

University of Arkansas, Fayetteville

ScholarWorks@UARK

Electrical Engineering Undergraduate Honors
Theses

Electrical Engineering

5-2018

Command Validation for Cybersecure Power Router

Isaac M. Kroger

University of Arkansas, Fayetteville

Follow this and additional works at: <https://scholarworks.uark.edu/eleguht>



Part of the [Electrical and Electronics Commons](#), and the [Other Electrical and Computer Engineering Commons](#)

Citation

Kroger, I. M. (2018). Command Validation for Cybersecure Power Router. *Electrical Engineering Undergraduate Honors Theses* Retrieved from <https://scholarworks.uark.edu/eleguht/57>

This Thesis is brought to you for free and open access by the Electrical Engineering at ScholarWorks@UARK. It has been accepted for inclusion in Electrical Engineering Undergraduate Honors Theses by an authorized administrator of ScholarWorks@UARK. For more information, please contact scholar@uark.edu, uarepos@uark.edu.

Command Validation for Cybersecure Power Router


An undergraduate honors college thesis submitted in partial fulfillment
of the requirements for the degree of
Bachelor of Science in Electrical Engineering with Honors

by

Isaac M. Kroger

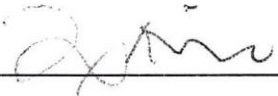
April 2018
University of Arkansas

This thesis is approved.

 H. Alan Mantooh
2018.04.30 18:00:25 -05'00'

Dr. H. Alan Mantooh

Thesis Director



Dr. Jingxian Wu

Department Honors Advisor

Abstract

For grid-connected homes equipped with solar panels, power electronics are necessary to manage and convert power between the solar panels, battery storage, grid, and residential load. A power router can be used to manage these power electronics and govern power generation, storage, and distribution within the household. This level of control makes power routers that do not employ cybersecurity a target for external attacks. The use of command validation is an effective way to prevent unauthorized commands from maliciously altering the state of a home's power router. The purpose of this thesis is to describe the development of the command validation module for the Cybersecure Power Router (CSPR) being developed under the Cybersecurity Center for Secure Evolvable Energy Delivery Systems (SEEDS).

Packets sent over serial communication to CSPR are decoded to obtain a command that then must pass command validation to ensure the command is safe and the source is trustworthy. The command validation for CSPR consists of two phases: syntax validation and modelling validation. Syntax validation is performed by analyzing the structure of the command, whereas modelling validation checks to ensure that CSPR will not enter an unsafe state if the command is executed. The command validation for CSPR was initially prototyped in Python to verify the module met all project requirements. The module was then implemented in VHDL to be uploaded to a Field-Programmable Gate Array (FPGA) for simulation and testing according to project requirements. The research in this paper evaluates the effectiveness of using command validation for preventing malicious attacks on CSPR. Simulation results for both the Python and VHDL implementation are compared to assess the usefulness of prototyping hardware descriptive code using Python.

Acknowledgements

I would like to thank Dr. Alan Mantooth for connecting me with the SEEDS team involved with CSPR and for serving as my thesis advisor. A very special thank you goes out to Joe Moquin, who has guided me every step of the way during my first undergraduate research experience. Joe's leadership and passion made this an exciting and impactful learning experience for me. I would like to recognize Chris Farnell for his leadership to the CSPR team and for holding the team accountable for clear communication. Finally, Nicholas Blair and Sang Yun Kim, the other two undergraduates on the CSPR team, both played a valuable role in this project by sharing their experiences, brainstorming ideas, and troubleshooting software with me.

Table of Contents

1.0	Introduction	1
1.1	Motivation	1
1.2	Thesis Objective	2
1.3	Thesis Organization.....	2
2.0	Background	3
2.1	Potential Threats to Power Routers.....	3
2.2	Cybersecure Power Router (CSPR)	3
3.0	Command Validation Module Design	6
3.1	System Design.....	6
3.2	Python Implementation	9
3.3	VHDL Implementation	12
4.0	Results	14
4.1	Simulation Goals	14
4.2	Python Testing and Simulation Results	14
4.3	VHDL Testing and Simulation Results.....	18
5.0	Conclusions and Future Research	21
	References.....	22

List of Figures

Figure 1 – DSP controller board and power electronics board used for CSPR [4].	3
Figure 2 – CSPR power flow diagram.	4
Figure 3 – Diagram of Cybersecure Power Router as defended layers [4].	5
Figure 4 – CSPR network packet.	5
Figure 5 – CSPR state machine diagram.	6
Figure 6 – Command validation logic flow chart.	8
Figure 7 – Python code: <i>payload_validation</i> function.	10
Figure 8 – Python code: <i>syntax_validation</i> function.	10
Figure 9 – Python code: <i>modelling_validation</i> function.	10
Figure 10 – Python code: State class.	11
Figure 11 – Python code: StateMachine class.	11
Figure 12 – Python code: <i>test_payload_validation</i> function.	15
Figure 13 – Python simulation: <i>syntax_validation</i> test.	15
Figure 14 – Python simulation: <i>pv_current</i> test.	16
Figure 15 – Python simulation: <i>modelling_validation</i> pass test.	17
Figure 16 – Python simulation: <i>modelling_validation</i> fail test.	17
Figure 17 – VHDL simulation: reset test.	18
Figure 18 – VHDL simulation: syntax validation test.	18
Figure 19 – VHDL simulation: <i>pv_current</i> test.	19
Figure 20 – VHDL simulation: state machine test for standby, 26ns – 46ns.	20
Figure 21 – VHDL simulation: state machine test for standby, 46ns – 66ns.	20
Figure 22 – VHDL simulation: state machine test for standby, 66ns – 98ns.	20

List of Tables

Table 1 – CSPR States 7

Table 2 – Command Validation Inputs and Outputs 12

1.0 Introduction

1.1 Motivation

In recent years, the number of homes supplying their own power independently of the grid has been increasing rapidly. From 2012 to 2015 residential solar installed capacity increased by over 50% [1]. Traditionally, the electrical grid has been fed almost exclusively by large, centralized power plants. However, as renewable energy sources increase in availability, the grid is gradually transitioning toward a distributed generation model where electricity is generated by smaller facilities located closer to their load [2].

Photovoltaic (PV) systems are unable to generate power in the absence of sunlight. If homeowners wish to have energy on demand, they must utilize some form of energy storage system in conjunction with their solar panels. Rather than using high-capacity battery storage systems, many owners of residential PV systems choose to buy and sell power to the utility company through the grid. In modern homes, power electronics within energy management systems (EMSs) are used to optimize the power flow between the various electrical systems in a home, often with the intention generating maximum energy savings [3].

The electronics that manage power generation, storage, and distribution within a grid-connected PV system are packaged together into a particular type of EMS referred to in this paper as a power router. A power router is responsible for supervising the current, voltage, and frequency between a home's photovoltaics, batteries, converters, and grid and load connections. Because a power router manages such a large amount of energy, it is important that homeowners choose power routers that are robust and secure. If an unauthorized entity were to gain control of the power router, they could cause great damage to the grid and the home of installation. One way attackers may control an unsecure power router is by sending unauthorized commands to the

router via serial communication. By incorporating command validation in the security of a power router, the threat that malicious commands impose on the system can be minimized.

1.2 Thesis Objective

This thesis aims to describe the design and development of the command validation module for the Cybersecure Power Router (CSPR). Also covered is the design of the finite state machine (FSM) used for modeling validation for CSPR. The command validation was first modeled in Python before being implemented in VHDL, so a discussion about the use of Python for prototyping is included as well. The simulation results of the VHDL module are included at the end for proof of concept. These results are compared to the Python module simulations to demonstrate the effectiveness of using Python to prototype code before implementing a design using a lower level language such as VHDL.

1.3 Thesis Organization

This thesis is divided into five chapters total, beginning with the introduction in Chapter 1. Chapter 2 provides background about the role of security in energy management devices and how CSPR is being developed to combat the threat of security breaches in power routers. Chapter 3 focuses on the design and development of the command validation module for CSPR. Chapter 4 showcases the results from testing the Python and VHDL implementations of the command validation module. Chapter 5 summarizes the research and concludes the work performed for this thesis, while also discussing the potential for future research.

2.0 Background

2.1 Potential Threats to Power Routers

To better understand the importance of security, it is helpful to discuss the threats that may be faced by energy management devices such as power routers. Because a power router acts as the controller for the main components of a household distribution network, if a hacker were to gain unauthorized access to the router they could manipulate the flow of electricity between these components. Frequent switching of device relays could be used to short-circuit expensive electronics and potentially cause fires. Unauthorized manipulation of a power router could even be used to inject harmonics into the grid through denial-of-service (DoS) attacks [4].

2.2 Cybersecure Power Router (CSPR)

CSPR is being developed under the Cybersecurity Center for Secure Evolvable Energy Delivery Systems (SEEDS). SEEDS aims to address the lack of security in the power routers available on the market today by sponsoring the design of CSPR, which will provide cybersecurity for a grid-connected EMS. CSPR is based on the Smart Green Power Node (SGPN) that is researched in [5-7].

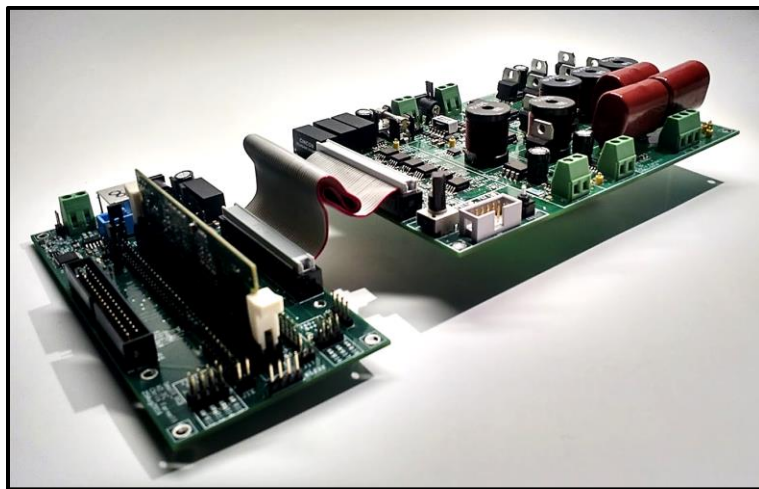


Figure 1 – DSP controller board and power electronics board used for CSPR [4].

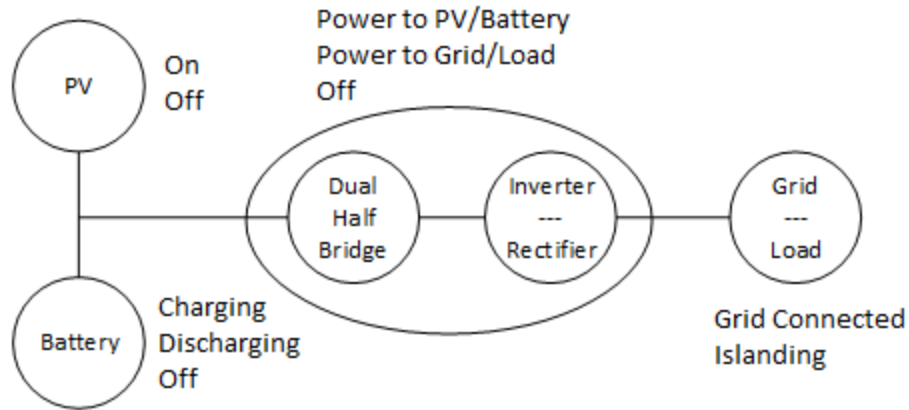


Figure 2 – CSPR power flow diagram.

CSPR manages the power flow between the solar panels, batteries, converters, grid, and load of the residential home. The state of these individual devices is defined by the direction of current flow through them, while the state of the power router is defined by the power flow through the system as a whole. The state and current flow between each of these devices must be coordinated depending on the current state of the power router. If an error were to occur that caused a conflicting state of power flow, equipment could get damaged. The power flow diagram in Fig. 2 shows the components managed by CSPR along with their corresponding states.

CSPR can be viewed as a collection of defended layers that work together to minimize the potential for malicious attacks. Each layer has its own security features while also being protected by the layers wrapping it. These layers all work together to ensure the confidentiality, integrity, and availability of the system. The work in this thesis focuses on the command layer, which is protected through validation of encrypted commands that are sent over the communication port.

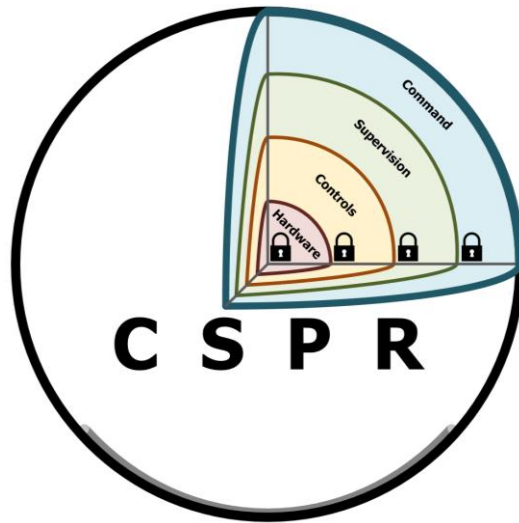


Figure 3 – Diagram of Cybersecure Power Router as defended layers [4].

CSPR is designed to receive network packets through serial communication. A single network packet consists of a frame header, an address, a payload, and a cyclic redundancy check (CRC). The frame header, address, and CRC are control data used to evaluate the destination and integrity of the message. The payload contains an encrypted command that carries information telling the power router what to do. Performing command validation on the decrypted payload adds a greater depth of security to the system, ensuring the command is a known command that will not cause the system to enter an unwanted state.

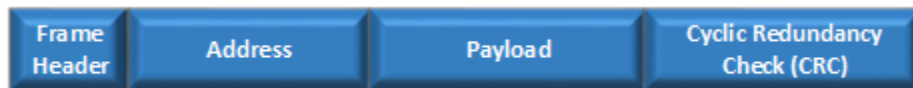


Figure 4 – CSPR network packet.

3.0 Command Validation Module Design

3.1 System Design

CSPR can be modeled by a finite state machine (FSM) that is based on the work proposed for the SGPN in [5]. Each state falls into one of two categories: islanding and grid-connected. Islanding states describe states where the relays between the grid and household are opened and therefore the house is disconnected from the grid. Grid-connected states, as the name suggests, are states where power may be exchanged between the grid and the house. The inputs to the FSM are the commands that are transmitted via serial communication, and the current being supplied by the home's solar panels. Fig. 5 shows an illustration of the state machine diagram. Note that Fig. 5 does not specify the inputs that would result in each state change, but instead demonstrates the state changes that are allowable.

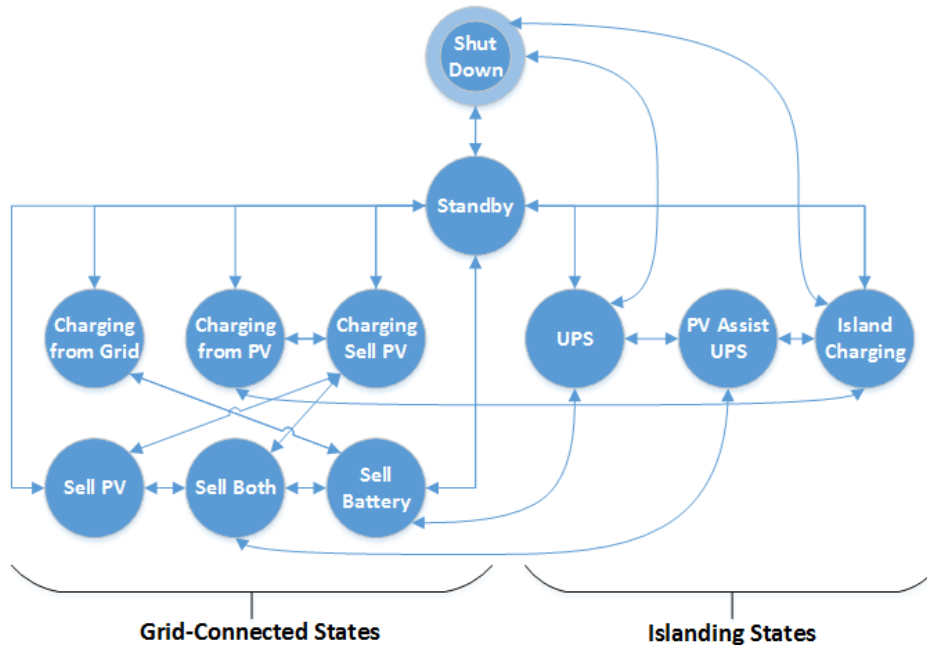


Figure 5 – CSPR state machine diagram.

Table 1 – CSPR States

Name	Bit Code	PV	Battery	Dual Half-Bridge	Grid Connected	Allowed State Changes
shut_down	0000	Off	Off	Off	No	standby, ups, island_charging
standby	0001	Off	Off	Off	Yes	shut_down, charging_from_grid, charging_from_pv, sell_pv, sell_battery, ups, island_charging
charging_from_grid	0010	Off	Charging	To PV/Battery	Yes	standby, sell_battery
charging_from_pv	0011	On	Charging	Off	Yes	standby, charging_sell_pv, island_charging
charging_sell_pv	0100	On	Charging	To Grid/Load	Yes	charging_from_pv, sell_pv, sell_both
sell_pv	0101	On	Off	To Grid/Load	Yes	standby, charging_sell_pv, sell_both
sell_both	0110	On	Discharging	To Grid/Load	Yes	charging_sell_pv, sell_pv, sell_battery, pv_assist_ups
sell_battery	0111	Off	Discharging	To Grid/Load	Yes	standby, charging_from_grid, ups
ups	1000	Off	Discharging	To Grid/Load	No	shut_down, standby, sell_battery, pv_assist_ups
pv_assist_ups	1001	On	Discharging	To Grid/Load	No	sell_both, ups, island_charging
island_charging	1010	On	Charging	Off	No	standby, charging_from_pv, pv_assist_ups

Detailed information about each state is given in Table 1. The name of each state as used in code is given along with the state’s corresponding bit code that is serving as a place holder for what might be transmitted through the payload of the serial communication. The state of each of the components shown in the power flow diagram of Fig. 2 is given as well. The PV system can either be on or off, and the battery can be charging, discharging, or off. The dual half-bridge converter can be pushing power to the grid/load or to the PV/battery system. Also, the system can either be grid connected or islanding. Each of the allowed state changes is given for every state, which can also be inferred from the state machine diagram of Fig. 5. Note that *uninterruptable power supply* mode is abbreviated to ups in Fig. 5 and Table 1.

The command validation consists of two phases: syntax validation and modeling validation. Syntax validation is performed by comparing the command to a list of acceptable commands stored in memory. If the command is contained in the list of acceptable commands, then the command passes syntax validation. Modeling validation is performed through the use of the state machine. If the state change suggested by the command is acceptable according to the state machine, then the command passes modeling validation. If either syntax validation or modeling validation fails, the command is rejected. If both syntax validation and modeling validation pass, the state machine is updated to reflect the state change carried by the command.



Figure 6 – Command validation logic flow chart.

3.2 Python Implementation

Beginning with a Python model before moving on to the final VHDL implementation was ideal because Python is a comparatively user-friendly language to work with. Python code is very clear and concise compared to VHDL, making it easy to share and review with others. The data structures available in Python are a great resource when modeling complex systems. Further, testing Python code is very simple and fast when compared to simulating VHDL modules. Consequently, verifying that the command validation model was designed correctly and met all requirements was made much easier by utilizing Python.

The Python code is split into two files: *payloadvalidation.py* and *statemachine.py*. The file *payloadvalidation.py* contains the function *payload_validation*, which accepts a payload and a photovoltaic (PV) current as inputs. The function *payload_validation* calls *syntax_validation*, a function that compares the payload to a dictionary containing each of the acceptable commands and the state change associated with each command. If *syntax_validation* returns false, the payload validation returns false and prints a relevant message stating why validation failed. If *syntax_validation* returns true, *modelling_validation* is called. If *modelling_validation* returns true, then *payload_validation* returns true to signal to the system that the command is authentic and safe. If either *syntax_validation* or *modelling_validation* return false, then a relevant message stating why the validation failed is printed to the console and *payload_validation* returns false.


```

6 def payload_validation(payload, pv_current):
7     """pv_current measured in mA."""
8     # Run syntax check on payload. Return false if fails.
9     if not syntax_validation(payload):
10        print("Payload validation failed.\n")
11        return False
12    # Run modelling check on payload. Return false if fails.
13    if not modelling_validation(payload, pv_current):
14        print("Payload validation failed.\n")
15        return False
16    # Return true if syntax and modelling check pass.
17    print("Payload \"" + payload + "\" is valid.\n")
18    return True

```

Figure 7 – Python code: *payload_validation* function.

```

21 def syntax_validation(payload):
22     if payload in _commands:
23         return True
24     else:
25         print("SYNTAX ERROR: Payload \"" + payload + "\" not in dictionary of accepted commands. ")
26         return False

```

Figure 8 – Python code: *syntax_validation* function.

```

26 def modelling_validation(payload, pv_current):
27     # Verify that payload is in command list and pv current is sensed for pv-connected states.
28     if ((_commands[payload] in cspr.current_state.allowed_state_change) and
29         ((pv_current > 0) == _commands[payload].pv_out)):
30         return True
31     else:
32         print("Illegal change of state.")

```

Figure 9 – Python code: *modelling_validation* function.

The backbone of the *modelling_validation* function is the *statemachine.py* file that contains the definition of the *State* and *StateMachine* classes. The *State* class contains properties inherent to each state of CSPR. A group of properties are used to define whether the direction of current flow is *in* or *out* for the PVs, batteries, and grid. Each of these properties accept a Boolean value to designate whether *current in* is true/false or *current out* is true/false. If the component is disconnected for the particular state being defined, then both *current in* and *current out* receive false values. In no case may both *current in* and *current out* receive the value true.

Another important property is a list of each allowable state change for the state being defined.

The *StateMachine* class contains definitions for each state of CSPR.

```
4 class State:
5     def __init__(self, name, pv_out, battery_out, battery_in, grid_out, grid_in, allowed_state_change):
6         self.name = name
7         self.pv_out = pv_out
8         self.battery_out = battery_out
9         self.battery_in = battery_in
10        self.grid_out = grid_out
11        self.grid_in = grid_in
12        self.allowed_state_change = allowed_state_change
```

Figure 10 – Python code: State class.

```
15 class StateMachine:
16     def __init__(self):
17         self._current_state = self.shut_down
18
19     @property
20     def current_state(self):
21         return self._current_state
22
23     @current_state.setter
24     def current_state(self, state):
25         if state in self._current_state.allowed_state_change:
26             self._current_state = state
27
28     # Define all possible states.
29     shut_down = State("shut_down", False, False, False, False, False, None)
30     standby = State("standby", False, False, False, True, False, None)
31     charging_from_grid = State("charging_from_grid", False, False, True, False, None)
32     charging_from_pv = State("charging_from_pv", True, False, True, True, False, None)
33     charging_sell_pv = State("charging_sell_pv", True, False, True, False, True, None)
34     sell_pv = State("sell_pv", True, False, False, False, True, None)
35     sell_both = State("sell_both", True, True, False, False, True, None)
36     sell_battery = State("sell_battery", False, True, False, False, True, None)
37     ups = State("ups", False, True, False, False, False, None)
38     pv_assist_ups = State("pv_assist_ups", True, True, False, False, False, None)
39     island_charging = State("island_charging", True, False, True, False, False, None)
40
41     # Define all allowable state changes.
42     shut_down.allowed_state_change = [standby, ups, island_charging]
43     standby.allowed_state_change = [shut_down, charging_from_grid, charging_from_pv, sell_pv, sell_battery, ups,
44                                     island_charging]
45     charging_from_grid.allowed_state_change = [standby, sell_battery]
46     charging_from_pv.allowed_state_change = [standby, charging_sell_pv, island_charging]
47     charging_sell_pv.allowed_state_change = [standby, charging_from_pv, sell_pv, sell_both]
48     sell_pv.allowed_state_change = [standby, charging_sell_pv, sell_both]
49     sell_both.allowed_state_change = [charging_sell_pv, sell_pv, sell_battery, pv_assist_ups]
50     sell_battery.allowed_state_change = [standby, charging_from_grid, ups]
51     ups.allowed_state_change = [shut_down, standby, sell_battery, pv_assist_ups]
52     pv_assist_ups.allowed_state_change = [sell_both, ups, island_charging]
53     island_charging.allowed_state_change = [standby, charging_from_pv, pv_assist_ups]
```

Figure 11 – Python code: StateMachine class.

3.3 VHDL Implementation

To make the transition from Python to VHDL, the set of inputs, processes, and outputs must be abstracted from the previous implementation. By modeling the command validation module in Python to begin with, all the inputs and outputs of the system had been identified and the basic logic had been determined. A 4-bit payload containing a command is input to the system along with a measurement of the current being produced by the PV system expressed in mA by a 16-bit vector. Single bit clock and reset inputs were added to the input list since a state machine is a synchronous system. The outputs are active high single bit signals used to express the type of error and whether the payload was valid or not. The system inputs and outputs are listed in Table 2.

Table 2 – Command Validation VHDL Implementation Inputs and Outputs

Inputs	Outputs
clock	syntax_error
reset	modelling_error
payload	payload_valid
pv_current	

The architecture of the VHDL code contains the *syntax_validation* and *modelling_validation* processes, as well as the asynchronous logic for determining the value of *payload_valid*. Designing the command validation to use two separate processes enables a multi-core FPGA to take advantage of parallel processing to decrease the latency of the command validation. Because *payload_valid* is determined by combinational logic that depends on whether a syntax or modeling error occurred and because VHDL does not support reading from outputs, two signals named *syntax_error_buffer* and *modelling_error_buffer* are used to temporarily store whether a syntax or modeling error occurred. In turn, the outputs *syntax_error* and *modelling_error* are driven by their temporary counterparts. The *syntax_validation* process uses *if* statements to compare the current payload input to all expected payload values. If the payload doesn't match any of the bit-strings coded in the *if* statements, *syntax_error_buffer* is set to '1'.

The *modelling_validation* process makes use of the state machine logic to control the value of *modelling_error_buffer*. The user-defined type *state_type* is used to define the signal *current_state* who can only take on a value that is the name of one of the states that the FSM may be in. The *modelling_validation* process contains hard-coded logic that uses nested *case* and *if* statements to determine the value that *modelling_error_buffer* will receive based on the value of the *current_state* signal, the *payload* port, and the *pv_current* port. If the modeling validation passes, then *modelling_error_buffer* is initialized to '0' and the value of *current_state* is updated to reflect the state change contained in the command. The temporary signals *syntax_error_buffer* and *modelling_error_buffer* are used as inputs to a *nor* gate whose output is *payload_valid*. This way, if either *syntax_error_buffer* or *modelling_error_buffer* is high, *payload_valid* goes low.

4.0 Results

4.1 Simulation Goals

When simulating the command validation module, the goal was to ensure that commands that belong to the list of expected commands that would not result in an illegal state change were confirmed valid by the code. Commands that do not belong to the list of expected commands should result in a syntax error. Commands that would result in an illegal state change should result in a modelling error. An illegal state change is characterized by trying to make a state change that is not supported by the state machine diagram or trying to change to a PV dependent state when *pv_current* is not greater than zero.

The goal in comparing the Python and VHDL simulation results is to show that both implementations of the command validation module produce the same results. The results of the Python code were verified before the VHDL implementation was started. Because of this, if the VHDL simulation results match the Python simulation results, then the VHDL code successfully implements the command validation module that was described in Chapter 3.

4.2 Python Testing and Simulation Results

To test the Python implementation of the command validation module, the function *test_payload_validation* (shown in Fig. 12) was created. This function accepts a string for *payload* and an integer for *pv_current*. The current state of the system is printed along with the payload that was input to the function, then the function makes a call to *payload_validation* using the *payload* and *pv_current* that were input. Finally, the function updates the current state of the system.

```

44 def test_payload_validation(payload, pv_current):
45     print("Current State: " + cspr.current_state.name + "\nCurrent Payload: " + payload)
46     if payload_validation(payload, pv_current):
47         cspr.current_state = _commands[payload]

```

Figure 12 – Python code: *test_payload_validation* function.

```

50     # Payload validation should pass since command is valid and state change is allowed.
51     test_payload_validation("standby", 0)
52
53     # Payload validation should fail since command is invalid.
54     test_payload_validation("invalid_command", 0)

```

if __name__ == "__main__"

Run payloadvalidation

```

Current State: shut_down
Current Payload: standby
Payload "standby" is valid.

Current State: standby
Current Payload: invalid_command
SYNTAX ERROR: Payload "invalid_command" not in dictionary of accepted commands.
Payload validation failed.

```

Figure 13 – Python simulation: *syntax_validation* test.

To begin, the proper operation of the *syntax_validation* function was tested. First, the test function was called using the payload “standby”, then once again using the payload “invalid_payload”. As expected, “standby” was accepted and “invalid_payload” resulted in a syntax error, as shown in Fig. 13.

To test the influence of the PV current on the modelling validation, the *test_payload_validation* function was called with the command “sell_pv” and a current of zero. As expected, this resulted in a modelling error because *sell_pv* requires that *pv_current* is greater than zero. Once again, *test_payload_validation* was called with the command “sell_pv”, but this time with a current of ‘1’. This time, the command was accepted, as shown in Fig. 14.

```
57 # Payload validation should fail since pv_current is not greater than zero.
58 test_payload_validation("sell_pv", 0)
59
60 # Payload validation should pass.
61 test_payload_validation("sell_pv", 1)
```

Run payloadvalidation

```
Current State: standby
Current Payload: sell_pv
MODELLING ERROR: Payload "sell_pv" would result in illegal change of state.
Payload validation failed.

Current State: standby
Current Payload: sell_pv
Payload "sell_pv" is valid.
```

Figure 14 – Python simulation: *pv_current* test.

Lastly, the *modelling_validation* function was tested to ensure that it allowed changes to states that were in the current state's *allowed_state_change* list and rejected changes to states that were not. The list of each state's permissible state changes is shown in Table 1 of Chapter 3. The modelling validation was tested for the state *sell_pv*, which should only allow state changes to *standby*, *charging_sell_pv*, and *sell_both*. For test purposes, each time the payload was validated and *current_state* was changed, *current_state* was then manually overridden back to *sell_pv* again. Fig. 15 shows that modelling validation passed for the three state changes that were expected to pass, and Fig. 16 shows that modelling validation failed for the other attempted state changes.

```

72     # Payload validation should pass.
73     test_payload_validation("standby", 0)
74     cspr.current_state = _commands["sell_pv"]
75
76     # Payload validation should pass.
77     test_payload_validation("charging_sell_pv", 1)
78     cspr.current_state = _commands["sell_pv"]
79
80     # Payload validation should pass.
81     test_payload_validation("sell_both", 1)
82     cspr.current_state = _commands["sell_pv"]

```

Run payloadvalidation

```

▶ ↑ Current State: sell_pv
■ ↓ Current Payload: standby
  Payload "standby" is valid.
⏸ ⏪ ⏩
⏮ ⏭ ⏯
⏲ ⏱ ⏴ ⏵
✖ ? Current State: sell_pv
  Current Payload: charging_sell_pv
  Payload "charging_sell_pv" is valid.
  Current State: sell_pv
  Current Payload: sell_both
  Payload "sell_both" is valid.

```

Figure 15 – Python simulation: *modelling_validation* pass test.

```

63     # Payload validation should fail since we can't change states from 'sell_pv' to 'shut_down'.
64     test_payload_validation("shut_down", 0)
65
66     # Payload validation should fail since we can't change states from 'sell_pv' to 'charging_from_grid'.
67     test_payload_validation("charging_from_grid", 0)
68
69     # Payload validation should fail since we can't change states from 'sell_pv' to 'charging_from_pv'.
70     test_payload_validation("charging_from_pv", 1)

```

Run payloadvalidation

```

▶ ↑ Current State: sell_pv
■ ↓ Current Payload: shut_down
  MODELLING ERROR: Payload "shut_down" would result in illegal change of state.
  Payload validation failed.
⏸ ⏪ ⏩
⏮ ⏭ ⏯
⏲ ⏱ ⏴ ⏵
✖ ? Current State: sell_pv
  Current Payload: charging_from_grid
  MODELLING ERROR: Payload "charging_from_grid" would result in illegal change of state.
  Payload validation failed.
  Current State: sell_pv
  Current Payload: charging_from_pv
  MODELLING ERROR: Payload "charging_from_pv" would result in illegal change of state.
  Payload validation failed.

```

Figure 16 – Python simulation: *modelling_validation* fail test.

4.3 VHDL Testing and Simulation Results

A testbench was created to test the VHDL command validation module's operation. To begin with, the *reset* input is tested by moving the current state into *standby* then raising *reset* to '1'. The expected result is that *current_state* will go to *shut_down*, *syntax_error* and *modelling_error* will go to '1', and, consequently, *payload_valid* will go to '0'. The waveforms in Fig. 17 over the time period of 6ns to 10ns confirms that the expected results were achieved.

The *syntax_validation* process was tested after this to be sure that any unexpected commands cause *syntax_error* to go to '1' and *payload_valid* to go to '0'. The results of this test are shown in Fig. 18, where the payload "1111" causes *syntax_error* to go to '1' from 14ns to 18ns.

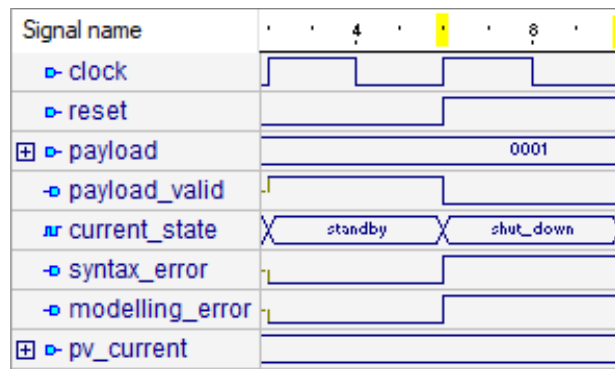


Figure 17 – VHDL simulation: reset test.

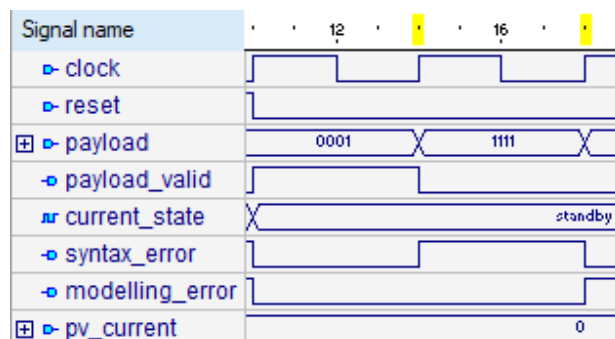


Figure 18 – VHDL simulation: syntax validation test.

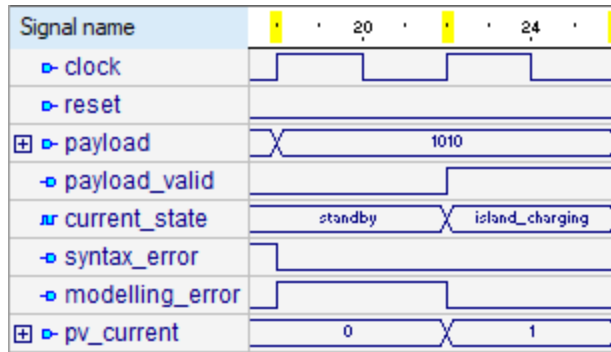


Figure 19 – VHDL simulation: *pv_current* test.

Next, the system was tested to make sure that states that are PV enabled could not be moved in to if *pv_current* was not greater than zero. If a payload commands the system to move in to a PV enabled state and *pv_current* is not greater than zero, then *modelling_error* should go to ‘1’, *payload_valid* should go to ‘0’, and *current_state* should remain the same. This is demonstrated in Fig. 19. From 18ns to 22ns, *pv_current* is 0 so the payload of “1010” that should result in a state change to *island_charging* fails. For the next clock cycle, shown from 22ns to 26ns, *pv_current* is set to ‘1’ so the payload “1010” results in a successful state change to *island_charging*.

For the next test, the *modelling_validation* process is tested for the *standby* state. To accomplish this, the system is moved into *standby* and the effect of each command is tested while the system is in *standby*. If an illegal state change is attempted, *modelling_error* should go to ‘1’, *payload_valid* should go to ‘0’, and *current_state* should remain *standby*. If the payload commands the system to make a legal state change, *payload_valid* goes to ‘1’ and *current_state* changes to reflect the command carried by the payload. Note that *current_state* is changed back to *standby* after each successful state change. All of this is demonstrated in the waveforms in Figs. 20-22. This test shows that the state machine is correctly implemented for *standby* state.

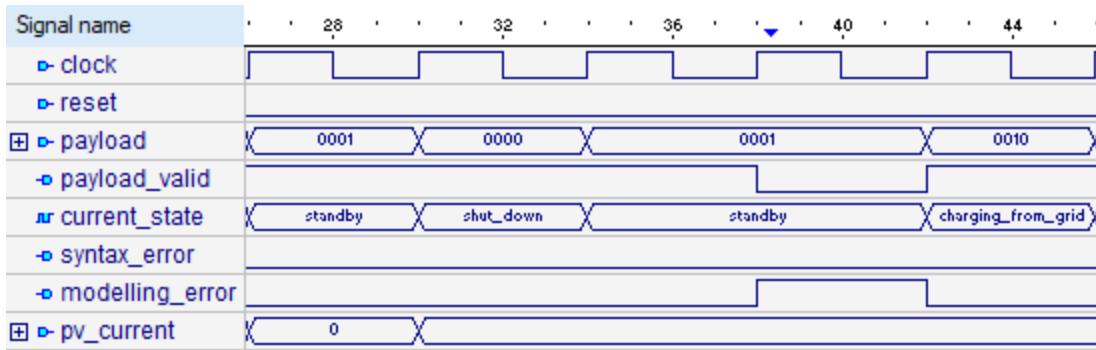


Figure 20 – VHDL simulation: state machine test for standby, 26ns – 46ns.

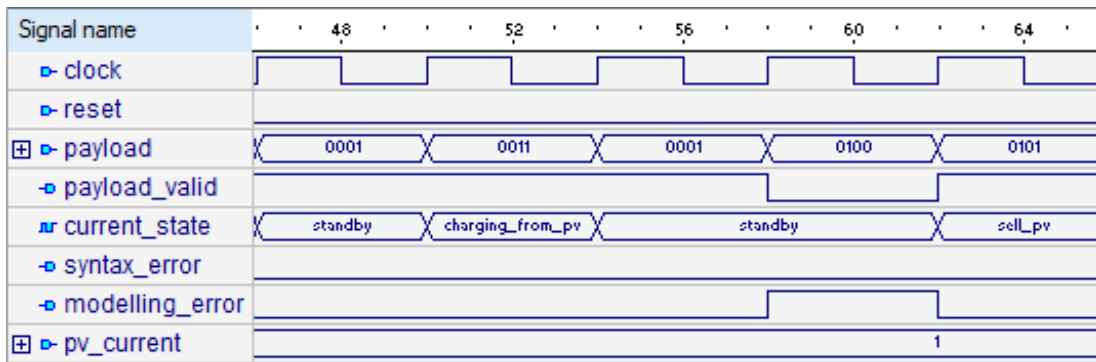


Figure 21 – VHDL simulation: state machine test for standby, 46ns – 66ns.

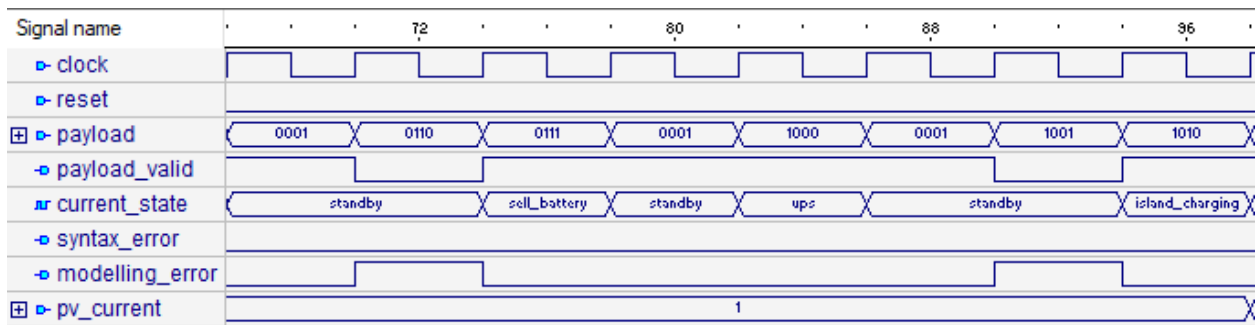


Figure 22 – VHDL simulation: state machine test for standby, 66ns – 98ns.

The results of the VHDL simulations show that the system yields the same results as the Python model given the same inputs, as discussed in Chapter 3. Further, the VHDL command validation module behaves just as the Python module in terms of rejecting payloads that do not pass both syntax and modelling validation.

5.0 Conclusions and Future Research

The simulation results of section 4.2 demonstrate that the Python code appropriately models the command validation module and state machine logic for C SPR. The payload input to the system was rejected in all the test cases where rejection was expected due to failing either the syntax or modeling validation. Python's simplicity and readability made it the ideal language for modeling the command validation logic before moving on to the more complicated VHDL implementation. The simulation results of section 4.3 demonstrate that the VHDL code correctly synthesized the command validation module that had been modeled in Python.

The next step in this project is to upload the command validation module to the Lattice MachXO2 FPGA for testing. This will allow for accurate timing tests to be performed on the command validation module to verify whether the code is too slow to be used in a production model of C SPR. Many problems can arise when moving VHDL code over to an FPGA, but the real-world testing of the command validation module would be the best way to demonstrate proof-of-concept. The command validation module may need to be tweaked as C SPR's state machine is revised, but the current VHDL implementation provides a solid framework for the C SPR group to work with in the future.

References

- [1] SEIA, “Solar Industry Research Data”, 2017. [Online]. Available: <https://www.seia.org/solar-industry-research-data>. [Accessed: April 24, 2018].
- [2] Zhao, B. Wang, C. Zhang, Xuesong, “Grid-Integrated and Standalone Photovoltaic Distributed Generation Systems – Analysis Design and Control”. John Wiley and Sons, 2018.
- [3] S. Lisauskas, R. Kline, “Energy Management Systems: Maximizing Energy Savings (Text Version)”. Office of Energy Efficiency & Renewable Energy. [Online]. Available: <https://www.energy.gov/eere/wipo/energy-management-systems-maximizing-energy-savings-text-version>. [Accessed: April 24, 2018].
- [4] H. A. Mantooth, J. Moquin, “Cybersecure Power Router (CSPR)”. SEEDS Proposal Presentation, 2018. [Powerpoint].
- [5] S. Zhao, J. Umuhoza, Y. Zhang, J. Moquin, C. Farnell and H. A. Mantooth, "Analysis and optimization of a high-efficiency residential energy harvesting system with dual half-bridge converter," *2017 IEEE Applied Power Electronics Conference and Exposition (APEC)*, Tampa, FL, 2017, pp. 2838-2844.
- [6] Y. Zhang, J. Umuhoza, Y. Liu, C. Farnell, H. A. Mantooth, R. Dougal, “Optimized control of isolated residential power router for photovoltaic applications,” *IEEE Energy Conversion Congress and Exposition (ECCE)*, pp. 53-59, Sept. 2014.
- [7] Y. Zhang, J. Umuhoza, H. Liu, F. Hossain, C. Farnell, H. A. Mantooth, “Realizing an integrated system for residential energy harvesting and management,” *IEEE Applied Power Electronics Conference (APEC)*, pp. 3240-3244, March 2015.