INTEGRATED SAFETY, RELIABILITY, AND DIAGNOSTICS OF HIGH ASSURANCE, HIGH CONSEQUENCE SYSTEMS

By

James R. Davis

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

ELECTRICAL ENGINEERING

May, 2000

Nashville, Tennessee

Approved: Date:

_____ _____

_____ _____

_____ _____

_____ _____

_____ _____

In memory of my mother, Priscilla C. Davis

ACKNOWLEDGEMENTS

An undertaking such as this cannot be completed alone, and I would like to acknowledge the many persons who helped me succeed in this task. First and foremost, I would like to thank my family. My father and my sister provided must needed support, understanding, and the occasional kick-in-the-pants I needed to survive this journey. Without them, I could not have started, let alone finished, graduate studies. Special thanks to Melissa for all of the support and motivation she gave. She provided my personal support group that got me through the tough times. Her unwavering support is immensely appreciated.

Secondly, I would like to thank each and every member of my Ph.D. committee for their comments, insight, and direction. My advisor, Dr. Janos Sztipanovits, merits special thanks for the many years of leadership. His insight, determination, dedication, and desire to explore new ideas has been an inspiration. Dr. Sztipanovits would always allow me the freedom to explore new areas that interested me. His approach to graduate students helps to create more than just a dissertation, but to create a researcher. Through the years, he has helped mold me from an enthusiastic, raw student to what I am today.

Lastly, I would like to thank each and every member of the Institute for Software Integrated Systems. Special thanks goes out to Jason Scott, Dr. Gabor Karsai, Dr. Ted Bapty, and Dr. Greg Nordstrom. Thanks for all of the advice, encouragement, and direction over the (many) years I have been a graduate student. I would also like to thank Jim Martinez of Sandia National Laboratories for his insight and direction in the areas of safety and reliability engineering.

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

LIST OF EQUATIONS

LIST OF ALGORITHMS

CHAPTER I

INTRODUCTION

High consequence systems can be defined as those where failures can cause catastrophic results. These results can include loss of life, loss of resources (i.e. money), or even loss of credibility. Having a failure in a high consequence system is unacceptable. At all costs, a high consequence system must be kept *safe*. Some examples of high consequence systems are nuclear power plants, commercial aircraft, chemical plants, automobile safety systems, and traffic control systems. For a particular example, if the air bag system on an automobile doesn't inflate when in a crash (of sufficient force), a safety failure has occurred. The occupants of the automobile may be injured. Safe operation is the primary goal in the design of high consequence systems.

High assurance systems are systems where reliability is of paramount importance. Instead of requiring the system to remain safe at all times, the primary design goal for high assurance systems is that the system always remains *reliable*. A high assurance system must retain critical system functionalities at all times. Failure of a high assurance system to retain certain functionalities can have catastrophic results. An example of a high assurance system is an air traffic control system. The air traffic control system must always maintain the ability to determine the number, position, and vector of an aircraft that is in range. Otherwise, the system has not met a critical reliability requirement. While system reliability is the primary goal in the design of high assurance systems, often safe system operation is also a concern.

Some systems are classified as both high consequence and high assurance systems. These systems exhibit the behavior of both classes of systems. In these systems, tradeoffs must be made between safety and reliability during system design. While in some systems, safety takes priority over reliability, the converse can also be true for other systems. Some of the common features of high assurance, high consequence systems are::

- The systems are *complex and heterogeneous*.
- The systems are *reactive;* they are in continuous interaction with their environment.
- *Dynamics* is an essential characteristic of system behavior.

Safety analysis methods originated in the early ICBM-based weapon systems [1]. This was when engineers first recognized that complex, system-level interactions and cascading effects are the primary sources of safety hazards. The goal of safety analysis is to answer the following question*: Is there any normal or abnormal system input or fault conditions under which the system is able to produce behaviors that are considered to be safety hazards?* Safety analysis focuses on abnormal behaviors that must be avoided or prevented by safe design (inherently safe systems) or active safety control mechanisms. Modern safety analysis techniques are discussed in detail in Chapter II of this dissertation.

System reliability analysis is a different, but strongly related, aspect of system design. System reliability, not component reliability, is the focus of this discussion. Component reliability and non-degrading system reliability, where component failures do not cause the loss of functionality, will not be discussed. The primary question to be

answered by reliability analysis is the following: *What is the probability of losing a required function as a result of component failures?* Reliability analysis, also referred to as dependability analysis, typically uses physical component models characterized by failure rate statistics. The goal of the analysis is to calculate system-level failure rates for selected functionalities and to determine which component faults contribute to the loss of the selected functionalities. Modern reliability analysis techniques are discussed in detail in Chapter II.

Safety and reliability are primary design goals of high consequence, high assurance systems. In addition to designing systems to meet safety and reliability requirements, designers are often faced with the task of improving safety and reliability of a design. This may be required by new design features or by the need to extend a system's deployed lifetime. While safety concerns must be addressed by the system design, there are two fundamental approaches to achieving higher system reliability: increase the reliability of the individual components and sub-assemblies, or develop an *active reliability arrangement* [59]. An active reliability arrangement detects component faults and re-configures the system to ensure continued system operation. Active reliability relies on the ability to detect and correct for individual component faults.

The need for active reliability management introduces the need for fault diagnostics to the design of high consequence, high assurance systems. In order to implement an active reliability management scheme, diagnostic analysis must be elevated into the primary design goals. The question to be answered by fault diagnostics is: *What are the possible causes of a system failure?* Diagnostic analysis can be based on causal network or fault propagation models that are augmented with observation models.

Diagnosability analysis determines if there are an adequate number of sensors in a system to detect, distinguish, and predict faults [9]. *Diagnosability* will not be examined, as many high consequence, high assurance systems do not have sensors incorporated into their design. Modern diagnostic analysis techniques are discussed in Chapter II.

Traditional safety, reliability, and fault diagnostic analysis tools use different modeling approaches and different analysis methods. Since the different analyses strive to answer questions about the same physical system, the different models are not independent. The consistency among the models, and consequently the analyses results, is not guaranteed due to the lack of a formal description of the relationships among the safety, reliability, and fault diagnostic models. This lack of consistency makes the objective evaluation of design tradeoffs impossible.

System-level analysis on high consequence, high assurance systems requires safety, reliability, and diagnostic analyses to be performed on the system models. Currently, separate models are built for each type of analysis. Available technology does not support an integrated modeling and analysis toolset for high consequence, high assurance systems. There is a need for an integrated modeling and analysis toolset in order to guarantee consistency between the different system analyses and to reduce the effort required to construct multiple models of one physical system. This defines the primary goal of my dissertation research:

It is possible to design and construct an integrated modeling and analysis environment to support safety, reliability, and diagnostic analyses of high consequence, high assurance systems. The

environment will be developed to overcome many of the shortcomings associated with independent, loosely coupled safety, reliability, and diagnostic modeling and analysis.

Chapter II of this dissertation reviews the state-of-the-art approaches for performing safety, reliability, and diagnostic analysis. Then Chapter III provides the necessary framework to integrate safety, reliability, and diagnostics into an integrated modeling and analysis environment. Since currently available safety, reliability, and diagnostic analysis tools rely on *modeling* for system information representation, current modeling approaches applicable to safety, reliability, and diagnostic analyses will be discussed. Several analysis techniques will be discussed in both a safety and a reliability context. Individual modeling and analysis techniques are examined in light of how they could be incorporated into a seamless, integrated modeling and analysis package. Promising techniques for allowing an integrated analysis environment to be designed are also discussed. Advantages and disadvantages of each technique are identified and discussed. Finally, Chapter III and Chapter IV present the concepts and methodology for developing an integrated safety, reliability, and diagnostic modeling and analysis environment for high assurance, high consequence systems. Chapter V describes the toolset developed to integrate safety, reliability, and diagnostics. A case study application of the environment to a representative problem is discussed in Chapter VI. Lastly, conclusions related to the research and recommendations for future research and enhancement of the modeling and analysis environment presented are given in Chapter VII.

CHAPTER II

BACKGROUND

This chapter provides an overview of the state-of-the-art safety, reliability, and diagnostic modeling and analysis techniques. It is important to examine the models used by the individual analysis techniques in order to determine what information can be combined into an integrated modeling environment. Without understanding individual modeling and analysis techniques, an integrated environment cannot be created. In addition to the modeling techniques used for representing high assurance, high consequence systems, this chapter will also examine analysis techniques based on the different modeling techniques.

First, modeling techniques used for representing high consequence, high assurance systems are discussed. Next, safety, reliability, and diagnostic analyses are examined. Lastly, previous efforts at providing an integrated modeling and analysis environment for high consequence, high assurance systems will be presented. In all cases the focus will be on how the techniques can be utilized in an integrated environment.

Modeling of High Assurance, High Consequence Systems

Safety, reliability, and diagnostic analyses all require a representation of the system in question. This section examines the different modeling techniques for representing high consequence, high assurance systems. In later sections of this chapter, safety, reliability, and diagnostic techniques will be discussed. These analysis techniques utilize the examined modeling approaches for system representation. The modeling

approaches presented are a representative set of techniques applicable to safety, reliability, and diagnosability analyses.

Finite State Machines

Discrete Event Systems (DES) can be used to model dynamic systems by modeling a finite operational space for the system and how modeled events (from a finite set) effect that operational space. Finite State Machines (FSM) are one example of a DES. FSM are commonly used to describe sequential logic circuits. FSM describe a system by partitioning the system into several "states" of operation. The FSM is always in one of these states. Stimuli, either external or internal, affect the FSM by triggering a change in state. The FSM detects a stimulus and then reacts by moving from one state of operation to another [2].

Mathematically, a FSM can be described as a finite state automaton (X, S, G, f, s_0, Y, g) where:

X is the input event set,
S is the set of system states,
$G(s)$ is a set of feasible events, defined for all $s \in S$ with $G(s) \in X$,
s_0 is the initial state,
Y is the output set,
f is the transition function, $\mathbf{f}: X \times S \rightarrow S$, defined only for $x \in G(s)$ when the state is s, and
g is an output function, $\mathbf{g}: X \times S \rightarrow P(Y)$, defined only for $x \in G(s)$ when the current state of the system is s.

Deterministic FSMs are those that for a given state and a given set of inputs, the output and next state are always the same. Non-deterministic FSMs can be defined as above if

the transition function f is changed to $\mathbf{f}: X \times S \rightarrow P(S)$. Note that in the case of a non-deterministic FSM, multiple transitions could occur for a given input and a given state. That is, the relations f and g would return sets of possible state configurations instead of singular state configurations.

Using FSMs to model system behavior allows the system behavior to be described in terms of discrete, finite operational spaces. Several tools are available to simulate and validate FSM models. Additional tools are available to convert FSM models into executable source code for direct system implementation [2]. Some of the disadvantages to using FSM models include the "state explosion problem" as system behaviors become more complex. For example, assume two independent component FSMs (*A* and *B*) are to be combined into a system level FSM (*S*). Then, $SS(S) = SS(A) \times SS(B)$ where $SS(X)$ is the number of states comprising *X*. This is a combinatorial expansion of the component state spaces. For complex systems, constructing FSM models would require extremely large state spaces.

Statecharts

Harel [3] devised a method for extending FSM modeling to include the concepts of parallelism and hierarchy. This methodology, a graphical representation known as Statecharts, extends FSM modeling by including multiple state types. OR-states, AND-states, and BASIC-states are the three types of states used in Statecharts. OR-states have sub-states (of any type) that are related by an "exclusive-or" relationship. The system can only be in one of the sub-states at any given time. AND-states have orthogonal components (sub-states) that are related by an "and" relationship. The system must be in

all of the sub-states if it is in any of the sub-states. BASIC-states are the leaf states in the hierarchy: they contain no sub-states [3].

Statecharts are formally defined on a finite state automaton ($X, S, G, f_x, s_0, Y, g_x, Fa_x, Ga_x, Fo_x, Go_x$) for each non-leaf state where:

X is the input event set,

S is the set of system states,

$G(s)$ is a set of feasible events, defined for all $s \in S$ with $G(s) \subseteq X$,

s_0 is the initial state configuration,

Y is the output set,

f_x is a transition function, $\boldsymbol{f_x}: X \times S \rightarrow P(S')$, defined only for $x \in G(s)$ when the states are the set s,

g_x is an output function, $\boldsymbol{g_x}: X \times S \rightarrow P(Y)$, defined only for $x \in G(s)$ when the current state(s) of the system are represent by the set $s \in S$,

Fa_x is a transition relation defined for all AND states (x), Fa_x: S' = { P ($\forall$y f_y | state y is an immediate child of state x) },

Ga_x is an output relation defined for all AND states (x), Ga_x: S' = { P ($\forall$y g_y | state y is an immediate child of state x) },

Fo_x is a transition relation defined for all OR states (x), Fo_x: S' = { $\forall$y f_y | state y is an immediate child of state x }, and

Go_x is an output relation defined for all OR states (x), Go_x: S' = { $\forall$y g_y | state y is an immediate child of state x }.

Each AND state has a transition function defined as :

F_x: { f_x | S' $\neq$ S $\vee$ Fa_x }

and an output relation:

G_x: { g_x | (f_x | S' $\neq$ S) $\vee$ Ga_x }.

Each OR state has a transition function defined as :

F_x: { f_x | S' $\neq$ S $\vee$ Fo_x }

and an output relation:

G_x: { g_x | (f_x | S' $\neq$ S) $\vee$ Go_x }.

Since the Statechart model must have a single root state, the overall transition and output relation becomes that of the root state. From the definition, it is easy to see that Statecharts are an extension of FSM modeling. Instead of the system being in one state, it can be in several states at any point in time. The transition function f and the output function g have *sets* of states as arguments, instead of single states. f can also return *sets* of states as the next states for the system, instead of a single state. Statecharts use the concept of hierarchy and parallelism to extend FSM models. The concept of parallelism shows up in the current system *states* being used to determine the next system *states* and output *events*.

Transitions between states are defined by $e[c]/a$. e is a logical statement of triggering events: if e evaluates to *true* the transition can occur. c is a guard condition, another logical statement: it guards the transition from being taken unless c evaluates to *true*. a is an action that occurs when the transition is taken. Actions are usually events that are used to trigger other transitions in the Statechart [4]. This allows interaction between the orthogonal (parallel) states of a Statechart. It should be noted that all of the transition parameters are optional. In the absence of a trigger or a guard, the missing logic statement evaluates to true by definition.

By using hierarchical, parallel states and the described methodology for modeling state transitions, parallel, interacting state machines can be constructed. Statecharts allow models to be composed, unlike standard FSM models. A new component can be integrated into an existing Statechart by adding the new component in parallel with existing components and explicitly modeling the interfaces to the new component [3]. Figure 1 shows an example Statechart.

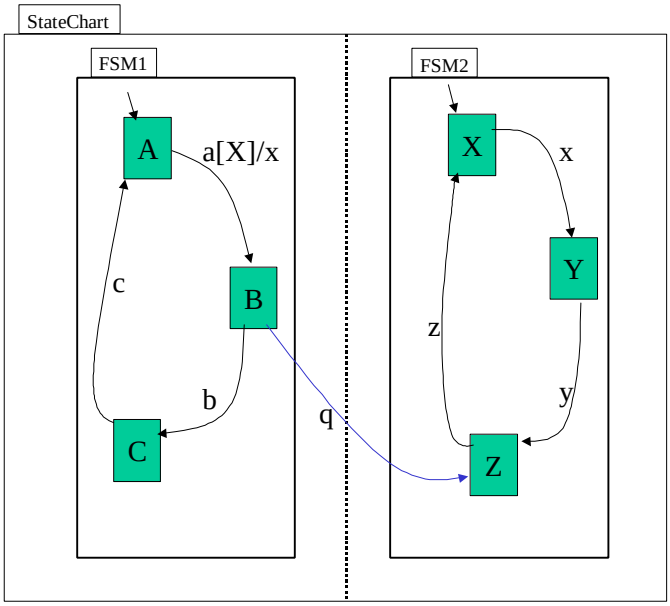

Figure 1: An example Statechart

Markov Modeling

Markov modeling is a formal methodology used to describe systems as discrete states and transitions between the states [5]. These models are used to represent system behavior and how the system behavior changes due to stimuli. Markov models can be graphically or mathematically represented. The transitions between Markov states are marked with probabilities of the transition occurring. Semantically, this means a transition has the marked probability of occurring, if, and only if, the system is in the state the transition leaves from.

Figure 2 shows an example Markov model in a graphical format. Markov models are traditionally represented with a graphical formalism. A stochastic process (**X(t), t** ^{3} **0**) is said to be a Markov process [5] if future development depends only on the present state of the process, or

$$P[X(t) \ ^3 \ x(t) \mid X(t_1) \ ^3 \ x_1,\ldots,X(t_n) \ ^3 \ x_n] = P[X(t) \ ^3 \ x(t) \mid X(t_n) \ ^3 \ x_n] \quad (1)$$

for all $t_1 < t_2 < \ldots < t_n < t$.

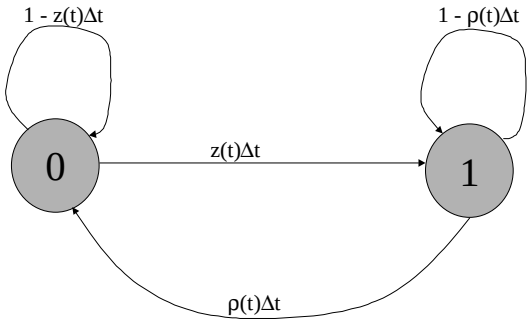

Figure 2: Example Markov chain

Markov processes with discrete state spaces are referred to as Markov chains. Markov chains can be examined to determine if they are independent of time. Markov chains have transition probabilities defined as:

$p_{ij}(t+s) = P[X(t+s) = j \mid X(s) = i]$, $s,t > 0$ where **p_{ij}** is the transition probability between state **i** and state **j**.

Markov chains are an important modeling mechanism for systems that are independent of time. Asynchronous systems can be modeled using Markov chains.

Fault Tree Modeling

Fault trees are graphs that relate a "top event" (an identified hazard) to a combination of (lower-level) faults. The combination is expressed graphically, and, eventually, in the form of Boolean logic [6]. The basic gates used in a graphical fault tree model are the AND, OR, and NOT gates. These gates combine events in the same manner as their corresponding Boolean operations. There is an onto mapping between a fault tree and an algebraic Boolean expression. An example fault tree is shown in Figure 3. This fault tree is from an automotive braking system analysis and is shown as modeled in WinR [7]. WinR is a fault tree analysis tool developed by Sandia National Laboratories. It contains facilities for constructing and analyzing fault trees.

Fault trees can be mathematically represented as:

$$FT(te) = n_0, \tag{2}$$

$$n_0 = \prod_{i=1}^{k+1} n_i \vee n_i \vee c_x \text{ where k = \# of children of } n_0, \tag{3}$$

In general,

$$n_j = \prod_{m=p}^{r} n_m \vee \sum_{m=p}^{r} n_m \vee c_x \tag{4}$$

where n_p through n_r represent the children of n_j,

te is the top event for the fault tree,

$c_x \in$ Component faults, and

n_y is a node in the fault tree.

The fault tree is simple a structure where each node in the tree is either a conjunction of its children, a disjunction of its children, or a node representing a

component failure. The leaf nodes in the tree must all represent component failures for the tree to be analyzable. The "top event" in the tree is the root of the tree. This node represents the failure being modeled and analyzed.

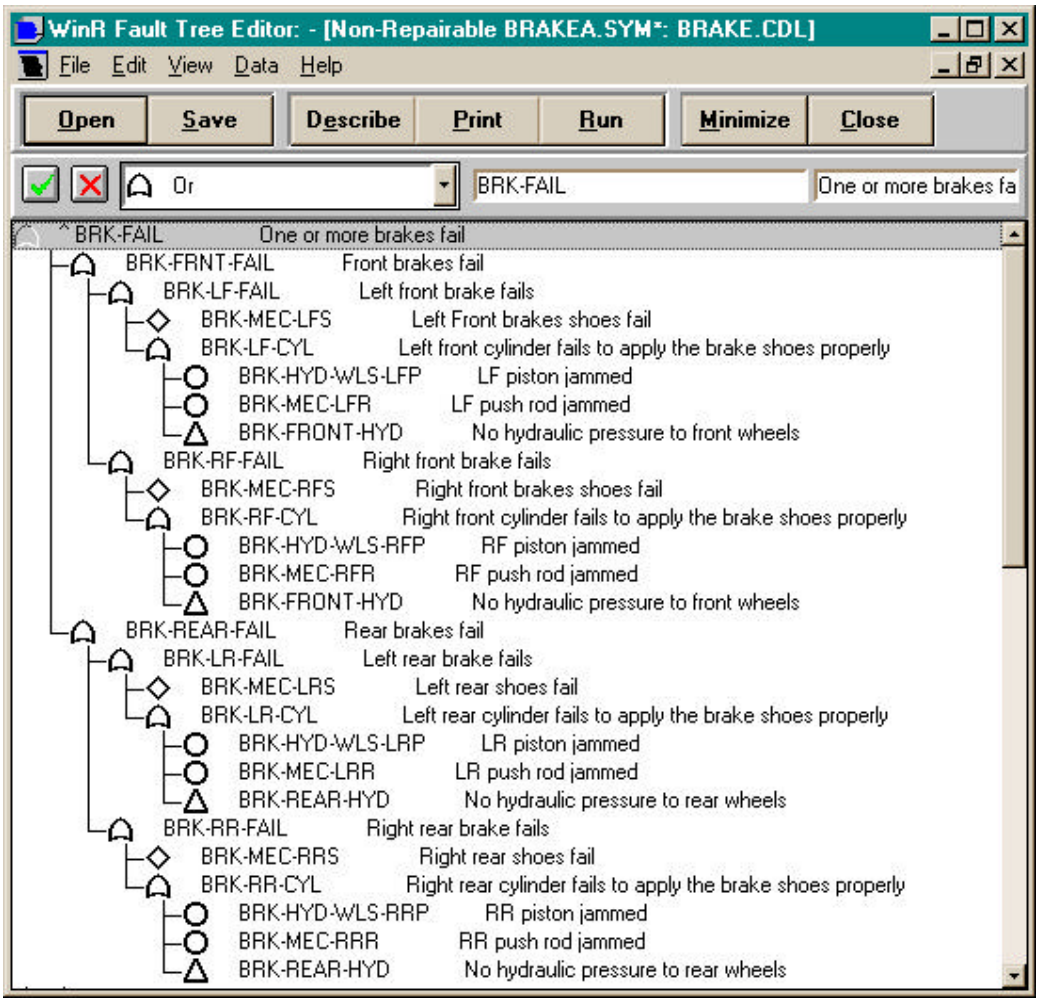

Figure 3: An example fault tree

Mathematically, this notation does not reflect the structure of the fault tree, fault tree models can be simplified to:

$$FT(te) = f(c1, c2, \ldots, cn) \text{ where:} \quad (5)$$

te is the top event for the fault tree,

c1, .., cn are the component faults,

f is a Boolean logic function relating the component faults, and

FT is the mathematical fault tree equivalent.

Fault Propagation Modeling

Fault propagation models are used to show how component faults can affect other components in a system. Fault propagation models are usually represented as labeled, directed graphs, referred to as fault propagation graphs (FPG) [8]. The nodes in the graphs are component faults, discrepancies or anomalies in the system behavior, and sensors. The edges in the graphs reflect the propagation of failures and capture the interactions between failures. Figure 4 illustrates an example FPG. The square nodes in the graph represent the component failure modes. The circles represent the discrepancies, while the ellipses are used to denote sensors in the system. Discrepancies that contain a dot are monitored discrepancies; these discrepancies can be monitored and the occurrence of the discrepancy will be indicated by the "ringing" of an alarm. The ringing of an alarm represents that a sensor has detected a discrepancy.

Mathematical fault propagation models can be represented as:

F the set of possible component faults,

D the discrepancies in the system,

S the set of sensors in the system,

$g_1: F \rightarrow D$, a function defined over *F* and *D*,

$g_2: D \rightarrow D$, a function defined over *D*,

and $g_3: D \rightarrow S$, a function defined over *D* and *S*.

Relation g_1, g_2, and g_3 define the relations represented by the edges of the fault propagation graph. Fault propagation models are generally viewed as graphical models. Traditional analyses operate directly on the graphical models [9]. In practice, the graphical model representation is used more often than the mathematical representation.

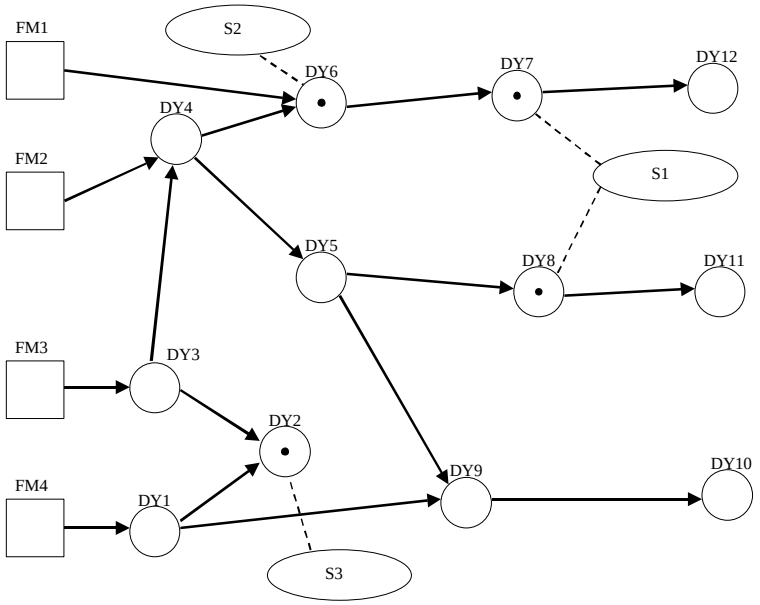

Figure 4: Failure Propagation Graph

Petri Net Modeling

Petri nets are a well-defined visual formalism intended for modeling discrete systems and analyzing their behavior [10]. This graphical notation uses places, transitions, and arcs to form bipartite graphs that represent discrete systems. Places are objects that can contain tokens. Arcs connect places to transitions and transitions to places. When a transition fires, it moves token(s) from the input place(s) to the output place(s). See Figure 5 for an example Petri net. In this example, when the producer

produces a product, a token is placed in the buffer (buff). The consumer waits until a token appears in the buffer and then consumes the token in the buffer.

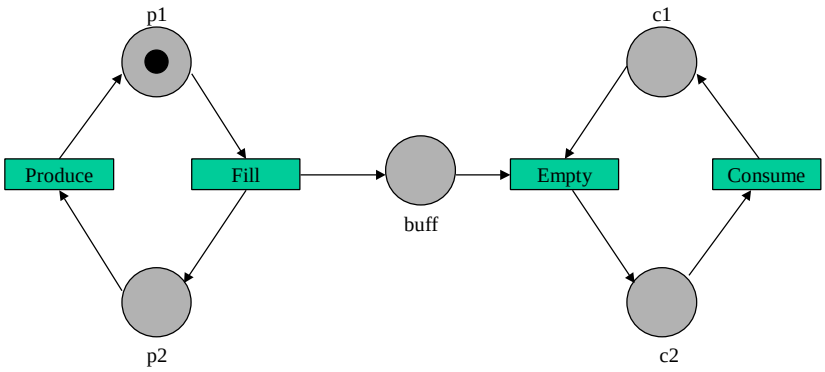

Figure 5: An example Petri Net (Producer/Consumer)

Extensions of Petri nets include hierarchical [11], colored [12], timed [13], and stochastic [14] Petri nets. Following is a formal definition of a colored Petri net. A **Petri Net** is defined as a tuple *CPN = (S, P, T, A, N, C, G, E, IN)* which satisfy the following requirements:

- *S* is a finite set of types, called **color sets**. Each color set is non-empty and finite.
- *P* is a finite set of places.
- *T* is a finite set of transitions.

- *A* is a finite set of arcs such that:

  $P \cap T = P \cap A = T \cap A = \emptyset$.

- *N* is a node function defined from *A* into $P \times T \cup T \times P$.
- *C* is a color function defined from *P* into *S.*
- *G* is a guard function defined from *T* into expressions such that:

  $\forall t \in T: [\text{Type}(G(t)) = \text{Bool} \wedge \text{Type}(\text{Var}(E(t))) \subseteq S]$.

- *E* is an arc expression function. *E* is defined from *A* onto expressions such that:

  $\forall a \in A: [\text{Type}(E(a)) = C(p(a)) \wedge \text{Type}(\text{Var}(E(a))) \subseteq S]$

  where *p(a)* is the place of *N(a).*
- *IN* is an initialization function defined from *P* into expressions such that:

  $\forall p \in P: [\text{Type}(IN(p)) = C(p) \wedge \text{Var}(IN(p)) = \emptyset]$.

Colored Petri nets allow more complex guard functions to be used for representing complex system behavior. Using colored Petri nets allows simpler graphs than standard Petri nets to be used to represent system behavior [12]. Hierarchical Petri nets allow model abstraction. Using hierarchy allows multiple system views from different levels of detail. Hierarchy is used as a visualization technique to simplify representation of complex Petri nets [11].

Petri nets are more than model representation tools. Tools exist for simulating and analyzing Petri net models. Some of the analyses that can be performed on Petri net models include performance analysis, optimization analysis, reachability analysis, and

deadlock analysis [10]. Diagnostic analyses can also be performed. Simulations are often used to perform the above analyses. Simulation is the important analysis feature of FSM systems. Not only does simulation allow examination of the modeled behavior, to ensure it does reflect reality, but it also allows potential problems such as deadlock to be examined. With a simulation engine, it can be determined if a Petri net is prone to deadlock.

Petri nets do have some problems with modeling complex system behavior. Complex systems can require extremely large models. While hierarchical Petri nets can be used to eliminate some of the size problems with representing large models, simulating and analyzing large, complex systems is not always possible. When systems become too large to simulate or analyze, a more scalable method must be found for system representation. Petri nets have proven to be useful for tasks such as proving the correctness of communication protocols. Petri nets do not support quantitative analysis of properties such as reliability. Extensions of Petri nets exist that allow quantitative analysis results. Real data types are necessary to represent the component fault probabilities needed for reliability analysis.

It is important to note that Petri nets can model any system that can be modeled with finite state machines. In fact, a transformation for mapping Petri nets to finite state machines, and vice-versa, exists [15]. By converting Petri net models to FSM models, FSM analysis tools and simulations can be used for model analysis.

Safety Modeling and Analysis

Engineers recognized that systems could become so complex that the related safety issues had to be addressed in a different way during early ICBM-based weapon

systems development [1]. Safety problems emerged, not necessarily because of single upsets, rather because of complex component interactions and cascading effects throughout the system. Another issue was that the causes of accidents could be traced to deficiencies in design, operations, and management. The traditional (a.k.a. "fly-fix-fly") approach was clearly insufficient. Instead, for the first time, a "system-level" solution had to be sought that would prevent accidents from happening. The field of System Safety Engineering (SSE) was born [1].

System safety involves a system-wide perspective, and uses a "holistic" approach. Safety is considered as an "emergent" property of a system, and not directly a property of the system components. Further characteristics of system safety include:

- system safety emphasizes building safety into a system (rather than adding it to the system)
- system safety deals with systems as a whole rather than with components/subsystems
- system safety focuses on hazards that may lead to accidents (instead of failures)
- system safety emphasizes analysis (instead of past experience or standards)
- system safety emphasizes qualitative (rather than quantitative) approaches, although quantitative approaches can provide valuable system safety information, qualitative data regarding which faults may cause a system failure are of greater interest
- system safety recognizes the importance of tradeoffs in system design

- system safety is more than mere system engineering.

Systems must be analyzed for both *quantitative* and *qualitative* safety information. *Qualitative* information is necessary to determine what system modifications will be required to mitigate safety concerns. With qualitative information, system safety engineers will be able to determine which components they need to focus on improving. *Quantitative* data provides the actual probability of a system safety failure occurring. This data is necessary to ensure overall system safety meets specified guidelines [1].

Petri Nets Analysis

Given a Petri net model, specific markings of the model may be considered "unsafe". Whenever the system modeled by the Petri net enters an operating region specified by the "unsafe" markings, the physical system will exhibit safety critical behavior. Based on the previous Petri net definition, all reachable markings from a specific marking can be defined as the set of all markings that can be realized from the original marking in a finite set of steps [16]. By determining the set of reachable markings from the initial system marking and the mapping of "unsafe" places to markings, it is easy to determine if any of the "unsafe" markings are in the reachable set. System safety cannot be guaranteed unless the intersection of the reachable set and the "unsafe" set of markings is empty.

Fault Tree Analysis

Fault Tree Analysis (FTA) is perhaps the single most-used technique for analyzing causes of hazards [1]. Fault trees are graphs that relate a "top event" (an identified hazard) to a combination of (lower-level) faults. The combination is expressed graphically, and, eventually, in the form of Boolean logic [6]. An example fault tree is shown in Figure 3. This fault tree is from an automotive braking system analysis and is shown as modeled in WinR [7]. FTA is performed as follows:

- **Fault tree construction**. Here, the system analyst must construct the actual fault tree corresponding to the top event of interest. Unfortunately, fault tree construction is a manual task.
- **Qualitative analysis**. In this step, the analyst determines:
  - the minimal set of primary events that leads to the top event, and
  - the logical combination of those events.

  The result of this analysis is a "minimal cut set", that satisfies the first requirement of a minimal set [17].
- **Quantitative analysis**. In this step, the probability of the occurrence of the top event is calculated based on the cut sets and the probabilities of the primary events occurring.

FTA is primarily used as a safety analysis tool by industry. By constructing fault trees for the system being designed, the overall system safety can be quantified. If the primary events have known failure probabilities, it is a simple mathematical process to calculate the probability of a top event occurring, given the corresponding fault tree.

Since the fault tree can be described as the Boolean function *FT(te) = f(c1, c2, …, cn)* and the failure rates for the component faults *(c1,…,cn)* are known, the overall probability of the top event occurring can be calculated. FTA also allows for qualitative safety analysis [4]. After constructing the fault tree, a safety engineer examines the fault tree to observe which combinations of primary events can lead to the top event. The ability to perform qualitative analysis allows system redesign to eliminate, minimize, or mitigate safety concerns [18].

While FTA is a widely used safety method, it has some limitations. For example: fault trees can be constructed only after the system has been designed, they show cause and effect relationships but little more, and they have problems modeling time (dynamic behavior) and state-dependent behavior. It is important to remember that fault trees only model success and failure. FTA does not deal with cases where system safety is merely degraded and does not cause a safety violation. FTA packages are available as commercial-off-the-shelf (COTS) products. One such analysis package is Sandia National Laboratories WinR software [7].

Symbolic Model Verifier

Symbolic Model Verifier (SMV) is a tool designed at Carnegie Mellon University for verifying finite state systems against their specifications. By explicitly stating safety constraints as part of the system specifications, SMV can be used to verify that the system meets the safety constraints. McMillan [19] describes in detail the SMV system. SMV uses a temporal logic called Computation Tree Logic (CTL) to represent system requirements. Since SMV was designed to verify finite state machine models, only finite data types are available for use in SMV. SMV consists of a specification language, the

CTL requirement specification language, and the Symbolic Model Verifier tool [20]. SMV is able to support behavioral modeling through its custom specification language. Both synchronous and asynchronous state machines can be modeled using SMV [19].

SMV does contain tools useful in verification of system specifications. A technique known as Symbolic Model Checking [21] is used to verify the system specification against modeled system requirements. These requirements can be safety, reliability, or diagnostic requirements. CTL is used to specify these high-level system requirements [20] in SMV. Since SMV only supports finite data types (i.e. no real numbers), it cannot be used to produce quantitative results. Only cases where the system is safe (i.e. where the chance of failure is zero) can SMV be used to perform safety analysis.

Software Cost Reduction

Software Cost Reduction (SCR) is a formal technique developed by Heitmeyer [22] for the Naval Research Laboratories. SCR grew out of the need to document the requirements for the Operation Flight Program for the U.S. Navy's A-7 aircraft. SCR is a language and a set of tools to support specification, verification, and validation of system requirements. As with SMV, specifying system safety constraints as part of the system requirements allows SCR to be used to verify system safety. Some of the tools provided in SCR include a model simulator, a consistency checker [23], and a verifier. The simulator is used to symbolically execute the system specifications. Well-formedness (syntax and type checking) of the specification can be verified with the consistency checker. The SCR verifier is used to analyze the specification with regard to specific critical system properties.

SCR uses a table-based specification language to represent system requirements [24]. This table-based language models systems using a finite state machine approach. This approach has become very popular among software engineers for documenting system requirements [23].

Advantages to using SCR as an integrated modeling technique include the existence of a consistency checker and a specification verifier. With these tools, the system specifications can be validated and verified before further analyses are undertaken. The SCR model checker uses state exploration for verifying system specification properties [25]. Unfortunately, SCR only supports data types such as Booleans, bytes, and integers. Without support for real numbers, deriving quantitative safety, reliability, or diagnostic values is not possible. While SCR can be used for determining qualitative information about a system, it does not support quantitative analysis. Another disadvantage to using SCR is the unknown scalability of the table-based specifications. Composing specifications from existing specifications requires the modification of the component specifications [23].

Common Aspects of Safety Techniques

System safety engineering uses a global, system-wide approach to safety engineering. It is based on the recognition that in complex systems, very complex sequences of events can lead to safety problems. The central activity of safety analysis is twofold: analysis and synthesis. The variety of systems being analyzed for safety allows for several different analysis techniques. All methods focus on verifying the system remains safe at all times.

Reliability Modeling and Analysis

Reliability modeling and analysis focuses on ensuring a system does not lose essential functionalities when operating. Safety and reliability can be complementary, or they can be opposing. A safe system is not always reliable and a reliable system may not be safe. In system reliability analysis, a reliability engineer's goals involve eliminating, or reducing, the probability of system failures that result in the loss of function. It is important to note that reliability will not be discussed as it pertains to individual components, or as it pertains to non-loss of function failures (i.e. performance degradation).

Reliability analyses can be either inductive or deductive. Both types of reliability analysis have as a goal determining how system level failures affect system performance and how they can be reduced or eliminated. Some reliability analysis techniques allow for both qualitative and quantitative analyses. These analyses enable both the probability of system failures to be calculated and the combinations of component failures that trigger system level failures to be identified. Different types of reliability modeling and analysis tools will be examined in the following sections. Both deductive and inductive analysis tools will be examined Analysis of these reliability tools will focus on how they could integrate into a combined safety, reliability, and diagnostic system.

Fault Tree Analysis

As previously stated, Fault Tree Analysis (FTA) is perhaps the single most-used technique for analyzing causes of hazards [6]. Traditionally, FTA is used as a safety analysis tool [1]. However, FTA can be, and is, applied as a reliability analysis tool. The major differences between safety FTA and reliability FTA are the definition of the "top

event" and the importance of quantitative analysis. By defining a reliability concern as a top event, the FTA process can be used to determine the probability of the reliability failure occurring [26]. Quantitative analysis is necessary to provide actual probabilities of a system failure occurring.

While FTA is a widely used method, it has some limitations. For example: fault trees can be constructed only after the system has been designed, they show cause and effect relationships, but little more, they have problems modeling time (dynamic behavior) and state-dependent behavior. For every top-level system reliability concern, or reliability "top event", a new FTA must be performed. There is no method for re-using fault tree models in other FTA. Combining this realization with the fact that fault trees are manually constructed, renders FTA an extremely time-consuming analysis technique.

Markov Analysis

Markov analysis provides for quantitative analysis of Markov models. Markov analysis relies on the underlying mathematical properties of the Markov models. Assuming a starting state for the system is known, the probabilities of entering any particular state can be calculated from the models. While many different Markov models for system reliability have been identified [27], they will not be discussed in detail here. It is sufficient to note Markov models can be constructed, and solved for reliability values, for a given system. Markov models can be utilized to determine the finite probability of entering a particular system state (i.e. a state representing a reliability failure) [27].

Performing system reliability analysis using Markov models does introduce two key problems. First, fault interactions must be incorporated into the system model. System models cannot be composed from component models; instead, a complete system model, which explicitly captures component interactions, must be constructed. Secondly, individual component failures that lead to a system level failure are not identified by the analysis. The system analyst must rely on the system models to discern which component failures induce the system level failure.

Failure Modes and Effects Analysis

Failure Modes and Effects Analysis (FMEA) is a procedure where every possible failure mode of the system or its components is analyzed as to its system-level effects and severity [26]. The purpose of FMEA is to identify which areas of a design need improvement in order to improve system reliability. FMEA uses design information to determine where to focus resources in order to decrease the chance of a system reliability failure. Failure rates for the components can be used to compute system level failure probabilities. This information can be captured by fault propagation graphs (FPG). Once a FPG has been constructed for a system, all component failures and the discrepancies they cause are modeled. By performing a reachability analysis on the graph, all discrepancies that can be triggered by a component failure are discovered. Documenting the results of the reachability analysis will produce a set of worksheets that describe component failures and their system-level effects.

The FMEA worksheets follow a fairly standardized format; the information contained in the worksheets is easily transferable to other engineers and designers. FMEA has the look of a checklist approach where the main purpose it to provide

documentation of reliability analysis. However, if used early in the design cycle, FMEA can be used to impact the design and improve upon system reliability. FMEA is a preventative design method based on [26]. FMEA is used to identify sources of failures and to provide prioritization of mitigation goals. Commonly, a FMEA is performed by analyzing failed or degraded components and then identifying the causes of the failures and then the effects of the failures. FMEA is an inductive analysis technique.

McKinney [28] discusses some major shortcomings of traditional FMEA: lack of pertinence to system operation, lack of support for the system, time consuming technique, narrowness of scope, and the "box-checking" nature of the technique. According to [29], a major problem with FMEA is using the technique as a perfunctory "checklist". This "checklist" is then used to satisfy contractual agreements with customers. Incorporating FMEA early in the design cycle would more effectively impact system design, and thus, system reliability [30].

New advances include using behavioral models and simulation for performing FMEA [30]. Eubanks [30] uses a behavioral system model to represent the state changes of design variables during normal system operation. Qualitative effects of failures on the modeled state variables are discovered through simulation. Disadvantages to using FMEA for reliability analysis include scalability, lack of model reuse, time consumption, and the "checklist" property of FMEA.

Petri Net Analysis

Petri net analysis requires Petri net system models, which were discussed previously. Reliability analysis with Petri nets is often performed using a special class of Petri nets called *stochastic Petri nets* [14]. Stochastic Petri nets appear similar to the

standard Petri net models. However, stochastic Petri nets (SPN) also have probability information assigned to the modeled transitions. Instead of transitions always firing when enabled, the transition has a probability of firing when enabled. General Petri nets can be viewed as a simplified version of SPN where all of the transition probabilities are one. SPN are based on Markov chains and Markov reward models [31].

Balakrishnan [31] has introduced a technique for performing reliability analysis using SPN. Included in this SPN must be transitions that represent the different failure modes for the system. The probabilities of reaching unreliable places can be determined through simulation of the SPN. Additional information that can be determined from this analysis includes a graph showing how system unreliability relates to use-time of the system.

One disadvantage to using this reliability analysis method is the scalability of the system. This system scales according to the underlying state space of the system being modeled. The method has been used to determine the reliability of a system consisting of 300,000 states [31].

Common Aspects of Reliability Techniques

System reliability engineering takes a global, system-wide approach to reliability engineering. It is based on the recognition that in complex systems, very complex sequences of events lead to system reliability dilemmas. The central activity in system reliability is twofold: analysis and synthesis. A set of techniques for hazard analysis is commonly used but there seems to be no single, unique, and "best" method. This can probably be attributed to the variety of high assurance systems that exist. There are, however, common traits shared by many approaches. All reliability analyses focus on

providing information as to which component faults are necessary and sufficient to trigger a reliability failure. System reliability lives in a type of symbiotic relationship with other engineering activities (e.g. safety analysis, requirements analysis, design, diagnostics, implementation, deployment, operation, maintenance, and decommissioning).

Diagnostic Modeling and Analysis

One method of modeling a diagnostic system is as an input-to-output system. The diagnostic system receives inputs from the system to be diagnosed, and if there is an error in the inputs, provides outputs to the system or user. These outputs are indications of the possible presence of an error in the diagnosed system's operation. The goal of a diagnostic system is to alert the system operator of a possible failure in the system and the of possible causes of the failure.

Structural diagnostics is the simplest diagnostic representation method available. The information needed to perform structural diagnostics is the connectivity of the system being examined. From this information, the diagnostic system can help isolate the cause of a problem [32] [33].

Another method of diagnosis uses a behavioral model of the system. By basing the diagnosis of any errors on system behavior, and not just on system interconnectivity, a more concise diagnostic solution can be achieved. This is realized as a smaller set of possible paths the error could have resulted from. By qualitatively modeling the system, error causes identified by a structural model can be eliminated if they do not actually affect the component of the system that has experienced an error [33][34].

Several state-of-the-art diagnostic techniques are described below. Each is evaluated as to how it can be incorporated into an integrated safety, reliability, and diagnostic analysis environment.

Fault Detection Isolation and Recovery

The Fault Detection, Isolation and Recovery (FDIR) methodology was developed for performing diagnostic and diagnosability system analyses [35]. Vanderbilt and Boeing [9] have developed an FDIR toolset called DTOOL. A fault analysis results in various documents that summarize results and provide evaluations of system faults and their effects. These serve as input to the design process to enhance the reliability properties of the system. There are various methods for designing for reliability. In the order of "acceptance" they are listed below [35].

- If possible, use 'Hazard Elimination'. Hazard elimination means that the system is intrinsically reliable, i.e. is incapable under normal or abnormal behavior of causing accidents.
- If that is not possible, use 'Hazard Reduction'. Hazard reduction means that the occurrence of a hazard is prevented or (at least) minimized.
- If that is not possible, use 'Hazard Control'. Hazard control means that the hazard's effects are mitigated once it has occurred, and thus rendered harmless.
- If that is not possible, use 'Damage Reduction'. Damage reduction means that hazards are accepted, and expected to cause damage, but the damage caused by them is contained, and/or reduced to an acceptable level.

For high consequence systems, only the first two methods are acceptable. Hazard control and damage reduction are unacceptable alternatives.

The FDIR toolset was designed as a tool for diagnostic and diagnosability analyses of complex systems. However, FDIR also shows promise as a reliability analysis tool. The fault propagation graph models that FDIR uses allow quantitative information regarding the *timing* of fault propagation in a system [8]. This information can be used to determine the delays possible between a system fault and a detectable error occurring. This timing information can be invaluable to controllers of systems such as nuclear power plants or chemical processing plants. The fault propagation graphs allow for the calculation of the probability of a system failure occurring.

Furthermore, FDIR captures qualitative system reliability information. While this information is not directly available from the analysis tools, it is contained explicitly in the fault propagation graph models. This may be a disadvantage, as the system engineer must develop the fault propagation graph models manually. In a manner similar to fault tree analysis, the reliability engineer must develop complex system reliability models based on knowledge of how the system may fail and which component failures may trigger each system level failure.

FDIR modeling uses a modified fault propagation graph for capturing system information. These graphs model the failure modes of system components, discrepancies, or anomalies, of the system behavior, alarms associated with discrepancies, system sensors, and the propagation between failures [35]. These graphs model not only how system failures can affect system behavior, but also the interactions between system failure modes. Each propagation between failure modes has a minimum

and maximum time associated which represents the time required for the source failure to propagate [8].

The three metrics determined by FDIR are: detectability, predictability, and distinguishability. Detectability gives the longest time required to detect a failure. Predictability represents the shortest time between a forewarning and the occurrence of a failure. Distinguishability describes the size of the ambiguity set given a time limit. An ambiguity set is the set of all suspected component failure modes. Algorithms are given in [35] for each of the metrics to be computed.

One of the disadvantages to using FDIR for diagnostic analysis is the complexity of the models for a high consequence, high assurance system. Determining all possible component failures, how they interact, and how they affect system performance is not a simple process. One feature of FDIR that limits the size of the models constructed is that FDIR does not require full behavioral models of the system being examined.

Fault Tree Analysis

Fault Tree Analysis (FTA) is a deductive safety analysis technique described in detail previously. FTA builds a graph of "top events" or system level failures that represent how component level failures can initiate the system level failure. From these graphs, it is easy to determine the qualitative diagnostics of the system. Given a failure, the root cause failures are explicitly modeled. Standard FTA tools can be used to perform quantitative analysis on the models.

Since many different system level failures might exist, multiple fault tree models are usually required. Building these models is inherently a manual process; performing diagnostic analysis with FTA would require an unusual amount of effort. A tool would

need to be developed to use multiple fault trees for real-time system diagnostics. Whenever system level failures were detected, the multiple fault tree models could be analyzed to determine which component faults are most likely to have caused the high level failures.

Petri Nets

Petri nets are a behavioral modeling technique described in detail previously. B-W Analysis is a technique designed by Anglano and Portinale [36] that allow the use of modified Petri nets for system diagnostics. With B-W Analysis, two types of tokens are required: normal tokens and *inhibitor* tokens. Inhibitor tokens are used to show the falsity of the condition associated to a marked place. In other words, while normal tokens are used to show a condition is true, inhibitor tokens are used to show a condition is false. This allows the discovery of inconsistencies of a given marking in a Petri net [36].

B-W Analysis starts with a given marking for a Petri net. Using a well-defined set of backward-reachability rules, the algorithm searches for markings that could immediately proceed the given marking. This backward reachability algorithm continues to construct a *backward reachability graph* for the given marking. This graph represents the trajectories through the design space the system could have taken to arrive at the given marking.

Anglano and Portinale then provide a mapping between a diagnostic problem and a B-W Analysis. From their mapping, it can be shown the B-W Analysis can be used for performing diagnostics [36]. The most significant problem with using B-W Analysis for

diagnostic analysis stems from modeling the system as a Petri net. Petri nets for large complex systems can become quite cumbersome.

Common Aspects of Diagnostic Techniques

Each diagnostic technique examined requires a system representation that incorporates both nominal and failure behavior. While only a small set of diagnostic techniques have been discussed, these techniques are the most applicable to discrete systems described with behavior and structure models. The point of these techniques is to identify the possible causes of fault behavior. The transitions from nominal to failure modes of operation are triggered by identified component or subassembly failures. Diagnostic analyses use the models to determine which failures could trigger system "alarms", or failures. This information is then relayed to the system user for further analysis and action.

Integrated Modeling and Analysis Approaches

As previous sections have illustrated, no current reliability, safety, or diagnostic modeling and analysis technique can sufficiently handle performing safety, reliability, and diagnostic tasks. Since safety and reliability are closely related, there are efforts to integrate safety and reliability into one modeling/analysis toolset. A recent effort will be examined to determine if it solves the safety and reliability analysis problems sufficiently. The possibilities of extending this technique to include diagnostic analysis will be pursued. Advantages and disadvantages of basing an integrated safety, reliability, and diagnostic analysis environment on this technique will be discussed.

Siemens

Siemens Corporation of Germany has performed research into integrating safety and reliability into a single modeling and analysis environment [37]. The integrated approach taken by Liggesmeyer and Rothfeld involves using a behavioral model to represent the system being considered. The language CSL (Control Specification Language) is used to represent the system behavior as a deterministic state machine. This language allows for the specification of systems as a FSM in a textual language. CSL does not support hierarchical or parallel finite state machines [3].

Safety and functional requirements are modeled using temporal logic, again specified textually. The following temporal operators are allowed in the Liggesmeyer specification language:

- *X f* is true in the present if and only if *f* will hold in the next state of the system.
- *F f* is true if and only if *f* will hold for some state in the future.
- *G f* is true if and only if *f* will hold at some moment if the future.
- *EF f* is true if and only if *f* can be true at some moment in the future.

Additionally, standard Boolean operators can be used in specifying complex safety or functional requirements. Safety specifications can also be composed of mathematical formulas involving variables in the CSL system behavior model. These specifications may limit the values a variable can take, in order to represent a region of operation that is inherently unsafe. By combining the CSL specifications with the temporal logic specifications, timing and physical safety requirements can be formulated [37].

Then, failure models are added to the system specification. The failure models are used to represent both explicit and implicit component failures. For system safety and reliability analysis, the behavior of the system under fault conditions is necessary information.

The behavioral specification, safety requirements, and functional requirements are then converted into a symbolic representation. OBDDs [38] are used to form a canonical representation of the system and the requirement specifications. By using OBDDs a process known as symbolic model checking (SMC) can be used to certify the system meets the safety and functional requirements. Reliability analysis can be undertaken using similar techniques to provide detailed information as to what component faults can result in a reliability failure.

A component cause effect graph (CCEG) is derived from the system specification. A CCEG is an extension of a traditional fault tree. A tool written specifically for analyzing CCEGs has been developed by Siemens. The CCEG analysis tool essentially provides the same information about the system as a traditional fault tree analysis tool [37].

The advantages of the Siemens's method are that (1) both safety and reliability analysis can be performed from an integrated model, (2) the ability to perform quantitative analyses, and (3) the ability to perform qualitative analyses. Disadvantages of using this method are problems in representing complex systems (i.e. no parallelism or hierarchy), scalability, non-traditional safety/reliability analysis tools, and the lack of system validation tools. Additionally, there is no evidence supporting the capability of expanding of the technique for incorporation of diagnostic modeling and analysis.

CHAPTER III

INTEGRATED MODELING

This chapter will focus on the concepts required to provide an integrated modeling environment for high consequence, high assurance systems. While the type of analyses to be performed on the models always affects the modeling paradigm to some degree [39], the specific analysis techniques to be used will not be discussed here. Analysis techniques will be described in detail in Chapter IV. All of the safety, reliability, and diagnostic analysis techniques surveyed utilize some type of model to capture system information. Existing system modeling methodologies will be utilized in the integrated modeling system. Only modeling techniques that can be extended and combined to capture all system information necessary to perform safety, reliability, and diagnostic analyses will be employed.

Multi-Domain Modeling

High consequence, high assurance system design and development is a complex process that often incorporates diverse, often conflicting requirements, new technologies, and involves many diverse disciplines. System engineers must identify objectives and requirements and formulate metrics that can be used by the design teams to assess the viability of the concepts in satisfying the diverse design and development objectives.

The *model integrated computing* [39] approach (see Appendix A) has the ability to incorporate strengths from various modeling and analytical techniques and employs methodology fragments in a hybrid structure to solve complex design problems. In the

specific problem domain of safety, reliability, and diagnostics, the current technologies being applied in a non-integrated fashion cannot solve the complex predictive problems that will enable a designer to certify a design solution. System certification is crucial in the design of high consequence, high assurance systems. The approach taken with *model integrated computing* is to take the strengths of a number of analytical and modeling techniques and develop an integrated approach that surpasses current approaches and also provides a venue for inclusion of new technologies.

Comparison of Safety, Reliability, and Diagnostic Modeling

While diagnostics analysis requires unique information and techniques, safety and reliability analyses are often grouped together in industry. Safety and reliability rely on much the same information and some of the same analysis techniques are applicable across domains [1] [27]. One of the primary differences between safety and reliability analysis deals with the definition of a system failure. Safety system failures often result in the loss of human life or the loss of large sums of resources [1]. Reliability failures can simply be the inability of the system to perform some required function [27]. Diagnostics is performed by trying to determine why the system exhibited observed behaviors. Diagnostics does not involve determining if something can happen or the probability of something happening. Instead, observations about the system are made and the diagnostics is performed to determine why the system behaved as it did. While the loss of a required function can be catastrophic, it does not have to be. In order to capture system information necessary for detailed analysis, both behavioral and physical system models must be constructed and interrelated.

Imagine a nuclear power generation plant as an example. The system is designed to generate electricity. Any failures that result in the inability of the plant to generate electricity, when it is expected to be generating, is a reliability failure. Any failure that causes the system to potentially cause damage to the surrounding area, or people residing in the surrounding area, is a safety hazard. If a component failure results in forcing the reactor to be "scrammed", but no harmful radioactive materials escape the containment vessels, a reliability failure has occurred. Another component failure could result in radioactive gas or water escaping the nuclear plant. If this failure did not cause the reactor to shut down, it could still generate electricity. This failure would definitely be classed as a safety failure. Lastly, if an accident similar to Chernobyl occurred, the reactor core would melt-down and be exposed. This type of failure would be a safety and a reliability failure as both harmful material was released to the atmosphere and the reactor could no longer produce electricity.

Another scenario might involve modern aircraft. An engine failure on a four-engine jet airliner would definitely be a reliability failure. However, the ability to safely fly on only three engines would keep a single engine failure from being a safety hazard. If an engine failure occurred on a single engine jet, the engine failure would also be classed as a reliability failure. However, since there is only a single engine, in this case the failure would also be a safety failure.

The two scenarios presented above are only illustrations of the fact that safety and reliability failures could be one and the same. In both situations, diagnostics could be performed to determine the possible causes of the failures. Different situations require

failures to be classified as a safety failure, a reliability failure, or both. Safety, reliability, and diagnostics are not functions of each other, instead, they are both defined by

- the system components,
- how the components are assembled,
- how the components behave,
- how component failures affect system behavior, and
- how the components interact.

Safety and reliability are both *emergent* properties – they cannot be directly observed. It is important to note that while safety and reliability concerns can be identical, they are not required to be. Many times safety and reliability are different concerns.

Integrating safety, diagnostics, and reliability addresses both complex design modeling and coupled model analysis. To accomplish these objectives, formal languages [1] representative of the problem and solution domains are incorporated to specify all functions and relationships for the specific domains (e.g. reliability, safety, diagnostics).

The objective of reliability modeling is to represent the major functions of the design in terms of expected and desired sub-system and component behaviors. This can be represented in many modeling techniques, but analysis requires a strong mathematical background. Assumptions affect the accuracy of the mathematical models and their evaluation. A successful design requires successful operation of all components modeled. Single components represent some operations, while other operations have two or more components - any of which can perform the needed operation. These functional relationships lead to a mathematical expression relating reliability failure probability to component failure probabilities.

Safety modeling and analysis must address external and internal events which, when introduced to a system, can lead to unsafe operation or conditions. Safe design is directed toward minimizing non-engineered or poorly engineered hazard controls and the elimination of safety critical design behaviors. Safety modeling is an extension of reliability modeling and includes an assessment of how frequently an excursion from the nominal design behavior results in a hazard. Safety analysis is extended in a more formal manner to include consideration of event sequences, which transform the hazard into an accident [27].

Diagnostic modeling and analysis focuses on how the system could present certain behaviors. Diagnostic analysis determines *why* some observed system behavior occurred. Modeling requirements often involve representing system structure, component failures, and failure propagation. This information must be linked to system behavior information in order to perform diagnostic analysis. The system behavior information may simply be system fault modes or sensor readings that reflect an actual problem.

Integrating safety, reliability, and diagnostic approaches under the framework of the MultiGraph Architecture (MGA) requires safety, reliability, and diagnostic domain experts to possess and maintain in-depth knowledge of individual sub-systems and components used in the target system that affect the solution domain [39]. It is the responsibility of the domain experts to formalize the design using a common formal language. The use of a formal language suitable for integrated modeling and analysis allows the synthesis of the multi-domain problem structure from singular aspect domain model structures. It is the singular aspect domain structures that allow domain experts to

perform specific analyses in the area of concern. This methodology allows both complex and coupled model analysis issues to be addressed.

System Representation

Systems can sometimes be better described by "what" they do rather than "what" they are composed of. Behavioral modeling is used to describe how systems function and how the different system components interact. Behavioral modeling has been applied to many different domains, including physical systems, software systems, and hybrid systems. Many different methods for modeling system behavior have been developed. Behavioral modeling is one modeling approach used by different safety, reliability, and diagnostic analyses. It is important to note that this dissertation only deals with analysis of discrete models. Continuous models have different properties that require other types of safety, reliability, and diagnostic approaches. System representations and analyses will only pertain to discrete systems.

Since reliability, safety, and diagnostics deal with undesired system behavior, behavior under both nominal and failure conditions must be described to fully capture system behavior. Figure 6 shows how a system must be approached for an integrated analysis. Since safety, reliability, and diagnostic analyses require knowledge of system behavior, the system can be thought of as a device that takes inputs, produces outputs, and is impacted by faults. Depending on the history of the device, and the inputs and faults currently detected, the system may react and produce some output. The fault inputs are a special class of inputs: they are input into the system when a component failure occurs. This allows the system models to reflect how the system will react to component failures as well as to normal system inputs.

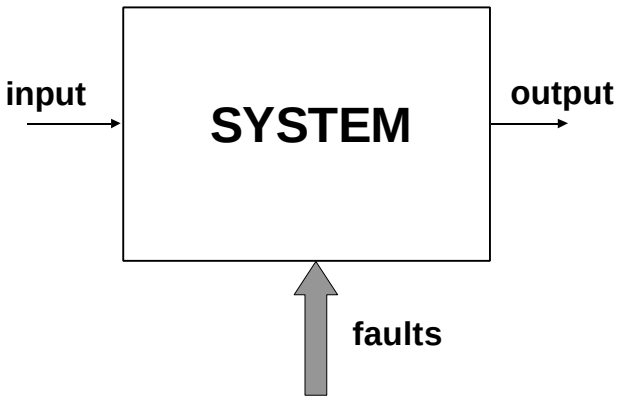

Figure 6: System view for integrated analysis

In order to use the system model described above, the FSM definition given in Chapter II must be extended to include fault information. Faults can be treated as a special class of input event that is asserted in the FSM model whenever the corresponding component failure manifests itself [40]. The addition of faults forces the modification of the FSM definition. An FSM model of a dynamic system (system with memory) is shown on the left side of Figure 7. The FSM model for the system in Figure 6 is the $(X, F_Y, F_S, S, G, \boldsymbol{f}, s_0, Y, \boldsymbol{g})$ finite state automaton, where:

X is the input event set,

F_S, F_I are the sets of component faults and instrumentation faults, both considered to be input events,

S is the state set,

$G(s)$ is a set of feasible or enabled events, defined for all $s \hat{I} S$ with $G(s) \hat{I} X$,

$\boldsymbol{f}$ is a state transition function, $\boldsymbol{f}: X \acute{} F_S \acute{} S \rightarrow P(S)$, defined only for $x \hat{I} G(s)$ when the state is s,

s_0 is the initial state,

Y is the output set,

$\boldsymbol{g}$ is an output function, $\boldsymbol{g}: X \times F_S \times S \rightarrow Y$, defined only for $x \in G(s)$ when the state is s,

Z is the observation set, and

$\boldsymbol{h}$ is an observation function, $\boldsymbol{h}: Y \times F_I \rightarrow Z$.

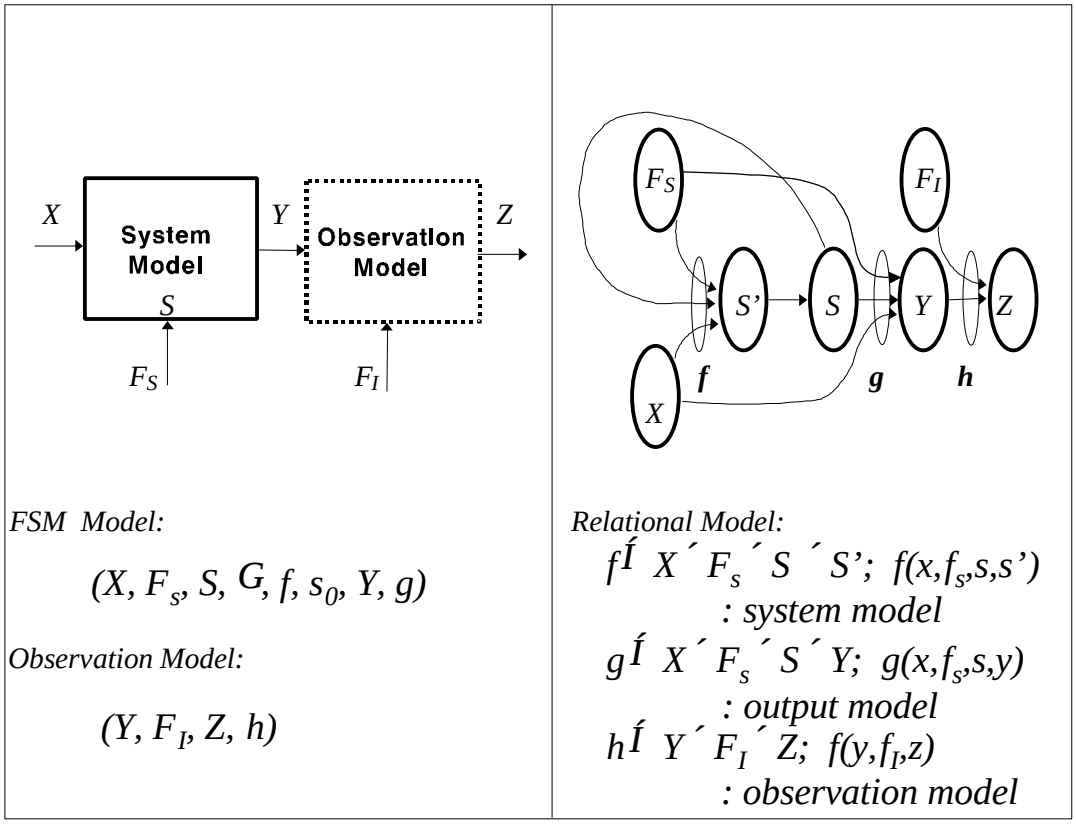

Figure 7: FSM and Relational Models

Since the models need to support diagnostic analyses and provide support for an active reliability management system, the model is divided into a *System model* and an

Observation model. The System model describes the behavior (nominal and fault) of the system and the observation model is used to model how sensors react to outputs of the system. The component faults are considered as additional inputs to the system. It is also possible to model the abnormal (out of range) inputs as elements of the X input set, creating a 'normal' and 'faulty' partition in X. In order to model partial observations of the state trajectory independently from the outputs of the dynamic system, we use the $\boldsymbol{h}: Y \times F_I \rightarrow Z$ observation model which describes the mapping between the Y outputs, F_I instrumentation faults, and the Z observations. The FSM formalism also allows the representation of non-deterministic state machines.

The right side of Figure 7 shows the equivalent *Relational Model* of a dynamic system. In the relational model, the $\boldsymbol{f}$, $\boldsymbol{g}$ and $\boldsymbol{h}$ mappings are considered to be the $\boldsymbol{f} \subseteq X \times F_S \times S \times S'$, $\boldsymbol{g} \subseteq X \times F_S \times S \times Y$ and $\boldsymbol{h} \subseteq Y \times F_I \times Z$ relations. The significance of the relational representation is that the models can be re-written as Boolean functions by introducing some binary encoding for the sets $X \times F_S \times S \times S'$, $X \times F_S \times S \times Y$ and $Y \times F_I \times Z$. The Boolean functions $\boldsymbol{f}(x,f_S,s,s')$, $\boldsymbol{g}(x,f_S,s,y)$ *and* $\boldsymbol{h}(y,f_I,z)$ evaluate to *true* for those elements of the sets $X \times F_S \times S \times S'$, $X \times F_S \times S \times Y$ and $Y \times F_I \times Z$ (encoded by the Boolean vectors (x,f_S,s,s'), (x,f_S,s,y) and (y,f_I,z)) that are related by the $\boldsymbol{f}$, $\boldsymbol{g}$ and $\boldsymbol{h}$ relations. The Boolean representation of the FSM model can be directly related to a relational representation, allowing the symbolic evaluation of the system models.

The observation model is included in the FSM model for the possible inclusion of diagnosability as an analysis. Diagnosability analysis can be used to determine if multiple failures can be identified and distinguished. In order to provide for an active reliability management system, there must be an *observer* to detect failures and provide

input to the reliability management system detailing what failures have occurred. The observation model can be described as a FSM model, but it operates outside the scope of the system model.

Integrated Modeling of High Assurance, High Consequence Systems

The integrated model for high assurance, high consequence system analysis is based on system behavior models and system structure (or physical) models. Safety, diagnostic, and reliability views of the models can be abstracted from the integrated model. Using an integrated approach, system level changes will be evident in the other system views only if the modification affects the specific view.

A key element of integrated modeling involves the interaction of different modeling aspects. Previously, issues related to the areas of dependence had to be resolved manually [40]. By using an integrated approach, changing the model in one aspect may affect many different aspects of the model. The system can be modeled in a more natural format. Instead of requiring safety, reliability, and diagnostic models of a system, the system can be modeled in a manner that is more natural to the system designer. When performing a safety or reliability analysis, the specific information of interest can be extracted from the augmented behavioral model automatically. Safety, reliability, and diagnostics are not separate, independent traits of a system [27].

Nominal and Fault Behavior

Behavioral modeling must encompass both nominal and fault behaviors in order to capture the necessary information for performing safety, reliability, and diagnostic analyses. It is assumed that a verified model will not exhibit safety or reliability

problems in the nominal behavior. Due to this assumption, nominal behavior must be extended with the fault behavior of the system.

The fault behavior of a system is a natural extension of the system's nominal behavior. Nominal behavior represents the "normal" modes of operation for a system while the fault behavior extends this representation to include all of the modes of operation. By extending the behavioral representation, the system modeler can capture the complete, and often complex, behavior of the system. Figure 8 shows a visual representation of the differences between nominal and fault behavior. Please note that the full fault behavior of the system does not need to be modeled. Distinct fault states do not have to be decomposed into their smallest states. By showing the system exits nominal behavior for fault behavior, even without specifying the details of the fault behavior, the system can accurately depict when fault behavior will be exhibited.

An integrated modeling paradigm must be able to capture both nominal and fault behavior. Any safety, reliability, or diagnostic analysis of a set of models could not be considered complete unless both the nominal and fault behaviors of the system are analyzed. This is true due to safety and reliability analyses relying on information about the system exhibiting undesirable (i.e. fault) behaviors. Diagnostic analysis is based on determining why a system was exhibiting fault behavior. The ability to capture both nominal and fault behavior is a necessary feature for an integrated safety, reliability, and diagnostic modeling and analysis environment.

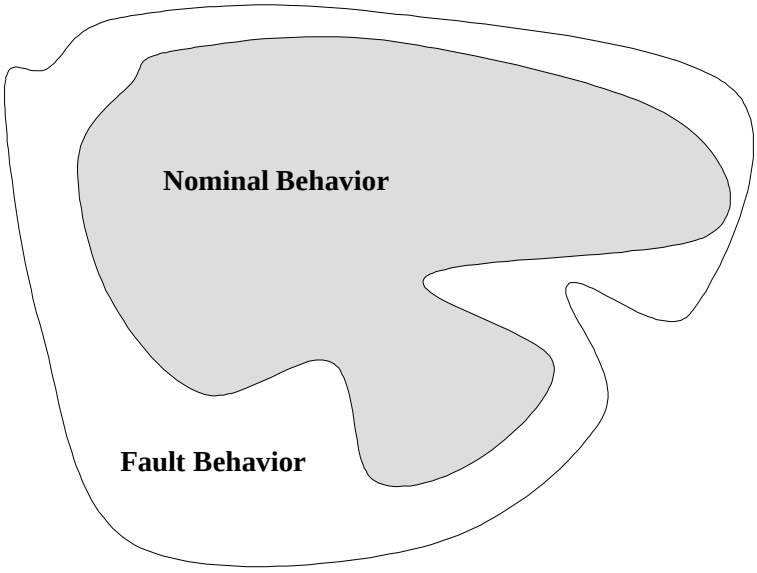

Figure 8: System behavorial space

Selection of Domain Specific Modeling Paradigm

Safety, reliability, and diagnostic analysis algorithms work with a "model" (a suitable representation) of the system. The required depth of the analyses determines the level of detail of the models. This section will show it is possible to perform safety, reliability, and diagnostics on the relational model of Figure 7. Assuming there exists a method to transform the FSM model into a relational model, the FSM modeling can become the basis for the domain specific modeling paradigm. The following sections will substantiate this claim.

Given a set of inputs to the system (X), the set of faults (F_s), the set of instrumentation faults (F_I), and the state of the system (S), the following sets can be calculated:

$$S' = f(X, F_S, S) = \{s' \mid \exists x,\ f_S, s[(x \in X) \wedge (f_S \in F_S) \wedge (s \in S) \wedge (x,\ f_S, s, s') \in f \wedge (x \in \Gamma(s))]\} \quad (6)$$

$$Y = g(X, F_S, S) = \{y \mid \exists x,\ f_S, s[(x \in X) \wedge (f_S \in F_S) \wedge (s \in S) \wedge (x,\ f_S, s, y) \in g \wedge (x \in \Gamma(s))]\} \quad (7)$$

$$Z = h(Y, F_I) = \{z \mid \exists y,\ f_I[(y \in Y) \wedge (f_I \in F_I) \wedge (y,\ f_I, z) \in h]\}. \quad (8)$$

These symbolic expressions calculate a one-step propagation forward in the state automata. The results of the expressions are the new set of possible states (*S'*), the related outputs (*Y*), and the observations (*Z*). It should be noted we can define two relations f^{-1} and g^{-1} that are the inverses of *f* and *g* as:

$$S_b = f^{-1}(X, F_S, S') = \{s \mid \exists x,\ f_S, s'[(x \in X) \wedge (f_S \in F_S) \wedge (s' \in S) \wedge (x,\ f_S, s, s') \in f \wedge (x \in \Gamma(s))]\} \quad (9)$$

$$X_b = g^{-1}(Y, F_S, S) = \{x \mid \exists y,\ f_S, s[(y \in Y) \wedge (f_S \in F_S) \wedge (s \in S) \wedge (x,\ f_S, s, y) \in g \wedge (x \in \Gamma(s))]\}. \quad (10)$$

S_b and X_b define a one-step propagation backward in the state automata. The relations f^{-1} and g^{-1} allow the determination of the previous states (S_b) and corresponding input events (X_b) from a given state (*S'*) and output (*Y*) of the system.

The above relations are key in defining safety, reliability, and diagnostics analyses on the relational model. The following sections will demonstrate how to perform safety, reliability, and diagnostics on the relational model of Figure 7. Details as to how to perform each type of analysis will be given.

Safety Analysis and the Relational Model

Safety analysis requires the development of models that represent the relationship between failure modes of physical components and discrepancies in the high-level behavior of the system. In order to perform a safety analysis, a system starting state (s_o), and a set of safety critical states (S_C) must be defined. A set of reachable states, denoted as S_R, can be found by computing the transitive closure of f using a fixed point calculation. Starting from $S_R = \{s_0\}$, we must calculate S_R where:

$$S_R' = S_R \cup \{s' \mid \exists x, f_S, s[(x \in X) \wedge (f_S \in F_S) \wedge (s \in S_R) \wedge (s \in S) \wedge (x, f_S, s, s') \in f]\}. \quad (11)$$

S_R must be iteratively calculated until the point where $S_R' = S_R$ is reached. At this point, the set S_R contains all reachable states from s_0.

After calculating S_R, the safety analysis requires determining the intersection of S_R and S_C. If $S_R \cap S_C = \varnothing$ then the system is safe: the system can never enter into a safety critical state. Otherwise, it is possible for the system to enter a safety critical state. The system cannot be declared safe in this case.

Possible extensions to the safety analysis on the relational model could include determining all events that could lead to the safety failure. By allowing the analyst to identify possible failure causes, the analyst or modeler would have an area of the system design to target for modifications. The set F_R will contain all component failures that could have led to the system safety failure. F_R can be calculated by computing the set

$$F_R' = F_R \cup \{f_S \mid \exists x, f_S, s, s'[(x \in X) \wedge (f_S \in F_S) \wedge (x, f_S, s, s') \in f \mid (x \in \Gamma(s)) \wedge (s' \in S_R) \wedge (s \notin S_R) \wedge (s \in S_R')]\} \quad (12)$$

along with each iteration of computing S_R. The terms $(s' \in S_R), (s \notin S_R)$, and $(s \in S_R')$ ensure that states are only examined once and that only states that are along the trajectory from the initial state configuration to the state configuration representing the safety

failure are examined. Whenever the point where $S_R' = f(X, F_S, S_R)$ is reached, the set F_R is complete. It is important to note that the relational model captures all of the necessary information for performing safety analysis.

Reliability Analysis and the Relational Model

Reliability analysis involves determining all possible causes of a system losing a required functionality. Reliability analysis inherently requires enumeration of trajectories leading to the failure state of the FSM model of Figure 7. To realize the trajectories, a tree structure is constructed where each node in the tree is defined as a tuple $R_i = (X_i \hat{I} X, F_i \hat{I} F_S, s \hat{I} S, k$ *[R_k is the parent of R_i])*. The root of the tree is defined as

$$R_0 = (Æ, Æ, s_R, -1), \tag{13}$$

where s_R represents the reliability failure state of concern ($s_R \hat{I} S$). The rest of the tree is defined by the following relation:

$$R_i = (X_i, s, F_i, k \mid \exists X_i, sF_i, s', k[(X_i \in X) \wedge (F_i \in F_S) \wedge (s \in S) \wedge (s' \in S) \wedge (s' \in R_k) \wedge (s, x, f_s, s') \in f \wedge (X_i \in \Gamma(s))]). \tag{14}$$

The R_i tuples are computed until all the leaf nodes in the tree contain the system starting state (s_o):

$$\forall R_i \mid \neg\exists R_j[(R_j(3) = i)] \; s_0 = R_i(2). \tag{15}$$

The nodes of this tree provide the enumeration of all possible states, events, and failures that can lead to the reliability failure becoming manifest. Each of the nodes in the tree must be unique:

$$\forall i, j[(i = j) \rightarrow R_i \equiv R_j]. \tag{16}$$

This ensures each node in the tree can be uniquely identified as R_i. The tree from R_0 specifies all possible trajectories that could lead to the selected reliability failure state (s_R). f^{-1} can be used in performing this analysis.

The models for reliability analysis have strong overlap with the models for safety analysis. Behavioral and physical models contain all the information required for reliability analysis except failure rate data for the component fault modes. Therefore, by extending the component fault models with probabilistic information, the modeling paradigm will allow safety and reliability analysis to be performed on the integrated set of models.

Diagnostic Analysis and the Relational Model

The models for diagnostic analysis have strong overlap with the models for safety analysis and reliability analysis. Offline diagnostics requires the calculation of a hypothesis set D for a given set of observations Z. The hypothesis set can be defined as:

$$D = d(Z) = \{x, f_S, f_I \mid \exists y, z[(z \in Z) \wedge (y \in Y) \wedge (y, f_I, z) \in h \wedge (x, f_S, y) \in f]\}. \quad (17)$$

$d(Z)$ propagates the observations backwards to obtain the set of admissible faults, which together form the diagnosis. The D hypothesis set contains all of the $d \in X \times F_S \times F_I$ hypotheses that are consistent with the observations.

Another diagnostics method involves on-line diagnosis, which is necessary for an active reliability management system. In an on-line system, new observations are gathered from the running system. As new observations become available, the hypothesis set is refined with the new data. Eventually, the diagnostics system will converge on a hypothesis set which includes all possible explanations for the observed

system trajectory. The hypothesis set $D \subset X \times F_S \times F_I \times S$ is recalculated whenever a new observation is made. Assuming the faults (including possibly faulty inputs) are persistent, the hypothesis set can be defined by:

$$D_{j+1} = d(D_j, Z_{j+1}) = \{x, f_S, f_I, s' \mid \exists s, y, z[(z \in Z_{j+1}) \wedge (y, f_I, z) \in h \wedge ((f_I \in F_{I,j}) \vee (f_i \in F_I)) \wedge (s, x, y) \in g \wedge ((x \in X_j) \vee (x \in X)) \wedge (x, f_S, s, s') \in f \wedge ((f_S \in F_{S,j}) \vee (fs \in F_S))]\}. \quad (18)$$

This assumes faults are persistent during the observation. If this assumption cannot be made, the terms $(x \hat{I} X_j)$, $(f_S \hat{I} F_{S,j})$, and $(f_I \hat{I} F_{I,j})$ must be removed from the calculation of D_{j+1}. Lastly, the diagnostic system may need to determine not only the possible fault combinations that lead to an observed behavior but to enumerate the ordering of the faults as the system transitions to the observed behavior. The above reliability relation R may be used to enumerate all possible trajectories between observed states. This enumeration may be used to determine why the system behaved in a specific method. This off-line diagnostic method is useful for recommending design changes as observations can be traced back to the corresponding models.

The conclusion is that behavioral and physical models contain all the information required for diagnostics analysis except failure rate data for the component fault modes [9][2]. Therefore, by extending the component fault models with probabilistic information, the integrated modeling paradigm will allow safety, reliability, and diagnostic analyses to be performed from the integrated model set.

Integrated Modeling with the MultiGraph Architecture

The MultiGraph Architecture (MGA) contains a configurable model editor for construction of multi-aspect, domain specific models. Appendix A contains details of the

MGA. In order to configure the MGA model editor, a *metamodel* must be constructed. This metamodel describes the syntax, static semantics, and presentation semantics of the domain specific modeling paradigm [41]. This section will detail the metamodel developed for the integrated modeling paradigm to be used for safety, reliability, and diagnostic modeling. Information about the paradigm includes what objects are needed to capture system information, how the objects are related, and how the objects are presented. With the metamodeling technology described in [41], objects must be mapped to the available visual resources of the MGA model editor.

In order to develop a domain specific modeling paradigm, first the key concepts that must be captured in the models must be identified. Among the reasons for using a domain specific modeling paradigm is the ease of use to the system modeler. Another feature necessary for the domain specific modeling paradigm is the scalability of the modeling paradigm. Modifying a set of models should require effort proportional to the size of the change to be made, not proportional to the size of the models.

While safety, reliability, and diagnostics are all defined in terms of the relational model described above, the modeling paradigm should focus on a more abstract behavioral model of the system. The FSM model described above makes an excellent beginning point for the domain specific modeling paradigm. Some types of FSM models are scalable and very natural to use for describing behaviors. Due to this, the integrated modeling paradigm will be based on FSM models.

Behavior Modeling

Figure 9 illustrates the UML object model for the behavioral specification of the integrated modeling paradigm. Statecharts were selected as the basis for the behavioral

models of the integrated paradigm. Statecharts were selected due to the scalability of the representation, the expressiveness of the language, and the ease-of-use to the modeler. Since training in the use of the integrated paradigm is an issue, incorporating well known modeling techniques in the domain specific paradigm is attractive. The first object needed to construct Statechart style models is a *State*. In order for the representation to be scalable, State hierarchy must be supported. Thus, States can contain other States. States also have the ability to be defined as an AND, OR, or LEAF state. This represents the cases where a state has an orthogonal decomposition, an exclusive-or decomposition, and where a state has no child states.

In addition to States, *Events* are needed to signal the occurrence of happenings that may affect the behavioral model. The concept of an event in the integrated paradigm is borrowed directly from the Statecharts notation. *Parameters* are global, Boolean variables used to configure a behavioral specification. They can be used to ensure specific transitions cannot occur unless certain parameters are set to specified values. This is a limited application of the concept of *Guarding Conditions* as they are defined in Statecharts.

Transitions are objects used to specify Trigger and Guarding Conditions. Transitions are used to model a transition from one state to another. An object was introduced in the metamodel to represent a transition. This object is only used to capture transition information. Instead of the traditional *e[g]/a* formalism used in Statecharts, a trigger statement is a Boolean expression of events. The guarding condition is also a Boolean expression, but of parameters and States. A transition is enabled only if the trigger and guarding condition hold true. Actions are restricted to events. By specifying

an action as an event, whenever the corresponding transition is taken, the event is made to occur.

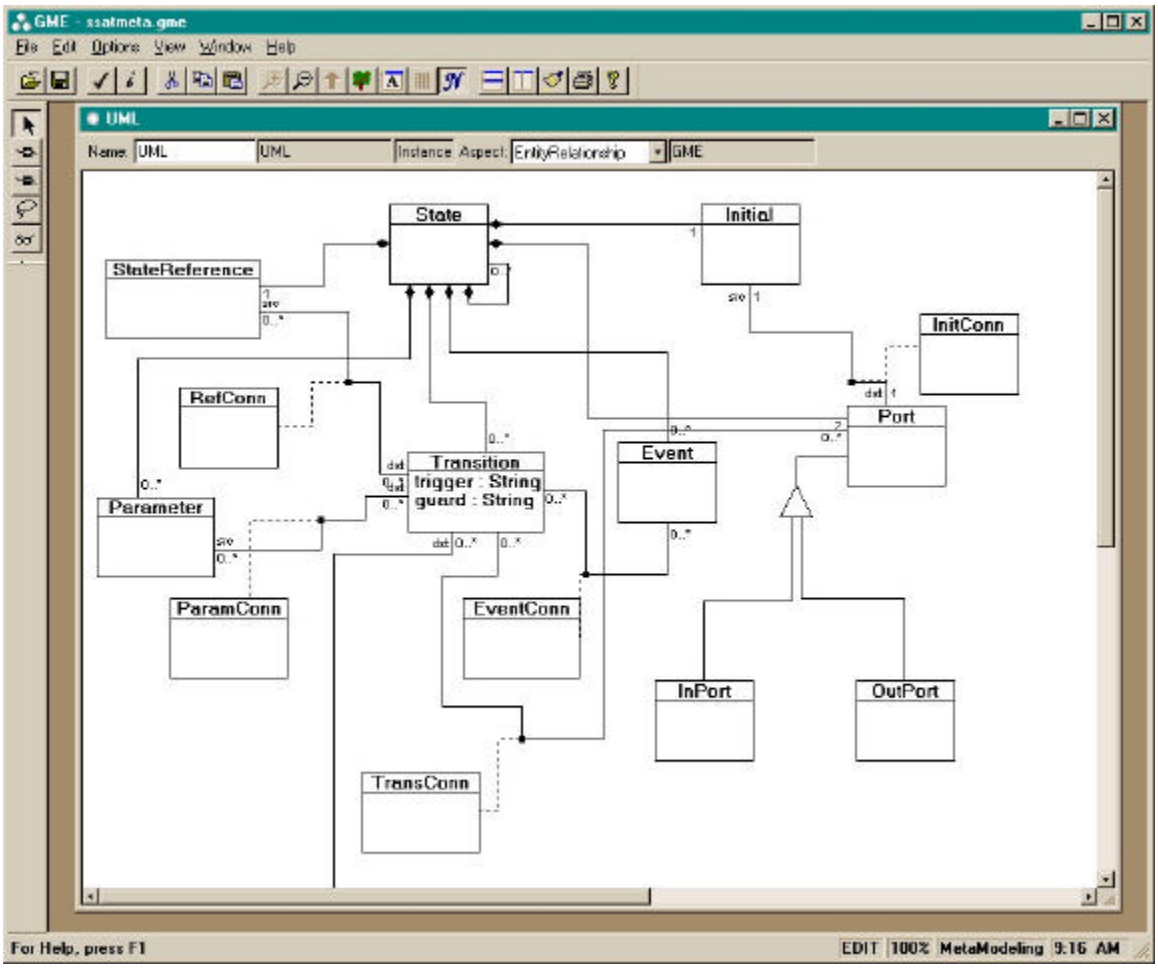

Figure 9: Metamodel for Behavior

Statecharts use a special transition connector for specifying *Initial States*. MGA does not support this visual formalism, so a special object was created for specifying Initial States. The Initial States object can be connected to any child State to specify the current default State. This concept is borrowed directly from Statecharts.

States are mapped to MGA *models*, as States must contain other objects and this concept is a perfect match to MGA *models*. Unfortunately, MGA does not support connecting models. MGA implements *module interconnection* [42] which only allows connections to be made to *ports* of models. These ports are atomic parts that appear on the model. States must contain two ports in the integrated paradigm; one must be an input port, the other an output port. These ports are used to facilitate connections with the Transition objects.

The relational model discussed above incorporates failures. These failures are used to represent the occurrence of component failures in the system. As previously stated, to complete model behavior both nominal and fault behavior must be represented. Due to this requirement, a specialization of a state, a *Failure State* was introduced. Failure states have all the same properties as normal states. They are used solely to represent the segmentation of the behavioral model into nominal and failure behaviors. It is intended that only through component failures can a failure state be entered. This concept is not enforced in the modeling.

Component faults are introduced as a special type of event. Internally, they are treated as events. However, they are used as inputs to transitions between nominal and failure system states. Component faults are necessary to illustrate when the system transitions from nominal to fault behavior.

Structure Modeling

The domain specific modeling paradigm must incorporate system structural information in addition to behavioral modeling. Structural models are intended to show the different components in the system, how they are connected, and their failure modes.

Failure modes for assemblies of components must be allowable in the paradigm. Even if the components do not fail, an assembly of components may fail. Figure 10 shows the structural modeling description from the metamodel.

Components are mapped to *models* in the MGA model editor. Components are allowed to contain other components (hierarchy), ports, and failures. Ports are again used for module interconnection. By connecting different components together, it is possible to graphically show the components interact. The details of this interaction are not needed, as the structural model is used only to capture model structure and component failure modes.

Component failures are modeled as specific objects to allow components to contain multiple failures. Also, by using *parts* to represent failures, sub-assemblies of components can contain failures. Component failures have an attribute to allow specification of failure properties. This is not used directly, but is intended to be a "pointer" to documentation on the failure mode. Component failures can take so many forms, from single probabilities to complex probability distributions, that it is not practical to attempt to capture failure probabilities in the MGA models.

Integrated Metamodel

Figure 11 represents the complete metamodel for the domain specific modeling paradigm. Some association between the structural models and the behavioral models is necessary to model how component failures can affect system behavior. Specifically, component failures can be used to illustrate the failure events used to transition the behavior into a failure state. The complete metamodel shows the objects needed to perform behavioral modeling, fault behavioral modeling, and structural modeling. By

explicitly modeling the interactions between behavior and structure, the models are able to capture the effects of component failures on system behavior.

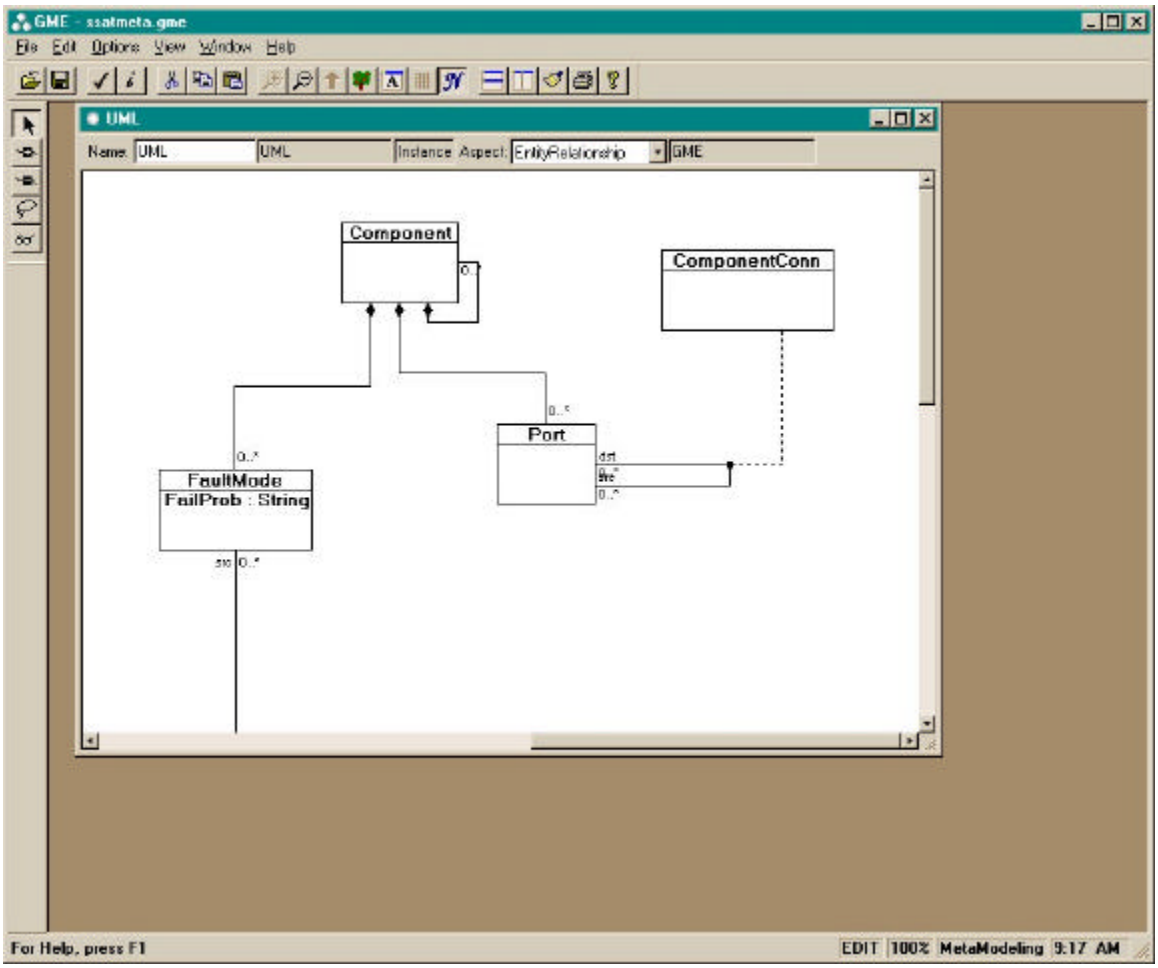

Figure 10: Metamodel for Structure

MGA Modeling Environment

An addition to the Statecharts formalism was the determination of nominal and fault behaviors. Whereas Statecharts model behavior as one type of "state", GME has the ability to perform *multiple aspect modeling*. Using this feature, the domain specific paradigm is able to separate the nominal behavior of the system from the fault behavior.

In the fault behavior aspect, the nominal behavior is extended with the fault behavior of the system. With GME, objects and connections can be *inherited* between model aspects. This feature allows the fault aspect to always extend the nominal behavior of the system. Figure 12 is a GME model constructed using the integrated paradigm. It illustrates a small behavioral model with fault information. The larger gray boxes are states. Icons with a "T" in the center are transitions. Icons with an "E" are events; faults are denoted by the red icons with the yellow lightening bolt.

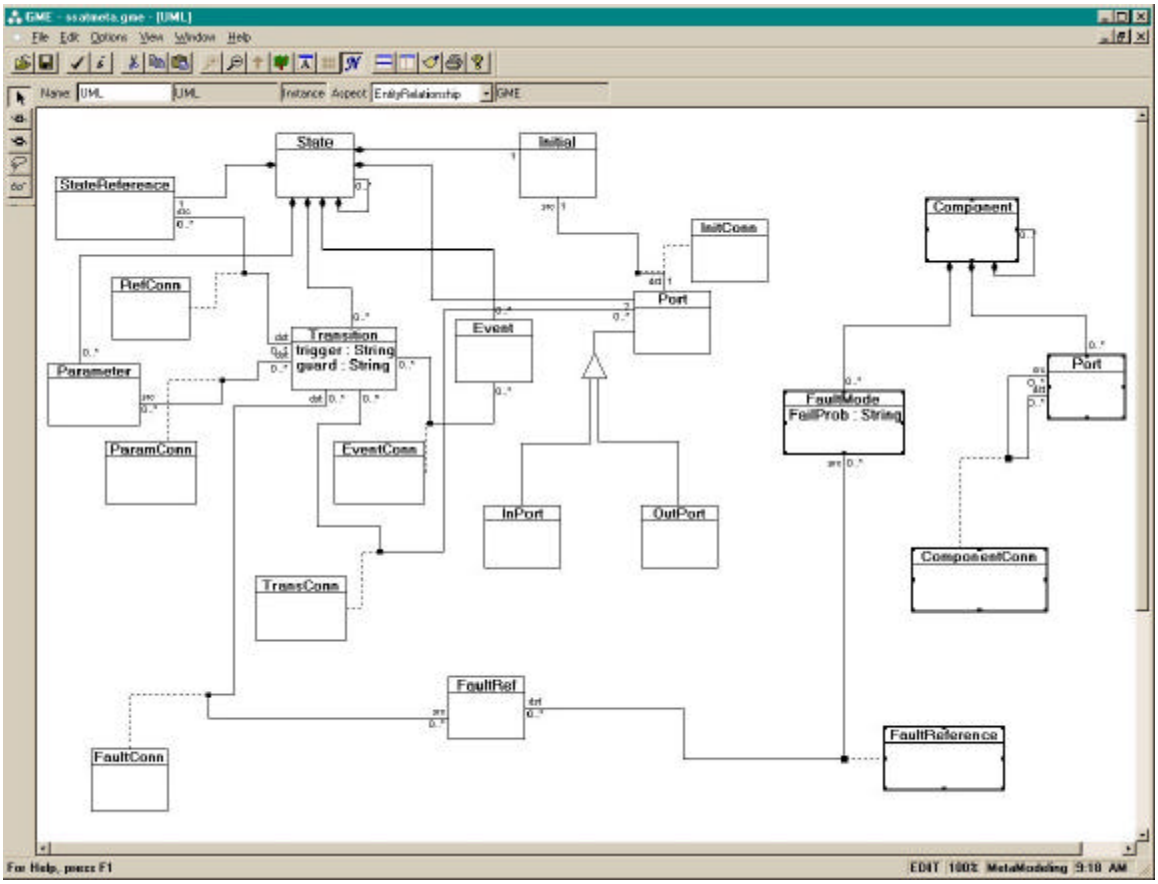

Figure 11: Complete Metamodel

The paradigm described is utilized for system specification. An integrated analysis environment will be described in Chapter IV that allows safety, reliability, and diagnostic analyses to be performed on the models constructed with the paradigm.

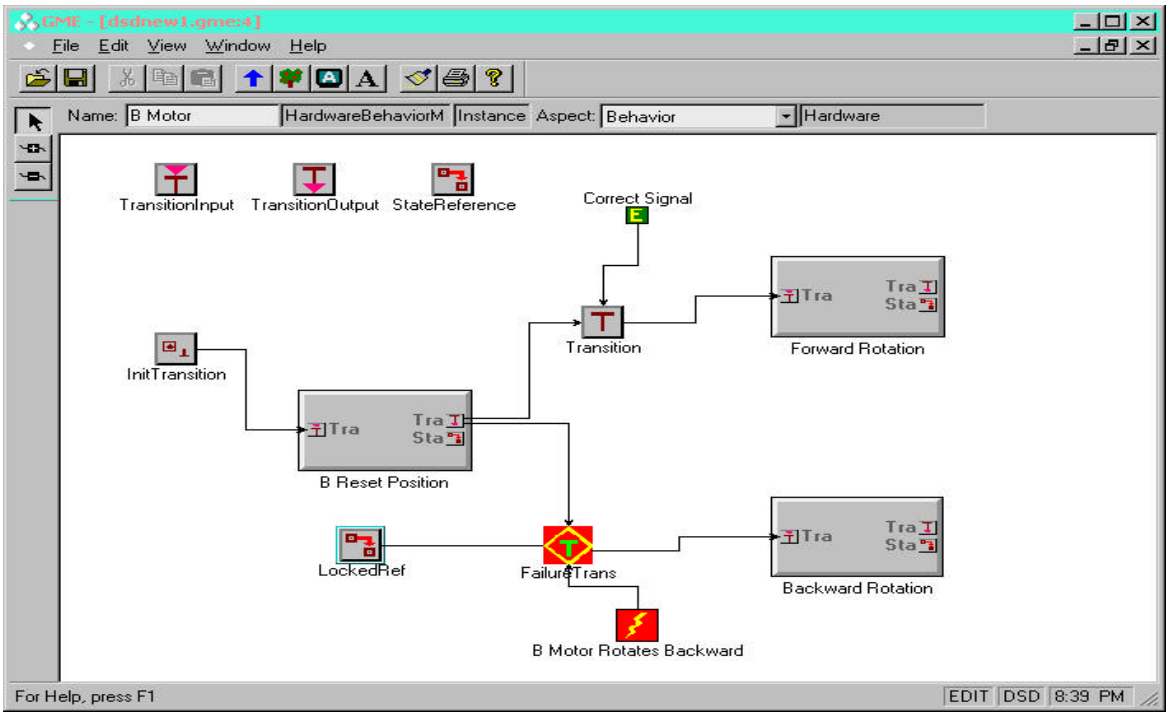

Figure 12: Example behavioral specification

CHAPTER IV

INTEGRATED ANALYSIS

An integrated analysis environment (IAE) must utilize the domain specific modeling tools to capture system specifications. The IAE behavioral specifications and physical system models serve as the basis for extracting safety, reliability, and diagnostic information about the system design. Safety and reliability are inherently *emergent* properties – that is, safety and reliability are not simply measurable features of a system. Instead, they are determined by the different components of a system, how the components are assembled, how the components behave, and how the components interact with other components, subassemblies, and systems. Safety and reliability analyses rely on extracting information from the behavioral specifications of a system about the interrelations of the component behaviors [27]. Diagnostic information is derived from the behavioral specifications and structural models to determine under what conditions an observed system failure could have occurred. By dissecting the behavioral specification, *all* of the combinations of component, or assembly, failures that could have caused the observed system behavior can be identified [8]. This chapter details the concepts of an integrated analysis environment for safety, reliability, and diagnostics of high consequence, high assurance systems.

The first step in developing an IAE is to develop a formal model for representing the modeled behavioral specifications. Traditional behavioral analysis techniques rely on the ability to completely enumerate the modeled state space. Employing symbolic model checking techniques allows the complete behavioral specification to be explored without

requiring explicit state space enumeration [21]. This eliminates the state space explosion problem. Without a formal model for the behavioral specification, analysis of large models (with respect to state space size) would not be possible. Chapter III describes the modeling concepts for the integrated analysis environment. Safety, reliability, and diagnostics are defined in terms of the relational model presented. In order to develop an IAE, a mapping from the domain specific model into a symbolic representation of the relational model must be developed. The rest of Chapter IV details this mapping and then describes the different algorithms necessary to perform safety, reliability, and diagnostics analyses on the integrated models.

Ordered Binary Decision Diagrams

Safety, reliability, and diagnostic analysis tasks use discrete models and operations over finite domains. The most common difficulty in all three of the analysis techniques is the size of the state space required by large-scale systems. Combinatorial explosion is the result of the exponential increase in the number of discrete elements (states, hypotheses, etc.) during operations, which sooner or later makes access to the individual elements unfeasible [43]. By introducing a binary encoding for the elements, the individual elements, sets of elements, and relations among them can be expressed as Boolean functions. This allows the realization of the relation model given in Figure 7.

Expressing the relational model as Boolean functions allows for symbolic manipulation of the models. Otherwise, the exponential increase in the discrete elements specified in the model will make manipulation of the models, and the examination of individual elements, impossible. For example, the 2^{100} states of a finite state automaton

can be encoded with binary variables *{s(1),…,s(100)}* forming a binary state vector ***s***. The Boolean functions

$$f_1[s(1),\ldots,s(100)]=s(1) \wedge s(23) \wedge s(99) \text{ and} \quad (19)$$

$$f_2[s(1),\ldots,s(100)]=s(1) \wedge s(22) \wedge s(89) \quad (20)$$

represent two subsets, S1 and S2, of the 2^{100} states including 2^{97} elements each. The set S3=S1∪S2 can be derived symbolically as the disjunction of the two Boolean functions:

$$f_3[s(1),\ldots,s(100)] = f_1[s(1),\ldots,s(100)] \vee f_2[s(1),\ldots,s(100)]$$

$$= s(1) \wedge s(23) \wedge s(99) \vee s(1) \wedge s(22) \wedge s(89) \quad (21)$$

without the need to enumerate and compare the individual elements, which would be a formidable task. In general, using Boolean function representations, operations and algorithms in diagnosis, reliability, and safety analysis can be expressed by means of symbolic Boolean function manipulations.

OBDDs provide a symbolic representation for Boolean functions in the form of directed acyclic graphs [38]. Figure 13 illustrates an example OBDD graphically. They are a restricted, canonical form version of Binary Decision Diagrams (BDD) [44]. Bryant [43] described a set of algorithms that implement operations on Boolean functions as graph algorithms on OBDDs. Taking advantage of the efficient symbolic manipulations, researchers have solved a wide range of problems in hardware verification, testing, real-time systems, and mathematical logic using OBDDs that would have otherwise been impossible due to combinatorial explosion. Symbolic model checking is extensively used in hardware design (see, e.g., [45]) and has shown to be efficient in state space sizes 10^{120} and beyond.

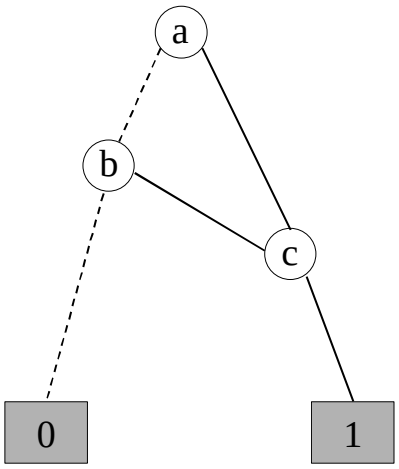

Figure 13: The OBDD for (a∧c) ∨ (b∧c)

OBDDs can be used to represent a system behavioral model in a mathematical format. Using graph algorithms, the system state space can then be explored mathematically instead of using traditional state exploration techniques, thereby eliminating the state explosion problem. OBDDs allow symbolic operations to be performed by providing a canonical, mathematical system representation. A canonical representation is necessary to allow direct comparison between two OBDDs. If the representation is canonical, the order the OBDD was constructed does not affect the resulting structure. Since OBDDs are canonical, determining if two OBDDs are equivalent can be done by determining if the OBDDs are isomorphic [38].

A disadvantage to using OBDDs for representing the relational model is scalability. The size of an OBDD representation is sensitive to the variable ordering when constructing the OBDD. There is no known method for efficiently discovering an optimal variable ordering for a system before assembling the OBDD. Heuristic methods are usually employed to ensure *most* systems' OBDD representations will be acceptable

in size. Another disadvantage to using OBDDs for representing the relational model involves developing routines to convert system models into OBDDs and to convert OBDD based results back into system model information. End users of the modeling and analysis environment do not desire to work directly with OBDDs. Instead, routines must be developed to convert from system models to OBDDs and from OBDDs back to system models. In addition, all analysis routines must be implemented using graph-based algorithms. High level algorithms must be converted into their basic, mathematical operations in order to be applied to OBDD system models.

Integrated Analysis With OBDDs

The primary difficulty with safety, diagnostic, and reliability analysis with state space modeling representations is combinatorial state space explosion. For example, safety analysis performed on the behavioral model requires the exhaustive enumeration of all possible state configurations that may be reached from an initial state under any conditions. By representing the behavioral model symbolically as an Ordered Binary Decision Diagram (OBDD), the required calculations can be completed symbolically without explicitly enumerating the exponential number of alternatives.

The application of OBDDs for the analysis requires the following steps:

1. Mapping the behavioral models into OBDDs: This step is completed automatically. In accordance to the general framework of the Multigraph Architecture (MGA), the integrated models in the Model Database are traversed by a Model Interpreter, which selects a binary encoding for the states and incrementally builds up the OBDD representation for the relational model.

2. Safety analysis: Safety analysis works directly with the relational representation of the behavioral model. The safety analysis methods detailed in Chapter III must be converted into a form that can operate directly on the OBDD system representation. Methods must be developed for transforming model information to and from the OBDD representation. The reachability set is calculated symbolically, therefore safety analysis is feasible even for very large state spaces.
3. Reliability analysis: The selected reliability analysis tool, WinR, expects a fault tree that represents all possible combinations of fault events leading to a selected top event. The analysis algorithm described in Chapter III enumerates all of the state trajectories leading to the top event using backward simulation, and simultaneously builds up the logic relationship between the fault events and the top event. This method will have to be implemented such that it can operate directly on the OBDD system representation. Construction of an analysis tool independent fault tree must be performed by this algorithm. Eventually, the tool will transform the fault tree to a format that is analyzable by available tools (i.e. WinR).
4. Diagnostic analysis: The diagnostics method given in Chapter III must be implemented to operate on the symbolic representation of the system. Diagnostic analysis can be performed using a fault tree. Therefore, the fault tree generation algorithm can be utilized here also. Instead of failure probabilities, the fault tree will be used to document which component failures and what failure combinations are necessary to manifest the

observed behavior in the system. The diagnostics algorithm must be able to work directly on the OBDD model representation.

From here, Chapter IV will detail how the four steps given above are performed. Detailed algorithms will be given. Lastly, work performed on increasing the scalability of the system will be presented.

Symbolic Representation of Finite State Machine Models

In order to perform symbolic analysis of behavioral models, a mathematical representation of the information captured in the behavioral specifications must be developed. The symbolic representation is used for the relational model of Figure 7. This mapping must be extended to incorporate the ability to represent Statechart behavioral specifications symbolically. Techniques for mapping Statechart models to symbolic models were developed at Vanderbilt University based on the work of Helbig and Kelb [46]. Jason Scott has performed this research as part of his dissertation [58]. Specifically, the extended Statechart models of the integrated modeling paradigm are translated into OBDDs. The rest of this section details how this translation takes place.

Boolean Encoding

Boolean variables must be assigned to different states and events in the behavioral specification before a symbolic representation of the Statechart model can be created. A technique has been developed which assigns a unique Boolean encoding to each state in the modeled state space. By utilizing features of the Statechart models, such as hierarchy and orthogonal states, a Boolean encoding utilizing $O(\log_2(s))$ variables can be created, where s represents the number of leaf states in the behavioral specification. This

encoding can uniquely identify any *legal* state configuration in the modeled behavioral specification [48][46]. Events and parameters must also have Boolean variables assigned to them. Since events and parameters are global in scope, each event and parameter can be represented by one Boolean variable. This value of each variable is assigned either 1 or 0 to represent the presence or absence of an event or the value of a parameter during the analysis of the behavioral specification. The number of Boolean variables required is $O(e+p)$, where e represents the number of events in the models and p represents the number of parameters in the models.

The overall Boolean encoding for any unique state configuration in the system uses $O(\log_2(s) + e + p)$ Boolean variables. Using this set of variables, any legal state configuration in the system can be identified. If the symbolic representation allows the additional value assignment of x to represent a "don't care" condition to a variable, then this encoding mechanism can represent sets of state configurations in the behavioral specifications. The value of x provides a shorthand notation that the value of the variable can be either 0 or 1. With the introduction of "don't care" conditions to the variable encoding, any single state configuration, or sets of state configurations, can be identified using the same set of variables. This allows the selected Boolean encoding to be used for representing the complete state, event, and parameter space of the behavioral models. This technique for assigning a variable encoding to the states and events modeled was first addressed by [46].

Individual Transition Relations

Once the task of variable assignment has been completed, a relation must be created to correspond to each transition in the modeled behavioral specification. The

transitions in the behavioral specification define the dynamic behavior of the system. Emergent behavior is determined by examining the interrelations between the individual transitions in the models. For each transition, a relation using 2*m variables is needed where m is the number of variables used in assigning an encoding to the states, events, and parameters. This relation will represent the current state, current events, and current parameters that are necessary to cause the transition to fire. The relation also represents the next state, events, and parameter values that result from the transition firing. These relational representations of a transition appear as:

$$s_0,..,s_i,e_0,..,e_j,p_0,..,p_k,s'_0,..s'_i,e'_0,..,e'_j,p'_0,..,p'_k.$$

The "prime" variables represent the state, event, and parameter variable values *after* the transition fires (the next state of the system).

Since "don't care" values must be assignable to the individual variables, the basic AND, OR, and NOT operations had to be modified to consider "don't cares". The operations are defined as follows:

- $0 \wedge X = 0$
- $1 \wedge X = X$
- $X \wedge X = X$
- $0 \vee X = X$
- $1 \vee X = 1$
- $X \vee X = X$
- $!X = X$

These "don't care" values can be used to represent variables that are not present in the current expression.

Harel's Statechart semantics [3][4] are enforced by the analysis environment. Standard Statechart features such as trigger conditions, guarding conditions, and implied transition priorities due to hierarchy are supported. Non-deterministic behavior is allowed. Transitions leaving super-states (AND/OR states) must be given priority over transitions that may occur within the super-state [4]. In other words, if a transition leaving a super-state is enabled to fire, then the firing of transitions among children of that super-state must be suppressed.

For each BASIC state of the system, states that do not contain other states, a *loop-back* transition with the proper firing condition is created. This transition is required to enable the cases where the model represents staying in the current state. Without the *loop-back* transition, it would require explicit modeling of the conditions necessary to remain in a state.

System Transition Relation

An algorithm for combining individual transition relations into a monolithic relation that represents *all* of the possible transitions that can occur was developed [46]. This relation was developed by combining the individual transition relations according to the model structure. Special relations had to be constructed to ensure the priority of transitions was enforceable. This technique was based on previous research [46] regarding symbolic representation of Statechart models. This monolithic transition relation can be thought of as the $\boldsymbol{f}$ and $\boldsymbol{g}$ relations in the relational model from Figure 7.

Statechart Traversal

Algorithms have been developed to traverse the OBDD representation of the Statechart model. By manipulating the symbolic representation of the behavioral specification, the system analyst can perform vital verification and validation routines on the models as well as higher level analyses. A forward step function allows for traversal through the state space one step at a time. A reachability algorithm has been implemented to exhaustively search the state space to determine if a specified state configuration is reachable from an initial state configuration. Algorithms for determining if the Statechart contains possible "loops" and if the Statechart is deterministic have been implemented. The following sections detail the different algorithms and how they are used to perform analysis on the symbolic representation of the models.

Simulation

Simulation of the symbolic model representation involves utilizing the system transition relation to determine all possible state configurations that could be *immediately* reached from the current state configuration with the current set of event values and the current set of parameter values. By creating a Boolean expression that represents the current state configuration, current event values, and current parameter values with the "prime" variables set to "don't cares", the task of simulation simply becomes an operation of an AND operation between this created Boolean expression and the distributed transition relation. Algorithm 1 defines forward simulation using the symbolic representation [43].

Reachability

The reachability algorithm determines if there are any possible trajectories between two state configurations defined by the transition relation [43]. Reachability relies on the ForwardStep algorithm discussed above. The reachability algorithm is illustrated as Algorithm 2.

```
bdd ForwardStep (bdd CurrentSC ) {
        // NSRelation is the next state relation
        NextSC = NSRelation ∩ CurrentSC
        // quantify out present-state variables
        NextSC = ∃ s0, … s n-1 (NextSC)
        // Move next-state variables to their respective present-state positions
        NextSC = NextSC | s'0→s 0, … , s'n-1→ s n-1
        return NextSC
}
```

Algorithm 1: ForwardStep

Algorithm 2 always determines if there is any path from one state configuration to another state configuration. At times, one would like to determine all possible state configurations that are reachable from a starting state configuration. This is the same process as the safety analysis presented in Chapter III. An algorithm that returns the fixed point reachability set is given below as Algorithm 3.

```
bool Reachability ( bdd start, bdd goal ) {
        I1 = start
        I2 = ForwardStep ( I1 )
```

```
        while ( I2 ≠ I1 )
        {
                I1 = I2
                I2 = ForwardStep ( I1 ) ∪ I2
                // if there is an intersection, we can stop
                if ( I2 ∩ goal ≠ ∅) return true
        }
        return false
}
```

Algorithm 2: Reachability

```
bdd FixedPointReachabilitySet ( bdd start ) {
        I1 = start
        I2 = ForwardStep ( I1 )
        while ( I2 ≠ I1 )
        {
                I1 = I2
                I2 = ForwardStep ( I1 ) ∪ I2
        }
        // I1 is the fixed point reachability set
        return I1
}
```

Algorithm 3: Fixed point reachability

Algorithm 3 computes an OBDD that represents all of the possible state configurations that can ever be reached from the specified state configuration [43]. This algorithm is used in several higher level model analyses. The first reachability algorithm is, on average, more efficient. As soon as a state configuration is found that meets the

goal conditions, the first algorithm concludes. The fixed point reachability algorithm will always continue until a fixed point reachability set is determined. In the worst case, Algorithm 2 and Algorithm 3 will have the same complexity.

Determination of Determinism

Determination of determinism is an algorithm designed to determine if a set of behavioral models contains any non-deterministic transitions. Deterministic transitions are defined as those that, for a given set of inputs, will *always* produce the same next state and the same set of actions [48] [40]. Non-deterministic transitions will not *always* produce the same set of output states and actions. While some natural systems are non-deterministic, high consequence, high assurance systems are usually designed to be deterministic systems. Applying this algorithm allows the system analyst to detect potential problems in the system behavioral specification.

```
bool Deterministic ( States non = ∅) {
        bool deterministic = false
        ∀ s ∈ States {
                bdd next;
                bdd cond = bdd_zero
                next = NSRelation ∩ s
                ∀ n ∈ next {
                        bdd intr = cond  ∩ n
                        if (intr ≠ ∅) {
                                deterministic = true
                                non = non + s
                                }
```

```
                cond = cond ∪ n
            }
        return deterministic
}
```

Algorithm 4: Deterministic determination

While Algorithm 4 will allow the analyst to detect potential problems in the specification, there is no method of automatically correcting the models. Instead, the analyst is notified of the potential problem. The analyst must then either examine, and correct if necessary, the models or notify the system modeler of the potential problems for review and correction.

Loop Detection

Using Statecharts, it is possible to construct models that contain loops. That is, from a given state configuration, some set of inputs can occur that, without any external events occurring, eventually lead back to some previous state configuration. Loops are situations where the system could "oscillate" between states. That is, the actions from a transition could produce the events needed to trigger another transition. If the other transition's actions also trigger the first transition, the system may alternate between two states. Oscillation between states is not usually a desirable design feature. This can be a symptom of an incorrect behavioral specification. Algorithm 5 determines if any potential loops exist in a given behavioral specification and notifies the user of potential loops.

While Algorithm 5 will allow the analyst to detect potential problems in the specification, there is no method of automatically correcting the models. One reason for

not attempting to automatically correct the models is the possibility of a correct specification containing a loop. The existence of a loop does not imply an incorrect behavioral specification; it may be a correct model. The analyst must then either examine, and correct if necessary, the models or notify the system modeler of the potential problems for review and correction [48][40].

```
bool Loops ( ) {
        bdd reach = bdd_zero
        ∀ s ∈ States {
                bdd r = FixedPointReachability(s)
                reach = reach ∩ r
                }
        bdd swap =  reach | s'0↔s 0, ... , s'n-1↔ s n-1
        bdd loops = swap ∪ reach
        if (loops ≠ ∅) return true
        return false
}
```

Algorithm 5: Loop detection

Automatic Fault Tree Generation

Manual derivation of fault trees is a labor-intensive process. In order to build a fault tree, an engineer needs to have a detailed understanding of the system being analyzed, the components of the system, and how the components interact. A deductive process is then undertaken to determine what are the necessary, sufficient, and immediate causes of the failure [27]. These causes are then broken down into their necessary,

sufficient, and immediate causes. This process concludes when the immediate causes are all basic failures, and not complex combinations of failures and system state configurations. Fault trees can be automatically generated from the modeled information.

The algorithm for fault tree generation is given as Algorithm 6 [47][48]. Briefly, the system user defines a system failure in terms of a state configuration of the system behavioral model. Using this information, all of the possible trajectories leading to the failure state configuration can be determined by stepping backward through the system behavioral space. As each trajectory is traversed, failure events along the trajectory are inserted into a tree structure.

```
FaultTreeGeneration(start, goal, Node n) {
        if (goal ≠ start) {
                R = reach(goal, start, NSRelation)
                if (R ≠ ∅)  {
                        B = BackwardStep(goal)
                        ∀ s∈B {
                                Node or = n.add(OR)
                                Node and = or.add(AND)
                                ∀ s∈R, ∀ e∈s {
                                        and.add(e)
                                }
                                FaultTreeGeneration(start,s,and);
                        }
                }
        }
}
```

Algorithm 6: Automatic fault tree generation

Once this internal tree has been built, a mapping to WinR's tree format is applied. This generates a text file WinR can access and analyze. Additional fault tree formats for other analysis tools can be generated from this internal fault tree representation. Using a tool-independent representation allows for later inclusion of other fault tree analysis tools. Other fault tree analysis tools overcome some of the limitations of WinR, such as the limit on the number of nodes in a fault tree [7]. The fault tree requires the system behavior models to be previously verified. Otherwise, the validity of the fault tree can be questioned.

This approach simply eliminates the manual task of deciding every set of conditions that are *immediate, necessary,* and *sufficient* to cause a fault to occur. Manually constructing a fault tree relies on a human to correctly deduce all the possible causes for a fault to occur [7][27]. However, this information is captured in the system specifications. The complete transition information completely defines *all* possible trajectories through the designed state space. All possible immediate, necessary, and sufficient causes of entering a failure state are explicitly modeled. The fault tree generation algorithm uses the backward-step algorithm (Algorithm 7) [43] to exhaustively enumerate all possible combinations of events that can lead to a system failure.

The backward-step algorithm allows the analyst to determine what are the possible previous states for a given state configuration. This allows backward simulation and exploration of the models. It is useful in determining prior conditions that could have

existed in the models, give the current state of the models. The algorithm is detailed as Algorithm 7.

```
bdd BackwardStep (bdd CurrentSC ){
        PreSC = NSRelation ∩ CurrentSC
        // quantify out next-state variables
        PreSC = ∃ s'0, … s' n-1 (PreSC)
        return PreSC
}
```

Algorithm 7: BackwardStep

Scaling Issues and Improvements

The previously described algorithms provide the necessary features to perform safety, reliability, and diagnostic analyses on the OBDD based symbolic models. As the algorithms were applied to sample problems, it became apparent the scalability of the relational models was a concern. Finding an optimal OBDD variable ordering is not efficient. Heuristic methods are used to provide *usually* good variable orderings. However, performing detailed analyses on large models often resulted in unacceptable performance. This section will detail the work performed to increase the scalability of the representation and the algorithms.

Scaling Issues with OBDD Models

The first approach to mapping the behavioral models into OBDDs involved creating one OBDD to represent the transition relation for the entire system model. Upon building models with a total design space of 2^{40}, the monolithic transition relation

grew too large to compute. Performance of the analysis tools became unacceptable once the transition relation had to be stored in virtual memory [49]. In order to allow analysis of larger models, research into scalability of the analysis environment had to be undertaken.

OBDD Virtual Memory Improvements

OBDD based analysis routines rely on heuristic variable ordering methods to produce OBDDs that are efficient in size. Among the improvements possible for OBDDs size reduction are the Cabodi and Quer methods [49]. These methods rely on using a variable to "split" a large OBDD into its cofactors. The conjunction of the cofactors, AND-ed with their corresponding splitting variable values, is mathematically equivalent to the original OBDD. For example,

$$X(v_1, v_2, .., v_m) = X(v_1, v_2, .., v_{n-1}, 0, v_{n+1}, \ldots v_m) \wedge v_n \vee$$
$$X(v_1, v_2, .., v_{n-1}, 1, v_{n+1}, \ldots v_m) \wedge v_n. \quad (22)$$

By splitting the original OBDD into the cofactors, each cofactor can be examined independently. Other improvements detailed in [49] include the ability to cache unneeded OBDDs to disk and bypass the operating system's virtual memory mechanism. By choosing which OBDDs to cache to disk and when to cache them, the analysis routines are no longer dependent on the operating system to manage virtual memory. Selectively caching large OBDDs allows the analysis package to decide when and what OBDDs can effectively be cached to disk. This prevents the operating system from caching OBDDs that will be required in the near future.

Distributed Transition Relation

While it is important that the transition relation represent all of the information from the behavioral specification, for large model sets the size of the transition relation can become unmanageable. Computing the transition relation for large behavioral specifications was not possible. The size of the OBDD representing the transition relation would grow too large to fit in physical memory. Since the transition relation could not be stored in physical memory, any operation that utilizes the transition relation could not be performed without requiring the use of virtual memory. Due to this limitation, the monolithic transition relation could no longer be used.

Rather than representing the transition relation in the form of a single OBDD, a more efficient distributed representation technique has been developed to solve problems encountered when representing large systems. For Statecharts with many orthogonal state components (AND states) the distributed method does not compute the product of the individual transition relations. This distributed transition relation method uses many smaller next-state relations stored in an and/or expression that is derived from the structure of the behavioral model. Each term of this and/or structure is an OBDD. The algorithm for constructing the distributed transition relation was developed by Jason Scott at Vanderbilt University [40].

The and/or structure is never evaluated to a single expression, but remains distributed in a tree structure. Each term is a disjunction or conjunction of transitions for a segment of the statechart. The strategy is to avoid the calculation of the monolithic transition relation. The distributed transition relation, if evaluated, does represent an enumeration of all valid transitions between state configurations, and is equivalent to the monolithic transition relation. This method introduces a slight increase in the complexity

of the forward/backward traversal steps but results in a large decrease in the number of OBDD nodes used to represent the transition relation.

With this new approach, models consisting of a total design space size of 2^{142} have been analyzed [47] without encountering memory size problems. The distributed transition relation technique only requires the memory to store the tree data structures and the individual transition relations. Since each individual transition relation requires relatively little memory, the distributed transition relation requires much less memory than the monolithic transition relation.

Asynchronous Statechart Semantics

The integrated behavioral models allow for multiple transitions to occur at the same time. In Figure 14, there are three methods for transitioning from the state configuration *A&C* to the state configuration *B&D*. Transition *t3* can occur after transition *t1* occurs. Transition *t1* can occur after transition *t3* occurs. Or, transition *t1* and transition *t3* can occur at exactly the same point in time. While this behavior is sometimes useful when modeling physical systems, the enumeration of all possible trajectories between state configurations based on such a model can become quite large.

Assume a behavioral model exists with m parallel, orthogonal, state models, each containing n transitions between contained states. Enumerating all possible combinations of transitions that could occur in one step, forward or backward, would entail

$$\sum_{j=1}^{\lceil m \div n \rceil - 1} \prod_{i=0}^{j} (m - i \times n) \times n \tag{23}$$

unique combinations based on traditional Statechart semantics.

Asynchronous Statechart semantics restrict analysis of the behavioral models to enforce that no more than one transition can occur at any point in time. Physically, asynchronous semantics reflect the assumption that if the units of time are sufficiently discretized, no two changes in state could occur at any single point in time. In the example shown in Figure 14, there are only two trajectories that traverse from state configuration *A&C* to state configuration *B&D*: transition *t3* occurs after transition *t1* and transition *t1* occurs after transition *t3*.

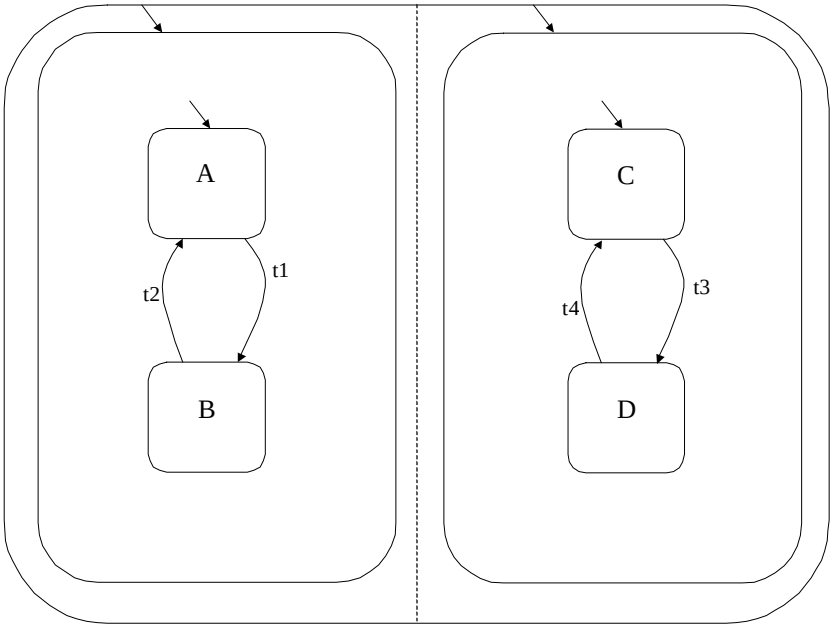

Figure 14: Example Statechart model

Assume, as in the example above, a behavioral model exists with m parallel, orthogonal, state models, each containing n transitions between contained states. Enumerating all possible combinations of transitions occurring during a single step would entail $m \times n$ unique combinations based on asynchronous Statechart semantics. For

sufficiently large models, utilizing asynchronous Statechart semantics can greatly reduce the number of trajectories that must be examined.

While asynchronous semantics can aid in reducing the number of trajectories to be enumerated, the problem exists of which type of semantics better reflects the natural system behavior. Each type of semantics has advantages, depending on the type of system being analyzed. In the case of high assurance, high consequence systems, modelers and analysts often approach problems with asynchronous assumptions in mind. For continuous and hybrid systems, the asynchronous assumptions are usually correct. For discrete systems, these assumptions are correct if the units of time are discretized into small enough slices. For discrete systems with large units of time, the synchronous semantics better reflect actual system operation.

Due to the method of representing the transition relation, only the lowest levels of OBDD algorithms must be modified to change from asynchronous to synchronous Statechart semantics. Only the functions that perform AND and OR operations between state configurations to the transition relation have to be modified. These operations now operate on a tree structure that represents the distributed next state relation. At each node in the tree, either an intersection between the current node and the state configuration of interest is computed, or the results of recursively operating on the node's children are collected. The type of node in the tree determines what operation is performed. Other, higher level, algorithms do not require any modification.

Fault Tree Size Reduction

For mid to large size behavioral specifications, the fault trees produced by the fault tree generation algorithm can grow too large to be imported by WinR. WinR has a

limitation of 4000 nodes in an individual fault tree [7]. Taking advantage of the structure of the algorithm, the "goal" state configuration chosen by the analyst can usually be dissected into several, more constrained, "goal" state configurations. For example, if the analyst leaves some parallel states in unspecified configurations, the fault tree generation algorithm analyzes all combinations of the unconstrained states. This was a requirement in order to guarantee completeness in the analysis – every possible combination must be examined. The overall fault tree generated is simply an "OR" gate with a child fault tree for every state configuration that meets the specified "goal" state configuration requirements. The ComponentFTG algorithm uses this knowledge in constructing a set of smaller, component fault trees.

```
Node ComponentFTG (bdd start, bdd goal) {
        Enumerate the possible goal state configurations (Goals)
        Node root = new Node (ORGATE)
        ∀ g ∈ Goals{
                FaultTreeGeneration (start, g,  root)
        }
        return root
}
```

Algorithm 8: Component fault tree generation

The improved fault tree generation algorithm takes this model structure into account when constructing fault trees. Instead of a monolithic fault tree, several fault trees, one for each state configuration that meets the specified "goal" state configuration are generated. These fault trees can then be loaded into WinR for further analysis. The

analyst must then combine the results of analyzing each component fault tree manually. While this task may require significant effort, without this technique, the fault trees that are produced cannot be analyzed using WinR, which has a size limit of 4000 nodes in an individual fault tree. Another possibility is to integrate other fault tree tools into the environment that do not have this limitation.

Symbolic Cut Set Determination

Fault trees are used for two primary types of analyses: qualitative and probabilistic. Qualitative analysis involves examining which component failures occur and how they affect the trajectory toward the failure condition. Probabilistic analysis involves computing probabilities of top events occurring given the probabilities of individual component failures occurring and the structure of the fault tree. Qualitative fault tree analysis prefers the fault tree structure to maintain direct links to the behavioral specifications. Qualitative fault trees are generated by Algorithm 6. Probabilistic fault tree analysis requires the "cut sets" of a fault tree be determined. A cut set of a fault tree is defined as the collection of basic events (component failures) such that if they all occur, the top event also occurs. Basic events contained in a cut set are all necessary, and each cut set is sufficient, for causing the top event to occur. Minimal cut sets are defined as the cut sets such that if any basic event is removed from the cut set, the top event will not occur [27]. A minimal cut set is the *smallest* combination of component failures such that if they all occur, the top event will occur.

Sinnamon and Andrews have developed a technique to determine the cut sets for a given fault tree using OBDDs [50]. Their technique involves traversing the existing fault tree and producing a unique OBDD encoding for each basic event in the tree. Then,

by following the structure of the fault tree and applying some basic identities, they reduce the structure of the fault tree into the cut sets. They do not guarantee the creation of minimal cut sets. Their technique for determining OBDD variable ordering depends on the order basic events are encountered in the original fault tree [50].

Since an internal fault tree is generated by the fault tree generation algorithm, Sinnamon and Andrews's technique can be applied to the automatically generated fault trees. However, a unique OBDD encoding for each basic event already exists. Instead of reconstructing an OBDD to represent the cut sets of the fault tree by applying the Sinnamon and Andrews algorithm, a new fault tree analysis algorithm has been developed. The algorithm developed below traverses the generated fault tree structure and produces an OBDD representing the cut sets of the given fault tree.

```
bdd CutSets()  {
        bool and = false
        bool basic = false
        bdd ret = ∅
        switch (type) {
                case ANDGATE :
                        ret = bdd_one
                case INTEREVENT :
                        and_op = true
                        break
                case ORGATE :
                        ret = bdd_zero
                        break
                case BASICEVENT :
                        basic = true
```

```
                ret = GetBDD
                // the above line retrieves the BDD representing the event
                break
        }
        if (!basic) {
                if (type == INTEREVENT) {
                        ret = GetChild.CutSets
                }
                else {
                        List children = GetChildren
                        ∀ c ∈ children {
                                bdd child = c.CutSets
                                if (and_op)
                                        ret = ret ∩ child
                                else
                                        ret = ret ∪ child
                        }
                }
        }
        return ret
}
```

Algorithm 9: Cut set generation

Algorithm 9 traverses the fault tree and applies the Boolean operations at each node. AND nodes produce a conjunction of their children OBDDs, while OR nodes produce a disjunction of their children OBDDs. BASICEVENT nodes return their corresponding OBDD, and INTEREVENT nodes return the resulting OBDD from their child. This algorithm applies the logic contained in the fault tree to the OBDD representation of the basic events. What eventually results is a single OBDD

representation of the fault tree. This representation is equivalent, mathematically, to the complete fault tree representation. Since all fault tree reduction is done using OBDDs, there is no need to use the identities and algorithms described in [50]. While this algorithm does not guarantee minimal cut sets, WinR can be used to determine the minimal cut sets from the minimized fault tree. The resulting fault tree is always a sum of product (SOP) representation of the cut sets, which simplifies fault tree construction. Algorithm 10 is used to convert the OBDD representation to a fault tree representation that can be analyzed by WinR.

```
Node ConvertFT (bdd ft) {
        Enumerate the possible event combinations that are represented by the
                fault tree bdd (bdds)
        Node root = new Node (ORGATE)
        ∀ b ∈ bdds {
                Node tmp = new Node (ANDGATE)
                tmp.AddChildren( GetEventsFromBDD(b) )
                root.AddChild (tmp)
        }
        return root
}
```

Algorithm 10: Convert fault tree

By determining the cut sets before exporting the fault tree to WinR, the impact of the WinR limitation on size of the fault tree can be reduced. In addition to allowing larger models to be analyzed, this algorithm reduces the computation time required by WinR. Since WinR uses many complex analysis routines for calculating probabilistic

values, using a smaller data set can increase overall analysis performance. Performance in computing the cut sets for the fault tree is on the order of $O(n)$ where n is the number of nodes in the complete fault tree. In the case study presented in Chapter VI, the original, full fault tree contained slightly over 40,000 nodes. This was over an order of magnitude too large for analysis with WinR. The cut set version of the fault tree contains 503 nodes, easily with WinR's analysis capability.

Cut Set Order Reduction

In some cases, analysts prefer to limit the order of the cut sets they examine. The order of a cut set is defined as the number of basic events that are members of the cut set. Limiting the order of the cut sets encompasses eliminating all cut sets that have an order larger than that specified by the analyst. The following algorithm describes the procedure utilized to ensure only cut sets of the order specified are output to the analysis program.

Algorithm 11 allows the analyst to ensure only cut sets of the specified limit or below are output to the analysis program. The analyst makes an assumption that the probability of occurrence of cut sets above a specified order are negligible. For large cut sets, the probability of all basic events occurring approaches some minimal value. This technique allows the analyst to focus on cut sets of smaller order, making the analysis task more manageable.

Preferably, the task of eliminating high order cut sets could be undertaken in the fault tree generation process. However, stopping the fault tree generation process after a specified number of basic events occur in a sub tree does not guarantee that all possible trajectories are explored. High assurance, high consequence systems must have a

complete analysis performed, so eliminating high order cut sets during the fault tree generation process is not a feasible solution.

```
RemoveHighOrderCutSets(Node ft, int order) {
        // fault tree will be in SOP format
        List remove = ∅
        if (ft.numChildren > 0) {
                ∀ ch ∈ ft.GetChildren
                {
                        if (ch.numChildren() > order) {
                                remove.append(ch)
                }
        }
        ∀ r ∈ remove
        {
                ft.RemoveChild(r)
        }
        if (ft.numChildren() == 0) {
                ft.SetName("Empty FT");
        }
}
```

Algorithm 11: Remove high order cut sets

Design Space Pruning

Design space pruning refers to eliminating non-essential states and events from the symbolic representation of the models prior to performing safety, reliability, or diagnostics analyses. Practice has shown that large sets of models have two potential problems. First, the fault trees that are generated may be too large for incorporation into

WinR. Other fault tree analysis tools may alleviate this problem, but any tool with a restriction on the number of nodes possible in a fault tree will introduce some limit on fault tree size. Secondly, large models may take extremely long periods of time, and use large amounts of memory, to analyze. Experience has shown that whenever an OBDD must be swapped to virtual memory performance suffers greatly. This swapping characteristic only exacerbates the computational complexity problem of analyzing large models. Reliability analysis naturally involves the enumeration of all trajectories leading to a reliability failure. This enumeration can often require large amounts of memory and computation resources.

By restricting the models to only those states that affect the behavior between the initial and goal states, the analysis complexity can be reduced. Instead of the analysis routines enumerating all possible trajectories, only those trajectories that directly impact the selected state configurations are enumerated. Examination of all trajectories often leads to many cases of "don't care" states and events being included in the analysis data set. In addition to increasing the computational performance of an analysis and decreasing the size of the fault trees to be further analyzed, the method of pruning the design space increases the productivity of the analyst by removing unnecessary trajectories from the generated fault trees.

Two separate methods for design space pruning were developed. While each is described below, the structural pruning algorithm shows the most promise. Each algorithm will aid in reducing the design space, and thus the fault tree sizes, but the structural pruning algorithm more aggressively prunes the design space. In addition, the structural pruning algorithm is efficient in that the size of the OBDD representations do

not affect the efficiency of the algorithm. The reachability based pruning algorithm uses the full OBDD representation of the system to determine the set of events and states to "prune".

Both the reachability and structural pruning algorithms work best on weakly inter-related models. Only models that have a significant number of primarily independent subassemblies will benefit from design space pruning. Models that are tightly coupled will not receive a sufficient improvement in analysis complexity.

Reachability Based Design Space Pruning

Reachability based design space pruning utilized the reachability algorithms described previously to determine which states and events can be eliminated from the next state relation before performing further analysis. Utilizing the existing algorithms ensures completeness. All possible combinations of states and events are analyzed to determine which are applicable to the current analysis. The initial and goal state configurations, which must be selected for fault tree generation, are used to determine which states and events can be eliminated from the next state relation. The next state relation is not modified; instead a working copy of the relation is created and modified. In order to keep consistency between the analysis algorithms and the modeled specifications, the simulation and reachability algorithms must use the complete next state relation.

Algorithm 12 automatically removes unreachable states from the transition relation. The algorithm returns a modified, distributed next state relation in the distributed tree format. By removing states that cannot be reached, under any circumstances, the fault tree generation algorithm is guaranteed to examine all possible

trajectories between the initial and goal state configurations without analyzing unimportant events and states. By only eliminating state configurations that can never be reached from the transition relation, the fault tree generation algorithm will still explore all combinations of states that may, in any way, impact the trajectories between the initial and the goal state configuration. This feature is a must since the analysis of high consequence, high assurance systems must be complete.

```
NSETree ReachabilityPruning ( bdd start, bdd goal ) {
        bdd reach = FixedPointReachability(start)
        List states = GetListOfAllStates();
        List remove = ∅
        ∀s ∈ states {
                if start ∩ s = ∅
                        remove.Append(s)
        }
        NSETree ret = nse
        ∀s ∈ remove {
                ret = ret | s→x
        }
        ∀s ∈ remove {
                s' = s | s'0↔s 0, ... , s'n-1↔ s n-1
                ret = ret | s'→x
        }
        return ret
}
```

Algorithm 12: Reachability based design space pruning

Reachability based design space pruning has some undesirable features. Only unreachable states are eliminated from the design space. Many state configurations are still examined, while enumerating the possible trajectories from the initial state configuration to the goal state configuration, that do not affect the trajectory information. State configurations exist that can be reached, but that are truly "don't care" states. The fault tree analysis in no way depends on the presence of these state configurations. Fault tree analysis performed following reachability based design pruning incorporates state configurations into the trajectory information that does not impact the trajectories in any way.

One of the advantages of using reachability based design space pruning is the speed of the operation. Determining the reachability set is not impacted by the state space explosion problem. In practice, the gains achieved through the use of reachability design space pruning are minimal. Only models that contain a large number of unreachable state configurations benefit from the reachability based pruning algorithm. While this algorithm can aid in providing fault tree information for some models, in general, the advantages of reachability design space pruning before performing fault tree generation is minimal.

Structural Design Space Pruning

Another method for design space pruning was necessary in light of the limitations presented by the reachability based design space pruning. In addition to eliminating unreachable state configurations from the transition relation, this new technique uses information from the behavioral models themselves to remove "don't care" states from the transition relation. Not only are unreachable state configurations not used in

calculating fault trees but the set of states that does not impact any of the trajectories between the initial and the goal state configurations are removed from the analysis. Algorithm 13 is the structural design space pruning method.

```
NSETree StructuralPruning ( bdd start, bdd goal ) {
        List examine = States(goal)
        List states = ∅
        List events = ∅
        while (examine ≠ ∅) {
                List next = ∅
                ∀s ∈ examine {
                        states.Append(s)
                        List trans = GetInTransitions(s)
                        ∀t ∈ trans {
                                List pstates = GetPStates(t)
                                ∀s ∈ pstates {
                                        if s ∉ states
                                                next.Append(s)
                                }
                                List gstates = GetGStates(t)
                                ∀s ∈ gstates {
                                        if s ∉ states
                                                next.Append(s)
                                }
                                List triggers = GetTEvents(t)
                                ∀e ∈ triggers {
                                        events.Append(e)
                                }
                        }
```

```
            }
            examine = next;
        }
        NSETree ret = nse
        ∀s ∈ ListOfAllStates {
            if s ∉ states
                ret = ret | s→x
        }
        ∀s ∈ ListOfAllStates {
            if s ∉ states {
                s' = s | s'0↔s 0, ... , s'n-1↔ s n-1
                ret = ret | s'→x
            }
        }
        ∀e ∈ ListOfAllEvents {
            if e ∉ events
                ret = ret | e→x
        }
        return ret
}
```

Algorithm 13: Structural design space pruning

In addition to providing all of the benefits of the reachability based design space pruning, the structural design space pruning removes all possible "don't care" state configurations from the transition relation prior to performing fault tree analysis. Structural design space pruning can also remove all unnecessary events from the transition relation before generating fault trees. By eliminating not only the unreachable state configurations, but also the "don't care" state configurations, the structural design

space pruning algorithm can guarantee that the trajectory information generated by the fault tree analysis algorithm contains only pertinent information.

The overhead associated with performing the pruning algorithm is easily overcome by the computational complexity gains in performing the fault tree analysis. Figure 15 and Figure 16 show the case study example state spaces for the full behavioral specification and for the pruned behavioral specification. In this example, taken from the case study in Chapter IV, the top event was defined as both front brakes failing.

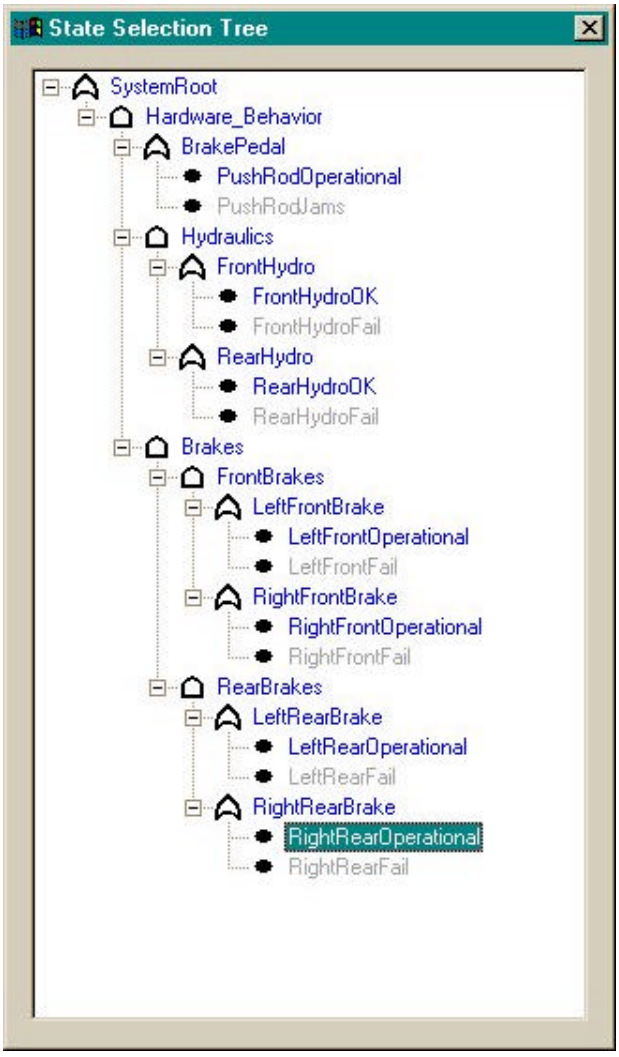

Figure 15: The brake example -- full design space

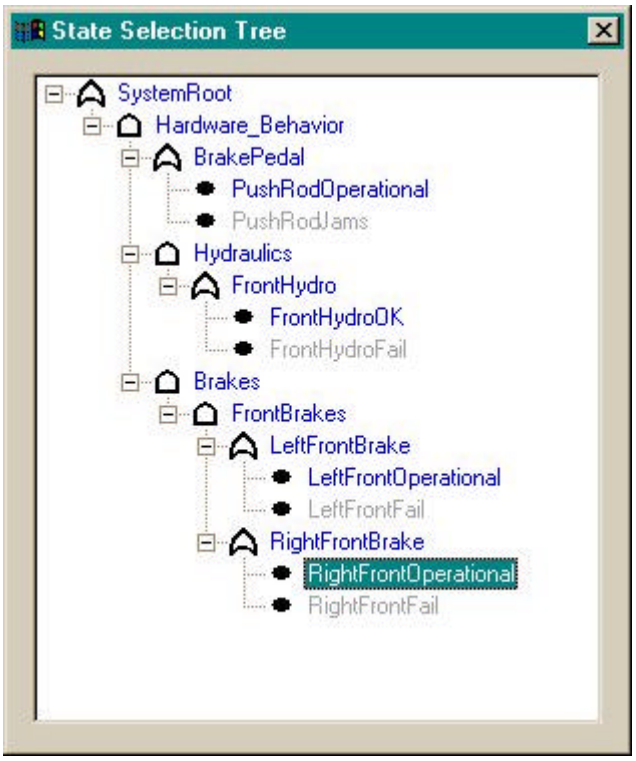

Figure 16: The brake example – pruned design space

In the case of the pruned design space, the complete state space of the system encompasses 16 unique state configurations. The full design space is comprised of 128 unique state configurations. By eliminating the unnecessary configurations, the information used to determine the possible trajectories in the design space only focuses on the necessary state configurations. Also, the pruned version must examine 10 events whereas the full version must examine 19 events. Overall, pruning reduces the total design space from 2^{26} to 2^{14}. It is very important to note that while the design space examined was extremely limited with respect to the full design space, the fault tree information derived from the pruned design space contains *all* information about possible trajectories between the initial and the goal state configurations.

Pruning Results

For the case study example, the full fault tree when the design space was pruned using the structural pruning methodology was 2603 nodes. The cut set fault tree for the pruned case was 115 nodes. For the non-pruned case, the full fault tree contained 40,138 nodes and the cut set fault tree was comprised of 503 nodes. There is a dramatic decrease in the size of the fault trees generated from the pruned design space. For the case study example the size reduction in the generated fault trees was approximately a factor of five. Only the example case of the front brakes failing is discussed, but the results are similar for cases where only the rear brakes fail, only the left brakes fail, and only the right brakes fail. Unfortunately, the case when all four brakes fail does not result in a size reduction of the generated fault trees. In this case the pruning algorithm does not prune the design space at all.

While the demonstrated level of improvement cannot be generalized or guaranteed, for many cases that involve loosely coupled, orthogonal behavioral specifications, the design space pruning algorithm greatly reduces the size of the design space that must be analyzed. The size improvement of the generated fault trees is a function of the modeled behavioral specification, the initial state configuration chosen, and the selected top event. For tightly coupled models, the model based design space pruning does not offer significant size savings. Unfortunately, without knowledgeable user intervention, tightly coupled behavioral specifications will still produce large fault trees.

The analysis routines described here have been implemented into an integrated modeling and analysis toolset. Chapter V describes the toolkit and details how to use the tools for performing model verification and analysis. Chapter VI describes a case study

where the tools have been applied. Lastly, Chapter VII details the results of this research and makes suggestions for future work to enhance the research results.

CHAPTER V

TOOLKIT FOR SAFETY, RELIABILITY, AND DIAGNOSTICS

Integrated Analysis Toolset

The integrated modeling and analysis environment, the Model Integrated Surety Analysis System (MISAS), leverages current technology for safety, reliability, and diagnostic modeling and analysis. MISAS consists of the domain specific modeling environment, which was described in Chapter III, implemented using the MultiGraph Architecture (MGA, see Appendix A). This modeling environment interfaces to the integrated analysis environment, described in Chapter IV, which allows model validation and verification. MISAS utilizes the previously denoted analysis techniques for model verification; it maps the necessary information from the integrated models into the individual analysis tools. Figure 17 shows the MISAS system architecture. A user can employ the Graphical Model Editor (GME) to construct system models in the domain specific paradigm. The user can then use the State Space Analysis Tool (SSAT) to validate/verify the models and perform diagnostic, safety, and reliability analyses and provide data for other, domain specific, analysis tools.

Semantic Integration

The first problem the IAE must address is mapping the *semantics* of the integrated models into SSAT. MISAS must be able to convert the integrated GME models into symbolic models while preserving the semantics of the behavioral models. SSAT must then be able to convert from the symbolic models into other formats while again

preserving the semantics of the models. This was accomplished using the techniques described in Chapter IV [46][48].

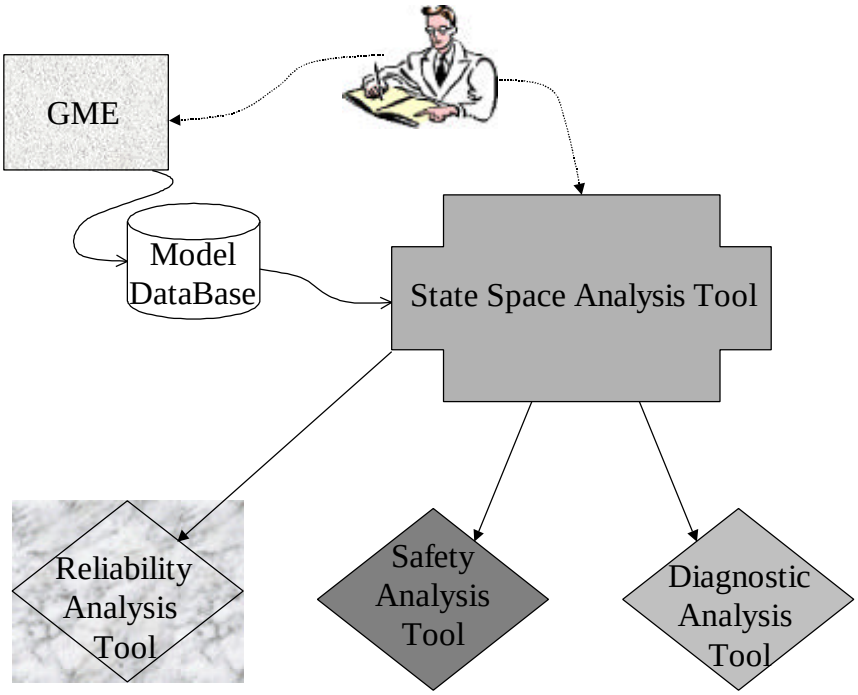

Figure 17: MISAS Architecture

State Space Analysis Tool

The State Space Analysis Tool (SSAT) serves as the execution environment for MISAS. The SSAT architecture is shown in Figure 18. SSAT can interface with both the MISAS model editor and the SSAT user interface (UI). These interfaces allow the building and examination of the internal FSM system representation. The internal FSM representation interfaces to the OBDD models and analysis routines. These routines were constructed using an existing low-level OBDD package from CMU. SSAT implements the following model validation routines:

- forward simulation,
- backward simulation,
- forward reachability analysis,
- backward reachability analysis,
- loop detection analysis,
- automatic fault tree generation,
- cut set fault tree generation,
- design space pruning,
- and deterministic analysis.

These routines allow the system to user to validate and verify the system models. The UI allows access to all of the analysis routines and displays the FSM models in a compact visual representation. SSAT also serves as an integration tool for existing diagnostic, safety, and reliability tools. For example, SSAT allows the system user to generate reliability information that can be used in external fault tree analysis tools.

Due to the fact that different classes of problems are better represented by either asynchronous or synchronous Statechart semantics, the SSAT supports both asynchronous and synchronous semantics. This allows the system analyst to choose the appropriate semantics for a given problem domain. Care must be taken to ensure the modeler and the analyst use the same semantics when building and analyzing a set of models. Inconsistencies could arise if the modeler and the analyst use different semantics for performing their part of the system modeling and analysis.

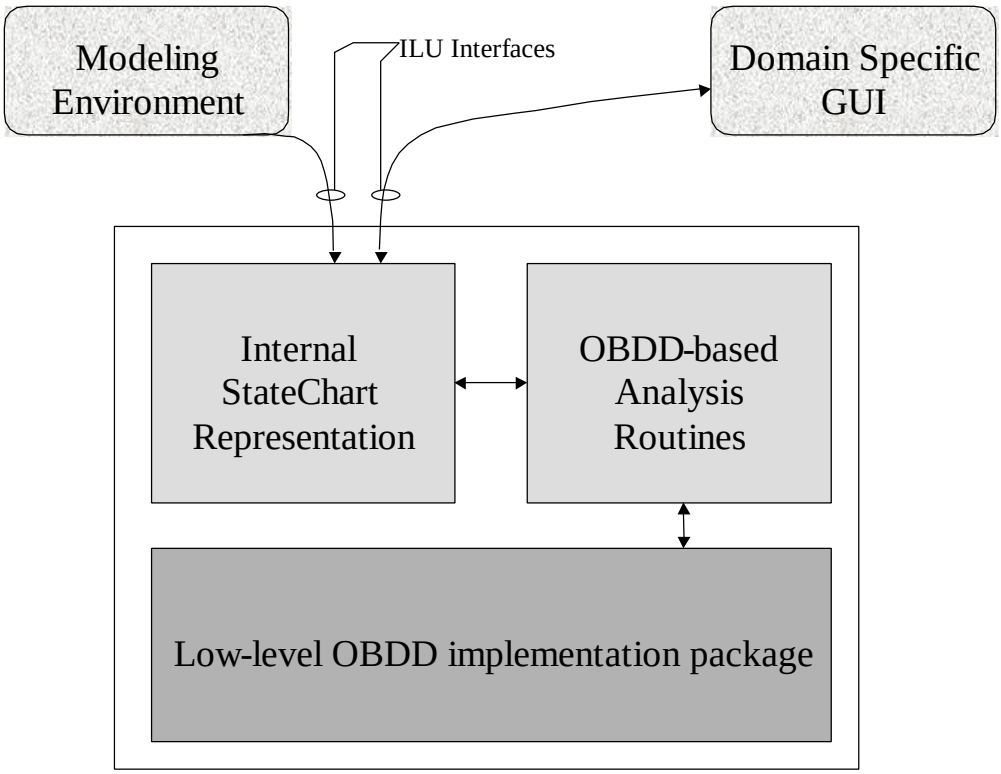

Figure 18: The SSAT Architecture

WinR Integration

WinR is a fault tree analysis tool developed by Sandia National Laboratories [7]. WinR has fault tree optimization utilities and fault tree reduction capabilities. Since FTA can be utilized for safety, diagnostic, and reliability analysis, WinR can be used as a quantitative and qualitative safety, diagnostic, and reliability analysis tool. SSAT has had the fault tree generation (FTG) algorithms from Chapter IV implemented. These allow fault trees to be generated from the integrated set of models. SSAT and WinR eliminate the problem of manual fault tree construction. MISAS provides many useful model analyses, but a key feature is the ability to apply the technology to perform safety, reliability, and diagnostic analyses. This rest of this chapter details using the SSAT for performing higher level (i.e. safety, reliability, and diagnostics) analyses. Without a

strong link to safety, reliability, and diagnostics, MISAS only allows complex analysis of behavioral and physical specifications.

Safety

The primary purpose of a safety analysis is to determine if, and under what conditions, a system safety failure may occur. The component failures that may lead to a system safety failure must be identified, and all necessary and sufficient combinations of component failures occurring may need to be discovered. The concept of necessary, immediate, and sufficient conditions limits the analysis to those sets of component failures that can, and will, directly lead to a system safety failure [1] [27].

Qualitative Safety Analysis

Qualitative safety analysis involves determining if a potential safety hazard can manifest itself in the system. In cases where the system is modeled with behavioral specifications, this analysis becomes determining if a state configuration, that represents the safety hazard, can ever be reached. If the safety critical state configuration can be reached under any condition, the qualitative safety analysis must discover this. This analysis maps directly to the reachability analysis routines described in Chapter IV.

The system analyst must specify both the "starting" state configuration and the hazard state configuration in order to perform a qualitative safety analysis. SSAT provides algorithms that only allow "valid" state configurations to be selected. If the "starting" state configuration is not fully specified, the modeled default states are used to represent the "starting" state configuration. At this point, the reachability algorithm is used to determine if any trajectory between the "starting" and the hazard state

configurations exist. The analyst is then notified with a text message as to the existence of a trajectory between the two state configurations [48].

After detecting a potential safety hazard that can occur in the target system, the analyst must then begin to determine how the safety hazard can occur. Another safety analysis technique is to determine the set of all the component failures that can occur along any trajectory leading to the safety hazard. Results from this analysis will alert the analyst to the possible causes of the safety hazard, but not the specifics of how and in what combinations the component failures must occur. The following algorithm is a modified reachability algorithm that determines a collection of all events that occur along any trajectory between selected state configurations [27].

```
bdd Safety ( bdd start, bdd goal )
{
        bdd result = ∅
        I1 = goal
        I2 = BackwardStep ( I1 )
        while ( I2 ≠ I1 )
        {
                result = result ∪ Faults(I2)
                I1 = I2
                I2 = BackwardStep ( I1 ) ∪ I2
                if ( I1 ∩ start ≠ ∅) return result
        }
        return ∅
}
```

Algorithm 14: Safety algorithm

In many cases, the knowledge of the combinations of faults that must occur is as important as which faults lead to a safety hazard. While the above algorithm can provide valuable information, in many cases information about specific trajectories are more valuable to the analyst. In these cases, the analyst can utilize the automatic fault tree generation techniques discussed in Chapter IV to generate a fault tree. The automatic fault tree generation algorithm will produce a detailed list of every fault that must occur, and in what combinations, to introduce a trajectory from the "start" to the hazard state configuration. These combinations are both sufficient and necessary for the system to reach the hazard state configuration. Every possible trajectory between the two state configurations will be enumerated. Using WinR, the analyst can then examine these trajectories. Figure 19 shows an example fault tree. In this fault tree, some nodes have comments such as "State Configuration #133". A generated mapping file maps these comments to actual state configurations in the behavioral specification. In this case, the analyst could open the mapping file and, by indexing "State Configuration #133", determine exactly what state configuration of the models (by state name) this fault tree node represents. This capability allows the analyst to generate fault trees from the models and still maintain a link to the behavioral specification. From these fault trees, the safety analyst can decide what system modifications are necessary to eliminate the possibility of the safety failure occurring.

Reliability

The purpose of a reliability analysis is to provide probabilistic failure occurrence properties. Component failures that may lead to a system failure must be identified, and the probability of all necessary and sufficient combinations of component failures

occurring must be calculated. Unlike safety analysis, reliability analysis is often concerned with producing probabilistic values for a system failure occurring. Reliability analysis is used to demonstrate that the probability of a system reliability failure occurring is below some specified critical threshold value [27].

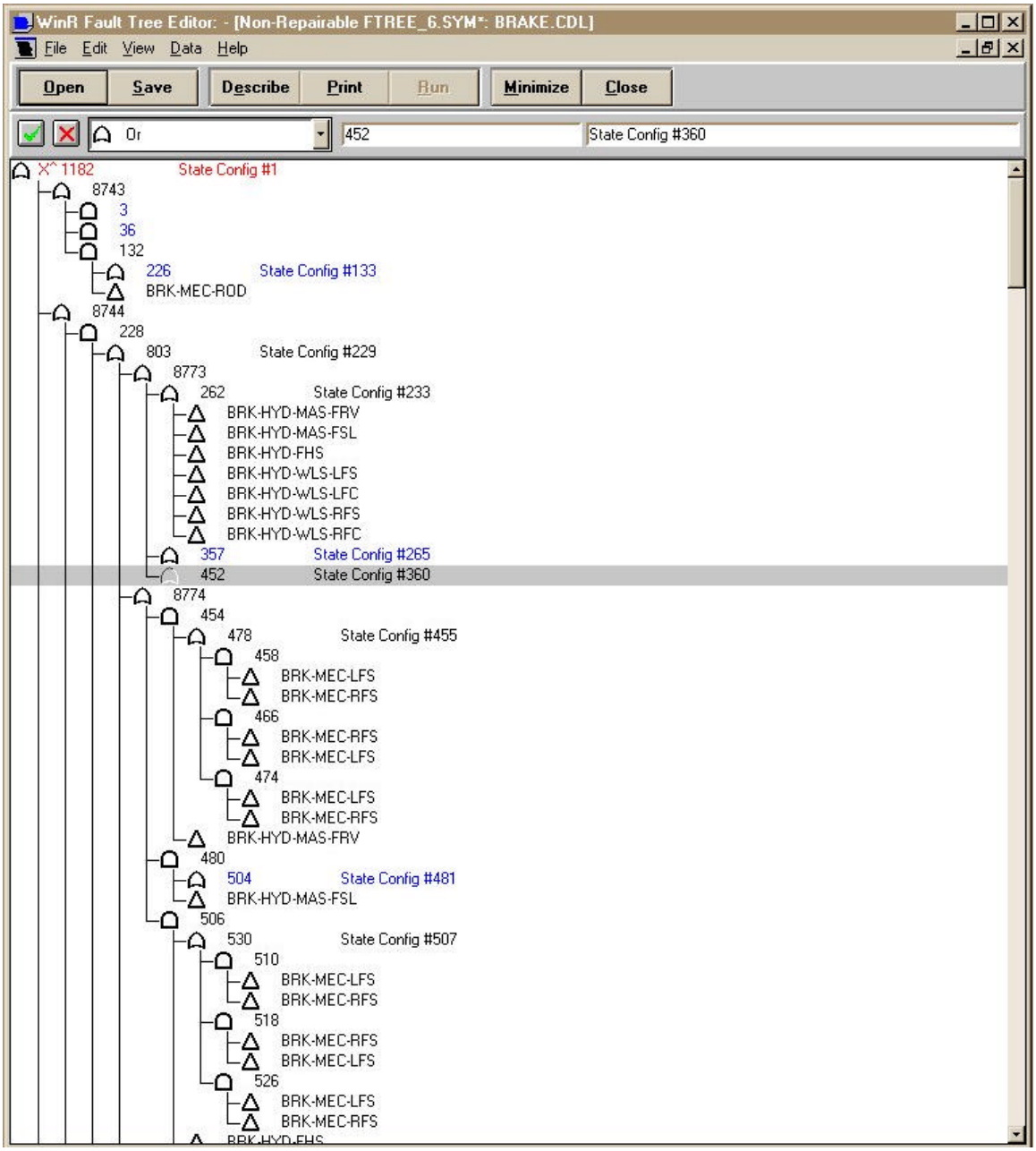

Figure 19: Example fault tree

Qualitative Reliability Analysis

Qualitative reliability analysis is used to determine if a possible trajectory between two state configurations exists. For performing this analysis, the standard reachability analysis algorithm described in Chapter IV is used. For more detailed qualitative analysis, those events that may occur along a trajectory need to be identified. The algorithm for qualitative safety analysis listed above is used for providing qualitative reliability analysis. Reliability analyses generally rely more heavily on probabilistic analysis than qualitative analysis [1].

Probabilistic Reliability Analysis

Probabilistic reliability analysis relies on enumerating which faults and fault combinations can lead to a system failure. These faults are identified based on conditions of being necessary and sufficient to cause the failure to manifest in the system design. The process of enumerating all possible trajectories that can lead to a reliability failure is accomplished using the automated fault tree generation algorithm described in Chapter IV. Since the main objective of reliability analysis does not involve trying to design "fail-safes" to eliminate the failure, cut set fault trees are generally more useful in reliability analysis than in safety analysis [27].

Cut set fault trees represent the same failure probability information as "full" fault trees. However, the cut set fault tree already has been reduced to the cut sets (not necessarily the minimal cut sets) of the corresponding fault tree. This enables a smaller set of data to be exported to the fault tree analysis tool. Cut set fault trees are determined using the cut set reduction algorithm listed in Chapter IV. Benefits include not only having smaller data sets to export, but the smaller data sets require less computation by

the analysis program. Providing cut sets for the generated fault trees allows the analyst to calculate reliability values for a given set of models in a shorter amount of time, while maintaining the same result. Figure 20 illustrates an example cut set fault tree.

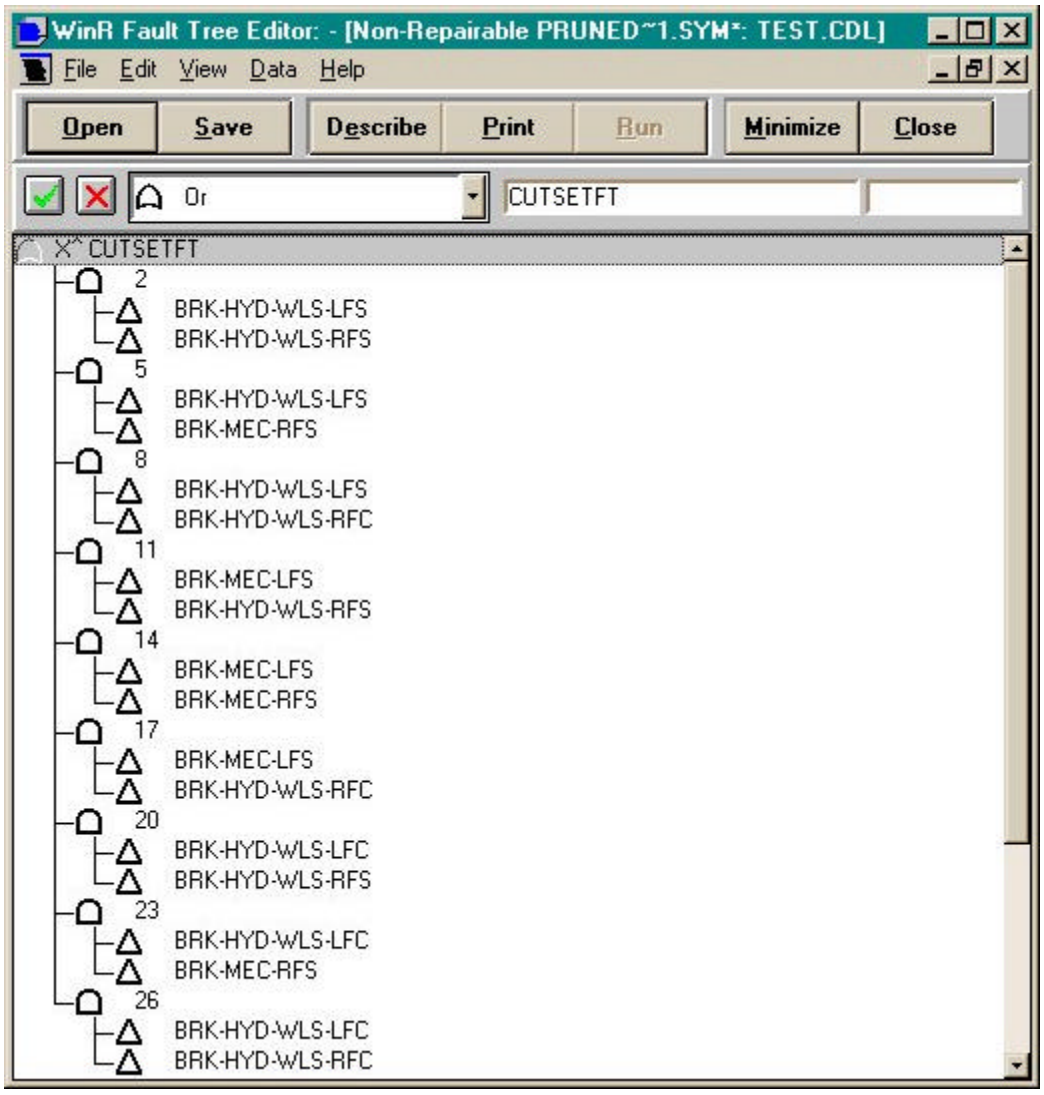

Figure 20: Example cut set fault tree

Diagnostics

A diagnostic analysis is used to determine under what conditions an observed system behavior has occurred. Component failures that lead to an observed system behavior must be identified. Unlike safety analysis and reliability analysis, diagnostic analysis is concerned with why an observed behavior occurred instead of being concerned with failures occurring in the future [35].

Qualitative Diagnostic Analysis

Qualitative diagnostics involves determining which component or subassembly failures could have caused the observed system behavior. It is known that the behavior can occur, since it has been observed. Due to this, the standard reachability algorithm does not apply to diagnostic analysis. Instead, Algorithm 15 is used to determine all possible events that could have occurred in reaching the sate configuration representing the observed system behavior. This algorithm requires the observed behavior to be mapped to a state configuration that is legal in the modeled behavioral specification [9]. The algorithm uses the modeled starting state as the starting state for the system behavior.

```
bdd Diagnostics ( bdd failure )
{
        bdd result = ∅
        I1 = failure
        I2 = BackwardStep ( I1 )
        while ( I2 ≠ I1 )
        {
                result = result ∪ Events(I2)
```

```
            I1 = I2
            I2 = BackwardStep ( I1 ) ∪ I2
            if ( I1 ∩ default ≠ ∅) return result
        }
        return ∅
}
```

Algorithm 15: Diagnostics

Using the above algorithm, the diagnostic analyst can determine a complete set of events that could have caused the observed behavior. However, the analyst cannot determine exactly which trajectory the system followed, so he cannot limit the set of events to those that are necessary and sufficient to manifest the observed behavior. The fault tree generation algorithm can provide detailed trajectory information to the analyst. By enumerating all possible trajectories, and the events that occur along them, the analyst can compare the trajectory information against actual observations to determine exactly how the system entered into the observed behavioral situation.

Differences of Safety, Reliability, and Diagnostics Analysis

While safety, reliability, and diagnostic analyses can all be performed using similar analysis techniques, there are major differences in how to apply the analysis techniques. Both safety and reliability are design-time issues. Analysts will qualify the safety and reliability of a system before system deployment. Diagnostic analyses are performed on fielded systems in response to system observations. The diagnostic information is used after a system failure occurred to determine why the system failed.

Safety and reliability analyses are used to determine the probability of a system failure or to verify the existence of a specific system failure.

While both safety and reliability use similar analysis techniques, they are very different analyses. Reliability focuses on the ability of a system to perform its desired function. Safety focuses on the ability of the system to remain "safe". In a nuclear power plant, for example, the reliability engineers wish to keep the plant producing energy. Safety engineers wish to shut the plant down at the first sign of potential safety problems. In the area of high consequence, high assurance systems, the safety engineers' concerns are of paramount importance. Safety usually takes priority in any high consequence, high assurance system.

CHAPTER VI

CASE STUDY

The tools described in Chapter V have been applied to a simple automotive braking system. A physical description of the system along with details as to how it is modeled using the integrated modeling paradigm is presented. The different analyses performed are then discussed.

Automotive Braking System

Figure 21 illustrates the physical construction of an automotive braking system. The brake pedal connects to the master cylinders through the brake rod. Each master cylinder operates autonomously. Only the brake rod interacts with both the front and the rear master cylinders. A fault condition in one master cylinder can only affect the other master cylinder through the brake rod. Each master cylinder is connected through a separate set of brake lines to the hydraulic system at each wheel – the front master cylinder is connected to both front wheels and the rear master cylinder is connected to both rear wheels. Each wheel's hydraulic system is connected to the brake shoes. By applying hydraulic pressure, the hydraulic system compresses the brake shoes onto the brake rotor – thus creating the friction that stops the car.

For this example, the number of failure trajectories is quite large compared to the size of the system's behavioral models. The behavioral models represent a state space of 128 discrete states. The fault tree generation algorithm must examine over 3 million

distinct trajectories for this small example. Even for this limited example, the number of failure trajectories would be difficult to discover manually.

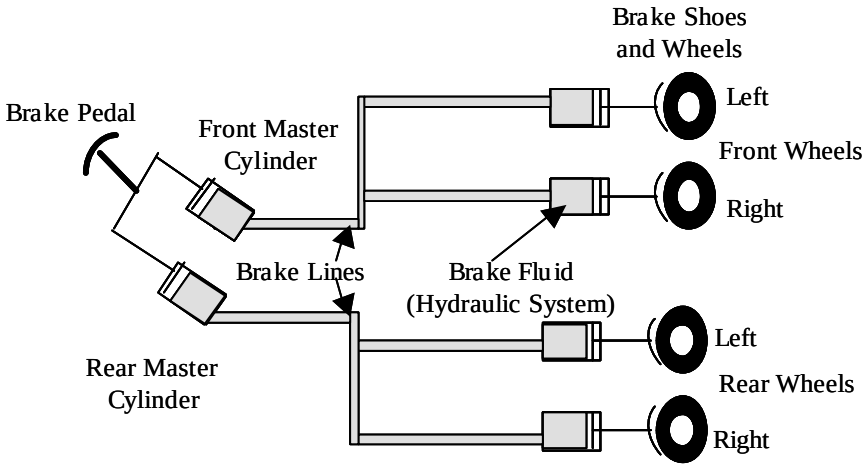

Figure 21: Braking system physical model

Figure 22 shows a piece of the behavioral specification for the brake system. This is the behavioral model for the front hydraulic system. The states represent the discrete modes of operation for each component. The modeled events map to system and/or component failures. The component failures that are considered are listed in Table 1. Table 1 also contains a description of each failure mode and the probability of the failure occurring. The complete finite state machine model for the system is not shown due to the size of the model. Each hardware component has its behavior modeled as a separate state machine. The hardware components are modeled as having two states: operational and failed. Different sets of discrete failures can trigger the transitions between states. A hardware failure can lead to other failures in the system. For example, if the front brake line ruptures, the front brake cylinders will eventually become non-operational. Then the front brake shoes cannot contact the brake rotors, so the front brakes have failed.

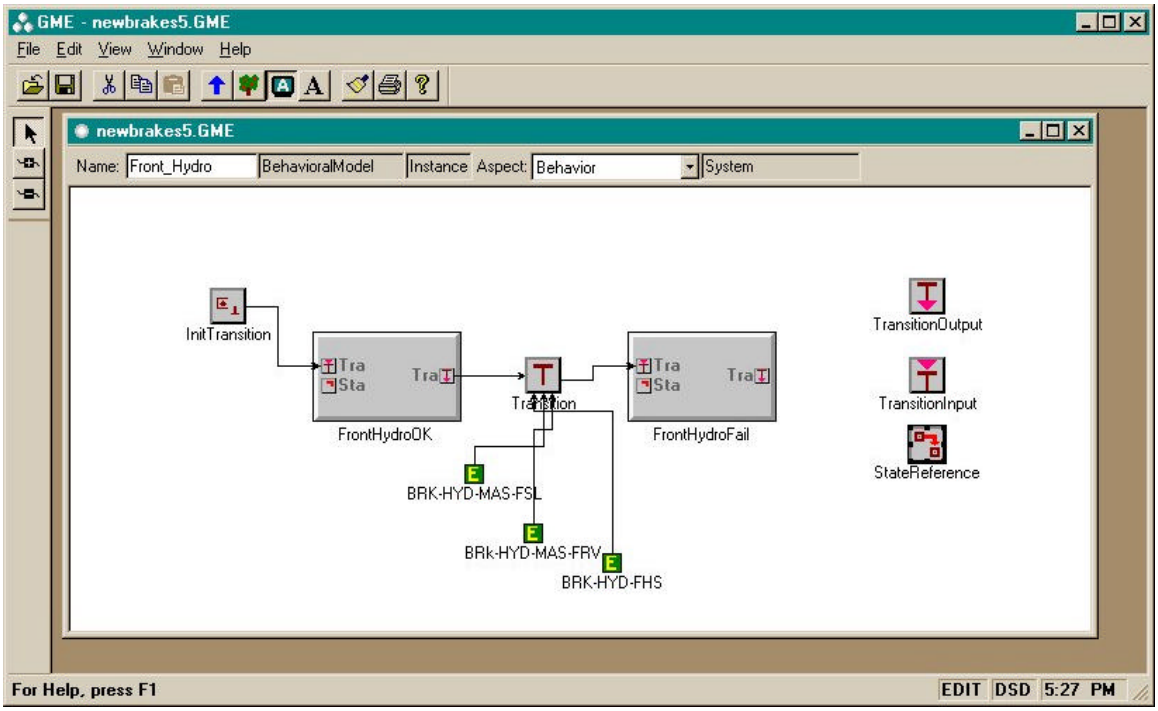

Figure 22: Example braking system GME model

Additional fault modes were discussed in the WinR documentation [7] for the automotive braking system. The set of component faults modeled was limited in order to decrease the scope of the example. Fault modes were chosen as a representative set to illustrate the use of the discussed analysis techniques. The automotive braking example was chosen because the domain is well known and understood. Details of the automotive braking system modeled were discovered from the WinR documentation [7] and from structural and behavioral diagrams provided by Sandia National Laboratories.

Table 1: Brake example failure modes

Failure ID	Failure Mode	Failure Probability
BRK-HYD-FHS	Front hose breaks	0.003
BRK-HYD-MAS-FRV	Front reservoir ruptures	0.0004
BRK-HYD-MAS-FSL	Front seals fail	0.0004
BRK-HYD-MAS-RRV	Rear reservoir ruptures	0.0004
BRK-HYD-MAS-RSL	Rear seals fail	0.0004
BRK-HYD-RHS	Rear hose breaks	0.003
BRK-HYD-WLS-LFC	LF chamber ruptures	0.0002
BRK-HYD-WLS-LFS	LF seals fail	0.002
BRK-HYD-WLS-LRC	LR chamber ruptures	0.0002
BRK-HYD-WLS-LRS	LR seals fail	0.002
BRK-HYD-WLS-RFC	RF chamber ruptures	0.0002
BRK-HYD-WLS-RFS	RF seals fail	0.002
BRK-HYD-WLS-RRC	RR chamber ruptures	0.0002
BRK-HYD-WLS-RRS	RF seals fail	0.002
BRK-MEC-LFS	LF brake shoes	0.002
BRK-MEC-LRS	LR brake shoes	0.002
BRK-MEC-RFS	RF brake shoes	0.002
BRK-MEC-RRS	RR brake shoes	0.002
BRK-MEC-ROD	Brake rod jams	0.002

SSAT performs the mapping from behavioral specifications to a symbolic representation. The modeling environment and the user interface (UI) use SSAT to perform all analysis operations. The modeling environment uses SSAT to construct a domain independent representation of the behavioral specification. After SSAT

transforms this set of models into symbolic models, the UI allows the user to perform analyses on the relational models.

Figure 23 illustrates the SSAT Graphical User Interface (UI). From this interface, the analyst can perform all primary analyses on the behavioral specifications. The user can use the State Selection Tree window to provide state configuration information to the server. An algorithm that only allows valid state configurations to be entered was developed to ensure only valid state configuration data was passed to SSAT. The State Result Tree window is used to show results from simulation (i.e. traversal) analysis. The tree structure that grows in the main window represents the distinct trajectories that have been traversed. In the example case, there is only one trajectory which travels through three distinct state configurations.

In addition to providing state configuration input from the user, the UI also allows the user to perform deterministic determination and loop analysis using the algorithms specified in Chapter IV. By selecting the appropriate commands from the *Validation* menu, the system analyst can perform model validation routines before further analyzing the behavioral specifications. This feature allows the analyst to ensure only valid, verified models are used to generate safety, reliability, or diagnostic data.

Reliability Analysis

Figure 24 shows an example reliability fault tree that was generated from the brake example models. In this case, the initial state configuration was the modeled default state and the goal state configuration (or top event) was the state where one or more brakes failed. By selecting different initial and goal state configurations, the analyst can generate different fault trees representing the different scenarios.

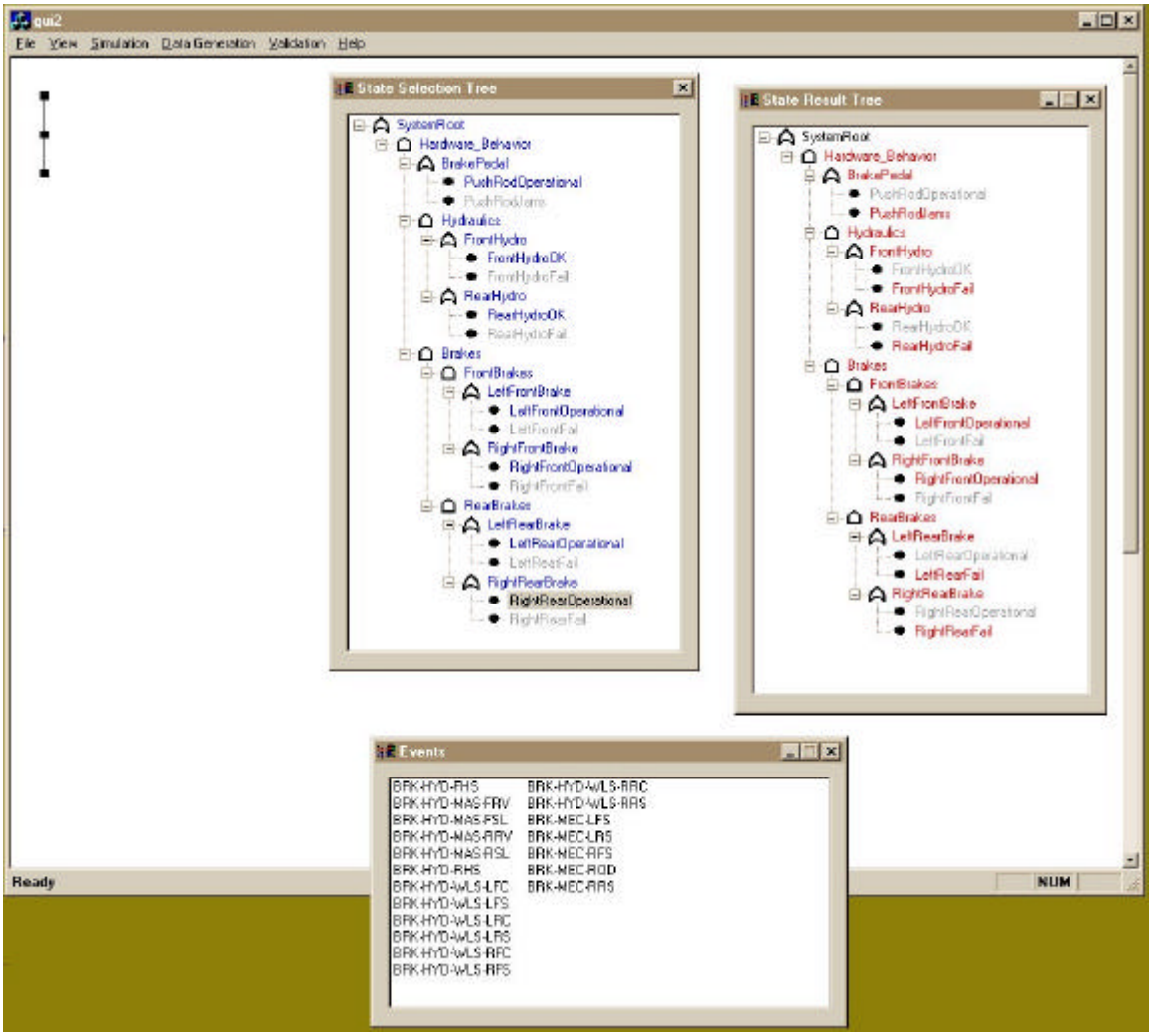

Figure 23: The SSAT User Interface

Figure 25 is an example cut set fault tree that was generated. In this case, the initial state configuration was the modeled default state and the goal state configuration (or top event) was the state where one or more brakes failed.

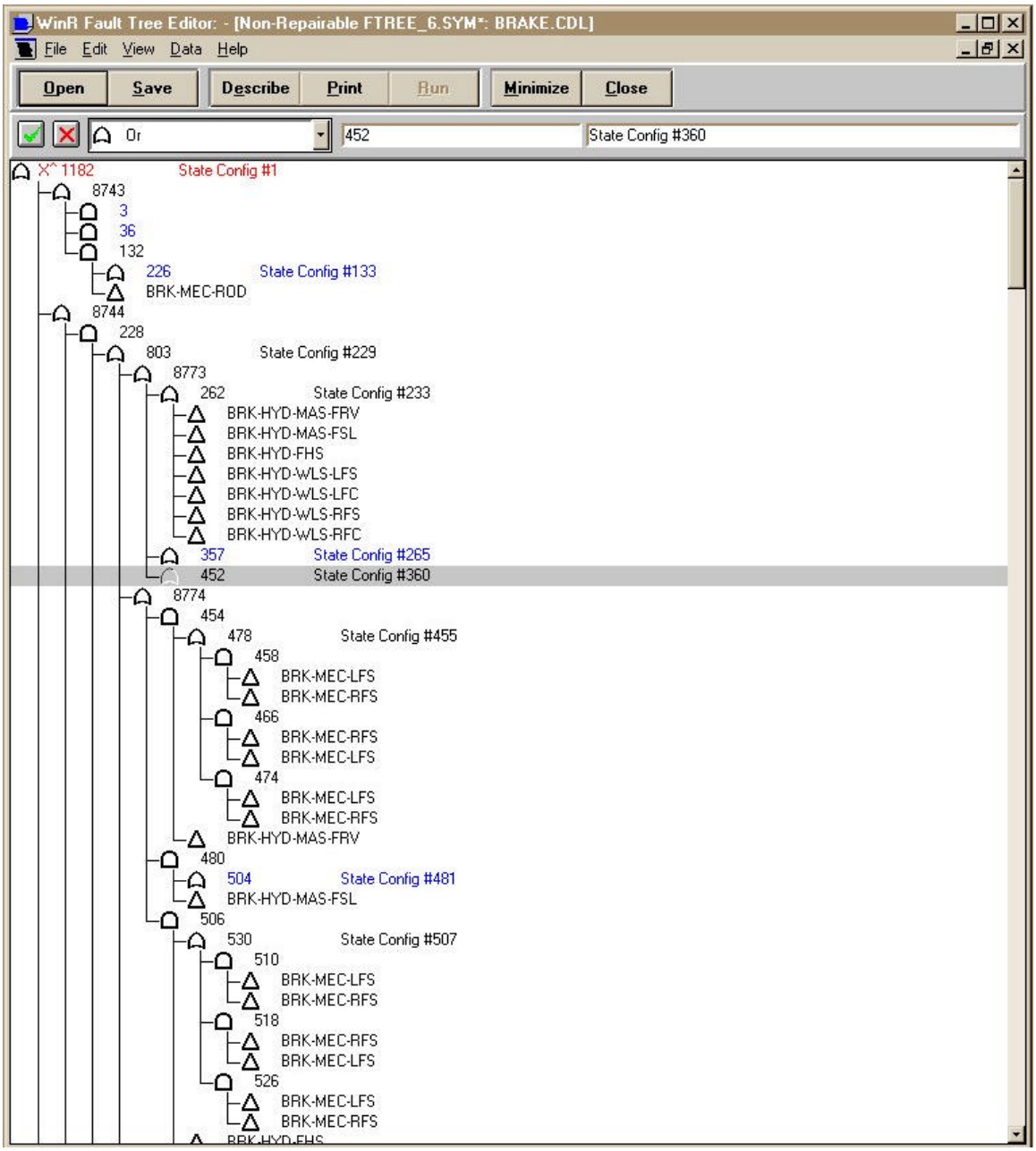

Figure 24: Generated fault tree

Safety Analysis

Now that the reliability of the system can be verified, the safety engineers need to be able to perform analysis using the same set of models. In this case, assume a safety failure occurs when either both front brakes or both back brakes fail. If only one front or

one back brake fails, the car can still safely stop. By selecting the corresponding state configurations in the UI, the analyst can perform a safety analysis and determine that a possible path to the safety failure condition does exist.

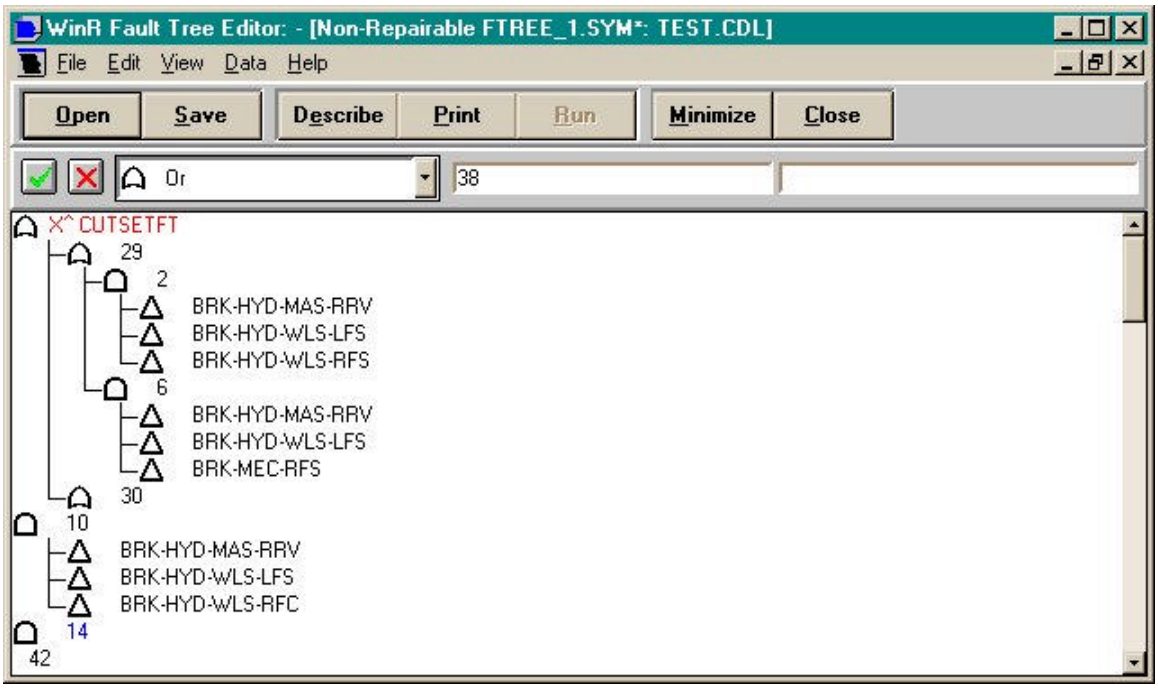

Figure 25: Generated cut set fault tree

Next, the safety engineer may apply the safety analysis algorithm to determine which faults may trigger the safety failure. In this case, the analyst will receive a list of every component fault in Table 1. This information may be valuable, but now the analyst must determine where to target safety improvements in the system. At this point, the analyst can use the fault tree generation tool to generate a list of all trajectories that lead to the safety failure. This fault tree will not be analyzed for probability of the top event occurring, but instead will be used to determine which faults and in what combinations can trigger the safety failure. By focusing on *single point failures* the safety analyst can

determine that if any of the events *BRK-HYD-FHS*, *BRK-HYD-MAS-FRV*, *BRK-HYD-MAS-FSL*, *BRK-HYD-MAS-RRV*, *BRK-HYD-MAS-RSL*, *BRK-HYD-RHS*, or *BRK-MEC-ROD* occur, the safety failure will result. The safety engineer must now work with the system designer to eliminate these single point failures or to modify the system design such that they do not trigger a safety failure.

Diagnostic Analysis

Once the system is fielded, there may be the need to perform diagnostic analysis. For example, the system may exhibit a failure in all four brakes at once. The diagnostics analysis is used to determine what may have caused the observed aberrant behavior. For this example, the analyst can use the diagnostic algorithm from Chapter IV to generate a list of all possible causes of the failure. This list would entail all of the component failures from Table 1.

At this point, the analyst can use the fault tree generation tool to generate a list of all trajectories that lead to the failure. This fault tree will not be analyzed for probability of the top event occurring, but instead will be used to determine which events and in what combinations can cause the observed behavior. The analyst may want to examine the different combinations of event failures and perform probabilistic analysis on the individual trajectories that lead to the observed behavior. This information can aid in determining the most probable cause of the failure.

If data that shows certain component failures occurred is available, this analysis could be performed with those component failure probabilities set to *1.0*. This analysis would change the diagnostic results to take into account component failures that did occur. By combining model information with actual system information, the diagnostic

analyst can determine the most probably causes of the failure given the actual component failures that did occur. The analyst can also determine which trajectory the system most likely followed in leading to the failure.

Analysis Results

The models shown were constructed based on an example in the WinR [7] documentation. Safety and reliability analyses were performed on the models by generating fault trees to represent different top events the system could encounter and be performing reachability analyses. Diagnostic data sets were generated for different error conditions that could be observed. Simulation was used to verify the models met the desired behavior. It is important to note that the models were not constructed by reverse engineering the fault trees from the WinR documentation. Instead, the models were constructed based on functional and physical descriptions of the braking system and a detailed map of potential component and subassembly failures.

Analysis results have shown the techniques described in Chapter III and Chapter IV allow validated and verified models to be used to perform safety, reliability, and diagnostic analyses. MISAS allowed a single, integrated set of models to be used to generate all three types of analysis information. Comparison of the fault tree data generated to independently constructed fault trees has verified the correctness of the fault tree generation algorithms. Without MISAS, the system analyst would have been forced to manually construct separate models for safety, reliability, and diagnostics analyses. MISAS also allows the generation of new fault trees from the integrated models in a fraction of the time required to construct the fault trees manually.

CHAPTER VII

RESULTS AND FUTURE WORK

Results

Previously, only separate modeling and analysis techniques existed for the design of high consequence, high assurance systems. Safety, reliability, and diagnostics were considered on an individual basis. When designing a complex system, separate organizations are often responsible for ensuring the design meets specific (i.e. safety or reliability) requirements. This implied the need for separate models and analyses to exist for each area of design verification. Associations between the separate models were weak at best. When system designs were changed, several sets of models had to be updated to represent the design modifications. With many models, and many organizations modifying the models, the opportunity for inconsistencies between the different models is evident.

Safety, reliability, and diagnostics all share similarities in the tools used to perform modeling and analysis. All three analyses deal with emergent system properties. Safety and reliability use very similar techniques; often the major difference between a safety analysis and a reliability analysis is the definition of the "top event" and the type (qualitative or probabilistic) data needed. Diagnostics are usually performed post-mortem using the same basic models as safety and reliability analyses. An important observation is that all three techniques rely on modeled information about how the system behaves, under both nominal and fault conditions. From the behavioral information, safety, reliability, and diagnostic analyses can be performed by generating

the specific data for the current type of analysis. By generating analysis data from an integrated set of models, the opportunity for inconsistency between reliability, safety, and diagnostic models is eliminated.

The environment described in this dissertation has been prototyped and applied to several representative systems. In addition to the automotive braking system, several high consequence weapons systems have been modeled and analyzed using the environment. Feedback from weapons designers at Sandia National Laboratories was used to tailor and modify the environment to best meet the needs of a high consequence, high assurance systems designer and analyst. Domain specific analysis tools were integrated to allow the system analyst to utilize familiar tools for detailed domain specific analysis.

While this research focused on modeling and analysis of discrete systems, some of the techniques can be applied to continuous systems. Discretizing a continuous system involves turning the system into a set of discrete models. It is important to note that this process inherently involves the loss of information. It is up to the system modeler to ensure no critical information is lost. The techniques described in this dissertation involve examining the entire design space for a system. Exhaustive examination of a continuous system is not possible for large systems. It is imperative that if these techniques are applied to a continuous system that the discrete models must contain all critical safety, reliability, and diagnostics information about the system.

This environment provides the first integrated safety, reliability, and diagnostics modeling and analysis suite of tools. Several problems with disjoint modeling and analysis environments have been minimized or solved by this integrated environment.

This environment allows for an integrated behavioral specification model to be used to generate safety, reliability, and diagnostic information. By integrating the models, the possibility of discrepancies between different model sets has been eliminated. Model verification and validation routines are used to determine whether the integrated models are correct before generating safety, reliability, or diagnostic analysis data. The system analyst must still ensure that the design meets safety and reliability requirements using domain specific analysis tools. This environment does not eliminate the need for safety, reliability, and diagnostic experts. Instead, it eliminates the tasks of ensuring all parties are using consistent models and data sets for performing their analyses. The environment helps to eliminate analysis errors, which results in greater confidence in the resulting safety, reliability, and diagnostic analyses.

Future Research

While this research resulted in a prototype environment for integrating safety, reliability, and diagnostics, it also uncovered areas where further research and development should be undertaken. Areas where improvements can be made include the integrated modeling environment, the scalability of the analysis environment, and the inclusion of new analysis tools in the analysis tool suite. Details of possible future work are detailed for each application area in the following sections.

Modeling

The Graphical Model Editor (GME) is constantly improving. New features and concepts are incorporated into GME on a continuing basis. As features become available, the integrated modeling paradigm should be examined to see if the new

features can add any functionality to the system. While the new features can and will enhance the user-friendliness of GME, they may also enable better characterization of the conceptual features captured by the integrated modeling paradigm. The integrated modeling paradigm will change over time, but hopefully only as new concepts, such as security, are deemed necessary in the integrated modeling and analysis environment.

A new GME feature that will be incorporated into the integrated modeling paradigm is the ability to "inherit" connections across aspects. Currently the nominal and fault behaviors are modeled together in order to facilitate easier modeling. Conceptually, fault behavior is a superset of nominal behavior and should retain all features of the nominal behavior. This concept could not be captured in the previous versions of GME without requiring the system modeler to manually reconstruct the details of the nominal behavior in another aspect (the fault behavior aspect). The latest version of GME contains features that allow the concept of fault behavior superceding nominal behavior to be realized. With this feature, the system modeler can construct the fault behavior by extending the nominal behavior. By allowing this type of modeling, GME better represents the conceptual ideas of the integrated modeling paradigm.

Model Migration

Model migration refers to the problem of taking existing models and modifying them to work in an updated modeling paradigm. Using the current tools, if significant paradigm changes are introduced, older models cannot be loaded into the improved modeling paradigm. GME contains some basic model migration features, but they primarily focus on cases where attributes are modified, added, or removed from the

paradigm. Modifications that involve introducing new atomic parts, models, connections, or conditionals often render the existing models unusable in the new paradigm.

For prototype systems, model migration is generally not a concern. However, when the system goes into widespread use, a large number of models may be constructed. Manually rebuilding these models to accommodate an updated paradigm often involves enough effort to suppress migrating to the new paradigm. Organizations are not likely to spend the resources required to manually reconstruct existing, verified models in a modified paradigm. In addition to the cost of reconstructing the models, is the cost of re-verifying the models to eliminate the introduction of human errors in the migration process. Models that are reconstructed manually would require validation and verification to ensure the model migration process was complete and successful.

Automated model migration would "convert" models constructed in an old paradigm to meet the requirements of the new paradigm. User interaction may be necessary, but the model migration process should alert the user to potential problems and allow the user to "guide" the model migration process. Any purely mechanical processes involved in migrating the models would be automated. By automating the model migration process, the need for verification and validation of the updated models can be reduced and possibly eliminated. While paradigm modifications are not frequent, they do occur and the cost required to update existing models to the new paradigm can be extensive.

Analysis

Throughout the course of this research, several areas of potential improvement to MISAS have been identified. Most of these improvements have to do with scalability or

with ensuring that the tools can analyze large models. Some of the improvement areas focus on increasing the performance of the existing tools. Lastly, some areas where new analysis routines could be added to the existing toolset have been identified. This section details several of the more promising potential areas for improvement.

Security and Performance

Surety is a term used to identify the combination of safety, security, reliability, and performance. Surety is often used in reference to verification of high consequence, high assurance systems. By extending the integrated modeling paradigm to incorporate features of security and performance, the modeling paradigm can be used for integrated surety modeling. Since safety and reliability have already been integrated into the environment, it is preferable to extend the integrated modeling and analysis system to incorporate security and performance.

In order to add security modeling and analysis to the integrated environment, an analysis of security modeling must be performed. In addition to examining traditional security modeling tools, how the concepts captured by these tools may be integrated into a behavioral modeling paradigm must be addressed. By incorporating security modeling concepts into the integrated modeling environment, the same set of integrated models can also be used for security modeling and analysis as for safety, reliability, and diagnostics. The same holds in the case of performance modeling. In addition, performance modeling must be restricted to those techniques that can be applied to problems in the given domain. A mapping between security models and security analysis tools must be developed after adding security to the modeling environment. The modeling environment does not allow integrated analysis until this crucial step has been

undertaken. Some security analyses make use of fault tree models and fault tree analysis – these security techniques can be performed with the current modeling and analysis environment. If additional security analysis tools are to be integrated, the mapping from the integrated models to the security analysis tools will have to be developed. This mapping may be a simple algorithm or a complex mapping problem. Until the selected security tools have been identified this mapping can not be explored. This is also true for the addition of performance to MISAS.

New Diagnostic Tools

Diagnosability is a science that deals with determining if systems can be diagnosed and the best distribution of sensors to allow a system to be diagnosed. Diagnosability tools exist that utilize physical fault propagation models [9]. The integrated modeling paradigm contains physical system models and could be extended to include fault propagation information. This improved modeling paradigm would allow DTOOL [35] to be used as a diagnostic and diagnosability analysis tool. The ability to extend the integrated modeling and analysis environment illustrates the flexibility of the approach taken.

Analysis as to the information required by new diagnostic routines and how to add that information to the integrated paradigm would have to be performed to fully integrate a new diagnostic analysis tool. As with security and performance above, it is impossible to predict the impact of adding new tools and techniques to the integrated system until the new tools have been identified and analyzed. Integrating new tools requires updating not only the integrated modeling paradigm, but also the SSAT algorithms.

Scalability

Scalability of OBDDs is still an unsolved problem. Heuristic methods can be used to minimize the size of generated OBDDs, but no known efficient method exists for determining an optimum variable ordering for an OBDD. Work should continue to incorporate state of the art research in OBDDs into MISAS.

OBDD Variable Ordering

Boolean variable ordering has been shown to have a great impact on the size of an OBDD representing a Boolean formula. Discovering an optimal OBDD variable ordering for a given problem has been shown to be a NP-Complete problem [38]. Heuristic methods exist to enable "better" variable orderings to be discovered. In fact, most available OBDD implementation packages contain routines for automatically re-ordering the variables to reduce OBDD size.

Heuristic methods such as interleaving variables have shown great promise in reducing the transition relation OBDD sizes. Further research into variable ordering could aid this work by allowing larger (in terms of design space) models to be analyzed. Also, smaller OBDD representations increase performance by enabling the analysis algorithm to operate in physical memory on the analysis platform. The elimination of the need for virtual memory enhances analysis performance. OBDD research from other areas, such as VLSI design verification, should be examined and leveraged to improve analysis performance whenever possible.

Potential Pruning Enhancements

While the reachability and structural design space pruning algorithms provide substantial increases in the size of models that can be analyzed, there are improvements that can be made. Since analyses of tightly coupled behavioral specifications do not receive any improvement from automatic design space pruning, techniques that allow the system analyst to determine which areas of the models to prune need to be developed. For high consequence, high assurance systems, the analyst must take great care to prune only those sections of the behavioral specifications that are not involved in any of the trajectories to be examined. Using user input to determine pruning sets would eliminate the ability to ensure complete analysis of the models. In order to claim completeness, some method for allowing a user to "guide" an algorithm that automatically prunes the design space would be necessary.

APPENDIX A

MODEL INTEGTRATED COMPUTING

In *Model-integrated computing* (MIC), integrated, multiple-view, domain specific models capture information relevant to the system under investigation. Models explicitly represent the designer's understanding of an entire system, including the information-processing architecture, physical architecture, and operating environment. Integrated modeling explicitly represents dependencies and constraints among various modeling views. The particular type of MIC discussed here is *Model Integrated Program Synthesis* (MIPS). MIPS uses integrated models to generate or configure run-time systems and analysis tools. A process known as *semantic translation* is used to extract information from the integrated models and translate the information into a format usable by the run-time system or analysis tools. This semantic translation can be a complicated transformation that applies the semantics to the models contained in the MIC database.

The Multigraph Architecture (MGA) provides an infrastructure for model-integrated computing and is described in detail elsewhere [39]. A typical model-integrated tool configuration is shown in Figure 26. The integrated environment includes Modeling Tools, Integrated Model Database, Analysis Tools, and Application Synthesis Tools. The Analysis Tools work with tool-specific analysis models; the applications are specified in terms of executable models. The modeling paradigm of the analysis tools and the executable models are domain independent. In a given domain, the relevant information about the design artifact is captured by a multiple-view, domain specific modeling paradigm. In MGA, the modeling paradigm is described by a meta-language.

The meta-language representation of the modeling paradigm is used to generate components of a Metaprogrammable Model Server, and a Metaprogrammable Graphical Model Editor (GME). Key components of the model server are the "Model Interpreters". The role of the Model Interpreters is to translate the domain specific model into the analysis models for the tools and the executable models of synthesized applications. This architecture allows that the analysis and synthesis tools to share design information that is common without requiring that the tools use the same modeling paradigm.

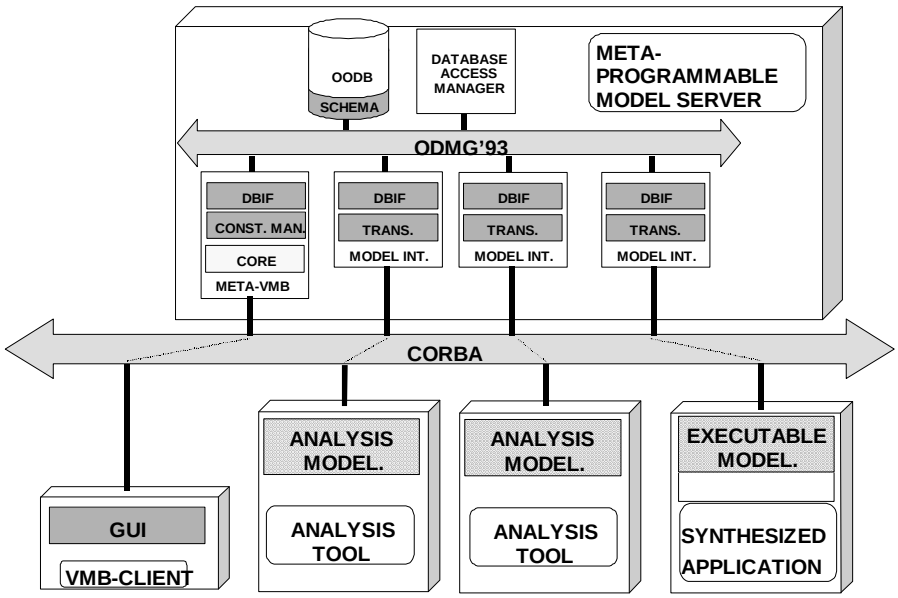

Figure 26: An example MultiGraph architecture tool environment

An integrated tool environment is built in the following steps using the MGA infrastructure:

1. Systems and domain experts conduct domain analysis and specify an integrated modeling paradigm, which can capture key aspects of the system.

The modeling paradigm is comprised of the concepts, relationships, model composition principles and constraints which are specific to the domain.

2. Using the formal representation of modeling paradigms, systems and domain experts specify and create a domain specific model building, model analysis, and software/system synthesis (MIPS) environment. The environment includes reusable domain specific components, general building blocks, domain specific model analysis tools, and software synthesis tools. Completion of this step is supported by MGA meta-tools.
3. Domain and application engineers build integrated, multiple view models of systems to be designed and implemented. The multiple view models typically include requirement and design models, are based on formally specified semantics, and support performance, safety and reliability analysis processes.
4. Domain and application engineers analyze the models according to the nature and needs of the domain. The analysis is typically supported by generic tools (i.e. simulation and reachability of behavioral models). The domain specific models are translated into the input languages or input data structures of the connected analysis tools. The model translation is completed by the MGA model interpreters.
5. If necessary, the validated models are used for the automatic synthesis of software applications.

MGA is useful for providing customizable, domain specific, visual model editors. MGA also provides a framework for gaining access to the information contained in the

models and for providing interfaces to analysis tools or a run-time support system. MIC, and in particular MIPS, provides an excellent framework for development of advanced software systems.

APPENDIX B

SYMBOLIC MODEL CHECKING TOOLS

Symbolic Model Checking

Symbolic Model Checking (SMC) is a process for mathematically analyzing system models. McMillan [21] discusses the theory of SMC in his doctoral dissertation. SMC is a method for performing *formal system verification*. Formal verification relies on the existence of a mathematical system model, a language for specifying the desired system properties, and a method for proving the system model satisfies the specified system properties. Automatic verification refers to computer assisted verification proofs. SMC is a method for performing automatic verification of system models. Specifically, McMillan deals with automatic verification of computer hardware [21]. Other complex systems, such as high consequence, high assurance systems, can benefit from the same reduction of errors during system design. System safety and reliability can be improved at design time by reducing design errors.

SMC is a computational technique that is being applied to the problem of system verification. The symbolic system model takes the form of a transformation from a state machine model to a Boolean model representation. The Boolean representation of the system is a mapping from the Boolean representation of the current state of the system to the next state of the system. A Boolean representation of the system state is a vector of Boolean variables of sufficient length to encode all of the possible state and event variables in the original state machine model. By representing all possible mappings

from any current system state to the next system state(s), the complete behavior of the system can be captured symbolically in a single relation.

Once a symbolic model representation is defined, the system can be examined using mathematical concepts. The ability to perform *backward* analyses (with respect to time) is an attractive feature of SMC. Since the symbolic representation is reversible, previous system states can be determined from the current system state. Operations on the symbolic model can be performed as easily backward as forward [21][43]. This feature allows complex system operations to be implemented once, with only small changes necessary to use the same operation for model checking both forward and backward with respect to time. By analyzing the system mathematically, the models do not suffer from the "state space explosion problem". The symbolic models capture all the information required about which states the system may be in at any given time.

Symbolic Model Verifier

Symbolic Model Verifier (SMV) has been described in several sections of this dissertation. SMV is a tool designed to allow the user to perform SMC on finite state machine models. McMillian [21] is one of the developers of the SMV toolset [19]. SMV has tools to allow for system model verification. SMV allows the use of CTL to specify system requirements. The SMV tool can then verify the finite state machine models against the system requirements. OBDDs are used to enable model verification.

SMV uses a textual based language to describe system requirements and system specifications. Concepts such as hierarchy are not part of the SMV language. SMV requires the user to become accomplished with the mathematical proof checker. Ideally, end users could perform analyses using tools and concepts they are familiar with.

Requiring the users to use the existing proof checker requires the system analyst to become proficient in a new domain. Additionally, SMV only supports finite data types. Performing quantitative reliability, safety, or diagnostic analyses with SMV is not feasible.

Software Cost Reduction

Software Cost Reduction (SCR) is advertised as a formal method for reducing the cost of software development by reducing the number of errors in the software specification. SCR uses a different method for model checking specifications. SCR works directly on a representation of the models: it does not translate the models into a symbolic representation. The SCR authors have published intentions to being using SMC, in order to utilize the distinct advantages SMC has over traditional model analysis techniques. The method proposed will utilize OBDDs to represent the symbolic models and for performing the SMC operations [24].

Verisoft

Verisoft has been developed by Lucent Technologies for the analysis of concurrent, reactive software systems [51]. Verisoft uses model-checking techniques to verify that software systems (actual source code) do not suffer from deadlock, livelock, divergence, and assertion violations. Verisoft differs from the other SMC techniques in that it does not operate on system models, but rather on system source code. In particular, Verisoft uses a special data structure called QDD-s (queue-content decision diagrams) to perform SMC operations [52]. QDD-s are intended to represent sets of unbounded first in, first out (FIFO) queues.

Traditional state space searches construct an entire reachability graph for a given system design. Techniques such as state-less searches do not store any intermediate system states in memory when exploring the state space of a concurrent system. However, these methods will not terminate if the complete reachability graph for a system has cycles. Even examining small examples, such as the dining philosophers problem with only four philosophers, can become inefficient. In the case of four philosophers, the complete reachability graph contains 708 transitions while a state-less search examines 386,816 transitions [51]. Verisoft uses sleep sets for pruning state spaces as the space is explored. Sleep sets are members of a set of algorithms know as *partial-order methods* [53]. Empirical results show that, in many cases, most states are visited only once when using sleep sets to prune the state space during state space exploration [53]. These state space exploration techniques depend on the notion of independence of transitions. If the transitions are independent, changing their order has no impact on their effects [51].

There are some problems with using Verisoft as a SMC tool. First, Verisoft can only guarantee complete coverage of the state space up to some depth [51]. Secondly, Verisoft only provides tools to check for deadlock, livelock, divergence, and assertion violations. Many different conditions may be interesting to system designer. Lastly, Verisoft's scalability is limited by the state-explosion problem [54], the very problem SMC is intended to reduce or eliminate.

REFERENCES

[1] Leveson, N., **SAFEWARE: System Safety and Computers**, Addison-Wesley Publishing Company, 1995.

[2] Sztipanovits, J., Carnes, R., Misra, A., "Finite-State Temporal Automata Modeling for Fault Diagnosis," *Proc. Of the 9th AIAA Conference on Computing in Aerospace*, San Diego, CA, October, 1993.

[3] Harel, D., "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming* **8**, 1987, pp.231-274.

[4] Harel, D., Naamad, A., "The STATEMATE Semantics of Statecharts", *ACM Transactions on Software Engineering Methods* **5**:4, Oct. 1996.

[5] Xie, M., **Software Reliability Modeling**, World Scientific, 1991.

[6] Lee, W., et. al., "Fault Tree Analysis, Methods, and Applications – A Review", *IEEE Transactions on Reliability*, Vol. R-34, No. 3, pp. 194-203.

[7] Sandia National Laboratories, *WinR Reliability Analysis Software*, Systems Reliability Department, 1996.

[8] Misra, A., Sztipanovits, J., Carnes, R., "Robust Diagnostic System: Structural Redundancy Approach", *Proceedings of the SPIE's International Symposium on Knowledge-Based Artificial Intelligence Systems in Aerospace and Industry*, Orlando, FL, April 1994.

[9] Misra, A., "Sensor-Based Diagnosis of Dynamical Systems", Ph.D. Dissertation, Vanderbilt University, May 1994.

[10] Peterson, J., "Petri Nets", *ACM Computing Surveys*, 1977.

[11] Huber, P., Jensen, K., "Hierarchies in Coloured Petri Nets", Lecture Notes in Computer Science 483**,** Springer Verlag, 1991.

[12] Jensen, K., "Colored Petri Nets: A High Level Language for System Design and Analysis", **Advances in Petri Nets 1990**, Lecture Notes in Computer Science 483, Springer Verlag, 1991.

[13] Nicol, D., Roy, S., "Parallel Simulation of Timed Petri Nets", *Proceedings of the 1991 Winter Simulation Conference*, pp. 574-583, Phoenix, AZ, December 1991.

[14] Choi, H., Kulkarni, V., Trivedi, K., "Markov Regenerative Stochastic Petri Nets", *Performance Evaluation*, pp. 337-357, 1994.

[15] S. Christensen, L.M. Kristensen, "State Space Analysis of Hierarchical Coloured Petri Nets", *Petri Nets in System Engineering*, Hamburg, Germany, September 1997.

[16] Jensen, K.: **Coloured Petri Nets**, Springer-Verlag, 1996.

[17] Fard, N., "Determination of Minimal Cut Sets of a Complex Fault Tree", *Computers and Industrial Engineering*, Vol. 33, pp. 59-62, 1997.

[18] Contini, S., "A new hybrid method for fault tree analysis", *Reliability and System Safety*, 49, pp. 13-21, 1995.

[19] McMillan, K., "The SMV System", CMU Tech Report.

[20] J. R. Burch, et. al., "Symbolic model checking: 10E20 states and beyond", *LICS*, 1990.

[21] McMillian, K., **Symbolic Model Checking**, Kluwer Academic Publishers, 1993.

[22] Heitmeyer, C., Kirby, J., Labaw, B., "Tools for Formal Specification, Verification, and Validation of Requirements", *Proceedings of COMPASS*, 1997.

[23] Heitmeyer, C., Jeffords, R., Labaw, B., "Automated Consistency Checking of Requirements Specifications", *ACM Transactions on Software Engineering and Methodology*, 5:3, pp. 231-261, July 1996.

[24] Bharadwaj, R., Heitmeyer, C., "Verifying SCR Requirements Specifications Using State Exploration", *Proceedings First ACM SIGPLAN Workshop on Automatic Analysis of Software*, January 1997.

[25] Holzmann, G., **Design and Validation of Computer Protocols**, Prentice-Hall, 1991.

[26] Department of Defense, "MIL-STD-1629A: Procedures for Performing a Failure Mode, Effects, and Criticality Analysis", November 1980, Washington DC.

[27] Andrews, J., Moss, T., **Reliability and Risk Assessment**, Longman Scientific and Technical, 1993.

[28] McKinney, B., "FMECA, The Right Way", *Proceedings of the IEEE Annual Reliability and Maintainability Symposium*, pp. 253-259, 1991.

[29] Kara-Zaitri, C., et. al., "An Improved FMEA Methodology", *Proceedings of the IEEE Annual Reliability and Maintainability Symposium*, pp. 248-252, 1991.

[30] Eubanks, C., et. al., "Advanced Failure Modes and Effects Analysis Using Behavioral Models", *Proceedings of the 1997 ASME Design Engineering Technical Conference*, Sacramento CA, September 1997.

[31] Balakrishnan, M., Trivedi, K., "Stochastic Petri nets for the reliability analysis of communication network applications with alternate-routing", *Reliability Engineering and System Safety*, 52, pp. 243-259, 1996.

[32] Sztipanovits, J., Bourne, J., "Architecture of Intelligent Medical Instruments", *Journal of Biomedical Measurements, Informatics, and Control*, London, UK, Vol. 1, No. 3, 1987.

[33] Milne, R., "Strategies for Diagnosis", *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. SMC-17, No. 3, 1987.

[34] Milne, R., "Fault diagnosis through responsibility", *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, CA, August 1985.

[35] Misra, A., et. al. , "Diagnosability of Dynamical Systems", *Proceedings of the Third International Workshop on Principles of Diagnosis*, pp. 239-244, Rosario, WA, October 1992.

[36] Anglano, C., Portinale, L., "B-W Analysis: a Backward Reachability Analysis for Diagnostic Problem Solving suitable to Parallel Implementation", *15th International Conference on Application and Theory of Petri Nets*, Zaragoza, **Lecture Notes in Computer Science 815**, pp. 39-58, Springer Verlag, 1994.

[37] Liggesmeyer, P., Rothfelder, M., "Automating Reliability and Safety Analysis Based on Formal System Models", *Engineering of Computer Based Systems*, Jerusalem, Israel, April 1998.

[38] Bryant, R. E., "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computers*, C-35(8), 1986.

[39] Sztipanovits, et al., "MULTIGRAPH: An Architecture for Model-Integrated Computing" *Proceedings of the IEEE ICECCS'95* Ft. Lauderdale, Florida, Nov. 6-10. 1995.

[40] Davis, J., Scott, J., Sztipanovits, J., Martinez, M.: "Multi-Domain Surety Modeling and Analysis for High Assurance System", *Proceedings of the Engineering of Computer Based Systems*, Nashville, TN, March 1999.

[41] Nordstrom, G.: "Metamodeling - Rapid Design and Evolution of Domain-Specific Modeling Environments", Ph.D. Dissertation, Vanderbilt University, 1999.

[42] Rice, M., Seidman, S.: "A Formal Model for Module Interconnection Languages", *IEEE Transactions on Software Engineering*, Vol. 20, January 1994.

[43] Bryant, R. E., "Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams", *Technical Report CMU-CS-92-160*, School of Computer Science, Carnegie Mellon University, June 1992.

[44] Lee, C. Y., "Representation of Switching Circuits by Binary Decision Programs," *Bell System technical Journal* pp. 985-999, 1959.

[45] Burch, J. R., Clarke, E.M., Long, D. E., "Symbolic Model Checking for Sequential Circuit Verification," Technical Report, CMU-CS-93-211, Carnegie Mellon University, July 15, 1993.

[46] Helbig, J., Kelb, P., "An OBDD Representation of Statecharts", *Proceedings of the European Conference on Design Automation*, pp. 142-151, Paris, France, 1994.

[47] Davis, J., Scott, J., Sztipanovits, J., Karsai, G., Martinez, M.: "An Integrated Multi-Domain Analysis Environment For High Consequence Systems", *Proceedings of the 1998 ASME Design Engineering Technical Conference / Computers in Engineering*, Atlanta, GA, September 1998.

[48] Davis, J., Scott, J., Sztipanovits, J., Karsai, G., Martinez, M., "Integrated Analysis Environment for High Impact Systems", *Engineering of Computer Based Systems*, Jerusalem, Israel, April 1998.

[49] Cabodi, G., Quer, S., Camurati, P.: "Memory Optimization in Function and Set Manipulation with BDDs", in *Software – Practice and Experience*, Vol. 28, pp. 99-120, January 1998.

[50] Sinnamon, R., Andrews, J.: "New Approaches to Evaluating Fault Trees", in *Reliability Engineering and System Safety*, Vol. 58, 1997.

[51] Godefroid, P., "Model Checking for Programming Languages using VeriSoft", *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pp.174-186, Paris, January 1997.

[52] Boigelot, B., et. al., "The Power of QDDs", *Proceedings of the Fourth International Static Analysis Symposium*, Paris, September 1997.

[53] Godefroid, P., "Partial-OrderMethods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem", *Lecture Notes in Computer Science*, Vol. 1032, Springer-Verlag, January 1996.

[54] "Model Checking Software with Verisoft", presentation to UC-Berkley and NASA Ames, available from http://www.bell-labs.com/projects/verisoft/papers.html, November 1998.

[55] Leveson, N., et. al., "Requirements specification for process-control systems", *IEEE Transactions of Software Engineering*, 20:9, September 1994.

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

[57] Bharadwaj, R., Heitmeyer, C., "Applying the SCR Requirements Specification Method to Practical Systems: A Case Study", *The 21st Software Engineering Workshop*, NASA Goddard Space Flight Center, Greenbelt MD, December 1996.

[58] Scott, J., "Efficient Verification of Distributed Real-Time Systems Using Symbolic Methods", Ph.D. Dissertation, Vanderbilt University, forthcoming.

[59] Fohler, G., "Adaptive Fault-Tolerance with Statically Scheduled Real Time Systems", *Proceedings of the 9thEuromicro Workshop on Real Time Systems*, 1997.