Model-Based Safety Analysis Final Report

Anjali Joshi
Mike Whalen
Mats P.E. Heimdahl

{ajoshi, heimdahl}@cs.umn.edu
(612)-624-7590

mwwhalen@rockwellcollins.com
(612) 625-4543

Advanced Technology Center
Rockwell Collins, Inc.,
Cedar Rapids, IA 52498 USA

Department of Computer Science and Engineering
University of Minnesota
4-192 EE/SC Building
200 Union Street S.E.
Minneapolis, Minnesota 55455

Abstract

System safety analysis techniques are well established and are used extensively during the design of safety-critical systems. Despite this, most of the techniques are highly subjective and dependent on the skill of the practitioner. Since these analyses are usually based on an informal system model, it is unlikely that they will be complete, consistent, and error free. In fact, the lack of precise models of the system architecture and its failure modes often forces the safety analysts to devote much of their effort to gathering architectural details about the system behavior from several sources and embedding this information in the safety artifacts such as the fault trees.

This report describes *Model-Based Safety Analysis,* an approach in which the system and safety engineers share a common system model created using a model-based development process. By extending the system model with a fault model as well as relevant portions of the physical system to be controlled, automated support can be provided for much of the safety analysis. We believe that by using a common model for both system and safety engineering and automating parts of the safety analysis, we can both reduce the cost and improve the quality of the safety analysis. Here we present our vision of model-based safety analysis and discuss the advantages and challenges in making this approach practical.

Table of Contents

1 Introduction: Model-Based Safety Analysis

Safety engineers traditionally perform analysis, such as fault tree analysis [24], based on information synthesized from several sources, including informal design models and requirements documents. Unfortunately, these analyses are highly subjective and dependent on the skill of the engineer. Fault trees are one of the most common techniques used by safety engineers, yet different safety engineers will often produce fault trees for the same system that differ in substantive ways. The final fault tree is often produced only through a process of review and consensus building between the system and safety engineers. Even after a consensus is reached, it is unlikely that the analysis results will be complete, consistent, and error free due in part to the informal models used as the basis of the analysis. In fact, the lack of precise models of the system architecture and its failure modes often forces the safety analysts to devote much of their effort to gathering information about the system architecture and system behavior and embedding this information in the safety artifacts such as the fault trees.

We hypothesize that this situation can be significantly improved by performing the safety analysis activities based on formal models of the system under development. In *model-based development* various development activities such as simulation, verification, testing, and code-generation are based on a formal model of the system under development. We propose to extend model-based development to incorporate the safety analysis activities in addition to the traditional development activities, an approach we call *Model-Based Safety Analysis*. Since the safety analysis requires knowledge of the different faults that can occur and the various ways in which the system components can malfunction, the *nominal* (non-failure) system behavior captured in model-based development must be augmented with the *fault behavior* of the system. Model-based safety analysis operates on a formal model describing both the nominal system behavior and the fault behavior. Our aim is to provide a precise model of system behavior and to automate parts of the safety analysis process and, consequently, both reduce the cost and improve the quality of the safety analysis process.

Much of the benefit of model-based development is derived from a tool framework that supports formal specification of the system model [13], [15], [43], formalizing requirements [30], and automated verification [14], [19]. To aid model-based safety analysis, this framework must be extended to support (1) specification of the fault behaviors of the system, (2) extension of the nominal system behavior with these fault behaviors to yield an *extended system model*, and (3) automated analysis and generation of safety artifacts, like fault trees, from the extended system model.

In this report, we describe the model-based safety analysis approach and discuss various research challenges that must be met to make this approach practical.

Report Organization

The remainder of the report is organized as follows. Section 2 briefly introduces the terminology and summarizes the traditional safety analysis process currently practiced in the commercial avionics industry. Section 3 discusses the model-based safety analysis approach as an extension to model-based development. We point out the important distinctions between these two approaches and discuss changes that might be necessary to accommodate model-based safety approach in the traditional safety analysis process. We then illustrate model-based safety analysis with the help of a *wheel brake system* example derived from the ARP 4761 safety analysis guidelines [2] in Section 4. This example was created using existing tools and techniques and was designed to help identify future research directions involving extending existing tools for model-based safety analysis. We discuss related work in automating safety analysis in Section 5. Section 6 concludes the report and contains a discussion of our short-term and long-term goals towards addressing the shortcomings in current modeling and analysis processes and tools.

2 Background

This section first introduces definitions for the terminology that will be used in the remainder of this report. Afterwards, we briefly describe the steps in a traditional safety analysis process, as specified in ARP 4761 [2].

2.1 Terminology

In the related areas of reliability and safety, there is no firm consensus on the terminology for even some of the basic terms used [2], [23], [24], [25], [32]. J.-C. Laprie [23] promoted *dependability* as a generic concept that included reliability, maintainability, availability, safety, with the view that all of the above are distinct perceptions of the same attribute of a system: its dependability. There was an effort to come up with a consistent set of concepts and terminology with the formation of IEEE-CS Technical Committee on Fault-Tolerant Computing in 1970 and of IFIP WG 10.4 Dependable Computing and Fault Tolerance in 1980 [32]. A recent paper by Avizienis, Laprie, Randell, and Landwehr [3] consolidates the basic concepts and taxonomy in dependability, and is paraphrased below.

2.1.1 Basic Definitions

The *service* delivered by a system is its behavior as it is perceived by its users. *Correct service* is delivered when the service implements the system function. The part of the provider's boundary, where service delivery takes place, is the provider's *service interface*. The part of the provider's total state that is perceivable at the service interface is its *external state*; the remaining part is its *internal state*. The delivered service is a sequence of the provider's external state.

2.1.2 Faults, Errors, and Failures

A *failure* is an event that occurs when the delivered service deviates from correct service. The deviation from correct service may assume different forms that are called service *failure modes*. Since a service is a sequence of the system's external states, a service failure means that at least one or more external states of the system deviates from the correct service state. The deviation is called an error. The adjudged or hypothesized cause of an error is called a *fault*. In most cases, a fault first causes an error in the service state of a component that is a part of the internal state of the system and the external state is not immediately affected. The definition of an *error* is the part of the total state of the system that may lead to its subsequent service failure. Note that, many errors do not reach the system's external state and cause a failure. A fault is *active* when it causes an error, otherwise it is *dormant*.

2.1.3 Relationship between Faults, Errors, and Failures

The creation and manifestation mechanisms of faults, errors, and failures as summarized in [42] are as follows:

1. A fault is active when it produces an error; otherwise, it is dormant. An active fault is caused by either 1) an internal fault that was previously dormant and that has been activated by the computation process or environmental conditions, or 2) an external fault that propagates from the environment. Fault activation is the application of an input (the activation pattern) to a component that causes a dormant fault to become active. Most internal faults cycle between their dormant and active states.

2. Error propagation within a given component (i.e., internal propagation) is caused by the computation process: An error is successively transformed into other errors. Error propagation from component A to component B that receives service from A (i.e., external propagation) occurs when, through internal propagation, an error reaches the service interface of component A. At this time, service delivered by A to B becomes incorrect, and the ensuing service failure of A appears as an external fault to B and propagates the error into B via its use interface.

3. A service failure occurs when an error is propagated to the service interface and causes the service delivered by the system to deviate from correct service. The failure of a component causes a permanent or transient fault in the system that contains the component. Service failure of a system causes a permanent or transient external fault for the other system(s) that receive service from the given system.

In the rest of the report, we adopt the above terminology. In addition to the above terms, we also use the term *propagated fault* and *fault propagation* to refer to activation of the external fault due to error propagation from another component. We refer to the faults that are dependent on other faults as *dependent faults* (e.g., a power failure causing the failure of a number of components it supplies power to).

2.2 System Safety Assessment Process

This section describes the overall safety assessment process that is practiced in the avionics industry along the lines of the SAE standard ARP 4761 [2]. The descriptions of the various phases of the safety assessment process covered in this section are essentially excerpts from the ARP 4761 document.

The safety assessment process is an inherent part of the system development process. Figure 1 shows an overview of the safety assessment process. The safety assessment process includes safety *requirements identification* (on the left side of the “V” diagram) and *verification* (on the right side of the “V” diagram) supporting the aircraft development activities. An aircraft-level Functional Hazard Analysis (FHA) is conducted at the beginning of the aircraft development cycle, which is then followed by system-level FHA for individual sub-systems. The FHA is followed by Preliminary System Safety Assessment (PSSA), which derives safety requirements for the subsystems, primarily using Fault Tree Analysis (FTA). The PSSA process iterates with the design evolution, with design changes necessitating changes to the derived system requirements (and also to the fault trees) and potential safety problems identified through the PSSA leading to design changes.

Once design and implementation are completed, the System Safety Assessment (SSA) process verifies whether the safety requirements are met in the implemented design. The system Failure Modes and Effects Analysis (FMEA) is performed to compute the actual failure probabilities on the items. The verification is then achieved through quantitative and qualitative analysis of the fault trees created for the implemented design, first for the subsystems and then for the integrated aircraft.

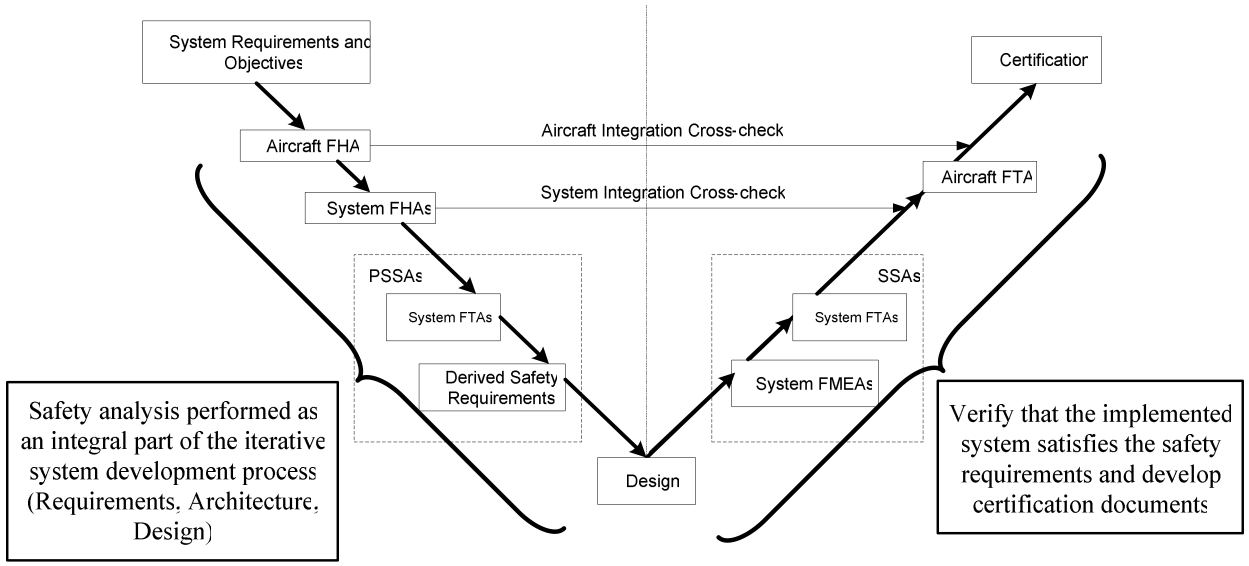
System Requirements and Objectives
Certification
Aircraft Integration Cross-check
Aircraft FHA
Aircraft FTA
System Integration Cross-check
System FHAs
PSSAs
SSAs
System FTAs
System FTAs
Derived Safety Requirements
System FMEAs
Design
Safety analysis performed as an integral part of the iterative system development process (Requirements, Architecture, Design)
Verify that the implemented system satisfies the safety requirements and develop certification documents

Figure 1: "V" Process for Traditional Safety Assessment

2.2.1 Functional Hazard Analysis

Functional Hazard Analysis (FHA) is conducted at the beginning of the life cycle. It *identifies* and *classifies* the failure conditions associated with aircraft functions (and combinations of aircraft functions) at the appropriate level, considering both loss of function and malfunctions. The FHA identifies the failure conditions for each phase of flight. There are two levels of FHA for avionics systems; the Aircraft level FHA and the System level FHA. The FHA establishes derived safety requirements needed to limit function failure effects, such as design constraints, annunciation of failure conditions, etc.

Starting from the high-level functions of the system, the failure conditions associated with these functions are considered. The effects of these failure conditions on the aircraft are determined and classified. These failure conditions can be further broken down through FHAs and Fault Trees. The failure conditions associated with safety are defined together with their respective safety objectives and the proposed means for demonstrating compliance. The aircraft level FHA specifies proposed methods for demonstrating compliance with aircraft-level safety requirements. For system-level requirements, methods for demonstrating compliance are presented in the Preliminary Systems Safety Analysis.

2.2.2 Preliminary System Safety Analysis

A Preliminary Systems Safety Analysis—PSSA—is used to complete the failure conditions list and the corresponding safety requirements. It is also used to demonstrate how the system will meet the qualitative and quantitative requirements for the various hazards identified. The PSSA process identifies protective strategies, taking into account fail-safe concepts and architectural attributes which may be needed to meet the safety objectives. The PSSA is iterative and continuous throughout the design process and identifies and captures all the derived system safety requirements.

The PSSA is a top-down approach to determine how failures can lead to the functional hazards identified by the FHA, and how the FHA requirements can be met. The aircraft (system) FHA process creates an initial set of safety requirements for the aircraft (systems). By combining this initial set of safety requirements with the design/architecture decisions made in the PSSA, a complete set of system requirements is generated. The design decisions are evaluated against the generated safety requirements with the help of Fault Tree Analysis (FTA). Since detailed item-level studies are generally not available during this phase of evaluation, PSSA failure-condition evaluation must rely in part on engineering judgment and on in-service experience with similar designs. Each design safety-requirement derived at the system-level must then be allocated to the items making up the system. Failure modes and associated probability budgets identified in PSSA Fault Tree Analysis should be used as requirements to drive the lower-level detailed studies.

Some of the important documents coming out of PSSA are planned compliance methods with FHA requirements, updated FHAs, lower-level safety requirements, qualitative FTAs, and operational requirements. The outputs of the PSSA are used as inputs to the SSA process.

2.2.3 System Safety Assessment

A System Safety Assessment (SSA) is a systematic, comprehensive evaluation of the implemented system, along with its architecture and installation, to show that the relevant safety requirements are met. The difference between the PSSA and the SSA is that a PSSA is a method to evaluate proposed architectures and derive system/item safety requirements, whereas the SSA is a verification that the implemented design meets both the qualitative and quantitative safety requirements as defined in the FHA and PSSA.

The SSA process is a bottom-up approach for verifying that the design safety requirements and objectives have been met. Through these upward hierarchical verification levels, hardware reliability requirements, architectural requirements and hardware and software Development Assurance Levels (DO-178B [33] procedures for software) are verified against the safety requirements delineated in the PSSA process. An item-level Failure Modes and Effects Analysis (FMEA) is performed and is summarized into the Failure Modes and Effects Summary (FMES) to support the failure rates of the failure modes considered in the item FTA. The system FMEA is summarized into the system FMES to support the failure rates of the failure modes considered in the system FTA. The system is reviewed via FTA to identify the failure modes and probabilities used in the aircraft FTA. The aircraft FTA is used to establish compliance with the aircraft-level failure conditions and probabilities described by the aircraft FHA. As items are integrated into systems and systems into aircraft, the failure effects are compared with the failure conditions identified in the FHA. This comparison is called an *integration cross-check*.

3 Model-Based Safety Analysis Process

In the safety-critical systems domain, *model-based development* is an increasingly popular approach for development of digital control systems. In this approach, various development activities such as simulation, verification, testing and code-generation are based on a formal model of the system expressed in a notation such as Simulink [13] or SCADE [15]. In *model-based safety analysis*, we propose to extend existing model-based development activities and tools to support safety analysis. In this section, we first briefly discuss model-based development and illustrate our model-based safety analysis approach. We then discuss how model-based safety analysis can be integrated into the traditional safety assessment process.

3.1 Model-Based Development

In model-based development, the development effort is centered on a formal specification (model) of the digital control system. This model can be subjected to various types of analysis, for example, completeness and consistency analysis, model checking, and theorem proving [30]. Model-based development tools often include automatic code generators that can derive implementations directly from models. There are currently several commercial and research tools that support model-based development. Examples of commercial tools include Simulink [13], Esterel and SCADE from Esterel Technologies [15], Statemate from i-Logix [17], and SpecTRM from Safeware Engineering [26].

3.2 Model-Based Safety Analysis

Model-based development focuses primarily on modeling the software components of the system. To perform system-level safety analysis, we must also consider the environment in which the system runs, which usually involves mechanical components. Fortunately, model-based tools and techniques can also be used to model physical components of interest. By combining models containing the digital components (software and hardware) with models of the mechanical components (pumps, valves, etc.), we create a model of the nominal system behavior. This model can then be augmented with fault models for the digital and mechanical systems to create the *Extended System Model* [8]. This model can be used to describe the behavior of the system in the presence of one or more faults.

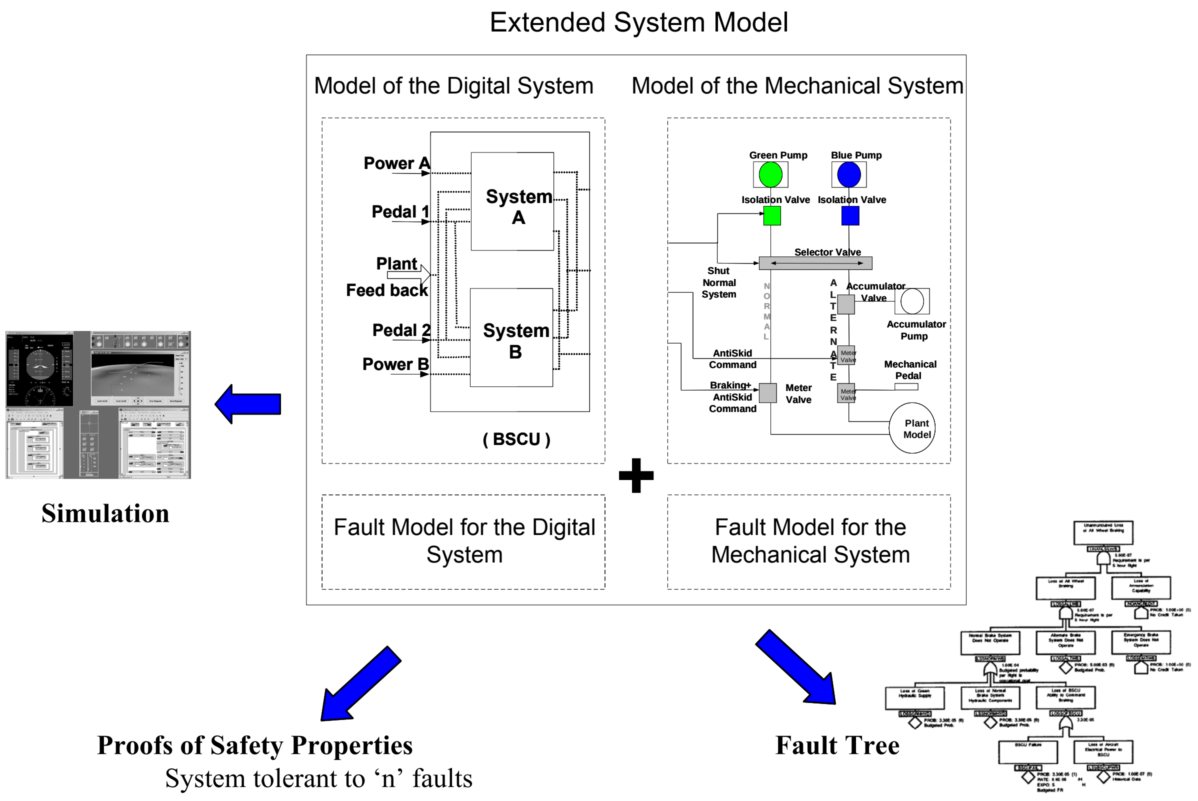

Figure 2: Automated Model-based Safety Analysis

The extended system model can be used for a variety of simulations and analyses (Figure 2). First, it allows trivial exploration of "what-if" scenarios involving combinations of faults through simulations. For more rigorous analyses, we can use static analysis tools, such as model checkers and theorem provers, to automatically prove (or disprove) whether the system meets specific safety requirements. Furthermore, these tools can also be extended to generate traditional safety analysis artifacts such as fault trees.

To support model-based safety analysis, the traditional "V" process is modified (Figure 3) so that the safety analysis activities are centered on formal system and fault models. These models are used both for systems design and safety analysis, and are the central artifact of the systems development process.

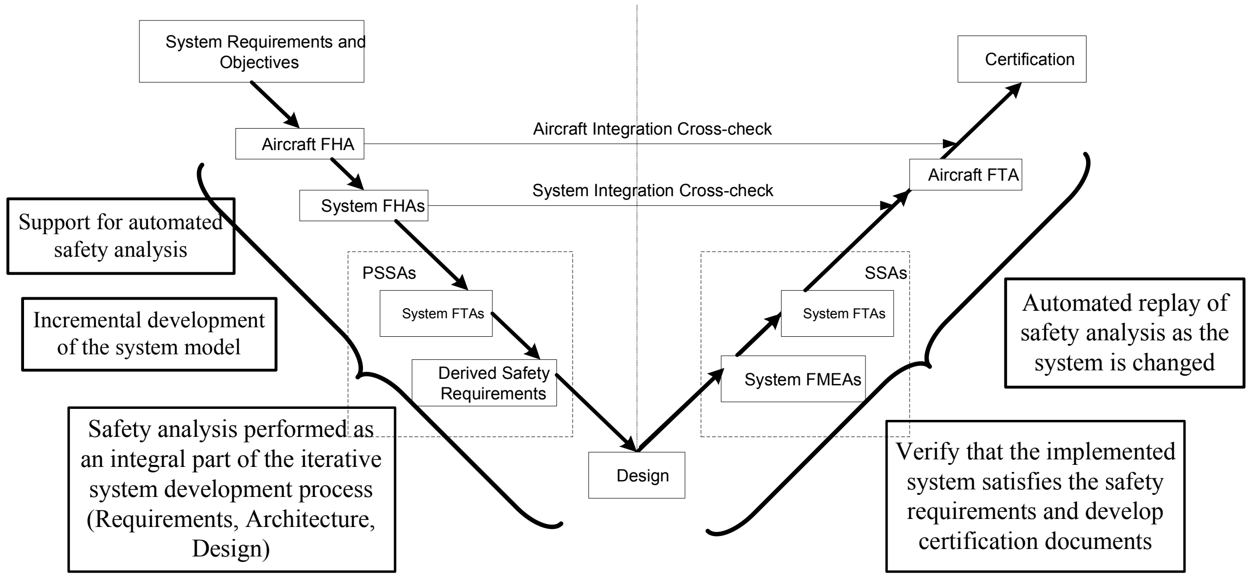
System Requirements and Objectives
Aircraft FHA
System FHAs
PSSAs
System FTAs
Derived Safety Requirements
Design
System FMEAs
System FTAs
SSAs
Aircraft FTA
Certification
Aircraft Integration Cross-check
System Integration Cross-check
Support for automated safety analysis
Incremental development of the system model
Safety analysis performed as an integral part of the iterative system development process (Requirements, Architecture, Design)
Automated replay of safety analysis as the system is changed
Verify that the implemented system satisfies the safety requirements and develop certification documents

Figure 3: Modified "V" Process for Model-Based Safety Analysis

Given extended system models, the safety analysis process consists of defining a set of formal properties to represent the (informal) safety requirements of the system and then using formal analysis techniques to determine whether the proposed system architecture satisfies the safety properties. Artifacts such as fault trees and FMEAs can be automatically generated as a byproduct of the formal analyses.

The main advantage of this approach is that the system and safety engineers work off a common, unambiguous model of the system leading to a tighter integration between the systems and safety engineering processes. The common model ensures that safety analysis results are relevant and up-to-date as the system architecture evolves, and allows safety assessment early in the system design process. Additionally, it supports exploration of different architectures and design choices by automatically determining which choices will satisfy critical safety properties.

Ideally, the use of computational tools such as model checkers can automate many safety analysis activities, and the safety engineer's task will consist primarily of reviewing the generated safety artifacts and confirming the assumptions made in the system and fault models. In this way, model-based safety analysis can lead to more accurate and complete safety analyses while reducing manual effort.

In the following sections, we describe the various model-based safety analysis activities in detail.

3.2.1 Nominal System Modeling

The primary step in model-based development (and model-based safety analysis) is creating a formal specification of the system under development. The behavior of the system can be specified in formal specification languages supporting graphical and/or textual representation; e.g., synchronous (textual) languages like Lustre [16], and graphical tools like Simulink [13] and

SCADE [15]. The logical and physical architecture of the system can also be specified in these notations or with an architecture description language such as AADL [34].

3.2.2 Formalizing Derived Safety Requirements

The derived safety requirements are determined in the same way as in the traditional "V" process. To support automated analysis, the safety properties must be expressed in some formal notation. There are several candidate notations, including temporal logics like CTL/LTL [11] or higher order predicate logics. It is also possible to specify requirements directly in the modeling language as *synchronous observers* [16] that are composed with the system model.

3.2.3 Fault Modeling

System level faults can occur due to failures of components, incorrect outputs, corrupted messages, or improper functioning of software in the absence of failures. A *fault model* captures information about the various ways in which the components of the system (both the digital controller and the mechanical system) can malfunction. It defines the behavior of common failure modes, such as *non-deterministic*, *inverted*, *stuck-at*, etc. The fault model also specifies the fault triggers that activate the component failures and their duration. We distinguish between *transient faults* (those that last for a short period of time) and *permanent faults* (those that last forever). The fault model can also specify more complex fault behaviors, such as *fault propagations*, *dependent faults*, etc. (refer to Section 2.1 for terminology). It can also specify *fault hierarchies*, in which the user can define the failure mode of a component as a function of its subcomponents or as an abstraction of the underlying fault behavior.

Depending on the system model, we can chose to model different types of digital faults, mechanical faults, timing faults, etc. The digital faults are those that relate to the digital component of the system – both hardware and software. For example, a digital fault could be inverting an output on a hardware chip. We would also like to be able to describe situations in which software fails to perform as expected (i.e. *software faults)* but it is still unclear how such faults can be described and modeled. Some software faults can be simulated by introducing failure modes on outputs, such as an inverted or non-deterministic, etc., but these failure modes do not closely match our intuitive notion of software faults and additional research is necessary to further explore this issue.

Mechanical faults are those that occur in the mechanical components of the system outside the digital controller. These are entirely dependent on the environment of the system in question, and could include electrical or hydraulic problems, network upsets, communications failures, and a variety of other kinds of problems.

3.2.4 Model Extension

To enable model-based safety analysis, the fault model is composed with the nominal system model to describe the behavior of the system in the presence of faults. We call this the *Extended System Model* (similar to the FSAP/NuSMV-SA documentation). There are two approaches to adding fault information to the system model. First, it is possible to embed the fault behavior directly into the system model. The second option is to develop the fault model as a separate entity from the system model and automatically merge these two models for analysis. We will investigate both these approaches later in the report.

3.2.5 Safety Analysis

Once we have the extended system model, the safety analysis involves verifying whether safety requirements hold in the presence of the faults defined in the fault model. The safety or system engineer can perform exploratory analysis by simulating faults on specific components and observing the behavior of the system. For more rigorous analyses, it is possible to use formal verification tools to determine whether safety properties of interest hold.

3.2.5.1 Simulation

Having a formal model of the system extended with the fault model immediately enables the engineer to simulate different failure scenarios. This is an important facility as the engineers can visualize the effect of faults on system functionality as they control their activation through a graphical user interface. This capability can be used to quickly detect safety problems in common scenarios before performing more rigorous static analysis.

3.2.5.2 Proofs of Safety Properties

Formal verification tools, such as model checkers and theorem provers, can be used to prove that a safety property holds over the extended system model. To prove interesting properties, an engineer will typically have to rule out certain *unlikely* combinations of failures. These can be encoded as assumptions or *axioms* that will be used in the proof process. If a property is proved, then the responsibility of the safety engineer is to review the assumptions that were used in the proof and check if they are realistic. If so, the engineers have a proof that the system satisfies the safety property with respect to the fault model. In case a property is not proved, it may be necessary to rearchitect the system or to relax the original safety property to accommodate delay or other acceptable constraints to allow system recovery.

This capability can also perform exploratory analysis to investigate the fault tolerance of a system; e.g., what is the largest *n* such that the particular safety requirement holds in face of *n* faults? It could also be specialized to a specific combination of faults, say, those combinations whose likelihood is above some reliability threshold (say, 10^{-7} failures / flight hour) rather than random combinations. The safety engineer may also want to investigate how the system behaves in presence of different durations of faults, e.g., permanent and transient faults.

3.2.5.3 Fault Trees

With adequate tool support, the formal verification results could be represented in the form of familiar safety artifacts like fault trees. There is a great deal of interest in this area, but none of the existing tools generate fault trees in a format that is intuitive and amenable for manual review (see Section 5.2).

4 Case Example: The Wheel Brake System

We illustrate the various activities involved in model-based safety analysis with the help of an example of a Wheel Brake System, as described in ARP 4761 – Appendix L [2]. We chose this example primarily because the majority of the safety engineers in the avionics community use the ARP 4761 document as their main reference for safety assessment. By using this familiar example, we hope to make it reasonably easy for engineers to understand the model-based safety analysis approach, and to evaluate the performance of MBSA against manual safety analysis techniques. For illustration of the safety analysis activity (Section 4.2), we use a safety requirement described by ARP 4761. The discussion of the wheel brake system below consists largely of excerpts of the informal requirements from the ARP 4761 document.

The informal wheel brake system diagram taken from the ARP 4761 document is shown in Figure 4. The Wheel Brake System is installed on the two main landing gears. Braking on the main gear wheels is used to provide safe retardation of the aircraft during the taxi and landing phases, and also in the event of a rejected take-off. A secondary function of the wheel brake system is to stop main gear wheel rotation upon gear retraction.

Braking on the ground is either commanded manually, via brake pedals, or automatically (autobrake) without the need for pedal application. The autobrake function allows the pilot to pre-arm the deceleration rate prior to takeoff or landing. When the wheels have traction, the autobrake function will control brake pressure to provide a smooth and constant deceleration.

The eight main gear wheels have multi-disc carbon brakes. Based on the requirement that loss of all wheel braking is less probable than $5*10^{-7}$ per flight, a design decision was made that each wheel has a brake assembly operated by two independent sets of hydraulic pistons. One set is operated from the Green hydraulic supply and is used in the NORMAL braking mode. The Alternate system is on standby and is selected automatically when the Normal system fails. It is supplied by a Blue hydraulic power supply and an Accumulator, both of which can be used to drive the brake. The Accumulator is a simple device with built up pressure that can be reliably released if both of the two primary pumps (the Blue and Green pumps) fail. The Accumulator supplies the Alternate system in the EMERGENCY braking mode.

Switchover between the hydraulic pistons and the different hydraulic sources is automatic under various failure conditions, and can also be manually selected. Reduction of the Green pressure below a threshold value, either from loss of the Green supply itself or from its removal by the BSCU due to the presence of faults, causes an automatic switchover to the Blue supply and the Alternate brake system. If the Blue pump fails, then the Accumulator is used to supply hydraulic pressure.

An anti-skid facility is available in both the NORMAL and ALTERNATE modes, and operates at all speeds greater than 2 meters per second. The anti-skid function is similar to the anti-lock brakes common on passenger vehicles and operates largely in the same manner. In the NORMAL mode, the brake pedal position is electrically fed to a braking computer. This in turn produces corresponding control signals to the brakes. In addition, the braking computer monitors various signals that denote certain critical aircraft and system states to provide correct brake functions and improve system fault tolerance, and generates warnings, indications and maintenance information to other systems. This computer is accordingly named the Braking System Control Unit (BSCU).

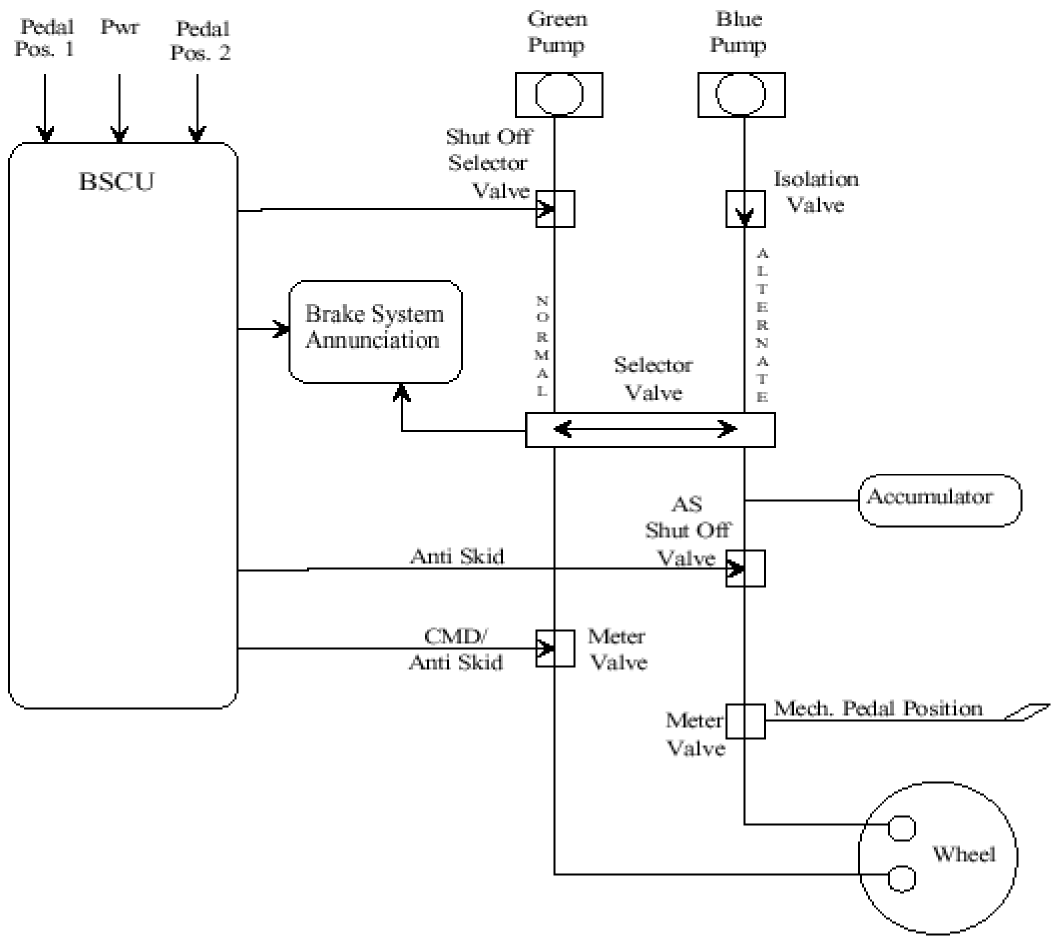

Figure 4: Wheel Brake System Diagram (from SAE ARP 4761)

4.1 Nominal System Modeling

The first step in automating safety analysis is a formal specification of the nominal system model. A formal model typically consists of components (both mechanical and digital) and the interconnections between them.

Figure 5 illustrates how we can model the Wheel Braking System (WBS) in Simulink. The model captures both the digital and the mechanical components of the system and reflects the informal structure of the system as given in the ARP document. As we implemented a formal model of the system, we realized that the informal requirements of the WBS were underspecified, and we had to make several assumptions about the system that still need to be confirmed with the authors of ARP 4761. It is worth noting that even the exercise of building a formal model reveals details that are missing in the informal model of Figure 4. We will point out where (and why) any assumptions about the system are made as we describe the formal model.

WBS (the highest level component/system) consists of a digital control unit, the BSCU, and two hydraulic pressure lines, Normal (pressured by the Green Pump) and Alternate (pressured by the Blue Pump and the Accumulator Pump) line. The system takes the following inputs from the environment – PedalPos1, AutoBrake, DecRate, AC_Speed, and Skid. All of the above inputs are forwarded to the BCSU for computing the brake commands. There are also a number of mechanical components along the two hydraulic lines, for example different types of valves. We have defined a library of common components such as the MeterValve, IsolationValve, Pump, etc., which are then instantiated at various locations in the WBS. The outputs of the WBS are Normal_Pressure (hydraulic pressure at the end of the Normal line), Alternate_Pressure (hydraulic pressure at the end of the Alternate line) and System_Mode (computed by the BSCU).

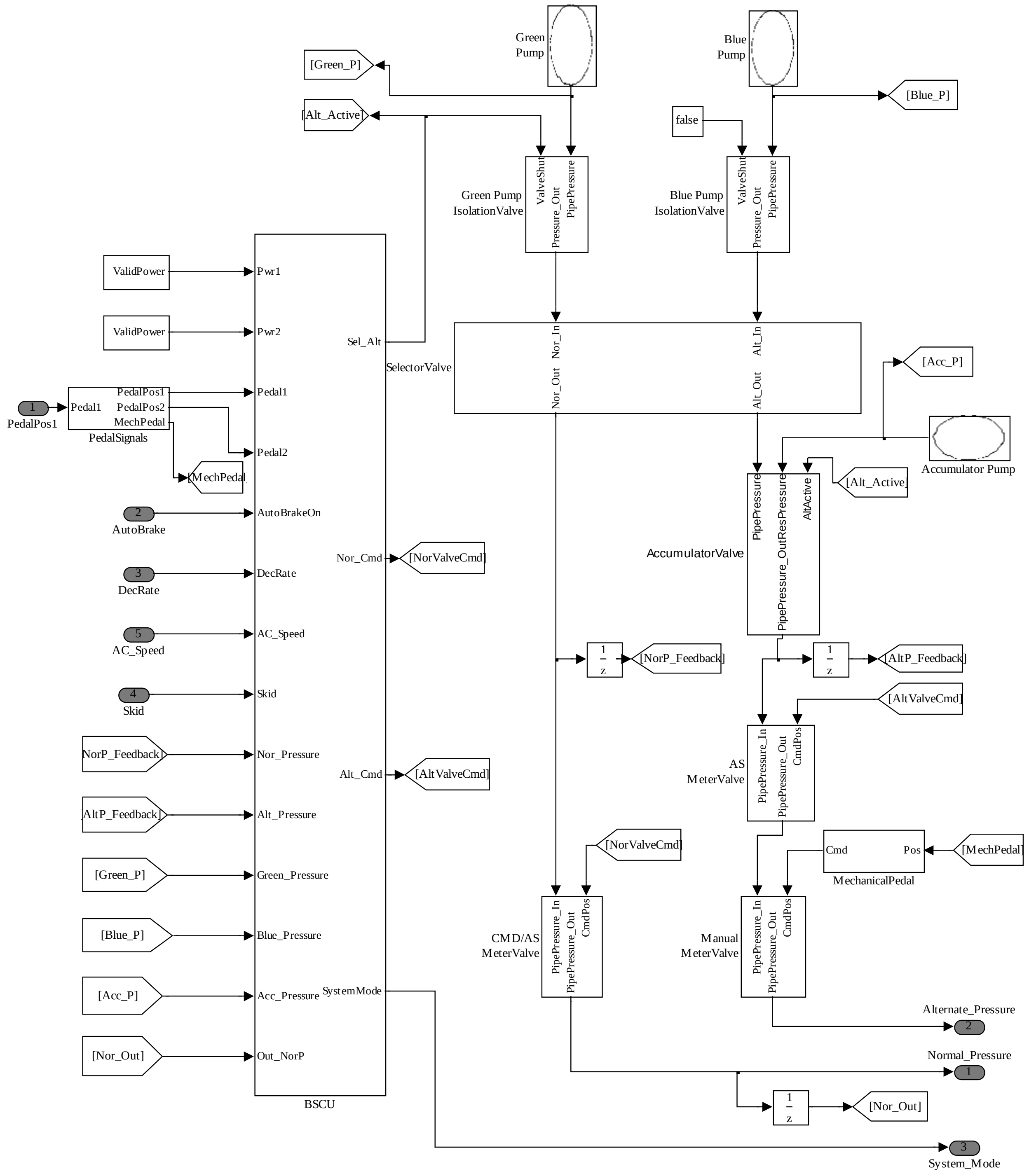

Figure 5 : Simulink model of the Wheel Brake System

4.1.1 Braking System Control Unit (BSCU)

The Braking System Control Unit (BSCU) is the only digital component of the system (Figure 6). Most of the BSCU inputs come from the higher level WBS. It also takes some feedback from different locations along the Normal and Alternate lines, and has two power inputs from separate power sources. The BSCU is composed of two redundant Command and Monitor units. The two subsystems (each containing a Command and Monitor unit) are powered independently by the two power supplies. DecRate (Deceleration Rate) and AC_Speed (Aircraft Speed) are used when AutoBrake is true. In the current model, AutoBrake is implemented by a stub component to which the actual control laws can later be added. Since this functionality is not specified in the informal requirements, we made the simplification that AutoBrake applies constant pressure on the brakes. The pedal position inputs map directly to some pressure value required at the output. When skidding occurs, the BSCU automatically decreases the pressure applied to the brakes.

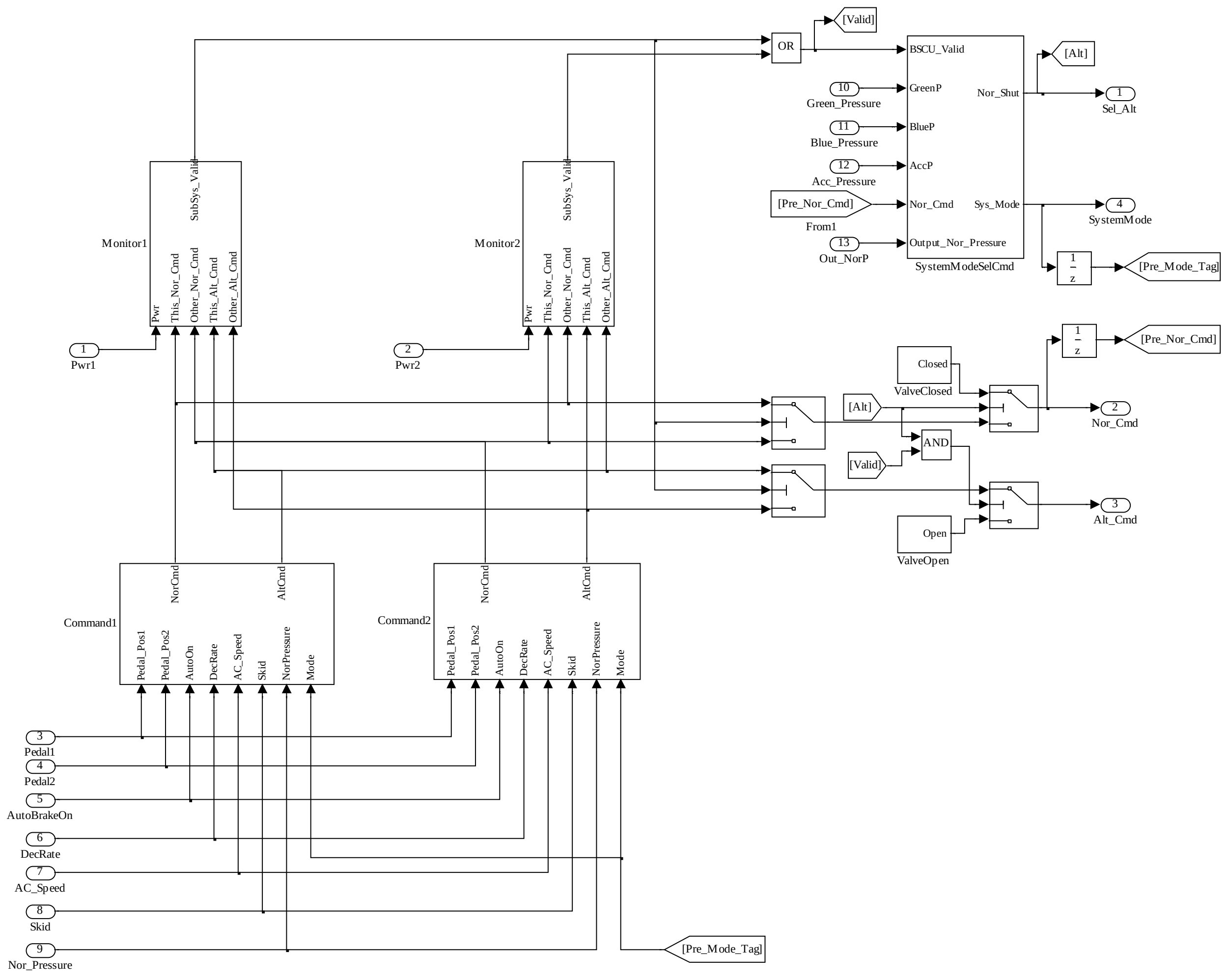

Figure 6 : Braking System Control Unit (BSCU)

The Command unit regulates the pressure to the brakes in the normal line through the normal brake command (Nor_Cmd). The computation of this command takes into account both the requested brake power as well as the skid information. The Command unit regulates the pressure in the alternate line only to prevent skidding; it does this through the (Alt_Cmd). The Monitor unit monitors whether its corresponding Command unit output is valid. When both Command units are valid, the BSCU forwards the commands of the default unit, Command1. BSCU forwards the commands of the valid unit when only one of the Command units is giving valid braking commands. The BSCU is not valid when both of the Monitor units indicate that the corresponding Command outputs are not valid.

The BSCU switches to the Alternate hydraulic system (Sel_Alt = true) under the following conditions:

- The BSCU is not valid, or
- The Green Pump is below threshold, or
- The system was previously in the NORMAL mode and the BSCU had commanded some pressure but the pressure at the Normal line output (feedback Nor_Out) is below the threshold.

Once the system has switched to the Alternate hydraulic system, it will not switch back to the normal hydraulic system.

The SystemMode is considered to be in one of:

- EMERGENCY mode (2), if the Blue Pump or the Accumulator Pump are below the threshold and Sel_Alt = true,
- ALTERNATE mode (1), if Sel_Alt = true, or in
- NORMAL mode (0) otherwise.

4.1.2 Hydraulic Pressure Pumps

There are three instances of the hydraulic pressure pump in the system – the Green Pump, the Blue Pump and the Accumulator Pump. Each pump provides a constant hydraulic pressure (modeled as an integer).

4.1.3 Isolation Valves

There are two instances of the isolation valve – the Green Pump IsolationValve and the Blue Pump IsolationValve. Each isolation valve takes two inputs – the PipePressure and ValveShut. If ValveShut is true, then there is no pressure at the output; otherwise the pressure at the output is the same as the input pressure. In Figure 4, there is no input shown for shutting the isolation valve on the Alternate line (the Blue Pump IsolationValve in our case). We have modeled the Green Pump IsolationValve and the Blue Pump IsolationValve in the same manner, with Blue Pump IsolationValve always getting a constant false value for the ValveShut input (i.e., the Blue Pump is never isolated).

4.1.4 Selector Valve

The SelectorValve is situated across the Normal and the Alternate hydraulic lines. This valve is used to select only one of the two redundant hydraulic systems. In the wheel braking system, we want to prevent a situation where both the Blue and Green system provide pressure to the brakes. The isolation valves in combination with the SelectorValve are designed to prevent this from happening. The SelectorValve takes the two pipe pressures as input and outputs a pressure above the threshold on only one of the two pipes. In the nominal situation, only one of the two input pipe pressures should be above threshold (assured through the two isolation valves). In this case, it would simply select the system with adequate pressure and block the system with no (or low) pressure; functionality that could be achieved through some mechanical implementation of the SelectorValve. From the informal requirement it is unclear how the SelectorValve operates if the pressure on both the incoming pipes is above threshold. We have modeled the SelectorValve such that the default is the Normal system if its pressure is above the threshold. This is another assumption that needs to be confirmed with the authors of ARP 4761.

4.1.5 Accumulator Valve

The AccumulatorValve is a component that we added to the formal model that is not found in the informal diagram and many assumptions about its operation need to be confirmed. The Alternate system is pressurized by the Accumulator Pump when the Blue Pump fails and the system is in the ALTERNATE mode of operation. There must be some mechanism to regulate the pressure provided by the Alternate system through the SelectorValve and the pressure provided through the Accumulator Pump. To accomplish this selection we have introduced the AccumulatorValve. The AccumulatorValve connects the pipes coming from the SelectorValve and the Accumulator Pump, and regulates which one will feed pressure to the downstream system. In addition to the two pipe pressures, the AccumulatorValve also takes the Sel_Alt output of BSCU (renamed as Alt_Active) as input. This signal is used to determine which pressure source to use. The AccumulatorValve will open and select one of the pressure sources *only* when the system is not in the NORMAL mode of operation. When the system is in NORMAL mode of operation, the SelectorValve blocks the pressure on the Alternate pipe.

4.1.6 Meter Valves

There are three instances of the meter valve – the CMD/AS MeterValve on the Normal hydraulic line and the AS MeterValve and the Manual MeterValve on the Alternate hydraulic line. The meter valve implementation takes two inputs – the incoming pipe pressure and the valve position command. The meter valve will adjust the valve position according to the command and the required amount of pressure will be transferred to the output. For example, if the incoming pressure is 100 and the valve position command is ValveHalfOpen, then the pressure at the output will be 50.

The CMD/AS MeterValve and the AS MeterValve take their valve position commands from the Nor_Cmd and Alt_Cmd outputs of the BSCU respectively. The Manual MeterValve takes its valve position command directly from the MechanicalPedal.

4.2 Formalizing the Derived Safety Requirements

After creating the system model, we would like to verify that some basic safety properties hold in this nominal system (i.e., an idealized system containing no faults). As a first step we need to formalize the derived safety requirements as safety properties. The derived safety requirements are determined in the same way as in the traditional "V" process. System hazards are identified through functional hazard analysis. Manual fault tree analysis will be potentially used to derive the initial set of safety requirements. The derived requirements may be at a higher (system) level or lower (component) level as considered appropriate.

We will illustrate the current activity by formalizing an example safety requirement in temporal logic, CTL. An example safety requirement for the wheel brake system as described in ARP 4761 is

> *Loss of all wheel braking (unannunciated or annunciated) during landing or RTO shall be less than $5*10^{-7}$ per flight.*

Since we are not considering annunciations in this model and we are not considering any quantitative analysis at this stage, let us simplify this safety requirement as simply,

> *Loss of all wheel braking during landing or RTO shall not occur.*

To achieve effective braking, the hydraulic pressure at the brake calibers must be above a minimum threshold. The braking pressure can be commanded either through the AutoBrake or the brake pedal. The AutoBrake function only works in the NORMAL mode of operation whereas the brake pedal is capable of commanding pressure in any mode of operation.

Note here that when the wheels are skidding, brake pressure is temporarily reduced or removed to stop the skidding. Based on the observations above, we can derive a safety property suitable for formalization,

> *When the brake pedal is pressed in the absence of skidding, then either the normal pressure or the alternate pressure must be above the threshold.*

To state this formally in CTL, we first define two intermediate variables in SMV to represent whether the pedal is pressed while we are not skidding (`PedP_NoSkid`) and whether any pressure is being provided to the brakes (`SomeP`).

```
PedP_NoSkid := (IsPressed(PedalPos1) & !Skid) ;

SomeP := ((Normal_Pressure > threshold) | (Alternate_Pressure > threshold)) ;
```

`IsPressed` is a predicate that returns true when the pedal is pressed. `PedP_NoSkid` and `SomeP` are then used in a CTL property as:

```
SPEC AG(PedP_NoSkid -> SomeP) ;
```

This property states that it is always globally true (AG) that when the pedal is pressed in the absence of skidding we will get brake pressure. This property can be proven to hold in our nominal system (where no failures occur) in seconds using NuSMV. Of more interest in this report is the behavior in the face of failures discussed in the next section.

It should be noted that this property only checks whether the system is safe in the absence of skidding; if the skid input is incorrectly set to 'true', then the system will incorrectly lower the brake pressure until braking is no longer effective. To determine the safety of the system, it would be necessary to ensure that this signal is correctly generated. This determination would be the responsibility of the safety analyst.

4.3 Fault Modeling and Extension

We now introduce the activities that are specific to the proposed model-based analysis approach. We discuss the fault modeling and extending the system model at the same time as the way one specifies the fault model directly affects the extension. For the WBS, we used Simulink to manually extend the nominal model but found the process slow and error-prone. Based on our experience, in Section 4.3 we suggest how additional tools could improve these steps.

4.3.1 Fault Modeling

We would like to specify different component failure modes, i.e., the way (or form) in which a particular component might fail. This component failure will be triggered by some internal or propagated fault. In order to trigger these faults, we add additional inputs to the extended model for each fault that can occur within a component in the nominal model. Thus, our simple fault model will contain:

1. component failure mode behavior specifications,
2. additional inputs for activating faults
    a. *intrinsic faults* activated through system level inputs, and
    b. *propagated faults* activated by the error propagating component

In the WBS example, we consider a simple fault model for the digital and mechanical components of the WBS. The fault model is implemented by subsystems (i.e. components) with additional inputs that can be used to control whether or not the fault is activated. In our initial example, for simplicity, we will not consider propagated faults.

4.3.1.1 Digital Fault Modeling

Let us consider two sample *digital* failure modes for the BSCU component—the *inverted* failure mode for the two Monitor subsystems and the *stuck* (at previous value) failure mode for the two Command subsystems. The *inverted* failure mode for a Boolean output of the Monitor unit of the BSCU is defined as simply the negation of the input when triggered (Figure 7). In this example, the Fail_Flag triggers the inverted failure. Note that this component can simply be dropped onto the Boolean output line of the Monitor component of the BSCU.

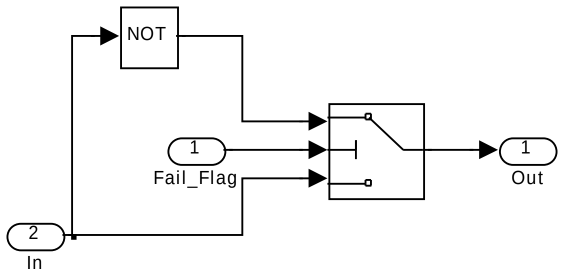

Figure 7: Inverted Failure Simulink Model

The *stuck* failure mode latches the previous value of the output when the Fail_Flag input triggers the failure.

The *Stuck-at* (a particular value) failure mode can be modeled as shown in Figure 8. Although we have not included many digital faults in our prototype model, we envision most, if not all, digital faults to be some form of corruption of the output from the digital component; outputs that are either stuck at some constant value or take on completely random values.

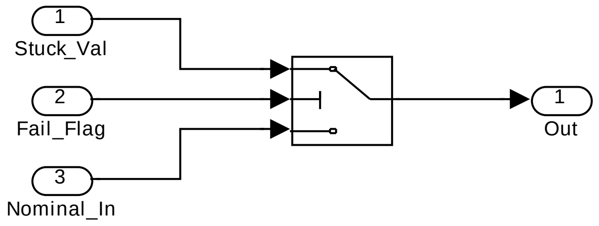

Figure 8: Stuck_at Simulink Model

4.3.1.2 Mechanical Fault Modeling

For the mechanical components, we consider basic failure modes such as a *stuck_at* failure mode for valves, *failure of the pumps* to provide adequate pressure, and the *failure of the power supplies*.

Consider the *stuck_at* failure mode for a valve where it can be stuck either open or closed. This failure model is more complex than a digital failure since the output pressure from the valve when failed open cannot be determined without knowing what the input pressure to the valve is. To model the failure mode suitable for the valves, consider the Binary_Stuck_at Simulink model in Figure 9. In this model the component can be stuck at one of two different values. This model allows us to easily model valves where the valve can either be stuck open or closed; if it is stuck open we output whatever the input pressure to the valve is, if it is stuck closed we output zero pressure.

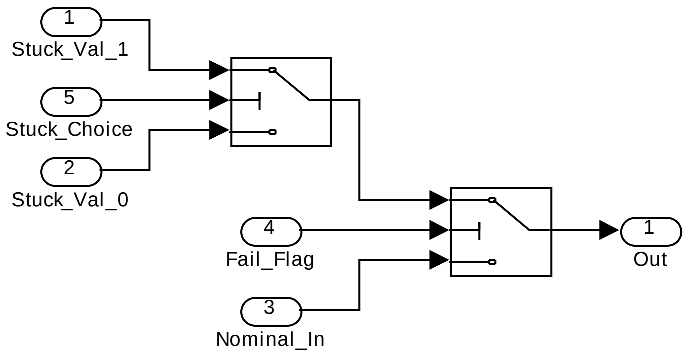

Figure 9: Binary_Stuck_at Simulink Model

4.3.2 Model Extension

In order to analyze the system behavior in presence of faults, we would like to extend the nominal system model with the fault model.

The method for model extension will differ based on the failure mode under consideration. We observe that the Binary_Stuck_at failure mode needs to access the inputs of the original component (Stuck at open assigns the original input PipePressure to the output pressure). This necessitates the failure mode extension in the form of a *wrapper* around the original component, as it needs to access the original input. The extension of the MeterValve component with the Binary_Stuck_at failure mode is shown in Figure 10. In the figure, the MeterValve is the nominal component implementing the meter valves as described in the previous section and the Stuck_at component is the Binary_Stuck_at discussed above. When Stuck_Choice is 1 we model a valve that is stuck open and the input pressure is forwarded as is to the output irrespective of the Cmd, and when Stuck_Choice is 0 we model a valve that is stuck closed and the output pressure is set to 0.

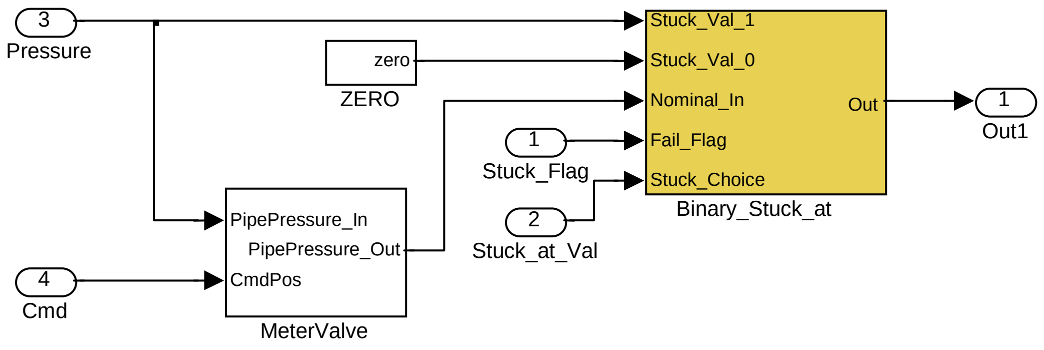

Figure 10: MeterValve_Stuck Simulink Model

Alternatively, a simple failure mode, like *inverted output* for a component generating a boolean output, can be simply added at the output signal of the affected component. There is no need to wrap the failure mode behavior around the original component since the failure mode behavior does not depend on the inputs to the original component. Another failure mode that does not need wrapping is power failure (as shown in Figure 11) that can directly operate on the output of the affected component, i.e., the PowerSupply.

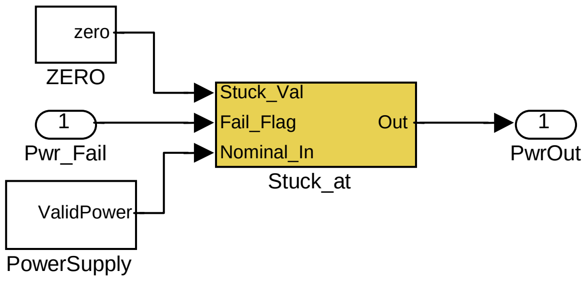

Figure 11: Power_Fail Simulink Model

Once the failure modes are manually inserted in the nominal model, we need to add new inputs and new connections to activate the faults which may consequently lead the extended components to fail. For the activation of independent and transient (or intermittent) faults, new inputs are added to the system model (at the topmost level). For example, all the valve components, extended by the Binary_Stuck_at failure mode, have two additional inputs: Stuck_Flag and Stuck_Val. The rest of the failure modes require a single input signaling the occurrence of a fault. For the activation of permanent faults, latched inputs (permanently active once activated) are added to the system model. In the case of fault propagation and dependent faults, there will be addition of more data paths to propagate faults (backward propagation, simultaneous propagation, delayed propagation, etc.).

After extension, the model is considerably larger and more cluttered due to the additional inputs needed to activate the possible faults, as shown in Figure 12 and Figure 13. Figure 12 shows the fault inputs (shaded) added to the system to control when the faults get triggered. To reduce clutter, “goto” Simulink tags are used to route the fault triggers to the corresponding component without actually drawing signal lines. Figure 13 shows the rest of the system. The shaded components are the mechanical components extended with failure modes. The Simulink “from” tags supply the fault inputs to the components from Figure 12.

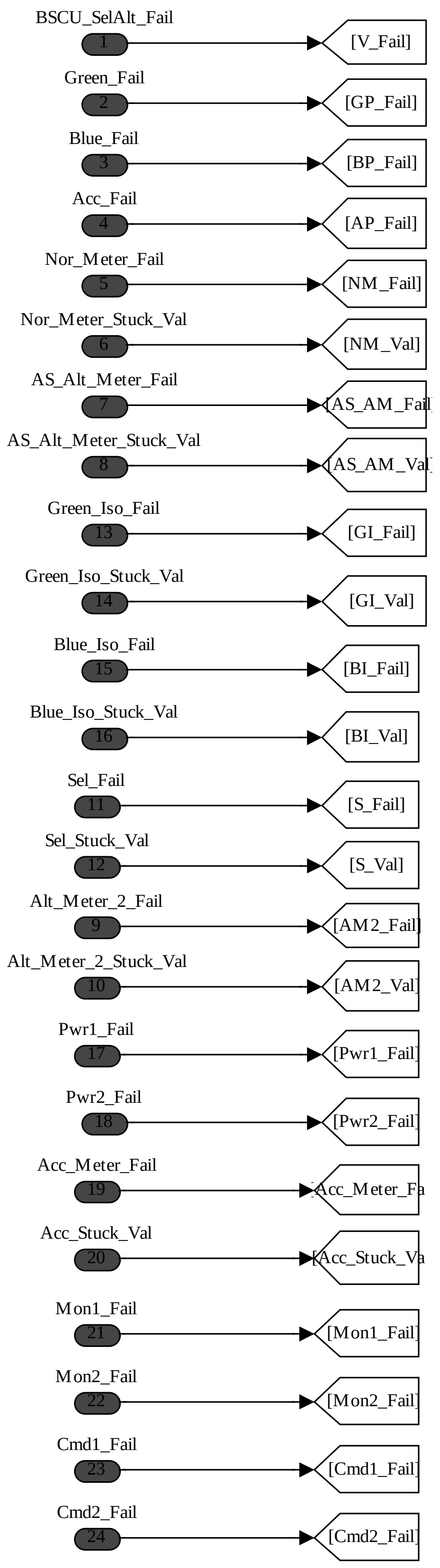

Figure 12 : Fault Trigger Inputs of the Extended Wheel Brake System

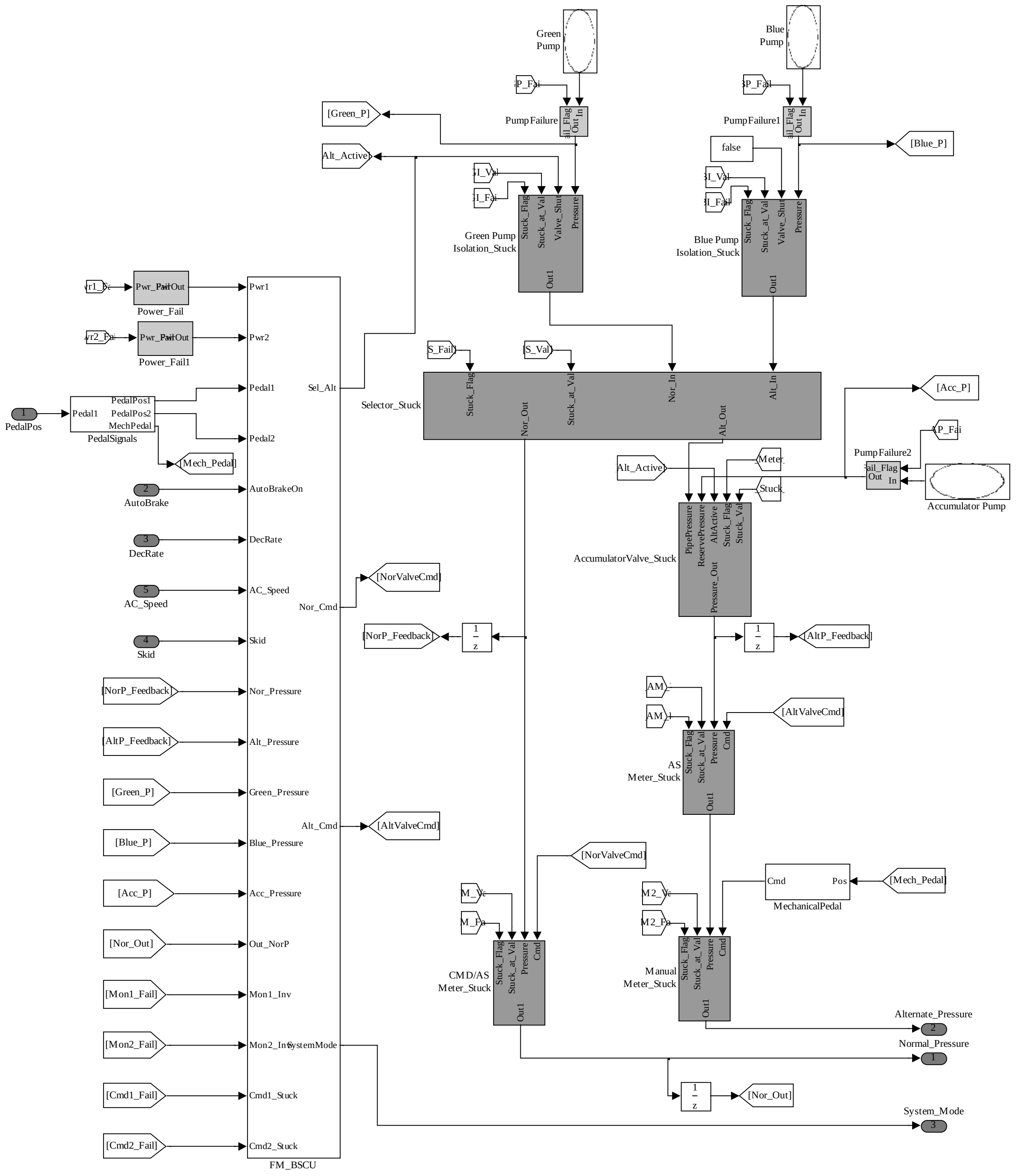

Figure 13 : Wheel Brake System Extended with Fault Model

4.3.3 Fault Modeling and Extension Issues

In the previous sections we used Simulink to illustrate how one could model the behavior of failure modes using existing modeling constructs. Our fault model was quite simple, consisting of only the definitions of component failures. Also, we only considered independent faults. Even so, we can identify several issues and shortcomings with using the existing tools for these activities.

Clutter: As noted in the previous paragraphs, even for simple fault models, the extended model is cluttered with considerable fault information, making it difficult to keep track of the original system functionality in presence of these faults. For more complicated fault models, say with fault propagations and dependent faults, additional data paths would be needed, adding even more clutter.

Manual Extension: Even for simple fault models, manually extending the nominal behavior with the fault behavior is *error-prone*. The manual model extension also leads to *model evolution* issues. If changes to the system model are required, systems engineers will have to make these changes in the context of a cluttered model including faults or the safety engineers will have to redo their fault modeling in the updated model produced by the systems engineers; both highly undesirable overhead.

Lack of Flexibility: Here we discuss some of the flexibility issues in the existing tools:

> **Composite Failure Modes:** To add more flexibility to the fault model, one might want to specify all possibly ways in which a component could fail. This means that there could be a number of different failure modes associated with a component. To do this with the existing tools, an engineer has to manually compose all applicable failure modes for a given component to create a *composite failure mode* which takes into account conflicting behaviors, priorities, etc. This composite failure mode can then be composed with the nominal behavior of the component. Since this composite failure mode will be different for different types of components, an engineer would have to construct many such composite failure modes.
>
> **Duration of the Fault:** Not only does a fault have a behavior, but it also has a duration. Broadly, we distinguish between *permanent* and *transient* faults. Permanent faults are straightforward to model, but for *transient* faults, we also have to consider the duration of the fault. For some classes of faults, this duration is parameterized depending on the component that the fault is applied to and the model of time used for the system model. Using existing techniques, it is difficult to create generic faults whose duration can be parameterized appropriately.
>
> **Fault Hierarchies and Dependencies:** One might also want to specify fault hierarchies for the system. For example, we might want to define the failure mode on the BSCU based on the failure modes of the underlying Monitor and Command units.
>
> We would also like to express fault propagation and other kinds of dependent faults flexibly. For example, if a pipe bursts in the WBS, this affects the pressure of both the

downstream pipes and upstream pipes. Due to the dataflow in the WBS system architecture, the failure of a pipe will automatically be propagated downstream. However, as there is no dataflow in the upstream direction, there is no way to propagate loss of pressure to these components in the system model. One solution is to propagate this failure by describing additional fault propagation connections to the upstream pipes in the fault model.

The fault model could also identify other dependent faults such as *common mode failures.* These are faults that simultaneously affect a number of components that may not be explicitly connected in the system model. For example, in the WBS, a number of components in the system might be supplied power by the same power supply. Failure of this power supply would lead to failure of all these components. The nominal system model might not even mention the power supply, since it is not necessary to describe the nominal behavior of the system. But the fault model will need to take this common mode failure into account.

4.3.4 Proposed Approach: Aspect-oriented Technique

We view the nominal model that captures the system functionality and the fault model as conceptually distinct. For example, in a model-based development approach, the nominal model of software is used for code generation to derive the implementation of the software. If fault modeling is integrated into these components, then it is no longer possible to generate desirable code for these components. Also, having an integrated fault model makes other MBD activities such as test-case generation and formal analysis of nominal model behavior more difficult.

In addition, integrating the fault model into the system model leads to problems in the creation and evolution of the extended system model. Even with an extremely simple fault model, the fault information can dwarf the description of the nominal behavior within an extended system model, leading to problems in system understanding, maintenance, creation, and evolution. Manually adding a single fault to a component to the system model may require several additional inputs to the top-level model and modifications to several components to “wire” the fault information to the appropriate place within the model. This step is further complicated if we wish to describe fault propagation or composite faults. Finally, we often want to separately evolve the system and fault models, for example, to easily introduce or modify faults into a stable system model. In short, if the fault model is not separated, it is extremely difficult and error-prone to manage the evolution of the combined model.

We believe that it is critical to have the ability to separate the fault model from the system model and provide flexible options for combining the two models to perform meaningful safety analysis. By keeping the functional system model and the fault models separate and automating the composition, we can (1) keep the individual models simpler and more focused, and (2) reduce the possibility of introducing errors while manually composing the original functionality with the failure functionality.

In addition to separation of the fault model from the nominal model, there must be support for *flexible fault modeling*. Having a notation that is specifically targeted towards fault modeling will promote ease of specification of complex fault behaviors, such as fault propagations and hierarchies, allowing the engineer to create realistic models for precise safety analysis.

Due to these shortcomings in using the existing tools for model-based safety analysis, there is a need to extend the existing framework to support separation of the fault and system models, flexible modeling of the fault model, and automatic system model extension. In this framework, the system and safety engineers can separately formalize the nominal system model and the fault model, which can be then automatically composed to form an extended model suitable for safety analysis. This extension should be performed at the modeling language/tool level so that the engineers can simulate the extended model in addition to performing formal analysis.

The fault model is not intrinsic to the basic functionality of the system, but is an artifact required for the safety analysis and defines the failure behavior of the system. We observe that the fault model affects various components of the system in different ways. This can be essentially seen as cross-cutting the system functionality – an *aspect* of the system – which can be woven into the nominal system only when required for safety analysis. Aspect oriented programming (AOP) [21], [22], is a recent technique that makes it possible to clearly express programs with crosscutting concerns, or aspects, including appropriate isolation and composition of the aspect code. Using the AO-technique, one can specify the components that implement the basic system functionality in the component language (component program), the crosscutting aspects in an aspectual language (aspect program), and define an aspect weaver that composes the two to give an extended component program. *We hypothesize that there is a natural application of these aspect-oriented techniques to resolve some of the research issues identified in fault modeling and model extension.* A fault model can be thought of as an aspect of the original system and, consequently, we can view model extension as aspect-weaving. We believe that aspect-oriented techniques can be successfully applied in the formal modeling domain and that fault modeling and model extension can be considered as a natural instance of this application. Given aspect-oriented tool support, we hope to achieve (1) separation of the fault model from the system model, (2) flexible specification of the fault model, and (3) sophisticated and flexible composition of the two models to create an extended system model.

4.4 Formal Safety Analysis

After extending the model with faults and failure modes, we want to check whether the safety property holds in the face of component failures. As mentioned in Section 3.2.5.2, there are two ways to perform this analysis – 1) one can either prove the safety property without constraining the number of faults that can occur in the system, or 2) one can prove the safety property after constraining it to some maximum number (*k*) of faults. In the first case, it will probably be necessary to assert that certain unlikely combinations of faults will not occur for the proof to go through. After ruling out all the unlikely combinations of faults, if the proof goes through, then the system adequately satisfies the safety property. In the second case, we restrict the safety property such that it will only consider *k* combinations of faults. If this property is satisfied, the engineer will get a proof that the safety property is satisfied for all combinations of *k* faults.

In this section we describe an example safety analysis we can perform on the extended wheel brake system model. We describe the system fault tolerance verification, in which we investigate if the system can handle some fixed number of faults.

4.4.1 Fault Tolerance Verification Using Model-Checkers

We want to investigate what is the maximum number of faults that the system can recover from and still satisfy the (relaxed) safety requirement. We would like to explore the effects of both *transient* and *permanent* faults on system fault tolerance.

First, let us attempt to verify that our safety requirement holds in the presence of at most one transient fault at any point in time.

> *In the presence of at most one transient fault, when the brake pedal is pressed in the absence of skidding, then either the normal pressure or the alternate pressure shall be above the threshold.*

For this example, we again formalize our safety properties in SMV. To make it easier to specify properties we extend our model to compute the number of faults triggered in the *current step* (given by `NumFails`). To flexibly formalize the notion of *at most n* faults, we introduce a variable, *k*, in NuSMV with range 0..n. We define the next relation for *k* such that it keeps its previous value. Thus *k* has some non-deterministic assignment in the initial state (a model checker considers all possible initial values in the range) and then holds that value constant.

We first formalize the notion of correct behavior of the system in a particular state. In CTL, this can be defined as:

```
DEFINE
   CorrectBraking := ((NumFails = k & k <= 1 & PedP_NoSkid) -> SomeP);
```

This definition states that if there is at most one fault occurring in the current step (`NumFails`) and if the pedal is pressed in the absence of skidding, then we will get some pressure at the output in the same step. We can formalize the property over all states CTL (using the intermediate variables defined in section 4.2) as follows:

```
SPEC AG (CorrectBraking);
```

As may be expected, this property does not hold and NuSMV returns a counterexample indicating that as soon as a critical component fails (e.g., the green pump) we will instantly lose pressure at the brake calipers. The underlying problem is that the system needs time to discover and react to the failure. To account for this interval, we introduce a delay into our property to give the system chance to recover,

```
SPEC AG (ABF 0..1 CorrectBraking)
```

This property introduces the `ABF` operator, which is a real-time CTL operator supported in NuSMV [19]. The `ABF 0..1` specification states that the `CorrectBraking` property must either hold immediately or in the following step (alternately, the property `CorrectBraking` can be false for no more than one step). Given this formulation, NuSMV comes back with a counterexample where the Green Isolation Valve fails. The only way for the system to detect this feedback is through the pressure feedback after the Meter Valve along the Normal line where there is a step delay in the model (Figure 14).

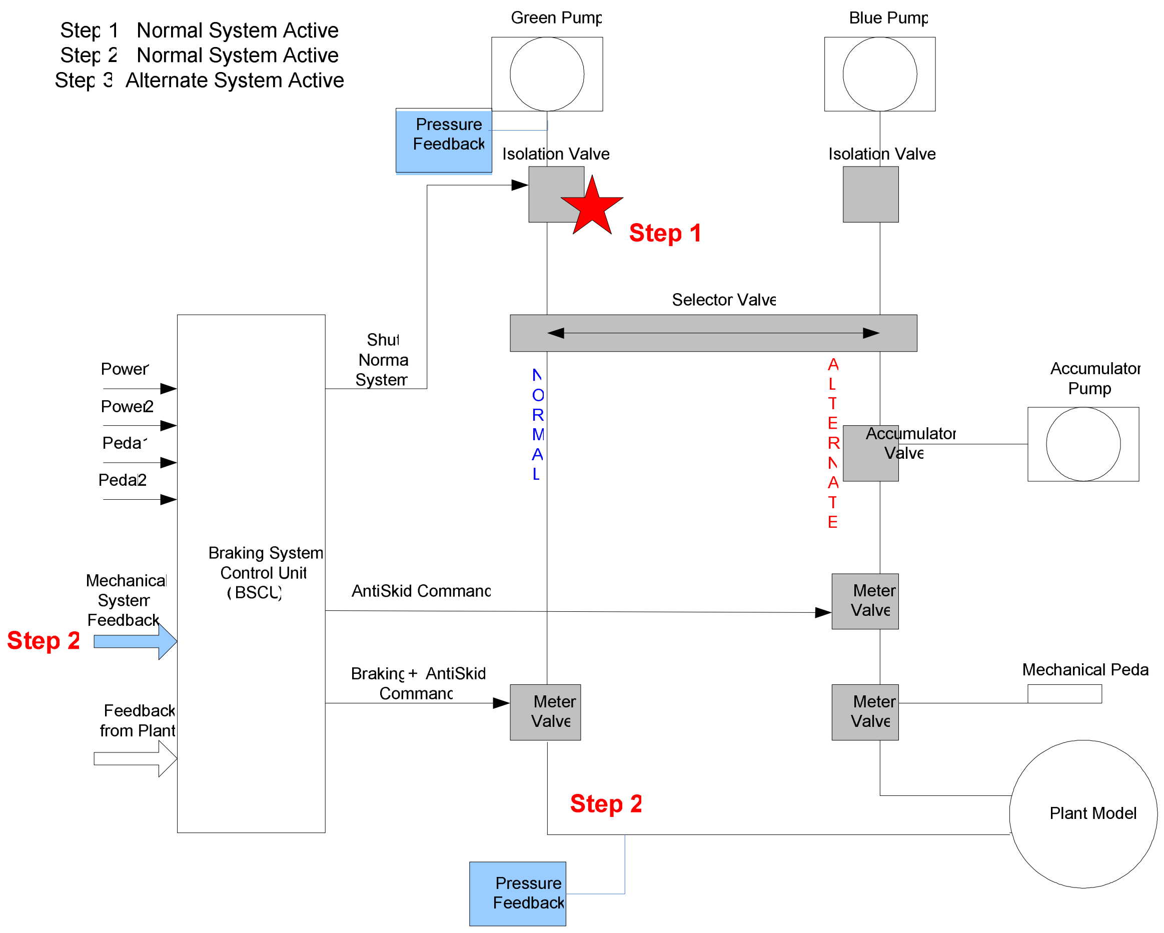

Figure 14 : Counter-example for a downstream fault requiring additional delay

From the counterexample, it is clear that we need to allow the system time to detect failures located on the Normal system and switch to the Alternate system. We deem this delay acceptable and refine our property to reflect this additional delay.

```
SPEC AG (ABF 0..2 CorrectBraking);
```

This property states that if there is a single fault and the pedal is pressed in the absence of skidding for three consecutive time steps, then we will get pressure at the brakes by the third step (i.e., the property `CorrectBraking` can be false for no more than two steps). Nevertheless, verification of this relaxed safety property is still not possible, as illustrated by the scenario shown in Figure 15: If there is some transient failure (e.g., the Green pump fails) then the BSCU will detect this failure and switch over to the Alternate system powered by the Blue pump. In this version of the WBS, the switchover to the Alternate system is not reversible.

Even if the fault that caused the switchover is transient and is repaired, the system will not switch back to the Normal hydraulic system. In our counterexample, even if the transient fault recovers, the active hydraulic system will still be the Alternate system. Now, if, for example, some meter valve along the Alternate system fails closed (stuck at closed), then the system cannot recover from this failure and will not generate any braking pressure.

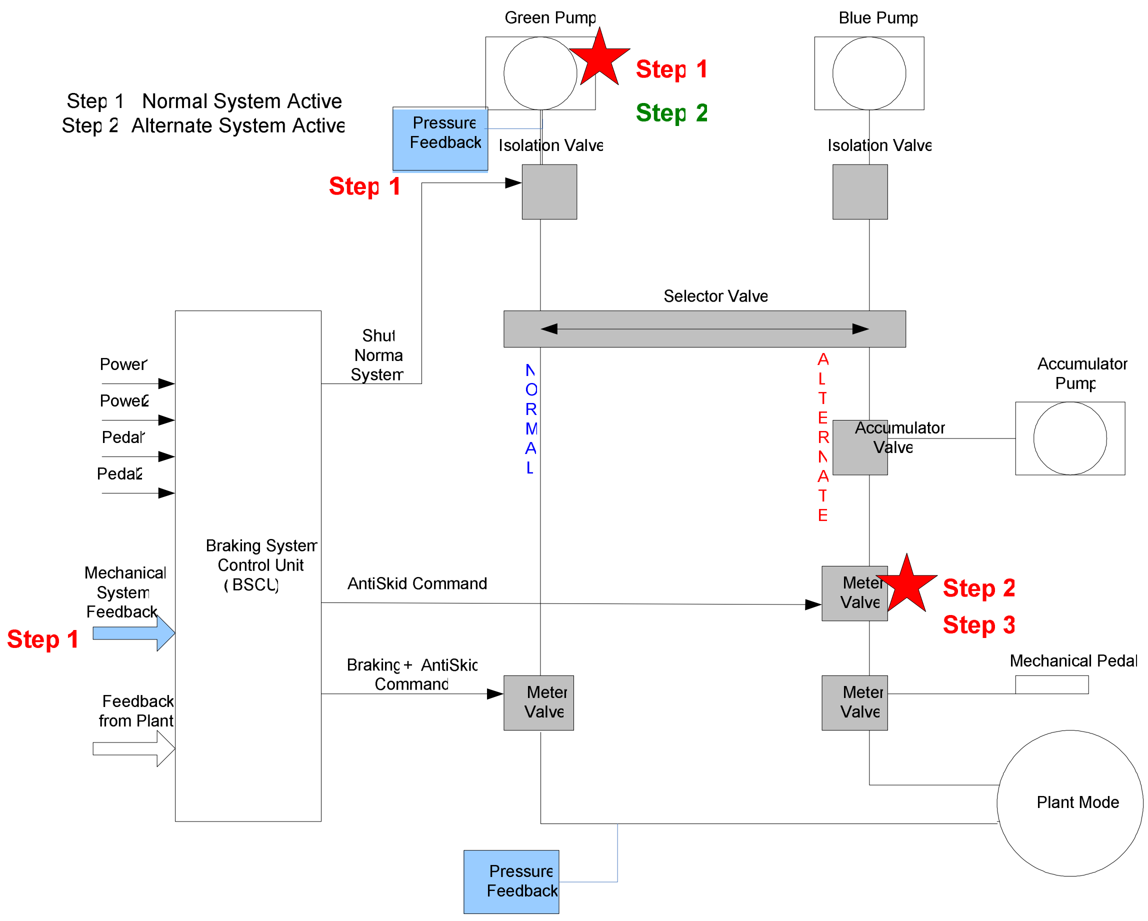

Figure 15 : Counter-example for two transient non-overlapping faults

The issue is that even though it took *two* transient faults to cause the loss of braking pressure, there was never more than *one* fault at any particular instant in time. After examining the system, we realize that the original formulation of the property cannot hold if faults are allowed to “migrate” between different components.

We realize that the real measure of interest is not the number of *current* failures, but the *total number of failures over the course of operation*. To capture this notion, we introduce a new variable (`TotalFails`) that will compute the total number of faults triggered until the current step. `TotalFails` considers only rising edges of faults, i.e., a fault input was false in the previous step and true in the current step. Thus, a persistent fault will only be counted once, regardless of how long it lasts. In the previous failure scenario, `TotalFails` will count two failures even though we never have more than one failure at any one instance in time.

We now redefine `CorrectBraking` to use TotalFails instead of NumFails as follows:

```
DEFINE
   CorrectBraking := ((TotalFails = k & k <= 1 & PedP_NoSkid) -> SomeP);
```

and given the property:

```
SPEC AG (ABF 0..2 CorrectBraking);
```

NuSMV now verifies that the property is satisfied. As noted before, `TotalFails` captures both scenarios for single transient and persistent (and permanent faults). We can conclude that our system can recover from any single transient or permanent failure. However, the system is not tolerant to two (or more) failures. In this case, NuSMV returns the counterexample described in Figure 15.

4.4.2 Formal Safety Analysis Issues

Currently the computation tools like model checkers do not generate results in the form of traditional safety artifacts, like fault trees. The result is either the property is true, or the property is false with a counterexample. There is research [8], [9], [27] that has begun to address turning counterexamples generated by model checkers into fault trees, but the current results are unacceptable for real safety analyses for reasons discussed in Section 5.2.1.

Ideally, we would like to represent the safety analysis results in the form of a traditional fault tree with all possible fault combinations encoded in a way that reflects the architecture of the system. We can then use existing fault tree analysis tools (cf. [4], [5], [35]) to compute the probability of the top level event and check whether it is within acceptable limits.

Alternately, the tool can rule out combinations of faults that are highly unlikely (based on some probability estimates) and only analyze for possible fault combinations. This can be performed as an iterative process, starting with no constraints. If counterexamples are returned, the user rules out certain combinations of faults and re-runs the tools. The end result that the tool will produce will be proof of the degraded safety property, in the presence of user specified constraints.

Also, current notations (such as temporal logic) used for describing complex safety properties are not very familiar to practicing engineers nor straightforward to use for specifying degraded properties. More support is needed for capturing complex properties for verifying system fault tolerance. Due to the size and complexity of the models, we often operate at the limit of the capabilities of automated tools, such as model checkers. Scaling up the models to significantly larger systems will require additional research into techniques involving model abstraction and partitioning, or the use of manually guided tools such as theorem provers like PVS.

4.4.3 Proposed Approach for Fault Tree Generation using PVS

Theorem proving is another method for performing verification on formal specifications of system models. Theorem provers (for example, PVS [37]) apply rules of inference to a specification in order to derive new properties of interest. Rather than exploring the global state space, theorem provers automate human reasoning, reducing a proof goal (with human guidance) to simpler sub-goals that can be discharged automatically by the primitive axioms or decision procedures of the prover.

Given a property and a model, the user is either able to verify the property by completing the proof or is presented with unproven subgoals that describe scenarios in which the property is violated. Theorem proving is generally harder than model checking and requires considerable technical expertise and understanding of the specification as well as the theorem proving environment. On the other hand, the process of creating a proof is an excellent way to gain insight into the specification. One of the major disadvantages of using theorem provers is that the prover does not help the user to determine if a proof is failing because the property is unprovable or the user is not providing the right steps to complete the proof.

Proof trees correspond closely to fault trees (see Figure 16). In many ways, the process of constructing a proof tree is similar to the construction of a fault tree. The safety requirements will guide the formulation of safety properties and Top Level Events (TLE), in case of fault trees, for the system or subsystem under consideration. In a fault tree, the system engineers encode all the combinations of failures that will make the top level event occur. While proving a safety property, the engineer will have to rule out bad scenarios (in the form of the unprovable sub-goals) with the help of assumptions or axioms that enable the proof to succeed. These assumptions will potentially encode all the failure combinations which could cause the safety requirement to fail. These assumptions can be checked later to see if they satisfy the probability constraints.

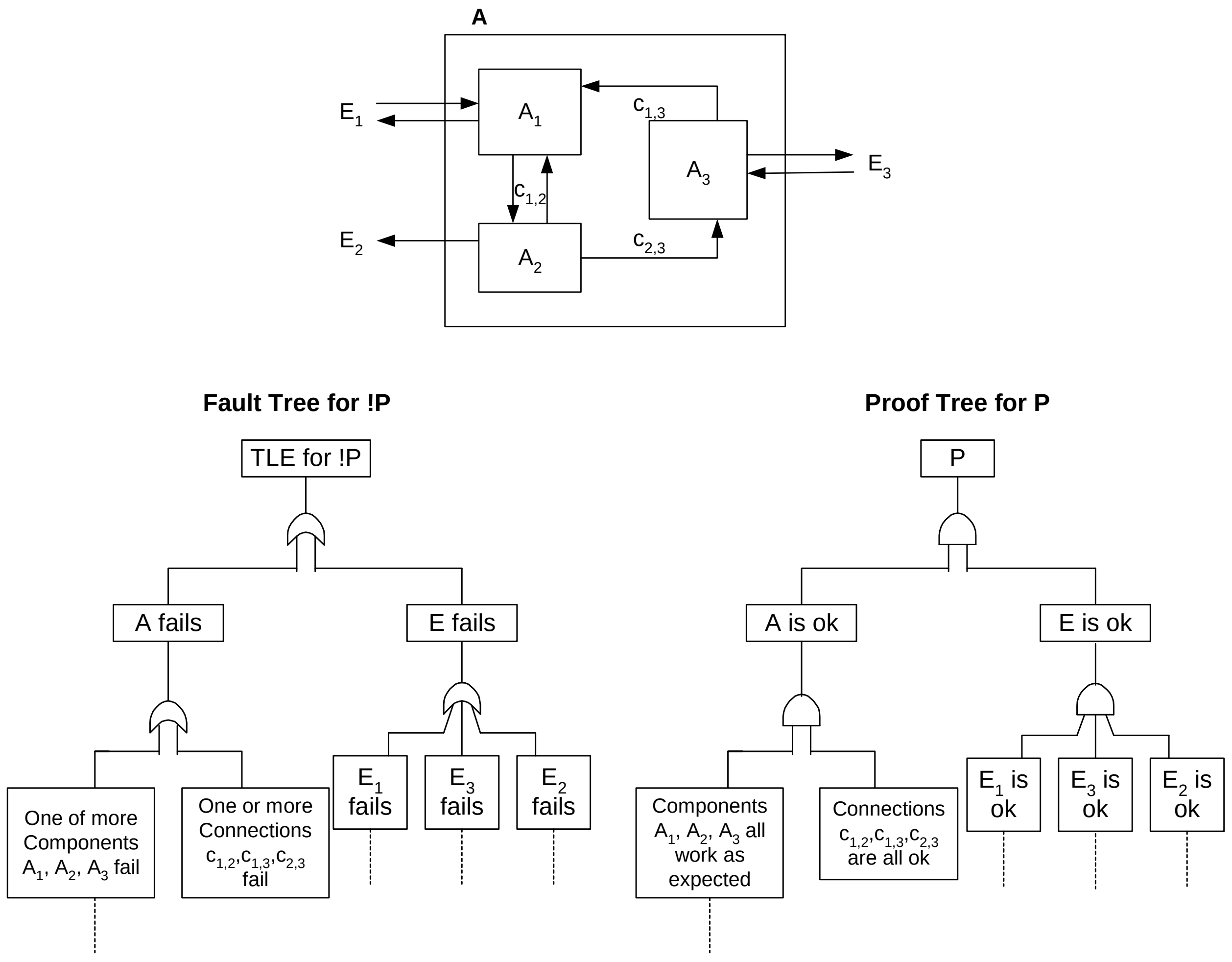
A
E1
E2
E3
A1
A2
A3
c1,2
c1,3
c2,3
Fault Tree for !P
TLE for !P
A fails
E fails
One of more Components A1, A2, A3 fail
One or more Connections c1,2,c1,3,c2,3 fail
E1 fails
E3 fails
E2 fails
Proof Tree for P
P
A is ok
E is ok
Components A1, A2, A3 all work as expected
Connections c1,2,c1,3,c2,3 are all ok
E1 is ok
E3 is ok
E2 is ok

Figure 16 : Correspondence between Fault Trees and Proof Trees

4.5 Summary

In this section, we illustrated the model-based safety analysis activities on a wheel brake system example using existing commercial modeling tool, Simulink [13], and verification tool, NuSMV [19]. We demonstrated how simulations and formal analysis can expose interesting failure scenarios even in case of a simple model. We identified important research challenges that need to be resolved in order to make the model-based safety analysis approach practical.

5 Related Work

This section discusses related work in automating safety analysis. We start by discussing architectural specification languages and how they may be useful for safety analysis. Following that, we look at research more specifically related to safety analysis. Since fault trees are considered one of the most important safety artifacts, most of the existing work in the field has been in fault tree generation and analysis. We discuss some existing tools for automated fault tree generation (in conjunction with fault injection) and fault tree analysis. We finally discuss some integrated formalisms for modeling and capturing safety information.

5.1 Architecture Specification Languages

There are several architectural specification languages that are already in use or are proposed for use in industrial applications. In this section, we discuss a few of the better-known languages.

5.1.1 Architecture Analysis and Design Language (AADL)

The *Architecture Analysis and Design Language* (AADL) is an SAE standard [34] specifically targeted to the design and analysis of the software and hardware architecture of performance-critical real-time systems. The language is used to describe the structure of systems as an assembly of software components mapped onto an execution platform. The language can describe functional interfaces to components (such as data inputs and outputs) and performance-critical aspects of components (such as timing). The language also describes how components interact, such as how data inputs and outputs are connected and how application software components are allocated to execution platform components. The language can also describe adaptable systems through the use of operational modes and mode transitions.

This standard does not specify how the detailed design or implementation details of software and hardware components are to be specified. Those details can be specified by a variety of software and hardware description languages. The relevant design and implementation characteristics are specified as AADL component properties, and as rules of conformance between the properties and the described components.

AADL is designed to be extensible to accommodate analyses of the runtime architectures that the core language does not completely support. Extensions to accommodate new analyses and unique hardware properties take the form of new properties and analysis specific notations that can be associated with components. Some annexes may be proposed to be added to the standard. There is a proposed annex that will provide support for specifying *error models*.

In the context of this work, we can use synchronous languages like RSML^{-e} or Lustre (SCADE) to specify the component implementations and AADL to specify the architecture. We can use properties associated with each component to specify safety requirements. Work is already underway on an error model annex that may be suitable for specifying failure modes of components. AADL supports specification of both the logical and physical architecture, which could be used to separate logical and physical faults.

5.1.2 EAST-ADL

EAST-ADL [28] is a language for the modeling and development of software based systems, with its primary domain of application in automotive systems. The language has been developed within the project EAST-EEA by representatives of European automotive industries and academic research sites. EAST-ADL is intended to capture all information needed for the development, from early analysis to implementation and evolution, and meets specific automotive requirements such as support for automatic code generation in the context of common automotive hardware.

The EAST ADL is structured into 7 layers, each layer only loosely coupled through requirements entities and associations. The EAST ADL abstraction layers are – *Vehicle View* (describing user visible features), *Functional Analysis Architecture* (capturing the behavior and algorithms of the Vehicle View functions), *Functional Design Architecture* (representing a decomposition of functionality in the Functional Analysis Architecture to meet constraints regarding allocation, efficiency, etc), *Function Instance Model*, *Hardware Architecture*, *Platform Model* (which models the operating system or middleware) and *Allocation Model* (which contains configuration information needed for mapping of application software to hardware). A progression through these models is implicit, but as there is overlap between the architectures, the path through them can be adapted to the needs of the different domains and companies. However, the language defines the artifacts in a unique and consistent way.

The description of the language elements is divided into parts corresponding to different language domains: the structure specifying structural relation, the behavior describing behavioral models, the requirements modeling requirements (functional and non functional ones) and their relations to other entities, the V&V elements describing entities related to testing and verification.

5.2 Automated Safety Analysis Tools and Techniques

This section discusses some of the related tools and techniques specifically proposed for automated safety analysis.

5.2.1 FSAP/NuSMV-SA

FSAP/NuSMV-SA is a tool for automating the generation of fault trees. FSAP/NuSMV-SA [8], [9], [10], [14] is based on two components – FSAP (Formal Safety Analysis Platform), which provides a graphical front-end through the Safety Analysis Task (SAT) manager, and the NuSMV2 model checker, which provides the safety analysis engine. FSAP/NuSMV-SA requires the system model to be specified in NuSMV. FSAP/NuSMV-SA has support for failure mode definition and model extension through automatic failure injection.

5.2.1.1 Fault Model

FSAP/NuSMV-SA provides certain predefined failure modes – *stuck_at* (stuck at a particular value), *inverted* (boolean value gets inverted), *non_determinism* (non-deterministic, i.e. random, value), *ramp_down* (integer value decreases by a fixed amount each step, down to a bottom value), and *glitch* (wrong, random value for a limited number of steps). The user can specify the failure mode of a particular input/output by selecting one of the failure modes from the pre-defined list of failures or the user-defined failures. The failures are assumed to be *non-deterministic* in the time and order they may fail. With the exception of *glitch,* they are also *persistent* failures, i.e. once the failure occurs, it will exist from then on.

5.2.1.2 Automatic Fault Injection and Model Extension

After the failure modes are defined, the user can automatically inject the failures in the system model to create a new extended model. The extended system model adds degraded behavior to the original system corresponding to the failure modes defined. This model can then be used for safety assessment of the system.

5.2.1.3 Automated Fault Tree Analysis

A significant advantage of an automated analysis tool like FSAP is that it removes the burden of manually creating fault trees once the system and the fault model are specified. This ensures that the system and safety engineer work off the same models and assumptions.

FSAP uses model checking to perform fault tree analysis [12], [27]. In this analysis, one describes a potential system failure, or Top Level Event (TLE) to be analyzed. Rather than generate a counterexample describing that failure, FSAP uses exhaustive state-space analysis to identify all sets of basic events, which may cause that TLE to occur. Thus, the tool will automatically extract all collections of basic events, i.e. all minimal cut sets, which can cause the given TLE. It creates all the minimum sets of basic events that cause the failure to occur independently, ensuring that all the events not affecting the TLE will not be a part of a minimum cut set. This gives a more exact and complete analysis than a manual fault tree analysis.

NuSMV-SA also provides a trace for each minimal cut set it generates. The trace shows how the TLE is reached, given a particular configuration of failures determined by the minimal cut set. FSAP/NuSMV-SA can also automatically perform event ordering analysis. Specifically, given a TLE and a minimum cut set, it will find out whether there are any ordering constraints, which hold between the pair of basic events in the cut set. Traditionally FTA is a static analysis; using FSAP we can investigate influence of fault modes in dynamic situations.

5.2.1.4 Discussion

Though FSAP is a very powerful tool, it has disadvantages, which might limit its applicability to practical systems. A fault tree generated by FSAP has a *flat* structure (see Figure 17) – the structure of the generated fault trees is an “or-and” structure, i.e. it is a disjunction of all the minimum cut sets, with each minimum cut set being a product of basic events. Thus the tree is only two levels deep and can be very broad. A fault tree generated by a traditional manual analysis is usually more intuitive to read, as the analyst creates the fault tree, which corresponds to the structure of the system. This is an important concern as it hampers the understanding of the fault trees, and in turn the acceptance of the tool by the safety engineers.

Another issue is that FSAP (or any automatically) generated fault trees are completely reliant on the correctness and completeness of the system and fault model. If all possible failures are not considered, or if the component specifications do not correspond to the behavior of the real system, then the fault trees will be incomplete or inaccurate. Manual creation of fault trees also serves as a human review that helps to catch such errors.

Finally, in FSAP most faults (except glitch) are considered to be permanent when they occur. There is no flexibility in defining the fault model – no good way of specifying fault propagation, simultaneous/dependent faults, and persistent/intermittent faults. The user cannot control triggering of faults; it is performed non-deterministically by the tool. An additional point to note is that there is no way to simulate faults since the fault injection is performed at a lower level in NuSMV. The fault injection is straight-forward – there is no *wrapping* of failures as discussed earlier.

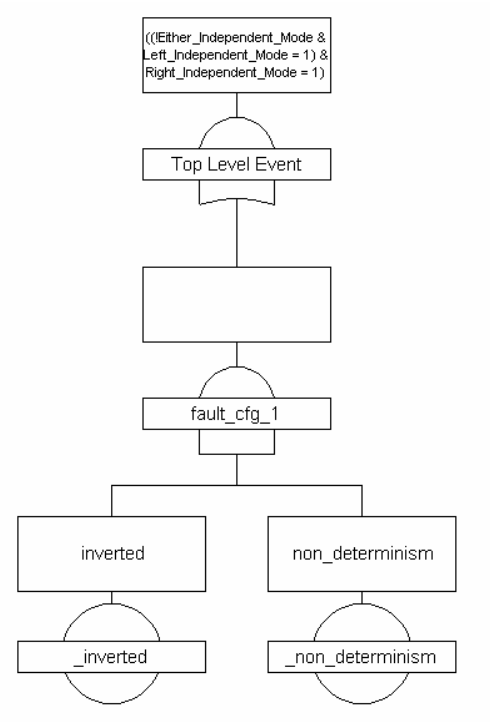
((!Either_Independent_Mode &
Left_Independent_Mode = 1) &
Right_Independent_Mode = 1)
Top Level Event
fault_cfg_1
inverted
non_determinism
_inverted
_non_determinism

Figure 17: Example Fault Tree Automatically Generated by FSAP/NuSMV-SA

5.2.2 Galileo – Dynamic Fault Tree Analysis Tool

Galileo [4], [39] is a dynamic (and static) fault tree *modeling* and *analysis* tool that incorporates DIFTree analysis methodology. Dynamic fault trees extend traditional (static) fault trees to enable modeling of fault-tolerant systems in which failure modes can depend on the ordering of component and sub-system failures and can include cascading and common cause failures (functional dependencies). DIFTree (Dynamic Innovative Fault Tree) analysis methodology [5] combines static and dynamic fault tree analysis techniques using a modular approach.

5.2.2.1 Fault Tree Creation

Galileo enables engineers to *edit* and *display* fault trees in textual and graphical form through widely used, commercially-supported components that are easy to integrate into real engineering practice.

5.2.2.2 Fault Tree Analysis Capabilities

Galileo performs reliability analysis on the constructed tree. The tool supports *coverage* modeling in static and dynamic trees. Failure probabilities in static trees may be constant (time-independent) or follow the exponential distribution. Dynamic fault trees only support exponential distribution of time to failure. A basic event is characterized by a failure probability and a coverage factor.

DIFTree is a hybrid technique that supports automatic decomposition, analysis and integration of partial results. During traversal, a subtree is marked as dynamic if a dynamic gate is present, otherwise it is marked as static. The static fault subtrees are solved by automatic conversion to the equivalent BDD, while dynamic subtrees are solved by automatic conversion to the equivalent Markov model. Each submodel is solved for the probabilities of covered and uncovered failures and is replaced by a basic event in the higher-level model.

The tool also supports a Monte-Carlo simulation engine that uses variance reduction techniques for the analysis of reliable systems.

5.2.2.3 Discussion

Since Galileo is a fault tree analysis tool, not fault tree generation tool, any automated fault tree generation tool can be used in conjunction with it. Given a qualitative fault tree (generated automatically from automatic fault tree generation tools like FSAP or our proposed tools), it can be imported into Galileo, which can then use it to do the quantitative analysis by plugging in the actual probabilities. Galileo is also useful for managing modular analysis of large systems.

5.2.3 HiP-HOPS

HiP-HOPS (Hierarchically Performed Hazard Origin and Propagation Studies) [31] is a method for safety analysis originating from a number of classical techniques such as Functional Failure Analysis (FFA), Failure Mode and Effects Analysis and Fault Tree Analysis. The method enables integrated assessment of a complex system from the functional level through to the low level of component failure modes. Though the HiP-HOPS process starts early in the design lifecycle with exploratory FFA, we describe its use only after we have a hierarchical model of the system (following the FFA). HiP-HOPS is currently supported by a tool called the Safety Argument Manager (SAM).

5.2.3.1 Fault Model

As the refinement of the system hierarchical model proceeds, the failure behavior of components in the model is analyzed using a modification of classical FMEA called Interface Focused-FMEA (IF-FMEA). The application of this technique generates a *model of the local failure* *behavior* of the component under examination, which is represented as a table. The table provides a list of component failures modes as they can be observed at the component outputs and for each such output failure, it determines the causes as a logical combination of *internal malfunctions of the component* or *deviations of the component inputs.* An IF-FMEA table records how a hardware or software component reacts to failures generated by other components. In addition, the table determines the failure modes that the component itself generates or propagates to other components.

5.2.3.2 Fault Tree Generation

Once the local failure behavior of all components is determined, we can determine how the functional failures that have been identified in the exploratory FFA arise from combinations of the low-level component failure modes that we have identified in the IF-FMEAs. In HiP-HOPS, this is achieved mechanically by synthesizing fault trees.

A fault tree is generated incrementally by parsing the expressions contained in the IF-FMEA encountered during hierarchical traversal. The fault tree structure records hazardous dependencies between components in the model (caused by data flow, functional dependencies between sub-systems and components). Input deviations received by each component are substituted by corresponding output failures by other components. The tool performs minimal cut-set analysis and probabilistic calculations on the fault tree.

5.2.3.3 Discussion

The fault model consists of the standard failure modes, such as stuck at, at the base level components (lowest level in the hierarchy). In addition to the intrinsic component failures, fault propagation is explicitly considered wherever relevant at the component outputs. The user has to explicitly construct the IF-FMEA table which will be later used in fault tree generation. One of the strong points of this approach is that the hierarchical structure of the system is captured neatly in the fault tree.

The component behavior is not considered while generating the fault trees – the fault tree seems to be just a hierarchical representation of what the user defined in the IF-FMEA table. We observe that there is no fault order dependency information, which will be extremely important in systems using synchronization. The fault trees do not contain additional trace information like in FSAP. We also observe that there is no direct support for formal analysis – it would depend on the modeling notation chosen.

5.2.4 Altarica – Language with support for Fault Modeling

The Altarica language [6], [7] was designed to formally specify the behavior of systems when faults occur. An Altarica model can be assessed of means of complementary tools such as a fault tree generator and a model-checker.

5.2.4.1 Fault Model

An Altarica model of a system consists of hierarchies of components called nodes. A node gathers flows, states, events, transitions and assertions. A failure can be defined as an event which can affect the state of the node. A transition is characterized by a guard, an event name and command part (assignment to some state variable). A failure mode can be defined using a transition which takes the particular failure event.

5.2.4.2 Fault Tree Generation

Altarica fault tree generator takes as input an Altarica model and some unexpected event and generates a fault tree for the non-temporal failure conditions. Once the fault tree is generated, a fault tree analyzer, ARALIA, can be used to compute the set of prime implicants of the non-temporal failure conditions.

5.2.4.3 Discussion

Since we do not have experience in using Altarica, we refer the reader to a short experience paper [7] that discusses the advantages, disadvantages and limitations of using Altarica for safety analysis. Some key points made in the paper are as follows. It was found to be difficult to model certain types of failure propagations in Altarica – e.g. propagating failures in both directions, upstream and downstream without adding additional delays. Also, Altarica does not differentiate between transient and permanent faults.

6 Conclusions and Future Directions

In this report, we introduced *Model-Based Safety Analysis*, an approach for automating portions of the safety analysis process using executable formal models of the system. We believe that this approach has several benefits when integrated into safety analysis processes:

- A tighter integration between systems and safety analysis based on common models of system architecture and failure modes.
- The ability to simulate the behavior of system architectures early in the development process to explore potential safety hazards.
- The ability to exhaustively explore all possible behaviors of a system architecture with respect to some safety property of interest using automated analysis tools.
- The ability to automatically generate many of the artifacts that are manually created during a traditional safety analysis such as fault trees and FMEA/FMECA charts.

Furthermore, this approach is based on existing commercial tools and techniques that are increasingly used for systems and software engineering for safety-critical systems.

Nevertheless, there are several research challenges that must be addressed before the full benefits of this approach can be realized. The first involves construction of the model: which languages and tools are most applicable and how much detail is necessary? The second involves the analysis tools: can they be scaled to the point of analyzing realistic systems for relevant properties? The third involves the user-interface and presentation issues: can we make the tools straightforward for system and safety engineers to use? Can we take the results of these formal analyses and turn them into artifacts that can be easily understood and used by system and safety engineers? Can these artifacts be used for certification credit?

Our goal in the short term was to use existing tools on industrially relevant examples to determine the actual problems and needs of safety engineers. Given this experience, we can set a realistic, grounded research agenda to improve the process in the future. To this end, we have modeled the Wheel Brake System example from ARP 4761 – Appendix L [2] in this report (Section 4). This example was chosen primarily because the ARP 4761 document is used as the main reference for safety assessment by a majority of the safety engineers in the avionics community, and this example is complex enough to illustrate many of the concerns that occur in practice. Further, it contains a detailed safety assessment of the system, which provides a benchmark with which we can compare the results of our automated analyses.

We modeled the system using the Simulink [13] modeling language (Section 4.1). As part of this model construction process, we had to make several assumptions about the behavior of the system in question that affect the safety of the system. This process of discovering “hidden” assumptions when constructing the model may be beneficial to the analysis process, as these models require that system behaviors be unambiguously specified. We were then able to exhaustively analyze one of the safety properties called out in the ARP document using the NuSMV model checker [19] (Section 4.4.1).

In the process of creating the Simulink models and performing the safety analysis activities, we discovered several shortcomings in regard to flexibility in modeling faults, fault injection, performing safety analysis, and formatting of the analysis results. Each of these deficiencies provides interesting avenues for future work.

6.1 Fault Modeling and Model Extension

The current process of creating fault models and injecting these faults into the system model is cumbersome, and it significantly clutters the model of the nominal behavior of the system with additional failure inputs. Such models can be difficult to create, difficult to read, and difficult to update as the system evolves.

In Section 4.3.4, we describe an alternate approach that separates the fault model from the nominal system behavior model. Using *aspect-oriented programming* [21], it may be possible to create a library of reusable faults that can be applied to several different mechanical or digital components within a system. Using this approach, it would becomes straightforward to add or modify faults without the tedious redrawing of the model that is currently required, and it allows more flexible fault models to be specified. Ideally, this process would be integrated into the GUI of existing tools, allowing the systems engineer to choose from a palette of pre-defined faults (or create specialized faults) and drag them onto existing system components. We believe this is possible to achieve with some existing tools, but this would require significant development effort.

6.2 Notations for Describing System Safety Properties

Current notations (such as temporal logic) for describing complex safety properties are not very familiar to practicing engineers or straightforward to use for specifying complex properties. More support is needed for capturing complex properties for verifying system fault tolerance.

6.3 Presenting Safety Analysis Results

Currently, the results generated by model checkers and theorem provers do not correspond to the expected artifacts of safety analysis. There is research that has begun to address turning counterexamples into fault trees, but the current results are unacceptable for real safety analyses for several reasons, as discussed in Section 5.2.1.4. To better fit existing safety analysis guidelines we need to be able to present analysis results in familiar forms, such as fault trees, in ways that better map to current safety analysis practice. In Section 4.4.2, we describe an approach using PVS to create fault trees that may yield intuitive and well-structured fault trees derived from the formal model.

6.4 Scaling the Formal Analysis Tools

Due to the size and complexity of the models, we often operate at the limit of the capabilities of automated tools, such as model checkers. Scaling up the models to significantly larger systems will require additional research into techniques involving model abstraction and partitioning, or the use of manually guided tools such as theorem provers, e.g., PVS [37].

Acknowledgements

The authors wish to thank Ricky Butler, Kelly Hayhurst, Celeste Bellcastro, and Carrie Walker of the NASA Langley Research Center for their ongoing support of this work. The authors also wish to thank Michael Peterson of Rockwell Collins Inc. for his invaluable and expert feedback.

7 Bibliography

[1] ARP 4754. Certification Considerations for Highly-Integrated or Complex Aircraft Systems. SAE International, November 1996.

[2] ARP 4761. Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment. SAE International, December 1996.

[3] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. Dependable and Secure Computing*, Volume 1, Number 1, pages 11-33, 2004.

[4] J. Bechta, B. Dugan, K. J. Sullivan, and D. Coppit. Developing a High-quality Software Tool for Fault Tree Analysis. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 222-31, Boca Raton, Florida, 1-4 November 1999.

[5] J. Bechta, B. Dugan, B. Venkataraman, and R. Gulati. DIFTree: A Software Package for the Analysis of Dynamic Fault Tree Models. In *Annual Reliability and Maintainability Symposium 1997 Proceedings*, Philadelphia, PA, 13-16 January 1997.

[6] P. Bieber, C. Castel, and C. Seguin. Combination of Fault Tree Analysis and Model Checking for Safety Assessment of Complex System. In *Proc. 4th European Dependable Computing Conference*, Volume 2485 of LNCS, pages 19-31, Springer- Verlag, 2002.

[7] P. Bieber, C. Bougnol, C. Castel, J. P. Heckmann, C. Kehren, S. Metge, and C. Seguin. Safety Assessment with Altarica - Lessons Learnt Based on Two Aircraft System Studies. In *18th IFIP World Computer Congress, Topical Day on New Methods for Avionics Certification*, Toulouse France, 26 - 26 August 2004. IFIP.

[8] M. Bozzano, A. Cavallo, M. Cifaldi, L. Valacca, and A. Villafiorita. Improving Safety Assessment of Complex Systems : An Industrial Case Study. In *Proceedings of Formal Methods 2003*, Volume 2805 of LNCS, Springer-Verlag, pages 208-222, 2003.

[9] M. Bozzano & A. Villafiorita. Integrating Fault Tree Analysis with Event Ordering Information. In *Proceedings of ESREL 2003*, pages 247-254, Maastricht, The Netherlands, 15-18 June, 2003.

[10] M. Bozzano & A. Villafiorita. Improving System Reliability via Model Checking: the FSAP / NuSMV-SA Safety Analysis Platform. *In Proceedings of SAFECOMP 2003*, pages 49-62, Edinburgh, Scotland, United Kingdom, 23-26 September, 2003.

[11] E. M. Clarke, O. Grumberg, and D. A. Peled. Model Checking, The MIT Press, Cambridge, MA, 2001.

[12] O. Coudert and J.C. Madre. Fault Tree Analysis: 10^{20} Prime Implicants and Beyond. In *Proc. Annual Reliability and Maintainability Symposium*, 1993.

[13] J. Dabney and T. Harmon. Mastering Simulink. Prentice Hall, Upper Saddle River, NJ, 2004.

[14] FSAP/NuSMV-SA. http://sra.itc.it/tools/FSAP/

[15] Esterel Technologies. SCADE Suite Product Description, http://www.esterel-technologies.com.

[16] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Dataflow Programming Language Lustre. In *Proceedings of the IEEE*, Volume 79, Number 9, pages 1305-20, September 1991.

[17] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, M. Trakhtenbrot. STATEMATE: A Working Environment for the Development of Complex Reactive Systems, *IEEE Transactions on Software Engineering*, 1990.

[18] IEEE Std. 610.12-1990, Standard Glossary of Software Engineering Terminology.

[19] IRST: http://nusmv.irst.itc.it/ The NuSMV Model Checker, IRST, Trento Italy

[20] A. Joshi and M. P.E. Heimdahl. Model-Based Safety Analysis of Simulink Models using SCADE Design Verifier. In *Proceedings of SAFECOMP 2005*.

[21] G. Kiczales, E. Hilsdale, J. Hungunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the Fifteenth European Conference on Object-Oriented Programming*, Number 2072 in LNCS, Springer Verlag, June 2001.

[22] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, JM. Loingtier, and J. Irwin. Aspect Oriented Programming. In *Proceedings of the Eleventh European Conference on Object-Oriented Programming (ECOOP'97)*, Number 1241 in LNCS, pages 220–242, SpringerVerlag, June 1997.

[23] J.-C. Laprie. Dependable Computing and Fault Tolerance: Concepts and Terminology. *International Symposium of Fault-Tolerant Computing*, Volume 1, Number 1, pages 2-11, June 1995.

[24] N. G. Leveson, Safeware: System Safety and Computers, Addison-Wesley Publishing Company: Reading Massachusetts, 1995.

[25] N. G. Leveson. Software Safety: Why, What, and How. ACM Computing Surveys, Volume 18, Number 2, June 1986.

[26] N. G. Leveson, M. P.E. Heimdahl, J. Reese. Designing Specification Languages for Process Control Systems: Lessons Learned and Steps to the Future. In *Proceedings of Foundations on Software Engineering*, 1999.

[27] P. Liggesmeyer and M. Rothfelder. Improving System Reliability with Automatic Fault Tree Generation. In *Proc. 28th International Symposium of Fault-Tolerant Computing (FTCS)*, 1998, Munich, Germany.

[28] H. Lönn , Tripti Saxena , Mikael Nolin , and Martin Törngren. FAR EAST: Modeling an Automotive Software Architecture Using the EAST ADL. In *ICSE 2004 workshop on Software Engineering for Automotive Systems (SEAS)*, Edinburgh, 2004.

[29] MIL STD 882C, System Safety Program Requirements, 19 January 1993.

[30] S. Miller, M. P.E. Heimdahl, and A.C. Tribble. Proving the Shalls. In *Proceedings of FM 2003: the 12th International FME Symposium*, Pisa, Italy, 8-14 Sept., 2003.

[31] Y. Papadopoulos, J. A. McDermid, HiP -HOPS: Hierarchically Performed Hazard Origin and Propagation Studies. In *SAFECOMP '99*, Toulouse, LNCS 1698, pages 139-152, Sept. 1999

[32] B. Randell. On Failures and Faults. *In Formal Methods (FME)*, Volume 2805 of LNCS, pages 18-39, Springer-Verlag, 2003.

[33] RTCA DO-178B. Software Considerations in Airborne Systems and Equipment Certification. DEC 1992.

[34] SAE. The SAE AADL Standard: A Language Overview: An Overview of the SAE AADL Language - extracted from the draft standard document. http://la.sei.cmu.edu/aadlinfosite/LinkedDocuments/AADLLanguageSummary.pdf

[35] SAIC, Inc. CAFTA - Fault Tree Analysis. Product Brochure. http://www.saic.com/products/software/cafta/cafta_brochure.pdf

[36] A. Schafer. Combining Real-Time Model Checking and Fault Tree Analysis. In *Proceedings of the International Symposium of Formal Methods Europe*, Pisa, Italy, LCNS 2805, September 2003.

[37] SRI International: http://pvs.csl.sri.com. The PVS Specification and Verification System, SRI International.

[38] SRI International: http://sal.csl.sri.com. The Symbolic Analysis Laboratory, SRI International.

[39] K. J. Sullivan, Joanne Bechta Dugan, and David Coppit. The Galileo Fault Tree Analysis Tool. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*, pages 232-5, Madison, Wisconsin, 15-18 June 1999.

[40] System Safety Society. System Safety Analysis Handbook 2nd Ed. July 1997.

[41] A. Tribble, S. Miller, and D. Lempia. Software Safety Analysis of a Flight Guidance System. NASA Contractor Report CR-2004-213004, March 2004 http://techreports.larc.nasa.gov/ltrs/dublincore/2004/cr/NASA-2004-cr213004.html.

[42] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. Fault Tree Handbook, Washington, D. C., 1981. NURGE-049.

[43] M. Whalen. A formal semantics for RSML^{-e}. Master's thesis, University of Minnesota, May 2000.