Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 05-039

FGS Partitioning Final Report

Steven P. Miller, Michael W. Whalen, Dan O'brien, Mats P. Heimdahl, and Anjali Joshi

December 14, 2005

FGS Partitioning Final Report

Steven P. Miller
Mike W.Whalen

Dan O'Brien
Mats P.E. Heimdahl
Anjali Joshi

{spmiller,mwwhalen}@rockwellcollins.com
{dobrien,heimdahl,ajoshi}@cs.umn.edu

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

Partitioning a system consists of dividing it into components that can be physically isolated from each other while preserving the essential behavior of the system. In this report, we describe a methodology for developing and reasoning about such systems. This approach allows a developer to start from an ideal system specification and refine it along two axes. Along one axis, the system can be refined one component at a time toward an implementation. Along the other axis, the behavior of the system can be relaxed to produce a more cost effective but still acceptable solution. We illustrate this process by applying it to the synchronization logic of a Dual Fight Guidance System, evolving the system from an ideal case in which the components do not fail and communicate synchronously to one in which the components can fail and communicate asynchronously. For each step, we show how the system requirements have to change if the system is be implemented and prove that each implementation meets the revised system requirements through model-checking.

Table of Contents

1 Introduction

Recent advances in modeling languages have made it feasible to formally specify and analyze the behavior of relatively large system components. Synchronous data flow languages, such as Lustre [2], SCR [14], and RSML^{-e} [24] seem to be particularly well suited to this task, and commercial versions of these tools such as SCADE [11] and Simulink [9] are growing in popularity among designers of safety critical systems. At the same time, advances in formal analysis tools have made it practical to formally verify important properties of these models [4], [5], [6], [12], [16], [19], [25], [26].

However, when several component models are assembled into a single system, these verification techniques typically fail when applied to the overall system. Often, the growth in the state space overwhelms analysis tools such as model checkers that work by state space exploration. System implementers often need to distribute a system over several computing platforms that are allowed to execute asynchronously. Techniques for the analysis of synchronous models do not scale well to such Globally Asynchronous/Locally Synchronous (GALS) architectures. Sometimes, problems occur simply because the system was built from the bottom up, with most of the attention focused on the behavior of the individual components and insufficient attention given to showing how these components work together to meet the overall system requirements.

A better approach is to invert this process, i.e., to start with the specification of the overall system architecture and minimal specifications of the behavior of the components so that we can formally verify the key system properties first. The system components can then be developed to meet their minimal specifications imposed by the overall system. For example, the overall system requirement for a Flight Guidance System might be to compute the correct pitch and roll commands even if some components or sensors fail. By first specifying and reasoning about the system architecture, we can prove that the system will always select valid data sources, will always select a non-failed computational component, and will always deliver its outputs to the appropriate servos. By combining this analysis with a proof that a non-failed computational component will compute the proper pitch and roll commands if it is provided correct inputs, we can prove the overall requirement of generating the correct pitch and roll even if components fail.

This report describes a methodology for developing and verifying a system architecture in steps. The developer starts by specifying the ideal system and verifying the behavior of its implementation. Often in the course of this process, we realize that the ideal system is over-specified and that trying to meet these requirements is impossible or too costly. In such cases, we have no choice but to go back and relax the system requirements to something more realistic. For this reason, the design process does not proceed in a straight line from specification to implementation, but iterates between refining the design and relaxing the system requirements.

We illustrate this approach by applying it to the synchronization logic of a Flight Guidance System. Although this is a relatively simple example, it covers many of the interesting features involved in reasoning about systems architectures, such as asynchrony and component failures.

While this example is primarily concerned with issues of system safety, it is likely that this same approach could also be applied to issues of system security as discussed in [21].

The remainder of this report is organized as follows. In the next section, we provide a general overview of the architectural design and verification process. In Section 3, we provide background information for our example problem, the synchronization logic of a Dual Flight Guidance System. In Section 4, we identify a few of the safety requirements our system must meet. In Section 5, we describe an implementation architecture for this example that satisfies the safety properties under ideal conditions in which all components execute synchronously and no components fail. In Section 6, we extend this model by allowing the components to execute asynchronously and show how this forces us to both relax the safety properties of the system and to change the behavior of individual components. In Section 7, we consider the extensions needed if individual components can fail. In Section 8, we generalize this into an overall process for system development. In Section 9, we close with conclusions and directions for future work.

2 Overview of the Architectural Design and Verification Process

A high level view of the proposed architectural design and verification process is shown in Figure 1.The *ideal system specification* describes the behavior of the system we would like to have if cost (and perhaps even the laws of physics) were not an issue. The *ideal system implementation* defines a set of components (subsystems) and their interconnections that collectively implement the behavior of the ideal system specification. This process can be repeated by stepwise refinement for each component until sufficient detail is reached so that the system can actually be constructed. This process is well understood and is traditionally referred to as *design refinement*. We refer to the opposite relationship, i.e., mapping the system implementation into the system specification, as *design abstraction*.

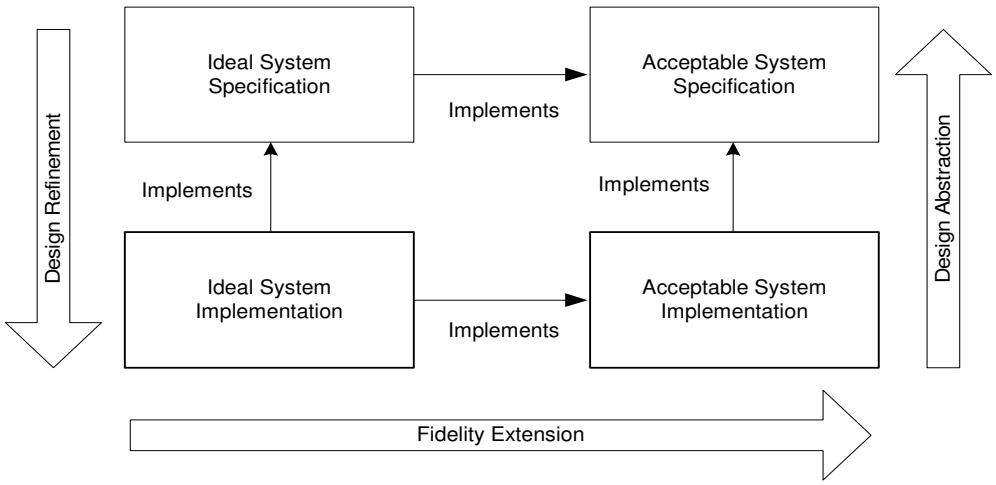


Figure 1 : The Architectural Design Process

It is often in the process of trying to implement the ideal system specification that we realize that perfection is either unrealizable or too costly. For example, the ideal specification may state that two displays will always be consistent, but in trying to implement this, we discover that making both displays change at the same instant is too expensive. In reality, we are willing to accept the displays being inconsistent for some brief period if this brings the cost down to a reasonable level without compromising safety or usability. The *acceptable system specification* relaxes the requirements of the ideal system specification to describe the system we are actually willing to accept. In effect, we acknowledge that the ideal system behavior is overly restrictive and that we did not identify the true system requirements.

It is not surprising that we need to relax the system requirements. The ideal system specification is almost always cleaner and simpler to state than one with all the exceptions we are actually willing to live with. For this reason, it is almost always what we write down when first starting to define a system. However, if that ideal is not achievable at reasonable cost, we must settle for something less if anything is to actually be built.

Since the ideal system behavior can always be substituted for the acceptable system behavior, it is by definition an implementation of the acceptable system behavior. Just as in stepwise refinement, there can be many such steps, each extending the acceptable system behavior a bit further towards the true system requirements. We refer to the relationship of extending the ideal system behavior to an acceptable behavior as *fidelity extension*.

Even if we are not relaxing the acceptable system behavior, our first conjectures about its behavior are almost always overly simplistic. In the process of showing that a model satisfies certain properties, we are more likely to discover the properties are wrong and need to be strengthened than that the model is wrong and needs to be corrected. For this reason, we often find ourselves going back and clarifying the properties the system needs to satisfy.

All of this suggests an architectural design process in which we start with the ideal specification we would like to achieve and attempt an implementation of that system. As we discover features that are too costly or impossible to implement, we return to our system specification, relax it to one that can be implemented, confirm that this new specification is still acceptable, and try again. In this way we progress in small steps from the unaffordable ideal system we would like to have to an acceptable design that meets our needs at reasonable cost.

Ideally, fidelity extension would be completely independent of design refinement. That is, fidelity extensions could be made without having to change the behavior of individual components. On the surface, this seems reasonable. Design refinement typically refines a single component into a more detailed implementation, while fidelity extension seems to be more concerned with the way components interact. In fact, the extent to which we have to modify individual components as we change the system architecture could be taken as a measure of the improper allocation of functionality between the system architecture and its components.

In the following sections, we illustrate this process through a simple example involving the synchronization between two Flight Guidance Systems (FGS). We start with the ideal case in which both FGS are assumed to not fail and to execute in lockstep over a synchronous communication channel. We then allow the two sides to execute asynchronously and show how this requires us to relax the system requirements and to change the behavior of each FGS in order to meet even these relaxed safety requirements. Following this, we make the fidelity extension of allowing each FGS to fail and explore how the system requirements have to be relaxed and what changes need to be made to each FGS.

Throughout this process, we use the NuSMV model checker [15] to formally prove that each implementation meets the revised safety requirements. NuSMV is a re-implementation and extension of SMV [7], [10], the first model checker based on Binary Decision Diagrams (BDDs). We automatically convert our models written in Simulink [9] into NuSMV using translators developed as part of this project. The properties to be verified are then specified using either Computation Tree Logic (CTL) or Linear Time logic (LTL) [7], [10] and checked using the model checker.

Unfortunately, each fidelity extension of our example requires us to change the behavior of the individual components in order to maintain the system safety requirements. This suggests that the synchronization logic should actually be part of the system architecture rather than the individual components. We discuss this further in the conclusions in Section 9.

3 The Dual FGS Example

To illustrate our approach to architectural analysis, we use a portion of a Flight Control System (FCS), the Flight Guidance System (FGS), shown in Figure 2. The FGS compares the measured state of the aircraft (position, speed, and attitude) to the desired state and generates pitch and roll guidance commands to minimize the difference between the measured and desired state. The FGS subsystem accepts input about the aircraft's state from the Air Data System (ADS) and Flight Management System (FMS). Using this information, it computes pitch and roll guidance commands that are provided to the autopilot (AP). When engaged, the autopilot translates these commands into movement of the aircraft's control surfaces necessary to achieve the commanded changes about the lateral and vertical axes.

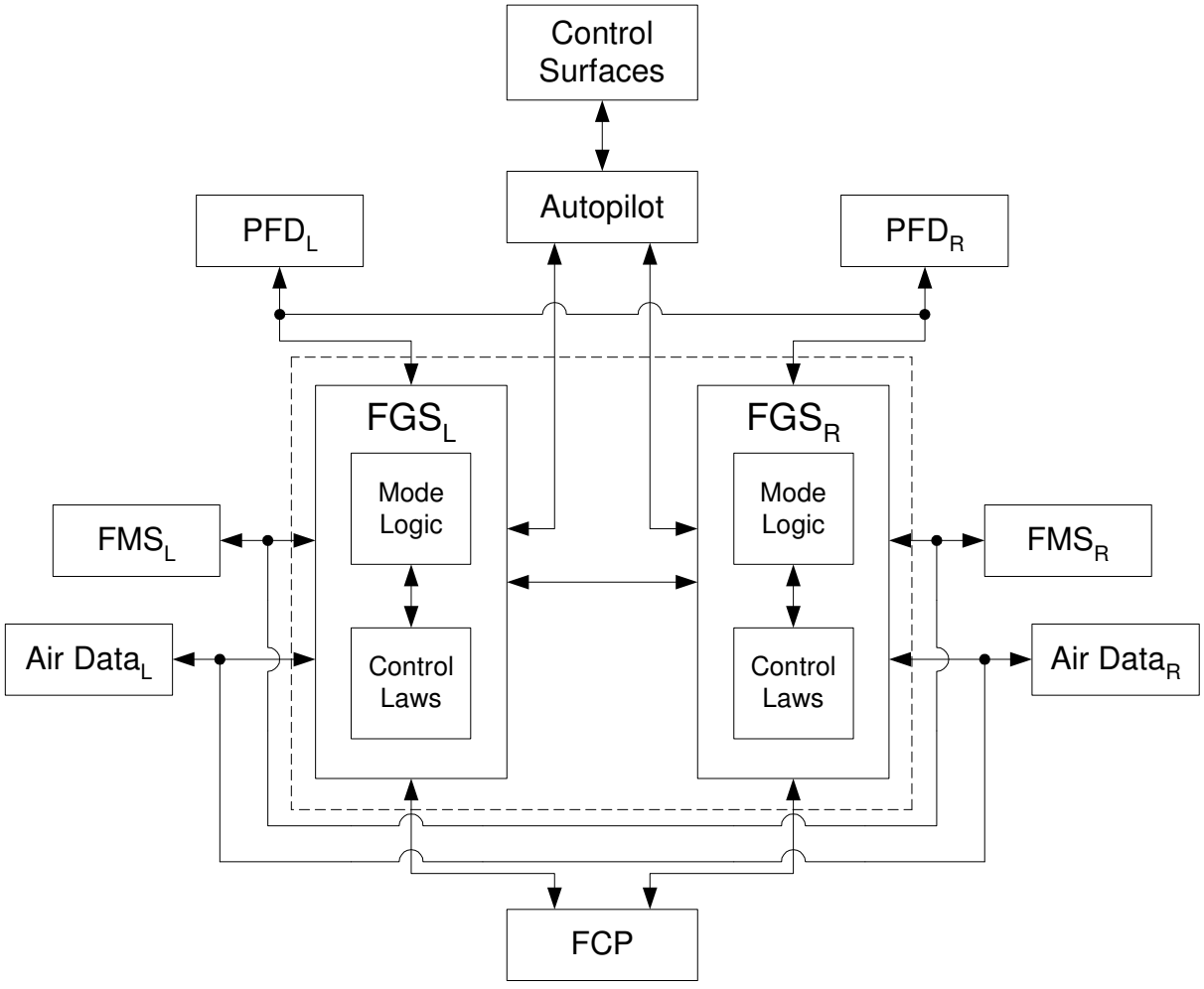


Figure 2 : Simplified Flight Control System. Two FGS system enclosed in dashed box.

The flight crew interacts with the FGS primarily through the Flight Control Panel (FCP). The FCP includes switches for turning the FGS on and off, switches for selecting the different flight modes such as Vertical Speed and Hold Selected Heading, setting reference values such as the

desired vertical speed or heading, and engaging or disengaging the AP. The FCP also supplies feedback to the crew through lamps indicating which switches have been selected.

The FGS has two physical sides corresponding to the left and right sides of the aircraft. These provide redundant implementations that communicate with each other over a cross-channel bus. Normally, only one FGS (the pilot flying side) is active, with the other FGS operating as a silent, hot spare. In this *dependent* mode of operation, the active FGS provides guidance values to the AP and the Flight Director (FD) (displayed to the pilot on each Primary Flight Display (PFD)). The pilot and copilot can switch which side is the pilot flying side by pressing the *Transfer Switch* on the FCP. This is frequently done when switching to a different navigation source.

However, in some critical modes, such as Approach and Go Around, both sides are active and independently generate guidance values for their own FD. In this *independent* mode of operation, both sets of guidance values are provided to the AP, which first verifies that they agree within a predefined tolerance value. If in agreement, the values are averaged and executed. If not in agreement, the situation is annunciated to the pilot and the AP disconnects.

Each side of the FGS can be further broken down into the mode logic and the flight control laws. The flight control laws accept information about the aircraft's current and desired state and compute the pitch and roll guidance commands. The mode logic determines which lateral and vertical modes of operation are active at any given time. These in turn determine which flight control laws are active. These are annunciated (displayed) by lamps on the FCP and text annunciations on the PFD along with a graphical depiction of the flight guidance commands generated by the FGS. A more detailed description of the FGS can be found in [18].

To make our example tractable, we restrict ourselves to a very simplified specification that deals only with the logic determining whether that side is the active side. While only a small part of the functionality of a FGS, it is still a useful example as the coordination of which side is active is a critical system function and there are several system safety properties that we are interested in, such as showing that at least one side is active at all times.

4 Determining the System Safety Properties

The first step is to specify the desired behavior of the system. We limit this to informally specifying the safety requirements for the dual FGS system, determined by traditional safety analysis techniques such as fault tree analysis and failure effects analysis [1], [8], [17], [23], [27]. We do not discuss these techniques here as they are well understood and described in detail in the literature. Instead, we simply present the necessary properties of the system[1].

Property 1 - At least one FGS shall always be active.

Whenever the system is turned on, at least one FGS must always be providing valid guidance values. This is necessary to ensure that valid guidance is being provided to the pilot and/or the autopilot.

Property 2 – Exactly one side shall be the pilot flying side.

Whenever the system is turned on, one and only one side shall be selected as the pilot flying side. This ensures that both sides do not try to provide guidance to the pilot and/or autopilot while the system is in dependent mode of operation.

Property 3 – If the system is in independent mode, both FGS shall be active.

If the system is in independent mode, both FGS must be active and sending valid guidance commands to be averaged by the PFD and/or the autopilot. This is necessary to meet system safety requirements during critical phases of flight.

In the following sections, we will formalize these requirements for the ideal system implementation, a fidelity extension that introduces asynchronous communication, and a fidelity extension that introduces component failures. For each case, we either prove the properties hold or show how they must be relaxed in order to be proven.

[1] While our properties are realistic, they are by no means meant to be a complete set.

5 Modeling and Analysis of the Ideal System

In this section, we model and analyze the dual FGS system assuming that neither FGS can fail, that they execute synchronously, and that they communicate over a channel that does not lose, corrupt, or generate spurious data. We first model the system architecture, then model the component behavior required to meet the system safety properties, and finally prove the safety properties are satisfied.

5.1 Defining the System Architecture

The first step is to specify the components, connections, and inputs and outputs of the overall system. In the context of Figure 1, this would be the design refinement step from the ideal system specification to the ideal system implementation.

Based on the discussion of Section 3, we would expect our system to consist of a left and a right FGS and the cross channel bus that connects them as shown in Figure 3. System inputs would include the sensor inputs to each FGS and the pilot inputs from the FCP. System outputs would be the outputs of each FGS.

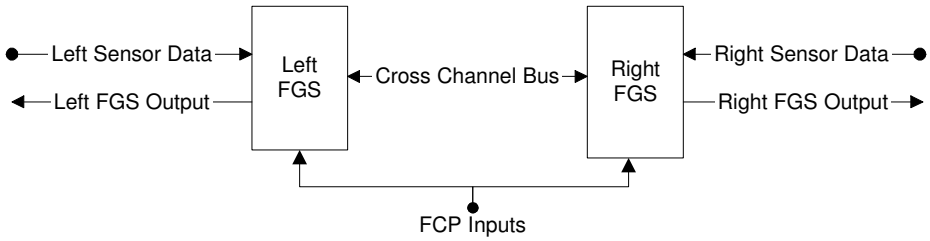


Figure 3 – System Architecture for the Dual FGS Example

For an actual system, the sensor inputs, the FCP inputs, the outputs, and the data exchanged over the cross channel bus are quite extensive. However, since we are only interested in those aspects of the system related to the synchronization of active status, we can use an abstraction of the full system to keep the analysis tractable.

It turns out that the desired system properties do not depend on how the independent mode of each FGS is actually computed. This allows us to model these quantities as unconstrained inputs to each FGS. If we can prove the system properties when these values are unconstrained, then we have also proven them for an implementation in which they are constrained. This also means that we can change the way in which independent mode is computed and still be assured that the synchronization logic is correct.

This also allows us to ignore the sensor inputs since their only impact on the active status is through the computation of independent mode. The same situation holds for all the inputs of the FCP except for the transfer switch, so we can reduce the FCP inputs to this one value.

A Simulink [9] model of the ideal system architecture is shown in Figure 4. The system inputs are the 1) *Transfer Switch*, 2) *Left Independent Mode*, and 3) *Right Independent Mode.* The system outputs are the 1) *Left FGS Active* and 2) *Right FGS Active* values describing whether each side believes it is active.

The *Pilot Flying* and *Independent* status of each side is communicated through a channel to the other side. These two values convey each side's determination of whether it is the pilot flying side and whether it is in independent mode. In this idealized synchronous example, the channels simply hold their inputs for one step and then pass them on to the other side.[2]

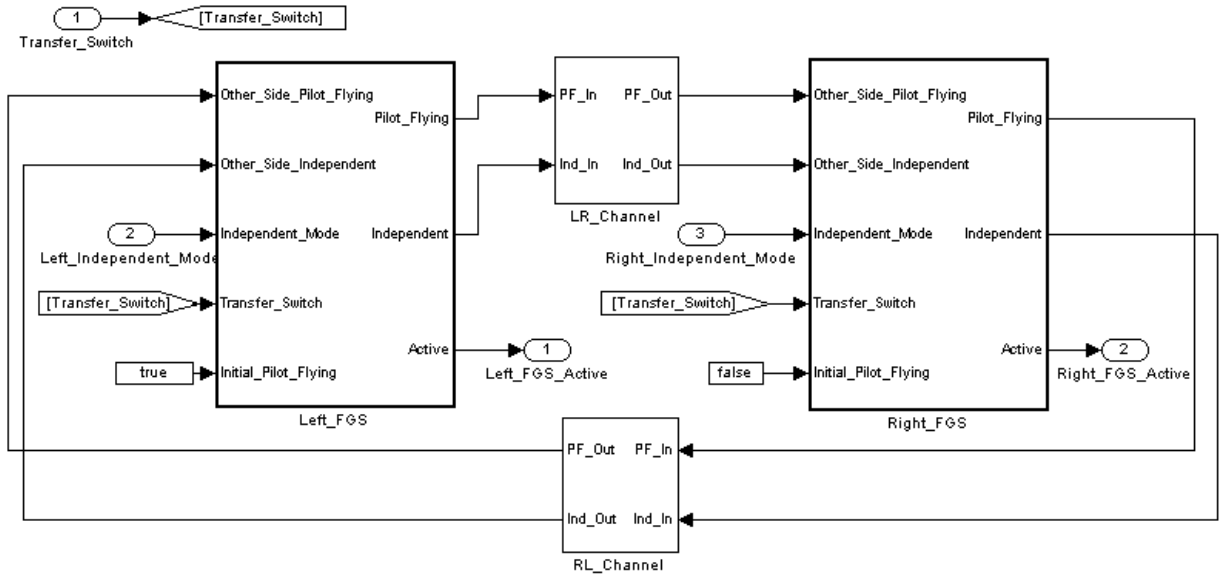


Figure 4 - System Architecture in the Ideal Case

5.2 Proving the System Safety Properties

Before we can prove whether the system safety properties hold for the ideal system, we must define the subset of the behavior of each component that is relevant to the overall system behavior we are trying to establish. In this example, we must specify how each FGS determines its active status. We choose to specify this as a constructive model in Simulink as shown in Figure 5. This specification is an abstraction of the full specification of the FGS obtained by abstracting away all parts of the full FGS not related to the determination of active status.

The active status of the FGS is set by the OR block in the lower right of Figure 5. The FGS is defined to be active if both sides are in independent mode or it believes itself to be the pilot flying side. Here, the determination of whether both sides are in independent mode represents this side's estimate of whether the overall *system* is in independent mode. To allow the other side to make a similar assessment of the system state, this side's local computation of independent mode is sent to the other side via the channel.

The remaining logic is concerned with determining if this is the pilot flying side. An FGS will become the pilot flying side if it is *not* the pilot flying side and it sees the *Transfer Switch* being

[2] To keep the example tractable, each channel sets its initial output to be the correct value for the side it is transmitting from. This assumes that both sides start at the same instant. In an actual implementation in which each side could start at different times, there would need to be a startup phase in which the system established a consistent initial state.

pressed. This is implemented by the upper switch of Figure 5. If the FGS *is* the pilot flying side, it will become the pilot *not* flying side when it observes the other side become the pilot flying side. This is implemented by the lower switch of Figure 5. If neither of these events occurs, the FGS leaves its pilot flying status unchanged. In this way, the pilot not flying side always initiates the change of which side is the pilot flying side, eliminating the possibility of neither side being the pilot flying side.

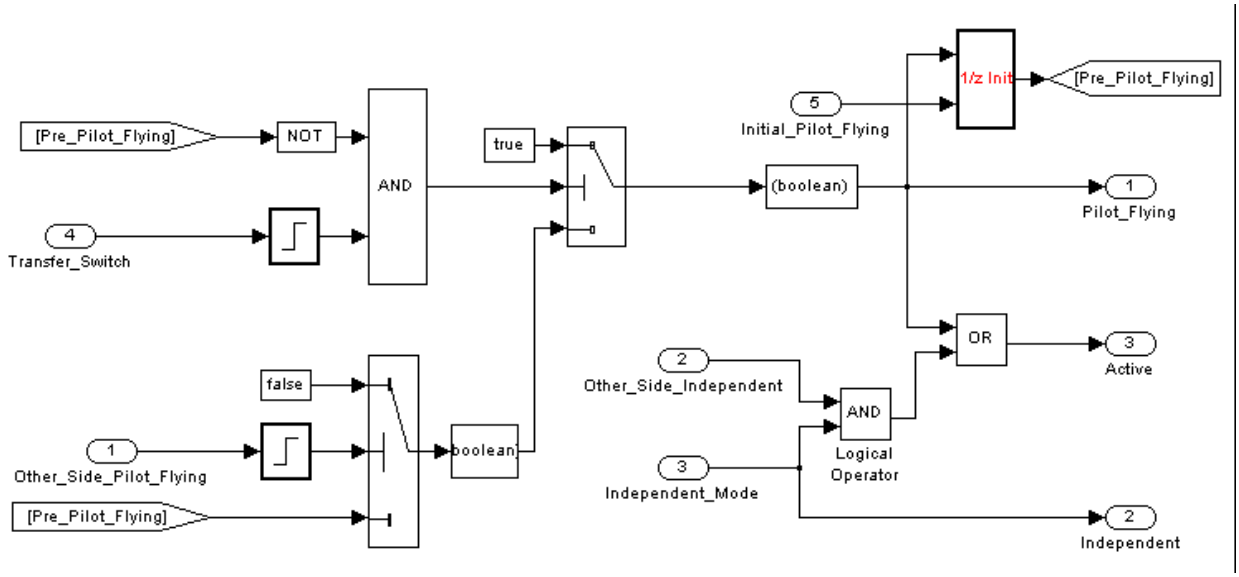


Figure 5 - Synchronization Logic of a Single FGS in the Ideal Case

The next step is to formalize the safety properties identified in Section 4 and use the model checker to determine whether they hold for the overall system. For the ideal system, this is straightforward and consists of translating each property into a combination of SMV and CTL so that it can be checked using the NuSMV model checker.

Property 1 - At least one FGS shall always be active

To show this, we first define Property 1 in NuSMV as

```
Property1 := (Left_FGS_Active | Right_FGS_Active);
```

and then attempt to prove the CTL property

```
AG(Property1)
```

which states that Property1 is "always globally" true, i.e., it holds for all reachable states. This is easily shown to hold for the ideal system using the NuSMV model checker.

Property 2 – Exactly one side shall be the pilot flying side.

We first define Property 2 in NuSMV as

```
Property2 := (Left_FGS_Pilot_Flying = !Right_FGS_Pilot_Flying);
```

and again attempt to prove the CTL property

```
AG(Property2)
```

To our surprise, this property does not hold for all reachable states. The counterexample (which requires a single step) is shown below.

		Step 1
INPUTS		
	Transfer_Switch	1
	Left_Independent_Mode	1
	Right_Independent_Mode	1
INTERNAL VARIABLES		
	Right_FGS.Pilot_Flying	1
	Left_FGS.Pilot_Flying	1
OUTPUTS		
	Left_FGS_Active	1
	Right_FGS_Active	1

In this case, the *Transfer Switch* was pressed in Step 1 and the pilot not flying (right) side has become the pilot flying side. However, the other (left) side has not yet observed this change due to the one step communication delay introduced by the channel, and will not become the pilot not flying side until the next step. While Property 2 will again hold in step 2, in step 1 both sides believe they are the pilot flying side.

Unfortunately, there is no way to change the model to satisfy both Property 1 and Property 2. Even if we wanted to eliminate the one step delay in the channel (effectively modeling a channel with no delay at all), doing so would introduce an illegal cyclic dependency (algebraic loop) into our synchronous model. Checking with the system safety engineers, we learn that it is actually acceptable for this property to not hold for one step, and we can relax our safety property to

```
AG(!Property2 -> AX(Property2))
```

The CTL format `AG(!P -> AX(P))` states that for all states, if the property P does not hold in a reachable state, then it must hold in all next states. This effectively states that the property can be false for at most one step. This CTL property is easily shown to be true with the model checker, establishing that neither or both sides can be the pilot flying side for at most one step.

Property 3 – If the system is in independent mode, both FGS shall be active.

Formalizing Property 3 raises a new question – what does it mean for the *system* to be in independent mode? Each individual FGS has its own input indicating if it is in independent mode, and we have specified that an FGS will be active if both it and the other side are in independent mode. Based on this, we decide that the correct interpretation of the informal safety property is that the system is in independent mode if both sides are simultaneously in independent mode (i.e., their *Independent_Mode* inputs are true). We define this to the NuSMV model checker as

```
System_Independent_Mode :=
     Left_Independent_Mode & Right_Independent_Mode;
```

Property 3 then can then be defined as

```
Property3 := (System_Independent_Mode ->
                    (Left_FGS_Active & Right_FGS_Active));
```

Unfortunately, Property 3 does not hold for all reachable states, producing the following counterexample:

		Step 1	Step 2	Step 3
INPUTS				
	Transfer_Switch	1	1	1
	Left_Independent_Mode	1	1	1
	Right_Independent_Mode	1	0	1
INTERNAL VARIABLES				
	Right_FGS.Pilot_Flying	1	1	1
	Left_FGS.Pilot_Flying	1	0	0
OUTPUTS				
	Left_FGS_Active	1	1	0
	Right_FGS_Active	1	1	1

In the first step of this counterexample, both sides are in independent mode and are thus both active (just as in the analysis of Property 2, they both believe they are the pilot flying side since the right side has just seen the *Transfer Switch* pressed and changed to be the pilot flying side). In step 2, the left side becomes the pilot not flying side as it receives the notification that the right side is now the pilot flying side. Also in step 2, the pilot flying (right) side drops out of independent mode as it has its *Independent Mode* input go false. Even though it is no longer in independent mode, the right side remains active since it is the pilot flying side. In addition, the pilot not flying (left) side also remains active since it has not yet received the information that the pilot flying (right) side is no longer in independent mode. In step 3, the pilot flying (right) side re-enters independent mode, so both sides now believe they are in independent mode. However, the pilot not flying (left) side now receives the information that the pilot flying (right) side dropped out of independent mode in step 2, so it becomes inactive. Thus, in step 3 both sides are in independent mode, but the pilot not flying (left) side is not active, violating our safety property.

Just as with Property 2, we suspect that Property 3 can be false for at most one step, so we relax our property and attempt proving

```
AG(!Property3 -> AX(Property3))
```

This turns out to indeed be true, establishing that Property3 can be false for no longer than one step. Checking with the system safety engineers, they confirm that this is also an acceptable behavior.

5.3 Observations on Proving the Safety Properties for the Ideal Case

We have confirmed that the ideal, synchronous architecture without failures satisfies the three system safety properties. However, even under these circumstances we had to relax the safety properties to deal with the reality of communication channels that introduce a one step delay. Even in the ideal case, our intuitive notion of the safety requirements was not completely correct. In the next section, we will introduce asynchronous behavior and find that we have to relax the safety properties much further.

6 Fidelity Extension - Introducing Asynchrony

In this section, we will consider the effect of allowing the two FGS and their connecting channels to execute asynchronously. This would be the situation if each component executed on its own processor with its own clock. While such Globally Asynchronous/Locally Synchronous architectures are quite common, this introduces the potential for race conditions, deadlock, and a variety of undesirable system behaviors. As we will see, the system safety properties have to be relaxed still further and their formalization becomes much more complex.

6.1 Defining the System Architecture

While conceptually a large change, the introduction of asynchrony only requires a small change to the system architectural model. As discussed in [3] and [13], this is done by introducing clocks for each component (*Left FGS Clock, LR Channel Clock, Right FGS Clock*, and *RL Channel Clock*), where a clock is a Boolean signal that constrains the component to execute only when the clock is high (see Figure 6)

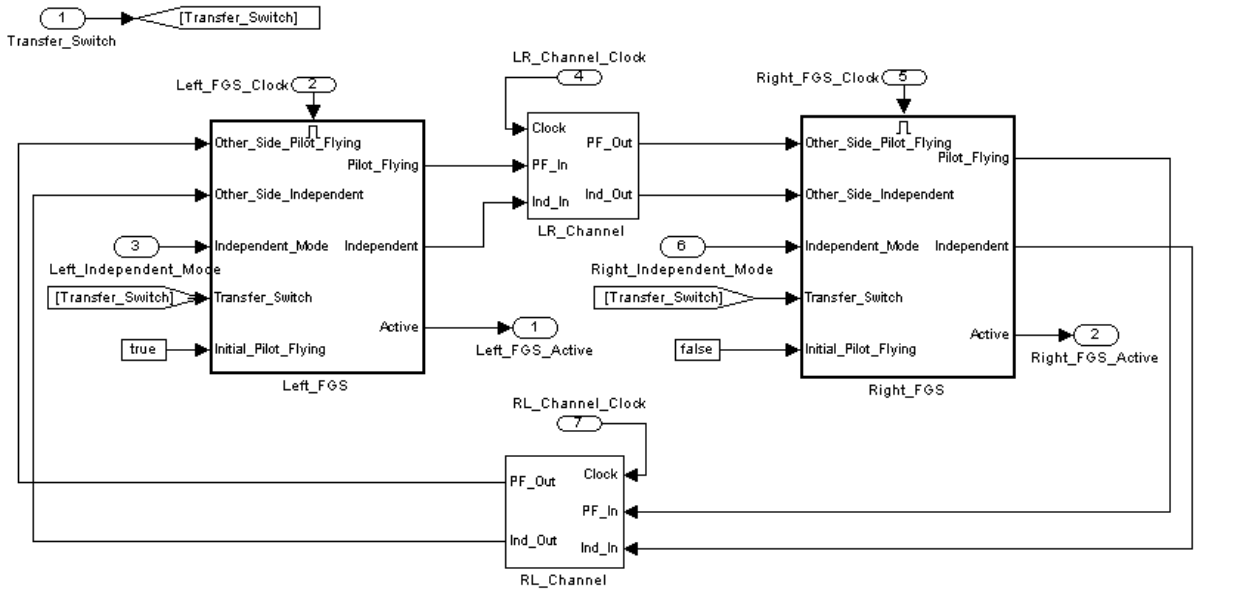


Figure 6 – System Architecture with Asynchrony

By leaving the clocks as unconstrained system inputs, we can analyze a completely asynchronous system. To revert back to a synchronous model, we constrain the clocks to be identical inputs tied to the global system clock of the modeling environment. Variations, such as bounded asynchrony, can be modeled by defining weaker constraints on the clocks. While it might appear that we have changed the system interface by introduction of the clocks, this is not really the case. In the synchronous case, the global system clock is implicitly provided by

Simulink as an input to each FGS and each channel. The lack of clock inputs in the ideal case is more an artifact of our modeling tools than an actual difference in the interface.

6.1.1 Modeling the Communications Channels

The reader may have noticed that we introduced clocks differently for the channels than for the two FGS in Figure 6. This is because we want to create a model that encompasses all the possible behaviors of the real channels of the system of interest in a concise way. This is difficult in a completely asynchronous environment because we do not constrain the clocks for the different processes. This means that most of the techniques that are used for buffering inputs between processes in "real" systems (e.g., using finite queues) will eventually overflow in a completely asynchronous environment.

We must also decide how to model the communications delays introduced by the channels. As in the synchronous model, we start with the assumption that communication between processes is not instantaneous. Within the Simulink modeling notation, this means that there must be a one-step delay somewhere between the sending and the receiving process. In addition, we must decide whether the communication delays introduced by the channels are negligible. If so, then we can simply use the channels created for the synchronous model that introduce a single step delay. If not, we need to introduce a channel clock to model these delays.

For the purposes of this example, we have chosen to model a lossy network with no queuing that may introduce unbounded delays. This yields a simple implementation for the channels and is sufficient to capture a wide range of complex system behaviors due to channel delays.

In order to model the additional delay introduced by the channel, we first create a very simple clocked process that passes through signals and add an enabling clock as shown in Figure 7. This process will act as a "gatekeeper" that will only pass through its input in the instants when its clock is high.

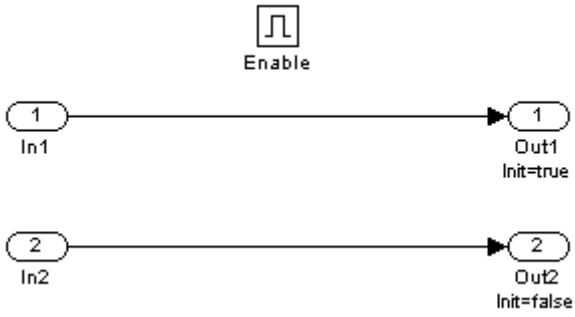


Figure 7 – A Simple Clocked Channel

Next, to satisfy our constraint that communication is not instantaneous, we embed this process inside another process that adds a single step delay, as shown in Figure 8. The result is a channel that passes through inputs with a one-step delay whenever its clock is high, and ignores the inputs otherwise.

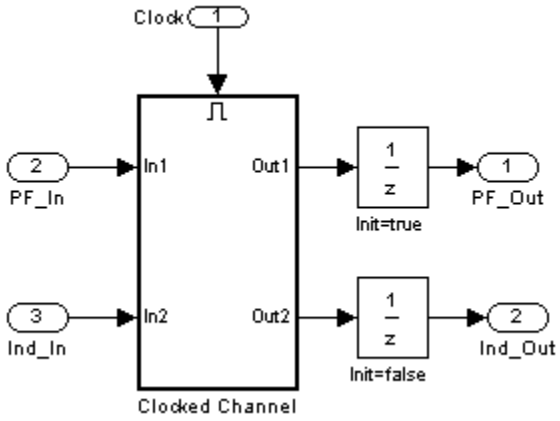


Figure 8 - Complete Asynchronous Channel

Is this sufficient to model arbitrary communication delays? The answer is yes, but the reason is a bit subtle. We are looking at the system through the lens of safety properties (more formally ACTL properties[3]) which assert that the property of interest must hold in *all possible execution paths* [7].

To accurately model communications delay, we need to capture the situation in which the left (right) FGS executes, followed by an arbitrary period of time before the right (left) FGS is able to see the result. This behavior is captured by an execution sequence in which the left side executes one step (because its clock is high), then does not execute again until some later point at which the channel executes, passing the message across to the right side. In the mean time, the right side may be executing, but it cannot see the results from the last execution of the left side. Since this is one of the possible behaviors of our system, and the model checker checks all possible behaviors, this scenario will be checked in the process of verifying the safety property.

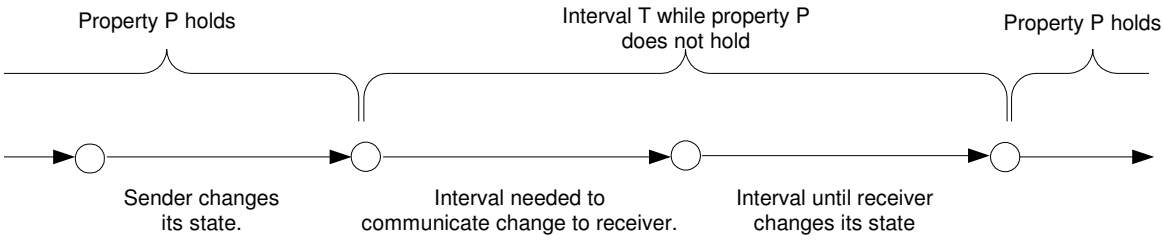
6.2 Proving the System Safety Properties

The introduction of asynchrony does not change the intent of the system safety properties identified in Section 4, but several of the properties do have to be relaxed. This is because system properties in an asynchronous environment often do not hold while a change in the system state propagates throughout the system. In the next section, we discuss in detail how the safety properties must be changed due to the introduction of asynchrony. In Section 6.2.2, we prove the actual safety properties.

[3] An ACTL property is a CTL property in which, after moving negations inwards to atomic formulae, do not contain any 'E' modalities. For example: AX (AG(Property1)) is in ACTL. The property !(EX (EX (Property1)) is also in ACTL, because it can be written as: (AX (AX (!Property)) by moving the negation inwards. However, the formula AX (EG(Property1)) is not in ACTL.

6.2.1 How Asynchrony Changes the System Safety Properties

For bounded properties, we are often interested in proving that the maximum time that some property fails to hold is bounded by the communication delay between the sender and receiver plus the delay until the receiver changes its state. For example, if one component changes its state so that some desirable safety property P no longer holds, it is impossible for the system to restore P until the change in that component's state is communicated to the other components and they can act upon the change.



Thus, if we have some safety property P that would always hold in the ideal case, the introduction of asynchrony may mean that the best we can do is to prove that P is false for no longer than some maximum duration T, i.e.,

$$\text{Duration } (!P) \leq T$$

In the synchronous case, it may be possible to reduce T to zero or to a single step. In the asynchronous case in which the component clocks are completely unconstrained, T can be unbounded since there are no guarantees that either the communication channel or the receiver will be enabled (i.e., its clock goes high) after the sender changes its state.

While the simplicity of using unconstrained clocks to model asynchrony is very appealing, it leaves us with the problem of how to formalize properties given that the communication delays may be infinite (since there are no constraints on the clocks). One solution is to simply state that property P can be false for no longer than the time required for the channel to communicate information from the sender to receiver plus the time required for the receiver to change its state. So long as we know that in the actual system this sum is always less than some acceptable period, proving this property is sufficient to prove the desired behavior.

We can use this approach to state that "property P can be false for no longer than the time required for the channel to communicate information from the sender to receiver plus the time required for the receiver to change its state" as the temporal logic property

```
AG(!P & Sender_Clock ->
     (A[!Channel_Clock U (P |
          (AX A[! Receiver_Clock U P]))]))
```

This expression uses the CTL operator A [X U Y], which requires that property X must hold in all paths starting from the current state until property Y holds. To understand the overall CTL formula, consider the example of Figure 9.

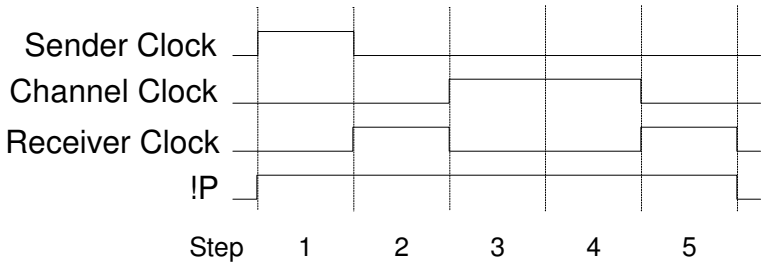


Figure 9 - Asynchronous Example Fails

In this example, the CTL formula fails because the property P is false for longer than the time that it takes to communicate the sender's change plus the time it takes for the receiver to change its state. In step 1, the sender has just changed its state such that the system property P spanning the sender and the receiver is no longer true. As a result, the antecedent of the formula (!P & Sender_Clock) is satisfied, so the consequent A[!Channel_Clock U (P | (AX A[!Receiver_Clock U P]))] must also be true. This consequent is satisfied in step 1 because the *Channel Clock* is false, so the change in the sender's state could not yet have been transmitted to the receiver. It is also satisfied in step 2 for the same reason.

In step 3, *Channel Clock* becomes true, allowing the channel to accept the sender's output. By the definition of UNTIL, the right side of the formula, (P | (AX A [!Receiver_Clock U P])), must now hold. If P were true (i.e., if the receiver has somehow changed such that the desired system property were true), the overall property would hold. Since P is not true in step 3, the formula (AX A [!Receiver_Clock U P]) must hold. The AX operator requires that the property must be true in all *next* states after step 3, so the property holds in step 3 if it holds. The AX operator is necessary since even if the *Receiver Clock* is true and the receiver is ready to input from the channel, it will still take one step for the value to move through the channel. In step 4 the formula is true because the *Receiver Clock* is false, making A [!Receiver_Clock U P]) true. However, in step 5, *Receiver Clock* is true, allowing the receiver to change its state (hopefully to restore property P). However, in this example P is still false, indicating that P is failing for some reason other than the introduction of asynchrony.

This CTL pattern accounts for the two delays, one due to the channel communication and one due to the processing of this information by the receiver. Since the clocks are unconstrained, these intervals are both potentially unbounded. In this case, T becomes infinite and the formula will still be true, even though P is never restored. However, if both intervals are bounded, the formula will be satisfied only if P is restored in less than T.

It also does not matter if the receiver's clock becomes active prior to the completion of the communication. For example, the execution of *Receiver Clock* in step 2 is irrelevant as the data from the sender has not yet reached the receiver. Also, note that the CTL formula allows P to become true in steps 2 through 4 due to some unrelated change in the state of the Receiver. Since we are only interested in bounding the time that P can be false, this is also acceptable.

Another example in which the CTL formula succeeds is shown in Figure 10. Just as with the previous example, the antecedent (!P & Sender_Clock) is satisfied in step 1, so the consequent

A[!Channel_Clock U (P | (AX A[!Receiver_Clock U P]))] must also be satisfied. It is satisfied in step 1 because the *Channel Cl*ock is false, so that the change is not transmitted from the sender to the receiver. It is satisfied in step 2 for the same reason. In the third step, *Channel Clock* is true, so by the definition of UNTIL the remainder of the formula, (P | (AX A [!Receiver_Clock U P])), must be satisfied. This is true in step 3 because of the AX operator. In step 4, *Receiver Clock* is true, so P must be satisfied, which it is.

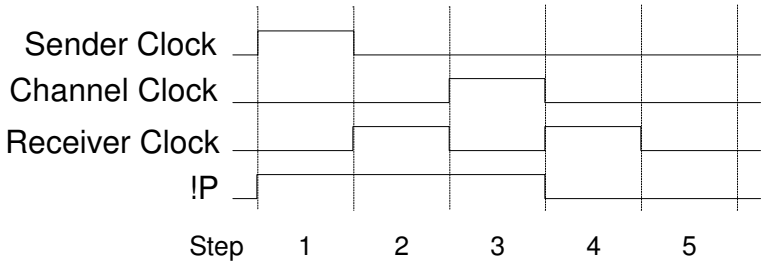


Figure 10 - Asynchronous Example Succeeds

There is one additional extension that must be made to the template for our CTL formula. The current template assumes that all changes that can make the system property P temporarily invalid start on the sender side. However, our dual FGS system has two identical sides and a system property P can be made temporarily invalid by a change that starts on either the left or the right side. To deal with this, we extend our CTL template to

```
AG( ((!P & Left_FGS_Clock) ->
          (A [!LR_Channel_Clock U (P |
               (AX A [!Right_FGS_Clock U P]))])) |
     ((!P & Right_FGS_Clock) ->
          (A [!RL_Channel_Clock U (P |
               (AX A [!Left_FGS_Clock U P]))])) );
```

This is the general property we wish to prove to show that a property P can be false for no longer than one “communication cycle” from either side to the other.

6.2.2 Proving the System Safety Properties of the Asynchronous System

In this section, we prove the system safety properties for the asynchronous system. We start by trying to prove the properties for the ideal system proven in Section 5.2. If these fail, we will either strengthen the system model so that they succeed, weaken the safety property to one we are willing to accept, or some combination of both.

Property 1 – At least one FGS shall always be active.

As in the synchronous case, we define Property 1 in NuSMV as

```
Property1 := (Left_FGS_Active | Right_FGS_Active);
```

and then attempt to prove the CTL property

```
AG(Property1)
```

This is easily shown to hold for the asynchronous system using the NuSMV model checker. This is not surprising since the FGS logic was designed to make this property very robust.

Property 2 – Exactly one side shall be the pilot flying side.

In the synchronous model discussed in Section 5, we showed that property 2 had to be relaxed to account for the one step delay introduced by the communication channel. When asynchrony is introduced, this property must be relaxed still further since it will take longer than one step for the change in state of one side to propagate and be acted on by the other side.

This is done by changing the property as discussed in Section 6.2.1. The desired property P is still the same as for the synchronous case

```
Property2 := (Left_FGS_Pilot_Flying = !Right_FGS_Pilot_Flying);
```

However, instead of AG(Property2), the CTL property that we need to prove is

```
AG(   ((!Property2 & Left_FGS_Clock) ->
            (A [!LR_Channel_Clock U (Property2 |
                  (AX A [!Right_FGS_Clock U Property2]))])) |
      ((!Property2 & Right_FGS_Clock) ->
            (A [!RL_Channel_Clock U (Property2 |
                  (AX A [!Left_FGS_Clock U Property2]))])) );
```

This property states that the amount of time there is not exactly one pilot flying side is less than one communication cycle of the system.

However, model checking this property on the asynchronous model reveals yet another error. It is possible for the system to get into a state in which each FGS will permanently believe that it is the *Pilot Flying* side. The reason for this is subtle and is easiest to see by the counterexample shown in Figure 11. The root cause of the problem is that if two pilot flying transfers occur in quick succession, the FGS that should be the pilot not flying side may miss a crucial notification from the other FGS causing it to remain the pilot flying side.

As shown in Figure 5, in order for the pilot flying side to become the pilot not flying side, it must see the other side change from pilot not flying to pilot flying. However, in Figure 11, the left FGS changes to the pilot not flying side and then back to the pilot flying side during an interval where the right FGS does not execute because its clock is low (shown by the dark bar). Therefore, when the right FGS next executes, there is no change in its perceived status of the left FGS, so it does not relinquish the pilot flying side. The system then enters a state in which both FGS permanently believe they are the pilot flying side.

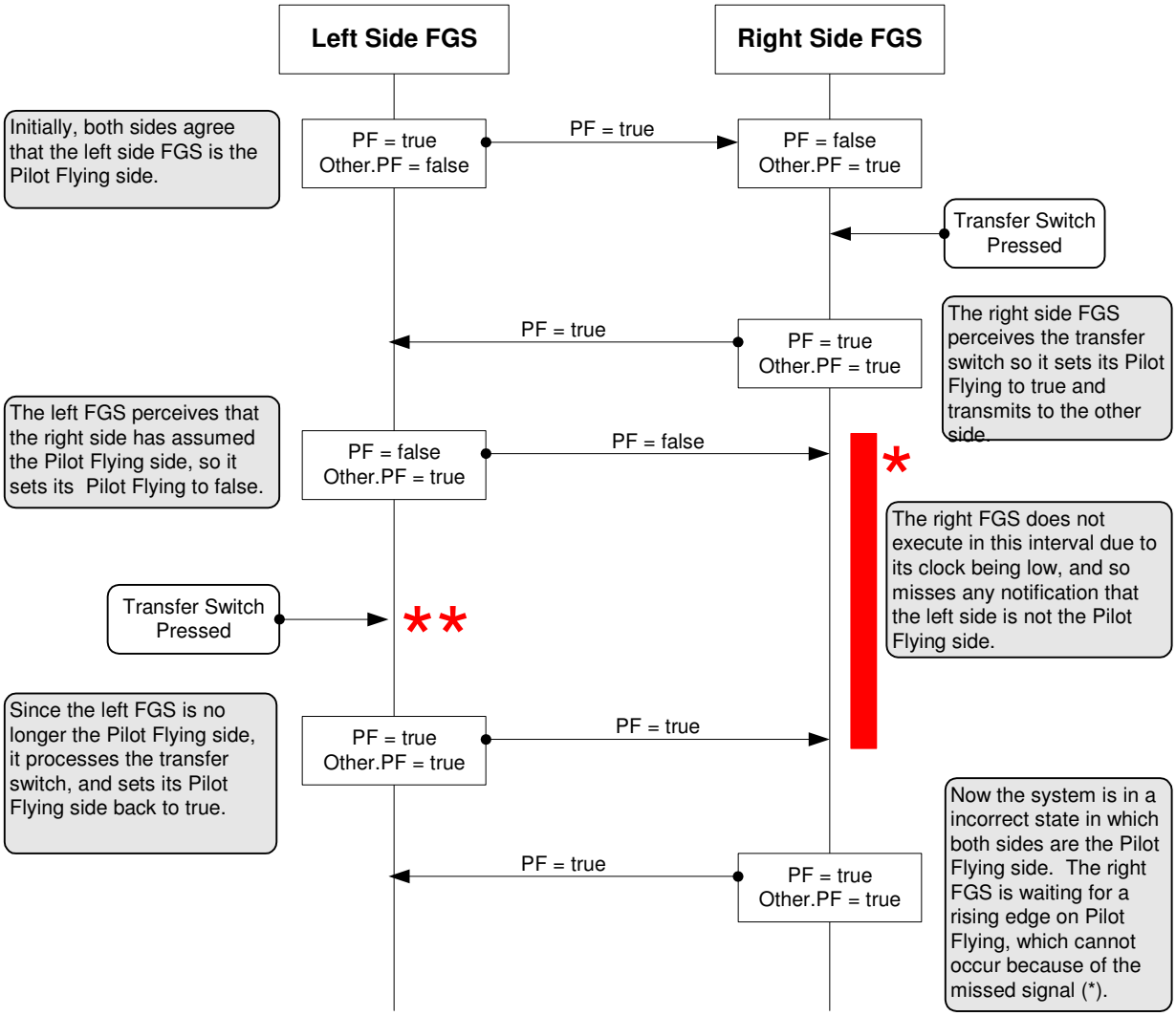


Figure 11 – Property 2 Counterexample for Asynchronous Model

In order to solve this problem, we have to ensure that the system is in a stable state in which both sides agree on which side is the pilot flying side before allowing the pilot not flying side to respond to the *Transfer Switch*. The mechanism that we use to accomplish this is an acknowledgement signal generated by each side. Most of the time, this signal is true when that side is the pilot flying side. It is set false when the current pilot flying side sees the other side become the pilot flying side. The new pilot flying side also keeps its acknowledgement signal false until it sees the other side lower its acknowledgement. During the (normally) brief period that a side has become the pilot not flying side and the acknowledgement from the other side is still false, the new pilot not flying side is inhibited from responding to the *Transfer Switch*. This ensures that the counterexample of Figure 11 cannot occur.

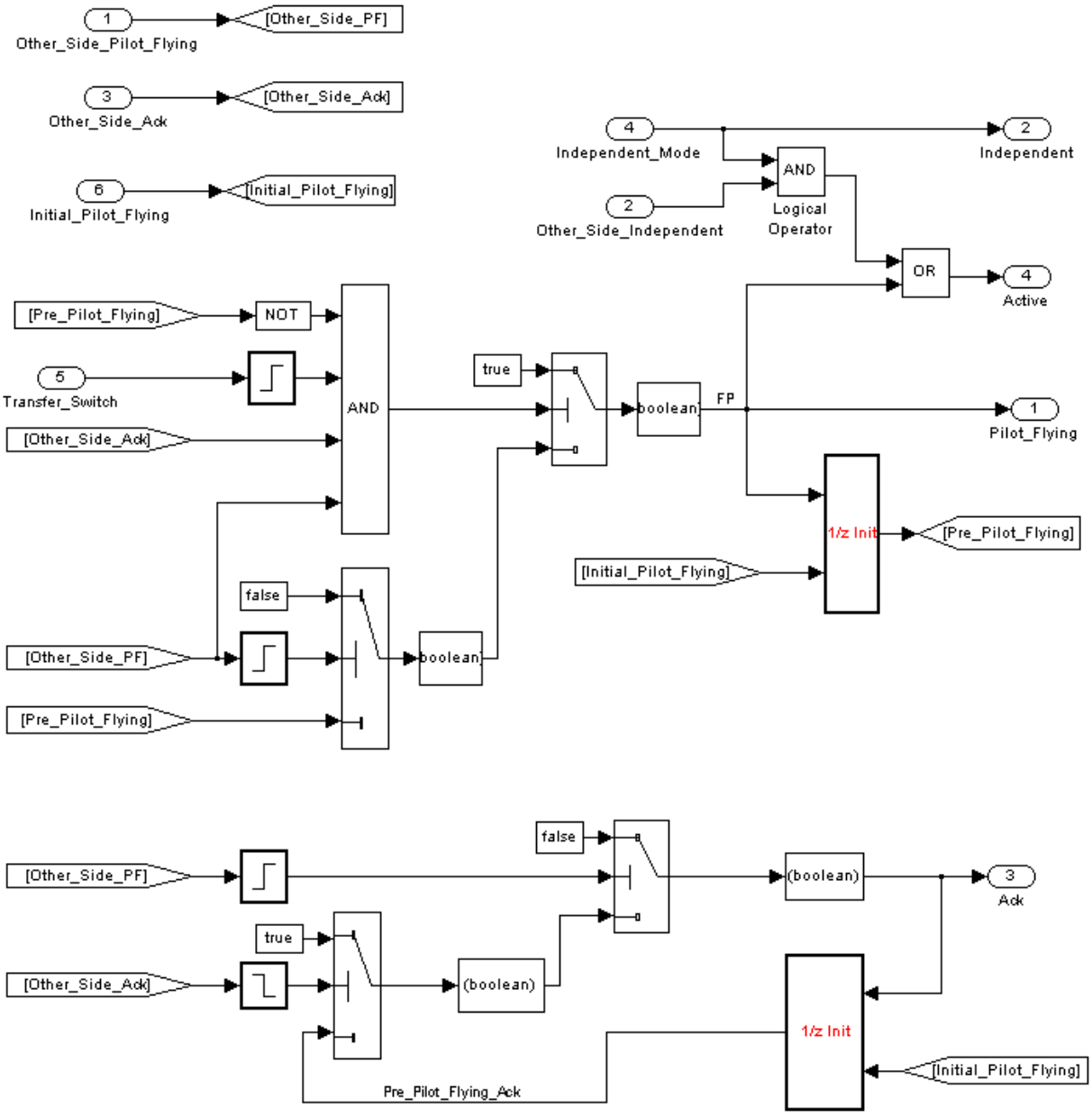


Figure 12 – Corrected Synchronization Logic of a Single FGS with Asynchrony

The details of the corrected logic are shown in Figure 12. In this figure, both the value of the *Pilot Flying* and the *Ack* from the other side are extracted from the channel (shown as inputs in the upper left hand corner of Figure 12). The pilot not flying side is prevented from becoming the pilot flying side when the *Transfer Switch* is pressed unless the other side has set its acknowledgement true (upper switch in Figure 12). Finally, this side's acknowledgement logic is specified in the lower part of the figure. With these changes, Property 2 can finally be shown to hold in the asynchronous model using the model checker.

Property 3 – If the system is in independent mode, both FGS shall be active.

Property 3 actually turns out to be simpler to prove than Property 2. As for the synchronous case, we formalize property 3 by defining

```
System_Independent_Mode :=
     Left_Independent_Mode & Right_Independent_Mode;

Property3 := (System_Independent_Mode ->
                      (Left_FGS_Active & Right_FGS_Active));
```

The CTL property we wish to prove then is

```
AG(   ((!Property3 & Left_FGS_Clock) ->
           (A [!LR_Channel_Clock U (Property3 |
                (AX A [!Right_FGS_Clock U Property3]))])) |
      ((!Property3 & Right_FGS_Clock) ->
           (A [!RL_Channel_Clock U (Property3 |
                (AX A [!Left_FGS_Clock U Property3]))])) );
```

stating that the maximum time that Property 3 can be false is less than one communication cycle. However, unlike Property 2, Property 3 turns out to be true for both the original synchronization logic of Figure 5 as well as the strengthened logic of Figure 12.

6.3 Observations on the Issues Introduced by Asynchrony

It is worth noting that even with asynchronous communication, we were able to design the system so that Property 1 would always hold. This appears to be a common situation for asynchronous systems - one can design the system so that some properties will always hold, but only at the expense of relaxing other properties.

However, even for a system as simple as this one, proving the other properties turned out to be much more complex than for the synchronous case. The properties themselves are more difficult to state, were weaker than could be achieved in the synchronous case, and required considerable complexity to be added to the model to ensure that even the weakened properties were true. Perhaps most disquieting, we probably would not have discovered the counterexample of Property 2 without the aid of automated verification.

7 Fidelity Extension – Introducing Component Failures

In this section, we will illustrate another fidelity extension by introducing failure modes to the asynchronous system of Section 6. To do this, we must add a notion of failure to our model.

7.1 Defining the System Architecture

Ideally, we would model failure of each FGS by adding an internal Boolean variable that indicated whether the component was failed. For purposes of simulation, this variable could be set by the user or assigned randomly according to some probability distribution. Unfortunately, our modeling and analysis tools do not provide direct support for modeling faults in this way, so we choose to model failure as unconstrained Boolean inputs *Left_Failed* and *Right_Failed*, much as we did with *Independent Mode* in Section 5.1. One might object that this changes our system interface, but this is again an artifact of our modeling environment. Ideally, we would prefer to be able to model internal failures without changing the interface at all. It is worth noting that the asynchronous model without failures can be obtained from this model by setting the failure inputs false. The revised architectural model is shown in Figure 13.

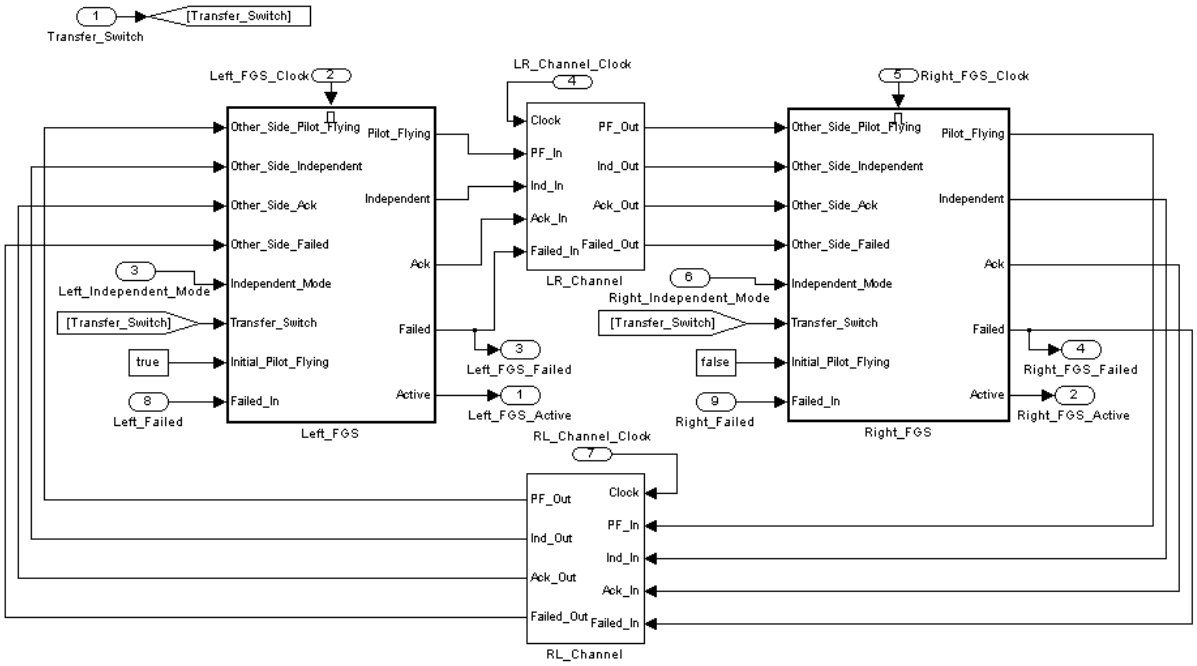


Figure 13 - Asynchronous System Architecture with Component Failures

7.2 Proving the System Safety Properties

As before, we must first define the new relevant behavior of each component. To deal with component failures, there are three changes (shown in Figure 14) that must be made to the logic shown in Figure 12 for the asynchronous case without failures

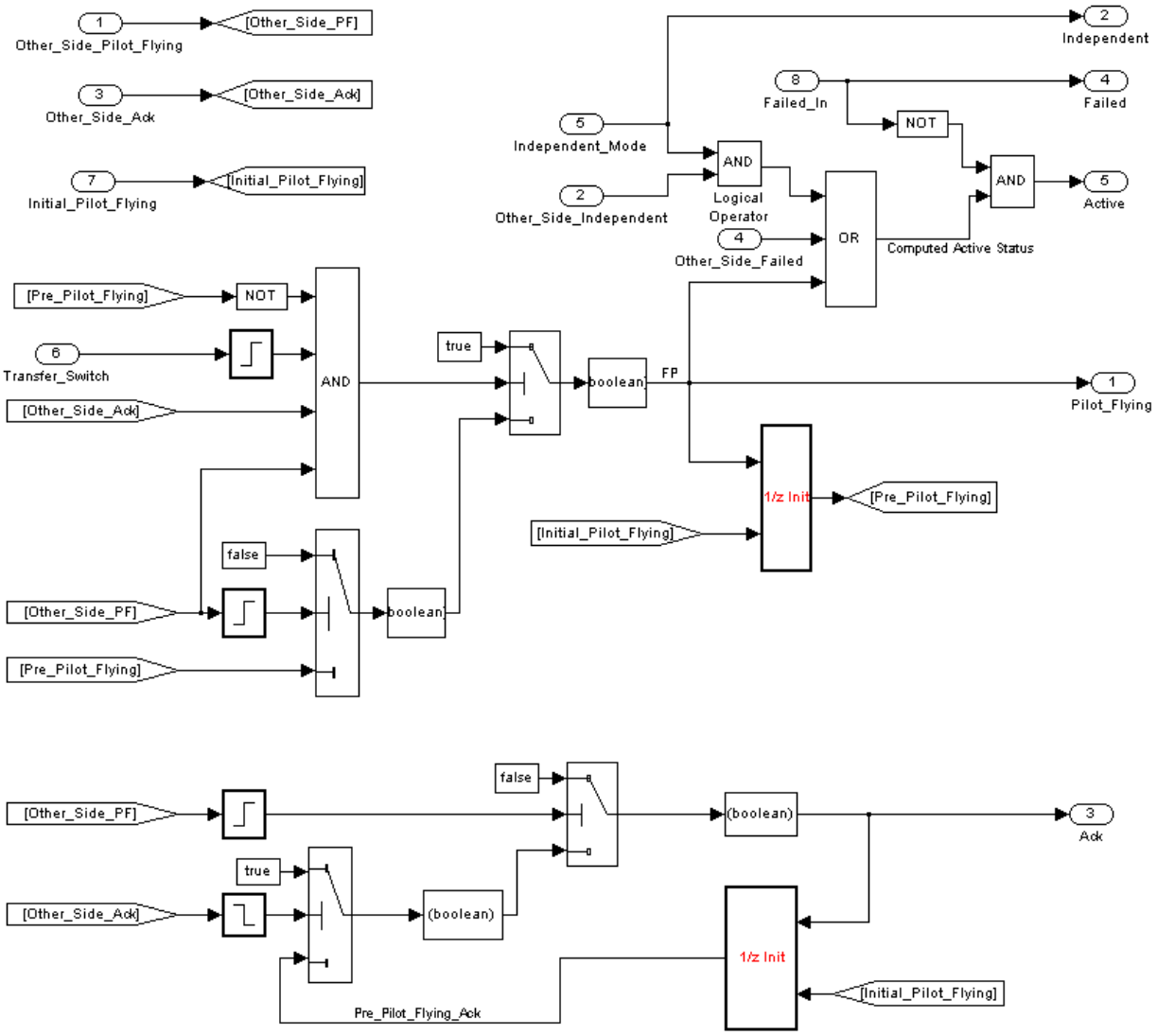


Figure 14 – Synchronization Logic of a Single FGS with Component Failures

First, if the other side is failed, this side will attempt to promote itself to active status as shown by the specification of the *Computed Active Status* line in the upper right hand corner of Figure 14. Second, if this side is failed, it will downgrade itself to inactive status, regardless of what the value of *Computed Active Status* is. Finally, we must also output our *Failed* status to the other side so that it can respond correctly when we fail. With these changes, we are ready to verify our system safety properties.

Property 1 – At least one FGS shall always be active.

In proving this property, we immediately realize we have to rule out the situation in which both sides are failed, since they will both clearly be inactive. Fortunately, a reliability analysis reveals

that the probability of both sides being failed at the same time is low enough that we can weaken our safety property to

Providing both FGS are not failed, at least one FGS shall always be active

Before proving this, we first check that the original formalization of Property 1 still holds if no failures occur by defining

```
No_Failures := !Left_FGS_Failed & !Right_FGS_Failed ;
```

and redefining Property 1 as

```
Property1 := No_Failures ->
                (Left_FGS_Active | Right_FGS_Active);
```

This is easily shown to hold for all reachable states. To check the case when at most one side is failed, we define

```
At_Most_One_Failure := !Left_FGS_Failed | !Right_FGS_Failed;
```

and redefine Property 1 as

```
Property1 := At_Most_One_Failure ->
                (Left_FGS_Active | Right_FGS_Active);
```

Unfortunately, even this version of Property 1 will not hold for all states. Since the active FGS can fail at any moment and immediately become inactive, it is possible for both sides to be inactive until the other side can step in and recover. Instead, we must settle for proving the weaker property that allows both sides to be inactive for no more than one communication cycle, i.e.,

```
AG(   ((!Property1 & Left_FGS_Clock) ->
           (A [!LR_Channel_Clock U Property1 |
                (AX A [!Right_FGS_Clock U Property1 ])])) |
      ((!Property1 & Right_FGS_Clock) ->
           (A [!RL_Channel_Clock U Property1 |
                (AX A [!Left_FGS_Clock U Property1])])));
```

This is easily proven to hold for all system states. We confirm with the system safety engineers that it is acceptable for Property 1 to be false for no more than one communication cycle.

Property 2 – Exactly one side shall be the pilot flying side.

Unlike Property 1, Property 2 holds even when failures are introduced, just as it did for the asynchronous case. Just as for the synchronous and failure free cases, we formalize Property 2 as

```
Property2 := (Left_FGS_Pilot_Flying = !Right_FGS_Pilot_Flying);
```

and prove the CTL property

```
AG(   ((!Property2 & Left_FGS_Clock) ->
            (A [!LR_Channel_Clock U (Property2 |
                  (AX A [!Right_FGS_Clock U Property2]))])) |
      ((!Property2 & Right_FGS_Clock) ->
            (A [!RL_Channel_Clock U (Property2 |
                  (AX A [!Left_FGS_Clock U Property2]))])) );
```

to establish that there must always be exactly one pilot flying side except for one communication cycle of the system.

Property 3 – If the system is in independent mode, both FGS shall be active.

Property 3 poses problems similar to Property 1 in that if even one side is failed, it is not possible for both sides to be active. We proceed by first proving that Property 3 holds when there are no failures by redefining Property 3 as

```
Property3 := ((No_Failures & System_Independent_Mode) ->
                  (Left_FGS_Active & Right_FGS_Active));
```

and proving that this form of Property 3 cannot be false for more than one communication cycle. This is easily shown to be true with the model checker.

Proving a weaker form of Property 3 in which even one side is failed is not possible. On further reflection, we realize that this is actually a shortcoming of our definition of *System Independent Mode*. The intent of being in *System Independent Mode* is that both sides should be able to produce outputs that can be compared and voted. Clearly, this is not the case if one side is failed. For this reason, we should redefine *System Independent Mode* as

```
System_Independent_Mode :=
      (!Left_Failed & Left_Independent) &
      (!Right_Failed & Right_Independent) ;
```

With this definition, we can define Property3 in its original form as

```
Property3 := System_Independent_Mode ->
                  (Left_FGS_Active & Right_FGS_Active);
```

This version of Property 3 is actually equivalent to one given immediately above, and can easily be shown to be false for no more than one communication cycle.

7.3 Observations on the Issues Introduced by Component Failures

The maximum number and type of components failures that a system can tolerate are often referred to as the fault assumptions for that system. In the case of the asynchronous dual FGS system, we made the fault assumption that at most one FGS would fail at a time and were forced to modify some of our safety properties to include the maximum fault assumptions. Other fault assumptions were also made. For example, our channels can fail to deliver an unbounded number of signals, but because of the design of the system and the implicit assumption that the channel clocks would eventually become true, these did not affect the verification of our safety properties.

Our approach to modeling failures as system inputs is not completely satisfactory. Since faults originate within each component, it would be far more desirable to be able to add the specification of how each component could fail without affecting its interface. This would be a first step in automating system safety analysis.

8 A Process for Architectural Refinement

The preceding examples have illustrated a process for refining an ideal system specification to an implementation. In this section, we define this process (illustrated in Figure 15) more thoroughly. This process can be applied recursively to each component as the system is refined towards an implementation, so that a component in one step may become the system in the next step. In the following discussion, the term "system" will be used relative to the current level of abstraction and can refer to either the overall system or to a component that is being refined. We will use the term "overall system" when we want to refer to only the full system at the highest level of abstraction.

1. **Define System Interface.** The first step is to define the interface (i.e., the inputs and outputs) of the system. This allows us to specify the syntax of how the system interacts with its external environment, though it does not define the semantics (how the outputs change as the inputs are varied) of the system. If the system is a component from a previous refinement step, this activity was already completed in Step 3a of that refinement.

2. **Determine System Requirements.** This step assigns a partial semantics to the system by stating constraints on its behavior that must be met if the overall system is to function correctly. In addition, if there are any assumptions the system is allowed to make about its environment, these are captured as invariants over its inputs. If the system is a component from a previous refinement step, these requirements were stated as formal properties or as an abstracted model in Step 3c of that refinement. If we are dealing with the overall system, these requirements may be developed through informal processes such as a system hazard analysis [1], [8], [17], [23], [27]. Such safety requirements may initially be stated informally, but at some point they will need to be restated in a formal notation such as CTL, LTL [7], [10], [20], [21], or as an abstracted model, before we can begin Step 4, model analysis.

3. **Model Capture.** The next step is to refine the system into smaller components and define their interconnections. We do not prescribe a particular formal model for representing the components and their interconnections, but any notation chosen must be able to specify the components, their interconnections, and some notion of component behavior. The steps within this process are:

    a. *Define Components.* First, we define the components of the system and their interfaces, in a manner similar to that of Step 1. In most cases, a large part of the system structure will be dictated by existing legacy systems. For example, a new FGS will likely have to fit into an existing Flight Control System architecture in which the components and their interfaces have been largely fixed.

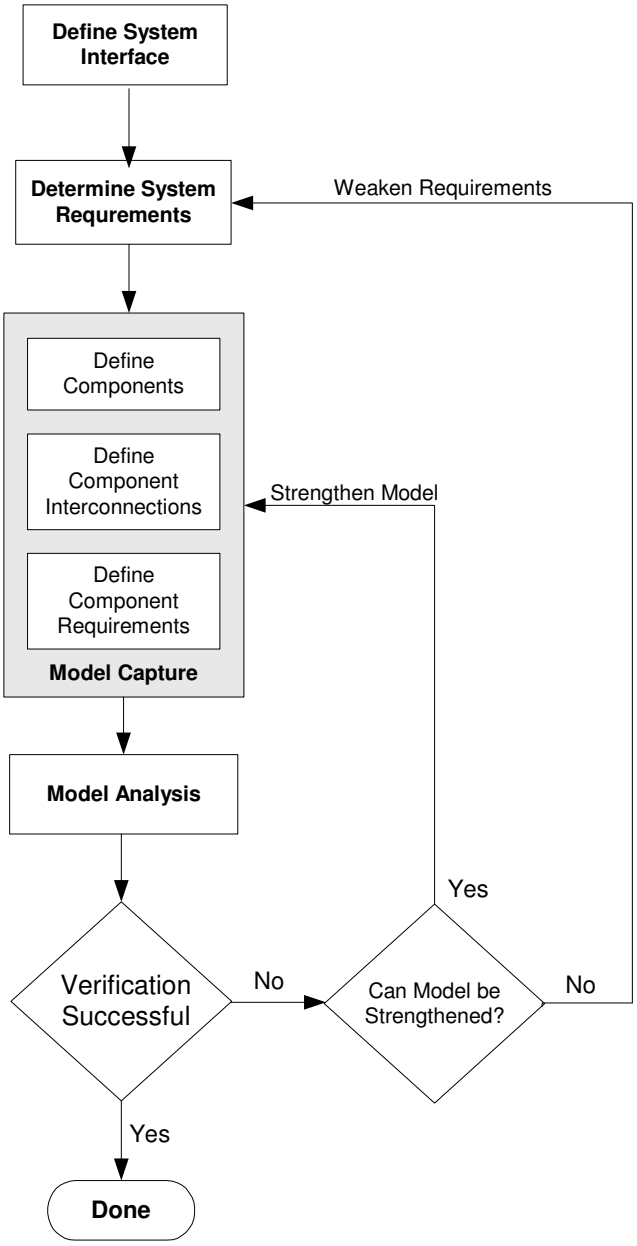


Figure 15 - Architectural Refinement Process

b. *Define Interconnections.* The next step is to define the interconnections between the components. As discussed below, we may decide to initially model these as idealized synchronous connections and analyze this implementation to determine if it can meet the system requirements. If this is successful, we can iteratively refine the implementation into a more realistic model by introducing behaviors such as asynchrony and failure modes and repeat the analysis.

c. *Define Derived Requirements.* In this step, we begin to define the behavior of each component that is needed to prove one or more of the system requirements. This can be done either by specifying properties of the component in a notation

such as CTL, LTL, or PVS, or by specifying a model that is an abstraction of the component's full behavior. Initially, we may specify very little of the behavior of a component. As our analysis progresses, we will evolve the specification of the components to be as accurate as needed to demonstrate that the components working together satisfy the system requirements. This partial specification for a component then becomes the system requirements for the next round of refinement.

4. **Model Analysis.** Once we have a formalized set of properties from Step 2 and formal model of our implementation from Step 3, we proceed to verification in step 4. As shown in the earlier sections, we use formal analysis tools such as model checkers or theorem provers to ensure that the system requirements are satisfied by the implementation.

   We may first want to verify that an ideal implementation (i.e., one that does not consider issues such as component failures or asynchronous communication) can satisfy the system requirements. Even if the ideal implementation can satisfy the system requirements, we will probably still need to refine it to deal with issues such as component failures and asynchrony. As this is done, it is likely that at some point, the system requirements will no longer be satisfied. In these cases, we either need to strengthen the implementation so that it can meet the system requirements (as was done in Section 6.2.2 when the "acknowledgement" signal was introduced) or weaken the system requirements to a set that can be satisfied (as was done several times in Sections 6 and 7).

   Of course, the weakening of requirements may propagate upward through several levels of refinement, a costly process. Anticipating and designing systems that are robust in the face of such changes is clearly important. For example, in critical systems, component failure usually has to be dealt with at some level, and a bit of anticipation could easily reduce the amount of rework necessary to implement a system. In some systems, the failure may be propagated all the way up to the operator (e.g., the pilot) who is expected to take appropriate action. In other systems, redundancy may have to be introduced to bring the probability of system failure to an acceptable level. This may be easier to do at one level rather than another, or across one set of interfaces as opposed to another. In a similar manner, if asynchrony has to be introduced somewhere in the system architecture, there may be places where it can be introduced more easily. Unfortunately, a full treatment of this topic is beyond the scope of this paper and is an area for further work.

   In any case, once the system requirements are no longer met, an iteration in the system design will be necessary. This can either be done within the current refinement by strengthening the system implementation or by weakening the system requirements as shown in Figure 15.

9 Conclusions and Future Directions

We have described an approach for the development of a system architecture that allows the designer to start with an ideal system specification and refine it to an implementation in a series of steps along two axes. Fidelity extension is used to weaken the specified system behavior to a still acceptable behavior when it becomes clear that the ideal system behavior is either too costly or impossible to implement. Design refinement is used to decompose an individual system component into simpler components that implement it. We illustrated this approach on the synchronization logic of a dual redundant Flight Guidance System and showed how the system specification (i.e., the safety properties) needed to be relaxed as asynchronous behavior and component failures were introduced.

The fidelity extension from synchronous to asynchronous behavior was particularly difficult. The introduction of asynchrony made the formalization of the safety properties more complex, made their verification harder, and required that the component models be strengthened in order to satisfy even these weaker safety properties. This argues strongly for the use of synchronous models whenever possible. However, as systems become very large, it seems inevitable that asynchronous behavior will need to be introduced at some point. Systems consisting of pockets of synchronous components connected asynchronously are often referred to as Globally Asynchronous/Locally Synchronous (GALS) systems. The fidelity extension from synchronous to asynchronous behavior described in Section 6 illustrates one approach to the modeling and verifying the properties of GALS architectures by assigning each component its own clock. However, the difficulty of verifying even a few properties of a simple system suggests that this approach may not scale well.

One possible solution might be to find better ways to model asynchrony. Much of the complexity of Section 6 was due to the use of unbounded asynchrony introduced by leaving the component clocks completely unconstrained. Most real systems are implemented with tighter bounds on the asynchrony. Finding a way to model asynchrony that exploits these bounds could help to keep the problem tractable.

Another solution may be to find better ways to extract the essence of the problem from the detailed component models. This is actually what we did manually in Section 6. A full flight guidance system is a very complicated piece of logic, but only a small portion of that logic is devoted to determining which side is the active side. We manually abstracted this portion and verified its correctness, discovering a very subtle error in the process. If we could find ways to automatically extract the reduced component models relevant to a particular system property, it would help to keep the verification manageable.

This suggests that we might want to reconsider the allocation of logic between the system architecture and the system components. Ideally, iterations along the fidelity extension axis would be largely independent of iterations along the design refinement axis, since design refinement typically refines a single component into a more detailed implementation while fidelity extension normally involves changing the way in which components interact. The extent

to which this is not true could be taken as a measure of the improper allocation of logic between the system architecture and individual components.

For example, we associated the logic for the synchronization of active status with the individual FGS components. As we changed how the components connected to each other and how they could fail, we also had to change the behavior of the individual FGS components. This suggests that this behavior should have been a part of the system architecture rather than of the individual components. While this provides a cleaner boundary between the components and the system architecture, it would greatly complicate the system architecture as it would now have to include all of the logic for handling component failures and different connection semantics.

An obvious way of managing such architectural complexity would be the development of design patterns for common architectural features. Examples of such architectural patterns are already well known in the fault-tolerant computing literature. For example, in Section 6.2.2, we described the use of the acknowledgment signal to inhibit further changes in the system state until a stable system configuration was restored. This is probably a specialization of an existing pattern for maintaining consistent system state.

The fidelity extension for component failures discussed in Section 7 illustrates the need for better tools for specifying component fault models, i.e., the specification of how they can fail. In Section 7, we introduced faults in the FGS components by adding a simple *Failed* input. This unfortunately required us to change the interface for the FGS model. Ideally, one would like to be able to add different fault models to a component *without changing the underlying specification of normal behavior* and explore how this would affect the safety properties of the overall system. This would be a first step towards integrating system safety analysis and system engineering about a common model of the system architecture.

10 Bibliography

[1] ARP 4761, Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment, Warrendale, PA: Society of Automotive Engineers, 1996.

[2] G. Berry and G. Gonthier, The Synchronous Programming Language Esterel: Design, Semantics, and Implementation, *Science of Computer Programming*, Volume 19, pages 87-152, 1992.

[3] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R.de Simone, The Synchronous Languages 12 Years Later, *Proceedings of the IEEE*, Volume 91, Issue 1, January 2003.

[4] Marco Bozzano, Antonella Cavallo, Massimo Cifaldi, Laura Valacca, and Adolfo Villafiorita, Improving Safety Assessment of Complex Systems : An Industrial Case Study. *Proceedings of Formal Methods 2003 (LNCS 2805),* Springer-Verlag, pages 208-222, 2003.

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

[6] R. Butler, S. Miller, J. Potts, and V. Carreno, A Formal Methods Approach to the Analysis of Mode Confusion, *Proceedings of the 17th AIAA/IEEE Digital Avionics Systems Conference*, Bellevue, WA, October 1998.

[7] E. Clarke, O. Grumberg, and P. Peled, *Model Checking*, The MIT Press, Cambridge, Massachusetts, 2001.

[8] CISHEC. *A Guide to Hazard and Operability Studies*. The Chemical Industry Safety and Health Council of the Chemical Industries Association Ltd., 1977.

[9] James Dabney and Thomas Harmon, *Mastering Simulink*, Pearson Prentice Hall: Upper Saddle River, NJ, 2004.

[10] E. A. Emerson, Temporal and Modal Logic, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics,* J. van Leeuwen, ed., North-Holland Pub. Co./MIT Press, Pages 995-1072, 1990.

[11] Esterel Technologies, SCADE Suite Product Description, http://www.esterel-technologies.com/v2/scadeSuiteForSafetyCriticalSoftwareDevelopment/index.html.

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

[13] N. Halbwachs and S. Baghdadi, Synchronous Modeling of Asynchronous Systems, *Proceedings of EMSOFT'02,* LNCS 2491, Springer-Verlag, Grenoble, October 2002.

[14] C. Heitmeyer, R. Jeffords., and B. Labaw, Automated Consistency Checking of Requirements Specification, *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(3):231-261, July 1996.

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

[16] A. Joshi, S. P. Miller, and M. P. E. Heimdahl, Mode Confusion Analysis of a Flight Guidance System Using Formal Methods, *Proceedings of the 22nd Digital Avionics Systems Conference (DASC'03)*, Indianapolis, Indiana, Oct. 12-16, 2003.

[17] N. G. Leveson, *Safeware: System Safety and Computers*, (Reading, MA: Addison-Wesley Publishing Co., 1995).

[18] S. P. Miller, A. C. Tribble, T. M. Carlson and E. J. Danielson, *Flight Guidance System Requirements Specification*, NASA/CR-2003-212426, June 2003.

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

[20] S. Owre, J. Rushby, N. Shankar, F. Henke, Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS, *IEEE Transactions on Software Engineering*, Volume 21, Number 2, pages. 107-125, February 1995.

[21] R. Richards, D. Greve, and M. Wilding, The Common Criteria, Formal Methods and ACL2, *Proceedings of the Fifth International Workshop on the ACL2 Prover and Its Applications (ACL2-2004)*, Austin, TX, November 2004.

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

[23] *System Safety Analysis Handbook*, 2nd Ed., System Safety Society, July 1997.

[24] J. Thompson, M. Heimdahl, and S. Miller.: Specification Based Prototyping for Embedded Systems, *Proceedings of the Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering, LNCS 1687*, September 1999.

[25] Alan C. Tribble, David D. Lempia, and Steven P. Miller, Software Safety Analysis of a Flight Guidance System, *Proceedings of the 21st Digital Avionics Systems Conference (DASC'02)*, Irvine, California, Oct. 27-31, 2002.

[26] Alan C. Tribble, Steven P. Miller and David L. Lempia, *Software Safety Analysis of a Flight Guidance System* , NASA/CR-2004-213004, March 2004.

[27] W. Vesely, F. Goldberg, N. Roberts, and D. Haasl, *Fault Tree Handbook*, Washington, D. C., 1981.