Implementing Symbolic Controllers into FPGAs

Antonio J. Rueda

Master of Science Thesis

TU Delft Delft University of Technology

Delft Center for Systems and Control

Implementing Symbolic Controllers into FPGAs

Master of Science Thesis

For the degree of Master of Science in Embedded Systems at Delft University of Technology

Antonio J. Rueda

August 26, 2019

Faculty of Mechanical, Maritime and Materials Engineering (3mE) · Delft University of Technology

Delft University of Technology
Department of
Delft Center for Systems and Control (dcsc)

The undersigned hereby certify that they have read and recommend to the Faculty of Mechanical, Maritime and Materials Engineering (3mE) for acceptance a thesis entitled

Implementing Symbolic Controllers into FPGAs

by

Antonio J. Rueda

in partial fulfillment of the requirements for the degree of

Master of Science Embedded Systems

Dated: August 26, 2019

Supervisor(s):

Dr. Manuel Mazo Espinosa

Reader(s):

Dr. Wei Pan

Dr. Manon Kok

Abstract

Embedded control systems are processor-based systems that need to run an application for an extended amount of time, such as months or years. Typically, they implement a real-time function to control a system. Embedded systems are implemented using hardware and software to perform an specific task. This is why they can be optimized to reduce its size and cost and increase its reliability and performance. In embedded control systems, a discrete time embedded system is controlling a continuous time plant. In order to deal with this complex interactions, there are some tools that synthesize symbolic controllers. However, the size of these controllers is still too large to be widely implemented in embedded systems for real-time applications. Although it is possible to implement them in CPUs with large memory, their time-step is limited by a few GHz. On the other hand, FPGAs can run at a higher frequency (MHz) but they have limited memory. In this project, we propose a tool that automate the process of compressing, determinizing and generating the necessary files to flash a symbolic controller into an FPGA. We propose three different ways of transforming the original controllers and we compare them with another similar tool from the Technische Universität München. We also simulate in real-time the controlled closed-loop of some of those symbolic controllers using a simulated plant to validate the entire process.

Table of Contents

List of Figures

List of Tables

Acknowledgements

First and foremost, I would like to express my gratitude to my supervisor Dr. Manuel Mazo Espinosa for giving me the chance of doing the master's thesis under the control department, for his advise and for his guidance during this project. I learned a lot during his weekly meetings with all the other supervised students.

I would also like to thank Dr. Wei Pan and Dr. Carlas Smith for being able to take part in the thesis committee.

Thanks to all my family and friends for their unconditional love and support.

Last, I would also like to thank a very special friend for visiting me from Spain every now and then. Her random visits during this last year made me not lose my mind.

Delft, University of Technology
August 26, 2019

Antonio J. Rueda

"Life is like riding a bicycle. To keep your balance, you must keep moving."
— *Albert Einstein*

Chapter 1

Introduction

1-1 Background

Embedded control systems are processor-based systems that need to run an application for an extended amount of time, such as months or years. Typically, they implement a real-time function to control a system. Embedded systems are implemented using hardware and software to perform an specific task. This is why they can be optimized to reduce its size and cost and increase its reliability and performance. In embedded control systems, a discrete time embedded system is controlling a continuous time plant. These interactions between discrete systems and continuous systems are studied in the hybrid systems field[1].

For the design and synthesis of controllers, there are some tools based on symbolic models as PESSOA, SCOTS, CoSyMA, LTLMoP, TuLiP, see [2], [3], [4], [5], and [6] respectively. They allow the synthesis of correct-by-design controllers of nonlinear systems. The main issue of these techniques is the big size of the generated controller. A typical controller generated by these tools is stored in the so-called Binary Decision Diagram (BDD). BDDs are huge tables mapping the state space grid with the corresponding valid input space grid.

There exists determinization techniques to reduce the size of the controllers in order to be able to implement them in embedded systems. In [7], they propose 2 new techniques to determinize controllers: Global Algorithm (GA) and Symbolic Regression (SR), and they compare these new determinization algorithms to one existing technique called Local Algorithm (LA) proposed by [8]. We will be making use of this techniques to reduce the size of our controller and make it optimal.

In this thesis project, we will be using a Field Programmable Gate Array (FPGA) to run a correct-by-design generated controller. FPGAs are silicon chips with unconnected logic gates. The functionality of the FPGA can be defined by using software to configure the gates. The benefits of using an FPGA are:

- Flexibility: Because they are reconfigurable, the controller can be updated at any time.

- Performance: They use hardware parallelism. Their computing power is greater than in a Digital Signal Processor (DSP).

- Reliability: Since an FPGA is hardware-based, they don't use operating systems and they use true parallel execution and deterministic hardware for every task.

- High-speed: Using a top level clock, they can operate at rates such us $40MHz$, $80MHz$ or $200MHz$.

In order to flash any code into an FPGA, we need to follow the next steps:

1. Synthesis: The synthesis takes a design and creates a netlist. A netlist is a textual description of the gate connections in the FPGA.

2. Place and route: In this process, the elements of the netlist are physically placed and mapped to the FPGA physical resources.

3. Flashing: Once the synthesis and the place and route are done, we can flash the FPGA.

There are some tools that perform the process of generating code for FPGAs or for real-time systems taking as input the BDD of a controller. One of them was created by the Technische Universität München (TUM) [9]. This tool is able to create the following code for symbolic controllers:

- Hardware: Verilog and VHDL.

- Software: C and C++ boolean-valued functions.

The procedure that this tool follows includes determinization of the controller, conversion of the multi-output BDD to multi single-output BDDs and generation of the code. However, the process is not optimal since in order to generate the code, it transforms the complete BDD to a particular programming language by brute-force. Also, this tool uses a random determinization algorithm which is not optimal. Thus ,we believe this process can be optimized further, gaining performance and reducing the final size of the controller.

1-2 Goals

The goal of this master's thesis is to test and validate symbolic controllers using an FPGA:

- Modifying existing symbolic controllers in order to be burnt into an FPGA.

- Simulating a closed-loop controlled plant using 2 real-time systems.

1-3 Outline

In Chapter 1, we introduce an overview of symbolic controllers for embedded control systems and we take a look at the current tools to implement them. Chapter 2 discusses the basics and the tools to understand how a symbolic controller can be prepared for an FPGA. In Chapter 3 we take a deeper look at the tool that transform controllers generated from Scots. Chapter 4 explains how a controller can be implemented in an FPGA using LabVIEW instead of VHDL or verilog. In Chapter 5 we describe a procedure to simulate a plant in order to test and validate the closed-loop. Chapter 6 summarizes the findings and compares our results with the ones from the TUM tool. Finally, the conclusions and future work are discussed in Chapter 7.

Chapter 2

Preliminaries

This chapter introduces the basics needed to understand the process of the tool. First, we explain the concepts of symbolic controllers and binary decision diagrams. After that, we introduce the tools that have been used in the project.

2-1 Symbolic Controllers

In symbolic control, first, physical systems are approximated using finite abstractions and then, discrete controllers are synthesized for those abstractions [10]. Finally, the controllers are refined to hybrid ones. This approach tries to deal with complex systems interactions between continuous and discrete dynamics on embedded controller systems. This three step approach to create symbolic controllers is represented in the next figure (from [11]):

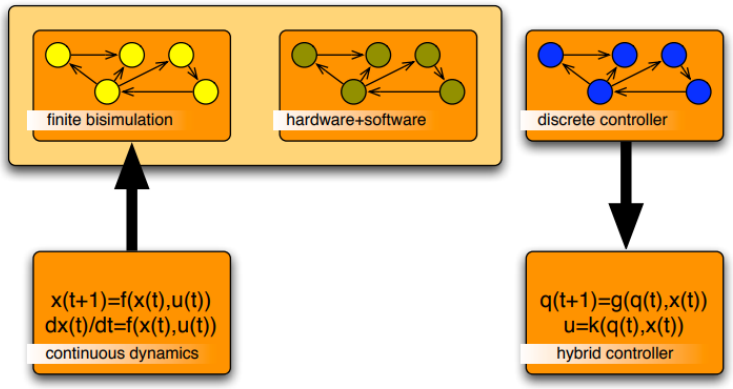


Figure 2-1: Three step approach to create symbolic controllers

In this thesis project, we will be using a tool (Scots) that creates symbolic controllers via a feedback refinement relation [12]. Later on, in section 2-4-1 we will describe the possibilities that this tool offers to synthesize symbolic controllers.

2-2 Binary Decision Diagrams

A binary decision diagrams (BDD) is a representation of a boolean function. It consists of:

- Nodes: They represent the variables of the boolean function. Each node has two child nodes.
- Leaves: They can be represented as dashed lines if the node is evaluated with the value of 0, or as continuous lines if the node is evaluated with the value of 1.
- Terminal nodes: They can be 0 or 1 and they will be the output of the boolean function.

Figure 2-2 shows an OR gate represented by a BDD:

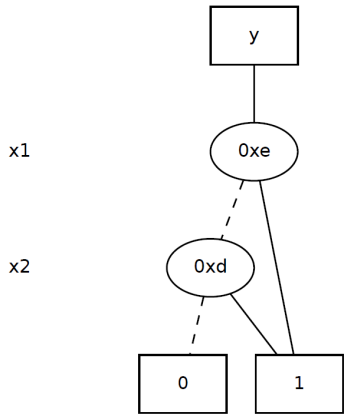


Figure 2-2: OR Gate represented by a BDD

x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	1

Table 2-1: OR gate truth table

2-3 Hardware

2-3-1 myRIO

MyRIO (figure 2-3) is an embedded device from National Instruments with one reconfigurable I/O device. It includes a dual-core ARM Cortex-A9 processor and a Xilinx FPGA along with other extra functionality.

Figure 2-3: myRIO-1900

In this project, we will make use of 2 myRIOs:

1. One myRIO to run the final controller.
2. Another to simulate a plant since we don't have a physical plant to test the controller.

The next figure (figure 2-4) shows the basic interconnection that we will be using in this project:

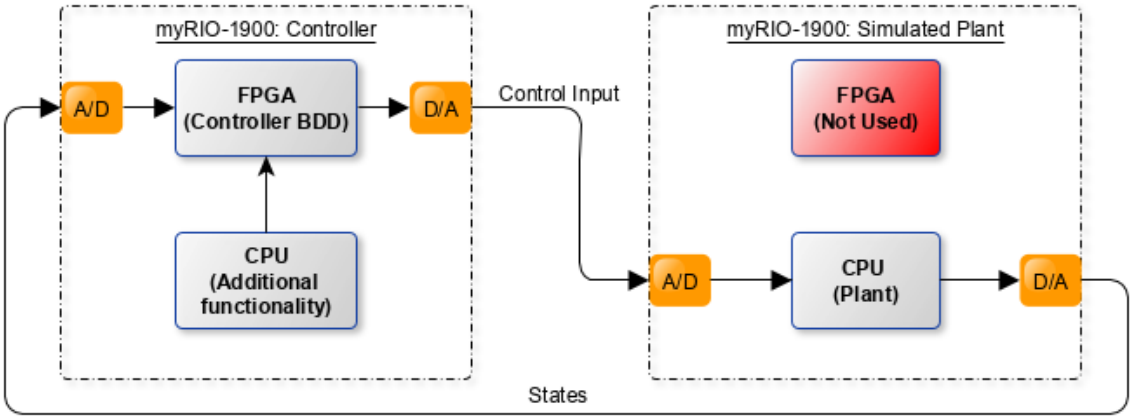


Figure 2-4: Interconnection of the 2 myRIO boards

As we can see, we will implement the BDD of the controller in the FPGA of the left board and its CPU will be used in particular cases (refer to section 5-3-2 for more details about the

use of the CPU). Finally, the right hand side board will only use its CPU to simulate the plant of the system.

We need to mention that the final controller will run in the FPGA since the clock frequency is at a much higher rate (up to $80MHz$) than the CPU, in which we will simulate the plant of a system.

2-4 Software

2-4-1 Scots v0.2

Scots [3] is an open source tool used to synthesize symbolic controllers. Scots is similar to Pessoa [2] and in order to create the controller, they represent the atomic propositions and the transition relations using boolean functions. Finally, Scots uses the Colorado University Decision Diagram (CUDD) library [13] to store and work with the boolean functions. [3] defines a *simple system* as a triple $S = (X, U, F)$, where the *state alphabet* X and *input alphabet* U are non-empty sets and the *transition function* $F : X \times U \rightrightarrows X$ is a set-valued map [14]. Then, given a simple system $S_1 = (X_1, U_1, F_1)$, let $X_1^\infty = \cup_{T\in\mathbb{N}\cup\{\infty\}} X_1^{[0;T[}$.

Scots has the next possibilities when defining the abstract specifications associated with $I_1, Z_1 \subseteq X_1$:

- Reachability: $\Sigma_1 = \left\{x_1 \in X_1^\infty | x_1(0) \in I_1 \Longrightarrow \exists_{t\in[0;T[} : x_1(t) \in Z_1\right\}$

  Then, trajectories will enter the target set Z in finite time and will remain within Z.

- Invariance: $\Sigma_1 = \left\{x_1 \in X_1^{[0;\infty[} | x_1(0) \in I_1 \Longrightarrow \forall_{t\in[0;\infty[} : x_1(t) \in Z_1\right\}$

  Then, trajectories starting in the target set Z will remain in Z.

Finally, Scots will store the controller in a BDD containing all possible pairs of state-input. Then, the input of the BDD will be a combination of state-input and the output of the BDD will be a boolean variable that will tell us if that combination is valid. A valid combination of state-input means that we are meeting the specifications defined when the controller was synthesized.

Example

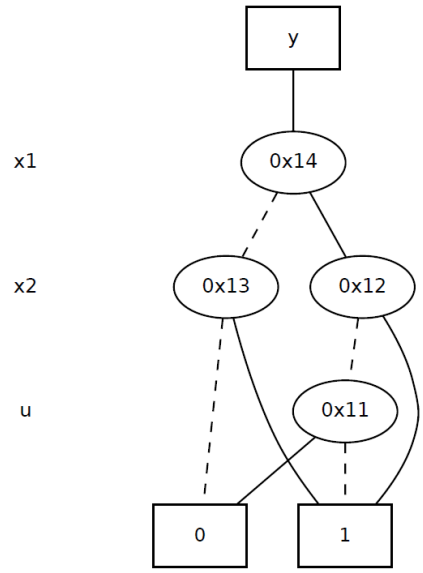


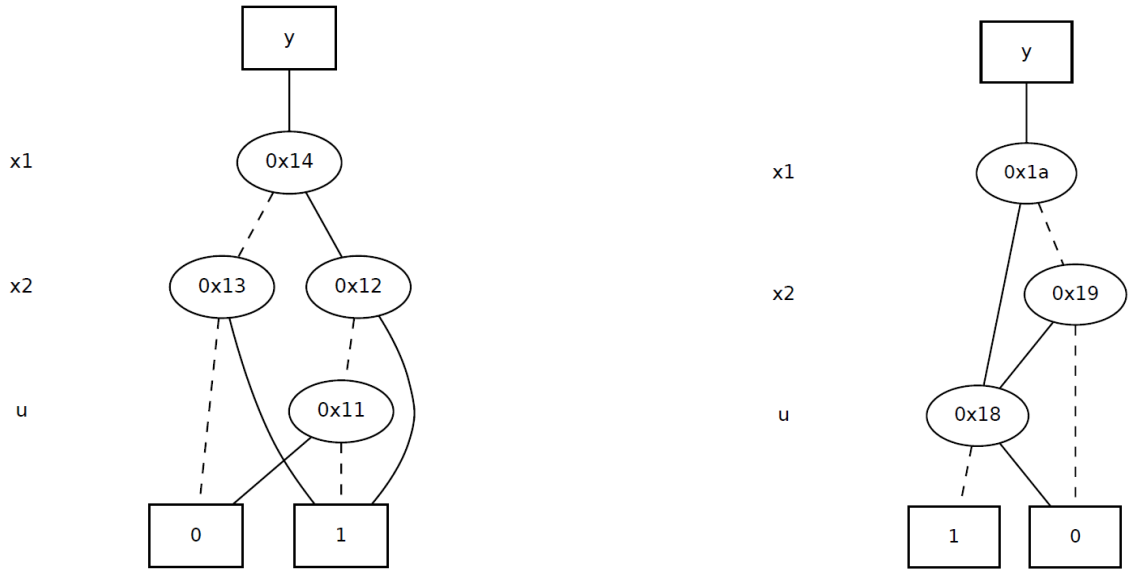
Figure 2-5: BDD of a simple controller synthesized by Scots

If we observe the previous figure (figure 2-5), the only possible combinations of state-input (x_1, x_2, u) needed to meet the controller specifications will be $(0,1,0)$, $(0,1,1)$, $(1,0,0)$, $(1,1,0)$ and $(1,1,1)$.

2-4-2 SCOTSv2.0 BDD controllers determinizer

The output of Scots will be a non-determinized controller. This means that for every particular state of a system, there will be more than one possible valid input. We will use the tool SCOTSv2.0 BDD controllers determinizer [15] to determinize the controller generated by Scots. Now there will be only one possible valid input. This will also help reducing the size of the controller.

Figure 2-6a shows a simple non-determinized controller and figure 2-6b is the output controller using this tool:



(a) Non-deterministic controller

(b) Deterministic controller

Figure 2-6: Determinizing a controller

x1	x2	u	y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

(a) Non-deterministic controller

x1	x2	u	y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

(b) deterministic controller

Table 2-2: Truth tables of a non-deterministic controller and its determinization

The highlighted values are the only ones that this tool has modified, and as we can observe, after determinizing the controller, there exists only one valid input for every different state.

2-4-3 SCOTSv2.0 BDD2Blif

Using the CUDD library, we have created a tool to modify the determinized controller and export it as `.blif` file. We will take a deep look at this tool in Chapter 3.

2-4-4 ABC: A System for Sequential Synthesis and Verification

ABC [16] is a software used for synthesis and verification of binary sequential logic circuits. In our case, it will be only used to convert the output of our tool (`.blif`) to verilog language.

2-4-5 VHDL Wrapper

Once we have the verilog file of the controller, we can synthesize the controller and generate the netlist using Vivado Design Suite[17]. Vivado is a software tool designed by Xilinx for syntheis and analysis of Hardware Description Language (HDL) designs. After that, it is necessary to create a VHDL wrapper that instantiates the component. We have build a simple text-based tool that makes this process easier. The tutorial [18] shows how to create this VHDL wrapper manually.

2-4-6 LabVIEW

LabVIEW [19] is a block programming environment for testing, measuring and controlling applications. One advantage of using LabVIEW is that we can program an FPGA without having knowledge of hardware programming languages such as VHDL or Verilog.

Once we have the netlist of the controller and the VHDL wrapper, we can import these files to LabVIEW and implement the controller. Please refer to Chapter 4 for a more detailed explanation.

2-4-7 Simulink

Simulink [20] is a block programming environment from Mathworks for designing and testing models. We will be using this tool to create the plant models that will run in one of the myRIOs CPU.

2-4-8 BDD2Implement (TUM)

We introduced this tool in Chapter 1 and the downsides of this tool are:

- It uses a random determinization algorithm.
- It convert the BDD to hardware or software code by brute force, outputting a large file.

Later on, we will test this tool in order to compare the results with ours.

2-5 Approach

Now that we have all the necessary tools, we can follow the next steps to obtain and run a controller on a myRio FPGA and simulate the controlled closed-loop:

1. Create a controller using Scots.
2. Determinize the controller using SCOTSv2.0 BDD controllers determinizer.
3. Split the controller with our tool SCOTSv2.0 BDD2Blif.
4. Convert the `.blif` files to verilog using ABC.
5. Generate the VHDL wrapper and synthesize the controller with Vivado.
6. Import the generated files of the controller to LabVIEW and flash it into the FPGA.
7. Simulate the plant in another myRIO.
8. Run the closed loop-simulation.

Chapter 3

Working with BDDs

This chapter explain the process behind the SCOTSv2.0 BDD2Blif tool. Since the output from Scots and also from SCOTSv2.0 BDD controllers determinizer is a BDD containing the valid combination of states-inputs, we need to transform this BDD to be able to input one particular state, and obtain as output the corresponding valid input.

3-1 First Approach: Transforming the BDD using Existential Abstractions

3-1-1 Intuitive implementation

We will use the next notation to express that a BDD contains a combination of states and inputs: $BDD(x, u)$.

Then we can split a BDD of a controller in 2 new BDDs according to the value of the input:

$$\exists u : BDD(x, 0)$$
$$\exists u : BDD(x, 1)$$

Example

We will use the next BDD controller as an example:

x_1	x_2	u	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Table 3-1: Truth table of a determinized controller

Then, using the CUDD library, we can apply the existential abstractions and we obtain:

x_1	x_2	y
0	0	0
0	1	0
1	0	1
1	1	0

(a) $\exists u$: $BDD(x, 0)$

x_1	x_2	y
0	0	1
0	1	1
1	0	0
1	1	1

(b) $\exists u$: $BDD(x, 1)$

Table 3-2: BDDs truth tables after splitting the original controller

Now it would be possible to flash both BDDs into the FPGA, and if the first one ($BDD(x, 0)$) evaluates to 1 we know that the plant needs 0 as an input or 1 if the second one ($BDD(x, 1)$) evaluates to 1. Since the original BDD was already determinized, both BDDs will not output 1 for the same state.

But we have considered only the case in which we have an input of only 1 bit. In case of having an input of more bits, the intuitive way of doing the same will be to existentially abstract every bit of the input. For example, if we have a controller with an input of 2 bits ($BDD(x, u_1, u_2)$), we will obtain 4 different BDDs:

$$\exists u_1 : BDD(x, 0, u_2)$$
$$\exists u_1 : BDD(x, 1, u_2)$$
$$\exists u_2 : BDD(x, u_1, 0)$$
$$\exists u_2 : BDD(x, u_1, 1)$$

And after that, apply another existential abstraction to remove the input bit that we are not considering:

$$\exists u_1 : BDD(x, 0)$$
$$\exists u_1 : BDD(x, 1)$$
$$\exists u_2 : BDD(x, 0)$$
$$\exists u_2 : BDD(x, 1)$$

Example

Now we will consider the next example (table 3-3):

x	u_1	u_2	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Table 3-3: Truth table of a determinized controller with an input of 2 bits

Applying the first set of existential abstractions we will obtain:

x	u_2	y
0	0	0
0	1	1
1	0	0
1	1	0

(a) $\exists u_1 : BDD(x, 0, u_2)$

x	u_2	y
0	0	0
0	1	0
1	0	1
1	1	0

(b) $\exists u_1 : BDD(x, 1, u_2)$

x	u_1	y
0	0	0
0	1	0
1	0	0
1	1	1

(c) $\exists u_2 : BDD(x, u_1, 0)$

x	u_1	y
0	0	1
0	1	0
1	0	0
1	1	0

(d) $\exists u_2 : BDD(x, u_1, 1)$

Table 3-4: BDD truth tables after applying the first set

And applying the second set:

x	y
0	1
1	0

(a)
$\exists u_1 :$
$BDD(x, 0)$

x	y
0	0
1	1

(b)
$\exists u_1 :$
$BDD(x, 1)$

x	y
0	0
1	1

(c)
$\exists u_2 :$
$BDD(x, 0)$

x	y
0	1
1	0

(d)
$\exists u_2 :$
$BDD(x, 1)$

Table 3-5: BDD truth tables after applying the second set

Now, observing table 3-5, we know the following:

- If table (a) evaluates to 1, u_1 will be 0 or if table (b) evaluates to 1, u_1 will be 1.
- If table (c) evaluates to 1, u_2 will be 0 or if table (d) evaluates to 1, u_2 will be 1.

3-1-2 A more efficient implementation

Now we could flash the previous 4 BDDs directly into the FPGA but instead of doing this, a better implementation was followed. If we look at the BDDs from tables (b) and (d) of table 3-5, we observe the following:

- If the output from (b) or (d) is $y = 1$, we know that $u_1 = 1$ or $u_2 = 1$ respectively.
- If the output from (b) or (d) is $y = 0$ we cannot conclude the value of the input. We need more information to determine this value.

In order to solve this problem, we can obtain a BDD containing the domain of the controller: $BDD(x)$. This BDD will tell us if for a particular state, there are valid inputs. Figure 3-1 shows the domain, safe set and reachable set of a controller for a dcdc boost converter (This example will be introduced later in section 5-1).

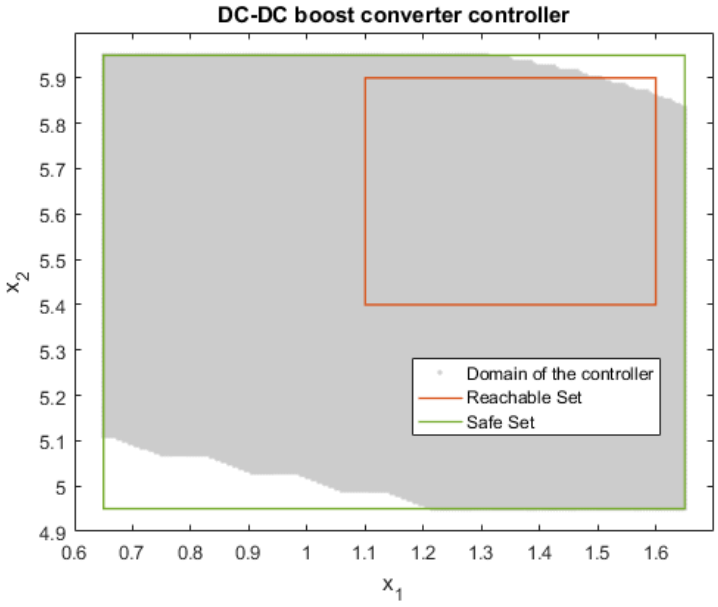


Figure 3-1: Domain, reachable set and safe set of a controller for a dcdc boost converter

From the figure we see that if we are inside the grey dotted area we will have $BDD(x) = 1$ (there are valid control inputs) and $BDD(x) = 0$ if we are outside.

Then, observing again table 3-5, we have to check:

- Check the BDD of the domain to know if there are valid inputs.
- If the output from (b) or (d) is $y = 1$, we know that $u_1 = 1$ or $u_2 = 1$ respectively.
- If the output from (b) or (d) is $y = 0$ and we know that there are valid inputs, the value of the input will be $u_1 = 0$ or $u_2 = 0$ respectively, since it is not 1 and there are no more possible values of the input.

Now we see that if we have valid inputs, the BDDs that we have to flash into the FPGA are (b) and (d), in which we can assume that $y = u_1$ or $y = u_2$:

x	u_1
0	0
1	1

(a) $BDD(x, 1)$

x	u_2
0	1
1	0

(b) $BDD(x, 1)$

Table 3-6: BDD truth tables ready to be flashed

In this approach, instead of having 4 different BDDs, we will have only 3, reducing the size of the final file. Once this process has been finished, we can export the resulting BDDs into a `.blif` file.

Figure 3-2 shows the number of BDDs needed for the first implementation and for the second one:

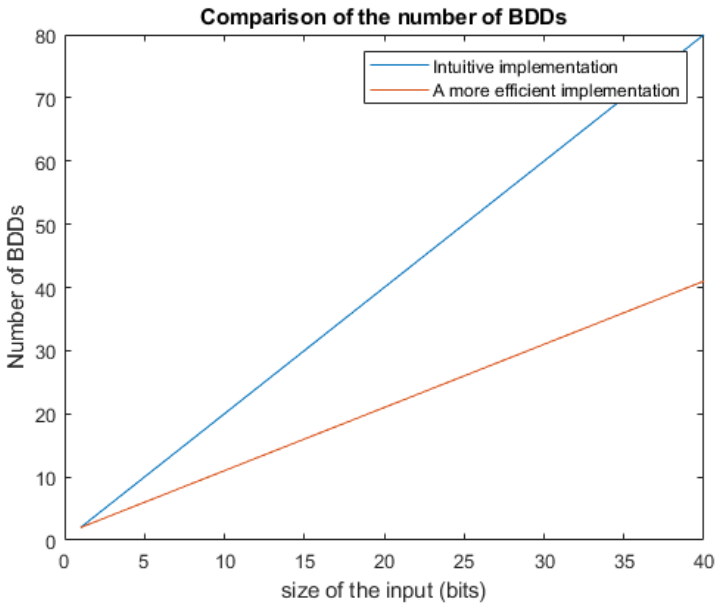


Figure 3-2: Comparison between the first intuitive implementation and the more efficient one

3-2 Second Approach: Exporting directly the BDD

The second approach was to flash the whole BDD of the controller ($BDD(x, u)$) in the FPGA and look for the correct input running a loop (from 0 to $2^n - 1$ where n is the total of number of bits of the input) to evaluate all the possibilities. In this case, no modifications are needed. Therefore, after knowing in which state is the system, we will need to test all possible input values until we obtain $BDD(x, u) = 1$ and pass that value to the plant.

This loop can be run:

- In the FPGA side alongside the controller:

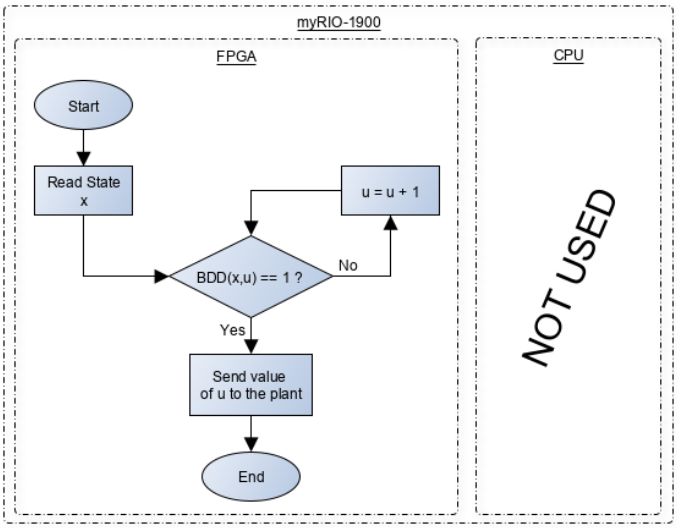


Figure 3-3: Running a loop in the FPGA side to obtain the input value

- In the CPU, sending every test case to the controller in the FPGA to check if that control input is valid:

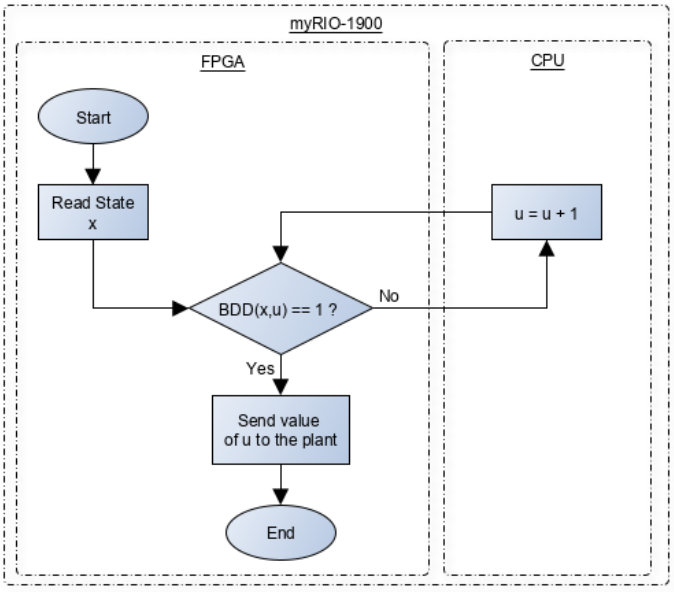


Figure 3-4: Running a loop in the CPU side to obtain the input value

Chapter 4

Flashing the files with LabVIEW FPGA

This chapter explains the procedure to flash the previous controller into the FPGA.

4-1 Example of controller

The next figure (figure 4-1) shows a controller implemented in LabVIEW. After reading the analog values of the states, they need to be transformed to a boolean array in order to be evaluated by the BDD of the controller.

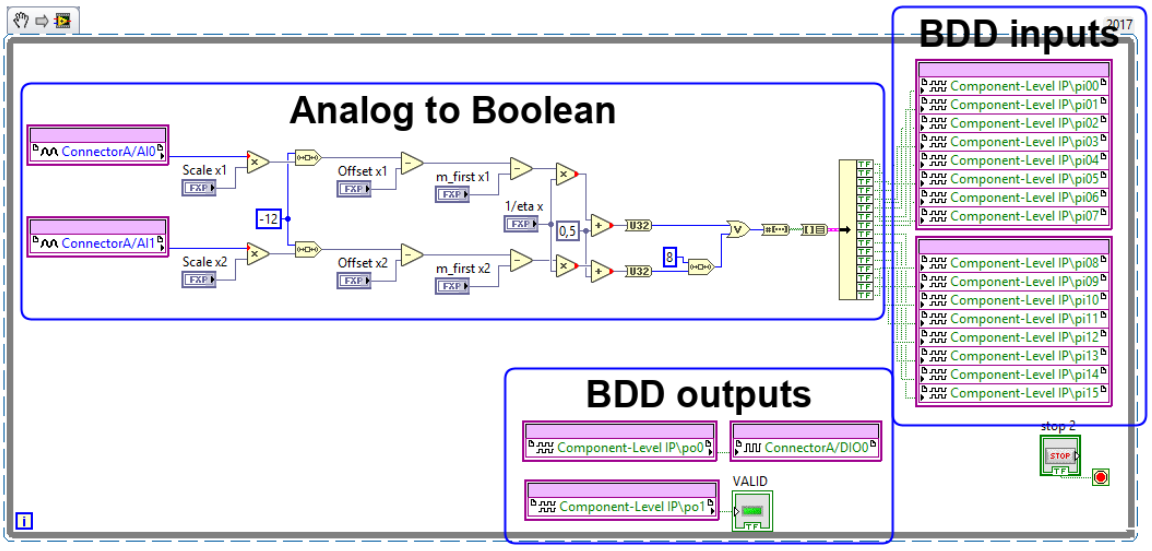


Figure 4-1: Example of controller in LabVIEW

In the right part of the figure, we see the BDD implemented as a CLIP (Component-Level Intellectual Property) module. This module allows us to import VHDL code of the controller

from Vivado to LabVIEW [21]. At the bottom part of the figure, we can also see the outputs of the BDD:

- $po0$: This will be the controller input that will be sent to the plant.
- $po1$: This output will tell us if the read current state is included in the domain of the controller.

In the controller showed in figure 4-1, the output $po1$ is only notifying the user, telling him that the system is inside the domain of the controller or it is not, but we could for example implement a safety routine to shut-down the system, log the error, etc.

4-2 Algorithm used to transform from analog to boolean

4-2-1 From state to grid point

Scots uses an algorithm to transform the real values of the states to the grid points in which the domain of the controller is defined. In scots, when the uniform grid is created, we have the following parameters:

- m_first : It defines the lower left bound of the grid.
- eta : It defines the distance between grid points. So if we have a lower value of eta, we will have more divisions of the grid. This can change drastically the size of the controller.

Then if we have an example of a system with 2 states as in the figure 4-1 we will have to perform in LabVIEW:

$$BDD\_input = \frac{x_1 - m\_first\_x_1}{eta_x} + \left(\frac{x_2 - m\_first\_x_2}{eta_x} << 8\right)$$

Where << is the bitwise left shift operator. The number of shifted positions will depend on the eta parameter. This value can be obtained taking a look at the `.scs` file generated by Scots. In this file, we can see how many bits are dedicated to the states and to the inputs. Finally, we will have to convert the obtained value to boolean in order to evaluate the BDD.

If we had for example a third state, the real value will have to be shifted 16 positions $(8+8)$. In the previous example, the control input has only 1 bit and it does not need any conversion.

4-2-2 From grid point to control input

In case of having a control input composed of more than 1 bit, we will need to do the following conversion from the BDD output to the real value:

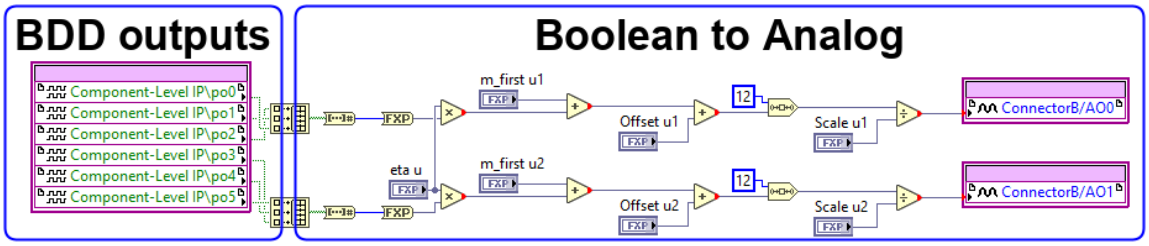


Figure 4-2: Conversion from the output of the BDD to real value

In the figure 4-2 we have a controller that will output 2 control inputs of 3 bits each one. Then we will have to do the following:

$$\begin{aligned} u_1 &= u_{1,BDD} \times eta_u + m\_first\_u_1 \\ u_2 &= u_{2,BDD} \times eta_u + m\_first\_u_2 \end{aligned} \tag{4-1}$$

Where $u_{1,BDD}$ and $u_{2,BDD}$ are the BDD outputs.

4-2-3 Working with analog inputs and outputs

From figures 4-1 and 4-2 we see that we have 2 more additional parameters:

- Offset
- Scale

These parameters where used to transform the raw data value of a state or control input to voltage in order to send the value through the outputs or read the value from the inputs of the myRIO. The formulas to work with analog inputs and outputs can be obtained in the User guide and specifications of NI-myRIO-1900 [22].

Chapter 5

Simulation environment

In this chapter we will explain how to simulate a plant using LabVIEW since we don't have a real system to test and validate the controllers.

5-1 Creating the model

In order to run a real-time simulation, we will create a model of a plant using Simulink and then we will run that model in one myRIO. Using the MATLAB Function block that includes the equations of the dynamics of the system in Simulink, we can create the model of the plant that we are trying to control:

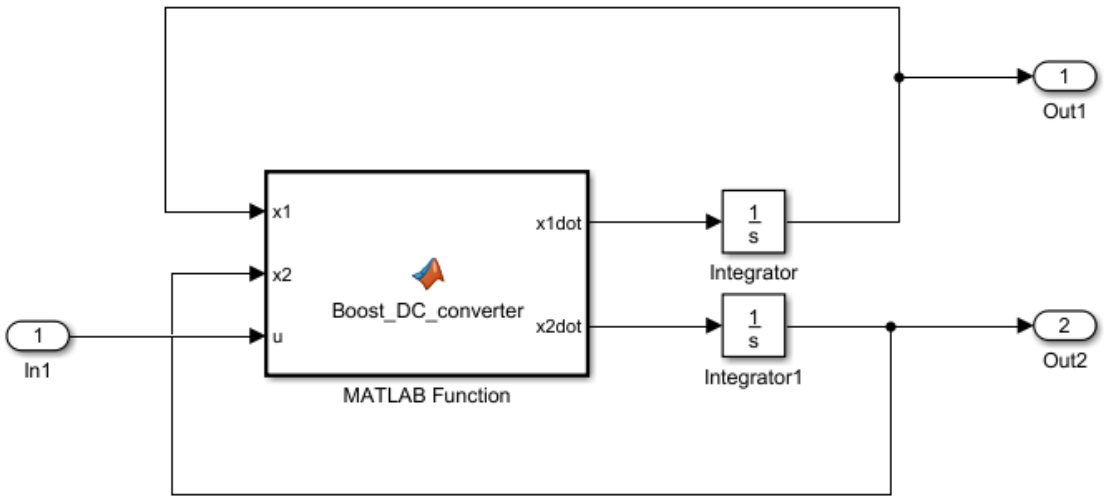


Figure 5-1: Example of plant in LabVIEW

The equations of the dynamics of the system are the same equations defined in the Scots examples when we create the controller in the early stage of the project. For the next example (figure 5-2), we have a boost DC-DC converter (obtained from [23]). It is an example of a switched system with two modes and we have:

$$x(t) = \begin{bmatrix} i_L(t) & v_c(t) \end{bmatrix}^T$$

- 2 states: Current through the inductor $i_L(t)$ and voltage across the capacitor $v_c(t)$.
- 1 control input: Closed or open transistor.

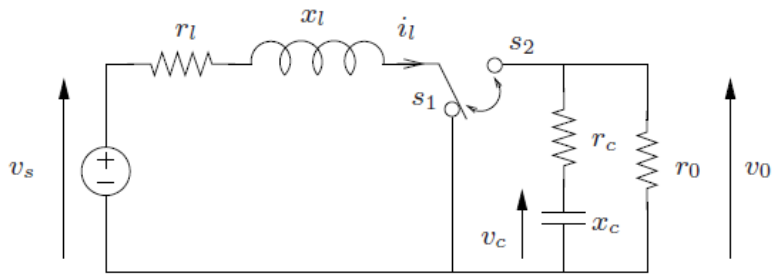


Figure 5-2: DC-DC converter

The dynamics describing the two modes are of the form $\dot{x}(t) = A_p x(t) + b$ $(p = 1, 2)$ With the next matrices:

$$A_1 = \begin{bmatrix} -\frac{r_l}{x_l} & 0 \\ 0 & -\frac{1}{x_c}\frac{1}{r_0 + r_c} \end{bmatrix}, A_2 = \begin{bmatrix} -\frac{r_l}{x_l}\left(r_l + \frac{r_0 r_c}{r_0 + r_c}\right) & -\frac{1}{x_l}\frac{r_0}{r_0 + r_c} \\ \frac{1}{x_c}\frac{r_0}{r_0 + r_c} & -\frac{1}{x_c}\frac{1}{r_0 + r_c} \end{bmatrix}, b = \begin{bmatrix} \frac{v_s}{x_l} \\ 0 \end{bmatrix}$$

Then, they can be implemented in a Matlab script as follows:

```
function [x1dot,x2dot] = Boost_DC_converter(x1,x2,u)

  x = [x1;x2];
  r0=1.0;
  vs = 1.0 ;
  rl = 0.05 ;
  rc = rl / 10 ;
  xl = 3.0 ;
  xc = 70.0 ;

  if(u==1)
    A = [ -rl / xl  0 ;  0  (-1 / xc) * (1 / (r0 + rc)) ] ;
  else
    A = [ (-1 / xl) * (rl + ((r0 * rc) / (r0 + rc)))  ((-1 / xl) * (r0 /
        (r0 + rc))) / 5 ;
          5 * (r0 / (r0 + rc)) * (1 / xc)  (-1 / xc) * (1 / (r0 + rc)) ];
```

```
16    end
17    b = [(vs / xl) ; 0 ] ;
18
19    dxdt= A*x+b;
20    x1dot = dxdt(1);
21    x2dot = dxdt(2);
22  end
```

After building the model, we will obtain the `.so` file of the plant.

5-2 Calling the model from LabVIEW

Now that we have the model, we will need to save it into a second myRIO and call it from LabVIEW. The next figure shows how to call the model from LabVIEW:

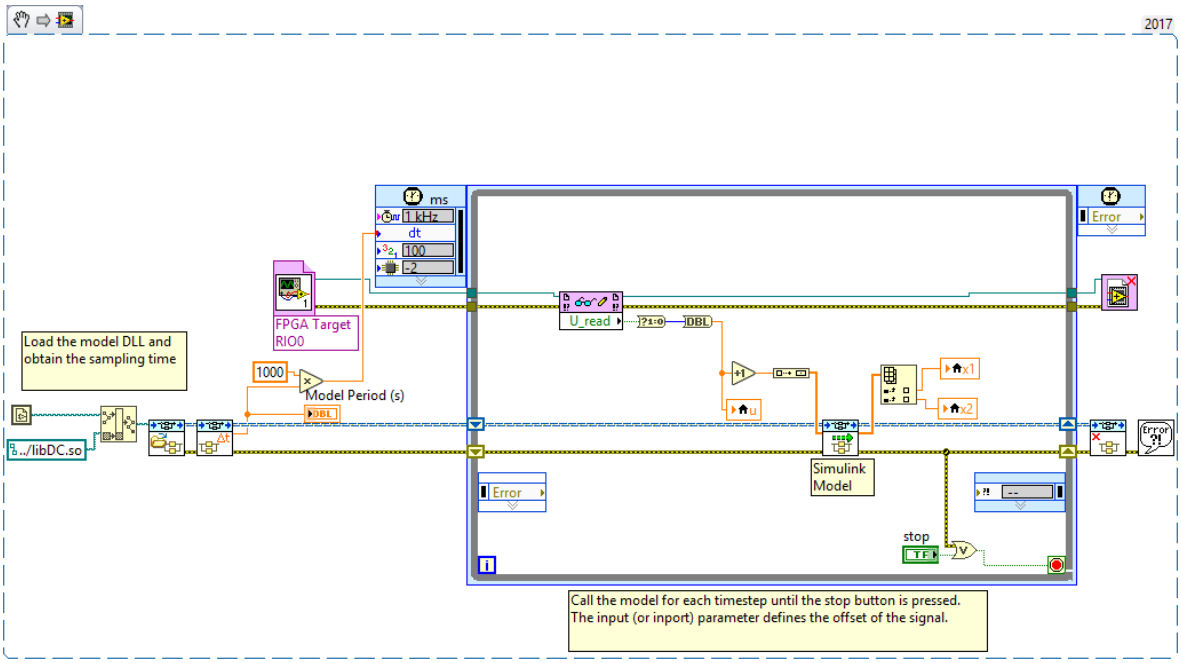


Figure 5-3: Calling a simulink model from LabVIEW

From the previous figure, we can observe the following:

- We are calling the `.so` file of the plant.
- The control input u is passed to the plant.
- The plant will output the states x_1 and x_2.

5-3 Interconnecting the 2 myRIOs

Now that we know how to create and burn a controller in a myRIO, and how to simulate a plant in another myRIO, we have the next possibilities:

5-3-1 First Approach: With a transformed BDD

If we obtain the BDD of a controller according to section 3-1 we will be only using:

- One FPGA in one myRIO to run the controller.
- One CPU in another myRIO to simulate the plant in real-time.

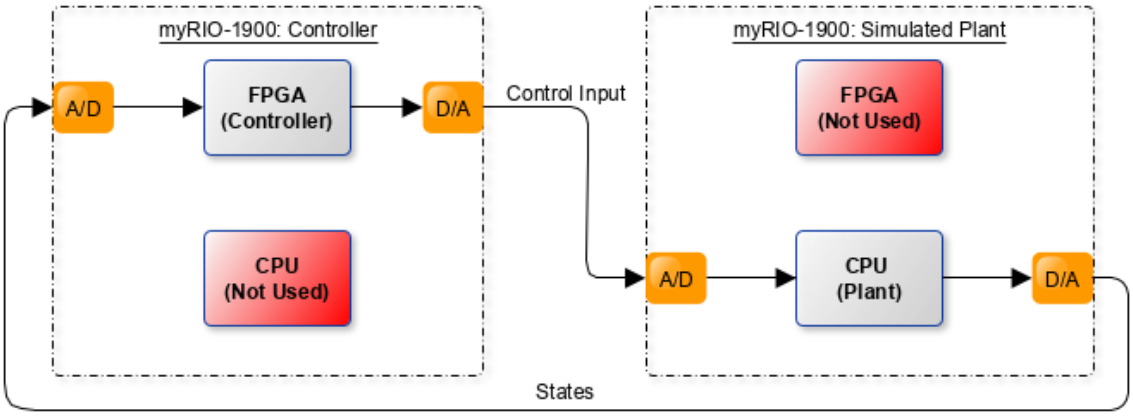


Figure 5-4: Diagram of the interconnection of the 2 myRIOs

5-3-2 Second approach: Running a loop

If the loop is run in the FPGA, the connection will be similar to the one in figure 5-4. But if the loop is run in the CPU, we will have the following:

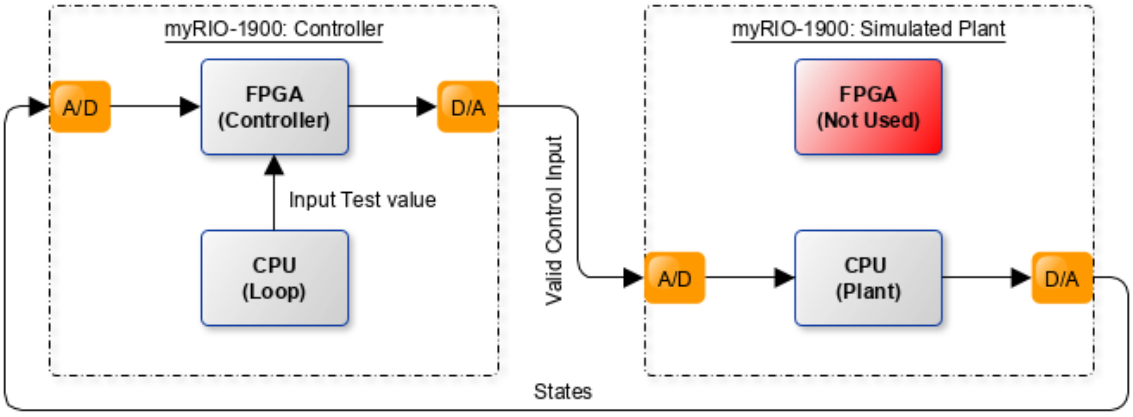


Figure 5-5: New Diagram of the interconnection of the 2 myRIOs

Chapter 6

Results

In this chapter, some of the examples provided by Scots have been simulated and tested using 2 myRIOs.

6-1 Examples

We have tested and validated different examples to check the scalability and feasibility of the project. The next examples are ordered according to the numbers of nodes that the BDD created from Scots contains:

Example	Number of nodes
dcdc	1585
vehicle	7987
aircraft	662173

Table 6-1: Nodes of the BDDs for different examples.

6-1-1 First Example: DC-DC converter

This example was introduced in section 5-1 and we can now simulate in real-time the controlled closed loop interconnecting 2 myRIOs. The next figure shows how both states are able to go from an initial state to the reachability region defined by Scots:

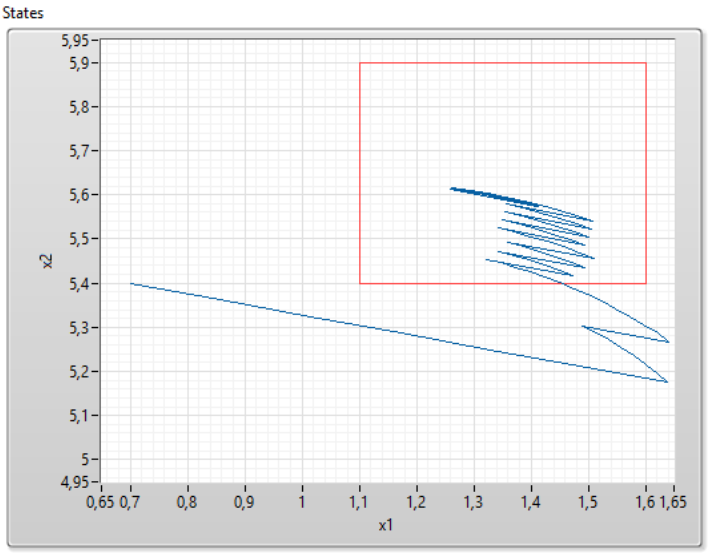


Figure 6-1: States of the DC-DC boost converter running the real-time simulation

Scots provides a Matlab script to simulate the controlled closed-loop. We can observe that the real trajectories of the states are very similar to the ones simulated in Matlab:

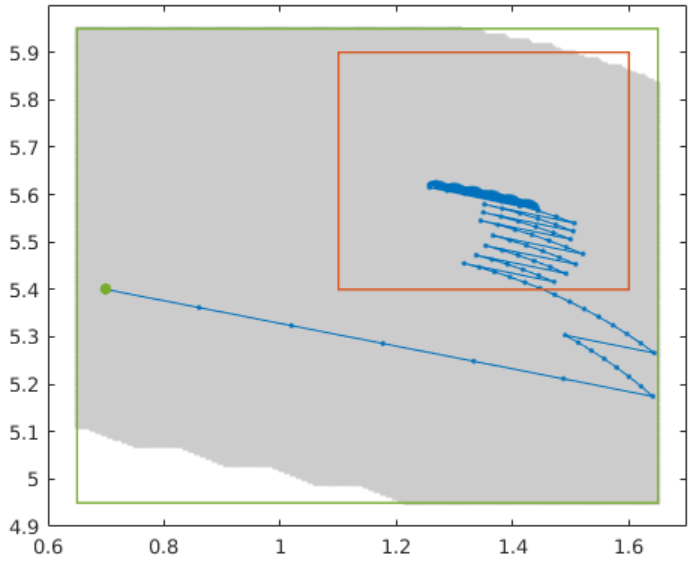


Figure 6-2: States of the DC-DC boost converter simulated using Scots in Matlab

6-1-2 Second Example: Vehicle in a maze

The next example from Scots is a vehicle in a maze path planning problem. This example, obtained from [12], is an autonomous vehicle whose dynamics are of the form $\dot{x} \in f(x, u)$ with $f : \mathbb{R}^3 \times U \longrightarrow \mathbb{R}^3$, $U \subseteq \mathbb{R}^2$ and $W \subseteq \mathbb{R}^3$.

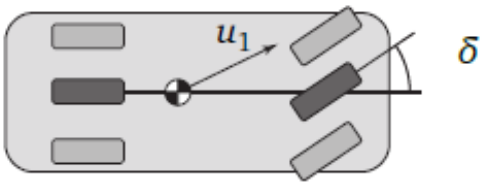


Figure 6-3: Vehicle model

We have now:

- 3 states: Position in the 2-dimensional plane and orientation.
- 2 control inputs: Velocity and the rate of steering angle δ.

And the dynamics are given by:

$$f(x,(u_1,u_2)) = \begin{pmatrix} u_1 \cos(\alpha + x_3)\cos(\alpha)^{-1} \\ u_1 \sin(\alpha + x_3)\cos(\alpha)^{-1} \\ u_1 \tan(u_2) \end{pmatrix}$$

With $U = [-1,1] \times [-1,1]$ and $\alpha = \arctan(\tan(u_2)/2)$.

The controller now will steer the vehicle through the maze. The goal is to reach the green box. After simulating in real-time the controlled closed loop, we obtain the next figure:

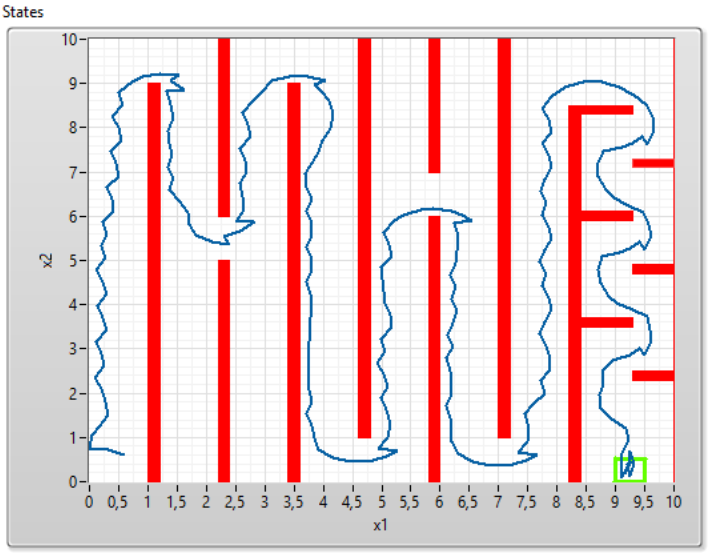


Figure 6-4: States x_1 and x_2 of the vehicle running the real-time simulation

And again, we can compare the real results with the simulated using Scots in Matlab:

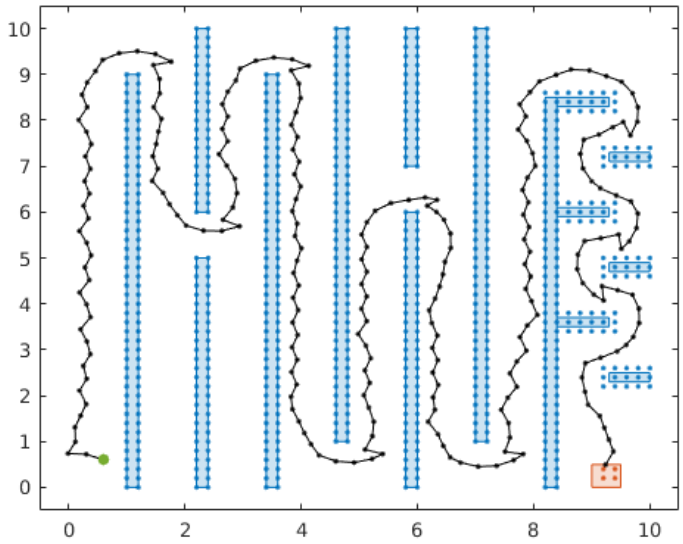


Figure 6-5: States x_1 and x_2 of the vehicle simulated using Scots in Matlab

6-1-3 Third Example: Aircraft

This last example was not simulated because the generated controller exceeded the physical limits of the FPGA, but we measured the utilization and the timings in order to compare them with other tools. The results are summarized later in this chapter. Again, this example description can be found in [12].

6-2 Comparison

In this section, we will call:

- *split*: to the implementation followed by the first approach (section 3-1-2).
- *FPGA*: to the implementation followed by the second approach (section 3-2) with the loop running in the FPGA alongside the controller BDD.
- *CPU*: to the implementation followed by the second approach (section 3-2) with the loop running in the CPU.
- *TUM*: to the implementation followed by the BDD2Implement tool from TUM.

The next table shows the utilization of LookUp Tables (LUTs) needed by the FPGA after synthesizing the controller with Vivado and after flashing it with LabVIEW into the FPGA:

Example	Vivado Usage	LabVIEW Usage
dcdc_split	0.31%	51.50%
dcdc_FPGA	0.54%	52.00%
dcdc_CPU	0.54%	51.70%
dcdc_TUM	1.73%	52.70%
vehicle_split	17.45%	83.10%
vehicle_FPGA	20.49%	84.80%
vehicle_CPU	20.49%	84.70%
vehicle_TUM	65.69%	130.90%
aircraft_split	227.98%	not tested
aircraft_FPGA	377.53%	not tested
aircraft_CPU	377.53%	not tested
aircraft_TUM	timeout	not tested

Table 6-2: FPGA LUTs utilization for different examples

It is possible to observe from Table 6-2:

- For each example, the percentage of LUTs used over the total is the same if the loop is run in the FPGA or in the CPU. This is because the synthesized controller is the same in both cases.

- When coding the controller and flashing it into the FPGA using LabVIEW, it needs less resources if the loop through the inputs is done in the CPU. This is expected since part of the code is running in the CPU instead of in the FPGA. Since there is not a significant difference in FPGA resource utilization, we could run everything in the FPGA, removing that way the inter-process communication CPU-FPGA, making the controller more robust.

- The aircraft example was not tested in LabVIEW since the Vivado Usage was already exceeding the maximum number of resources of the FPGA. Also, Vivado was not able to synthesize the controller generated by the TUM tool due to the large size of the input file.

- Comparing our tool with the TUM tool, the split controller yields the best result.

The next table shows the required time for each of the stages of the project:

Example	Determinization	to Blif	to Verilog	Total
dcdc_split	0.653s	0.001s	0.025s	0.679s
dcdc_FPGA/CPU	0.653s	0.001s	0.025s	0.679s
dcdc_TUM	0.245s	-	0.004s	0.249s
vehicle_split	0.998s	0.010s	0.044s	1.052s
vehicle_FPGA/CPU	0.998s	0.002s	0.032s	1.032s
vehicle_TUM	0.459s	-	0.099s	0.558s
aircraft_split	104.434s	0.159s	0.274s	104.867s
aircraft_FPGA/CPU	104.434s	0.032s	0.245s	104.711s
aircraft_TUM	55.210s	-	5.932s	61.142s

Table 6-3: Timing Comparison

From the previous table, we observe the following:

- The time for determinizing the BDD is the same for the FPGA/CPU and split approach since the controller is the same.
- The tool from TUM is faster determinizing the controller. This can be explained since that tool is using a random determinization method while the tool from [7] is using a local determinization method. In this method, the BDD is compressed and determinized.
- Converting the whole BDD (FPGA/CPU) to `blif` and to `verilog` is faster than converting the split BDD.
- With smaller controllers, when generating the Verilog file, the tool from TUM is faster but the ABC tool yields better results for bigger controllers.

After generating the files, we can also measure the time that Vivado takes in synthesizing the controllers:

Example	Synthesis time (hh:mm:ss)
dcdc_split	00 : 00 : 39
dcdc_FPGA/CPU	00 : 00 : 37
dcdc_TUM	00 : 00 : 50
vehicle_split	00 : 02 : 06
vehicle_FPGA/CPU	00 : 02 : 08
vehicle_TUM	00 : 04 : 48
aircraft_split	02 : 31 : 02
aircraft_FPGA/CPU	01 : 24 : 20
aircraft_TUM	timeout

Table 6-4: Synthesis Time for different examples from Vivado

From table 6-4 we can see that the synthesis of a FPGA/CPU controller is faster than for the split controller for bigger controllers. We also see that our approaches outperform the one followed by the TUM tool.

We can conclude that the tool from TUM is faster generating the files and converting the controller in comparison to the tools used in this project. However, the final utilization in our case and hence the synthesis time is smaller.

Chapter 7

Conclusions and Future Work

In this chapter, the conclusions and recommendations for future work are described.

7-1 Conclusions

In this thesis, we have created a tool that takes any symbolic controller stored in a BDD generated by Scots and outpust the necessary files to be flashed into an FPGA. In chapter 5 we have described a method to run the output of the tool using an FPGA from National Instruments: myRIO FPGA. In this method, the inputs and outputs of the FPGA are connected to the BDD of the controller using LabVIEW. In order to extrapolate this method to a generic FPGA, we will have to configure the inputs and outputs of the generic FPGA and implement the algorithm used to transform from analog value to boolean in VHDL. This new process will depend of the FPGA used.

If we compare our tool with the ones that are being currently used, ours takes more time generating the FPGA files but it is more efficient, it consumes less resources of the FPGA. This makes possible the use of higher size symbolic controllers, allowing the closed-loop system to fulfill faster and more accurate the required specifications.

For the testing and simulation of the controlled closed-loop we have used a simulated plant since we didn't have a real system to work with. The plant was simulated on a myRIO board, it was wired to another myRIO running the controller and we have seen that the controller was able to drive the system to the required specifications. Once we have a physical plant, we can replace the myRIO board running the simulated system and the behavior should be the same, but we may want to test this further considering additional effects as noise.

7-2 Future Work

In order to continue with this project, there are some next steps that couldn't be done due to time limitations:

- Create larger controllers and test them in a RIO device. Since the board used in this project (myRIO - 1900) is for educational purposes, the number of resources of its FPGA is limited. But NI provides a variety of RIO devices with larger capacity [24]. Using one more powerful board will allow us to test larger controllers. To generate bigger controller the user can simply play with the `eta` parameter in Scots.
- Build a Graphical User Interface (GUI) in order to make the process more user-friendly and more intuitive.
- Integrate the graphs generated by Scots in the GUI. This will allow us to check the domains of the controller as well as the simulated closed-loop created by Scots.
- Implement in VHDL the algorithm to transform from analog(read states) to boolean(input of the BDD). This way, the user will only need to configure the I/O interface if another FPGA is used rather than a myRIO board.
- Test the controller using a real plant instead of simulating it.

Bibliography

[1] P. Tabuada. Verification and control of hybrid systems: A symbolic approach. *Springer Publishing Company, Incorporated, 1st ed*, 2009.

[2] M. Jr. Mazo, A. Davitian, and P. Tabuada. Pessoa: A tool for embedded controller synthesis. *Computer Aided Verification. Springer*, page 566–569, 2010.

[3] M. Rungger and M. Zamani. Scots: A tool for the synthesis of symbolic controllers. *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control, HSCC*, pages 99–104, 2016.

[4] Sebti Mouelhi, Antoine Girard, and Gregor Gössler. Cosyma: A tool for controller synthesis using multi-scale abstractions. In *Proceedings of the 16th International Conference on Hybrid Systems: Computation and Control, HSCC '13*, pages 83–88. New York, NY, USA, 2013.

[5] Cameron Finucane, Gangyuan Jing, and Hadas Kress-Gazit. Ltlmop: Experimenting with language, temporal logic and robot control. In *IEEE/RSJ Int'l. Conf. on Intelligent Robots and Systems*, pages 1988–1993. Taipei, Taiwan, October 2010.

[6] Tichakorn Wongpiromsarn, Ufuk Topcu, Necmiye Ozay, Huan Xu, and Richard M. Murray. Tulip: A software toolbox for receding horizon temporal logic planning. In *Proceedings of the 14th International Conference on Hybrid Systems: Computation and Control, HSCC '11,*, pages 313–314. New York, NY, USA, 2011.

[7] Ivan S. Zapreev, Cees Verdier, Manuel Mazo Jr. Optimal symbolic controllers determinization for bdd storage. *Center for Systems and Control, Technical University of Delft, The Netherlands*, March 2018.

[8] Antoine Girard. Low-complexity switching controllers for safety using symbolic models. *IFAC Proceedings Volumes*, 45(9):82–87, 2012.

[9] Khaled Mahmoud, Technische Universität München. Bdd2implement - a code generation tool for bdd-based symbolic controllers. https://gitlab.lrz.de/hcs/BDD2Implement, January 2018.

[10] Mahmoud Khaled, Eric S. Kim, Murat Arcak, and Majid Zamani. Synthesis of symbolic controllers: A parallelized and sparsity-aware approach. In Tomáš Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 265–281, Cham, 2019. Springer International Publishing.

[11] P. Tabuada. On the synthesis of correct-by-design embedded control software, April 2009.

[12] Gunther Reissig, Alexander Weber, and Matthias Rungger. Feedback Refinement Relations for the Synthesis of Symbolic Controllers. *arXiv e-prints*, page arXiv:1503.03715, Mar 2015.

[13] F. Somenzi. Cudd: Cu decision diagram package. *University of Colorado at Boulder*, 1998.

[14] R. T. Rockafellar and R. J.-B. Wets. Variational analysis. *3rd corr printing 2009. Springer*, 317, 1998.

[15] Dr. Ivan S. Zapreev. Scotsv2.0 bdd controllers determinizer. https://github.com/ivan-zapreev/SCOTS2C, May 2018.

[16] Berkeley Verification and Synthesis Research Center. Abc: A system for sequential synthesis and verification. https://people.eecs.berkeley.edu/~alanmi/abc/abc.htm, July 2005.

[17] Xilinx. Vivado design suite. https://www.xilinx.com/products/design-tools/vivado.html.

[18] National Instruments. Using xilinx vivado design suite to prepare verilog modules for integration into labview fpga. http://www.ni.com/tutorial/54793/en/, May 2018.

[19] National Instruments. Labview. http://www.ni.com/nl-nl/shop/labview.html.

[20] Mathworks. Simulink. https://www.mathworks.com/products/simulink.html.

[21] National Instruments. Importing external ip into labview fpga. http://www.ni.com/tutorial/7444/en/, October 2018.

[22] National Instruments. User guide and specifications, ni myrio-1900. http://www.ni.com/pdf/manuals/376047c.pdf, May 2016.

[23] Antoine Girard, Giordano Pola, and Paulo Tabuada. Approximately bisimilar symbolic models for incrementally stable switched systems. pages 116–126. IEEE, 2009.

[24] National Instruments. Slices on an fpga chip. http://www.ni.com/product-documentation/54503/en/, Nov 2018.

Glossary

List of Acronyms

BDD	Binary Decision Diagram
GA	Global Algorithm
SR	Symbolic Regression
LA	Local Algorithm
FPGA	Field Programmable Gate Array
DSP	Digital Signal Processor
TUM	Technische Universität München
CUDD	Colorado University Decision Diagram
HDL	Hardware Description Language
LUTs	LookUp Tables
GUI	Graphical User Interface