Evaluation of the Benefits and Limitations of Verification Activities in Developing a Critical System using Model-Based Development

A THESIS

SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL

OF THE UNIVERSITY OF MINNESOTA

BY

Young Sub Lee

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

Dr. David Lilja, Advisor and Dr. Mats Heimdahl, Co-Advisor

August, 2018

Acknowledgements

First of all, I would like to thank my family, especially my parents, who has been a constant source of support and love.

Also, I would like all the members of the Crisys research team. Especially like to thank Sanjai and TaeJoon. They were always willing to help me when I needed help. Without their helping, it would have been much more difficult to complete my Master's Degree.

Finally, I would many thanks to Prof. Mats Heimdahl, my advisor. He has introduced me to new ways of thinking about research approach, software, and hardware testing. His support and feedback in this research were invaluable.

Abstract

In developing safety-critical cyber-physical systems, model-based development (MBD) promotes design and verification activities at the model-level, which is an abstract description of the behavior of the software to be implemented. MBD tool suites such as MATLAB's Simulink and Stateflow implements the principles of MBD so that system developers can easily incorporate various verification and validation activities such as modeling, simulation, testing, and even formal verification. Among all activities involved in MBD, simulation and model-level testing are widely adopted to identify bugs in the early phase of the model development. Formal verification of the design model, on the other hand, is often considered difficult or impractical so it is less widely adopted in the practice.

This report investigates the practicality, limitations, and benefits of adopting various model-based V&V activities in developing a safety-critical cyber-physical system. We have developed a rocket-launch control system in cooperation with a local rocketry club following MBD principles. More specifically, we developed a Simulink model, performed model-level simulation, testing, and model-level property verification of three safety properties. In each phase of the development, we demonstrate that each activity can be easily incorporated into the development process. The verification result of the Simulink Design Verifier shows that our safety properties of concern hold, with which we could gain confidence in the correctness of the model. This report will help people who design a critical system using MBD to chose the verification techniques that they need.

Contents

List of Figures

Chapter 1

Introduction

In developing safety-critical cyber-physical systems, model-based development (MBD) promotes design and verification activities at the model-level, an abstract description of the behavior of the software to be implemented. MBD tool suites such as MATLAB's Simulink and Stateflow implements the principles of MBD so that system developers can easily incorporate various verification and validation activities such as modeling, simulation, testing, and even formal verification. Among the activities involved in MBD, simulation and model-level testing are widely adopted to identify bugs in the early phase of model development. Formal verification of the design model, on the other hand, is often considered difficult or impractical so it is less widely adopted in practice.

This report evaluates the practicality, limitations, and benefits of adopting various V&V activities in developing a safety-critical cyber-physical system. We have developed a rocket-launch control system in cooperation with a local rocketry club [1] following MBD principles. More specifically, we developed a Simulink/Stateflow model, performed model-level simulation, testing, and model-level property verification of three safety properties. In each phase of the development, we demonstrate that each activity can be easily incorporated into the development process. The verification result of the Simulink Design Verifier shows that our safety properties of concern hold, with which we could gain confidence in the correctness of the program. This report will help people who design a critical system using MBD to select the verification techniques which they need.

In the remainder of the introduction, we will provide an overview of model-based development, describe our case examples, and the discuss the overall organization of this report.

1.1 Model-Based Development

Model-Based Development (MBD) is a method in the software and systems development fields. MBD focuses on high-level executable models of the system to be fielded. MBD is able to support design of a system from concepts to C, C++, or

HDL code implementing the actual software/hardware system. To be specific, a user can design the concepts in a graphical programming environment as models and the user can generate C, C++, or HDL code that has the same behaviors as the models. One more advantage to use MBD is that the models support analysis, including simulation, hardware-in-the-loop testing, and formal model verification. These activities can help find faults at the early in development cycles so that we can reduce development costs.

MATLAB/Simulink [2], a popular tool suite for MBD that provides a graphical user interface for modeling using block diagrams and customized block libraries. The environment also supports add-on products such as Stateflow, which is a design environment for developing and simulating state machines. Also, MATLAB supports Embedded Coder to generate C/C++ code from the Simulink and Stateflow models. In this report, we used MATLAB/Simulink and Stateflow to design a critical rocket launch system.

1.2 Software Verification and Validation

Verification and Validation (V&V) are the activities of analyzing that a system meets specifications and requirements. Both activities look similar but they are not the same thing. Barry W. Boehm expressed the difference between V&V [3].

- **Verification**: Are we building the product right?
- **Validation**: Are we building the right product?

The verification activities help us in evaluating a software system. To be specific, we can say that verification are the activities that are checking the correctness of a development phase. The validation activities help us in confirming that a software system meets its designated use (does what it needs to do).

In this report we show software V&V methods that are a part of the MBD process. We performed three activities for V&V, two activities (Software Simulation and Hardware in the Loop Simulation) in the course of the development process and another (Formal Model Verification) at the end of the development to ensure the safety requirement were indeed met.

1.2.1 Software Simulation

Software Simulation is based on the process of modeling. After developing a software model, you can simulate the model under a set of conditions to observe the model behavior in a virtual environment. Simulation has benefits when developing a critical system since it is expensive and hard to test the software in a real system.

1.2.2 Hardware in the Loop Simulation

Hardware in the loop (HIL) simulation is one of the techniques used to verify a real-time embedded system by simulating a virtual (software) system which connects with physical (hardware) system by a data acquisition (DAQ) device. The physical system includes a plant, sensors, and actuators. The DAQ is a sensor that processes electrical signals and also a generator that generate electric signals as digital or analog. In HIL simulation, we used the DAQ device (National Instruments USB-6001) to connect the software system and hardware system [4].

1.2.3 Formal Model Verification

Formal model verification is the act of proving or disproving the correctness of model with respect to a formal specification or property. This method depends on mathematical procedures to check all possible execution paths of the model to find hidden design errors and to prove the design meets the stated requirements.

1.3 The Rocket Launch System

Tripoli Minnesota is a club that flies rockets as a hobby [1]. They develop their rocket individually but share the rocket launch system that they call mLaunch 3. At their launch facility, they launch all sizes of rockets from small to huge models.

The high power rockets can be hazardous even if the rocket launch system causes a small malfunction. Hence, the system must have rigorous V&V activities before the system is released.

Using MBD, we developed a rocket launch system with the mLaunch 3 requirements and specifications. The basic function of this system is launching rockets in a safe manner as well as having fun. mLaunch 3 has four pad units (the unit that sits on a rocket launch pad) and each unit can launch up to two rockets. A launch controller can command to an individual pad unit to launch the rockets. In order to launch a rocket, the rocket owner places the rocket on the pad unit to connect the pad-unit's ignition to the ignitor in the rocket. The owner can press a button on the pad unit to send an activation signal to the controller. When the controller gets the pad activated signal, the Launch Control Officer (LCO) can press a armed button on the controller sending a pad armed signal to pad unit to close the first of two relays. After the pad unit is ready to launch and the owner is back to a safe place, the LCO presses a rocket launch button on the controller. This button causes a second relay to close providing current to the ignition system in the rocket causing the rocket to launch.

1.4 Scope of Study

In this report we investigate the practicality, limitations, and benefits of three V&V activities when we developed the rocket launch system. When starting this investigation, I had no background in MBD and formal verification. In the process of conducting the work, I explored modeling using Simulink and State-flow, hardware in the loop-simulation, model testing, and formal verification. For some of these activities, this investigation was my first exposure and there was a significant—but rewarding—learning curve. To allow for a successful development and learning environment, we planned on staging the development from technology familiarization with simple models to a more complete modeling of the software system. We did the same with the system hardware where we first built simple buttons and programmed the embedded platform, moved on to use the DAQ card to connect models and hardware, and finally conducted hardware in the loop simulation. The organization of the report reflects this staged development and is structured as follows:

- **The next chapter** discusses the design and simulation processes of the rocket launch system in chronological order, from a simple model to a more complete model including hardware in the loop simulation. We discuss the lessons learned as we describe our modeling efforts.

- **Chapter 3** discusses the design of the formal verification models for our safety properties, the results of the verification, and the lessons learned from the effort.
- **Chapter 4** concludes and discusses merits and limitations of the V&V techniques in MBD.

Chapter 2

Modeling and Simulation

In this chapter, we will discuss how we modeled and simulated the rocket launch system using Simulink and Stateflow. We developed a small part of the system as a start to learn how to use the tools. We also developed initial hardware needed for the system, for example, the launch buttons needed to control the launch. We then gradually added functionality and structure to our models to create a whole system.

In short, we first developed a model of the launch logic to learn and determine if this approach seemed suitable for this modeling project. After establishing feasibility, we started adding functionality, developing and connecting the system hardware (and the models needed to communicate with the hardware), and separating the models to reflect the distributed nature of mLaunch 3 (a central controller and launch units placed at each launch pad).

Each model used simulation to anticipate and evaluate behaviors. We designed the system by iterating between modeling and simulation in the early developed models to improve the quality of the system and to reduce the number of errors found later in the design process.

It is worth noting that the investigation also included exploring programming Intel x86 embedded controller and connecting it to hardware models. We conducted this study to understand how a powerful embedded computer was programmed and how to build the hardware that would communicate with it (the buttons and relay controllers). This part of the project was surprisingly time-consuming since we had no experience with this controller and little experience with building the hardware to communicate with it. Although this part of the project did not directly contribute to our modeling efforts and will not be further discussed in this report, the lessons learned were valuable.

In the following four subsection we will describe how we developed a simplified launch system and connected it to the launch buttons as a demonstration and learning experience. We will then discuss the development of the unit that goes on the launch pad and controls the relays providing current to the rocket engine. We then discuss how the simplified launch system is extended to serve as the launch controller and we conclude the section with a discussion on how the pad unit and launch controller are brought together to a complete launch system.

2.1 Simple Launch System

Before developing the launch controller and the pad units, we first design a simple launch system that can fly a rocket using three buttons. Each of the buttons works as same as the buttons of mLaunch 3. The desired behavior of the Simple Launch System is if buttons are pushed sequentially, first B1 (pad ready), then B2 (arm launch pad), and finally B3 (launch), then a rocket will launch. As we will see, even such simple functionality led an inexperienced developer to make mistakes and finding such mistakes in modeling and simulation was helpful.

2.1.1 Designing LaunchLogic

To design the initial system logic, LaunchLogic, we used Stateflow, which is a toolbox of Simulink, and is based on state machines. The LaunchLogic has three inputs, the buttons B1 (pad ready), B2 (arm launch pad), and B3 (launch) and two outputs, relay1 and relay2 closing the circuit that ignites the rocket engine. Figure 2.1 shows there are four states connected with transition conditions. Each state has two output status so that the outputs can be changed when the current state transits to another state. In the Initial State, if buttons are pressed sequentially

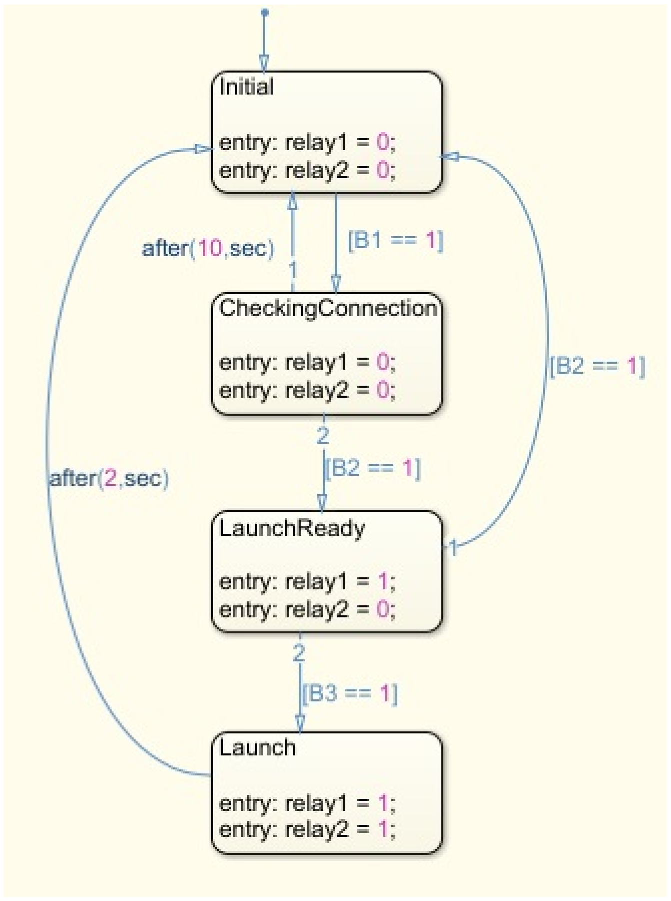


Figure 2.1: LaunchLogic Stateflow Chart

B1, B2, and B3 then the current state machine changes to the Launch State, which means both relays are closed and a rocket will fire. After two seconds in the Launch State, the state machine goes back to the Initial State and both relays opened. To avoid dangerous situations, we built two safety conditions. The first condition is that after 10 seconds in the CheckingConnection State, if there is no further activity the state machine times out and goes back to the initial state.

The second condition is to allow a reset if the user presses B2 in the LaunchReady State (the user arms the launcher with a B2 button push and resets the system with a subsequent B2 push).

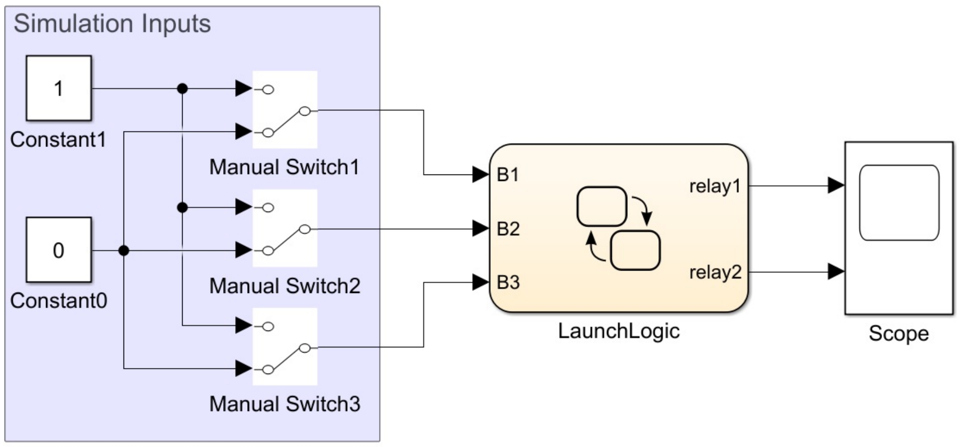


Figure 2.2: Simulation Model for the LaunchLogic

2.1.2 Simulation for LaunchLogic

To check the behavior of the LaunchLogic, we simulated the Stateflow model. Figure 2.2 shows a simulation model that included three Manual Switch blocks to generate input signals and a Scope block to see the result of the simulation outputs. We confirmed the model operated as desired behavior when we pushed the button individually.

However, through this simple testing we also found that the system had critical faults. The faults, we called them crushed button errors, occurred when the model got input signals simultaneously generated by the Switch blocks (buttons). One case of undesired behavior is that the Manual Switch3 first generate value 1 then Manual Switch1 and Switch2 generate value 1 sequentially. In this case, the sequence of the button pushes was B3, B1, B2 (the wrong order) but the system model closed both relays anyway; clearly, it is not enough to check of the signals are high, the order in which they are pressed is of imperative importance. The result of our erroneous model means that a rocket might be fired even if it is not provided the correct launch sequence.

2.1.3 Designing ButtonLogic and ArbiterLogic

To solve the above button problem, we designed two models. one is an ArbiterLogic model and the other is the ButtonLogic model. Both of these models work between inputs and the LaunchLogic model, and their role is to create an environment where the LaunchLogic can handle only one input signal at the time. Figure 2.3 and Figure 2.4 show Stateflow models of the ArbiterLogic and the ButtonLogic.

The function of the ArbiterLogic is to allow the system to provide only one input to the LaunchLogic and prioritize simultaneous inputs. To be specific, if several Manual Switch blocks generate value 1 at the same time, the ArbiterLogic compares priorities of the input signals and makes just one output to activate

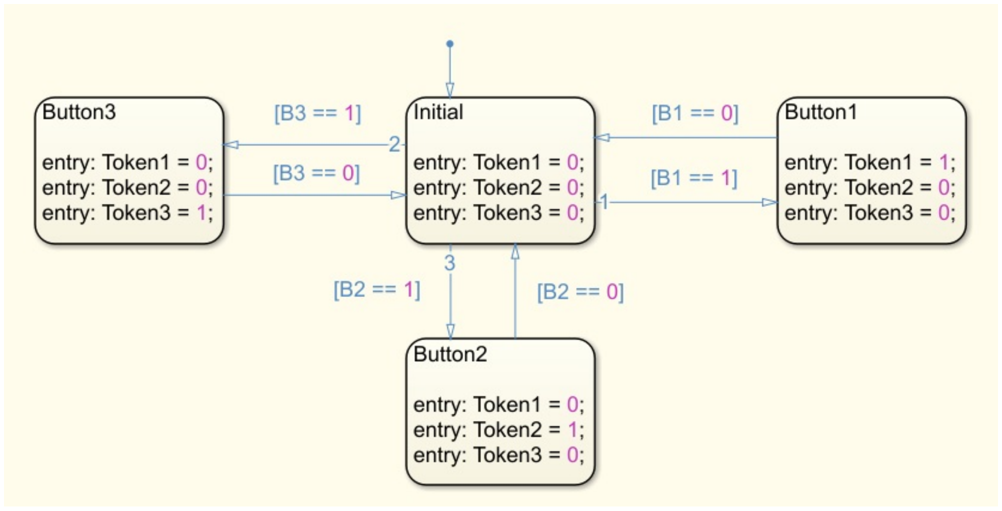


Figure 2.3: ArbiterLogic Stateflow Chart

one specific ButtonLogic. The ButtonLogic model produces output value 1 when the input values meet a condition, otherwise, it maintains output value 0. The condition is that an input signal, B, is a rising edge and another input, Token, is true. The input B connects to one of the Manual Switch models, and Token wires with the output of the ArbiterLogic

2.1.4 Simulation for Simple Launch System

Figure 2.5 shows how we composed the Simple Launch System. The system has a LaunchLogic at the main model, an ArbiterLogic model, and three ButtonLogic library models. We made the ButtonLogic model a library to allow us to reuse the ButtonLogic model several time.

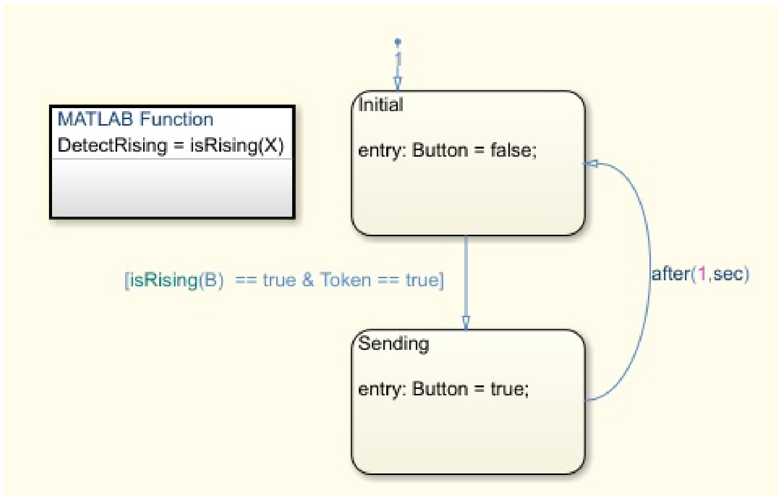


Figure 2.4: ButtonLogic Stateflow Chart

We simulated the system using the same simulation environment above in Figure 2.2. In the simulation, we put the same input sequences which caused errors when we simulated the LaunchLogic alone and now the system did not exhibit the errors at these conditions. The system only closed both relays given the expected input sequences, which is sequentially on and off for Manual Switch1, Manual Switch2, and Manual Switch3. After this simulation, we moved on to see if we could implement the buttons in hardware.

2.1.5 HIL Simulation for Simple Launch System

We used Hardware-in-the-loop (HIL) simulation to verify the Simple Launch System function in real-time simulation with hardware inputs. Figure 2.6 shows the HIL simulation model of the software. The simulation model has a Simple

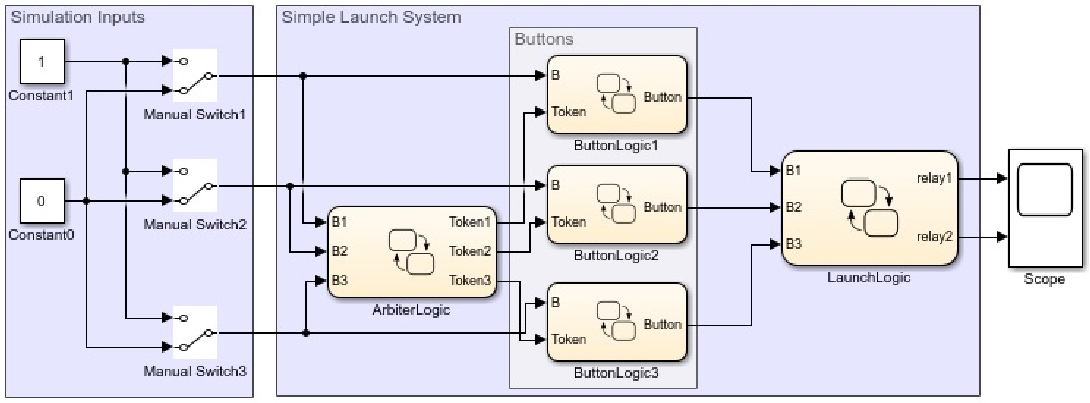


Figure 2.5: Simulation Model for Simple Launch System

Launch System model and two built-in blocks. The built-in blocks can connect the Simulink environment with a Data Acquisition (DAQ) device. The roles of the DAQ was measuring three input signals from the physical world and generating four outputs to the physical world.

Figure 2.7 shows the hardware part of the HIL simulation. At the left side of the figure, we have the DAQ (model name is USB-6001 produced by National Instrument) [4]. The breadboard on the right side of the figure has two simple circuits. One is an input circuit generating inputs using toggle buttons and also indicating with a LED each status of the pressed buttons. The second is the output circuit which makes it easy to see the output result from the DAQ. The meaning of a red LED on and green LED off is to indicate the status of the opened relay. The closed relay appeared the red LED off and green LED on.

The result of the HIL simulation was enough to show that the Simple Launch

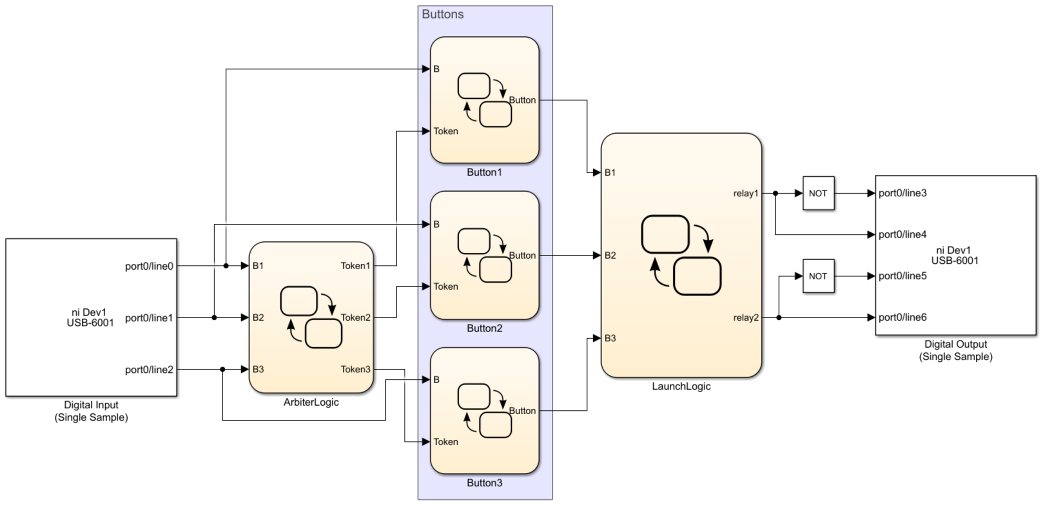


Figure 2.6: HIL Simulation Model for the Software Part of the Simple Launch System

System valid. We generated input sequences as in similar input cases with section 2.1.4 using the hardware buttons. The system was processing the input signals in real time and controlled the LEDs in the hardware circuit. We could only find one sequence of button pushes to turn on both green LEDs at the second circuit and the sequence was what we expected (B1, B2, and then B3). Thus, we considered the initial modeling a success and we had good understanding of the modeling, the hardware, and the DAQ; it was time to move on to modeling the rest of the system and bring it all together.

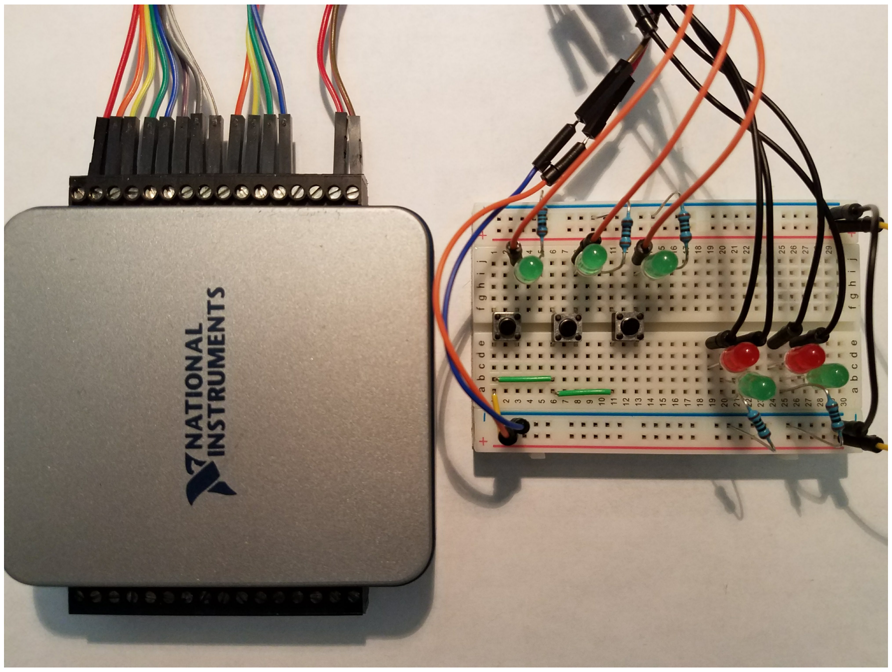


Figure 2.7: HIL Simulation Model for Software Part of Simple Launch System

2.2 Launch PadUnit

The next step was to develop a separate model of the unit that was going to sit out on the pad and actually launch the rocket, we separated the launch logic that is going in the launch controller from the logic that manages the relays sitting out on the pad in the launch pad unit. Each Launch PadUnit/PadLib is capable of managing two Padlogic models. The PadUnit can manage two rockets and has four relays (two per rocket and is managed through what we call the Padlogic), two pad activate buttons (one for each rocket; the pad activate button is pushed when the rocket has been wired to indicate that the rocket is ready for launch), and LEDs to show the status of the relays. The PadUnit is a critical model for the entire rocket launch system because users approaching the pad to attach rockets and it is the place to fire rockets, we do not want a rocket to launch inadvertently while a user is standing by the pad. If one of the PadUnit fires a rocket due to a malfunction, it could lead to personal injury. Therefore, it was necessary to be careful in model development and test simulation.

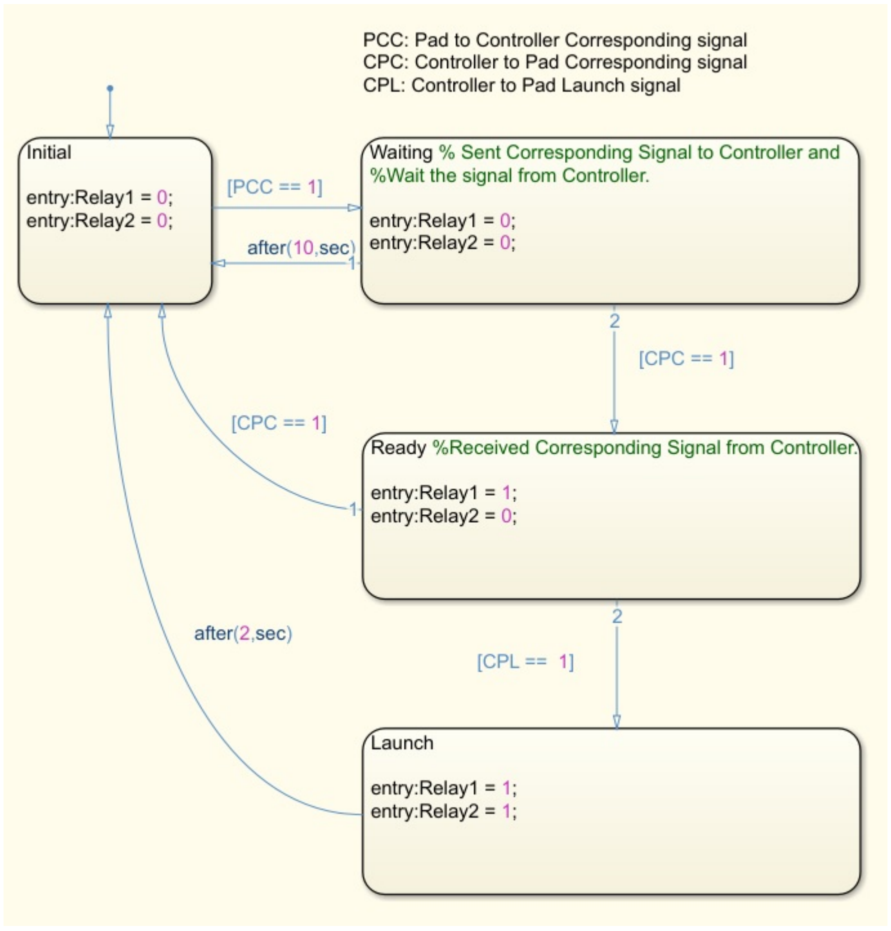

PCC: Pad to Controller Corresponding signal
CPC: Controller to Pad Corresponding signal
CPL: Controller to Pad Launch signal
Initial
entry:Relay1 = 0;
entry:Relay2 = 0;
[PCC == 1]
after(10,sec)
1
Waiting % Sent Corresponding Signal to Controller and %Wait the signal from Controller.
entry:Relay1 = 0;
entry:Relay2 = 0;
2
[CPC == 1]
[CPC == 1]
1
Ready %Received Corresponding Signal from Controller.
entry:Relay1 = 1;
entry:Relay2 = 0;
2
[CPL == 1]
after(2,sec)
Launch
entry:Relay1 = 1;
entry:Relay2 = 1;

Figure 2.8: PadLogic Stateflow Chart

2.2.1 Designing PadLogic

We designed a Stateflow model, PadLogic, which was the main operating logic as Figure 2.8. The basic operation is the same as shown in the Simple Launch-Logic explained above. The difference of them is the input method. The Simple LaunchLogic has three hardware push button to launch a rocket. But PadLogic has one hardware button (PCC, Pad to Controller Corresponding) and two digital signals (CPC, Controller to Pad Corresponding and CPL, Controller to Pad Corresponding) from the Controller that we develop after. The PadLogic takes input from one button for each rocket on the pad (indicating that the rocket has been wired with an ignition system) and two external inputs from the Launch Controller, one to arm the launch pad and one to launch.

2.2.2 Simulation for PadLogic

We configured the simulation environments as shown in Figure 2.9. It has similar input mechanisms as the ones described in subsection 2.1.2. The difference is using the visualization model, Lamp Simulink block, to indicate the status if each relay. If the relay is closed, the lamp will change to green and in the other case it is red. The simulation result was the same as what we expected. The Padlogic worked perfectly under an assumption. The assumption was that all input signals go back to the 0 value after generating a 1 value to prevent the crushed button error that we discussed above in subsection 2.1.2.

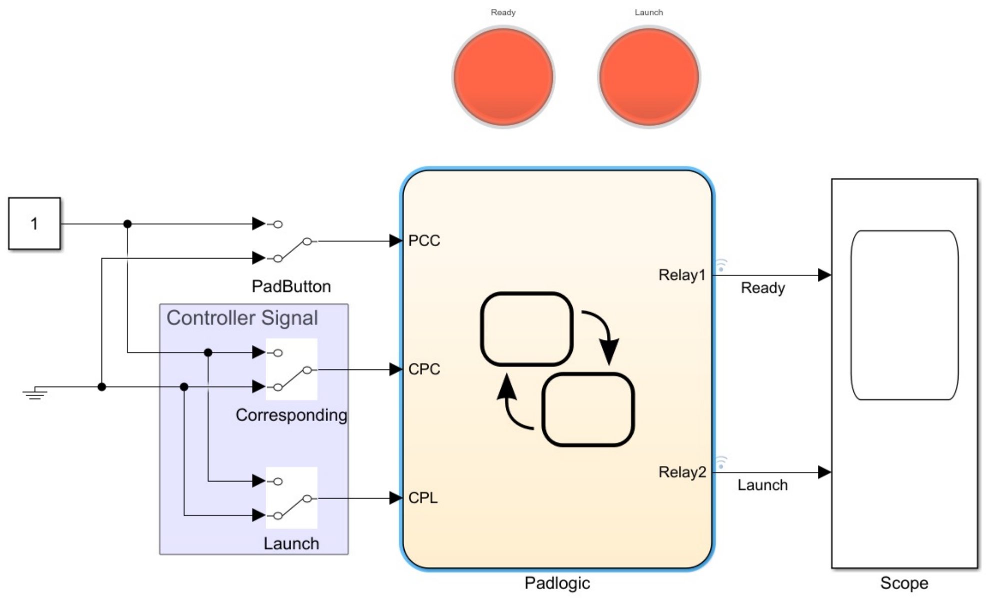


Figure 2.9: Simulation Model for PadLogic

2.2.3 Designing Atomic Padlogic

The Padlogic functionally worked well but the inside logic looked was hard to understand because each state had lots of output. So, we decided to make the main logic simply using parallel states. Also, the PadUnit contained two identical Padlogic state machines. Hence, we changed the Padlogic type to atomic subcharts that helps you reuse the same state or subchart across multiple charts and models; basically an atomic subchart is a template that can be instantiated several times.

In Figure 2.10, the Padlogic has two parallel (AND) states, Operating and Relay that can be active at the same time. From now on, where we refer to Padlogic we are referring to the logic in Figure 2.10. Parallel states are displayed as dashed rectangles. The Operating state machine shows how the Padlogic operates as inputs are received. The Relay logic controls when the relays on the bad are opened and closed.

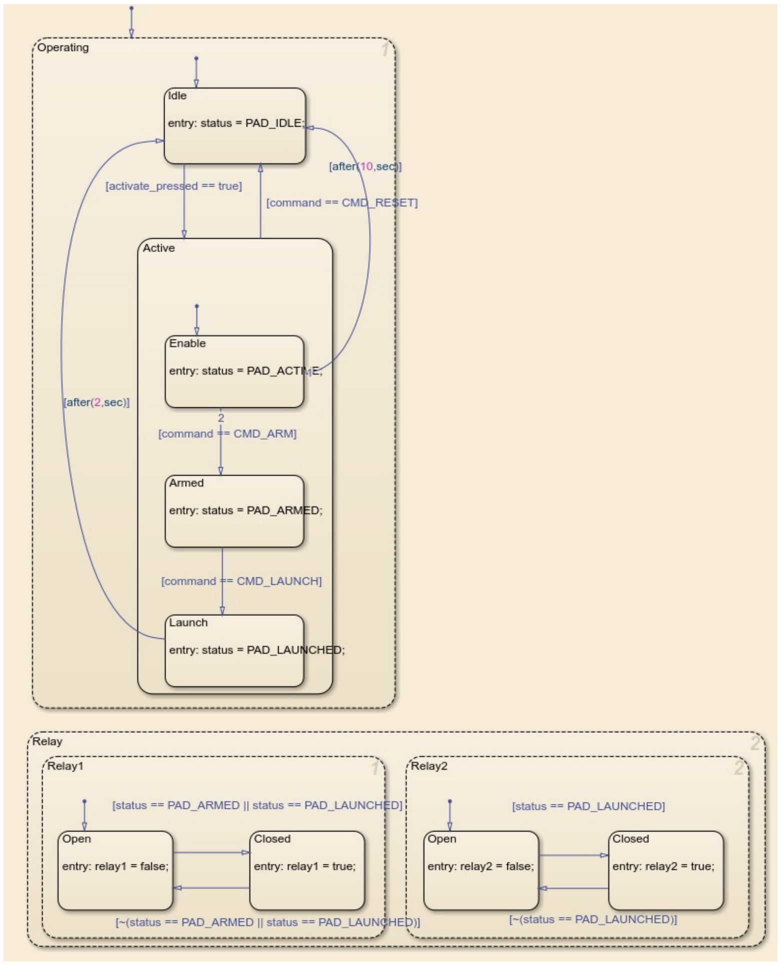


Figure 2.10: Simplified PadLogic Stateflow Chart

In the Operating state, each state is represented by one output, status, which data type is an enumerated type (enum). The status, the output of the Operating state, opens and closes the two relays; in other words an input of Relay state is the output of the Operating state. Also, the status of Operating state is sent to the Controller. In the Relay state, there are two parallel states. Each state controls a relay reacting to the output of Operating state.

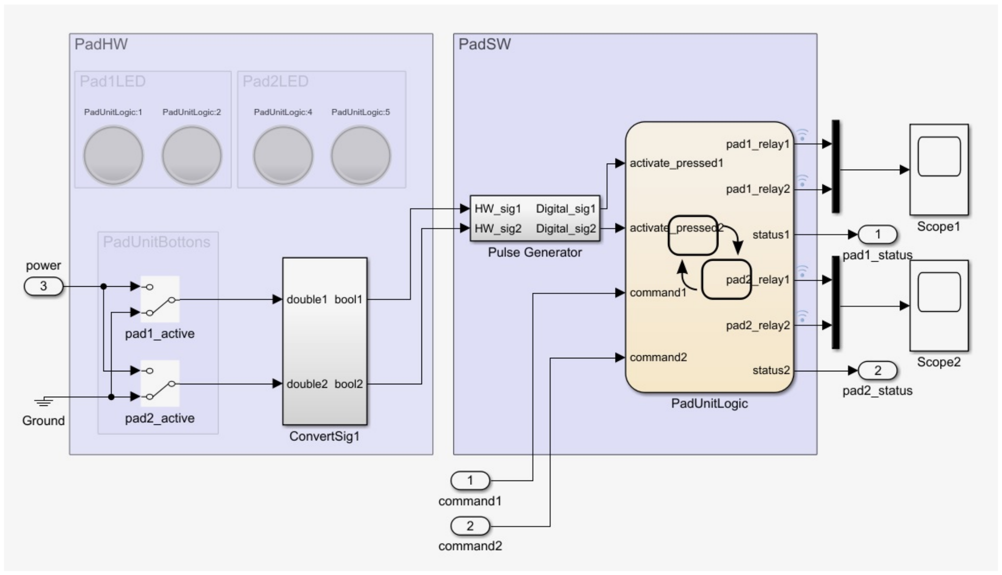


Figure 2.11: Simplified PadLogic Stateflow Chart

2.2.4 Simulation for PadLib

Using our previous models, we combine a PadLib model for one hardware pad that can control two rockets with our launch logic. Figure 2.11 shows the PadLib, which is divided into two big parts, PadSW and PadHW. The PadSW is the main parts of the PadLib for operation. Others are there to support simulation and test. PadUnit has two Atomic Padlogic. In the PadSW, a Pulse Generator has two ButtonLogic which we developed in subsection 2.1.3.

2.2.5 Designing Pad Communication Blocks

The Pad and a Controller communicate with each other to send the Pad status information from the pad to the controller and the Controller's commands to the pad. The PadLib generates an enum type output to indicate the Padlogic's Stateflow status. Since the DAQ we used in our work has binary signals, we created converters that would encode an enumerated output as a collection of binary signals.

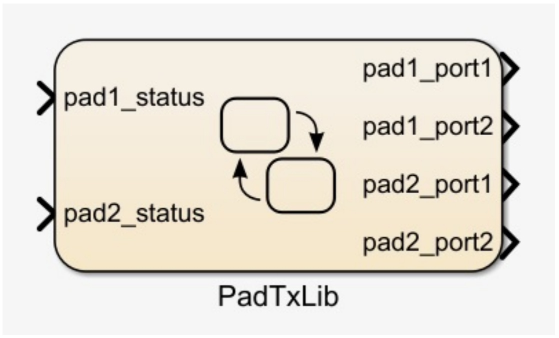


Figure 2.12: PadTxLib Stateflow Model

PadTxLib (Figure 2.12) encodes the enum type to binary information. PadRxLib (Figure 2.14) decodes binary signals to enum type information. The PadTxLib

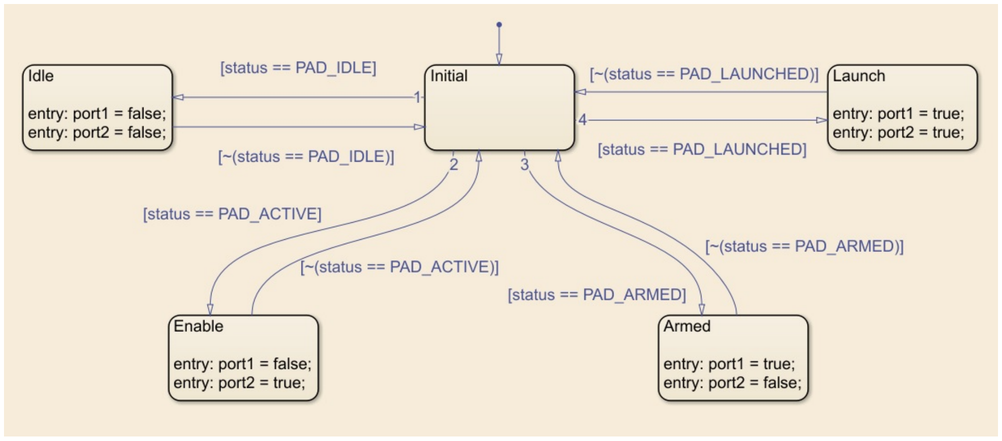


Figure 2.13: TXlogic State Machine

has two TXlogics which contains five states in Figure 2.13. The TXlogic generates two binary output according to input conditions. For example, if the input is PAD_LAUNCHED then the binary output are port1 is true and port2 is also true. You can see other cases in Figure 2.13. The PadRxLib also has two RXlogics shown in Figure 2.15. The RXlogic get the binary signals from the Controller and it decodes the inputs to enum type and sends it to PadLib.

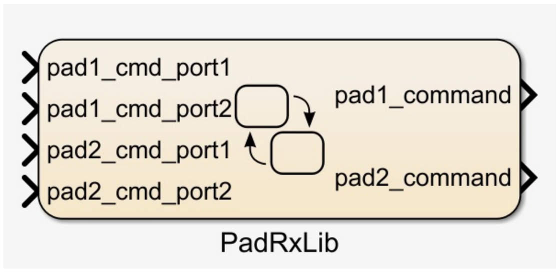


Figure 2.14: PadRxLib Stateflow Model

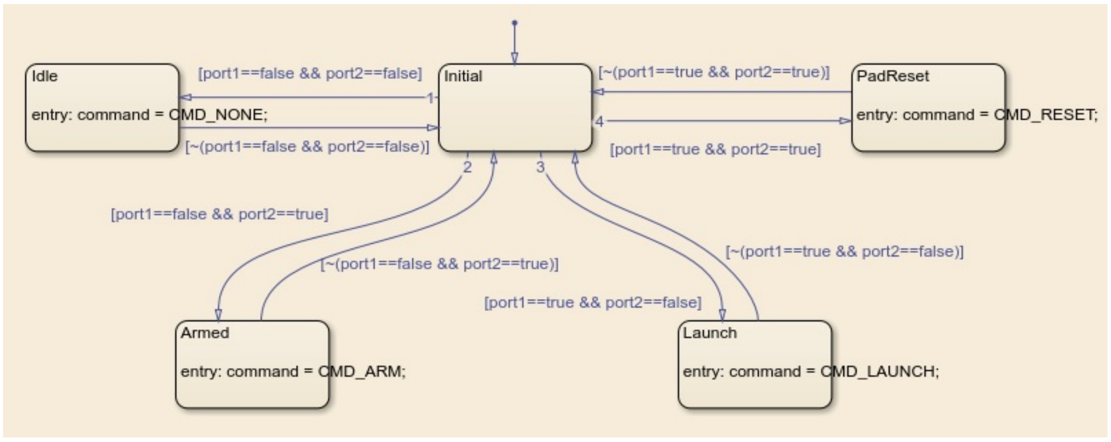

Idle
entry: command = CMD_NONE;
[port1==false && port2==false]
[~(port1==false && port2==false)]
Initial
[~(port1==true && port2==true)]
[port1==true && port2==true]
PadReset
entry: command = CMD_RESET;
[port1==false && port2==true]
[~(port1==false && port2==true)]
[~(port1==true && port2==false)]
[port1==true && port2==false]
Armed
entry: command = CMD_ARM;
Launch
entry: command = CMD_LAUNCH;

Figure 2.15: RXlogic State Machine

2.2.6 PadLib with Communication Block

The final PadLib model that we developed is shown in Figure 2.16; we combine the communication blocks with the pad logic. We put the communication part of PadLib in the PadSW section. From this point, when we refer to PadLib, we refer to the model in Figure 2.16. Since we had a simulation test for each small parts, we hoped the PadUnit model had the right behavior. To verify the behavior of PadLib, we will have simulation tests together with the Controller Model because they have to communicate with each other.

2.3 Launch Controller

The Controller can control four PadUnits/PadLibs. The Controller has eight ControllerLogic and each ControllerLogic processes four inputs to launch a rocket. The

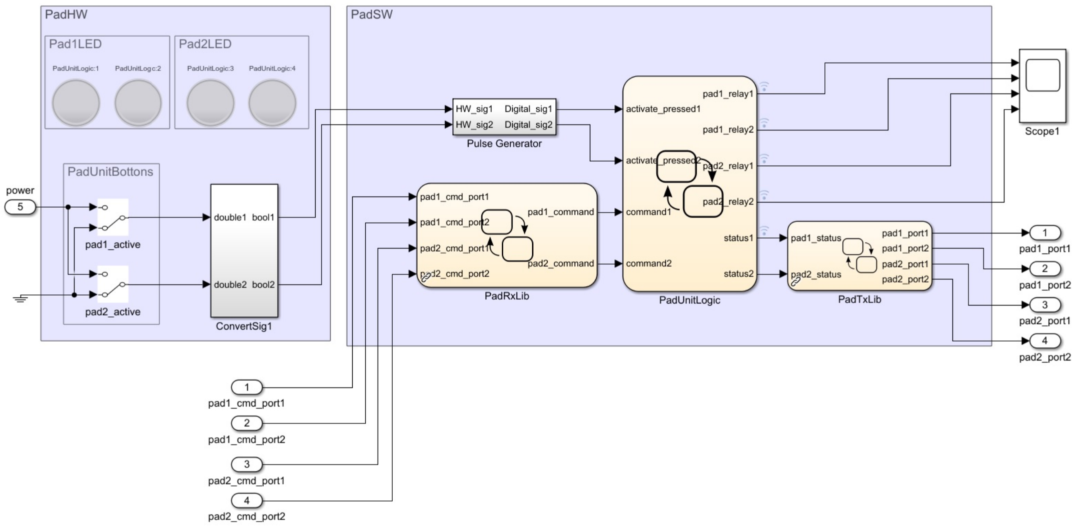


Figure 2.16: PadLib Model

Controller has 17 hardware buttons. Most of the models were already developed as libraries earlier so we focused on the design of the ControllerLogic and on the composition of the models to a full model of the launch control system.

2.3.1 Designing Atomic Controller Logic

The Controller model commands to launch eight rockets on the PadUnits. So we designed a reusable ControllerLogic as an atomic state that controls a rocket. The logic is heavily influenced by our original simple logic designed as a prototype and technology demonstration early in the project (and described first in this chapter). The role of the logic is to sends commands to the PadLib to launch the rockets. It processes three binary inputs from hardware buttons (Launch, Arm, and Reset) and one digital input from the PadLib (pad active).

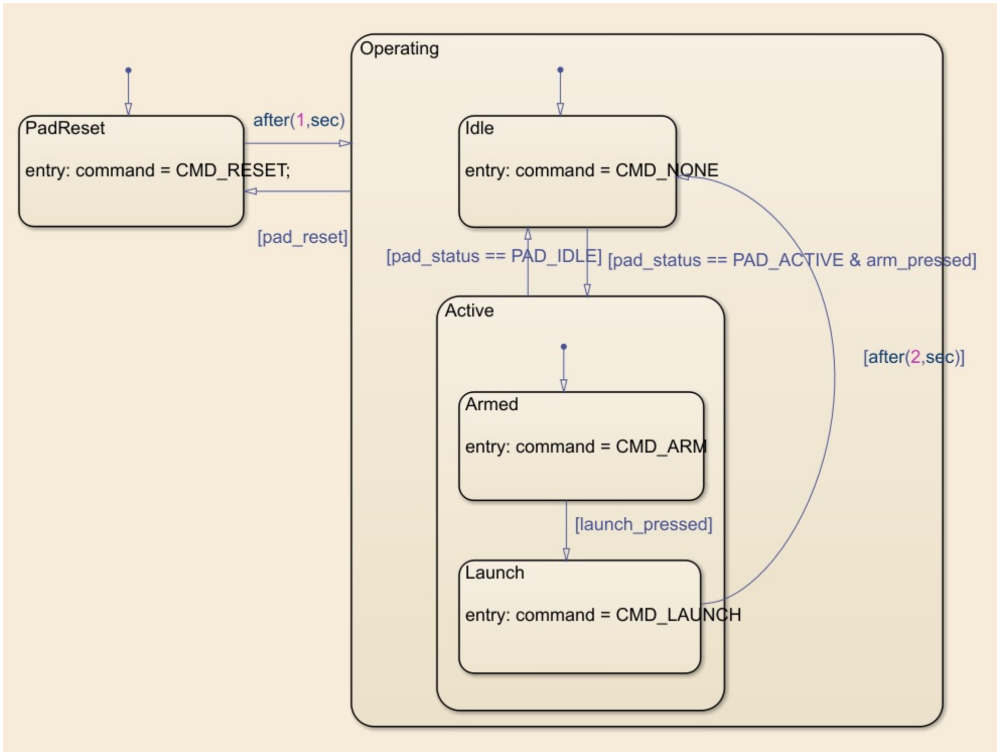


Figure 2.17: ControllerLogic State Machine

The Figure 2.17 shows the state machine of the ControllerLogic. It is divided into two parts, PadReset and Operating. Whenever the user presses a Reset button, the current state changes to PadReset and it means that the Controller-Logic sends a reset signal to the PadLib to reset the Padlogic for safety reasons. In the Operating state, the ControllerLogic generates three different commands according to the input conditions.

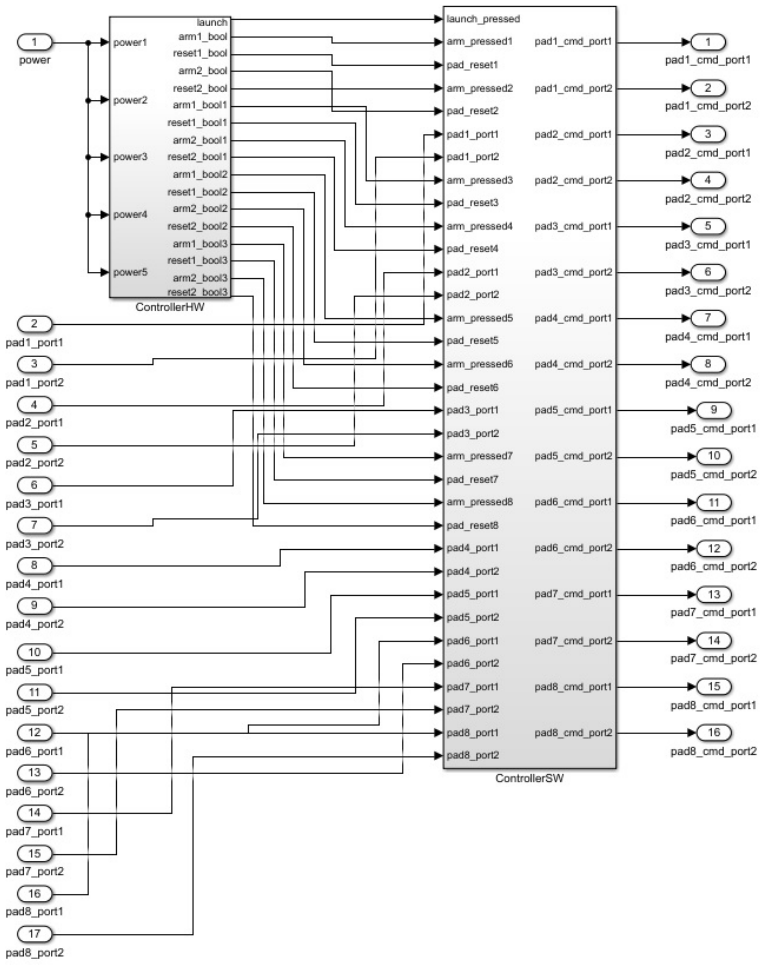


Figure 2.18: Controller Model Combined Hardware and Software Blocks

2.3.2 Building the Controller Model

Using the ControllerLogic and other Stateflow models, we built a Controller Model (Figure 2.18). The Controller also has a ControllerHW subsystem (Figure 2.19) for the simulation and a ControllerSW subsystem (Figure 2.20) that includes main operation logic. In the ControllerSW, it contains a EightControllerButtons, four ControllerRxLibs, four ControllerTxLibs and a EightControllerLogic. The EightControllerButtons has four ButtonsForPad subsystem (Figure 2.21) which includes ArbiterLogic, and two Button models. The atomic ControllerLogic handles two different hardware button signals so prevent a critical button problem above subsection 2.1.2, we decided to use ArbiterLogic. The Figure 2.22 shows the EightControllerLogic which is the main logic of the Controller. The eight atomic ControllerLogics process the input signals as simultaneously.

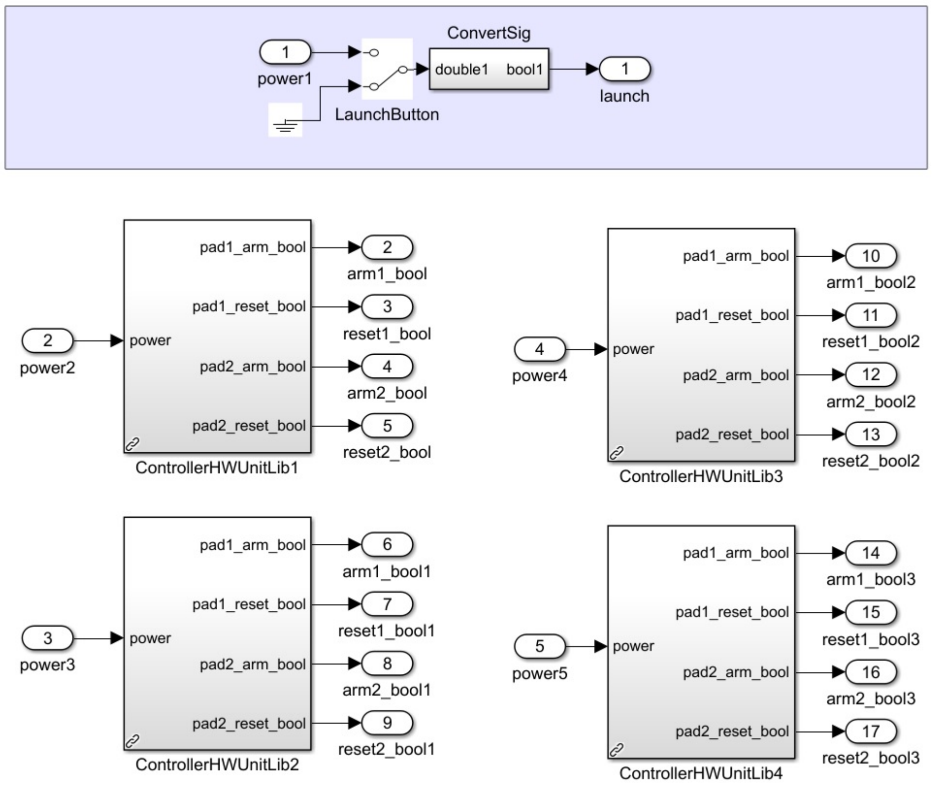


Figure 2.19: ControllerHW Model

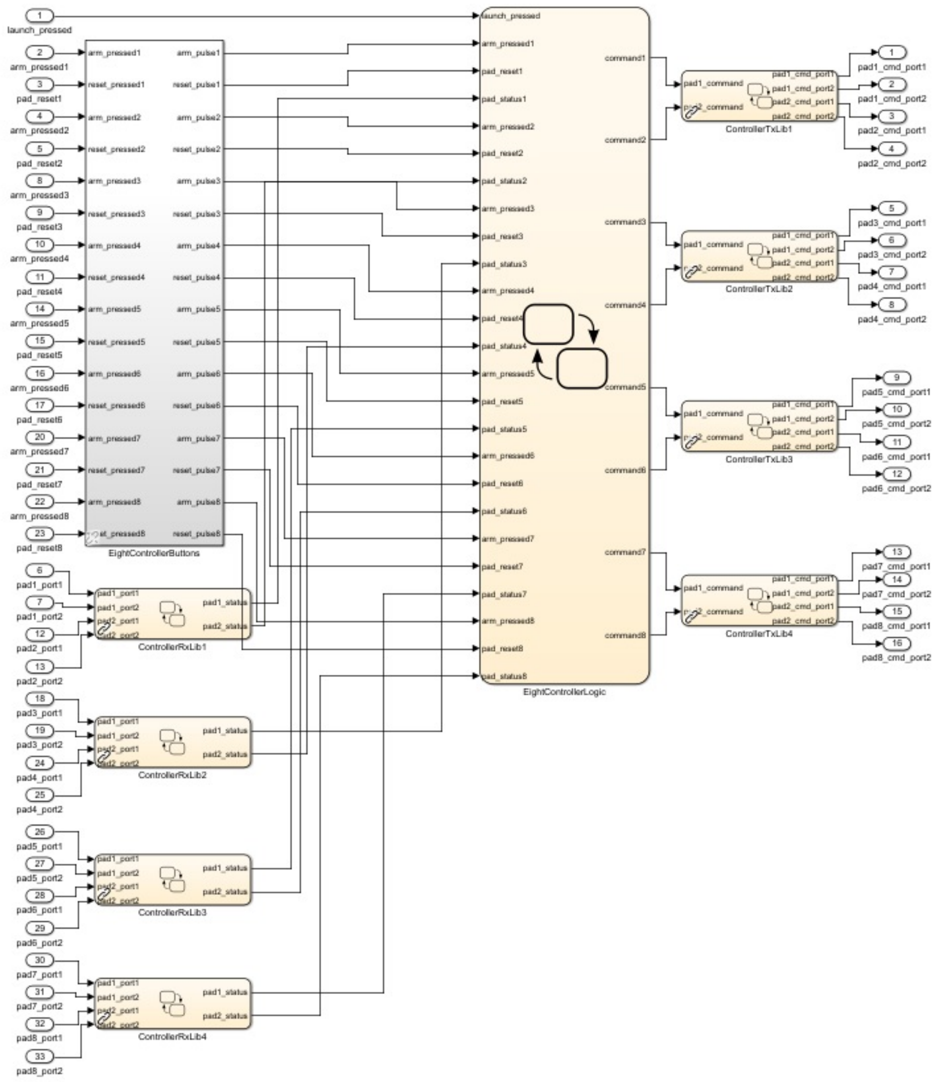


Figure 2.20: ControllerSW Model

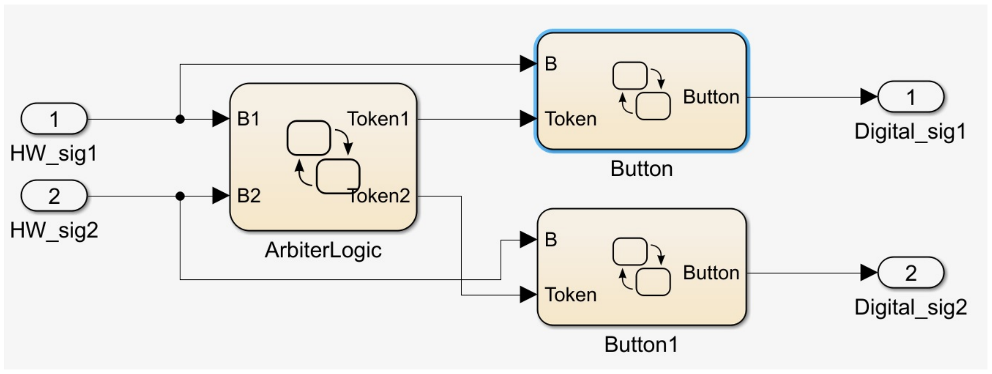


Figure 2.21: Button Library for One Atomic ControllerLogic

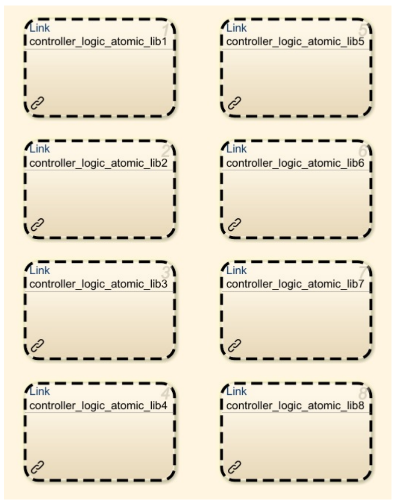

Link
controller_logic_atomic_lib1
Link
controller_logic_atomic_lib5
Link
controller_logic_atomic_lib2
Link
controller_logic_atomic_lib6
Link
controller_logic_atomic_lib3
Link
controller_logic_atomic_lib7
Link
controller_logic_atomic_lib4
Link
controller_logic_atomic_lib8

Figure 2.22: Eight Atomic ControllerLogic

2.4 Rocket Launch System

Given the modeling described this far, we are now ready to show the entire Rocket Launch System, simulation test, and the way to do the hardware in the loop (HIL) test.

2.4.1 Combining Models to Build Rocket Launch System

Figure 2.23 shows the entire rocket launch system which includes four PadLibs, a Communication Delay block, and a Controller. The PadLib is the same we developed in section 2.2. The Controller is also the same as in section 2.3. The purpose of the Communication Delay is generating a unit delay to simulate a wireless networking environment which has a delay time.

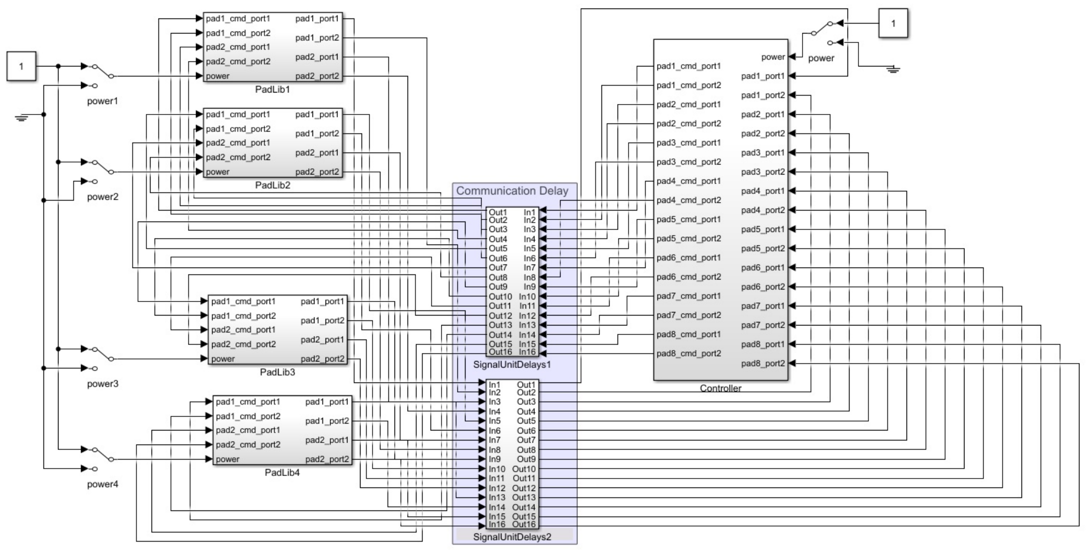


Figure 2.23: Rocket Launch System

When I did simulation testing, the whole system showed the desired behaviors. Since we only simulate on the software virtual environment, so we could not completely trust the result of the simulation testing when the system is realized to a physical system. Hence we decided to do HIL simulation to generate the model to C code and to upload code to a microprocessor. For the HIL simulation, we split the system into a smaller system which contained a PadLib and a Controller. The reason was that the number of DAQ port was less then the number of communication channels. But we designed the system using libraries and atomic states so we assumed that if we ensure the smaller system, we can also trust the entire system.

2.4.2 HIL Simulation Environment

To search for PadLib and a Controller design errors, we decided to do HIL simulation tests. We constructed two parts which were PadLib as software part and Controller as hardware parts. To simulate software and hardware at the same time, we used the DAQ device to connect the two.

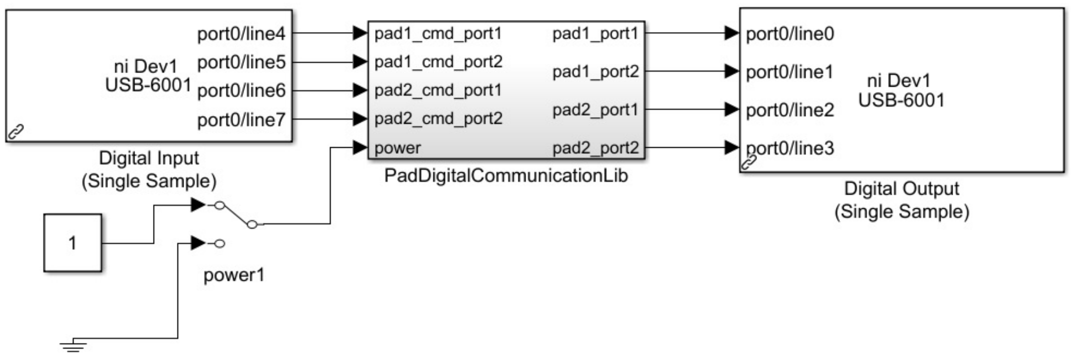


Figure 2.24: HIL Simulation Software Environment

Figure 2.24 shows the software part. The PadLib (PadDigitalCommunicationLib) connects with the Digital Input/Output blocks. Simulink supports the blocks to connect to the NI-DAQ device. The ControllerSW, Figure 2.25, was designed to control one PadLib. We generated C code from the ControllerSW model using the embedded coder which is the add-on toolbox in Simulink. To build a hardware system, we used a microprocessor (a Nucleo-F412ZG). We developed glue code to connect the generated C codes to the given Nucleo libraries in the online IDE. We built a simple circuit to generate the hardware button signals and to connect the DAQ to the microprocessor. Figure 2.26 represents the hardware system and DAQ device.

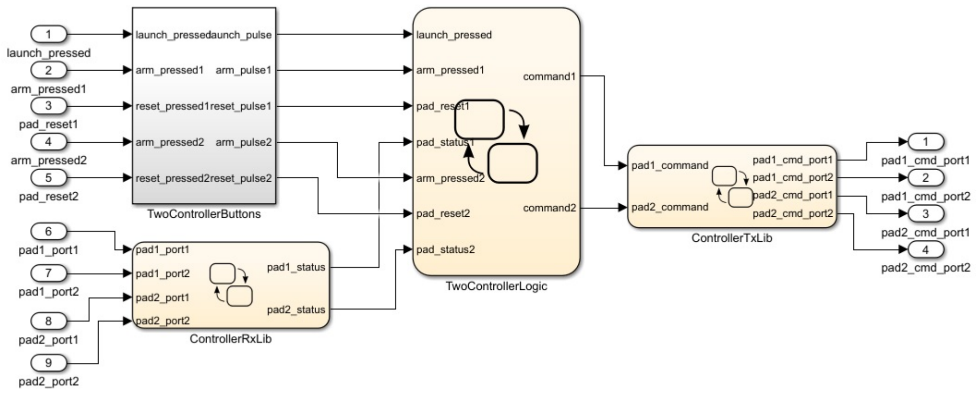


Figure 2.25: ControllerSW Model for One PadLib

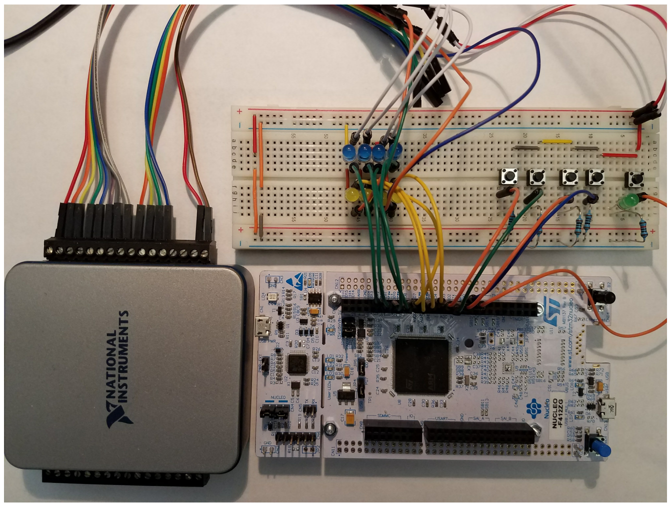


Figure 2.26: HIL Simulation Hardware Environment

2.4.3 HIL Simulation Result

When I simulated the system, I got the expected outputs according to the input sequences. Also, I tested the previous button problem in subsection 2.1.2 and there were no errors. The binary communication through the DAQ device was successful. According to the HIL test result, we could say this system was reliable and correct, and consequently we could trust the entire system. However, given the consequences of failures if a person was standing next to a rocket inadvertently launched, we proceeded to formal verification to ensure that our safety requirements were satisfied in our system.

Chapter 3

Model Formal Verification

In the previous chapter, we confirmed that the Rocket Launch System works well through simulation and testing. Nevertheless, there was some uncertainty because we tested it with a limited number of simulation input conditions. Also, a critical system shall always meet its safety requirements. Hence, we examined the system with all possible input conditions using the Design Verifier, which is a toolbox in Simulink.

The first thing to do to formally verify that the design behavior meets its requirements is to write the safety requirements in a natural language. After that, using MATLAB functions, Simulink, and Stateflow, the formal requirements can be expressed. The requirements model, which is called property check model in the Simulink Design Verifier, can be used to prove the correctness of a design with respect to these properties using formal methods, which Design Verifier supports.

In this chapter, we show that we verified the design correctness of the PadSW in the development model in the PadLib with respect to three safety requirements. The reason why we only tested the PadSW at this time was twofold; the notion of formal versification was new to me so it was a steep learning curve to get the verification going and even when more experienced, the formal verification involved surprisingly much effort and cost to build the property check models. We discuss the three property check models and the results of the analysis in the next few sections. At the end of this chapter, we illustrate how the results of the analysis was used to modify the model so that it met its requirements.

3.1 Designing Property Check Models

Figure 3.1 shows the overall formal verification model. The PadSW is connected to input and output ports. The three property check models are connected to some system inputs and outputs relevant to the property at hand; the requirement models only care about the inputs and outputs from the model under investigation, they care about the black-box behavior without regard to the internal design of the model under investigation. In the next subsection, we will cover the translation processes for each of the property check models.

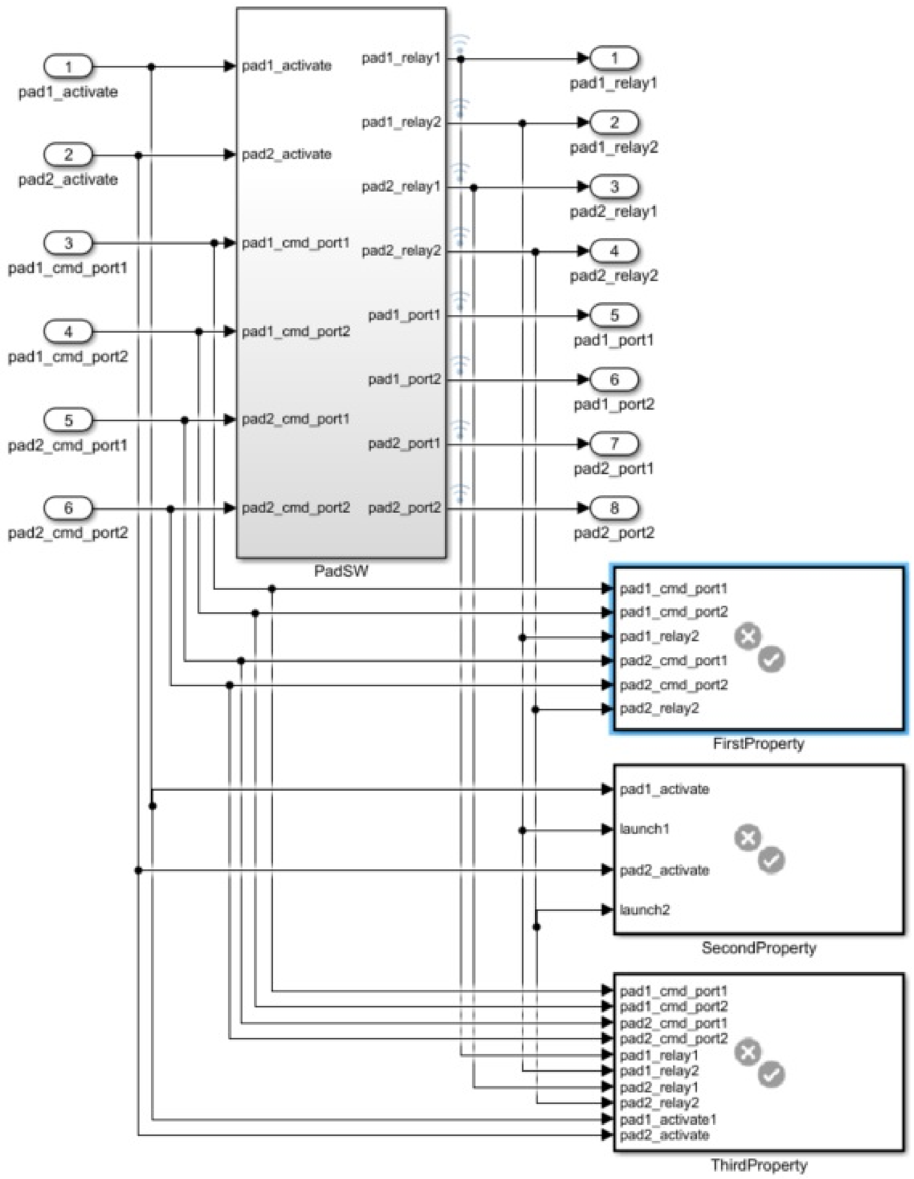


Figure 3.1: Overall Formal Verification Model for PadSW Model

3.2 First Property Model

The first safety requirement for the PadSW model is that the rocket shall launch only when the Controller sends the launch signal. In other words, if the controller does not send the launch command, the PadSW will never fire a rocket. Failure to observe this requirement can be fatal, for example, users can be at risk if their rockets are fired when they install their rockets on the Pad. We wrote the requirement informally in natural language as follows: Without a launch signal, the 'pad1_relay2/pad2_relay2' shall not be high. The launch signal is an input condition which is that the port1 is true and port2 is false (the binary encoding we developed for the enumerated types) as shown in Figure 2.15. The pad1_relay2 and pad2_relay2 are outputs that are only true when the Pad launches rockets on the Pad, these are the outputs that cause the rocket engine to ignite.

3.2.1 Designing the First Property Model

For the first requirement, we designed a property check models using Simulink logical operators as shown in Figure 3.2. If the outputs of the property check models are always false then the result of the property testing says that the PadSW fulfilled the first requirement. The Implies block is a logical operator, its outputs false when the first input is true and second input is false, otherwise, the output true. The condition that pad1_relay2/pad2_relay2 are true is the same as the false output condition of the Implies block; hence, the outputs of the property check model should always be false.

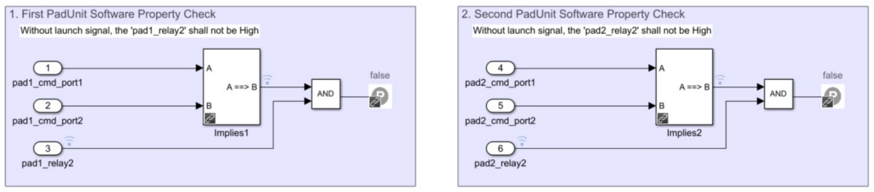

1. First PadUnit Software Property Check
Without launch signal, the 'pad1_relay2' shall not be High
pad1_cmd_port1
pad1_cmd_port2
pad1_relay2
A ==> B
Implies1
AND
false
2. Second PadUnit Software Property Check
Without launch signal, the 'pad2_relay2' shall not be High
pad2_cmd_port1
pad2_cmd_port2
pad2_relay2
A ==> B
Implies2
AND
false

Figure 3.2: Overall Formal Verification Model for PadSW Model

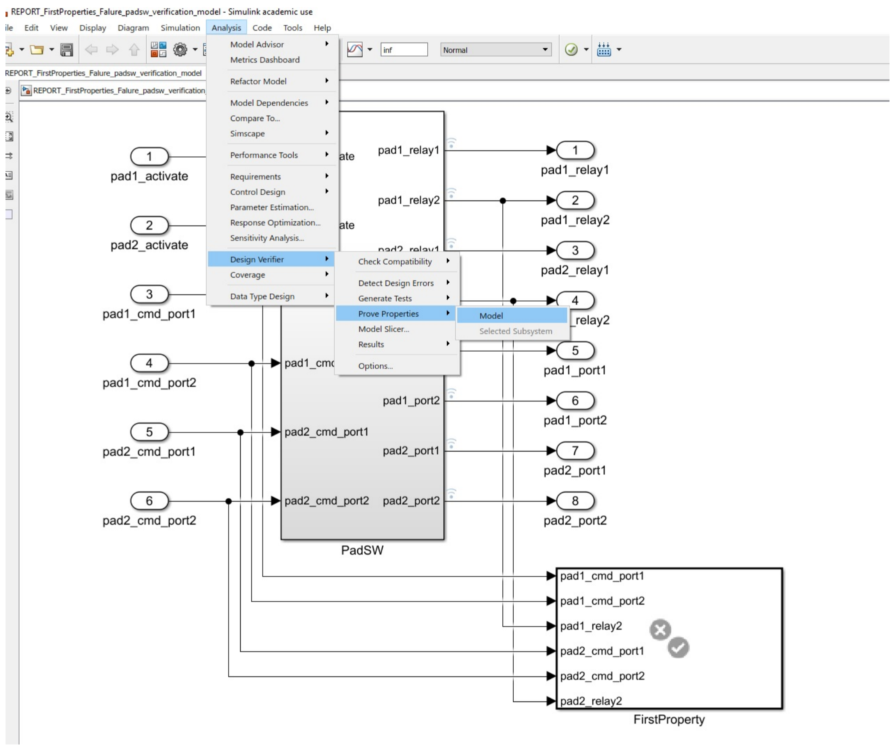
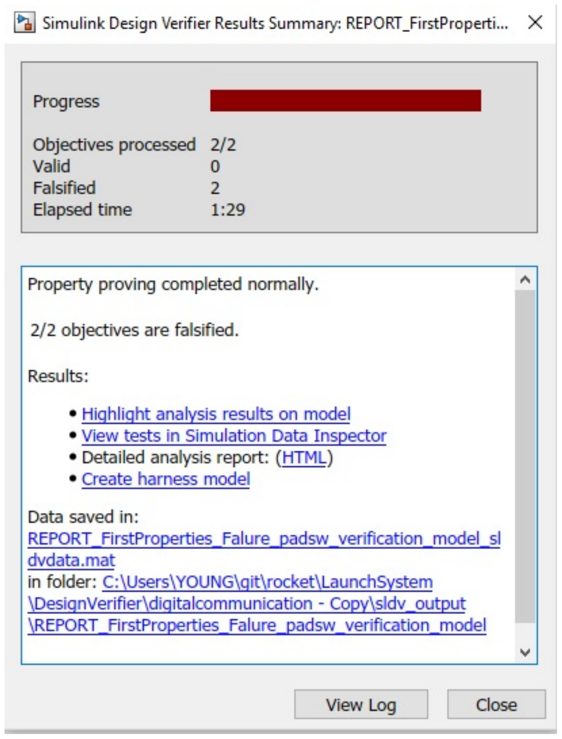


Figure 3.3: Running the Prove Properties Program in the Simulink

3.2.2 Running Design Verifier for First Property

Running the Design Verifier was easy. As Figure 3.3 explains, the way to run the prove properties program is achieved through menu selections. The Design Verifier created a report that shows the list of objectives for which the design was proven valid and the list of objectives for which the analysis found counterexamples. A counterexample is a sequence of inputs that will lead to one of the properties being violated (indicated by the output from the property block being true). The report shown in Figure 3.4 shows that the PadSW was falsified against the requirement, a surprise to the analyst since this seemed like one of the easiest safety properties to ensure true. One of the merits to use Simulink Design Verifier is they can create a harness model (Figure 3.5) with counterexamples (Figure 3.6) that the analysis found; in essence, the harness model is a test driver that can show us how the property is violated in the model.

Using the counterexample, we found a faulty design of the properties check model (Figure 3.2); at this point, we considered the model to be correct, but the safety requirement we formalized was wrong. The problem was that we did not consider the output delay time. To be specific, when the inputs changed, the output does not change immediately. Processing time in the PadSW leads to a delay until the output is produced; the command is given, but the close


Simulink Design Verifier Results Summary: REPORT_FirstProperti...

Progress

Objectives processed 2/2
Valid 0
Falsified 2
Elapsed time 1:29

Property proving completed normally.

2/2 objectives are falsified.

Results:

- Highlight analysis results on model
- View tests in Simulation Data Inspector
- Detailed analysis report: (HTML)
- Create harness model

Data saved in:
REPORT_FirstProperties_Falure_padsw_verification_model_sl
dvdata.mat
in folder: C:\Users\YOUNG\git\rocket\LaunchSystem
\DesignVerifier\digitalcommunication - Copy\sldv_output
\REPORT_FirstProperties_Falure_padsw_verification_model

Figure 3.4: Design Verifier Report for the First Property Check Model

relay signal is not produced until the next time step. One can look at the safety property and think that this behavior would be OK; it is fine to give the launch command and not launch (step 1) and then in step 2 the launch command remains and we launch the rocket. The problem is when the launch command goes away, the relay will not be closed until in the subsequent step. Thus, before the relay signal goes to zero, there will be a step where there is no launch command, but the launch signal is high–a violation of the property. As mentioned above, there is a delay in setting the launch signal high; similarly, there is a delay in setting it to off and that leads to a property violation.

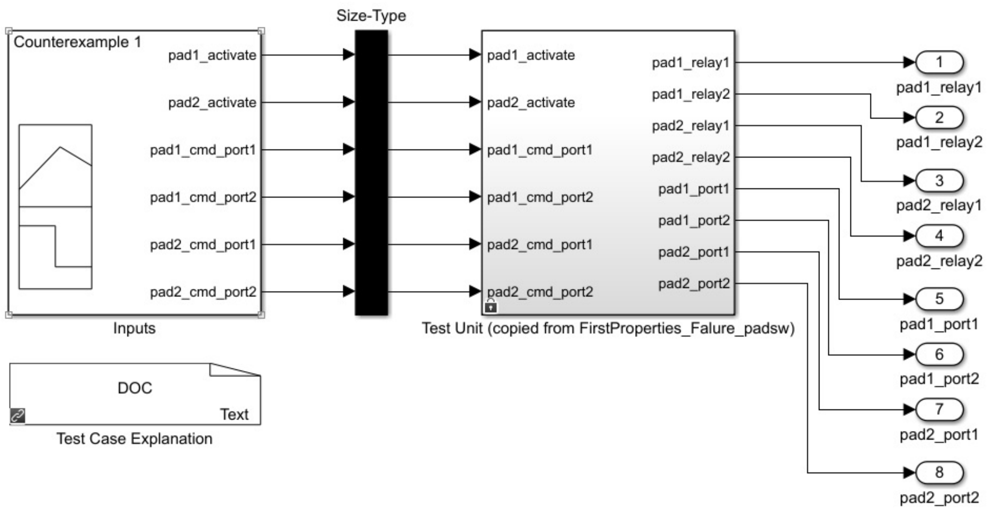


Figure 3.5: Harness Model created by Design Verifier

We modified the requirement blocks and added a delay block, Not and And logic operators as shown in Figure 3.7. These operators made the pad1_relay2/pad2_relay2 signals delayed so that the last And operator compared synchronized signals. We analyzed the PadSW against with the first property, using the Design Verifier. The report (Figure 3.8) showed that the PadSW model was proven correct with respect to this requirement.

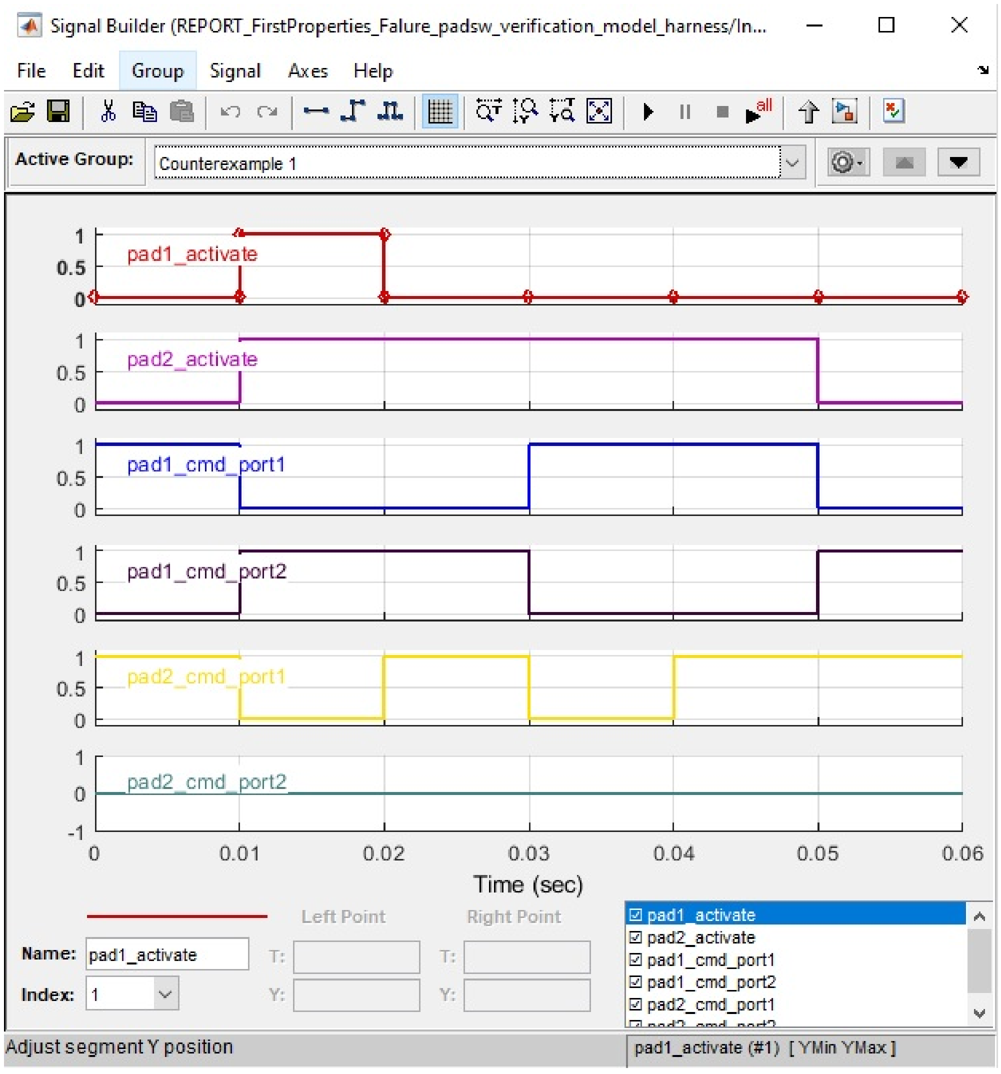
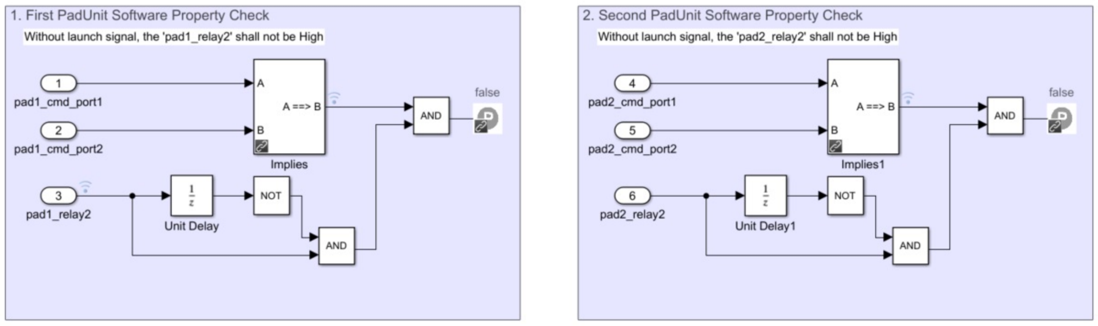
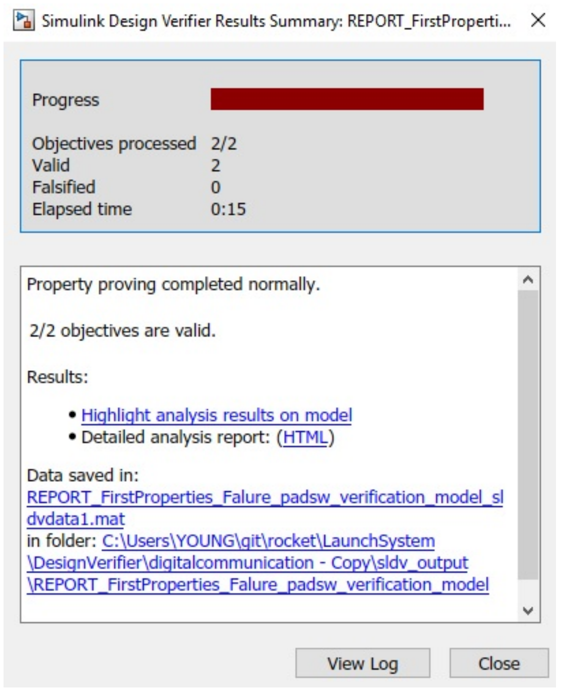

Signal Builder (REPORT_FirstProperties_Falure_padsw_verification_model_harness/In...
File Edit Group Signal Axes Help
Active Group: Counterexample 1
pad1_activate
pad2_activate
pad1_cmd_port1
pad1_cmd_port2
pad2_cmd_port1
pad2_cmd_port2
Time (sec)
Left Point
Right Point
Name: pad1_activate
Index: 1
Adjust segment Y position
pad1_activate (#1) [YMin YMax]

Figure 3.6: A Counter Example for the First Property Check Model



1. First PadUnit Software Property Check
Without launch signal, the 'pad1_relay2' shall not be High
1
pad1_cmd_port1
2
pad1_cmd_port2
3
pad1_relay2
A
B
A ==> B
Implies
Unit Delay
NOT
AND
AND
false
2. Second PadUnit Software Property Check
Without launch signal, the 'pad2_relay2' shall not be High
4
pad2_cmd_port1
5
pad2_cmd_port2
6
pad2_relay2
A
B
A ==> B
Implies1
Unit Delay1
NOT
AND
AND
false

Figure 3.7: Revised Version of the First Property Check Model


Simulink Design Verifier Results Summary: REPORT_FirstProperti...

```
Progress

Objectives processed  2/2
Valid                 2
Falsified             0
Elapsed time          0:15
```

Property proving completed normally.

2/2 objectives are valid.

Results:

- Highlight analysis results on model
- Detailed analysis report: (HTML)

Data saved in:
REPORT_FirstProperties_Falure_padsw_verification_model_sldvdata1.mat
in folder: C:\Users\YOUNG\git\rocket\LaunchSystem\DesignVerifier\digitalcommunication - Copy\sldv_output\REPORT_FirstProperties_Falure_padsw_verification_model

Figure 3.8: Design Verifier Report for the Renewal First Property Check Model

3.3 Second Property Model

The second critical requirement is that the rocket must not fire if the owner did not push a pad_activate button. To be specific, if the rocket can be fired before pressing the pad button, the rocket may be fired while an owner is installing the rocket, resulting in a risk of serious injury to the user. We have expressed the property to the formal requirement as follows: If 'pad1_activate/pad2_activate' have not been set high (we have not seen a rising edge on the signal), the 'pad1_relay2/pad2_relay2' shall not be high.

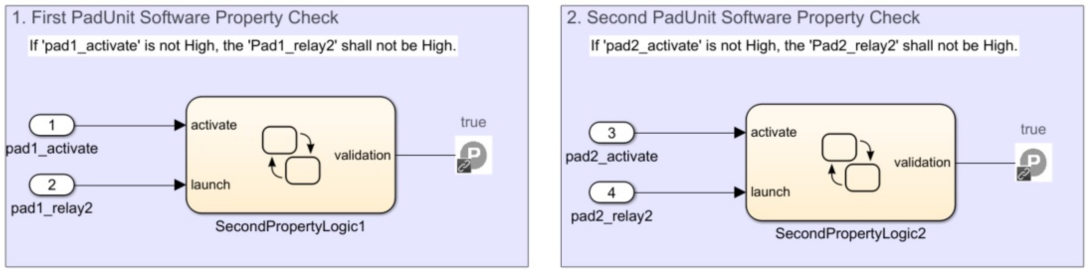

1. First PadUnit Software Property Check
If 'pad1_activate' is not High, the 'Pad1_relay2' shall not be High.
1
pad1_activate
2
pad1_relay2
activate
launch
validation
SecondPropertyLogic1
true
2. Second PadUnit Software Property Check
If 'pad2_activate' is not High, the 'Pad2_relay2' shall not be High.
3
pad2_activate
4
pad2_relay2
activate
launch
validation
SecondPropertyLogic2
true

Figure 3.9: Second Property Check Model

3.3.1 Designing Second Property Model

The difference as compared to the first property model is that the second model is designed with Stateflow as shown in Figure 3.9, not using the MATLAB Simulink logical operators. Developing a property model the Stateflow was more intuitive and easier to model than when modeling requirements with logical operators.

In Figure 3.10, there are two state machines that operate in parallel. The DetectActivate detects the pad1_activate/pad2_activate signal and the Verification judges if the PadSW model is correct or falsified. If the pad1_relay2/pad2_relay2 is true and the status of DetectActive is false, then the system does not fulfill the requirement, in all other conditions, the system is correct with respect to our second safety requirement.

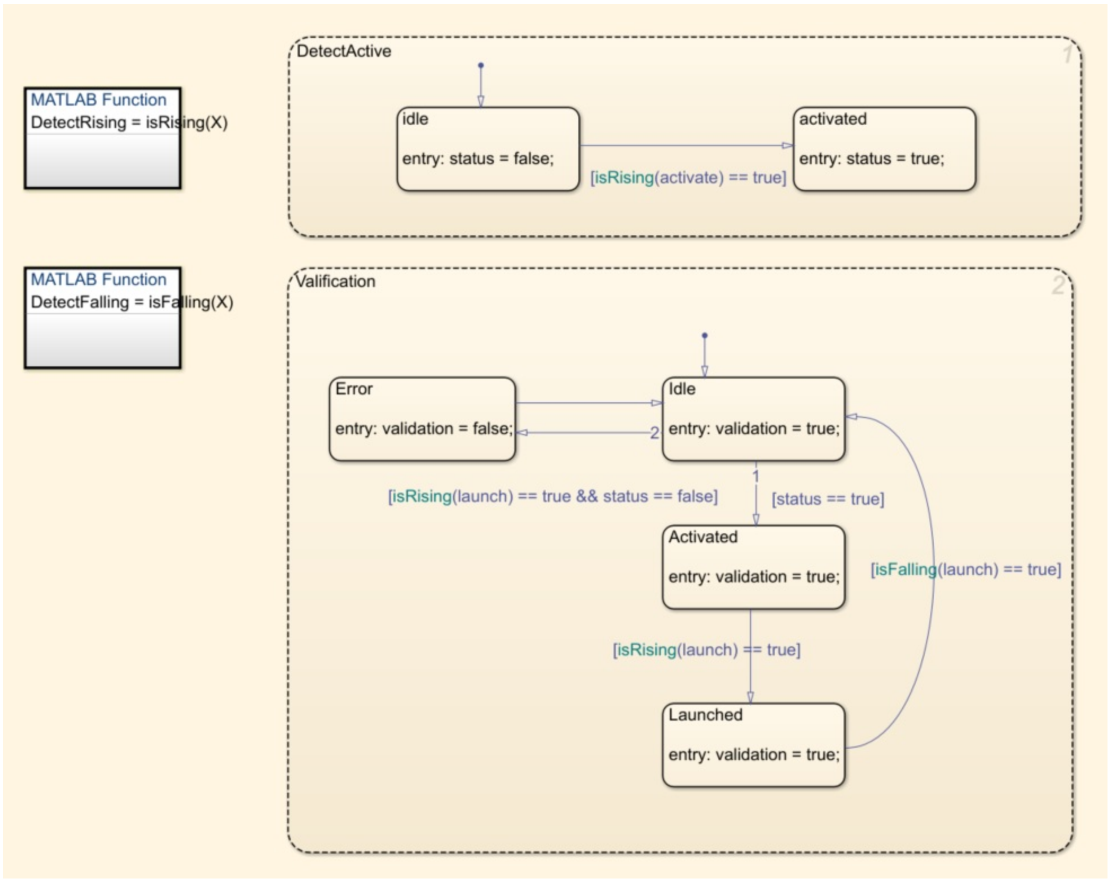
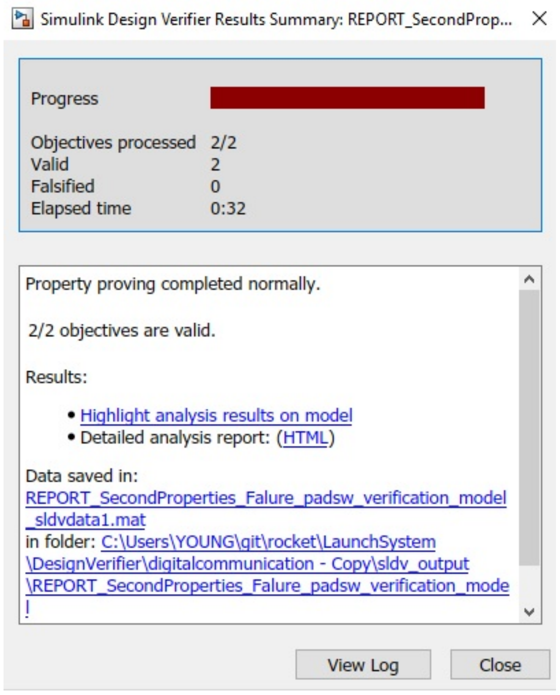

MATLAB Function
DetectRising = isRising(X)
MATLAB Function
DetectFalling = isFalling(X)
DetectActive
1
idle
entry: status = false;
[isRising(activate) == true]
activated
entry: status = true;
Valification
2
Error
entry: validation = false;
Idle
entry: validation = true;
2
1
[isRising(launch) == true && status == false]
[status == true]
Activated
entry: validation = true;
[isFalling(launch) == true]
[isRising(launch) == true]
Launched
entry: validation = true;

Figure 3.10: Second Property Logic State Machine

3.3.2 Running Design Verifier for Second Property Check Model

When we run the Design Verifier to attempt to prove our second property, as shown in Figure 3.11, the results summary said that the PadSW was valid against the second requirement. Based on the formal verification, we surely can say that the system meets the second requirement formalized requirement. What we cannot guarantee is that we have formalized the requirement correctly. This is a concern with formal verification and the validation of the formalized properties is an ongoing area of research; “How do we know when we have gotten our properties right?”

In particular, this is a problem when the verification indicates that there is no violation. This can be because the model and property are both correct, but it can also be because the property is incorrect and it now fails to catch an incorrect model. The user does not know which one and might erroneously believe it is because of the former. On the other hand, if we get a counterexample, as for the verification of the first property above, the user has concrete information about what went wrong and can correct the model or the property depending on which one is deemed to be incorrect (in the example above, the property was the one that was wrong).


Simulink Design Verifier Results Summary: REPORT_SecondProp...

```
Progress

Objectives processed  2/2
Valid                 2
Falsified             0
Elapsed time          0:32
```

Property proving completed normally.

2/2 objectives are valid.

Results:

- Highlight analysis results on model
- Detailed analysis report: (HTML)

Data saved in:
REPORT_SecondProperties_Falure_padsw_verification_model_sldvdata1.mat
in folder: C:\Users\YOUNG\git\rocket\LaunchSystem\DesignVerifier\digitalcommunication - Copy\sldv_output\REPORT_SecondProperties_Falure_padsw_verification_mode l

Figure 3.11: Design Verifier Report for the Second Property Check Model

3.4 Third Property Model

The last safety requirement is that if the PadSW gets a reset command from the Controller, then the rocket shall not launch and the PadSW goes back to the initial state. The meaning of pressing the reset button on the Controller is the launch officer needed to cancel the launch for some reasons. Hence, it is one of the critical safety requirements. We had expressed the property as a natural language expression as follows: If pad1/pad2 is reset, the pad1_relays/pad2_relays do not close. An alternate formulation is that upon a reset, the two relays must be opened preventing a launch.

3.4.1 Designing the Third Property Model

We realized when defining property two that using Stateflow to capture properties was relatively easy (as described on Section 3.3) so we decided to use Stateflow again for our third property. Figure 3.12 shows the inputs that the ThirdPropertyLogic uses and Figure 3.13 represents the details of the ThirdPropertyLogic.

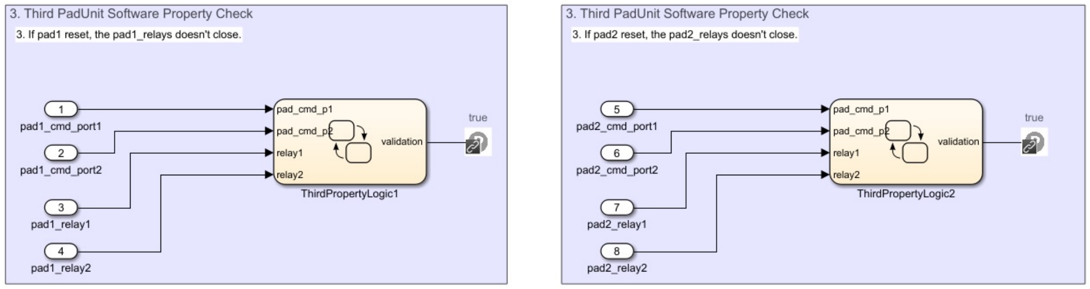


Figure 3.12: Third Property Check Model

In the logic, the DetectReset state and Verification state operate simultaneously. The reset signal detected in the DetectReset state and the detecting condition is that pad_cmd_p1 and pad_cmd_p2 are true (the encoding for a reset). In the Verification state, the results of the property check are set. If one of the relays is true after the reset status is true, then the PadSW is not correct with respect to the third safety requirement. The purpose of the MATLAB function, after(80, msec), is to synchronize the time delay to decode the reset signal in the PadRxLib model.

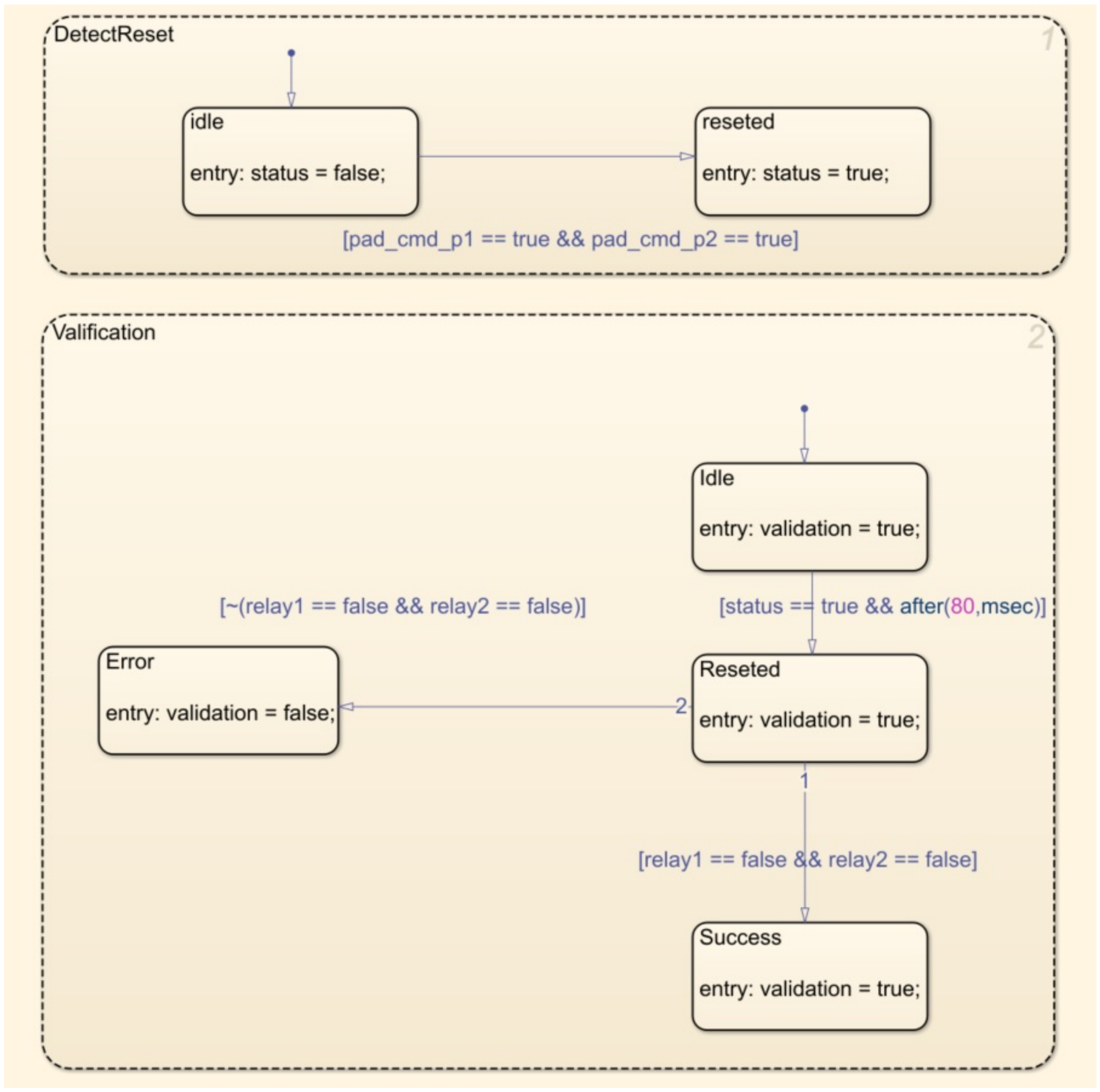
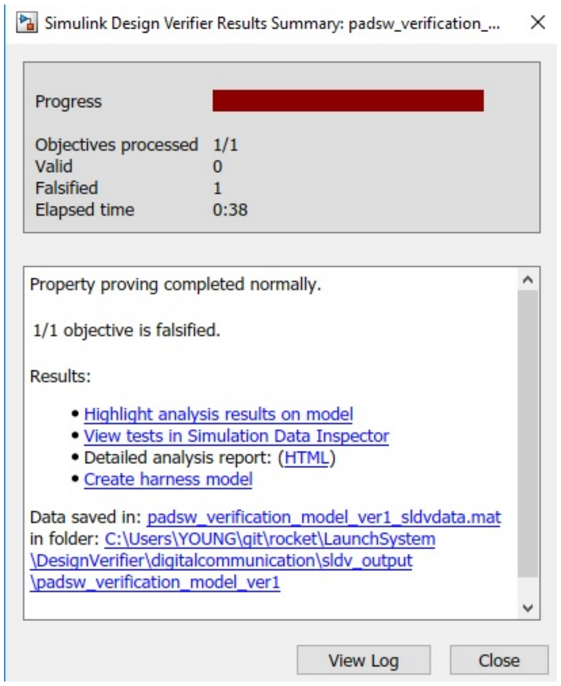

DetectReset
1
idle
entry: status = false;
reseted
entry: status = true;
[pad_cmd_p1 == true && pad_cmd_p2 == true]
Valification
2
Idle
entry: validation = true;
[status == true && after(80,msec)]
[~(relay1 == false && relay2 == false)]
Error
entry: validation = false;
2
Reseted
entry: validation = true;
1
[relay1 == false && relay2 == false]
Success
entry: validation = true;

Figure 3.13: Third Property Logic State Machine

3.4.2 Running Design Verifier for Third Property Check Model

We ran the Design Verifier to attempt to prove the correctness of the PadSW design by examining all possible input conditions. The report (Figure 3.14) represented the objectives are falsified.


Simulink Design Verifier Results Summary: padsw_verification_...

Progress	
Objectives processed	1/1
Valid	0
Falsified	1
Elapsed time	0:38

Property proving completed normally.

1/1 objective is falsified.

Results:

- Highlight analysis results on model
- View tests in Simulation Data Inspector
- Detailed analysis report: (HTML)
- Create harness model

Data saved in: padsw_verification_model_ver1_sldvdata.mat
in folder: C:\Users\YOUNG\git\rocket\LaunchSystem\DesignVerifier\digitalcommunication\sldv_output\padsw_verification_model_ver1

Figure 3.14: Design Verifier Report for the Third Property Check Model

To figure out what is the problem we created a harness model and got a counterexample (Figure 3.15). According to analysis of the counterexample, we found out the PadRxLib in the PadSW could not process the received signal if the signal is too short.

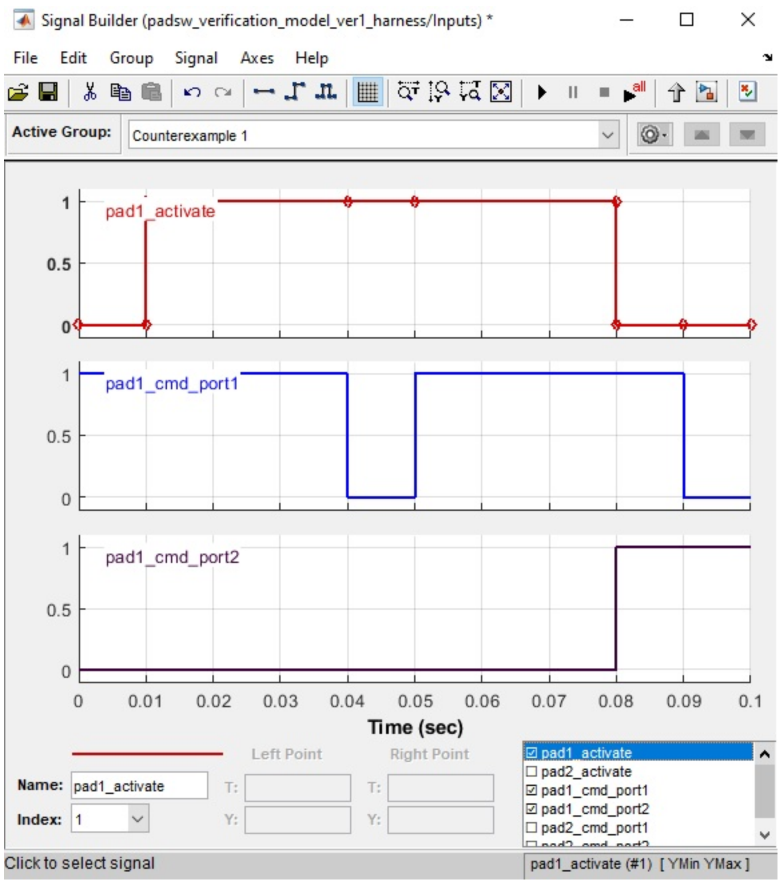


Figure 3.15: Design Verifier Report for the Third Property Check Model

Also, we figured out the pad_activate can be a problem. If the signal has a falling edge right before the reset signal, then the Pulse Generator in the PadSW activates the PadUnitLogic for 1 second. shown in Figure 2.4, the Pulse Generator generates 1 second true signal when the pad1_activate has a falling edge. So following the signal sequence brings out a critical safety problem that the reset button do not stop to fire a rocket in the dangerous situation. The signal sequence is that a short CMD_RESET signal (less than 1 second) after falling edge of pad1_activate signal, CMD_ARM, and CMD_LAUNCH then the rocket will be launched.

3.4.3 Re-designing Third Property Model

Based on the results of the previous test, we built a new property model (Figure 3.16). The requirements model was simpler than the previous model and added a condition to detect the pad_activate signal.

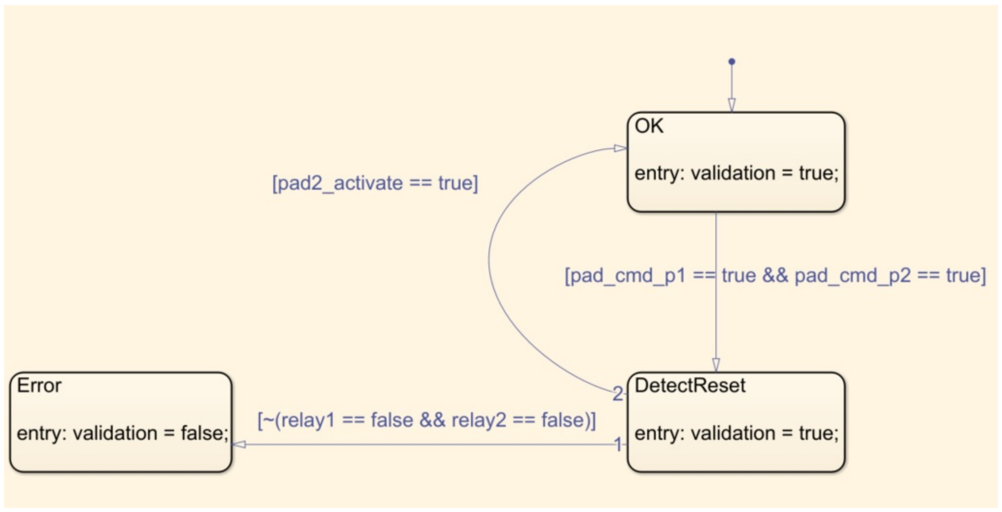

OK
entry: validation = true;
[pad2_activate == true]
[pad_cmd_p1 == true && pad_cmd_p2 == true]
Error
entry: validation = false;
[~(relay1 == false && relay2 == false)]
2
1
DetectReset
entry: validation = true;

Figure 3.16: Improved State Machine that Describes Third Property Check Model

3.4.4 Re-running Design Verifier for Third Property Check Model

Using the above property check model, we verified the PadSW again. However, the system did not meet the requirement. The counterexample (Figure 3.17) showed the same problem as the previous verification result. The PadRxLib did not detect the reset signal because it was too short. At this point, we considered that the PadSW design might be wrong and we decided to fix the property model and to change the PadRxLib model.

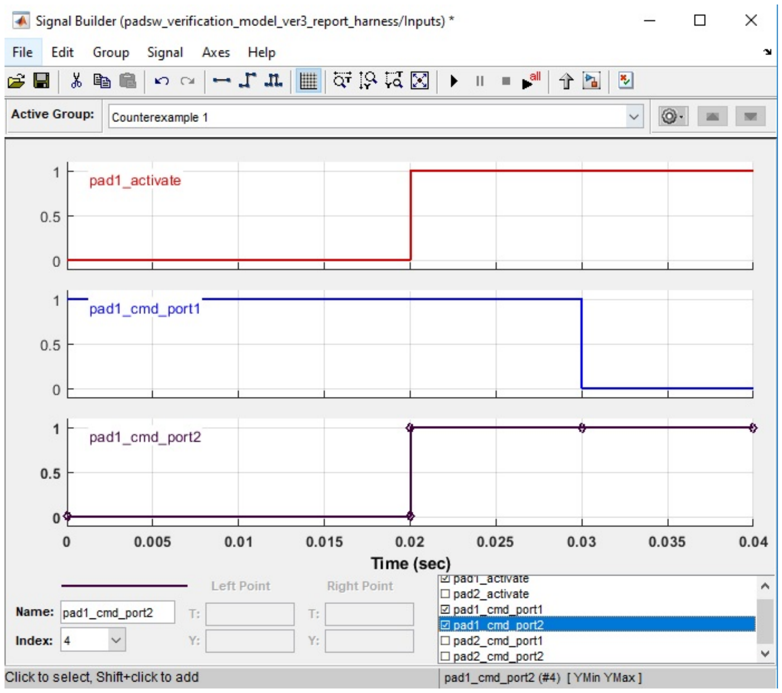


Figure 3.17: A Counter Example for Improved Third Property Check Model

3.4.5 Fixing the defects of the PadSW

When we saw the PadRxLib in Figure 2.15, it needed two state transitions to change the command output value. We redesigned the PadRxLib as shown in Figure 3.18 to respond to short inputs signal by reducing the number of transition step needed to react to the input. Then ran the Design Verifier to check the new PadRxLib solved the problem. However, the result (Figure 3.19) said the objectives had the same problem (counterexample).

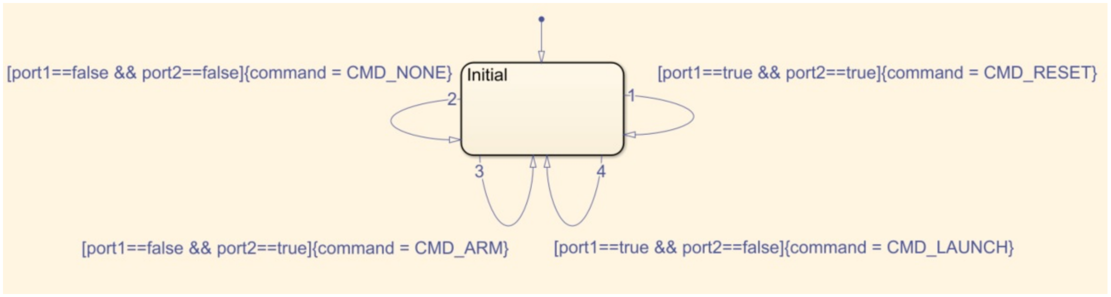
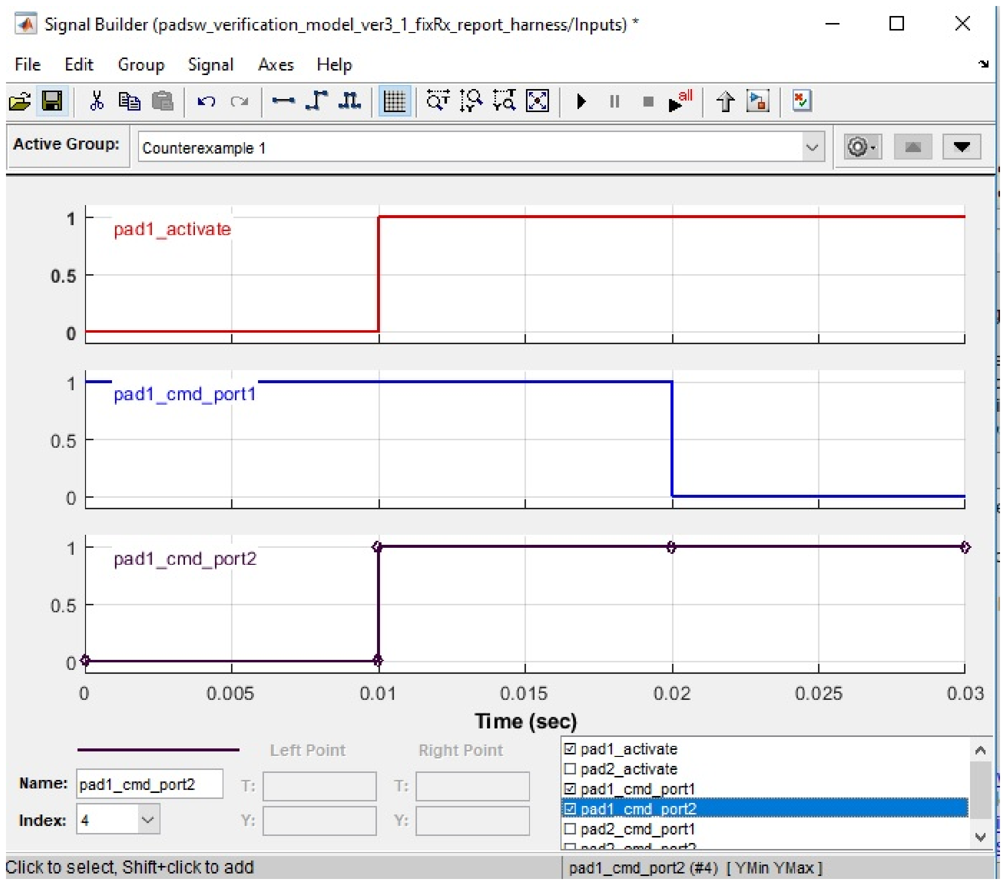

[port1==false && port2==false]{command = CMD_NONE}
Initial
[port1==true && port2==true]{command = CMD_RESET}
2
1
3
4
[port1==false && port2==true]{command = CMD_ARM}
[port1==true && port2==false]{command = CMD_LAUNCH}

Figure 3.18: Redesigned PadRxLib Model

The second problem that we found was the Pulse Generator activates the PadUnitLogic for 1 second at the falling edge of a pad_activate signal. Figure 3.20 (a) shows the edited Pulse Generator Stateflow logic. The output is true during a transition time when the input signal is the rising edge. After changing the Pulse Generator, the Design Verifier could prove that PadSW satisfies third property (Figure 3.20 (b)).



Active Group: Counterexample 1

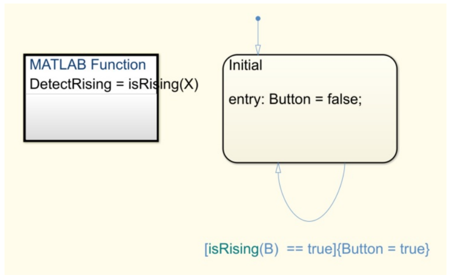
Name: pad1_cmd_port2

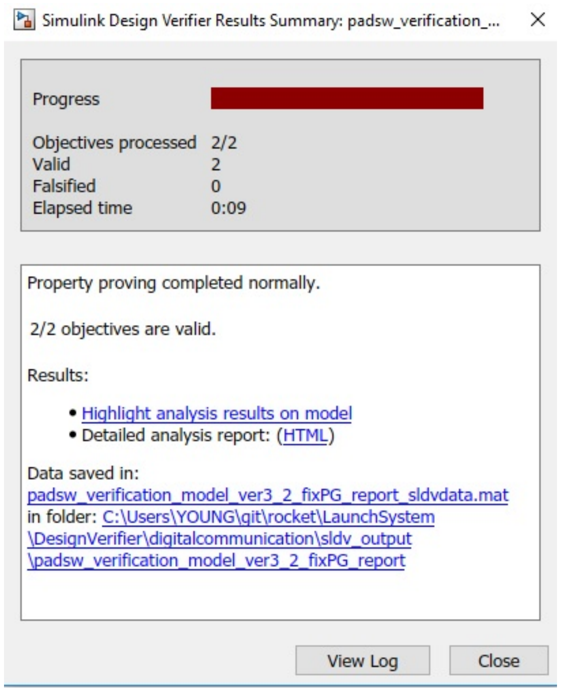
Index: 4

Figure 3.19: A Counter Example for Improved Third Property Check Model after Redesigned PadRxLib Model

3.4.6 Discussion

As mentioned earlier, if the verification leads to a counterexample, there is concrete information that can be used to improve the model or the property that has been violated. On the other hand, if the properties are verified, there is a concern whether or not the properties have been captured correctly. Unfortunately, there is little guidance as to how to ensure that the properties are correct. This is a


MATLAB Function
DetectRising = isRising(X)
Initial
entry: Button = false;
[isRising(B) == true]{Button = true}


Simulink Design Verifier Results Summary: padsw_verification_...

```
Progress
Objectives processed  2/2
Valid                 2
Falsified             0
Elapsed time          0:09
```

Property proving completed normally.

2/2 objectives are valid.

Results:

- Highlight analysis results on model
- Detailed analysis report: (HTML)

Data saved in:
padsw_verification_model_ver3_2_fixPG_report_sldvdata.mat
in folder: C:\Users\YOUNG\git\rocket\LaunchSystem
\DesignVerifier\digitalcommunication\sldv_output
\padsw_verification_model_ver3_2_fixPG_report

Figure 3.20: (a) Redesigned Pulse Generator Model (b) Design Verifier Report of Verifying Third Property

concern in this project since a miss-specified property is seemingly very easy to construct. In our opinion, much more guidance as to how to effectively specify and validate properties is needed.

Chapter 4

Summary and Conclusion

We have developed a critical system, the rocket launch system, using MATLAB/Simulink, which are model-based development tool suites. In the development process, I used three different verification techniques that are supported by MATLAB/Simulink. The verification techniques have merits but also have disadvantages.

In chapter 2, we used the simulation test and hardware in the loop simulation. The advantages of the simulation test are that they are quite easy to use and it is fast to check the objective behaviors, but a limited number of input conditions reduces confidence in the simulation test result. The merit of the hardware in the loop simulation is that the software model can be tested in the real hardware environment. The disadvantage is that it takes more cost than the simulation test to design a circuit and to write glue code to connect the auto-generated code with the libraries of the microprocessor.

In chapter 3, we used the formal verification technique. The verification is requirement based testing. The Design Verifier examines all possible input conditions to prove the correctness of a system against safety requirements. The Design Verifier caught design errors, however, other verification technique did not. The reliability of test results is the highest of three techniques, but developing property check model involves more effort, cost and can't make easy for big systems.

In this paper, we focus on the development of the critical system. For the future works, I want to compare the results to verify whole system using formal verification and testing using object code coverage [5]. The object code coverage is one of the research topics in the UMN Crisys Group.

Bibliography

[1] *TripolyMN: High Power Rocketry Club*. URL: `http://www.tripolimn.org/`.

[2] *MATLAB*. URL: `https://www.mathworks.com/?s_tid=gn_logo/`.

[3] Hoang Pham. *Software reliability*. Springer Science & Business Media, 2000.

[4] *National Instruments*. URL: `http://www.ni.com/data-acquisition/`.

[5] Taejoon Byun et al. “Toward Rigorous Object-Code Coverage Criteria”. In: *Software Reliability Engineering (ISSRE), 2017 IEEE 28th International Symposium on*. IEEE. 2017, pp. 328–338.