Presented at the 24th Digital Avionics Systems Conference, Washington, D.C., October, 2005. Awarded best paper of the Open Systems Architecture track.

A PROPOSAL FOR MODEL-BASED SAFETY ANALYSIS

Anjali Joshi, Steven P. Miller+, Michael Whalen+, Mats P.E. Heimdahl**

**Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455*

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

Abstract[1]

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 finding undocumented details of the system behavior and embedding this information in the safety artifacts such as the fault trees.

In this paper we propose an approach, *Model-Based Safety Analysis,* in which the system and safety engineers use the same system models created during 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.

Introduction

Safety engineers traditionally perform analysis, such as fault tree analysis [10], based on informal design models and various other documents such as requirements documents. Unfortunately, these analyses are highly subjective and dependent on the skill of the practitioner. 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.

We hypothesize that most of the review effort is focused on uncovering and resolving misunderstandings and missing information in the system design or the informal fault model. By redirecting some of this manual analysis and review effort to build formal models of the system and its fault model and performing analysis based on these models, an approach we call *model-based safety analysis*, we believe we can both reduce the effort involved (by introducing automated tools) and increase the quality of the safety analysis.

In the remainder of this paper we describe our model-based safety analysis approach. We begin by briefly summarizing the traditional safety analysis process currently practiced in the commercial avionics industry. We then discuss 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 the potential changes that might be required to accommodate the model-based safety approach in the traditional context. We then illustrate this approach with the help of the wheel brake system example described in ARP 4761 [1]. We conclude with a brief discussion of the different organizational and research challenges in making this approach practical.

Traditional 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

[1] This work was supported in part by the NASA Langley Research Center under contract NCC-01001 of the Aviation Safety and Security Program.

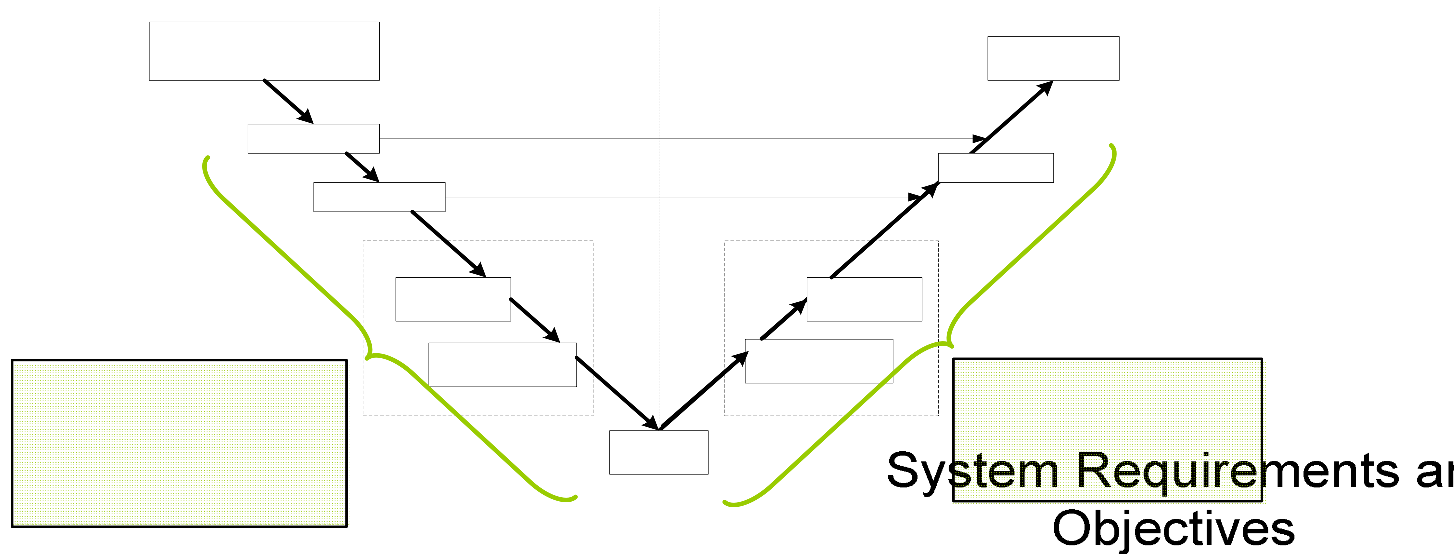

Figure 1: "V" Process for Traditional Safety Assessment

The safety assessment process is an integral part of the development 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 the 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; design changes necessitate changes to the derived system requirements (and also to the fault trees) and vice-versa.

Once the design and implementation are completed, the System Safety Assessment (SSA) process verifies whether the safety requirements are met in the implemented design. If not already known, Failure Modes and Effects Analysis (FMEA) may be performed to compute the actual failure probabilities on the items (components). 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.

Model-Based Safety Analysis

In the safety-critical systems domain there is an increasing trend towards *model-based development* of the 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 [14] or SCADE [8]. In *model-based safety analysis,* we propose to extend the existing model-based development activities to incorporate safety analysis. In this section, we first briefly discuss model-based development and then illustrate our model-based safety analysis approach. We also discuss how we can modify the traditional safety assessment process to accommodate model-based safety analysis.

Model-Based Development

In model-based development, the development effort is centered around a formal specification (model) of the digital control system. This model can then be subjected to various types of analysis, for example, completeness and consistency analysis, model checking (details on page 5), and theorem proving [12]. Ideally, one would then *automatically and correctly generate* the implementation from this specification.

Presented at the 24th Digital Avionics Systems Conference, Washington, D.C., October, 2005. Awarded best paper of the Open Systems Architecture track.

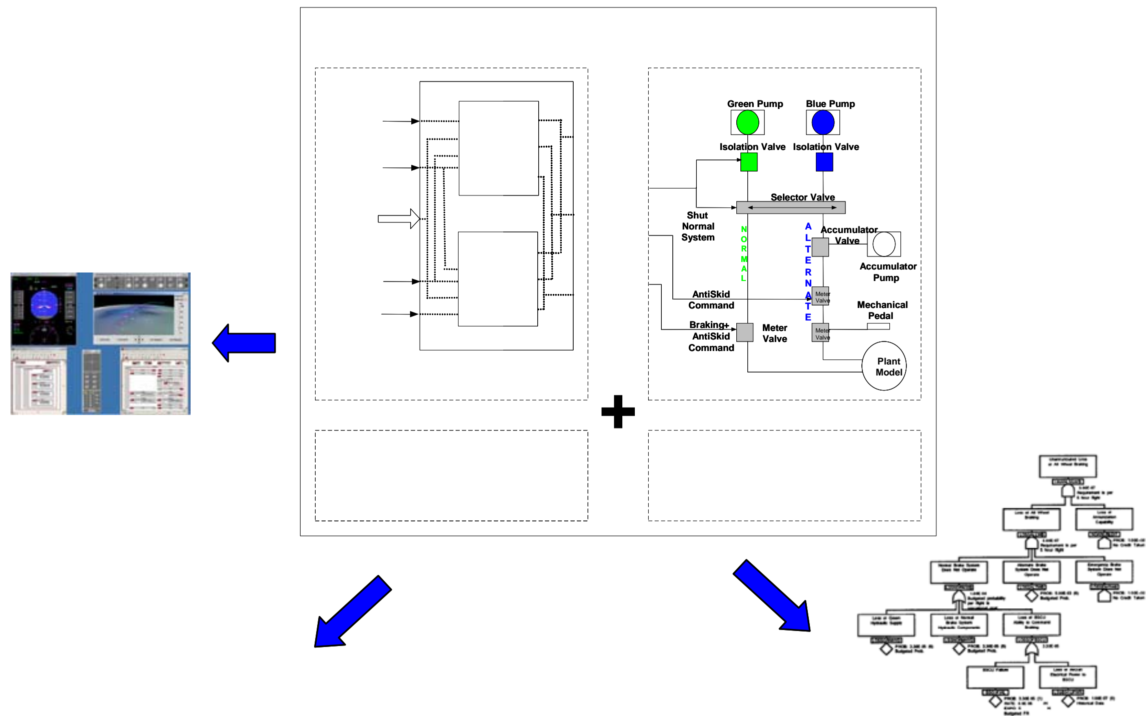

Figure 2 : Automated Model-based Safety Analysis

There are currently several commercial and research tools that attempt to provide these capabilities. Examples of commercial tools include Esterel and SCADE from Esterel Technologies [8], Statemate from i-Logix [15], and SpecTRM from Safeware Engineering [17].

Model-Based Safety Analysis

To perform model-based safety analysis, we propose to leverage the framework of mature development and analysis tools that are successfully used for model-based development.

Model-based development focuses primarily on formally modeling only the software components of the system. To perform system-level safety analysis, we also have to consider the mechanical components involved. Fortunately, similar tools and techniques can be used to model the physical components of interest. By combining the 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 Safety Analysis System Model. This model can be used to describe the behavior of the system in the presence of one or more faults.

The safety analysis system model can be used for a variety of simulations and analyses (Figure 2). First, the models allow 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 (shown in Figure 3) is modified such that the safety analysis activities are centered around formal system and fault models. These models are used both for systems analysis and design and safety analysis, and are the central artifact of the systems development process.

Presented at the 24th Digital Avionics Systems Conference, Washington, D.C., October, 2005. Awarded best paper of the Open Systems Architecture track.

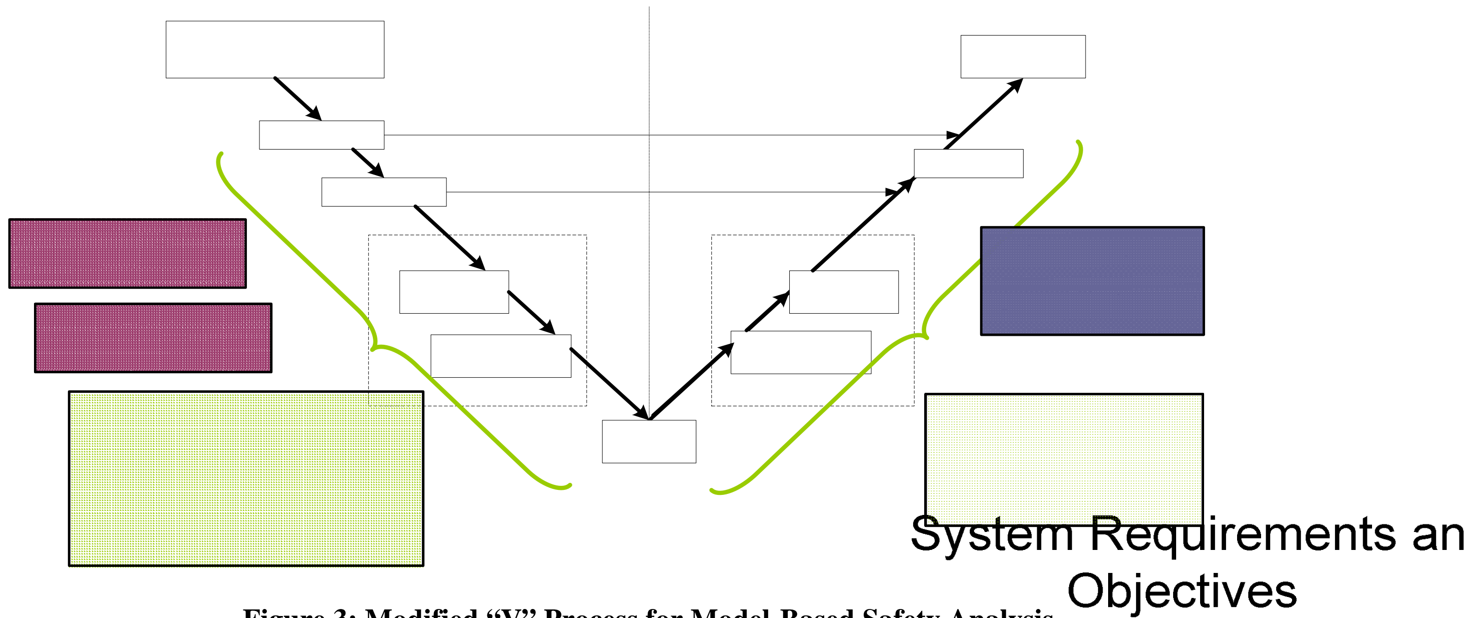

Figure 3: Modified "V" Process for Model-Based Safety Analysis

Given these models, the safety analysis process consists of defining a set of formal properties to describe 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 model of the system leading to a tighter integration between the two. The safety engineers will be automatically notified of changes in the system model when their analyses fail. System engineers can run the safety analyses to determine the effect of design changes. Formally capturing the fault models reduces ambiguity. Ideally, the use of computational tools such as model checkers will help 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 the manual effort required.

We now describe the various model-based safety analysis activities in detail.

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 [5], and graphical tools like Simulink [14] and SCADE [8]. The logical and physical architecture of the system can also be specified in these notations or with an architecture description language such as AADL [11].

Formalizing Derived Safety Requirements

To support automated analysis, the safety properties must be expressed in some formal notation. The derived safety requirements are determined in the same way as in the traditional "V" process. There are several candidate notations, including temporal logics like CTL/LTL [16] or higher order predicate logics. One can also specify safety requirements as small behavioral models in some formal specification language.

Fault Modeling

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 captures the various ways in which the components of the system (both the digital controller and the mechanical system) can malfunction. It could consist of common failure modes like *non-deterministic*, *inverted*, *stuck_at*, etc. The fault model could potentially be much more complex, encoding fault propagations, dependent faults, etc. In this case, the fault model is also dependent on the architecture of the system. We would also like to be able to specify both *persistent* and *intermittent* failures.

Composing System and Fault Models

To enable model-based safety analysis, the fault model is merged with the nominal system model to describe the behavior of the system in the presence of faults.

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. Unfortunately, this embedding clutters the system model with failure information which may not be of interest to systems engineers. Also, it becomes difficult to evolve the system and fault models separately. This manual composition will also become error prone as the complexity of the fault model increases.

A better option is to develop the fault model as a separate entity from the system model and automatically merge these two models for analysis. By keeping the system model and the fault model separate, they each can be refined somewhat independently. We hypothesize that keeping the two models separate will ease evolution, reduce errors and unnecessary clutter.

Formal Safety Analysis

Once we have the composed model, the safety analysis involves verifying whether the 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 like model checkers to verify safety properties of interest.

Model Checkers

Model checking is often called "push-button" formal methods: given a formal model specified in the notation of the model checker and a property of interest, a model checker will automatically determine whether a property holds of a given model. It performs this task by exploring the full state space of the system model to check whether the given system properties are satisfied by the model. If the checker determines that the property holds, it guarantees that the system satisfies the property under *all possible* executions of the system. Alternately, if the model checker is able to disprove the property, it generates a *counterexample,* which describes a sequence of execution steps (similar to a test case) in which the property does not hold.

There are two main advantages of using model checking compared to other formal verification methods. First it is fully automatic, and second it provides a counter example whenever the system fails to satisfy a given property. The primary limitation of model-checking is the size of the reachable state space, though recent breakthroughs allow very large (> 10^{100} reachable states) state spaces to be explored in reasonable time.

Proofs of Safety Properties

The safety engineer may want to explore the fault tolerance for the system; 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. The safety engineer may also want to investigate how the system behaves in presence of different types of faults, e.g. permanent and transient faults. We can use computational tools such as model checkers for verifying these kinds of safety properties.

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 the discussion of related work on page 11).

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) 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 safety engineers in the avionics community.

This section consists of excerpts from the ARP 4761 document giving the informal requirements for the WBS. The WBS diagram taken from the ARP 4761 document is shown in Figure 4. 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 rejected takeoff. 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 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 Brake 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, improve system fault tolerance, and generate warnings, indications and maintenance information to other systems.

Nominal System Modeling

As with most informal specifications, the requirements of the WBS as specified in the ARP document left many questions unanswered. 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 5 illustrates how the WBS can be modeled in Simulink. The model captures both the digital and the mechanical components of the system and reflects the structure of the system as given in ARP 4761.

The WBS (the highest level component/system) consists of a digital control unit, the BSCU, and two hydraulic pressure lines, the Normal (pressured by the Green Pump) line and the Alternate (pressured by the Blue Pump and the Accumulator) 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 to compute the brake commands. The BSCU is in turn composed of two redundant Command (computes the brake and anti-skid commands) and Monitor (monitors whether the command outputs are valid) units.

There are also a number of mechanical components along the two hydraulic lines such as the different types of valves. We have defined a library of the common components such as the MeterValve, the IsolationValve, the Pump, etc., which are instantiated at various locations in the WBS. The

outputs of the WBS are Normal_Pressure (hydraulic pressure at the end of the Normal line),

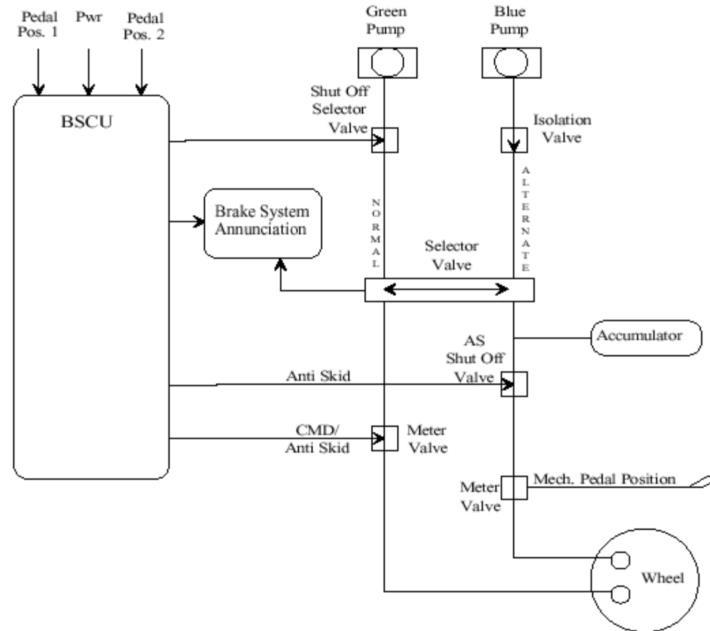

Figure 4 : Wheel Brake System Diagram (SAE ARP 4761)

Alternate_Pressure (hydraulic pressure at the end of the Alternate line) and System_Mode (computed by the BSCU).

Space does not allow us to describe the Simulink model in full detail. To illustrate some aspects of the fault modeling presented later, we explain the implementation of the MeterValve component, which is used in three places in Figure 5: 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 that determined by the valve position.

Formalizing Derived System Requirements

After creating the system model, we would like to verify that some basic safety properties hold on the *nominal system*, an idealized model of the digital controller and the mechanical 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, whose underlying textual notation is Lustre. As part of this project, we have developed a translator framework with which we can translate Lustre specification into the input languages of different analysis tools, for example the NuSMV model checker [9] and PVS theorem prover [12]. We can also perform such analyses directly in SCADE, which contains the Design Verifier model checker. In this paper, we will describe the analysis performed using the NuSMV model checker. In [13], a more detailed analysis of a slightly different version of the wheel brake system is described using Design Verifier.

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

> *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 including any quantitative analysis at this stage, we can simplify this safety requirement and state it simply as,

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

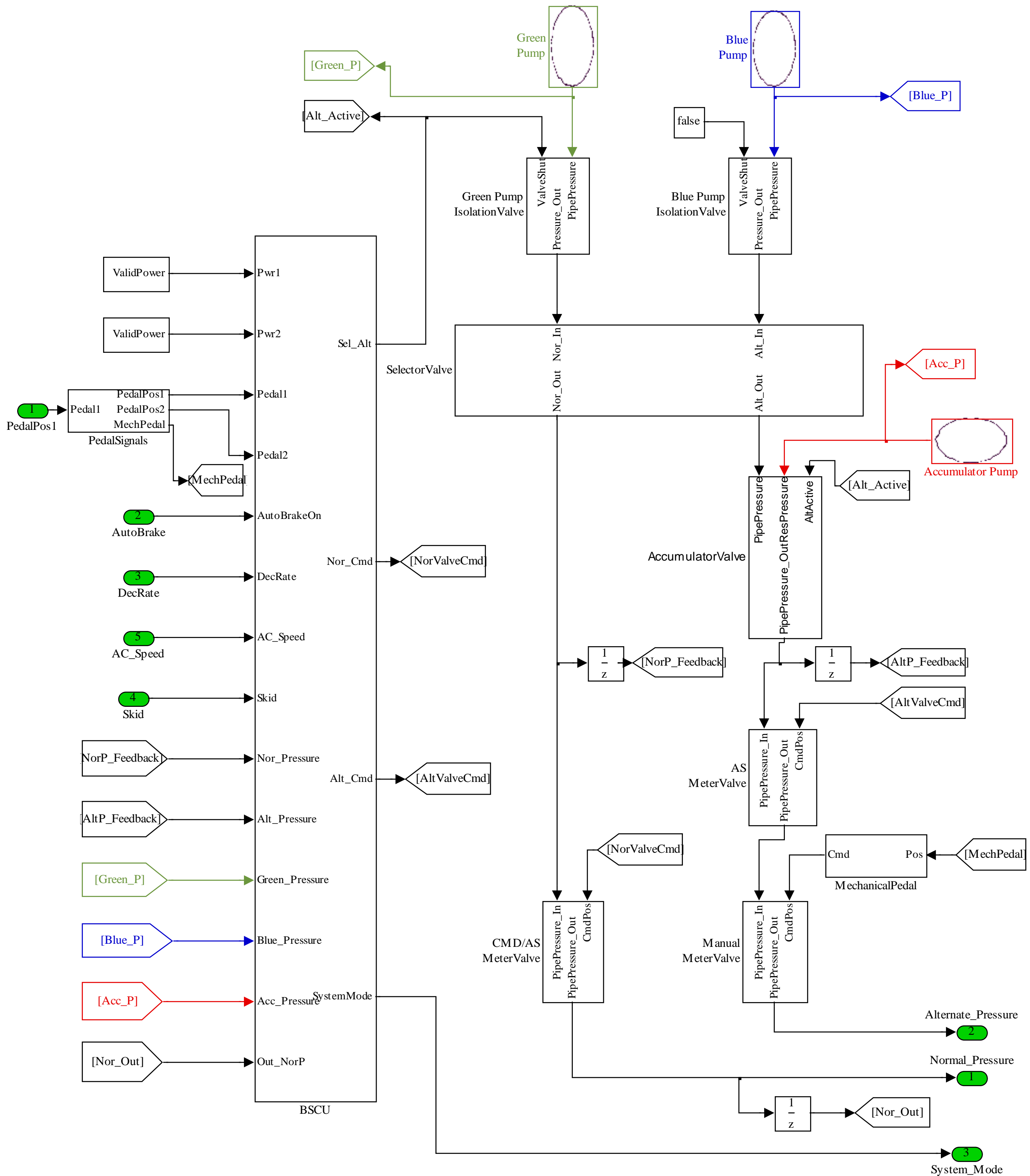

Figure 5 : Simulink model of the Wheel Brake System (WBS)

To achieve effective breaking, the hydraulic pressure at the break 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 break 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 (PedP_NoSkid) and whether any pressure is being provided to the brakes (SomeP).

```
PedP_NoSkid := ((IsPressed(PedalPos1) |
AutoBrake)  & !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 break pressure. This property can be proven to hold in our nominal system (where no failures occur) in seconds using NuSMV.

Fault Modeling

In the WBS example, we consider a simple fault model containing both mechanical failure modes and digital failure modes. We can implement failure modes in Simulink as subsystems (or components) with additional *failure flags* that can be used to control whether or not the failure has occurred.

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

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

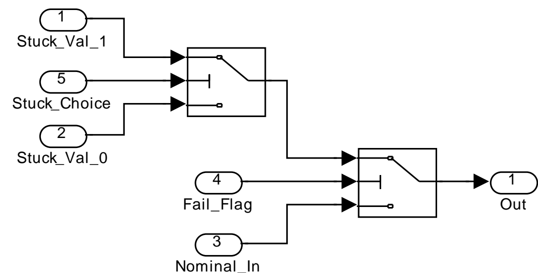

Figure 6 : Binary_Stuck_at

We create a simple fault model in which a component can either be stuck open or closed as shown in Figure 6. The Binary_Stuck_at failure mode switches between the stuck value and the nominal value depending on the Boolean Fail_Flag input (the fault trigger). The stuck value could be either Stuck_Val_1 (open) or Stuck_Val_0 (closed) depending on the Boolean Stuck_Choice. We extend the MeterValve component to MeterValve_Stuck using this failure mode. 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.

Composing System and Fault Models

To extend the original model, the nominal mechanical components from the original model

(Figure 5) 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 similar to Figure 5, but with some additional clutter due to the additional inputs needed to describe the possible faults.

Model-based System Safety Analysis

After extending the model with the faults, we would like to check that the system is tolerant to a certain maximum number of faults. In particular, we would like to investigate two types of faults using this approach: *transient faults* and *permanent faults*. 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 total number of fault inputs that are true in the *current step* (this number given by NumFails).

First, let us attempt to verify that our safety requirement holds in the presence of one (transient) fault.

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

```
SPEC AG (NumFails = 1 & PedP_NoSkid ->
        SomeP);
```

This property does not hold and NuSMV comes back with a counterexample indicating that as soon as a critical component fails (e.g., the green pump) we will loose pressure at the brake calipers. 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—a process that requires two time steps. We deem this delay acceptable and refine our property to reflect this delay.

```
SPEC AG((NumFails = 1 & PedP_NoSkid) ->
   AX ((NumFails = 1 & PedP_NoSkid) ->
    AX ((NumFails = 1 & PedP_NoSkid) ->
        SomeP)));
```

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. However, verification of this relaxed safety property is still not possible as illustrated by the following scenario: If there is a single transient failure (e.g., the Green pump fails) then the BSCU will detect this failure (in a couple of steps) and switch over to the Alternate system powered by the Blue pump. In this version of the WBS, the switch over to the Alternate system is not reversible. Even if the fault that caused the switch over is transient and is repaired, the system will not switch back to the Normal hydraulic system. In our counterexample, even if the Green pump recovers, the active hydraulic system will still be the Alternate system. Now, if some meter valve along the Alternate system fails closed (stuck at closed), then the system cannot recover from this failure and we will not get any braking pressure. So even though it took two failures (one a transient failure) to cause the loss of braking pressure, there was never more than one failure at any particular instant and our property does not hold. One of the great advantages of automated tools is that they force the consideration of such scenarios. Of course, we can easily change the system model so that the switch-over to the Alternate system is not permanent and the system can recover from a transient fault in a certain number of steps. For a discussion of the analysis of such a version of the system, refer to [13].

We would also like to verify that the system will recover from a single permanent failure. In our extended model, the fault triggers are modeled as inputs as we want the engineer to be able to control the triggering of faults during simulation. During analysis, the model checker will consider all combinations of inputs, including the cases where faults are transient and cases where faults are permanent. If we truly want to consider only permanent faults during analysis, we must latch the corresponding fault triggers when they become true.

Now, we try to prove the same property noted above (with delays) in the presence of permanent faults. NuSMV verifies that this property holds. Thus, the WBS model can recover from one permanent failure in three steps. However, we can easily observe that the system is not tolerant to two (or

more) simultaneous continuous failures. NuSMV immediately comes back with a counterexample where two meter valves along both the Normal and the Alternate hydraulic lines fail. Note that, the safety engineer can explore different combinations of faults that the system can tolerate.

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. First, 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. On a large system, this can yield a fault tree with a root node that has a single or-gate with hundreds or thousands of branches. Also, we observed that there isn't a lot of 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) [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 [6].

Conclusions and Future Directions

In this paper, we proposed *Model-Based Safety Analysis*, an approach for automating portions of the safety analysis process using executable formal models of the system. We also illustrated this approach in a step by step process using a wheel brake system model as described in the ARP 4761.

Model-based safety analysis represents a large and significant change from current practices. There are several research challenges that need to be addressed if it is to be smoothly integrated into current safety engineering practices.

Nominal and Fault Modeling

The current process of adding faults into the system model is cumbersome and significantly clutters the nominal system model with fault trigger inputs. Such models can be difficult to create, difficult to read, and difficult to update as the system evolves. To make this approach practical, we need to address several important questions. For example, which languages and tools are the most applicable? What information would a fault model encode? Can we automate the process of merging the fault model and the nominal model?

Flexibility and Scalability of Analysis

Due to the size and complexity of system models, the analysis we have described will probably exceed the limitations of automated tools such as model checkers. Scaling up to significantly larger systems will require additional research into techniques for model abstraction and partitioning or the use of manually guided tools such as theorem provers, e.g., PVS. Ideally, we would like to allow the systems and safety engineers to easily check the effects of different combinations of fault models and system designs, both through simulation and automated analysis. This is not cleanly supported in any of the currently available tools.

User-interface and presentation issues

Currently, the results generated by model checkers and theorem provers do not look like the artifacts expected by safety analysts. There is research that has begun to address turning counterexamples into fault trees, but the current results are unacceptable for real safety analyses (see Related Work section). To better fit with existing safety analysis guidelines,

we need to be able to present analysis results in familiar forms, such as fault trees, that better map to current safety analysis practice.

Process Issues

Finally, there are numerous process issues to be addressed. For example, will the system engineers run the automated safety analysis tools as they build their system models, or will that be done by the safety engineers. Will the fault models be created by the system engineers and reviewed by the safety engineers, or vice-versa? As the system models are updated, who will be responsible for redoing the safety analysis? Many of these issues will depend critically on how much of the safety analysis can be automated and how much will remain a manual process.

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.

References

[1] SAE ARP4761, December 1996, Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment.

[2] Pierre Bieber, Charles Castel, Christel Seguin. 2002, Combination of fault tree analysis and model checking for safety assessment of complex system, In Proceedings of the 4th European Dependable Computing Conference on Dependable Computing, pages 19 - 31.

[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, 2003, ESACS: An integrated methodology for design and safety analysis of complex systems. In Proceedings of ESREL, pages 237 - 245.

[4] Marco Bozzano and Adolfo Villafiorita, 2003, Improving system reliability via model checking: the FSAP / NuSMV-SA safety analysis platform, In Proceedings of SAFECOMP, pages 49 - 62.

[5] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, 1991, The synchronous dataflow programming language Lustre, In Proceedings of the IEEE, 79(9):1305- 1320.

[6] Yiannis Papadopoulos and Matthias Maruhn, 2001, Model-based synthesis of fault trees from Matlab-Simulink models, In The International Conference on Dependable Systems and Networks.

[7] Yiannis Papadopoulos and John A. McDermid, 1999, Hierarchically performed hazard origin and propagation studies, In Proceedings of the 18th International Conference, SAFECOMP'99.

[8] Esterel Technologies. SCADE suite product description, http://www.estereltechnolgies.com.

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

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

[11] 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

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

[13] Anjali Joshi and Mats P.E. Heimdahl, 2005, Model-Based Safety Analysis of Simulink Models using SCADE Design Verifier, To appear in Proceedings of SAFECOMP'05.

[14] The Mathworks, Simulink Product Description, http://www.mathworks.com

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

[16] Clarke, Edmund M., Orna Grumberg, and Doron A. Peled, 2001, Model Checking, The MIT Press, Cambridge, Massachusetts.

[17] Nancy G. Leveson, Mats P.E. Heimdahl, Jon Reese, 1999, Designing Specification Languages for Process Control Systems: Lessons Learned and Steps to the Future, in Proceedings of Foundations on Software Engineering.

Presented at the 24th Digital Avionics Systems Conference, Washington, D.C., October, 2005. Awarded best paper of the Open Systems Architecture track.

October 30, 2005