Let \( BT = \{t_1, \ldots, t_l\} \) be the set of such paths.

C. Phase 2: Determine Points of Failure

Let the set of failures F be defined as \( \{f_1, f_2, f_3, \ldots, f_k\} \). A failure is injected into the system by manipulating parameters that indicate to the software under test (SUT) that a particular failure has occurred (obviously, we do not want a failure event like a gas leak to actually occur). This is modeled by inserting a failure injection action directly at the point of failure in the test suite. A point of failure is a particular state in a test path at which the failure is injected. Let CT be the concatenation of test paths in BT. That is \( CT = t_1 \circ t_2 \circ t_3 \ldots t_l \). Let \( len(t) \) be the number of nodes in \( t \). Then \( I = len(CT) = \sum_{i=1}^{l} len(t_i) \). The position of failure \( p \) is a position in the behavioral test suite where a failure is injected. It indicates a point of failure \( (I \leq p \leq I) \). We are also selecting failure type \( e \) \( (I \leq e \leq |E|) \) to apply at the point of failure \( p \). Hence we are selecting \( (p,e) \) such that node-failure coverage criteria are met. Note that not all combinations \( (p,e) \) are applicable since not all failures are possible or relevant in every node. Therefore, we define a node-failure applicability matrix \( A(i,j) \).

\[ A(i,j) = \begin{cases} 1, & \text{if failure type } j \text{ applies in node } i \text{ in } S \\ 0, & \text{otherwise} \end{cases} \]

Let \( s = node(p) \) that is \( s \) is the behavioral node in the test suite at position \( p \). Obviously there has to be at least one state in the behavioral model where a given fault applies. Hence no row in the failure applicability matrix can have all zeros. Hence for a given failure type \( j \) there must be some node \( s \) such that \( A(s,j) \) is true. Failure coverage criteria FC for selecting \( (p,e) \) need to be defined next. What combinations of test suite positions and failure types \( (p,e) \) (failure scenarios) should one require?

**Criteria 1:** All combinations, i.e. all positions \( p \), all applicable failure types \( e \) (test everything). This is clearly infeasible for
all but the smallest models. It would require $|I| \times |F|$ pairs if $A$ contains all “1”s.

**Criteria 2:** All unique nodes, all applicable failures. This only requires $\sum_{j=1}^{k} \sum_{i=1}^{[S_i]} (A(i,j)=1)$ combinations i.e. the number of one entries in the applicability matrix. When some nodes occur many times in a test suite only one needs to be selected by some scheme. This could lead to not testing failure recovery in all tests. A stronger test criterion is to require covering each test as well.

**Criteria 3:** All tests, all unique nodes, all applicable failures. Here we simply require that when unique nodes need to be covered they are selected from tests that have not been covered.

A weaker criterion is not to require covering all applicable failures for each selected position.

**Criteria 4:** All tests, all unique nodes, some failures (only one failure per position, but covering all failures). Some failure means that collectively all failures must be paired with a position at least once, but not with each selected position as in Criteria 3.

**Example:** This example shows the differences between the four types of coverage criteria for all combinations $(p,e)$. Suppose we have a test suite that has three test cases $T=\{t_1,t_2,t_3\}$ where each test case contains a path.

$t_1 = \{s_1,s_2,s_3,s_4\}, \ t_2 = \{s_1,s_2,s_1,s_3,s_1\}, \ t_3 = \{s_2,s_4,s_3,s_2,s_3,s_4\}$

$CT = t_1 \circ t_2 \ldots t_3 = \{s_1,s_2,s_3,s_4,s_1,s_2,s_1,s_3,s_1,s_4,s_2,s_4,s_3,s_2,s_3,s_4\}$

$I= [1,16]$. Assume we have four applicable failures $F=\{f_1,f_2,f_3,f_4\}$. $|F|=4$ failures and we have four failure types $E=[1,4]$. The applicability matrix is shown in Table I. Tables II-V show $(p,e)$ pairs marked with “1” that, if selected, would collectively meet the associated test criteria.

Table II shows required combinations for criteria 1. This would need $|I| \times |F|$ minus the zeros entries (not applicable) in Table II (16 x 4 - 12 = 64-12=52 pairs). For a tiny model with only 4 nodes and 4 failure types this is clearly too much.

For Criteria 2 consider Table III. The options selected (marked 1) provide the desired coverage, but only test $t_1$ is used to fulfill this coverage. A total of 13 pairs is needed.

According to Table III the following position-failure pairs $(p,e)$ are selected: \{(1,1),(1,2),(1,3),(1,4),(2,1),(2,3),(2,4),(3,1),(3,2),(3,4),(4,1),(4,2),(4,4)\}. A large portion of the test suite is unused. Random selection of nodes in I can improve this somewhat.

Criteria 3 requires using all tests. Table IV shows an example of a set of position-failure pairs $(p,e)$ that fulfills this criterion \{(1,1),(1,2),(1,3),(1,4),(6,1),(6,3),(6,4),(10,1),(10,2),(10,4),(13,1),(13,3),(13,4)\}. As before, 13 pairs are needed, but the selection of unique nodes is spread over all three tests.

Criteria 4 does not require that all failures be applied at every selected position although each failure must be selected at least once. Table V shows an example of selecting position-failure pairs $(p,e)$. This is the weakest criterion, since it only requires selecting each failure at least once and each unique node at least once. The four position-failure pairs in Table V that fulfill this criterion are \{(1,1),(6,3),(10,2),(13,4)\}.

**D. Phase 3: Generate Mitigation Test (MT)**

Safety critical systems (SCSs) require mitigation of failures to prevent adverse effects. This can take a variety of actions. Mitigation patterns have been defined in [19][6] as follows:

1. **Rollback** brings the system back to a previous state before the failure occurred. A mitigation action may occur and the system may stop or proceed to re-execute the remainder of the test.
2. **Rollforward** mitigates the failure, fixes and proceeds.
3. **Try other alternatives** deals with decisions about which of several alternatives to pursue.
4. **Immediate(partial) fixing** when a failure is noted, an action
is taken to deal with the problem that caused this failure prior to continuing with the remainder of the test.

5. **Deferred (partial) fixing** when a failure is noted, an action must be performed to record the situation and deal with the failure either partially or temporarily because handling the failure completely is not possible.

6. **Retry** when a failure is detected immediately after the execution of the activity causing the problem, an action is performed to solve the failure and then the activity that caused the problem is tried again.

7. **Compensate** means the system contains enough redundancy to allow a failure to be masked.

8. **Go to fail-safe state** a system is transferred into a mitigation state to avoid dangerous effects and stops.

These mitigation patterns can be expressed in the form of mitigation models. For example, try other alternatives is shown in Figure 2. Each failure $f_i$ is associated with a corresponding mitigation model $MM_i$ where $i = 1, ... , k$. We assume that the models are of the same type as the behavioral model BM (e.g. an EFSM). Graph-based [23], mitigation criteria $MC_i$ can be used to generate mitigation test paths $MT_i = m_{t_1}, ..., m_{t_{k_i}}$ for failure $f_i$. Figure 2 shows an example of a mitigation model of type "Try alternatives". Assuming MC as "edge coverage", the following three mitigation test paths fulfill MC: $MT = \{m_{t_1}, m_{t_2}, m_{t_3}\}$ where

$$mt_1 = \{n_1, n_2, n_5\}, mt_2 = \{n_1, n_3, n_5\}, mt_3 = \{n_1, n_4, n_5\}$$

Mitigation models can be very small for some failures and the mitigation can be an "empty action". For example, if there is a rollback to state $s_b$ with immediate stop, the mitigation action only consists of adding a transition from $s_b$ to $s_f$, the final state. Hence, $mt = \{s_b, s_f\}$. The weaving rule would specify what node to rollback to, in this case $s_b$. On the other hand, some mitigation models may consists of a full set of alternative behaviors that completely replace the remainder of the original test. We will illustrate this in the next section.

**E. Phase 4: Generate Safety Mitigation Tests using weaving rules**

Assume we have $t ∈ BT$, $p ∈ I$, $e ∈ E$ and $mt ∈ MT_e$. We now build a safety mitigation test $smt ∈ SMT$ using this information and the weaving rules $wr_e ∈ WR$ as follows:

- keep path represented by $t$ until failure position $p$.
- apply failure of type $e (f_e)$ in $p$.
- select appropriate $mt ∈ MT_e$.
- apply weaving rule $wr_e$ to construct $smt$.

We now explain weaving rules more formally for each type of mitigation. Let $t = \{s_1, ..., s_b, ..., node(p), ..., s_f, ..., s_k\}$

1. **Fix**
   - Option 1: Compensate ($\{Partial\ Fix\ and\ proceed\}$) mitigates a failure and continues with the remainder of the behavioral test. So,
     $$smt = s_1, ..., node(p), mt\ node(p), ..., s_k.$$
   - Option 2: Go to fail-safe state ($Fix\ and\ stop$) mitigates a failure and ignores the remainder of $t$: $smt = s_1, ..., node(p), mt$.

2. **Rollforward**
   - Option 1: Rollforward mitigates the failure, and proceeds.
     $$smt = s_1, ..., node(p), mt\ s_f, ..., s_k$$
     where $s_f$ is the node in $t$ to which we rollforward. If only rollforward and no other actions are required $mt$ is empty and $smt = s_1, ..., node(p), s_f, ..., s_k$.
   - Option 2: Deferred fixing. If the failure can only be fixed after reaching the rollforward node $s_f$ then $smt$ becomes:
     $$smt = s_1, ..., node(p), s_f, mt\ s_{f+1}, ..., s_k$$
     Note that further variants of this weaving rule can exist, like a state $s_{df}$ between $s_f$ and $s_k$ at which the failure mitigation $mt$ is inserted, $t = s_1, ..., s_b, ..., node(p), ..., s_f, ..., s_{df}, ..., s_k, mt = s_1, ..., node(p) s_f, ..., s_{df}, mt, s_k$.

3. **Rollbackward**
   - Option 1: Rollbackward. Apply mitigation path $mt$ from point of failure and rollback to node where failure occurred and continue with remainder of behavioral test.
     $$smt = s_1, ..., node(p), mt\ s_b, ..., s_k$$
     where $s_b$ is a node before node $p$.
   - Option 2: Rollbackward and stop.
     $$smt = s_1, ..., node(p), mt\ s_b.$$

4. **Internal compensate (no user action required)**. Test immediate system fix. For example, this can happen if a system switches to backup/redundant sensors. To test this merely requires applying the failure and continuing to execute the original test $t$. In this case, we do not have to modify the original test at all (note that the assumption is that the system deals with the failure internally without any change in black-box behavior).

While weaving rules in this section are representative, they are not meant to be comprehensive. We expect that, over time, we may find some more or find that some are more common than others.

IV. **Example: Railroad Crossing Control System**

A. **Behavioral Model (BM), Test Criteria (BC), Test Suite (BT)**

Figure 3 depicts the behavioral model for the controller of a Railroad Crossing Control System (RCCS) in EFSM format. The model specifies that gates are to be closed and warning lights are to be turned on when a train approaches, and that they are to stay that way until the train is leaving. When the train is leaving, the gates are opened and...
TABLE VI
TEST PATHERS THROUGH EFSM EXAMPLE

<table>
<thead>
<tr>
<th>Test Paths</th>
<th>States in EFSM</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>$s_0, s_1, s_2, s_3, s_4$</td>
</tr>
<tr>
<td>$t_2$</td>
<td>$s_0, s_1, s_2, s_3, s_4$</td>
</tr>
<tr>
<td>$t_3$</td>
<td>$s_0, s_1, s_2, s_3, s_4$</td>
</tr>
<tr>
<td>$t_4$</td>
<td>$s_0, s_1, s_2, s_3, s_4$</td>
</tr>
</tbody>
</table>


TABLE VII
FAILURE TYPES

<table>
<thead>
<tr>
<th>F</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1$</td>
<td>sensor fails (to detect approaching or leaving train)</td>
</tr>
<tr>
<td>$f_2$</td>
<td>warning lights fail</td>
</tr>
<tr>
<td>$f_3$</td>
<td>gate stuck open</td>
</tr>
<tr>
<td>$f_4$</td>
<td>controller fails</td>
</tr>
</tbody>
</table>

the lights switched off. Gates stay open and lights are off while no train is approaching. The model contains 4 states, $S = \{s_0, s_1, s_2, s_3\}$ where the initial state is $s_0$. There are 8 transitions $X = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8\}$. Assuming edge coverage is required, the test paths in Table VI fulfill this requirement.

Controller

Fig. 3. Extended Finite State Machine for RCCS

B. Failures, Applicability Matrix

Table VII shows four possible failures. The applicability matrix is defined as shown in Table VIII indicating that not all failures are applicable in all states.

C. Failure Coverage Criteria (FC)

There are $\sum_{i=1}^{S} \text{len}(t_i)$ positions $p$ to select for failure injection. Concatenating the tests gives

$CT = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}, s_{13}, s_{14}, s_{15}, s_{16}, s_{17}, s_{18}, s_{19}, s_{20}, s_{21}, s_{22}, s_{23}, s_{24}\}$. There are 24 positions. We now apply coverage criteria for positions of failure ($p$) and type of failure ($e$). The resulting required $(p, e)$ pairs are shown in Table IX for each criteria as "1" entries.

D. Mitigation Requirements, Models, Safety Mitigation Tests

The mitigation requirements are summarized in Table X which also specifies the corresponding mitigation models and associated weaving rules. Figures 4-6 show the mitigation models for failures $f_2 - f_4$. Note that $f_1$ does not need a model, since it is an implicit fix that does not use extra test inputs (category 4 under weaving rules). Again, assuming edge coverage, the mitigation tests listed in figures 4-6 fulfill this coverage. Note, that only failure $f_4$ has more than one mitigation test path.

Fig. 4. Fix and Stop: Mitigation Model $MM_2$

Fig. 5. Fix and Proceed: Mitigation Model $MM_3$

Construct Safety Mitigation Tests

Table XI indicates tests for the 4 position-failure pairs that fulfill coverage criteria 4 above. Note that because $f_4$ has two mitigation paths required, there are two test paths for pair (23,4) resulting in 5 test paths in SMT. Similarly, criteria 1 requires $76 + 24 = 100$ tests (instead of 5) which is clearly not desirable. Criteria 2 has $13 + 4 = 17$ tests. Criteria 3 results in $13 + 4 = 17$ test paths.

Even with the small size of this example, it is clear that, depending on the coverage criteria chosen, testing proper
TABLE IX
ALL POSITIONS, ALL APPLICABLE FAILURES

<table>
<thead>
<tr>
<th>F/S</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
</tr>
</thead>
<tbody>
<tr>
<td>j0</td>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
</tr>
<tr>
<td>j1</td>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
</tr>
<tr>
<td>j2</td>
<td>0 1</td>
<td>0 1</td>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>j3</td>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
</tr>
</tbody>
</table>

TABLE X
MITIGATION REQUIREMENT

<table>
<thead>
<tr>
<th>MM</th>
<th>Explanation</th>
<th>Model</th>
<th>WR</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM1</td>
<td>compensate; switch to backup sensor (internal action); send alarm</td>
<td>none(internal compensate);</td>
<td>4</td>
</tr>
<tr>
<td>MM2</td>
<td>fix and stop; close gate; send alarm; stop</td>
<td>see figure 4</td>
<td>1b</td>
</tr>
<tr>
<td>MM3</td>
<td>fix and proceed; turn warning light on; send alarm</td>
<td>see figure 5</td>
<td>1a</td>
</tr>
<tr>
<td>MM4</td>
<td>compensate; switch to back up and close gates or turn warning light on; send maintenance request</td>
<td>see figure 6</td>
<td>1a</td>
</tr>
</tbody>
</table>

Fig. 6. Compensate: Mitigation Model MM4

TABLE XI
SAFETY MITIGATION TESTS

<table>
<thead>
<tr>
<th>SMT</th>
<th>Covers</th>
<th>Explanation</th>
<th>B1 used</th>
</tr>
</thead>
<tbody>
<tr>
<td>smt1</td>
<td>(1,1)</td>
<td>f1</td>
<td>t1</td>
</tr>
<tr>
<td>smt2</td>
<td>(1,2)</td>
<td>so, st, s21, s22</td>
<td>t2</td>
</tr>
<tr>
<td>smt3</td>
<td>(18,3)</td>
<td>so, si, s3, s1, s2</td>
<td>t3</td>
</tr>
<tr>
<td>smt4</td>
<td>(23,3)</td>
<td>so, st, s3, s41, s42</td>
<td>t4</td>
</tr>
<tr>
<td>smt5</td>
<td>(23,4)</td>
<td>so, s1, s3, s41, s43</td>
<td>t4</td>
</tr>
</tbody>
</table>

This paper presented a novel approach to test safety-critical systems with respect to both regular functionality and fail-safe behavior. Its advantages include:
1. It leverages existing behavioral test suites.
2. It systematically constructs safety mitigation models based on commonly occurring mitigation patterns.
3. It defines failure coverage criteria.
4. It provides mechanisms to construct safety mitigation tests.
5. It avoids state-space explosion by keeping failure models qualitative, rather than quantitative, analysis, this information is not included in FT. One would have to switch to a more quantitative failure “model”.

V. CONCLUSION
and behavior models separate.
6. It is flexible with respect to coverage criteria.
We applied the technique to a Railroad Crossing Control System (RCCS). Future work includes more comprehensive case studies showing: (1) Applicability to other safety-critical domains, (2) Generalizability to apply and investigate the possibility of a larger range of mitigation patterns to other behavioral models (e.g. UML), (3) Comparison with existing techniques such as [3] and [4] especially with respect to scalability, efficiency, and effectiveness.

ACKNOWLEDGMENT
This work was supported, in part, by NSF IUCRC grant #0934413 to the University of Denver.

REFERENCES


