Model-Based Safety Analysis of Simulink Models Using SCADE Design Verifier[*]

Anjali Joshi and Mats P.E. Heimdahl

Department of Computer Science and Engineering, University of Minnesota
200 Union St SE, Minneapolis, MN 55455, U.S.A.
Phone: 1 612 624 7590, Fax: 1 612 625 0572
{ajoshi,heimdahl}@cs.umn.edu

Abstract. Safety analysis techniques have traditionally been performed manually by the safety engineers. Since these analyses are based on an informal model of the system, it is unlikely that these analyses will be complete, consistent, and error-free. Using precise formal models of the system as the basis of the analysis may help reduce errors and provide a more thorough analysis. Further, these models allow automated analysis, which may reduce the manual effort required.
The process of creating system models suitable for safety analysis closely parallels the model-based development process that is increasingly used for critical system and software development. By leveraging the existing tools and techniques, we can create formal safety models using tools that are familiar to engineers and we can use the static analysis infrastructure available for these tools. This paper reports our initial experience in using *model-based safety analysis* on an example system taken from the ARP Safety Assessment guidelines document.

1 Introduction

Traditionally, safety engineers manually perform analyses, such as fault tree analysis, based on informal design models and requirements documentation. Unfortunately, these analyses are highly subjective and dependent on the skill of the practitioner. We hypothesize that by redirecting the effort to build models of the system under study and its fault model we can both reduce the effort involved and increase the quality of the analysis. To this end, we propose a *model-based safety analysis* process in which engineers create formal models for both the system design and safety analysis, and use automated analysis tools to analyze their behavior. We describe our early experience towards this goal in this paper.

Our approach is to adapt model-based development techniques using formal modeling languages and tools such as SCADE [9] and Simulink [5] for safety analysis. By integrating these tools into safety analysis, it is possible to create system models that can be simulated and analyzed using a variety of static analysis techniques. This combination allows an analyst to quickly explore different "what-if" scenarios on combinations of faults using simulation, and also

[*] This work has been partially supported by NASA contract NCC-01-001.

allows formal verification of different aspects of fault tolerance and, potentially, autogeneration of safety analysis artifacts such as fault trees.

We describe our preliminary experiences using model-based safety analysis with a wheel brake system example adopted from ARP 4761 [1], a standards document for safety analysis in the avionics industry. With the help of this example, we illustrate how we can derive benefits from a model-based safety analysis in a practical setting using existing tools. At the same time, this exercise exposes several issues and shortcomings that need to be addressed to make formal safety analysis acceptable in practice.

2 Safety Assessment Process

The overall safety assessment process that is followed in practice in the avionics industry is described in the SAE standard ARP 4761 [1]. Our summary in this section is largely adopted from ARP 4761.

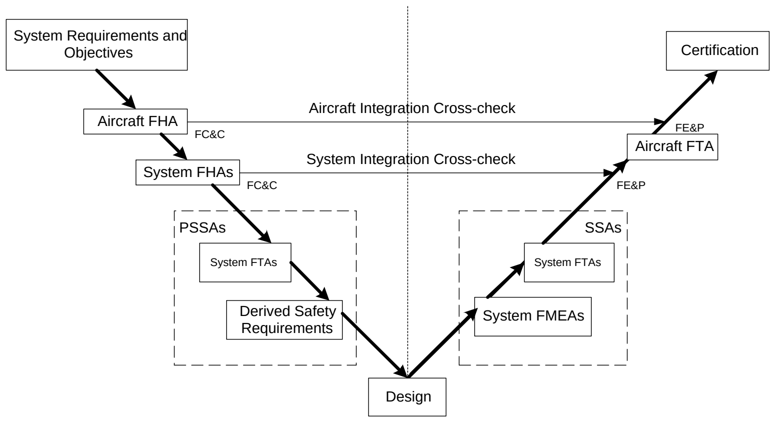
System Requirements and Objectives
Certification
Aircraft FHA
Aircraft Integration Cross-check
FC&C
FE&P
Aircraft FTA
System FHAs
System Integration Cross-check
FC&C
FE&P
PSSAs
SSAs
System FTAs
System FTAs
Derived Safety Requirements
System FMEAs
Design

Fig. 1. Traditional "V" Safety Assessment Process

Figure 1 shows an overview of the safety assessment process as recommended in ARP 4761. The process includes safety requirements identification (the left side of the "V" diagram) and verification (the right side of the "V" diagram), that support 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 completed through quantitative and qualitative analysis of the fault trees created for the implemented design, first for the subsystems and then for the integrated aircraft.

We propose to modify this traditional "V" process so that the lower level PSSA and SSA activities are performed based on a formal model of the system under consideration. Figure 2 shows the modified "V" diagram for model-based safety analysis. The shaded blocks are those activities that will be modified or added.

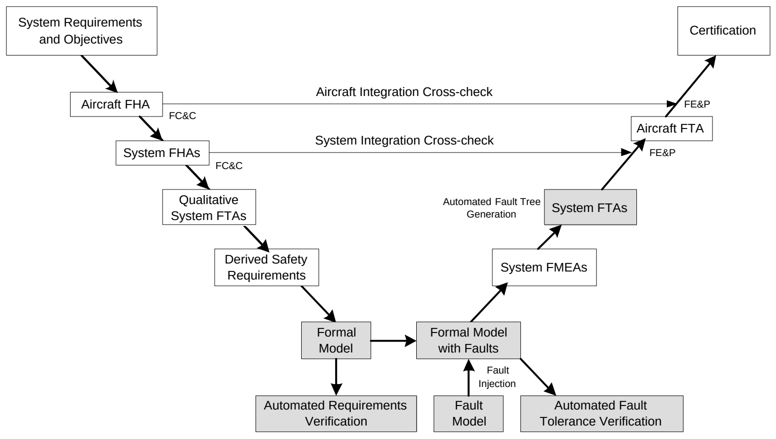
System Requirements and Objectives
Certification
Aircraft Integration Cross-check
Aircraft FHA
FC&C
FE&P
Aircraft FTA
System Integration Cross-check
System FHAs
FC&C
FE&P
Qualitative System FTAs
Automated Fault Tree Generation
System FTAs
Derived Safety Requirements
System FMEAs
Formal Model
Formal Model with Faults
Fault Injection
Automated Requirements Verification
Fault Model
Automated Fault Tolerance Verification

Fig. 2. Modified "V" Safety Assessment Process

As we can observe from Figure 2, the parts of the analysis that are primarily affected are at the bottom of the "V". The biggest difference is that the safety analysis activities at this level are now focused around a formal model of the system behavior, and that many of the artifacts of the safety analysis can be derived from this model. The idea is to try to pose the right verification questions to formal tools (such as model checkers and theorem provers) so that it is possible to derive the necessary safety analysis information. We then wish to turn the results of these analyses back into artifacts that can be easily understood and used by safety engineers.

3 Model-Based Safety Analysis Process

The primary step in a model-based safety analysis is creating a formal specification of the system model. The behavior of the system can be specified in formal specification languages supporting graphical and/or textual representation; e.g., synchronous (textual) languages like RSML^{-e} [10] and Lustre [6], and graphical tools like Simulink [5] and SCADE [9]. The logical and physical architecture of the system can be specified in an architecture description language.

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 or higher order predicate logics. One can also specify safety requirements as small behavioral models in some formal specification language.

To be able to apply formal verification tools to perform safety analysis, in addition to formalizing the system model, we also need to formalize the fault model. The fault model, in addition to common failure modes like *non-deterministic*, *inverted*, *stuck_at* etc, could encode information regarding fault propagation, simultaneous dependent faults and fault hierarchies, etc.

After specifying the fault model and composing it with the original system model, the safety analysis involves verifying whether the safety requirements hold in presence of the faults defined in the fault model. The safety engineer can perform exploratory analysis using formal verification tools, e.g., what is the largest n such that the particular safety requirement holds in face of n faults?. The notion could also be specialized to a specific combination of faults rather than random combinations. With adequate tool support, the formal verification results could be represented in the form of familiar safety artifacts like fault trees.

In the following sections, we illustrate some of our early results in applying the model based safety analysis process on a wheel brake system (WBS) example derived from the ARP safety analysis guidelines [1]. In section 4, we describe the informal requirements of the example. Next, in Sections 5 and 6, we describe how the system model without failures can be encoded in Simulink and how we can verify safety properties of interest on the model. In section 7, we describe a simple fault model for the WBS components and extend our system model to include component faults. Section 8 briefly describes the exploratory safety analysis performed on the extended model using the SCADE Design Verifier.

4 Wheel Brake System Example

We illustrate some of the basic activities involved in model based safety analysis with the help of an example of a Wheel Brake System (WBS), as described in ARP 4761 - Appendix L [1]. We chose this example primarily because the ARP 4761 document is used as the main reference for safety assessment by majority of the safety engineers in the avionics community.

This section consists of excerpts from the ARP 4761 document giving the informal requirements for WBS. The informal WBS diagram taken from the ARP 4761 document is shown in Figure 3. The WBS is installed on the two main landing gears. Braking on the main gear wheels is used to provide safe retardation of the aircraft during taxiing and landing phases, and in the event of a

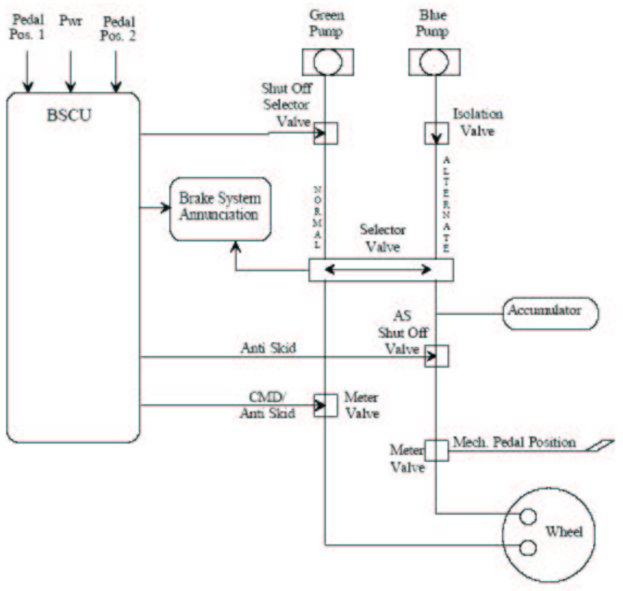

Fig. 3. Wheel Brake System as shown in ARP 47-61

rejected take-off. 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 autobreak function will control break pressure to provide a smooth and constant deceleration.

Based on the requirement that loss of all wheel braking is less probable than $5 \cdot 10^{-7}$ 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` pump and is used in the `NORMAL` braking mode. The `ALTERNATE` braking system is on standby and is selected automatically when the `NORMAL` system fails. The `ALTERNATE` system is supplied pressure by both the `BLUE` pump and an `ACCUMULATOR`, both of which can be used to drive the brake. The accumulator is the reserve pressure reservoir with built up pressure that can be reliably released if both of the two primary pumps (the Blue and Green pumps) fail. The accumulator drives the `ALTERNATE` system in the `EMERGENCY` braking mode.

Switch-over between the hydraulic pistons and the different pumps is automatic under various failure conditions, or can be manually selected. Reduction of `GREEN` pressure below a threshold value, either from loss of the `GREEN` pump itself or from its removal by the Break System Control Unit (BSCU) due to the presence of faults, causes an automatic selector to connect the `BLUE` supply to 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` system modes. 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 electronically provided 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.

5 Nominal Wheel Brake System in Simulink

The informal requirements of the WBS as specified in the ARP document were not found to be particularly rigorous. To implement a working model, we had to make several assumptions about the system that still need to be confirmed with the authors of ARP 4761. Figure 4 illustrates how we can model the WBS in Simulink. The model captures both digital and mechanical components of the system and reflects the informal structure of the system as given in the ARP document.

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) line. The system takes the following inputs from the environment - PedalPos1, PedalPos2, AutoBrake, DecRate, ACSpeed, Skid, and MechPedal. All of the above inputs, except MechPedal, 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).

Due to lack of space, we cannot describe the Simulink model in full detail[1]. To illustrate some aspects of fault modelling, we explain the implementation of the `MeterValve` component, which is used in three places in Figure 4: the `CMD/AS MeterValve` on the Normal hydraulic line and the `AS MeterValve` and `Manual MeterValve` on the Alternate hydraulic line. The meter valve implementation takes two inputs, the incoming pipe pressure and the valve position command, and generates an output pressure which depends on the valve position.

[1] We will publish the complete Simulink model on our web site: http://www.cs.umn.edu/crisys

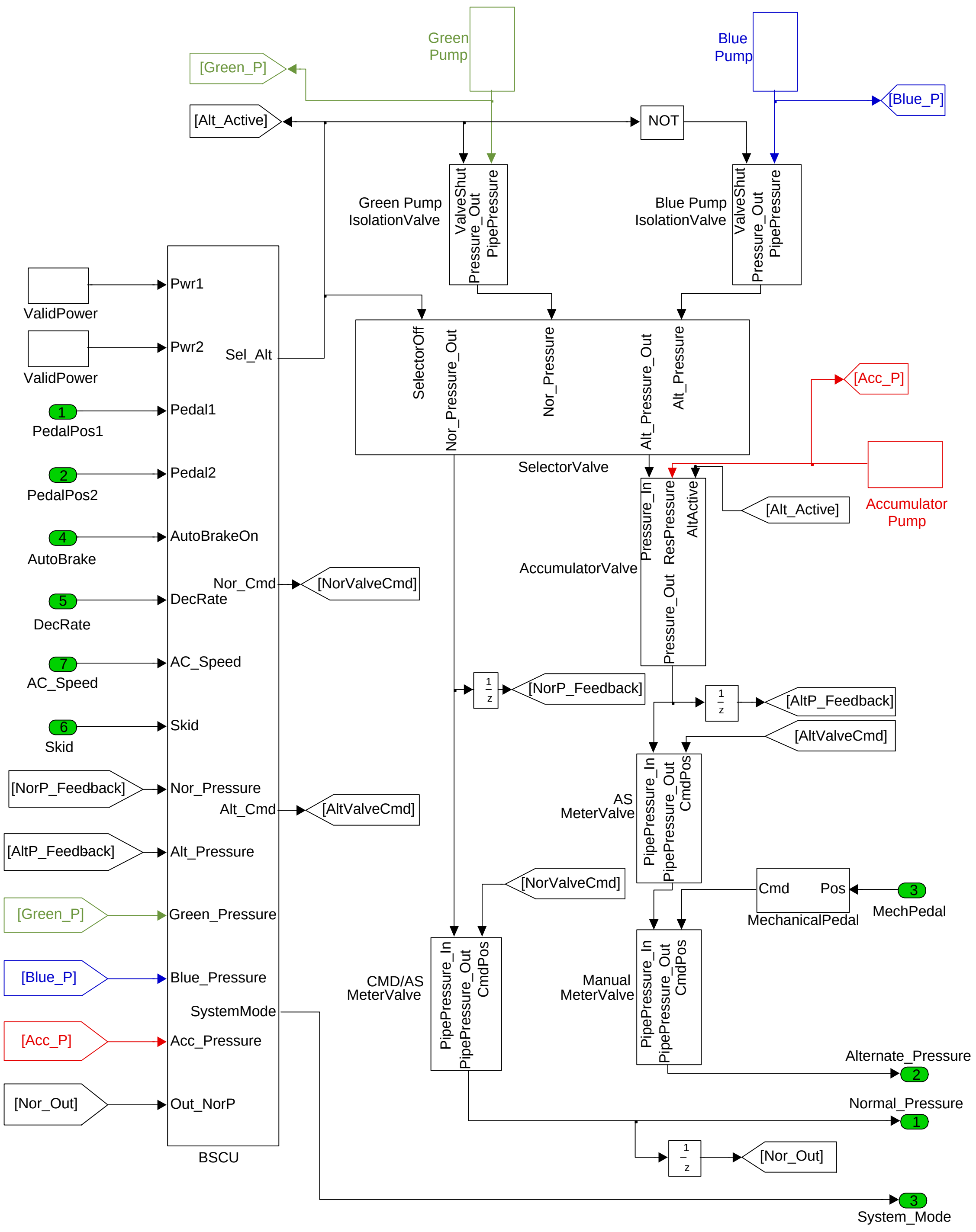

Fig. 4. Nominal Wheel Brake System in Simulink

6 System Verification

After creating the system model, we would like to verify that some basic safety properties hold on the *nominal system*, an idealized system containing no faults. As a first step, we need to formalize the derived safety requirements as safety properties. Simulink does not directly support any model-checking tools, so to perform this step, we import the Simulink model into SCADE, which contains the Design Verifier model checker. The properties can be formalized in Lustre, which is the underlying textual notation for SCADE.

Throughout this paper, we use an example safety requirement that is given in the ARP 4761 document,

> *Loss of all wheel braking (unannunciated or annunciated) during landing or RTO shall be less than* $5 \cdot 10^{-7}$ *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 and state the undesirable event we are trying to prevent as simply,

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

We consider that the hydraulic pressure at the output should be above some minimum constant threshold to have any effect on the braking. Recall from Section 4 that we have variables PedalPos1, PedalPos2, and MechPedal, that describe the electric and mechanical pedal positions, respectively. We can state our safety property as,

> *When all pedals are pressed, then either the normal pressure or the alternate pressure should be above the threshold.*

We first define two intermediate variables in Lustre to represent whether all of the pedals are being pressed (AllPed) and whether any pressure is being provided to the brakes (SomePressure).

```
AllPed       = (IS_PedalPressed(PedalPos1) and IS_PedalPressed(PedalPos2)
                and IS_PedalPressed(MechPedal));
SomePressure = (Normal_Pressure > threshold) or
               (Alternate_Pressure > threshold);
```

IS_PedalPressed is a predicate that returns true when pedal is pressed. AllPed and SomePressure are then used in the property SomePressure_Property as

```
SomePressure_Property = Implies(AllPed,SomePressure);
```

We used Design Verifier in an attempt to verify this property, which was initially found to be falsifiable. If the wheels do not have traction, the anti-skid functionality will be activated and the pressure at the wheels may indeed be lowered below the threshold to allow the wheels to regain traction. Since this is expected and acceptable behavior, we modify our safety property accordingly, by extending AllPed to AllPedNoSkid, where we require that the pedals are pressed and that we are not skidding.

```
AllPedNoSkid = (IS_PedalPressed(PedalPos1) and IS_PedalPressed(PedalPos2)
               and IS_PedalPressed(MechPedal) and not (Skid));
```

Now, the SomePressure property is verified by Design Verifier: if all pedals are pressed and we are not skidding then we will have some pressure at the brakes.

7 Extension with a Fault Model

In Section 5, we created a model describing the nominal behavior of the system. To perform the safety analysis on this model, we would like to extend it to describe possible fault behavior. This section illustrates specification of the fault model and extension of the nominal model with this fault behavior in Simulink.

Failure modes are introduced in the analysis to capture the various ways in which the components of the system can malfunction. We want to be able to model both persistent and intermittent failures and also multiple simultaneous failures. Traditionally, failure modes specify predefined ways in which components can fail, e.g., the output from a digital component might be stuck at a particular value, inverted, take on a nondeterministic value (unconstrained value), etc. In the WBS example, the mechanical failures considered include different variants of stuck valves corresponding to the different kinds of valves, power failure to the BSCU, and pump failures. We also consider one digital failure mode for the BSCU component, an inverted signal for the Boolean Sel_Alt (select alternate system) output.

Let us consider the notion of a valve stuck open or closed in more detail. The manifestation of this failure must consider the original input pressure to the component (in case the valve is stuck open) and override the normal output of the valve. We create a simple fault model in which a component can either be stuck open or closed in Figure 5. Binary_Stuck_at failure mode switches between the stuck value and the nominal value depending on the boolean Fail_Flag (fault trigger). The stuck value could be either Stuck_Val_1 (open) or Stuck_Val_0 (closed) depending on the boolean Stuck_Choice. Thus, we define two 'special' outputs for the failure mode depending on whether the component is stuck open or closed; if it is not stuck, we output the nominal value of the original component. We then extend the MeterValve component to MeterValve_Stuck using this failure mode (Figure 5). When Stuck_Choice is 1 the meter valve is stuck open and the input pressure is forwarded as is to the output, ignoring the valve position command. When Stuck_Choice is 0 the valve is stuck closed and the output pressure is set to 0.

To extend the original model, the nominal mechanical components from the original model (Figure 4) are replaced by the corresponding components extended with failure modes. To control the fault behavior of the extended model, a number of fault inputs need to be added to the system. For example, all the valve components, extended by the 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. After extension, the model looks fairly

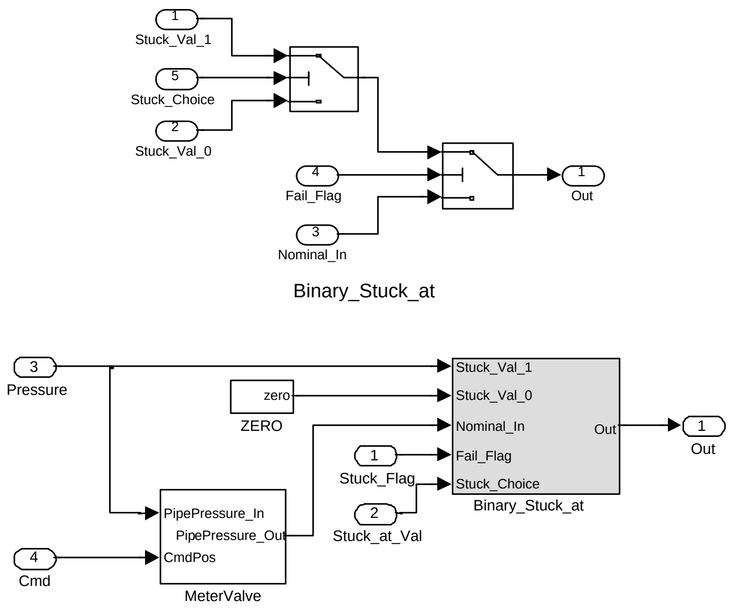

Fig. 5. Binary Stuck at failure mode and MeterValve fault extension

similar to Figure 4, but adds some complexity and clutter due to the number of additional inputs necessary to describe the possible faults.

8 Exploratory Safety Analysis

After extending the model with the faults, we would like to check the fault tolerance of our system, i.e., we want to check that the system is tolerant to a certain maximum number of faults. More specifically, we would like to investigate two types of faults using this approach—*transient single step faults* and *faults lasting over an arbitrary number of steps*, which can simulate permanent faults. For this example we again formalize our safety properties in Lustre and use the SCADE Design Verifier for verification. To make it easier to specify properties, we extend our model to compute the total number of fault inputs that are true in the current step (this number given by NumFails).

First, let us verify if our safety requirement holds in the presence of one fault.

> *If there is one fault and all pedals are pressed in absence of skidding, then either the normal pressure or the alternate pressure should be above the threshold.*

We can formalize this in Lustre as,

```
Prop_Orig = fby(Implies(((NumFails = 1) and AllPedNoSkid),
                         SomePressure), 1, true) ;
```

Lustre expressions always look at the current and past instants. Implies encodes implication and pre operator examines values of variables from previous steps. The *fby* operator looks at the n-th previous value of an expression (in this case, the *Implies* expression). The second argument (1) of the fby expression is the value for n. The third argument (true) describes the value of the fby operator in the initial state.

When attempting to verify `Prop_Orig` using Design Verifier, it returns a counterexample. We realize that, due to some latency, the system cannot respond to most faults in the same step in which they occur. Unfortunately, even after extending the number of steps to respond, if our only constraint is on the number of faults, the model checker finds a counterexample. It gives a scenario in which the fault *migrates*: the system toggles between faults on the Normal line and the Alternate line and can never recover. Since this situation is highly unlikely, we rule it out. We do so by stating that any transient fault will be followed by a few steps in which no other transient fault occurs. In other words,

> *If there is one single step fault and in the next step all pedals are pressed in absence of skidding, then in the next step either the normal pressure or the alternate pressure should be above the threshold.*

The encoding in Lustre is as follows:

```
Antecedent = pre(NumFails = 1) and AllPedNoSkid and (NumFails = 0);
Consequent = SomePressure ;
Prop_SingleStepSingleFail = fby(Implies(Antecedent, Consequent),2,true) ;
```

However, the Design Verifier still returns with a counterexample. We observe that there is an additional step delay for the system to detect failures located on the `NORMAL` system and switch to the `ALTERNATE` system. We deem this delay acceptable and modify our property again. After allowing for an additional delay in the property, the Deign Verifier verifies it. Thus, we can formally verify that our system can recover from one transient fault in at most three steps.

Now, we want to investigate how our system responds to *persistent* faults. To describe this fault scenario, we define a boolean variable, `Changed`, which takes on the value true when one of more of the fault trigger inputs change their values. Using this variable, we can describe persistent faults in which *the same* fault occurs for an arbitrary number of steps. The following property is the same as the earlier transient property, except that now we have `not(Changed)` instead of `(NumFails = 0)` to encode that the same fault persists in the following two steps.

```
Antecedent = pre(pre(NumFails = 1)) and pre(AllPedNoSkid and not(Changed))
             and AllPedNoSkid and not(Changed) ;
Consequent = pre(SomePressure) or SomePressure ;
Prop_MultiStepSingleFail = fby(Implies(Antecedent,Consequent),3,true) ;
```

Design Verifier again finds a counterexample, and from this, we observe that there is an additional delay required for the system to respond to some persistent faults, in the situation when the system is switching back to the `NORMAL` hydraulic system from the `ALTERNATE` system. In this instance, it takes an additional step to check if a persistent fault on the `NORMAL` line is still present. To handle this case, we add one additional step for the system to stabilize. Design verifier no verifies that the system will behave as expected. Thus, we verify that the system can recover from single transient or persistent faults within an acceptable time frame.

However, we can easily observe that the system is not tolerant to two (or more) simultaneous continuous failures. Design Verifier immediately comes back with a counter-example where two meter valves fail along both the normal and the alternate hydraulic lines. Note that, the safety engineer can explore different combinations of faults that the system can tolerate. There will not be even a glitch in the output pressure if all the components on the Alternate line fail when no component along the Normal line fails.

9 Related Work

Most of the work in automating safety analysis has been in automatically generating fault trees. FSAP/NuSMV-SA [4] is a tool, developed as part of the ESACS project [3], for automating the generation of fault trees. The ESACS methodology supports integrated design and safety analysis of systems. The FSAP tool requires the system model to be specified in NuSMV and has support for failure mode definition and model extension through automatic failure injection. FSAP uses the NuSMV model checker to generate a fault tree given a top level event in temporal logic. 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; 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. A fault tree generated by a traditional manual analysis is usually more intuitive to read as the analyst creates the fault tree to correspond to the structure of the system. Also, we observed that there isn't a lot flexibility in defining the fault model - no good way of specifying fault propagation, simultaneous/dependent faults, and persistent/intermittent faults. Also, FSAP cannot describe even moderately complex faults, such as stuck_at, as it can only affect the output of a component.

HiP-HOPS (Hierarchically Performed Hazard Origin and Propagation Studies) [8] [7] is a method for safety analysis that enables integrated assessment of a complex system from the functional level through to the low level of component failure modes. The failure behavior of components in the model is analyzed using a modification of classical FMEA called Interface Focused-FMEA (IF-FMEA). One of the strong points of this approach is that the fault tree synthesis algorithm neatly captures the hierarchical structure of the system in the fault tree.

The Altarica language was designed to formally specify the behavior of systems when faults occur [2]. An Altarica model can be assessed by means of complementary tools such as fault tree generator and model-checker. In terms of fault modeling, there seems to be no good support for simultaneous and dependent failures. Altarica does not differentiate between transient and permanent faults.

10 Summary and Conclusion

We describe Model-Based Safety Analysis, an approach for automating portions of the safety analysis process using executable formal models of the system. This approach is based on existing commercial tools and techniques that are increasingly used for systems and software engineering for safety-critical systems. We have modelled the Wheel Brake System example from ARP 4761 - Appendix L [2]. We illustrated how this system can be modelled and investigated for safety and fault tolerance. We believe that the model-based safety analysis approach has several benefits to offer to a next-generation safety analysis process. For instance,

- 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 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.

Although we have received positive feedback from our industry partners, there are several research challenges that must be addressed before the full benefits of model-based safety analysis can be fully realized. First, there are questions as to which languages and tools are most suitable and how much modeling detail is necessary to perform useful analysis. Second, we observed that directly composing the fault model with the system model clutters the 'nominal' model with failure information, which obscures the nominal system functionality. This complexity may make model evolution difficult, error prone, and costly. In our opinion, the system model and the fault model should be defined separately and some automatic composition mechanism should be created allowing the system model and fault model to be easily merged for analysis. Third, although we were able to successfully analyze a realistic example, there are serious questions about the scalability of the analysis tools.

Acknowledgements

We would like to thank Dr. Steven Miller and Dr. Michael Whalen of Rockwell Collins Inc. for their valuable insights and feedback in course of our collaboration with them on this project.

References

1. SAE ARP4761. *Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*. SAE International, December 1996.
2. Pierre Bieber, Charles Castel, and Christel Seguin. Combination of fault tree analysis and model checking for safety assessment of complex system. In *In Proceedings of the 4th European Dependable Computing Conference on Dependable Computing*, pages 19 – 31. Springer-Verlag, 2002.
3. M. Bozzano, A. Villafiorita, O. kerlund, P. Bieber, C. Bougnol, E. Bde, M. Bretschneider, A. Cavallo, C. Castel, M. Cifaldi, A. Cimatti, A. Griffault, C. Kehren, B. Lawrence, A. Ldtke, S. Metge, C. Papadopoulos, R. Passarello, T. Peikenkamp, P. Persson, C. Seguin, L. Trotta, L. Valacca, and G. Zacco. Esacs: an integrated methodology for design and safety analysis of complex systems. In *In Proceedings of ESREL 2003*, pages 237–245. Balkema Publishers, June 15-18 2003.
4. Marco Bozzano and Adolfo Villafiorita. Improving system reliability via model checking: the fsap / nusmv-sa safety analysis platform. In *In Proceedings of SAFECOMP 2003*, pages 49–62, Edinburgh, 2003. Springer.
5. James Dabney and Thomas Harmon. *Mastering Simulink*. Prentice Hall, Upper Saddle River, NJ, 2004.
6. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.
7. Yiannis Papadopoulos and Matthias Maruhn. Model-based synthesis of fault trees from matlab-simulink models. In *The International Conference on Dependable Systems and Networks (DSN'01)*, July 01 - 04 2001.
8. Yiannis Papadopoulos and John A. McDermid. Hierarchically performed hazard origin and propagation studies. In *In Proceedings of the 18th International Conference, SAFECOMP'99*, volume LNCS 1698. Springer-Verlag, 1999.
9. Esterel Technologies. Scade suite product description. http://www.esterel-technologies.com/v2/scadeSuiteForSafetyCriticalSoftwareDevelopment/index.html.
10. Michael W. Whalen. A formal semantics for RSML^{-e}. Master's thesis, University of Minnesota, May 2000.