AN AUTOMATED APPROACH FOR DISCOVERING RISK-INDUCING FUNCTIONAL FLAWS IN SOFTWARE DESIGNS

A Thesis
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Amro Salem Salem Hassan

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Department:
Computer Science

July 2015

Fargo, North Dakota

North Dakota State University
Graduate School

Title

AN AUTOMATED APPROACH FOR DISCOVERING FUNCTIONAL RISK-INDUCING FLAWS IN SOFTWARE DESIGNS

By

Amro Salem Salem Hassan

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

Hyunsook Do
Chair

Saeed Salem

Sudarshan Srinivasan

Approved:

7/10/2015
Date

Brian M. Slator
Department Chair

ABSTRACT

For safety critical applications, it is necessary to ensure that risk-inducing flaws do not exist in the final product. To date, many risk-based testing techniques were proposed. The majority of these techniques address flaws in the implementation. However, since the overhead of software flaws increases the later they are discovered in the development process, it is important to test for these flaws earlier in the development process. Few approaches have addressed the problem of testing for risk-inducing flaws in the design phase. These approaches are manual approaches, which makes them hard to apply on large complicated software designs. To address this problem, we propose an automated approach for testing designs for risk-inducing flaws. To evaluate our approach, we performed an experiment focusing on specifications of safety critical systems. Our results show that the proposed approach could be effective in discovering functional flaws in behavioral designs that is exposing a risk.

ACKNOWLEDGMENTS

I would like to thank the advisory committee members and professors, Prof. Hyunsook Do, Prof. Saeed Salem and Prof. Sudarshan Srinivasan for their support. Without Prof. Do's technical guidance, technical feedback, and cooperation throughout this study, this research would have not been possible.

DEDICATION

I would like to thank my parents, and my fiancée for their never-ending patience, help, and support. Without their support this work would have never been completed.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

Software defects can be catastrophic. In particular, in the case of safety critical systems, software defects could lead to risks such as a loss of equipment or even a loss of human lives [36]. To date, many approaches were proposed to reduce the chances of such risks taking place. For example, risk-based test case generation techniques (e.g. [4], [5], [7]) utilize software fault models to guide the process of test case generation; hence, increasing the chances that risk-inducing flaws are discovered during testing. Risk-based test case selection techniques (e.g. [1], [2]) select and prioritize test cases based on the severity and the number of risks they address; therefore, enhancing the ability of the test suite to discover risk-inducing flaws.

Although these existing approaches produce test cases that can discover risk inducing flaws, they mainly target implementation flaws rather than flaws that can occur during an earlier stage of the software development process. Because the software flaws that are discovered in later stages of software development are more expensive to fix [19], discovering these flaws earlier reduces the cost of fixing them.

Few studies discuss discovering risk inducing flaws during the software design phase [12], [13]. However, these studies rely on manual techniques that are hard to scale to large complicated designs. Further, these studies target designs created in informal languages (e.g. UML state charts). Because Informal specification languages are less accurate and more error-prone than formal languages, safety critical designs are usually created in formal languages. Thus, these studies are unsuitable for safety critical applications.

To address these limitations, in this thesis we present an automated approach for discovering risk-inducing flaws in software behavioral designs. The proposed approach integrates LOTOS (e.g. Language of Temporal Ordering Specification) behavioral designs with

Fault Tree risk models and then uses the integrated model to look for risk-inducing flaws in the behavioral design. Because the proposed approach is automated, it can handle large complicated designs. Further, our approach considers software designs created in LOTOS that is a formal specification language. Thus it is suitable for safety critical systems that require high accuracy.

To evaluate the proposed approach, we performed an experiment using safety-critical systems' specifications. The specifications we used have a number of predefined risk-inducing flaws. The results show that our approach demonstrated all the design flaws in the designs that used event names consistently (used the same name to describe the same events throughout the design). However, the results show that if two different events are modeled using the same name in the design, the approach could fail to identify flaws that involve these events. Therefore, the results mainly show that as long as the design under test is consistent (e.g. names are used consistently throughout the design) the approach can discover risk-inducing flaws.

In the next chapter, we provide background information and related works relevant to risk-based testing and design flaw discovery approaches. Chapter 3 describes our proposed approach, Chapter 4 discusses the empirical study, Chapter 5 presents the results of the study and analysis, Chapter 6 contains a discussion of the results, and Chapter 7 presents conclusions and future work.

CHAPTER 2. BACKGROUND AND RELATED WORK

In this chapter we presents preliminary information necessary to the presentation of our approach, then we briefly survey existing related studies. The following section of the chapter discusses the fundamentals of Fault Tree models. Section 2.2 presents some fundamentals about LOTOS specification language. Section 2.3 surveys some related work. Finally, Section 2.4 compares our approach to using model checking to discover design flaws.

2.1. Fault Tree Models

The Fault Tree model is a model of the various parallel and sequential combinations of faults that will result in the occurrence of a predefined undesired event [20]. Fault Tree models are used for risk analysis in many aspects. Studies suggest leveraging Fault Tree models in various phases of software development including requirements specification, design, implementation, and testing [21, 23, 22, 24].

A Fault Tree breaks an undesired event or a risk to basic events that might lead to the occurrence of this risk. These basic events are combined using Boolean logic gates. Figure 2.1 shows an example of a Fault Tree. The Fault Tree describes the risk of a car accident. In this example, “Brakes Failure” and “Headlights Failure” are the basic event combined with an “OR” Boolean gate. Thus, the Fault Tree captures the fact that if there is a failure in the brakes or the headlights, an accident might take place.

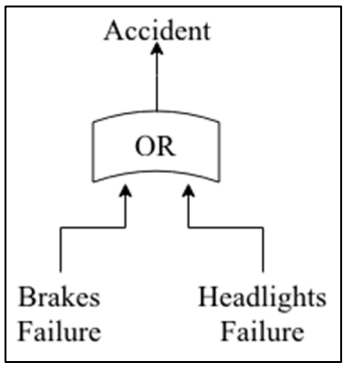

Figure 2.1. Fault Tree Example

2.2. LOTOS Specification Language

LOTOS is a Formal Description Technique (FDT) developed and standardized by ISO [25]. In LOTOS, "processes" are used to model the system under design. A LOTOS process describes the sequences of events and actions that are acceptable by the component which the process models [26].

Figure 2.2 shows a simple example of a LOTOS process. The example describes a system that receives two numbers and outputs their maximum. As demonstrated in the figure, a LOTOS specification does not show how the maximum is calculated but rather captures the allowed sequences of events and actions exposed by the system to its environment. In LOTOS, the symbol ";" is called the "action prefix" and denotes that the system is waiting to accept the next event or to produce the next action. The symbol "[]" represents a choice (e.g. a selection between two different execution paths).

```
Process Max(in1, in2, max) : noexit :=
        in1; in2; max
        []
        in2; in1; max
endproc (*Max*)
```

Figure 2.2. LOTOS Process Example

In this example “in1”, “in2”, and “max” are the events exposed by the process to its environment. The process “Max” allows two different execution paths based on which event takes place first. If the “in1” event occurs first, the system waits for event “in2” to occur then produces the max; but if event “in2” occurs first the system waits for event “in1” to take place then produces the max. These are the only allowed execution sequences, which means that the system should not accept any other sequence such as “in1; in1”. For more detailed discussion about LOTOS specification language please refer to [26] and [27].

2.3. Related Work

Risks are closely related to all activities of the software development process. As mentioned before, an unaddressed risk in software development could result in catastrophic outcomes. Thus, many researchers suggested approaches that utilize risk analysis in the activities of software development including specification, design, implementation, and testing. In this study, we limit our discussion to testing and design, as they are related to our work.

Risk-based test case selection techniques (e.g. [1], [2]) utilize risk analysis in selecting and prioritizing test cases in order to increase the possibility of addressing high risks in testing. For example, Chen et al. [1] suggest a risk-based model for test case selection. In this model, test cases with the highest risk exposure are selected as safety tests. Test exposure is calculated by multiplying the cost of the test case according to the severity of the risk this test case addresses. In another research, Amland et al. [2] discusses an approach to prioritize tests based on the probability of the occurrence of the risk they address, and the cost associated with this risk happening. Although these techniques increase the chances of discovering risk-inducing flaws, they fail to address the situation in which the existing tests do not address the risk in the first

place. Thus, these approaches are more suitable for regression testing rather than risk-based test generation.

Many studies propose using risk models for test-case generation [3], [4], [5], [6], [7]. For example, Nazier et al. [3] suggest an approach that integrates Fault Tree risk models with UML State chart diagrams for test case generation. This approach implements a user-created table that maps risk events to events in the state chart diagram to generate a risk-based test model. Querying language is then used to generate test cases from the integrated model. Also, Sanchez et al. [4] suggested transforming Fault Tree models to a set of Duration Calculus formulas; and using a set of transformation rules to integrate these formulas with UML state chart behavioral diagrams. The resulting integrated state chart is then transformed to a flattened Extended Finite State Machine (EFSM) and state coverage is used to generate test cases. Gario et al. [5] transforms Fault Tree models into a set of Communicating EFSMs (CEFSM) called Gate CEFMS (GEFSM). Afterwards, the GCEFSM is integrated with the behavioral CEFSM for test case generation. Although such studies prove effective in testing the implementation for risk-inducing bugs, they fail to address risk-inducing flaws introduced in the design phase. This allows risk-inducing flaws in the design to make to the implementation, which makes them more expensive to fix and increases the chances that such flaws make it to the end product.

Moreover, many studies discuss testing software designs for flaws. Many of these studies use object-oriented metrics to guide the flaw-detection process [8], [9], [10], [11]. For example, Marinescu [8] suggested using “Weight of Class” (WOC), “Number of Public Attributes” (NOPA), and “Number of Accessor Methods” (NOAM) as heuristics to guideline design flaws detection. In another study, Miceli [9] used a set of inheritance metrics such as “Depth of Inheritance Tree” (DIT), and “Class to Leaf Depth” (CLD) among other metrics to detect and

correct design flaws. Also, approaches that do not utilize OO metrics were suggested. Mekruksavanich et al. [8] propose creating a declarative Prolog meta-model that describes the object-oriented system under test. By running a set of inference rules against this meta-model, they can detect certain types of design flaws. Such studies address the problem of non-functional design flaws related to quality aspects (e.g. maintainability, and reusability, etc.), but they fail to discover functional design flaws that expose risks due to the software not behaving correctly.

Only a few studies address the problem of functional risk-inducing flaws introduced in the design phase of software development [12], [13]. Kim et al. [12] suggest using behavioral UML State chart diagrams to transform the Fault Tree to an UML State Chart model that describes the behavior of the system under risk conditions. This model resulting from the transformation is, then, used to manually look for risk inducing design flaws. El Ariss [13] suggests a method for integrating Fault Tree models with State Chart behavioral models, then using the integrated model to manually look for functional risk-inducing design flaws. However, these studies offer manual solutions that could prove inefficient and error prone in case of large complicated systems that has a large number of components. In addition, they offer no clear solution to discover flaws so it is left to the engineer to decide the method to use for discovering flaws. Furthermore, these studies rely on informal specification language (e.g. State Chart Diagrams), which is inherently error prone and not suitable for modeling safety critical systems. To further inform these studies, in this paper we suggest an automated approach to test formal LOTOS designs for risk-inducing functional flaws.

Model checking is a design verification method that is used to verify a desired behavioral property over a given model [37]. In other words, model checking is used to verify that a given behavioral property is never violated by the design under test. These behavioral properties are

usually specified using Computational Tree Logic (CTL) expressions. The CTL formulas describe properties of the computation tree of the model [37]. The computation tree is the tree that starts with the initial state of the system at the root and then lists all the possible state reachable from that state of the system. Construction and Analysis of Distributed Processes (CAPD) offers a tool called "XTL" [38]. This tool is able to check LOTOS models against properties described in CTL expressions. Hence, this tool could be used to check whether a LOTOS design violates a given property.

However, using model checking to check if the design has risk-inducing flaws has disadvantages. First, without providing a model that specifies the risks that should be prevented, model checking cannot be used to mitigate risks solely; as the risks that might be associated with the behavior could be unknown so formulating properties to describe those risks could be impossible. Second, when the number of the concurrent components of the modeled system increases, the number of the states – in which the system can be –increases exponentially; this problem is known as state explosion [39]. As model checking methods rely on enumerating the state space in order to verify the system against properties, they fail to scale to systems with large number of concurrent components due to state explosion. Finally, using model checking to search for design flaws requires knowledge of formal methods or CTL expressions. This makes model checking less user-friendly.

On the other hand, the approach that we suggest in this paper addresses these disadvantages. First, our approach combines Fault Tree models that specify risks with the behavior models. By combining models that specifies risks that should be prevented, we can identify risk-inducing behavior and therefore mitigating these risks. Second, the time complexity of our approach depends linearly on the number of basic events in the Fault Tree model of the

system under test rather than the number of states in the behavioral design. Hence, our approach is not subject to the state explosion problem and scale to systems with large number of concurrent components. Finally, our approach does not require any additional knowledge from the user. The approach incorporates Fault Tree models, which are simple Boolean expression, to look for risk inducing design flaws. This makes our approach easy to use and user-friendlier than model checking.

CHAPTER 3. PROPOSED APPROACH

In this chapter, we describe an algorithm that transforms a given Fault Tree to a LOTOS process, and then integrates this process with a given behavioral LOTOS model. We, then, present a method that uses the integrated LOTOS specification to discover risk inducing design flaws.

3.1. Overview

Fault Tree models are combinatorial models; combinatorial models map given inputs (risk factors in case of Fault Tree models) to an output (the modeled risk in case of Fault Tree models), so the output depends only on the input regardless the previous inputs or the state of the system. This means that Fault Tree models provide no means to model sequences of actions and temporal orders of states and events [20], [28]. In addition, LOTOS behavioral models mostly model event-driven sequential systems. This means that the modeled system behavior depends on the input, and the current state of the system (e.g. previous inputs). This difference in nature makes the two models initially incompatible. Thus, the integration process requires more than just implementing a merge operator between the two models. Figure 3.1 shows an activity diagram describing the overview of the integration process.

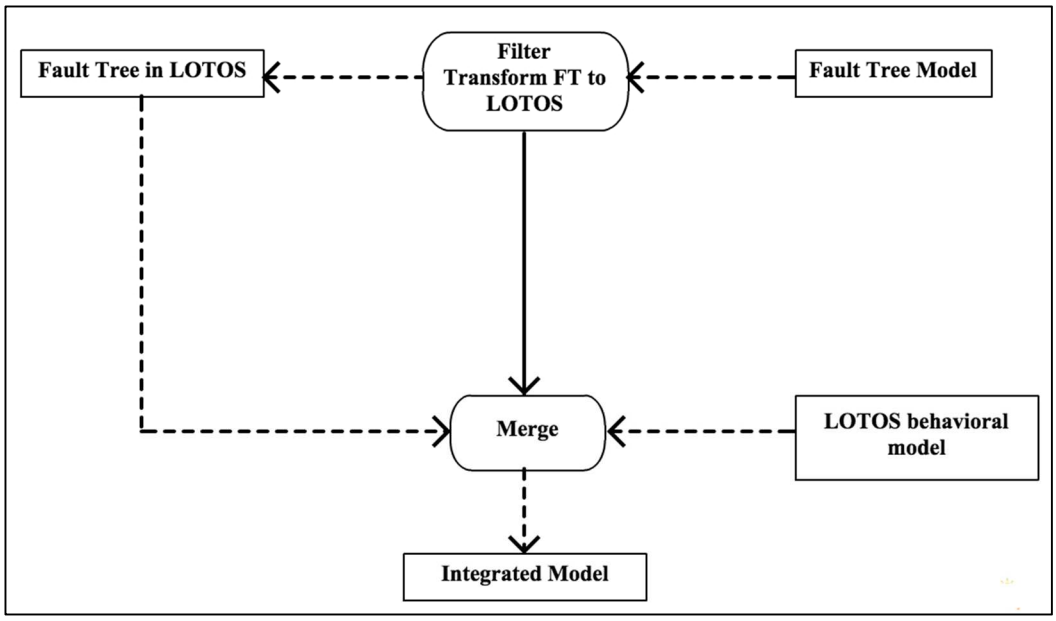

Figure 3.1. Approach Overview

To be able to integrate two models, the models must be compatible. Model compatibility must be achieved on many levels. First, the models to be integrated must be captured in the same specification language (e.g. syntax compatibility). Second, the models should be on the same abstraction level, and the same entities should be modeled with the same name in both models (e.g. semantic compatibility). Semantic level compatibility was discussed in previous research [5] and should be considered before model creation, since models created at different levels of abstraction could not be integrated. Thus, in this study, we consider syntax compatibility between the models to be integrated and we assume that the input models are semantically compatible, meaning that both models use the same names for the same events, and they are on the same abstraction level.

As the Fault Tree model initially composes of a tree of binary gates but the behavioral model is assumed to be in LOTOS, the two models are initially incompatible. In order to integrate the two models, we transform the Fault Tree model to an equivalent LOTOS process; thus, achieving syntax compatibility between the two models.

The second phase of the integration is merging the two models. In this phase, we implemented a merge operator that relies on the similarity between event names in the two models to carry out the integration. The output of this phase is an integrated model that allows events that are common between the two models (e.g. events used in both models) to be shared between models. Meaning that if an event is used is both models, a single event-bus (e.g. event gate) is created and shared between the two models.

Throughout this chapter, we will be using an example to illustrate the steps of the integration process. The example models, a back up assistance system of a vehicle, which is used to activate the brakes of the vehicle when it is backing up, in case there is an object behind the vehicle. Figure 3.2 shows the state-chart diagram of the system. The backup assistance system consists of two communicating state machines (the object detection state machine and the braking system state machine) linked by a communication link (m_o) that carries output messages from the object detection system to the braking system. The object detection state machine listens to "object" events that indicate whether there is an object behind the vehicle. When the object event is true – indicating that the sensor is detecting an object behind the vehicle – the object detection sends a "Brake" output message to the braking system state machine. When the object is no longer behind the vehicle, the "object" event becomes false, and the object detection state machine sends a "Release Brake" output message to the braking system state machine. The braking system state machine listens to the "brake" event; if a "Brake" message is received (e.g. meaning the "brake" event is true) it activates the vehicle's brakes, but if a "Release Brake" message is received (e.g. the "brake" event is false) it deactivates the vehicle's brakes.

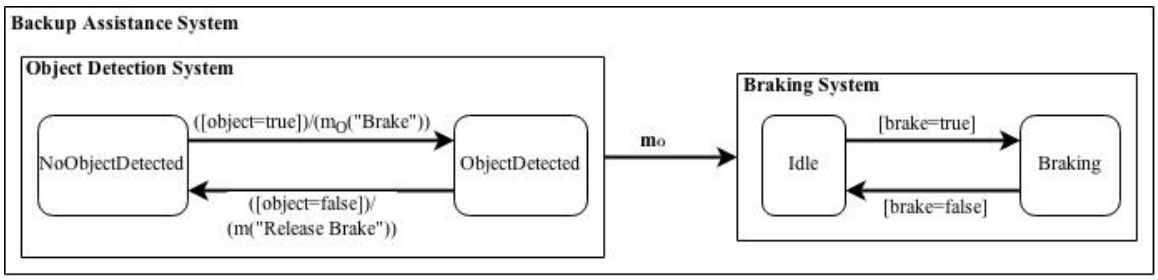

Figure 3.2. Backup Assistance System State-chart Diagram

The LOTOS specification of the backup assistance system is given in Figure 3.3. In the specification, each component of the backup assistance system (e.g. object detection system, braking system) is modeled in LOTOS format. The specification exposes the "object" event gate to the environment, as this event gate is used for communicating events between the sensor and the obstacle detection component. On the other hand, the specification hides the "brake" event gate from the environment, as it is an internal event used to synchronize the obstacle detection component and the braking system component.

```
specification BackupAssistance[object] : noexit
        library
                Boolean
        endlib
        behavior
                hide brake in
                (
                                ObstacleDetection[object, brake]
                                |[brake]|
                                BrakingSystem[brake]
                )
        where
                process ObstacleDetection[object, brake] : noexit :=
                        Idle[object, brake]
                where

                        process Idle[object, brake] : noexit :=
                                object!true;brake!true;ObstacleDetected[object, brake]
                        endproc(*Idle*)

                        process ObstacleDetected[object, brake] : noexit :=
                                object!false;brake!false;Idle[object, brake]
                        endproc(*ObstacleDetected*)
                endproc(*ObstacleDetection*)

                process BrakingSystem[brake] : noexit :=
                                Idle[brake]
                where
                        process Idle[brake] : noexit :=
                                brake!true;Braking[brake]
                        endproc(*Idle*)

                        process Braking[brake] : noexit :=
                                brake!false;Idle[brake]
                        endproc(*Braking*)
                endproc(*BrakingSystem*)
endspec(*BackupAssistance*)
```

Figure 3.3. Backup Assistance System LOTOS

One risk associated with the backup assistance system is that if an object is detected but the brakes are not activated, an accident might take place. Figure 2.4 shows the Fault Tree model that captures the basic events that leads to the accident risk.

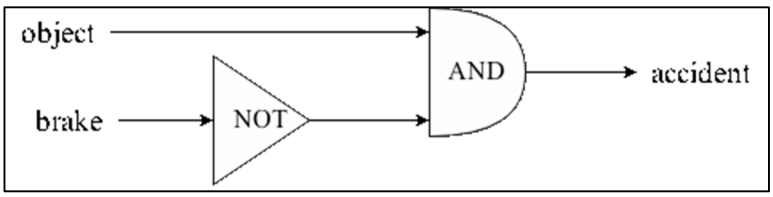

Figure 3.4. Accident Fault Tree

3.2. Transforming The Fault Tree Model to A LOTOS Process

To transform Fault Tree models, we created a transformation table that guides the transformation process. This transformation table maps each of the Boolean logic operators to an equivalent LOTOS process that has the same behavior as the logic gate. These LOTOS processes are created using the truth tables of the corresponding logic gates. As an example, Table 3.1 shows the truth table of the "AND" logical operator and the equivalent behavior in LOTOS. As shown in the table, each different pair of inputs is translated to a LOTOS expression by parameterizing the LOTOS event with a Boolean variable that represents whether the event occurred. Table 3.2 shows the transformation table that maps each logic gate to an equivalent LOTOS process. As shown in the table, the equivalent processes are created by combining all the possible execution paths – produced from the truth table of the corresponding gate – using LOTOS choice construct "[]".

Table 3.1. AND Gate Transformation Table

First Operand	Second Operand	Output	Equivalent LOTOS Expression
True	True	True	event1!true; event2!true; output!true; stop [] event2!true; event1!true; output!true; stop
True	False	False	event1!true; event2!false; output!false; stop [] event2!false; event1!true; output!false; stop
False	True	False	event1!false; event2!true; output!false; stop [] event2!true; event1!false; output!false; stop
False	False	False	event1!false; event2!false; output!false; stop [] event2!false; event1!false; output!false; stop

Table 3.2. Boolean Logic Gates Transformation Table

Logic Gate	Operator Symbol	Equivalent LOTOS process
AND	&	process AND [event1, event2, output] : noexit := event1!true; (event2!true; output!true; stop [] event2!false; output!false; stop) [] event2!true; (event1!true; output!true; stop [] event1!false; output!false; stop) [] event1!false; (event2!true; output!false; stop [] event2!false; output!false; stop) [] event2!false; (event1!true; output!false; stop [] event1!false; output!false; stop) endproc

Table 3.2. Boolean Logic Gates Transformation Table (continued)

Logic Gate	Operator Symbol	Equivalent LOTOS process
OR	|	process OR [event1, event2, output] : noexit := event1!true; (event2!true; output!true; stop [] event2!false; output!true; stop) [] event2!true; (event1!true; output!true; stop [] event1!false; output!true; stop) [] event1!false; (event2!true; output!true; stop [] event2!false; output!false; stop) [] event2!false; (event1!true; output!true; stop [] event1!false; output!false; stop) endproc

Table 3.2. Boolean Logic Gates Transformation Table (continued)

Logic Gate	Operator Symbol	Equivalent LOTOS process
XOR	^	process XOR [event1, event2, output] : noexit := event1!true; (event2!true; output!false; stop [] event2!false; output!true; stop) [] event2!true; (event1!true; output!false; stop [] event1!false; output!true; stop) [] event1!false; (event2!true; output!true; stop [] event2!false; output!false; stop) [] event2!false; (event1!true; output!true; stop [] event1!false; output!false; stop) endproc
NOT	!	process NOT[event, output] : noexit := event!true; output!false; stop [] event!false; output!true; stop endproc(*NOT*)

A study by Turner and Sinnott [29] discusses a method that combines Boolean gates to be able to build LOTOS specification that describes digital logic circuits. In our transformation algorithm, we adapted this method to combine the LOTOS logic processes in order to build fault

trees. The method combines two LOTOS processes – that represents logic gates – by using the LOTOS synchronization operator "|[ ]|" to create a common event bus, which can be used as output bus by one process and input bus by the next process. For example the Boolean logic expression "AND (event1, NOT (event2))" is presented in LOTOS as "NOT [event2, NotOutput] |[NotOutput]| AND [NotOutput, event1, AndOuput]". In this case the "NotOuput" event bus (e.g. event gate) is used by the "NOT" process as output bus and used by the "AND" process as input bus; thus, connecting the two processes.

We used these approaches to build the transformation algorithm shown in Figure 3.5. The transformation algorithm traverses the gates of the fault assigning a unique ID for each gate; these IDs are used to link the output of one gate to following gate up the fault tree. After assigning an ID to each gate, the algorithm traverses the tree in pre-order fashion. For each gate we traverse, we construct an equivalent LOTOS expression.

Figure 3.6 shows the LOTOS process resulting from transforming the accident Fault Tree model from the example discussed in the previous section. As the figure shows, the algorithm traversed the tree assigning id "1" to the "NOT" gate, and id "2" to the "AND" gate. The algorithm then traverses the tree in pre-order fashion. It instantiates the "AND" process first, and then starts processing its inputs. Since the first operand of the "AND" gate is the leaf event "object", this operand is not further processed. For the second operand (the "NOT" gate), the algorithm instantiates the corresponding LOTOS process using the gate inputs (in this case the "brake" event), and uses the gate id to create the name of the output event (e.g. "NOT1"). Also, the algorithm links the gates using the output event names; as Figure 3.6 shows, the event "NOT1" is used to synchronize the "AND" process and its child "NOT" process. After creating

the Fault Tree LOTOS process, the algorithm appends the definition of the LOTOS processes given in Table 3.2 that corresponds to the logic gates.

```
Procedure FT_TO_LOTOS (FaultTree : ft)
{
        assignIdsToGates(ft);
        String faultTreeBehavior = "";
        Gate rootGate = ft.getRootGate();
        addGateToLOTOSBehavior(faultTreeBehavior, rootGate);
        addLogicGatesProcessDefnitions(faultTreeBehavior);      //adds the process
                                                                //definitions given in
                                                                //Table 3.1 to the
                                                                //Fault Tree process
}
addGateToLOTOSBehavior( faultTreeBehavior : String, gate : Gate)
{
        instantiateCorrespondingProcess(gate); //instantiate the gate process
                                               //(e.g. AND, OR,...)
                                               // Using the gate input names and
                                               // Uses the gate Id to create the name
                                               // of the output.
        Input[] inputs = gate.getInputs();
        for each input in inputs
        {
                if( input is of type Gate)
                {
                        addGateToLOTOSBehavior(faultTreeBehavior, input);
                }
        }
}
```

Figure 3.5. Transformation Algorithm

```
process FaultTreeModel [object, brake, accident] : noexit :=
(
        (
                accident[ brake, object, accident]
        )
)
where
        process accident [brake, object, AND2] : noexit :=
        (
                hide NOT1 in
                (
                        AND[object, NOT1, AND2]
                        |[NOT1]|
                        (
                                NOT[brake, NOT1]
                        )
                )
        )
        endproc (*accident*)
        process AND
                                …
endproc(*FaultTreeModel*)
```

Figure 3.6. Backup Assistance Fault Tree LOTOS Process

3.3. Integrating Fault Tree LOTOS Process and Behavioral LOTOS Specification

By transforming the Fault Tree model to a LOTOS process, we have achieved compatibility between the LOTOS behavioral model and the Fault Tree model; thus, the two models can be integrated.

To guide our description of the event integration algorithm, we will use the process structure diagrams given in Figure 3.7, which describes the process structure of the backup assistance system behavioral model and the accident fault tree process. In the process structure diagram, the specification is represented by a dotted square, each process is represented by a

solid square, and event gates are represented by ovals on the boundaries of the square representing this process.

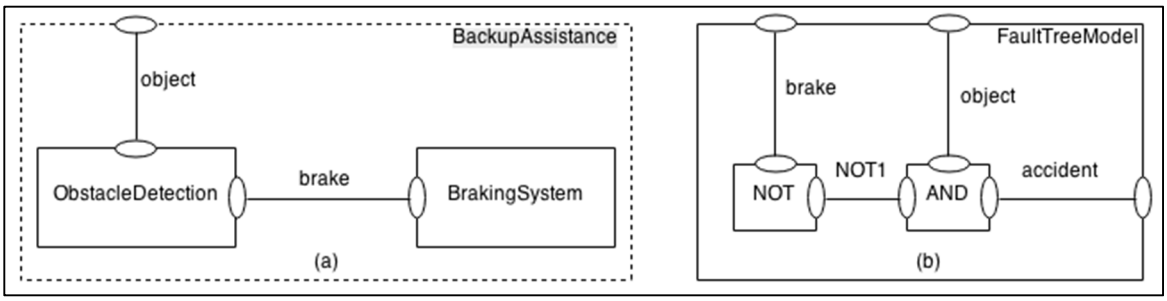

Figure 3.7. The Process Structure of (a) The Backup Assistance Behavioral Model, and (b) The Accident Fault Tree LOTOS Process

To perform the integration, we iterate over the list of input events of the Fault Tree LOTOS process, integrating one event at a time with the behavioral specification. For any input event of the fault tree, one of the following is true:

i. The event does not have a similar event in the behavioral model, in which case the event should be used as an input to the integrated model's specification coming from the environment.

ii. The event has a similar unhidden (e.g. exposed or public) event in the behavioral model, in which case the two gates from the behavioral model and the fault tree model are connected directly by using the LOTOS synchronization operator (e.g. |[common_event]|). The "object" event in the backup assistance system is an example of this case.

iii. The event has a similar event hidden – in the behavioral model's specification or in one of it's sub-processes, in which case the hidden event is exposed or made public first then it is connected to the fault tree event using LOTOS synchronization operator. An example of such case is the "brake" event in the backup assistance system.

Figure 3.8 shows the process structure of the integrated model. As the process structure shows, the “brake” event in the backup assistance model was hidden before integration; in the integrated model, the “brake” event is exposed as an interface gate of the backup assistance process. After exposing the brake event, it is used to synchronize the backup assistance process and the fault tree model process. It should be noted that even though the brake event is exposed in the backup assistance process, it is still hidden in the integrated model’s specification allowing the external behavior of the modeled system to be preserved. The “object” event was unhidden in the backup assistance specification; as a result, the event was directly used to integrate the backup assistance specification and fault tree model process. Also, the “object” event remains unhidden in the integrated model’s specification interface.

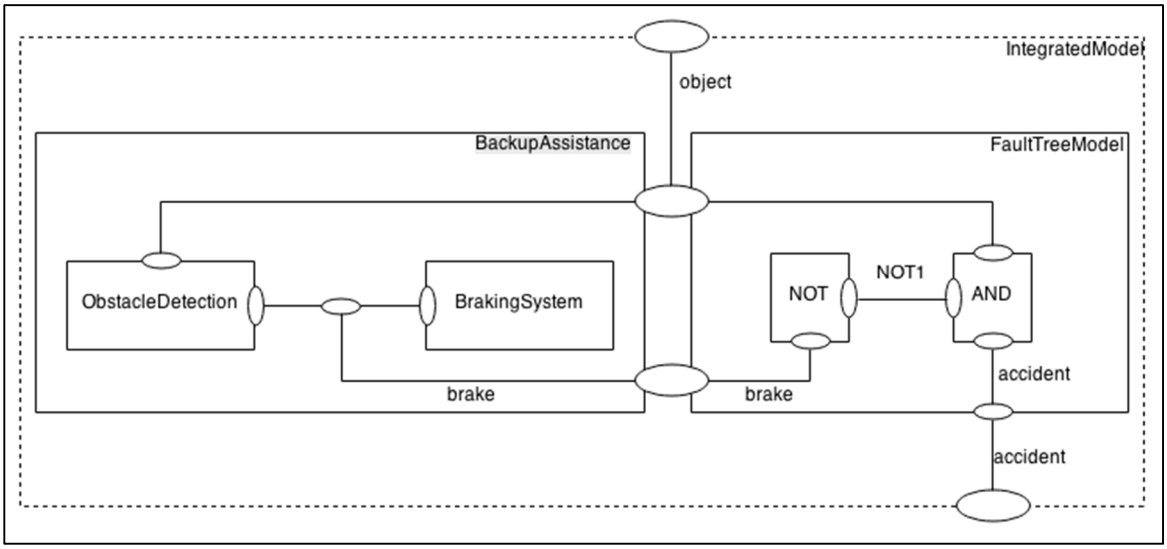

Figure 3.8. Backup Assistance Integrated Model's Process Structure

```
specification IntegratedModel [object,accident]  : noexit
library
        Boolean
endlib
behavior
        hide brake in
        (
                BackupAssistance [object, brake]
                        |[object, brake]|
                FaultTreeModel [object, brake, accident]
        )
where
        process BackupAssistance [ object, brake ]  : noexit :=
                (
                                ObstacleDetection[object, brake]
                                |[brake]|
                                BrakingSystem[brake]
                )
        where
                        …
        endproc (*BackupAssistance*)
        process FaultTreeModel [object, brake, accident] : noexit :=
                        …
        endproc(*FaultTreeModel*)
endspec
```

Figure 3.9. Integrated Model in LOTOS

Figure 3.9 shows the LOTOS specification of the integrated model. To build the LOTOS specification of the integrated model, the integration algorithm transforms the behavioral model's specification to a LOTOS process. The algorithm then creates the behavior of the integrated model's specification by synchronizing the instantiation of the behavioral model's process and the fault tree model's process using all common events (e.g. input events of the fault tree model that has similar events in the behavioral model).

3.4. Testing Design for Risk Inducing Flaws

In the integrated LOTOS specification, the behavior and the associated risks are combined into a single model that has the same set of input events. This means by providing one set of inputs, one can simulate behavior of the system, and see whether any of the risk events is to occur. For example, this could be applied to the integrated model shown in Figure 3.9 by passing the values “object !true” and “object !false” to the system, and testing whether the accident event becomes “true”.

If the behavioral design addresses associated risks correctly, the risk events should never be true regardless of the value of the input events. For example, in the case of the Backup Assistance System, the event “accident !true” should never be produced in the integrated model given in Figure 3.9, regardless whether the “object” event is true or false. Thus, the integrated model should not have any execution sequences that ends with one of the risk events being true, if the behavioral design correctly mitigates the risks listed in the given fault tree model. Therefore, any execution sequence ending with one of the risk events being true represents a risk-inducing design flaw. Hence, by searching the integrated model for action sequences that ends with one of the risk events being true, one can discover risk-inducing flaws in the behavioral design.

In other words, with $I(e_n)=\{s_1=(e_i;\ldots;e_n), s_2=(e_j;\ldots;e_n), \ldots\}$ being a function that returns the set of execution sequences ending with the event e_n (e.g. s_1, s_2), and r_i being the event of the risk being true. If the design addresses the risk correctly, then the result of running the search function should be $I(r_i)=\{\phi\}$ (e.g. an empty set). On the other hand, if the result of running the search function is $I(r_i)=\{df_1, df_2, \ldots, df_n\}$, then the set of execution sequences $\{df_1, df_2, \ldots, df_n\}$ represents the set of risk-inducing flaws in the design. So in the case of the Backup Assistance

System, the result of I(“accident !true”) represents the faulty execution sequences in the design that might induce an accident.

Many tools exist for searching LOTOS specifications for execution sequences. One tool that we utilizes in the next chapter to evaluate our approach called “exhibitor” [30]. “Exhibitor” is offered as part of the “Construction and Analysis of Distributed Processes” (CADP) toolset that offers many functions for verifying and manipulating LOTOS specifications. The “exhibitor” tool transforms the LOTOS specification to a labeled transition system (LTS), and then uses graph search algorithms (e.g. breadth first search, and depth first search) to search the produced LTS for execution paths that ends with a given event (e.g. label).

CHAPTER 4. EMPIRICAL STUDY

To evaluate the proposed approach that we presented in Chapter 3, we performed an empirical study considering the following research questions:

RQ: Can the proposed approach discover risk-inducing design flaws?

The following subsections present the objects we used for the evaluation, and the setup of the experiment. In the next chapter, we present our data and analysis, and in Chapter 6, we discuss the practical implications of the results.

4.1. Objects of Analysis

We considered four functional specifications of control systems adapted from related research as our objects of analysis: the specification of a computer-controlled gas-burning unit [5], [13], the specification of a railroad-crossing control system [31], the specification of an aerospace vehicle launch control system [32], [33], and the specification of an insulin pump control system [34]. In order to use these specifications for our study, we translated the state chart diagrams of the specification to LOTOS.

Table 3.1 lists metrics that describe the size and complexity of the specifications we considered. The metrics shown are: the total number of transitions that represents the sum of all transitions in the specification's state chart, the number of components that represents the number of communicating extended finite state machines (EFSMs) within the specification, and the number of associated risks that represents the number of risks or Fault Trees in the associated Fault Tree model.

Table 4.1. Objects of Analysis

Specification	Total Number of Transitions	Number of Component EFSMs	Number of Associated Risks
Railroad-Crossing Controller	21	4	1
Gas Burner Controller	12	5	1
Aerospace Launcher Controller	20	5	4
Insulin Pump Controller	18	4	1

The following subsections describe the specifications we used in our empirical study.

4.1.1. The Gas Burner Specification

We adapted the gas burner specification from [5], [13]; the specification captures the design of a unit that controls a gas combustion system. Figure 4.1 shows the state chart of the gas burning system. The system consists of five components:

- The "controller" unit receives a heat request from the environment and controls the air, gas, and ignition inputs accordingly.
- The "Air Valve" enables and disables air input based on the controller's input.
- The "Gas Valve" enables and disables the gas input based on the controller's input.
- The "Igniter" is responsible for producing spark to ignite the gas input.
- The "Flame Detector" senses whether a flame is produced from ignition.

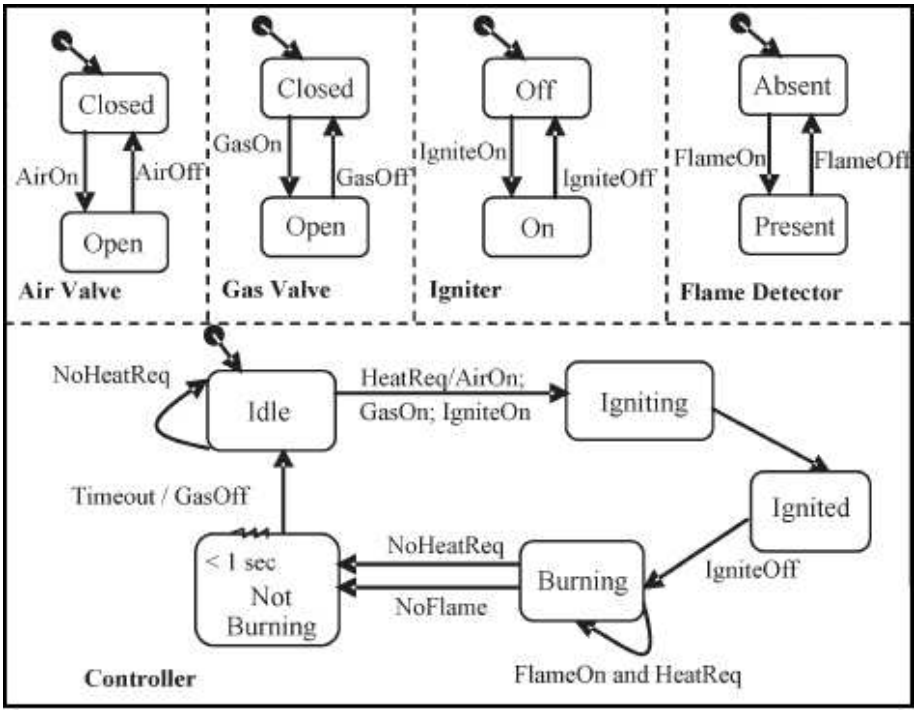

Figure 4.1. The Gas Burning System State Chart

The Fault Tree model that captures the risk associated with the gas burning system is shown in Figure 4.2. The "Fire" risk – documented by the Fault Tree model – occurs when air is present and there is a gas leak combined with the spark produced from the ignition and electric short in the cables.

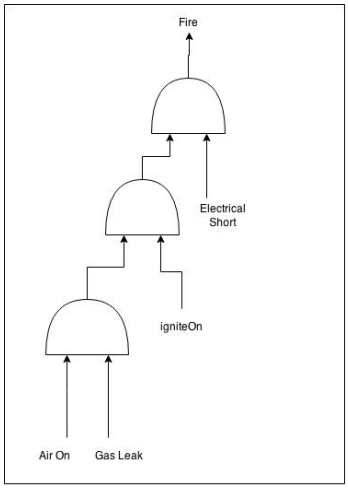

Figure 4.2. Gas Burner System - Fire Risk Fault Tree

4.1.2. Railroad-Crossing Specification

The railroad-crossing controller specification is adapted from [31]; the specification describes a control unit that automates the movement of a railroad-crossing gate based on a sensor's signal that detects approaching trains. Figure 4.3 shows the state chart diagram of the railroad-crossing control unit. The railroad-crossing system consists of four components:

- A train sensor that detects approaching trains and sends an activate signal to the controller when a train is crossing.
- A controller that controls the railroad-crossing gate and the warning light based on the message coming from the sensor.
- The gate motor unit that opens or close the crossing gate based on the message coming from the controller
- The warning light controller that turns the warning light on or off based on the message coming from the controller.

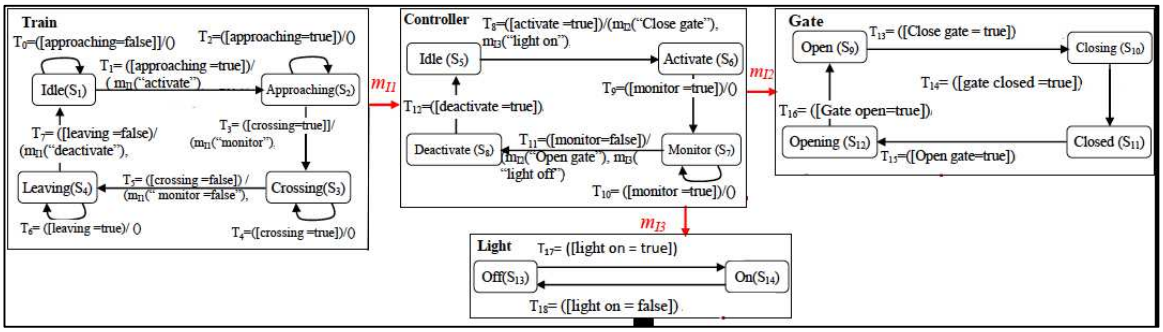

Figure 4.3. Railroad-Crossing System State Chart

Figure 4.4 shows the Fault Tree model describing the "accident" risk. The Fault Tree model captures the fact that if the controller is deactivated or the gate is open and the warning light is off, while a train is approaching or crossing; a car might attempt to cross the railroad causing an accident.

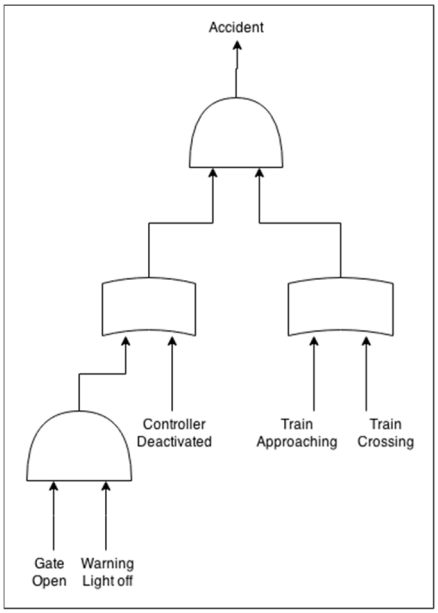

Figure 4.4. Railroad-Crossing - Accident Fault Tree

4.1.3. Aerospace Launcher Specification

The aerospace launcher specification is adapted from [32], [33]; the specification describes the launch controller unit responsible for initiating the launch sequence, and verifying the safety of the launch control system throughout the launch of an aerospace vehicle. Figure 4.5 shows the state chart diagram of the aerospace launch controller. The control system consists of the following five units:

- The initialization unit (launch conductor) that checks the network connection to the ground station. If there is no problem with the connection, the unit turns on the hazard lights, and starts the count down to the launch.
- The environmental control system (ECS) that solicits weather briefing then starts the air condition.
- The preflight unit that performs instrumentation checking, and begins the Cryogenic tanking.

- The fuel check unit verifies the launch vehicle's Liquid Oxygen (LO2) as well as the upper stage's Liquid Hydrogen (LH2).
- The flight unit starts the flight.

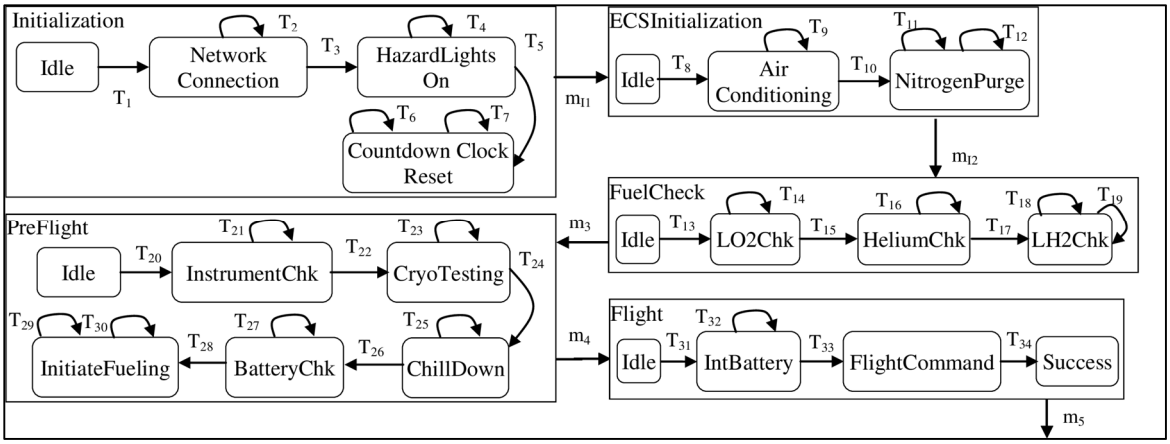

Figure 4.5. Aerospace Launcher State Chart

There are multiple risks associated with launching an aerospace vehicle. Figure 4.6 shows the Fault Tree of the "initialization fail" risk. Initialization could fail due to any of the following reason causing catastrophic outcomes:

- A network connection failure that results in the launch being cancelled or delayed.
- A countdown clock failure results in synchronization failure causing a tank to be over/under filled, thus, producing an explosion.
- A failure in the hazard lights that are used for safety around a launch vehicle compromises the safety of personnel.

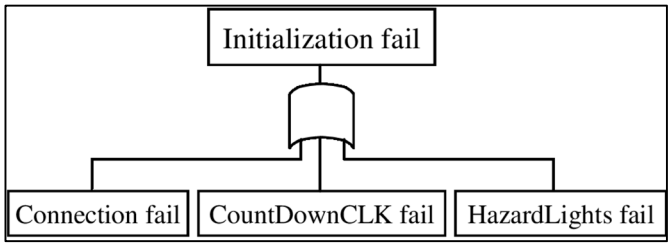

Figure 4.6. Aerospace Launcher - Initialization Fail Fault Tree

Another major risk is the occurrence of a fire. Figure 4.7 shows the Fault Tree of the fire risk. A fire could take place due to a leakage or over or under pressure in the tanks of LO2, Helium, or LH2.

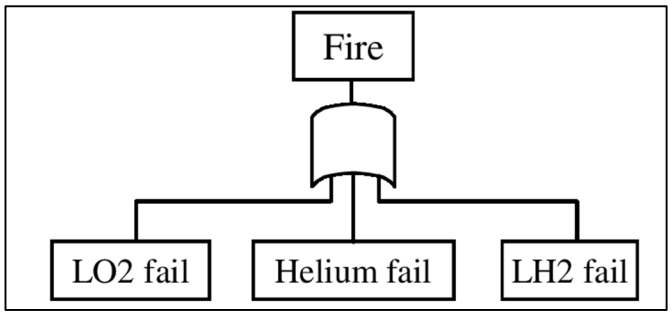

Figure 4.7. Aerospace Launcher - Fire Fault Tree

Also, the preflight check failure poses a risk to the flight safety. Figure 4.8 shows the fault tree of the preflight failure risk. The preflight failure risk could take place for one of the following reasons:

- A failure in the batteries due to bad conditions, low voltage, or low life expectancy.
- A failure in fuel initialization (e.g. such as low fuel pressures or bad valves).
- A failure in the batteries switch causing a failure to switch from external power to internal power, which is mandatory for launching.

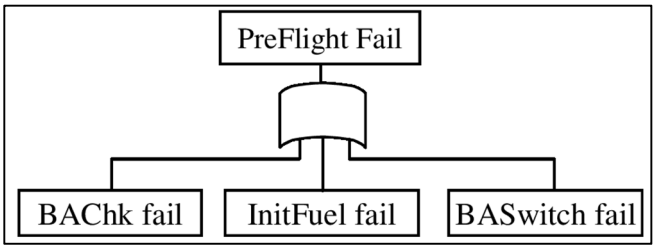

Figure 4.8. Aerospace Launcher - Preflight Failure Fault Tree

Finally, a failure in the launching system is another risk associated with the aerospace launcher. Figure 4.9 shows the Fault Tree of the launch failure risk. A launch failure could be due to either a failure in the ECS or an internal failure.

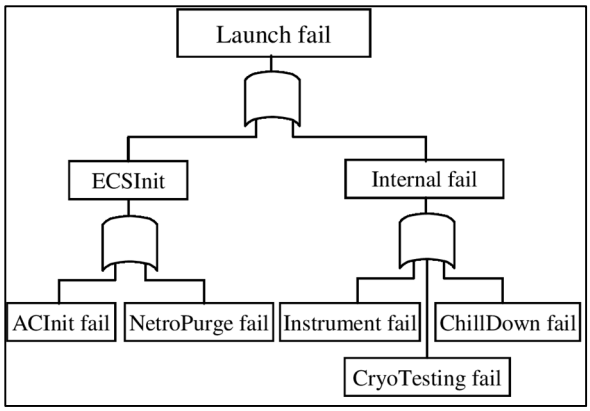

Figure 4.9. Aerospace Launcher - Launch Fail Fault Tree

4.1.4. Insulin Pump Specification

We adapted the insulin pump specification from [34]; the specification captures the design of an insulin pump that controls the sugar level in the blood. Figure 4.10 shows the state chart of the gas burning system. The system consists of four components:

- A sensor component that measures the level of sugar in blood and sends it to the controller.
- The controller unit decides the dosage of insulin to be administered based on the level of sugar in the blood sent by the sensor.
- The pump unit administers the insulin to the patient's blood based on the message sent by the controller.
- The alarm unit warns the patient when the level of sugar in blood is high or very high.

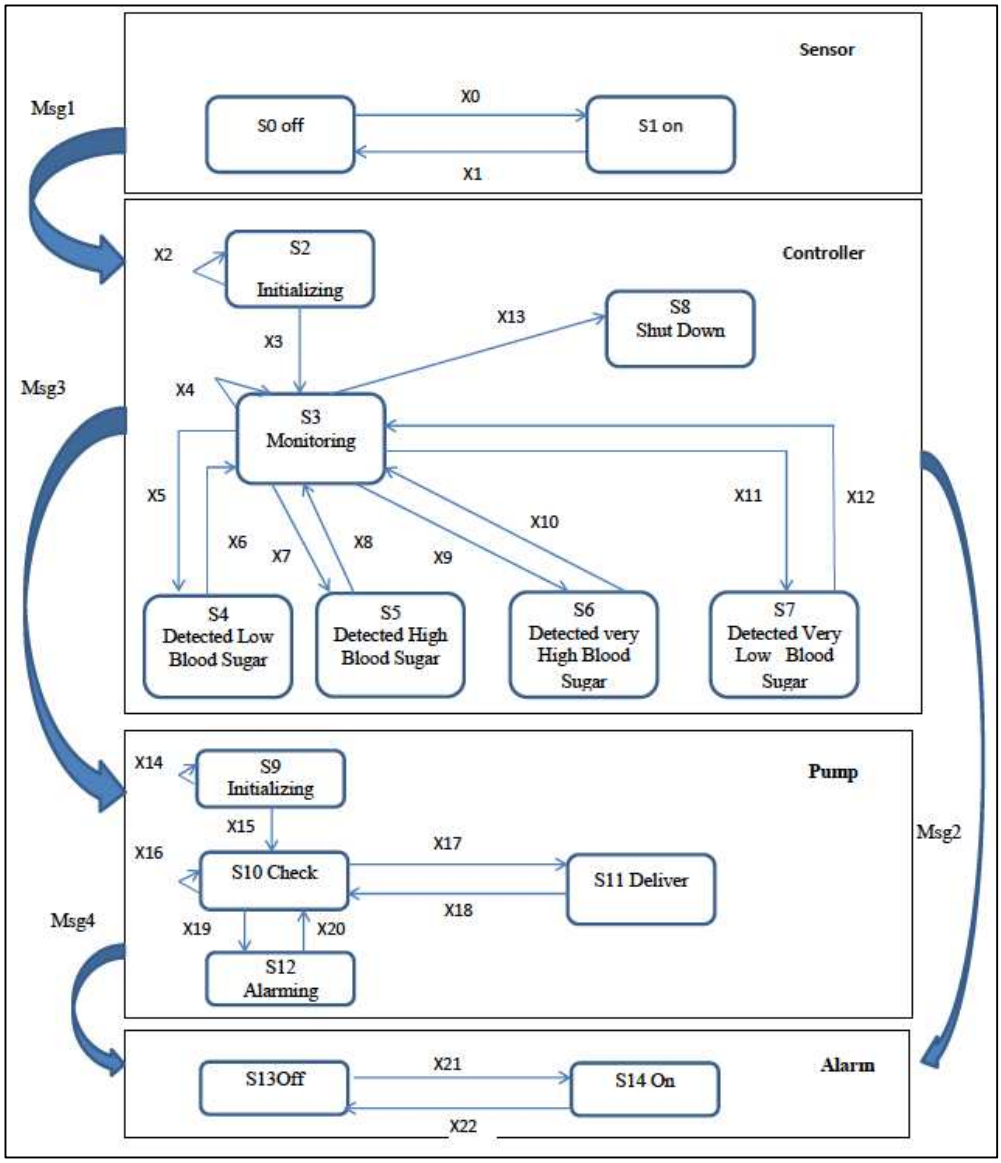

Figure 4.10. Insulin Pump State Chart

Figure 4.4 shows the Fault Tree model describing the "Over/Under Dose" risk associated with the specification. The Fault Tree captures the fact that a wrong dosage of Insulin could be administered in one of the following conditions:

- The battery of the pump is low, causing the pump not to receive messages from the controller correctly.

- If monitoring is not function correctly, the controller will not detect high/low levels of sugar in a timely fashion.
- If the pump has a malfunction, it will fail to administer the correct Insulin dose even though it received the controller's message.

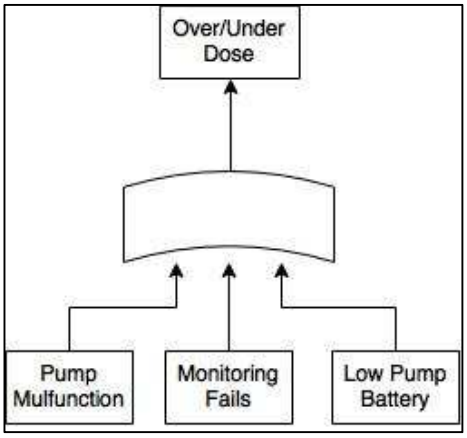

Figure 4.11. Insulin Pump - Wrong Dose Fault Tree

4.2. Measures

To investigate our research question, we recorded the number of flaws discovered for each of the specifications we used in the experiment. By comparing the number of flaws discovered by the approach to the number of flaws known to exist in each specification, we are able to measure the ability of the approach to discover risk-inducing flaws. If a large portion of the flaws were discovered, then the approach is capable of discovering risk-inducing flaws. On the other hand, if the approach does not report any flaws even though flaws are known to exist, then the approach is not capable of discovering risk-inducing flaws.

4.3. Setup and Procedure

To evaluate our methodology, we created a tool that implements the integration algorithm previously discussed. The implementation is 3.4 KLOC written in Java. Figure 3.1 shows an

activity diagram describing the implementation. The implementation consists of three main modules:

- A “parser” that parses the Fault Tree model files into Java objects, and detects syntax errors in the behavioral LOTOS specifications.
- A “filter” that implements the transformation algorithm – discussed in the previous chapter – to transform the “FaultTreeModel” object to a LOTOS process.
- An “integrator” that integrates the Fault Tree model LOTOS process with the given behavioral LOTOS specification.

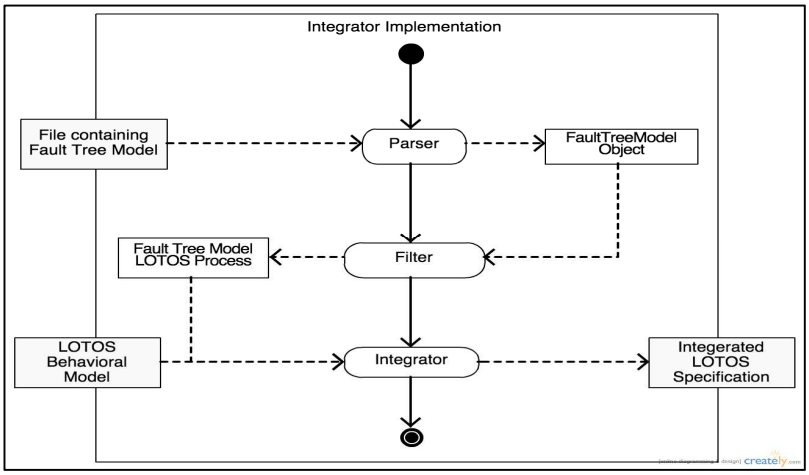

Figure 4.12. Integrator Implementation Activity Diagram

The implementation takes two inputs: a text file containing the Fault Tree model in the Backus-Naur form given in Figure 3.2, and a “.lotos” file containing the LOTOS behavioral specification of the system under test. Note that the “NL” label used in the Backus-Naur form represents the “new line” character. The tool then produces a “.lotos” file containing the integrated specification.

<fault-tree-model> ::= <fault-tree-list>
<fault-tree-list> ::= <fault-tree>NL<fault-tree-list>
<fault-tree> ::= <risk-name> “=” <logic-gate>
<logic-gate> ::= “(“ <logic-operator> <operands-list> “)”
<logic-operator> ::= “!” | “&” | “^” | “|”
<operands-list> ::= <operand> “,” <operand-list>
<operand> ::= <leaf-event-name> | <logic-gate>

Figure 4.13. Fault Tree Model Backus-Naur Format

With the exception of the “NOT” gate, logic gates can have more than two inputs; but, since the LOTOS equivalent of the gates have only two input gates, the implementation internally breaks gates with more than two inputs into multiple gates. For example, the Fault Tree model can have an “AND” gate with three inputs such as (& event1, event2, event3); the tool will break this gate into two “AND” gates as following (& event1, (& event2, event3)). This facilitates specifying the Fault Tree model using the format accepted by the implementation.

To search the produced integrated specifications for execution sequences, we used an execution-sequence-searching tool offered as part the CADP tool set, called “exhibitor”. The CADP “exhibitor” tool verifies the validity of an input LOTOS specification then outputs errors – if any exist – in the specification such as LOTOS syntax errors or deadlocks. If no errors are found in the specification, the “exhibitor” tool then searches for execution sequences that contains a given execution sequence provided in a “.seq” file to the tool. The “.seq” files contains an execution sequence given in the “sequence” format [35]. The “sequence” format is specified in details in the exhibitor manual. For more information about the “sequence” format, please refer to [35].

The “exhibitor” tool provides multiple configurations options to search for execution sequences within a specification; Figure 3.3 shows a snapshot of the “exhibitor” tool interface. As the figure shows, the “exhibitor” tool utilizes two searching algorithms (breadth-first, or depth-first based search) for finding execution sequences. The tool operates in one of the following modes: returning the shortest path, returning the first path, or returning all the execution paths that contains given execution sequence.

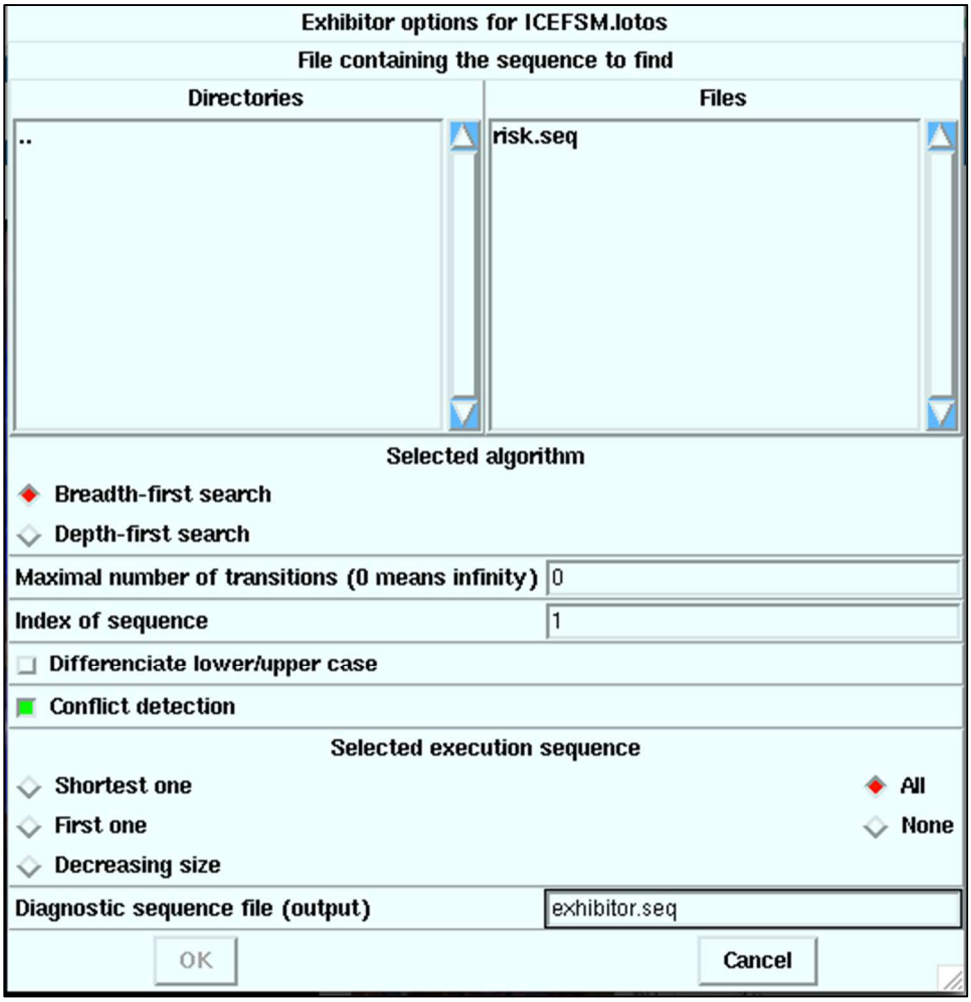

Figure 4.14. Exhibitor Interface

For each of the specifications listed in Table 4.1, we applied the following steps:

- Created a file containing the Fault Tree model in the Backus-Naur form given in Figure 4.13.
- Created a ".seq" file representing any sequence path that ends with the risk event being true.
- Ran the Java implementation of the integration methodology with the LOTOS behavioral model and the file created in the previous step as inputs.
- Input the generated LOTOS specification from the previous step and the ".seq" file to the "exhibitor" tool and documented the output.
- Compared the reported flaws to the functional flaws known to be in the specification.

For each of the specifications used in the experiment we ran the "exhibitor" tool once with the breadth-first option and once with the depth-first option. For every run of the tool, we used the mode that enables finding all execution paths. This configuration enabled finding all the execution sequences that leads to the risk, as each of the execution sequences returned represents a design flaw.

CHAPTER 5. DATA AND ANALYSIS

In this chapter, we present the results of our study. We summarize the data in Table 5.1. The table contains two sub-tables, one that lists the results running "exhibitor" in BFS mode, and the other lists the results running in DFS mode. For each of the specifications used in analysis, the table lists the following:

- The number of known flaws: represents the number of risk-inducing flaws known to be in the specification.
- The number of reported flaws: represents the number of flaws discovered by the proposed approach.

Except for the railroad-crossing specification, design flaws were reported. In the case of the railroad-crossing specification, there were no known design flaws in the design.

For all specifications for which design flaws were reported, the number of flaws was the same using BFS and DFS. Although the faulty execution sequences (e.g. sequences that end with the risk event) were different between BFS and DFS, those sequences were actually different representations of the same design flaw. In other word, the order in which events are visited is different between BFS and DFS, but the resulting execution sequence represents the same design flaw in both cases.

An important property of the results is the number of faults discovered per risk. The results shows that in some cases more than one flaw were reported for a single risk. In other words, more than one execution sequence could end with the risk event occurring. For example, the Insulin pump Fault Tree model contains only one risk (e.g. one Fault Tree) but there were two bugs in the design that induce this risk as shown in the last row of table 5.1.

Table 5.1. Experiment Results

Breadth-First Search (BFS)			
Specification	Risk	Number of Known Flaws	Number of Reported Flaws
Gas Burner	Fire	1	1
Railroad-Crossing	Accident	0	0
Aerospace Launcher	Initialization Failure	3	3
	Fire	3	3
	Launch Failure	5	4
	Preflight Failure	3	2
Insulin Pump	Over/Under Dose	2	2
Depth-First Search (DFS)			
Specification	Risk	Number of Known Flaws	Number of Reported Flaws
Gas Burner	Fire	1	1
Railroad-Crossing	Accident	0	0
Aerospace Launcher	Initialization Failure	3	3
	Fire	3	3
	Launch Failure	5	4
	Preflight Failure	3	2
Insulin Pump	Over/Under Dose	2	2

CHAPTER 6. DISCUSSION

Using the results of our analysis, in this chapter, we present our conclusions, the limitations of the approach, and then we derive practical implications.

6.1. The Approach Ability to Discover Flaws

The results of the empirical study show that, except for two flaws, the approach discovered all the flaws known to exist in the designs. For these two flaws, there were inconsistencies in the input designs; the same event name was used to describe different events in different parts of the design. With these inconsistencies in the design one event can hide a totally different event that shares the same name, causing the tool to fail to identify faulty execution sequences. We discuss possible solutions for this situation in the next chapter. However, if names are used consistently throughout the design (e.g. different events are modeled using different names), the approach discovers all the risk-inducing flaws in the design.

Also, the results show that the risk-inducing flaws discovered were the same with both search algorithms (e.g. DFS and BFS). This means that the algorithm used to search the integrated model for execution sequences does not affect the ability of the approach to discover flaws. Hence, the approach could be used with different search tools without decreasing the ability of discovering flaws; this makes the approach more flexible.

6.2. Approach Applicability

The results show that the approach can discover all design flaws that induces a given risk, given that there are no inconsistencies regarding the way event names are used in the design (e.g. same names are used to model same events, and different events are modeled using different names). This means that if multiple design flaws induce the same risk, the approach is capable of

discovering all these flaws in a single run. Hence, this eliminates the need to test designs more than once in order to discover all the flaws that induces the same risk. This helps reduce the time needed for safety analysis in the design phase.

From the experiment we performed, we learned that the approach could be useful in the following application areas:

- Risk Mitigation: The approach could be used to mitigate associated risks in the design phase. By identifying the associated risks, the approach could be used to decide whether the design addresses those risk; if not the design should be modified to mitigate them.
- Iterative Design: Designers could use the approach to check their designs for risk-inducing flaws. As long as flaws are reported, designers should reiterate over the design fixing those flaws until the design is risk-proof.
- Safety Analysis: The approach could be used to make sure that the design does not expose any safety hazards. This could be done creating Fault Tree models that capture the safety hazards, and running the approach to decide whether the design handles them correctly.

However, the effectiveness of the approach depends on the quality of the Fault Tree model. This means that, if the events used in the Fault Tree model, and the events used in the behavioral model are from different levels of abstraction, the flaws reported by the approach may not be very useful. Thus, the approach proves more effective when the Fault Tree model captures all the risks, and breaks them to basic events on the same level of abstraction as the behavioral design.

6.3. Limitations of The Approach

Although the results of our empirical study were promising, there are two limitations to the proposed approach.

The first limitation with the approach comes from the compatibility of the input models. As mentioned in Chapter 3, we assume that the same events are captured using the same names in both models (e.g. the behavioral model, and the Fault Tree model). However, if the same event has different names between the two models or if two different events have the same name in both models, the approach might report counterfeit flaws. Also, we mentioned earlier that this issue was addressed in [5]; however, the case in which the same name is used to describe different events was not addressed. For example, if the event name "fire" describes the occurrence of a fire in the Fault Tree model but describes an ignition spark in the behavioral model, design flaws will be reported that not necessarily expose the fire risk. This can be currently handled by manually checking the models for these events before running the approach but that will impact the time cost of using the approach adversely.

The second limitation with the approach is that it only targets LOTOS specifications. Gario et al. [5], and Ariss et al. [13] suggested approaches that utilize the technique of integrating CEFSM and State Chart behavioral models respectively with Fault Tree models for testing. But as mentioned in Chapter 2, the modeling languages that these approaches target are not formal languages making these approaches less useful in case of safety critical system. However, the approach could be adapted to other formal specification languages making the approach more usable.

6.4. Practical Implications

Based on the results and findings we presented, we draw their practical implications. From Boehm et al. [19], we learned that discovering flaws early in the software development process could significantly reduce the cost of fixing them and eliminate the chance that these flaws make it to the final product.

The study suggested an approach for discovering functional software flaws in the design stage of software development. Our proposed approach produced promising results, and could cut down the cost of discovering and fixing software flaws. We believe that the suggested approach has many applications and could help mitigating safety related risks in software design.

CHAPTER 7. CONCLUSION

In this study, we presented an automated approach for discovering risk-inducing software design flaws. The approach utilizes a framework that integrates Fault Tree models with LOTOS behavioral models, and uses the result of the integration to search for design flaws. We presented an empirical study to evaluate our approach. Our results show that the approach precisely identifies design flaws within the behavioral model.

In the previous chapter, we discussed limitations with our approach. These limitations could be addressed with further studies that extend our integration framework. For future work, we intend to investigate several extensions to our approach.

First, to address the case in which the same event name is used for different events, we are planning to add a module to our approach to check for such events, and other consistency issues. The module will have a graphical user interface that prompts the user to match events from the behavioral model with events from the Fault Tree model. The module also should perform static verification of the models to ensure compatibility. This will reduce the manual intervention needed to use the approach.

Second, to expand our approach to handle behavior designs created in languages other than LOTOS, we are planning to adapt the approach to other Formal Description Techniques (FDT) such as Estelle and SDL. Other FDTs are very similar in nature to LOTOS, which makes adapting our approach to these languages feasible. This will enhance the usability of our approach.

REFERENCES

[1] Chen, Y., & Probert, R. L. (2003, November). A risk-based regression test selection strategy. In Proceeding of the 14th IEEE International Symposium on Software Reliability Engineering (ISSRE 2003), Fast Abstract (pp. 305-306).

[2] Amland, S. (2000). Risk-based testing:: Risk analysis fundamentals and metrics for software testing including a financial application case study. Journal of Systems and Software, 53(3), 287-295.

[3] Nazier, R., & Bauer, T. (2012, November). Automated risk-based testing by integrating safety analysis information into system behavior models. In Software Reliability Engineering Workshops (ISSREW), 2012 IEEE 23rd International Symposium on (pp. 213-218).

[4] Sánchez, M. A., & Felder, M. A. (2003). A systematic approach to generate test cases based on faults. Proc. ASSE.

[5] Gario, A., & von Mayrhauser Andrews, A. (2014, April). Fail-Safe Testing of Safety-Critical Systems. In Software Engineering Conference (ASWEC), 2014 23rd Australian (pp. 190-199).

[6] Stallbaum, H., Metzger, A., & Pohl, K. (2008, May). An automated technique for risk-based test case generation and prioritization. In Proceedings of the 3rd international workshop on Automation of software test (pp. 67-70).

[7] Kloos, J., Hussain, T., & Eschbach, R. (2011, March). Risk-based testing of safety-critical embedded systems driven by fault tree analysis. In Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on (pp. 26-33).

[8] Marinescu, R. (2001). Detecting design flaws via metrics in object-oriented systems. In Technology of Object-Oriented Languages and Systems, 2001. TOOLS 39. 39th International Conference and Exhibition on (pp. 173-182).

[9] Miceli, T., Sahraoui, H., & Godin, R. (1999, October). A metric based technique for design flaws detection and correction. In Automated Software Engineering, 1999. 14th IEEE International Conference on. (pp. 307-310).

[10] Mao, Y., Sahraoui, H., & Lounis, H. (1998, October). Reusability hypothesis verification using machine learning techniques: a case study. In Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on (pp. 84-93).

[11] Marinescu, R. (2004, September). Detection strategies: Metrics-based rules for detecting design flaws. In Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on (pp. 350-359).

[12] Kim, H., Wong, W. E., Debroy, V., & Bae, D. (2010, November). Bridging the gap between fault trees and UML state machine diagrams for safety analysis. In Software Engineering Conference (APSEC), 2010 17th Asia Pacific (pp. 196-205).

[13] El Ariss, O., Xu, D., & Wong, W. E. (2011). Integrating safety analysis with functional modeling. Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on, 41(4), 610-624.

[15] Bach, J. (1999). Heuristic risk-based testing. Software Testing and Quality Engineering Magazine, 11(9).

[16] Gebizli, C. S., Metin, D., & Sozer, H. (2015, April). Combining model-based and risk-based testing for effective test case generation. In Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on (pp. 1-4).

[17] Souza, E., Gusmão, C., & Venâncio, J. (2010, April). Risk-based testing: A case study. In information technology: new generations (ITNG), 2010 seventh international conference on (pp. 1032-1037).

[19] Boehm, B., & Basili, V. R. (2007). Software defect reduction top 10 list. Software engineering: Barry W. Boehm's lifetime contributions to software development, management, and research, 34(1), 75.

[20] Vesely, W. E., Goldberg, F. F., Roberts, N. H., & Haasl, D. F. (1981). Fault tree handbook (No. NUREG-0492). Nuclear Regulatory Commission Washington DC.

[21] Malhart, B. E. (1995). Software fault tree analysis for a requirements system model. In Systems Engineering of Computer Based Systems, 1995., Proceedings of the 1995 International Symposium and Workshop on (pp. 133-140).

[22] Friedman, M. (1993, January). Automated software fault-tree analysis of Pascal programs. In Reliability and Maintainability Symposium, 1993. Proceedings., Annual (pp. 458-461).

[23] Towhidnejad, M., Wallace, D. R., & Gallo, A. M. (2002, December). Fault tree analysis for software design. In Software Engineering Workshop, 2002. Proceedings. 27th Annual NASA Goddard/IEEE (pp. 24-29).

[24] Ying, R., Hong, L., & Hua-wei, L. (2011, December). Research on technique of software testing based on fault tree analysis. In Computer Science and Network Technology (ICCSNT), 2011 International Conference on (Vol. 3, pp. 1718-1720).

[25] Bolognesi, T., & Brinksma, E. (1987). Introduction to the ISO specification language LOTOS. Computer Networks and ISDN systems, 14(1), 25-59.

[26] Turner, K. J. (Ed.). (1993). Using formal description techniques: an introduction to Estelle, LOTOS and SDL (Vol. 85). Wiley.

[27] ISO: Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique based on the Temporal Ordering of Observational Behaviour, ISO 8807, International Organisation for Standardisation, Geneva, CH, 1989

[28] Kaiser, B., Gramlich, C., & Förster, M. (2007). State/event fault trees—A safety analysis model for software-controlled systems. Reliability Engineering & System Safety, 92(11), 1521-1537.

[29] Kenneth J. Turner and Richard O. Sinnott. DILL: Specifying digital logic
in LOTOS. In Richard L. Tenney, Paul D. Amer, and M. Umit Uyar, editors, Proceedings of Formal Description Techniques VI, pages 71-86.

[30] Garavel, H. (1998). Open/Cæsar: An open software architecture for verification, simulation, and testing. In Tools and Algorithms for the Construction and Analysis of Systems (pp. 68-84).

[31] Gario, A. (2014, November). Fail-Safe Testing of Safety-Critical Systems. PhD. Dissertation, University of Denver.

[32] Gario, A., Andrews, A., & Hagerman, S. (2014, March). Testing of safety-critical systems: An aerospace launch application. In Aerospace Conference, 2014 IEEE (pp. 1-17).

[33] Hagerman, S., Andrews, A., Elakeili, S., & Gario, A. (2015, March). Security testing of an aerospace launch system. In Aerospace Conference, 2015 IEEE (pp. 1-11).

[34] Elakeili S. (2015, March). Fail-Safe Test Generation of Safety Critical Systems. PhD. Disseretation, University of Denver.

[35] Garavel, H., & Mateescu, R. (2004). SEQ. OPEN: a tool for efficient trace-based verification. In Model Checking Software (pp. 151-157).

[36] Wong, W. E., Debroy, V., Surampudi, A., Kim, H., & Siok, M. F. (2010, June). Recent catastrophic accidents: Investigating how software was responsible. In Secure Software Integration and Reliability Improvement (SSIRI), 2010 Fourth International Conference on (pp. 14-22). IEEE.

[37] Clarke, E. M., Grumberg, O., & Peled, D. (1999). Model checking. MIT press.

[38] Mateescu, R., & Garavel, H. (1998, July). XTL: A meta-language and tool for temporal logic model-checking. In Proceedings of the International Workshop on Software Tools for Technology Transfer STTT (Vol. 98, pp. 33-42).

[39] Clarke, E. M., Klieber, W., Nováček, M., & Zuliani, P. (2012). Model checking and the state explosion problem. In Tools for Practical Software Verification (pp. 1-30). Springer Berlin Heidelberg.