



Technische Universität München

Department of Hybrid Control System

Master's Thesis in Formal Verification

Implementation of a symbolic controller
using SCOTS

Author: Rafif Hassis

Matriculation Number: 03639600

Supervisors: Mahmoud Khaled

: Prof. Dr. Majid Zamani

Date: January xx, 2019

I confirm that this master thesis in is my own work and I have documented all sources and material used.

Munich, January 2019

Rafif Hassis

Summary

Acknowledgments	7
Abstract	9
Chapter 1.....	13
General Introduction	13
Chapter 2.....	15
Background and definitions	15
Chapter 3.....	19
Theoretical Formulation.....	19
and presentation of SCOTS.....	19
3.1. Presentation of Scots	19
3.2. Control Problem	19
3.3. Auxiliary Control Problems.....	21
3.4. Growth Bound.	22
3.5. Closed Loop.	23
3.6. Synthesis via Fixed Point Computations.....	24
3.7. Conclusion	26
Chapter 4.....	27
Implementation of symbolic controllrs on FPGAs	27
4.1. Introduction.....	27
4.2. Simulation on opal 4200.....	27
4.3. OPAL-RT and test unicycle dynamics in Simulink (Hardware in the loop simulator)	28
4.4. Software In the Loop Simulation of the synthesized controller.....	30
4.5. Hardware in the Loop Simulation of the synthesized controller	31
4.6. Testing the whole system (Controller and Dynamics system) on Simulink	32
4.7. Testing the whole system (Controller and Dynamics system) on RT-Lab	32
4.8. Hardware in a loop with Zedboard	33
4.8.1. Axi protocol	34
4.8.2. Design of the IP core	36
4.8.3. Implementing a C application with Petalinux.....	41
4.8.4. Result.....	42
4.9. Conclusion	43
Chapter 5.....	45
Implementation of the control over Raspberry pi	45

5.1. THE RASPBERRY PI 3	45
5.2. THE RASPBERRY PI 3 CAPABILITIES.....	45
5.3. THE C++ Code	46
5.4. Conclusion	47
Chapter 6.....	49
Robot Control	49
6.1. . The State space.....	50
6.2. The Controller synthesis.....	51
6.3. The header file implementation.....	53
6.5. One robot control	54
6.6. Control of two robots	58
Chapter 7.....	63
Conclusion and Future Work.....	63
7.1. Conclusion	63
7.2. Future Work	64
List of Figures.....	65
Bibliography.....	67
Appendix 1 vehicle_simul.m	69
Appendix 2 BDD.m	73
Appendix 3 bddhil.c.....	75
Appendix 4 bddReader.h.....	79
Appendix 5 bdd_vehicle.h.....	83

Acknowledgments

To my family, nucleus of love, symbol of living together, giving and getting and building history.

To my native country, and to the adopted one, for education, and for giving me an opportunity to learn, to develop interesting things, and simply to have a good life.

To my native culture, my adopted one, and to any real culture, that have making me better.

To the science, the innovation spirit, and to technologies, making our life more interesting.

To my teachers, professors, mainly to my TUM professors.

A special thanks to my supervisor Mahmoud Khaled for his effort, kindness, and availability.

Abstract

The purpose of this thesis is the implementation of a symbolic controller, for unicycle dynamics on FPGAs and Raspberry Pi microcontrollers. The controller is then tested on robots.

The formal verification, which is the only known way to guarantee that a system is free of errors, is used in the synthesis of this controller.

For a given:

- safety property, which is that a system will never do something contradictory to the desired behavior,
- reach ability specification, which is that the system will eventually reach the desired goal,

A symbolic controller is synthesized. For this purpose, a tool for automated synthesis of symbolic controllers is needed. The used one in this thesis is SCOTS [1].

. SCOTS is intended for the use in the area of formal methods for Cyber-Physical Systems (CPS) which is integrations of computation and physical processes. Embedded computers and networks monitor and control the physical processes, usually with feedback loops where physical processes affect computations and vice versa.

Kurzfassung

A short abstract of the Diplomarbeit in German. Only about 4 sentences.

Chapter 1.

General Introduction

We attempt to test and verify the controller generated from SCOTS, which is a tool for the synthesis of symbolic controllers, using different real-time platforms. This includes:

- a real-time hardware in-the-loop (HIL) simulator called RT-LAB OP4200,
- a FPGA board: Zed-board,
- a Raspberry Pi SOC,
- Robot

We use SCOTS for the synthesis of formally-verified controllers that will be targeted to different implementation platforms.

SCOTS is an open source software tool, mainly implemented in C++, used for the synthesis of symbolic controllers for possibly perturbed, nonlinear, control systems. With a small MATLAB interface, we can easily access the synthesized controller, and using the CUDD library, the controller is provided in BDD format.

The package of CUDD [2], which is written in C, supports Binary Decision Diagram (BDDs), Algebraic Decision Diagrams (ADDs) and Zero-suppressed Binary Decision Diagrams (ZDDs).

SCOTS is given as a set of header files including:

- Symbolicset.h,
- Symbolicmodel.h
- SymbolicModelGrowthBound.h.

The following steps shows how SCOTS works:

1. The user provides scots with the dynamical system as differential equations and a sampling period.
2. SCOTS uses the class SymbolicSet to quantize the state and input sets.
3. SCOTS uses the class SymbolicModelGrowthBound to compute the transition function.

4. The user uses the class `SymbolicSet` to specify the atomic propositions in the specification
5. SCOTS uses the class `FixedPoint` to solve the auxiliary control problem
6. SCOTS uses the class `SymbolicSet` to write the resulting controller to a BDD file.

As the controllers synthesized by SCOTS take the form of BDD binaries, they cannot be directly used in targeted implementation platforms. Even if the target has support for such encoded files, this requires the availability of the CUDD library running in it, which is not always practically possible.

This thesis is divided into three parts:

- In the first part, we tested our controller in a first stage using mainly a software in the loop technique. In a second stage, we simulated our dynamics in a real time simulator, Hardware in the loop(HIL). Finally we implemented the controller on the FPGA using VHDL and we designed the hardware using Vivado, a tool for FPGA development.
- In the second part, we simulate the dynamics of the unicycle together with the controller from within Raspberry Pi using C++.
- In the third part, we test our controller on a robot that should go to a specific goal while avoiding some states of the urban like environment platform.

Finally, the work is concluded, and some opening perspectives are briefly described.

Chapter 2.

Background and definitions

Here presented some classical definitions obtained from scientific sources and not modified. The references are here done.

Cyber-Physical Systems [3]

Cyber-Physical Systems are integrations of computation and physical processes. Embedded computers and networks monitor and control the physical processes, usually with feedback loops where physical processes affect computations and vice versa

HIL [4]

Hardware-in-the-loop (HIL) simulation is a technique used in the development and test of complex real-time embedded systems. HIL simulation provides an effective platform by adding the complexity of the plant under control to the test platform. The complexity of the plant under control is included in test and development by adding a mathematical representation of all related dynamic systems. These mathematical representations are referred to as the “plant simulation”. The embedded system to be tested interacts with this plant simulation.

A HIL simulation must include electrical emulation of sensors and actuators. These electrical emulations act as the interface between the plant simulation and the embedded system under test. The value of each electrically emulated sensor is controlled by the plant simulation and is read by the embedded system under test (feedback). Likewise, the embedded system under test implements its control algorithms by outputting actuator control signals. Changes in the control signals result in changes to variable values in the plant simulation.

Opal4200 [5]

The OP4200 RCP/HIL system offers Hardware-in-the-Loop (HIL), Rapid Control Prototyping (RCP), data acquisition and I/O expansion capabilities in a desktop-friendly package. OP4200 supports power electronics, automotive and other real-time applications for industry and academia.

OP4200 comes equipped with a Xilinx Zynq® All Programmable SoC, featuring a Kintex™7 FPGA and dual-core ARM® Cortex processor, and the same class-leading FPGA-based I/Os and real-time solvers provided throughout OPAL-RT product line, making it the ideal choice for closed-looped applications.

FPGA [6, 7]

Field Programmable Gate Arrays (FPGAs) are semiconductor devices that are based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects. FPGAs can be reprogrammed to desired application or functionality requirements after manufacturing. This feature distinguishes FPGAs from Application Specific Integrated Circuits (ASICs), which are custom manufactured for specific design tasks. Although one-time programmable (OTP) FPGAs are available, the dominant types are SRAM based which can be reprogrammed as the design evolves.

You can think of an FPGA as a blank slate. By itself an FPGA does nothing. It is up to you (the designer) to create a configuration file, often called a bit file, for the FPGA. Once loaded the FPGA will behave like the digital circuit you designed!

ZEDBOARD [8, 9]

ZedBoard™ is a complete development kit for designers interested in exploring designs using the Xilinx Zynq®-7000 All Programmable SoC. The board contains all the necessary interfaces and supporting functions to enable a wide range of applications.

The expandability features of this evaluation and development platform make it ideal for rapid prototyping and proof-of-concept development. The ZedBoard includes Xilinx XADC, FMC (FPGA Mezzanine Card), and Digilent Pmod™ compatible expansion headers as well as many common features used in system design. ZedBoard enables embedded computing capability by using DDR3 memory, Flash memory, gigabit Ethernet, general purpose I/O, and UART technologies.

Formal verification [10]

In the context of hardware and software systems, formal verification is the act of proving or disproving the correctness of intended algorithms underlying a

system with respect to a certain formal specification or property, using formal methods of mathematics.

The verification of these systems is done by providing a formal proof on an abstract mathematical model of the system, the correspondence between the mathematical model and the nature of the system being otherwise known by construction. Examples of mathematical objects often used to model systems are: finite state machines, labeled transition systems, Petri nets, vector addition systems, timed automata, hybrid automata, process algebra, formal semantics of programming languages such as operational semantics, denotation semantics, axiomatic semantics and Hoare logic

Validation and verification [11]

Validation and verification are the two steps in any simulation project to validate a model.

- Validation is the process of comparing two results. In this process, we need to compare the representation of a conceptual model to the real system. If the comparison is true, then it is valid, else invalid.
- Verification is the process of comparing two or more results to ensure its accuracy. In this process, we have to compare the model's implementation and its associated data with the developer's conceptual description and specifications.

Chapter 3.

Theoretical Formulation and presentation of SCOTS

Here, we introduce the theoretical formulation and a brief presentation of SCOTS taken from its manual [1].

3.1. Presentation of Scots

For the synthesis of symbolic controllers for possibly perturbed, nonlinear, control systems, SCOTS which is an open source software (available at <http://www.hcs.ei.tum.de>) is here presented and used. It is implemented in C++ and access to the synthesized controller from within a MATLAB interface.

SCOTS is intended to be used and extended by researches in the area of formal methods for cyber physical systems. SCOTS provides a baseline implementation of one of the most basic approaches to symbolic synthesis.

The term cyber-physical systems (CPS) refer to a new generation of systems with integrated computational and physical capabilities that can interact with humans through many new modalities. The ability to interact with, and expand the capabilities of, the physical world through computation, communication, and control is a key enabler for future technology developments. Opportunities and research challenges include the design and development of next-generation airplanes and space vehicles, hybrid gas-electric vehicles, fully autonomous urban driving, and prostheses that allow brain signals to control physical objects.

The main concepts of SCOTS were introduced as three steps. In the first step, the concrete infinite system is lifted to an abstract domain where it is substituted by a finite system which is referred to as abstraction or symbolic model. In the second step, an auxiliary problem on the abstract domain is solved and finally the controller that had been synthesized for the abstraction is refined to the concrete system.

3.2. Control Problem

SCOTS supports the controller computation of symbolic models of perturbed control systems of the form:

$$\dot{\xi}(t) \in f(\xi(t), u) + [[-w, w]] \quad (3.1)$$

Where:

- f is a function given by $f: R^n \times U \rightarrow R^n$ and $U \subseteq R^m$.
- $w = [w_1, \dots, w_n] \in R_+^n$ is the vector perturbation bound
- $[-w, w]$ is the hyper-interval $[-w_1, w_1] \times \dots \times [-w_n, w_n]$

Given a sampling time $\tau > 0$, SCOTS define a solution of (3.1) on $[0, \tau]$ under (constant) input $u \in U$ as an absolutely continuous function $\xi: [0, \tau] \rightarrow R^n$ that satisfies (3.1) for almost every $t \in [0, \tau]$.

The desired behavior of the closed loop is defined with respect to the τ -sampled behavior of the continuous-time systems (3.1). To this end, the sampled behavior of (3.1) is casted as simple system.

$$S_1 = (X_1, U, F_1) \quad (3.2)$$

with the state alphabet $X_1 = R^n$, input alphabet $U_1 = U$ and the transition function $F_1: X_1 \times U_1 \rightrightarrows X_1$ defined by

$$F_1(x, u) := \left\{ x' \mid \exists \xi \text{ is a solution of (1) on } [0, \tau] \text{ under } \begin{array}{l} u: \xi(0) = x \wedge \xi(\tau) = x_0 \end{array} \right\}.$$

A specification Σ_1 for a simple system $S_1 = (X_1, U_1, F_1)$ is simply a set

$$\Sigma_1 \subseteq \bigcup_{T \in \mathbb{Z} \geq 0 \cup \{\infty\}} (U_1 \times X_1)^{[0; T[} =: (U_1 \times X_1)^\infty \quad (3.3)$$

of possibly finite and infinite input-state sequences. A simple system S_1 together with a specification Σ_1 constitute a control problem (S_1, Σ_1) .

SCOTS natively supports invariance (often referred to as safety) and reachability specifications. Consider two sets $I_1 \subseteq X_1$ and $Z_1 \subseteq X_1 \times U_1$. A reachability specification associated with I_1, Z_1 is defined by

$$\Sigma_1 := \{(u, x) \in (U_1 \times X_1)^\infty \mid x(0) \in I_1 \Rightarrow \exists t \in [0; T[: (x(t), u(t)) \in Z_1 \}.$$

An invariance specification associated with I_1, Z_1 follows by

$$\Sigma_1 := \{(u, x) \in (U_1 \times X_1)^{[0; \infty[} \mid x(0) \in I_1 \Rightarrow \forall t \in [0; \infty[: (x(t), u(t)) \in Z_1 \}.$$

In this context, the sets I_1 and Z_1 are referred to as atomic propositions. SCOTS allows to define arbitrary sets as atomic propositions. In the BDD implementation, it provides customized commands to define

- polytopes $\{x \in R^n \mid H_x \leq h\}$ parameterized by $H \in R^{q \times n}$, $h \in R^q$, and

- ellipsoids $\{x \in \mathbb{R}^n \mid |L(x - y)|_2 \leq 1\}$ parameterized by $L \in \mathbb{R}^{n \times n}$ and $y \in \mathbb{R}^n$

To summarize it, we first provide SCOTS with the state- and input-space, we then define the dynamical system representing the plant. Giving a sampling time τ , we compute the solution of the dynamical system which results into a symbolic system. We finally define the specification (safety or reach ability) using polytopes and ellipsoids. For example, given some parameter, a defined ellipsoid would practically represent a small portion of the state space. This portion would be our target set or our obstacle depending on how we compute it later with some other SCOTS functions. The figure 3.1 below represents a polytope and an ellipsoid.

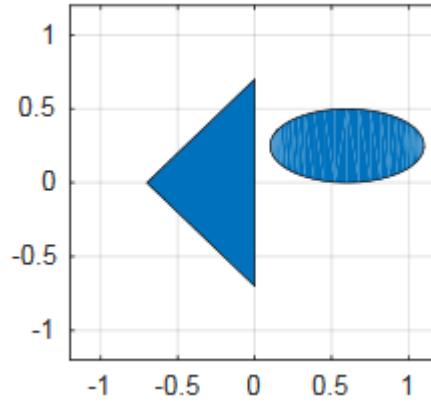


Figure 3.1. Representation of a polytope on the left of the figure and an ellipsoid on the right

3.3. Auxiliary Control Problems

Given a simple system $S_1 = (X_1, U_1, F_1)$ representing the τ -sampled behavior of (3.1) and a specification Σ_1 for S_1 , the control problem (S_1, Σ_1) is not solved directly, but an auxiliary, finite control problem (S_2, Σ_2) is used in the synthesis process. Here, $S_2 = (X_2, U_2, F_2)$ is a symbolic model or (discrete) abstraction of S_1 and Σ_2 is an abstract specification.

The state alphabet of X_2 is a cover of X_1 and the input alphabet U_2 is a subset of U_1 . The set X_2 contains a subset \bar{X}_2 , representing the “real” quantizer symbols, while the remaining symbols $X_2 \setminus \bar{X}_2$ are interpreted as “overflow” symbols. The set of real quantizer symbols \bar{X}_2 are given by congruent hyper-rectangles aligned on a uniform grid

$$\eta Z^n = \{c \in \mathbb{R}^n \mid \exists_{k \in \mathbb{Z}^n} \forall_{i \in [1;n]} c_i = k_i \eta_i\} \quad (3.4)$$

with grid parameter $\eta \in (\mathbb{R}_+ \setminus \{0\})^n$, i.e.,

$$x_2 \in \bar{X}_2 \implies \exists_{c \in \eta Z^n} x_2 = c + [-\eta/2, \eta/2]. \quad (3.5)$$

SCOTS computes symbolic models that are related via feedback refinement relations with the plant. A feedback refinement relation from S_1 to S_2 is a strict relation $Q \subseteq X_1 \times X_2$ that satisfies for all $(x_1, x_2) \in Q$ and for all $u \in U_2$ with

$$F_2(x_2, u) \neq \emptyset$$

With the conditions

$$(1) F_1(x_1, u) \neq \emptyset \text{ and}$$

$$(2) Q(F_1(x_1, u)) \subseteq F_2(x_2, u).$$

In SCOTS, the feedback refinement relation Q is given by the set-membership relation

$$Q := \{(x_1, x_2) \mid x_1 \in x_2\}. \quad (3.6)$$

Given an invariance (reachability) specification Σ_1 for S_1 associated with (I_1, Z_1) an abstract specification is given by the invariance (reachability) specification for S_2 associated with

$$I_2 = \{x_2 \in X_2 \mid x_2 \cap I_1 \neq \emptyset\} \text{ and } Z_2 = \{x_2 \in X_2 \mid x_2 \subseteq Z_1 \neq \emptyset\}. \quad (3.7)$$

For the solution of the auxiliary control problems (S_2, Σ_2) SCOTS provides minimal and maximal fixed point algorithms.

Practically, here we create a hyper-rectangle representing a cover of the real space giving the lower and upper bound. We then quantize our hyper-rectangle into small boxes giving a quantization step. That way we obtain our auxiliary state and input space.

3.4. Growth Bound.

The construction of a symbolic model S_2 of S_1 is based on the over-approximation of attainable sets. A growth bound of (1) is a function

$$\beta : \mathbb{R}_+^n \times U' \rightarrow \mathbb{R}_+^n,$$

which is defined with respect to a sampling time $\tau > 0$, a set $K \subseteq \mathbb{R}^n$ and a set $U' \subseteq U$. Basically, it provides an upper bound on the deviation of solutions ξ of

(3.1) from nominal solutions φ of (3.1), i.e., for every solution ξ of (3.1) on $[0, \tau]$ with input $u \in U'$ and $\xi(0), p \in K$, we have

$$|\xi(\tau) \varphi(\tau, p, u)| \leq \beta(|\xi(0) - p|, u). \quad (3.8)$$

Here, $|x|$ for $x \in \mathbb{R}^n$, denotes the component-wise absolute value. A growth bound can be obtained essentially by bounding the Jacobian of f .

Let $L : U' \rightarrow \mathbb{R}^{n \times n}$ satisfy

$$L_{i,j}(u) \geq \begin{cases} D_j f_i(x, u) & \text{if } i = j \\ |D_j f_i(x, u)| & \text{otherwise} \end{cases} \quad (9)$$

for all $x \in K' \subseteq \mathbb{R}^n$ and $u \in U' \subseteq U$. Then

$$\beta(r, u) = e^{L(u)\tau} r + \int_0^\tau e^{L(u)s} w \, ds, \quad (10)$$

is a growth bound on $[0, \tau]$, K, U' associated with (3.1). The domain K' on which (3.9) needs to hold, is assumed to be convex and contain any solution ξ on $[0, \tau]$ of (3.1) with $u \in U'$ and $\xi(0) \in K$.

In order to use SCOTS, we need to provide a growth bound, which for nonlinear control systems can be provided in terms of the parameterized matrix $L(u)$ whose entries satisfy (3.9).

3.5. Closed Loop.

The solution of a control problem (S, Σ) is a system

$$C = (X_c, X_{c,0}, U_c, V_c, Y_c, F_c, H_c)$$

which is feedback composable with S_1 , so that

$$B(C \times S) \subseteq \Sigma.$$

Here $B(C \times S)$ denotes the behavior of the closed loop $C \times S_1$. The main statement enabling the symbolic synthesis approach reads as follows.

Consider two control problems (S_i, Σ_i) , $i \in \{1, 2\}$. Suppose that Q is a feedback refinement relation from S_1 to S_2 and Σ_2 is an abstract specification of Σ_1 . If C solves the control problem (S_2, Σ_2) , then $C \circ Q$ solves the control problem (S_1, Σ_1) .

The controller $C \circ Q$ for S_1 is given by the serial composition of the quantizer $Q: X_1 \Rightarrow X_2$ with the controller C . The closed loop resulting from a simple system Σ_1 which represents the τ -sampled behavior of (3.1) and a controller $C \circ Q$ is illustrated in Figure 3.2. At each $k \in \mathbb{Z}_{\geq 0}$ sampling time $\tau > 0$, the plant state $x_1 = \xi(k\tau)$ is measured and fed to the quantizer Q , which is used to determine a cell $x_2 \in X_2$ that contains $x_1 \in x_2$. Then x_2 is feed to the controller C to pick the input $u \in U_2 \subseteq U_1$ which is applied to (3.1).

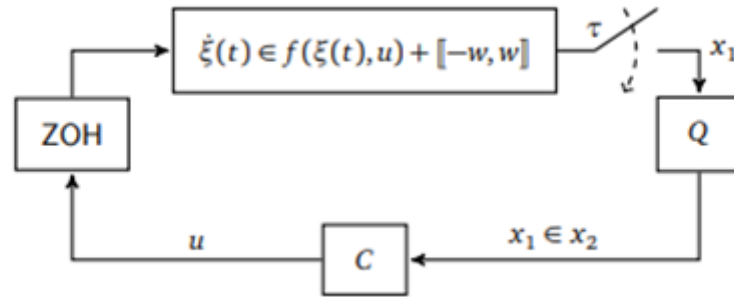


Figure 3.2. Sample-and-hold implementation of a controller synthesized with SCOTS

3.6. Synthesis via Fixed Point Computations.

For the synthesis of controllers C to enforce reachability, respectively, invariance specifications, SCOTS provides two fixed point algorithms.

Consider

$S_2 = (X_2, U_2, F_2)$ with X_2 finite and $I_2 \in X_2$, $Z_2 \subseteq X_2 \times U_2$. For $Y \subseteq X_2 \times U_2$,

we define the map

$$\text{pre}(Y) := \{(x_2, u) \in X_2 \times U_2 \mid F_2(x_2, u) \neq \emptyset \wedge F_2(x_2, u) \subseteq \pi_{X_2}(Y)\}. \quad (3.12)$$

where $\pi_{X_2}(Y)$ denotes the projection of Y onto X_2 .

We compute a controller to enforce a reachability specification Σ_2 associated with I_2, Z_2 , by computing the minimal fixed point of the map $Y \rightarrow \text{pre}(Y) \cup Z_2$, which we denote by using the usual μ calculus notation as

$$\mu Y. \text{pre}(Y) \cup Z_2 \quad (3.13)$$

In order to extract a controller, we introduce the function $j : X_2 \rightarrow \mathbb{N} \cup \{\infty\}$ by

$$j(x) = \inf\{i \in \mathbb{N} \mid x \in \pi_{X_2}(Y_i)\}$$

where the sets Y_i are recursively given by

$$Y_0 = \emptyset$$

$$Y_{i+1} = \text{pre}(Y_i) \cup Z_2.$$

Of course we have $Y_i = Y_{i+1}$ implies $Y_i = \mu Y. \text{pre}(Y) \cup Z_2$. Let us define the map

$$H'_c(x_2) = \{u \in U \mid (x_2, u) \in Z_2 \vee (F_2(x_2, u) \neq \emptyset \wedge F_2(x_2, u) \subseteq \pi_{X_2}(Y_{j(x_2)-1}))\}$$

which is non-empty for all $x_2 \in \mu Y. \text{pre}(Y) \cup Z_2$. We derive a controller as by

$C = (\{q\}, \{q\}, X_2, X_2, U_2, F_c, H_c)$ with

$$H_c(q, x_2) = \begin{cases} H'_c(x_2) \times \{x_2\} & \text{if } x_2 \in \mu Y. \text{pre}(Y) \cup Z_2 \\ U_2 \times \{x_2\} & \text{otherwise} \end{cases}$$

$$F_c(q, x_2) = \begin{cases} \{q\} & \text{if } x_2 \in \mu Y. \text{pre}(Y) \cup Z_2 \\ \emptyset & \text{otherwise} \end{cases}$$

Similarly, if Σ_2 is an invariance specification associated with I_2, Z_2 , we compute the maximal fixed point of $Y \rightarrow \text{pre}(Y) \cap Z_2$, which is denoted by

$$\nu Y. \text{pre}(Y) \cap Z_2.$$

Given $\nu Y. \text{pre}(Y) \cap Z_2$ we define the map

$$H'_c(x_2) = \{u \in U \mid F_2(x_2, u) \neq \emptyset \wedge F_2(x_2, u) \subseteq \pi_{X_2}(\nu Y. \text{pre}(Y) \cap Z_2)\} \quad (3.14)$$

and the controller follows again by

$$H_c(q, x_2) = \begin{cases} H'_c(x_2) \times \{x_2\} & \text{if } x_2 \in \nu Y. \text{pre}(Y) \cap Z_2 \\ U_2 \times \{x_2\} & \text{otherwise} \end{cases}$$

$$F_c(q, x_2) = \begin{cases} \{q\} & \text{if } x_2 \in \nu Y. \text{pre}(Y) \cap Z_2 \\ \emptyset & \text{otherwise} \end{cases}$$

In either case, it is well known that C solves the control problem (S_2, Σ_2) with Σ_2 being a reachability (invariance) specification iff $I_2 \subseteq \pi_{X_2}(\mu Y. \text{pre}(Y) \cup Z_2)$

$(I_2 \subseteq \pi_{X_2} (\nu Y. \text{pre}(Y) \cap Z_2))$. Also for both types of specifications the controller is memoryless or static, i.e., the output is independent of the state.

3.7. Conclusion

Scots is a powerful tool for the synthesis of symbolic controller for perturbed control system. SCOTS begins by lifting the control system to an abstract domain by discretizing the state space given a sampling time τ and applying a constant input. This results into a simple system, with states x and a transition relation F , representing the τ -sampled behavior of concrete systems. SCOTS then, solve the abstract control problem using fixed point computation and finally refine the synthesized controller to match the concrete system.

Chapter 4.

Implementation of symbolic controller on FPGAs

4.1. Introduction

In order to test the controller, we used various techniques of simulation. We began first by doing software in the loop technique, where we simulated the dynamics and the controller using the MATLAB interface. Then in a second step we have used hardware in the loop simulation, where we integrated the dynamics into a hardware simulator called OP4200. Finally, we implemented our controller into an FPGA to get the control input and fed these inputs in the dynamics using a MATLAB interface to check the correctness of the synthesized controller.

4.2. Simulation on opal 4200

The OP4200 RCP/HIL system is a desktop-friendly package that offers Hardware-in-the-Loop (HIL), Rapid Control Prototyping (RCP), data acquisition and I/O expansion capabilities. Power electronics, automotive and other real-time applications for industry and academia are supported by OP4200.

OP4200 contains a Xilinx Zynq® All Programmable SoC, featuring a Kintex™7 FPGA and dual-core ARM® Cortex processor, and the same class-leading FPGA-based I/Os and real-time solvers provided throughout OPAL-RT product line, making it the ideal choice for closed-looped applications.

Hardware-in-the-loop (HIL) simulation, is a technique that is used in the development and test of complex real-time embedded systems. HIL simulation provides an effective platform by adding the complexity of the plant under control to the test platform. The complexity of the plant under control is included in test and development by adding a mathematical representation of all related dynamic systems. These dynamics are referred to as the “plant simulation”. The embedded system to be tested interacts with this plant simulation.

Electrical emulation of sensors and actuators must be included in the HIL simulation. These electrical emulations act as the interface between the plant simulation and the embedded system under test. The value of each electrically emulated sensor is controlled by the plant simulation and is read by the embedded system under test (feedback). Likewise, the embedded system under

test implements its control algorithms by outputting actuator control signals. Changes in the control signals result in changes to variable values in the plant simulation.

4.3. OPAL-RT and test unicycle dynamics in Simulink (Hardware in the loop simulator)

The next test scenario is to run the controller using a MATLAB script (computer-side) and to process the unicycle dynamics in SIMULINK before running it directly on the HW simulator. We test the unicycle dynamics model on the HIL (Opal-RT) with different valid inputs to verify the model.

OP4200 which we use is illustrated in figure 4.1. It has 64 Analog I/O and 128 digital I/O. It has ETHERNET and Jtag peripherals. For further specification, its toturials is on https://www.opal-rt.com/wp-content/themes/enfold-opal/pdf/L00161_0519.pdf.



Figure 4.1. OP 4200

RT-LAB is OPAL-RT's real-time simulation software which combines performance and enhanced user experience. Since Opal-RT works with

MATLAB Simulink, we first model the whole system including controller and dynamic system in MATLAB with an embedded controller.

Quick start with OPAL-RT

1. We install RT-LAB together with the corresponding compatible MATLAB version (Have a look to the RT-LAB guide), for example RT-LAB 11.1.8 is compatible with MATLAB 2017a
2. We open OPAL-RT and wait for software to recognize the hardware
3. We have to configure it's IP address correctly if OPAL-RT doesn't recognize the OP4200, .
4. We build a new project
5. We build the Simulink model for unicycle dynamics system based on differential equations with off-the-shelf blocks in Simulink (more details are at **test_vehicle_dynamics.slx** in GITLAB)
6. We modify the Simulink model so that we exchange the output with scopes and then regroup them with the inputs into one subsystem.
7. We regroup the remaining computation system into another subsystem
8. We rename the newly created core blocks with SC_(console display) and SM_(process) in Simulink.
9. The top level should look like in figure 4.2.

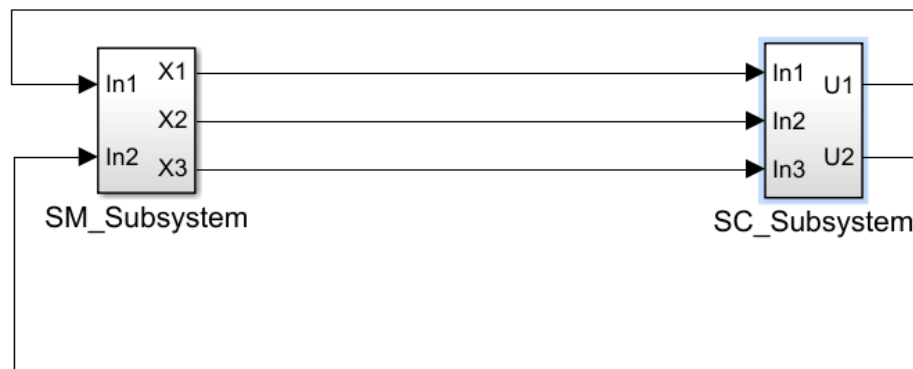


Figure 4.2. Top level representation of the RT-Lab simulink model

10. We add a model to your OPAL-RT project and we copy the created subsystem into the model
11. We configure the model, $[U_1, U_2]$ should be in this range $[-1 \ 1]$. $[X_1, X_2, X_3]$ can be any real number.
12. We add OP-COMM block to input in SC and SM from the RT-LAB library in SIMULINK like in the figure 4.3 (here it is the SC subsystem)

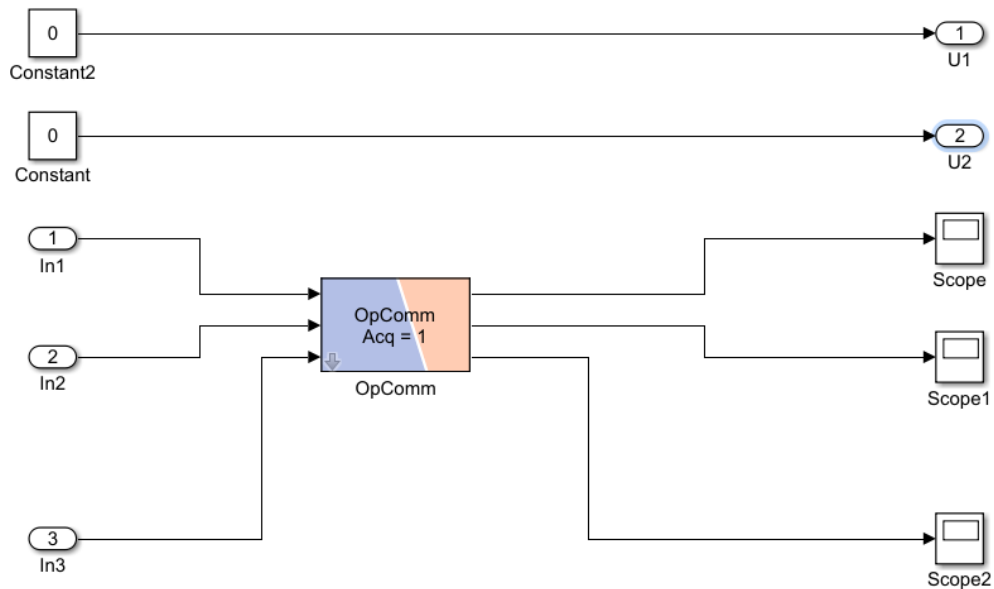


Figure 4.3. Comsol Subsystem

13. We set solver in Rang-kutta Model configuration parameter in Menu bar in SIMULINK. Time-step should be 0.3 s.

14. We build Model and Load it and run it

Here we are just testing the response of the simulator giving the dynamics of the unicycle.

4.4. Software In the Loop Simulation of the synthesized controller

In this section, we simulate the controller generated from SCOTS and dynamics of the unicycle model all by using Simulink, like the implementation in vehicle_simul.mat. To do it, we complement the vehicle.mat file and we implement dynamics system in ODE function as follows(the whole code is depicted in Appendix 1):

1. We build function $[t \ x] = \text{ode}(t, Y, U)$ (We took as example the ode function in vehicle_simul.mat here presented in Appendix 1)
2. We build Simulink model for unicycle system
3. We run simulation with sim function
4. We verify that Cudd and mexfile library are functional
5. We configure below parameter for simulation
`set_param(model, 'AbsTol', '1e-6', 'StopTime', '0.3')`
6. Initial value of the unicycle trajectories should be set in the simulink model

Here, the controller gets input from the SIMULINK and result in controller output by reading the BDD-file. The simulation parameter can be modified in ODE function. The while loop continues till it reaches the target. Finally, a vehicle path through the obstacles is displayed.

4.5. Hardware in the Loop Simulation of the synthesized controller

In order to search input and output inside the BDD synthesized by SCOTS, the trajectories must be converted to new values with an abstraction operation. This helps the search function to find respective output conveniently.

We come up with an indexing algorithm using abstraction. We use an embedded c function in SIMULINK to model a controller that can be run on Rtlab. We have to convert BDD file to mat file which is done with bdd.mat. By abstraction of trajectories, a unique index is generated for every valid trajectory. The respective controller's values are stored in controller.mat. To generate an index, the following functions are carried out:

Quantization:

$$(X, Y, \theta) \rightarrow (X', Y', \theta') = \frac{(X-X_0)}{\tau_0}, \frac{(Y-Y_0)}{\tau_1}, \frac{(\theta-\theta_0)}{\tau_2}$$

Indexing:

$$Index = \theta' \times 51^2 + Y' \times 51 + X'$$

where, τ_i are quantization steps. X_0, Y_0 and θ_0 are minimum values of X, Y and θ .

Note that based on the quantization parameters used in this example, we have 51 values at both X, Y, and θ dimensions.

Assuming each control action needs 8 bytes (as an integer in C/C++), the LUT needs $(51^3) \times 8$ byte spaces in memory.

To build matfiles:

- As reading BDD files are not possible on SIMULINK, all the information, including states and respective control actions, are transferred into a

MATFILE called controller.mat, controller1.mat and controller2.mat using bdd.mat (appendix 2).

- bdd.mat loads the bdd file(vehicle_controller.bdd) and put its data in controller
- The applied abstraction is $(X_0, Y_0, \theta_0, \tau_0, \tau_1, \tau_2) = (0, 0, -3.4, 0.2, 0.2, 0.2)$ on all rows in controller
- Indexing is also applied on all rows
- Controller1.mat includes first control unit and its index
- Controller2.mat includes second control unit and its index

4.6. Testing the whole system (Controller and Dynamics system) on Simulink

We tried to implement the simulation in Simulink (simul_function_2015_new_matfunctio.slx) with an interpreted function (getControlandstoremat.m) as controller in MATLAB. To run the whole system in SIMULINK, the MATLAB embedded function is used in SIMULINK model

- We load controller.mat
- We get state in interpreted function (getControlandstoremat.m)
- We apply Abstraction on state
- We run simulation

4.7. Testing the whole system (Controller and Dynamics system) on RT-Lab

In Rt-lab, the embedded function has to be written in c (sfunction_vehicle.c). To use the controller in a C script, the controller outputs should be copied in two separate vectors called controller1 and controller2. To build function written in sfunction_vehicle.c:

- We construct the controller block which has three inputs and two outputs using matlab s-function block (level 1) in SIMULINK
- The control units are written in different vectors in c file
- The controller input and output dimension and number of them are defined in method:
static void mdlInitializeSizes(SimStruct *S)
- The main algorithm is written in:
static void mdlOutputs(SimStruct *S, int_T tid)
- When the controller gets inputs, the abstraction and indexing respectively are carried out on it. Then the value based on the index is written on two outputs of the controller block.

Next, a block containing the dynamics of the unicycle model is created and related to the sfunction block.

To run the simulation on Opal-rt:

- We open opal-rt and wait for software to recognize the hardware
- We build a new project
- We add a new model in the project and we copy the constructed block into the model by editing it
- We modify the model in Simulink. Then, rename the core blocks with SC_(console display) and SM_(process) in Simulink
- SM Contains our dynamics block and SC contains our sfunction
- We add op-comm block to input in SC and SM
- We set solver in Rang-kutta Model configuration parameter in Menu bar in SIMULINK. Time-step should be 0.3 s.
- the sfunction_vehicle.c should be added to the **Files** tab in OPal-rt. Transfer mode must be **ASCII** and transfer time (aka Step) must be **Before Compilation**.
- We Build the Model, Load it and run it

4.8. Hardware in a loop with Zedboard

The real Hardware implemented controller is tested in HIL (Hardware in the loop). In this part, the Controller's implementation is generated as a VHDL module and implemented on a Zedboard. The VHDL module is encapsulated as IP-Core called AXI Slave. A ZYNQ processor system interfaces this IP-Core. The ZYNQ runs under an Embedded Linux system. Finally, a Linux application is used to provide networked hardware-in-the-loop (HIL) simulation server.

BDD2Implement, [12], is a tool to automatically generates VHDL codes for the symbolic controller. BDD2Implement is a C++ tool to generate hardware/software implementations of BDD-based symbolic controllers. Having the tools SCOTS that generate BDD-based symbolic controllers of general nonlinear dynamical systems, BDD2Implement completes missing ring in the automatic synthesis technique. BDD2Implement can generate codes in the following formats VHDL/verilog module or in C/C++. The BDD2Implement structure is represented in figure 4.4.

It starts by converting the multi-output boolean functions inside BDDs to multi singel-output functions. If the provided controller is not determinized, BDD2Implement provides a determinization of the controller using several possible determinization methods. For VHDL, the boolean functions are dumped to the VHDL module which contains the boolean functions as maps from input-port to output-port.

We build the example (ex1-vhdl-raw) corresponding to a VHDL implementation of a controller for the unicycle model. The output file is bdd_vehicle.vhdl.

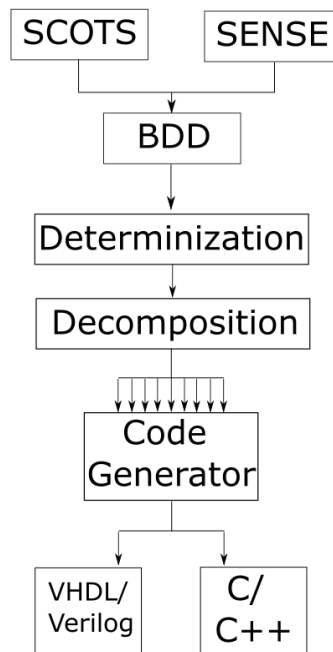


Figure 4.4. Schematic of the BDD2implementation tool

In the next stage, the `vhdl_wrapper` has to be implemented in Vivado to incorporate an IP on FPGA.

4.8.1. Axi protocol

We have decided first to focus on a very basic concept which is connectivity. In fact, the question here is how different modules inside our embedded system will talk to each other. We will be then able to design the embedded system in a very short amount of time.

In order to solve the connectivity problem, we need a kind of a standard so that all of our units can talk based on this standard. So, there should be a kind of unique language through which all of the units can talk.

So, if one module wants to perform a write operation it should obey the rules of this standard. This standard is in fact ways of signaling through which different modules inside our chip will talk to each other.

Arm has developed a set of system-on-chip and they have extended and enhanced these system-on-chip protocols through time. AXI is one of them. AXI

is a kind of protocol; it is a kind of way of signaling through which our modules can talk to the outside world and when we design our module, we will obey the rules of AXI.

These AXI Modules will initiate transactions. Transactions are basically operations through which data is transferred from one point of our chip to another point. For example, data is transferred from the memory inside our chip to one module or vice versa. The transactions in these cases can be either read transactions or write transactions. The module that starts the transaction is called an AXI-master.

The module which will be the target of the transaction is called an AXI-slave. Suppose that we have two modules one of them will be an AXI-master and the other will be an AXI-slave, for example AXI-master can be a CPU, and AXI-slave can be a block of memory. The CPU initiates read and write transactions to the memory, the memory responds to these read and write transactions. When we have write transactions the flow of data will be from the CPU to the memory, and when we have read transactions the flow of data will be from the memory to the CPU. Whenever an AXI-master needs to perform a transaction, it sends a set of commands and a set of initial information about the transaction that is going to happen inside the AXI-slave. For example whenever the AXI-master wants to perform a write transaction or a read transaction it is necessary that the AXI-master sends to the AXI-slave the address for this read or write transaction, and then when the AXI-slave is performing the transaction and is doing the transaction and is providing the data if it is the read or accepting the data if it is the write then the AXI-slave should produce the suitable response to the AXI-master.

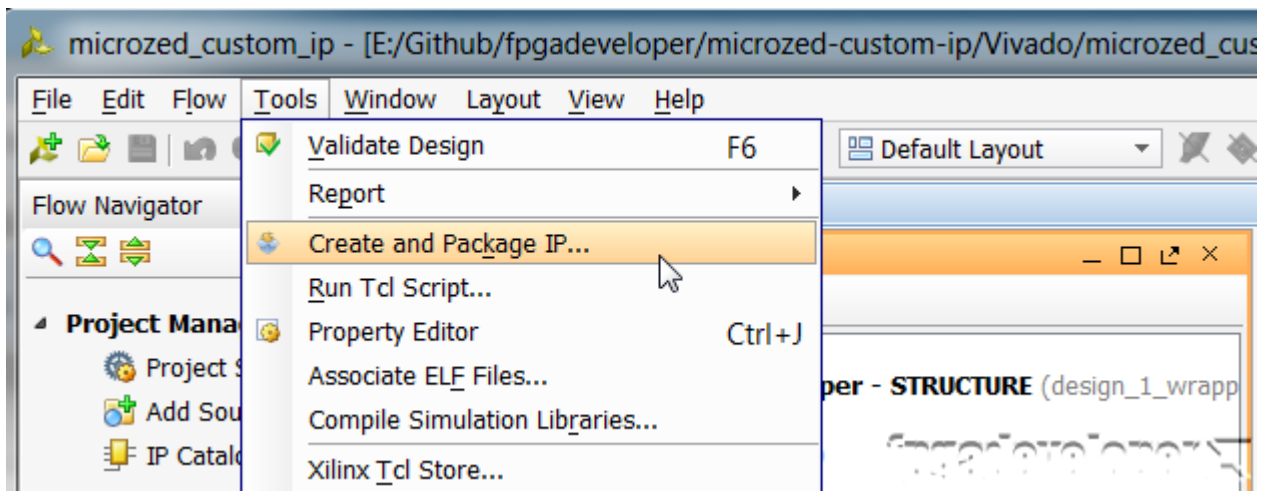
A connection between an AXI-slave and an AXI-master has in fact five basic components. These are called channels and each channel contains a set of signals. For example, whenever the AXI master wants to perform a write transaction to the AXI-slave, this write operation happens through three channels. First the AXI-master through the write address channel sends the write address to the AXI-slave, then through the write data channel sends the data to be written to the AXI-slave. Finally, the AXI-slave through the write response channel responses to the master, if the write operation is done successfully. For the read operation it is the same but with read channel.

As in this thesis we do need an AXI-slave through which the CPU can perform a read transaction in order to get the control input from the VHDL computation. For that purpose, we need to design an IP core.

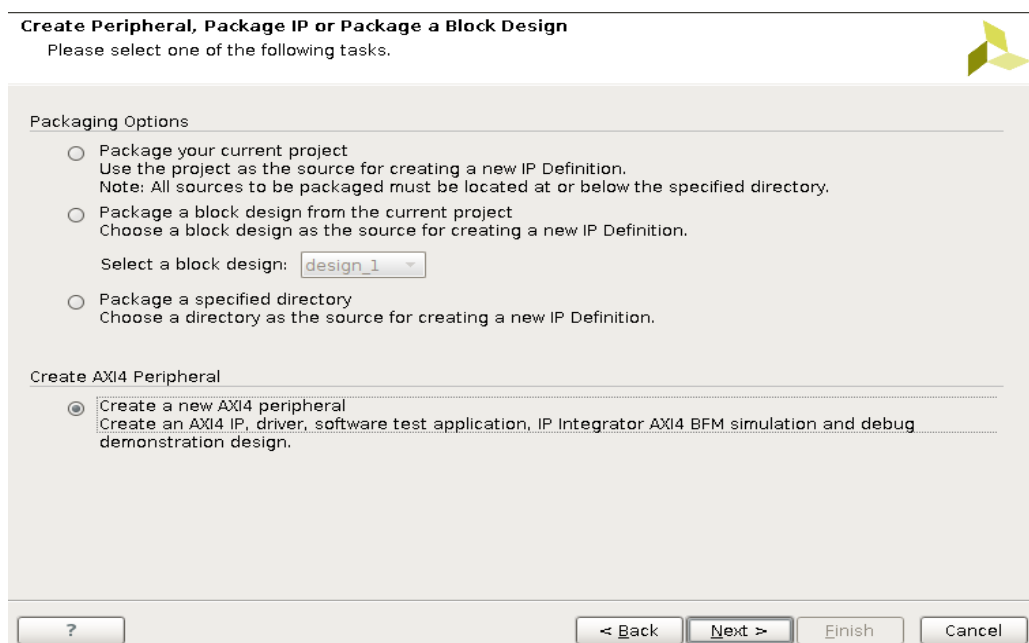
4.8.2. Design of the IP core

To make an IP based core:

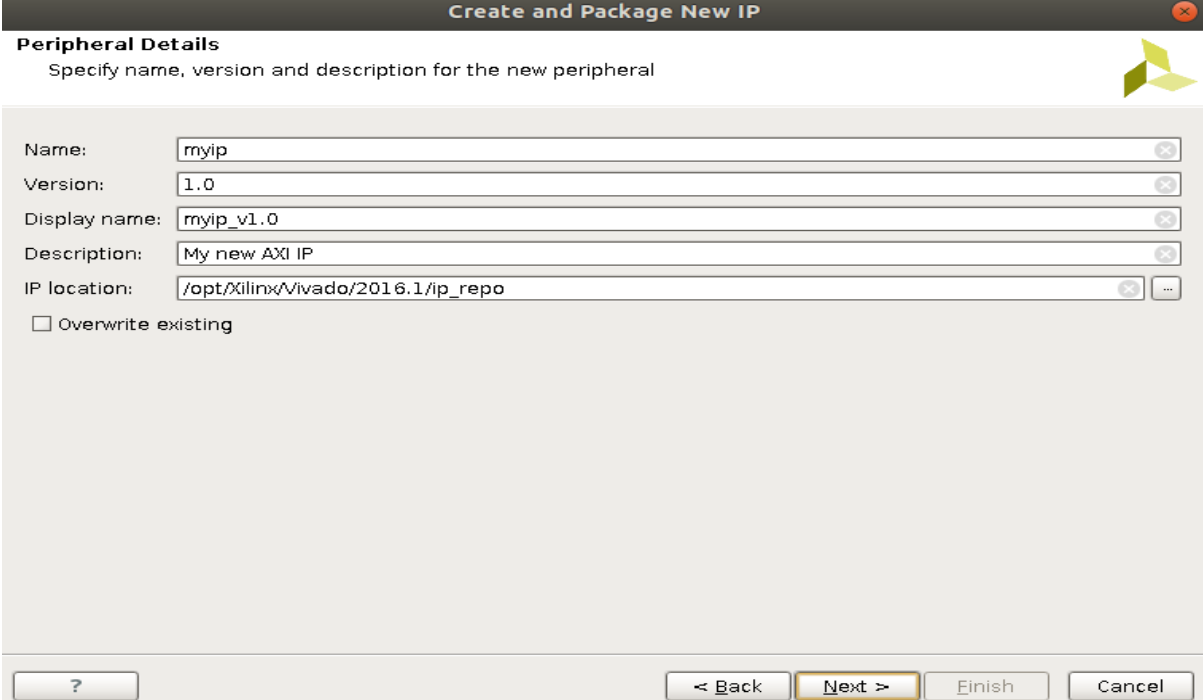
1. We create new project for zedboard
2. We click into project Settings and we choose VHDL as the target language
3. We create a block design
4. We add zynq processor
5. We click the Run Block Automation link
6. With the base Vivado project opened, from the menu, we select Tools->Create and package IP.



7. The Create and Package IP wizard opens we click next.
8. On the next window, we select “Create a new AXI4 peripheral”. We click “Next”.

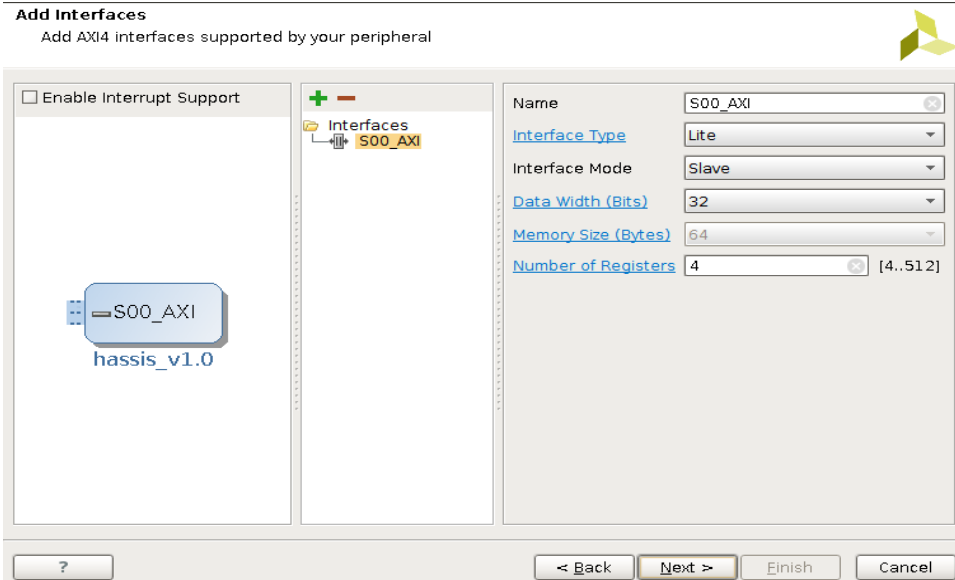


9. Now we can give the peripheral an appropriate name, description and location. We click “Next”.



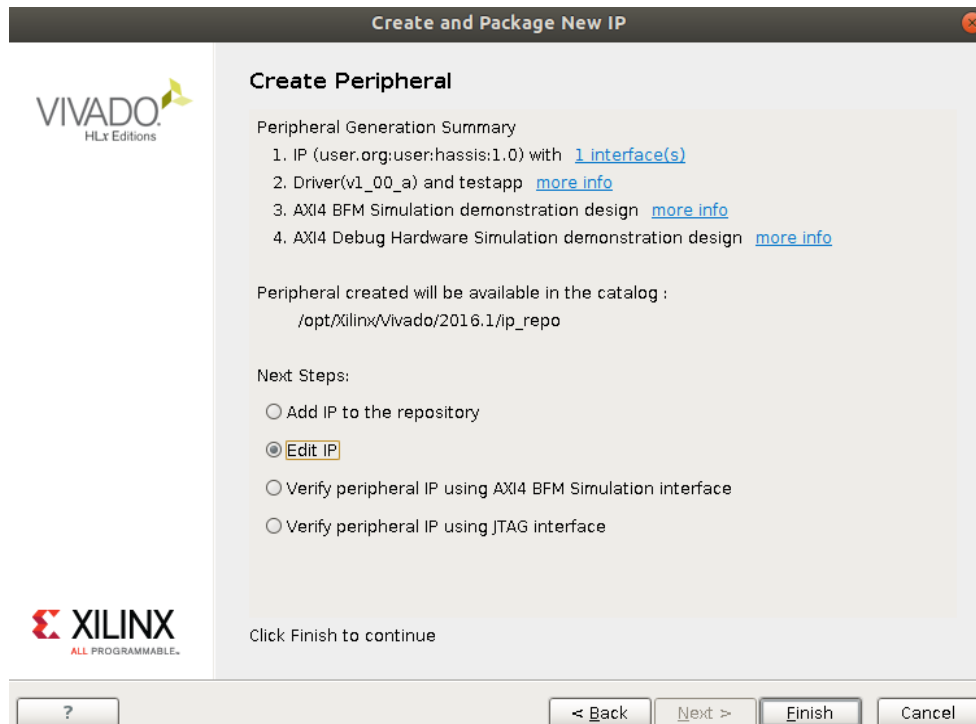
The image shows the "Create and Package New IP" dialog box in Vivado. The title bar is "Create and Package New IP". The main section is titled "Peripheral Details" with the subtitle "Specify name, version and description for the new peripheral". There are five input fields: "Name" (myip), "Version" (1.0), "Display name" (myip_v1.0), "Description" (My new AXI IP), and "IP location" (/opt/Xilinx/Vivado/2016.1/ip_repo). There is an "Overwrite existing" checkbox which is unchecked. At the bottom, there are buttons: "?", "< Back", "Next >", "Finish", and "Cancel".

In the next page we can configure the AXI bus interface. We'll use AXI lite, and it'll be a slave to the PS, so we'll stick with the default values



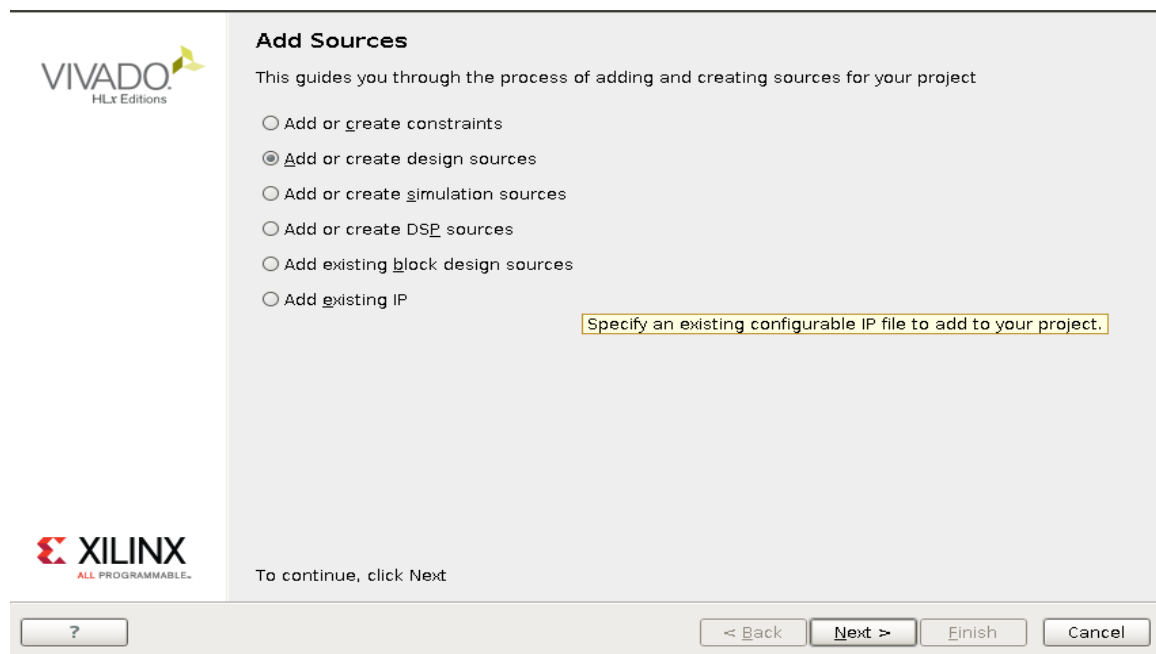
The image shows the "Add Interfaces" dialog box in Vivado. The title bar is "Add Interfaces" with the subtitle "Add AXI4 interfaces supported by your peripheral". There is an "Enable Interrupt Support" checkbox which is unchecked. On the left, there is a diagram showing a block labeled "S00_AXI" with the text "hassis_v1.0" below it. In the center, there is a tree view showing "Interfaces" with "S00_AXI" selected. On the right, there are configuration fields: "Name" (S00_AXI), "Interface Type" (Lite), "Interface Mode" (Slave), "Data Width (Bits)" (32), "Memory Size (Bytes)" (64), and "Number of Registers" (4) with a range of [4..512]. At the bottom, there are buttons: "?", "< Back", "Next >", "Finish", and "Cancel".

10. On the last page, we select “Edit IP” and we click “Finish”. Another Vivado window will open which will allow us to modify the peripheral that we just created



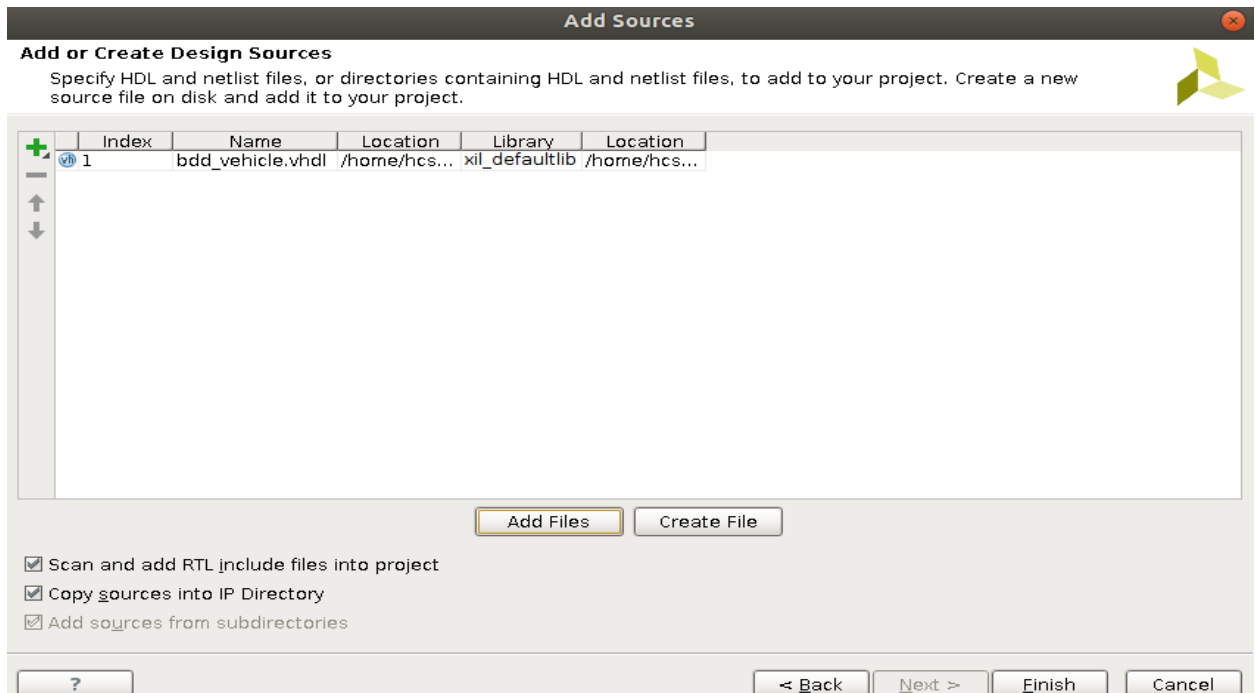
11.From the Flow navigator, we click “Add Sources

12.In the window that appears, we select “Add or Create Design Sources” and we click “Next”.



13.On the next window, we click “Add Files”.

14.We Browse to the “bdd_vehicle.vhd” file, we select it andwe click “OK”.



15. We double click on the “name_of_axi_v1_0_S00_AXI_inst” file to open it.
16. The source file should be open in Vivado. We find the line with the “begin” keyword and add the following code just above it to declare the BDDWrapper_entity and the output signal

```

signal bdd_out : std_logic_vector(31 downto 0);
component BDDWrapper_entity
port(
x0: in std_logic;
x1: in std_logic;
x2: in std_logic;
x3: in std_logic;
x4: in std_logic;
x5: in std_logic;
x6: in std_logic;
x7: in std_logic;
x8: in std_logic;
x9: in std_logic;
x10: in std_logic;
x11: in std_logic;
x12: in std_logic;
x13: in std_logic;
x14: in std_logic;
x15: in std_logic;
x16: in std_logic;
x17: in std_logic;

f0: out std_logic;
f1: out std_logic;
f2: out std_logic;
f3: out std_logic;
f4: out std_logic;
f5: out std_logic
);
end component;

```

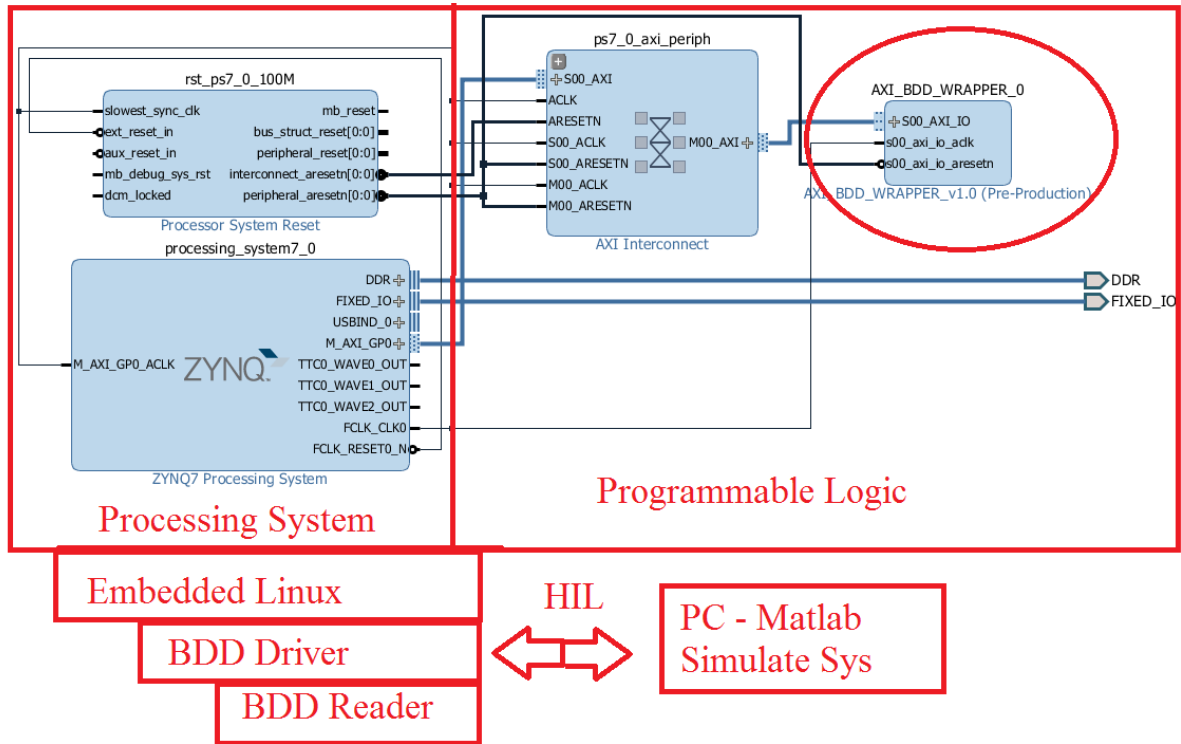
17. The line that says “– Add user logic here” is fined, we add the following code below it to instantiate the BDDWrapper_entity

```

-- Add user logic here
bddwrapper_0 : BDDWrapper_entity
port map (
    X0 => slv_reg0(0),
    X1 => slv_reg0(1),
    X2 => slv_reg0(2),
    X3 => slv_reg0(3),
    X4 => slv_reg0(4),
    X5 => slv_reg0(5),
    X6 => slv_reg0(6),
    X7 => slv_reg0(7),
    X8 => slv_reg0(8),
    X9 => slv_reg0(9),
    X10 => slv_reg0(10),
    X11 => slv_reg0(11),
    X12 => slv_reg0(12),
    X13 => slv_reg0(13),
    X14 => slv_reg0(14),
    X15 => slv_reg0(15),
    X16 => slv_reg0(16),
    X17 => slv_reg0(17),
    f0 => bdd_out(0),
    f1 => bdd_out(1),
    f2 => bdd_out(2),
    f3 => bdd_out(3),
    f4 => bdd_out(4),
    f5 => bdd_out(5));
-- User logic ends

```

18. We find this line of code “reg_data_out <= slv_reg1;” and we replace it with “reg_data_out <= bdd_out;”.
19. In the process statement just a few lines above, we replace “slv_reg1” with “bdd_out”
20. We save the file
21. We should notice that the “bddvehicle.vhd” file has been integrated into the hierarchy because we have instantiated it from within the peripheral
22. In the “Packaging Steps” we go to “Files Group” and we click “Merge changes from File Groups Wizard”
23. We go to Review and Package and we click on “Re-Package IP”
24. In the diagram we click on add IP and we select the newly created AXI4
25. We run connection automation
26. In the Sources right we click the “design_1” (Or the chosen name for our design) under Design Sources and we click on “create HDL wrapper”



We should now synthesize, implement and build the new design to generate Bit stream and HDF file. Finally we should export the design to Hardware. We can then use these files to generate a new Petalinux project.

4.8.3. Implementing a C application with Petalinux

The PetaLinux Software Development Kit (SDK) is a Xilinx development tool that contains everything necessary to build, develop, test and deploy Embedded Linux systems.

To run Petalinux on Zedboard, we do it as follow

1. We run the command `Source petalinux-installation-path/settings.sh`
2. We create a new petalinux project with the command `petalinux-create --type project --template zynq --name <name of project>`
3. We go to the file containing the HDF file given from the design that we used inside vivado and we import it to our newly created project with the command `petalinux-config --get-hw-description -p <plnx-proj-root>`
4. We create a C application template, we pass the `--template c` option, as follows: `petalinux-create -t apps --template c --name bddhil`
5. We change to the newly created application directory in `/<project name>/components/apps/bddhil`

6. We copy the bddhil code (depicted in appendix 3), the bddReader header file (depicted in appendix 4), the make file as also the apps.common.mk to the bddhil application. Those files are in the gitlab under documentation [14].
7. In order to use the bddhil application, we have to configure our project we use the command `petalinux-config -c rootfs` and then under apps check whether the bddhil is selected with * if not select it
8. We run the command `petalinux-build`
9. We run `petalinux-build -c rootfs/bddhil -x build`
10. We use the `petalinux-package` command to generate a `BOOT.bin` file containing the first stage bootloader, U-BOOT and the bitstream. For that run the command `petalinux-package --boot --fsbl ../zynq_fsbl.elf --fpga ../name.bit --uboot`
11. We copy the `boot.bin` and the `image.ub` together with the bddhil executable (bddhil executable is in rootfs folder)

4.8.4. Result

For running the new `vehicle_hil_simulation` on zedboard:

1. We connect to Fpga linux with UART
2. We insert the SD-card
3. We power on the fpga
4. We launch minicom after configuring it to connect to `ttyACM0`
5. We write the following command in order to allow the fpga to boot from the SD-card instead of the system memory
`U-Boot-PetaLinux> setenv bootcmd 'run sdboot'; saveenv`
6. As Login and password we write root
7. We launch the bddhil application this will respond with HIL STARTED
8. We run the command `exit` and we close the terminal
9. We run `vehicle_hil_simulation.m` in matlab
10. Display of the result (Figure 4.5)

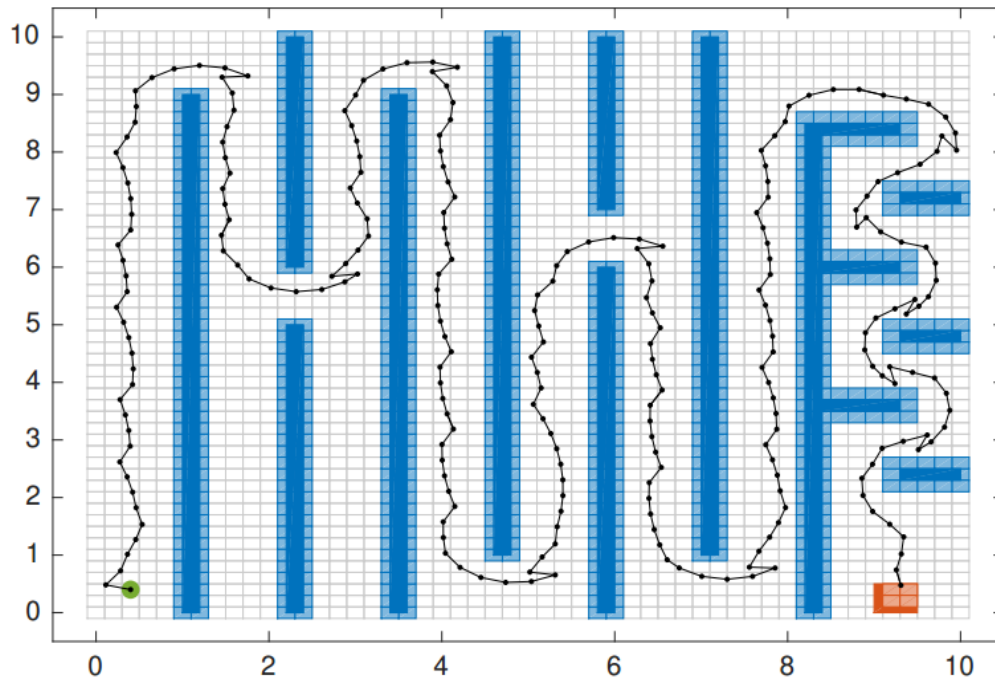


Figure 4.5. Path of the vehicle, modeled by a unicycle model, inside the state space

4.9. Conclusion

First and using mainly software in the loop technique, the controller was tested.

Second the controller inputs were obtained from the function `getInputs`, and then were fed to the dynamics that were simulated using the hardware simulator.

Finally, the controller was implemented on the FPGA, where we substituted the controller into a VHDL code and then integrated it in an AXI peripheral. This AXI serves as interface between the computer and the FPGA to get the control input given the state.

Chapter 5.

Implementation of the control over Raspberry pi

In this chapter the main goal is to implement our controller in the Raspberry Pi 3 shown in figure 5.1. The controller is generated as C++ code by help of the various functions of the SymbolicSet library. We will first introduce the Raspberry Pi then we will go through the C code and the function that we used to interact with the synthesized controller.

5.1. THE RASPBERRY PI 3

The Raspberry Pi 3 Model B is the latest version of the Raspberry Pi computer. In its cheapest form it doesn't have a case, and is simply a credit-card sized electronic board of the type you might find inside a PC or laptop but much smaller.

5.2. THE RASPBERRY PI 3 CAPABILITIES

The Pi 3 can be used as a budget desktop, media center, retro games console, or router for starters. The Pi can be used to build tablets, laptops, phones, robots, smart mirrors, to take pictures on the edge of space, to run experiments on the International Space Station and that's without mentioning the wackier creations

One thing to bear in mind is that the Pi by itself is just a bare board. We will need a power supply, a monitor and a mouse and a keyboard.

The installer makes it simple to set up various operating systems, we choosed to install Ubuntu mate because it was a lot easier to install SCOTS and set up the Cudd library

The Pi 3 has the following component:

- Four USB 2.0 ports (up to 480 megabits per second)
- HDMI port
- 3.5mm 4-pole Composite Video and Audio jack
- MicroUSB Power Input
- DSI Display Port
- CSI Camera Port
- MicroSD card Sold
- 40-pin GPIO (Male headers)

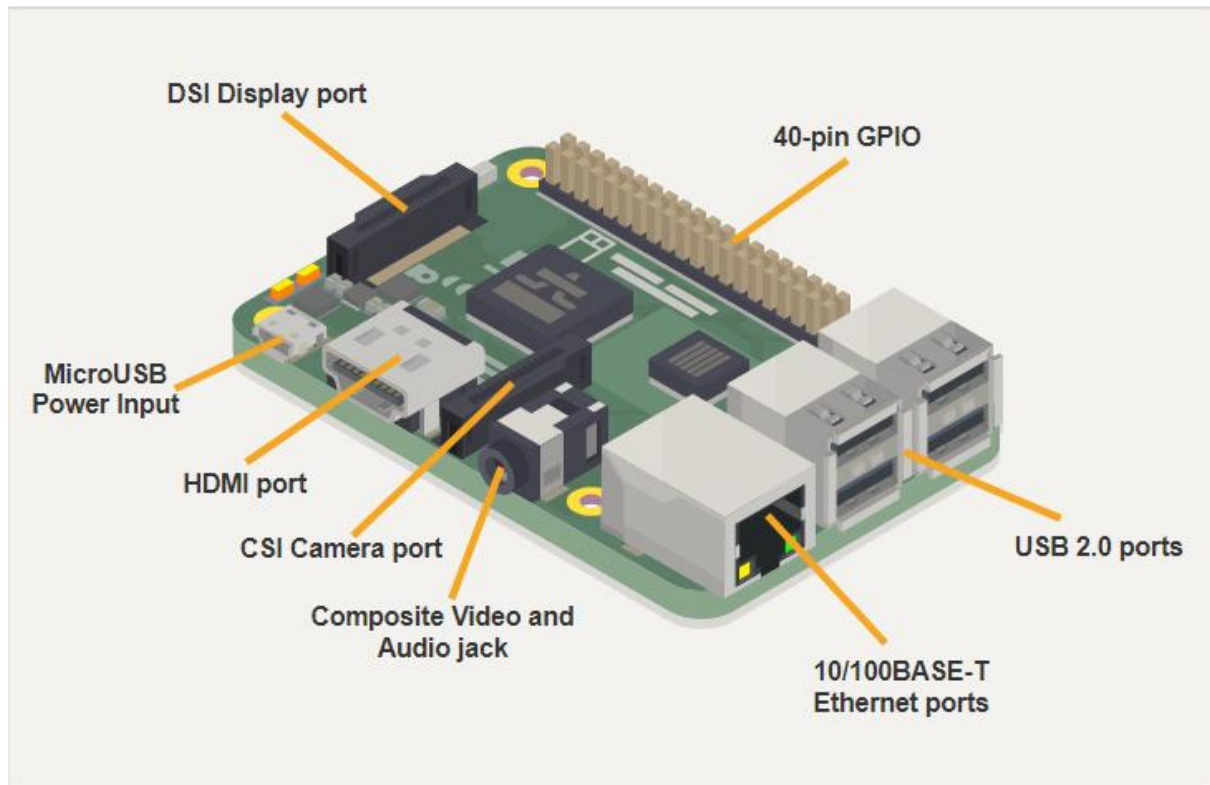


Figure 5.1. Connexion and expansion

5.3. THE C++ Code

As we tested our controller with the MATLAB interface where we used special MATLAB functions like `getinput` and `isElement` in order to interact with the controller. These MATLAB functions are supported from the SymbolicSet Library. In the C++ code we will also use these functions.

These functions are called `SetValuedMap` and `IsElement`. That's all we need to test our controller. The dynamics of the unicycle model has first to be set, using an `ode_solver` to solve the computation of the dynamics as we have done it with SCOTS. After setting up the dynamics, we can then make a while loop, where we check whether the current state is in the target set or not and that with the `IsElement` function. If it is the case, we close our while loop, if not we get our input from controller using the `setValuedMap` function and then re-compute the dynamics to get the new state given the new input.

Practically, the `IsElement` function is a Boolean function that is part of the `symbolicSet` header file. It accepts as input a vector representing the current state and a bdd file, and then returns as output `true` if this state is part of the `symbolicset` or return `false` if this isn't the case. The `isElement` function begin by

computing a bdd representation of the given state. Then through an indexing algorithm searches the index of the bdd cube. The algorithm consist in substracting each dimension of the given x from the minimum value then it's division by our quantization step. Finally we check if this index is inside the loaded bdd file.

The `setValuedMap` is also another function of the `symbolicSet` header files that returns a vector of vector given as input a vector representing the state and also a vector of indices representing the dimension of this state.

The whole step should look this way

- We load our target set and store it in the `symbolicSet` `ts` (`mgr` is our CUDD manager)

```
scots::SymbolicSet ts (mgr,"vehicle_target.bdd");
```

- We also load our controller and store it in the `symbolicSet` `control`

```
scots::SymbolicSet control (mgr,"vehicle_controller.bdd");
```

- We then check whether our current state `x` is inside this target set

```
while(!ts.isElement(x)){
```

- If it is the case the while loop ends if not we get the input from the controller as follow

```
std::vector<std::vector<double>> u = control.setValuedMap(x,ind);
```

- The `setValuedMap` takes as input the state and an index showing the dimension of the state space: For example if it is three dimensional our index should be {0, 1, 2}
- Finally we run the function that computes the dynamics given the input `u` and the state `x`

5.4. Conclusion

Using C++ and by help of the various functions of the `SymbolicSet` library, the controller is here implemented over the Raspberry Pi 3.

After introducing the Raspberry Pi, the interaction with the synthesized controller is done through the C code functions.

Chapter 6.

Robot Control

In this chapter, we test our controller on a real robot given an urban like environment as depicted in Figure 6.1. As a first part, the controller will be synthesized then implemented in the form of a C code. This C code is given the specification and the state space and it will generate inputs to the robot in order to steer it toward the goal while avoiding obstacles. As a second part, the controller will be synthesized online while moving the robots giving a dynamically changing obstacle. Two robots can then reach their respective targets while avoiding each other.

In this chapter we will relate about this and finally we will see the result on the real platform.

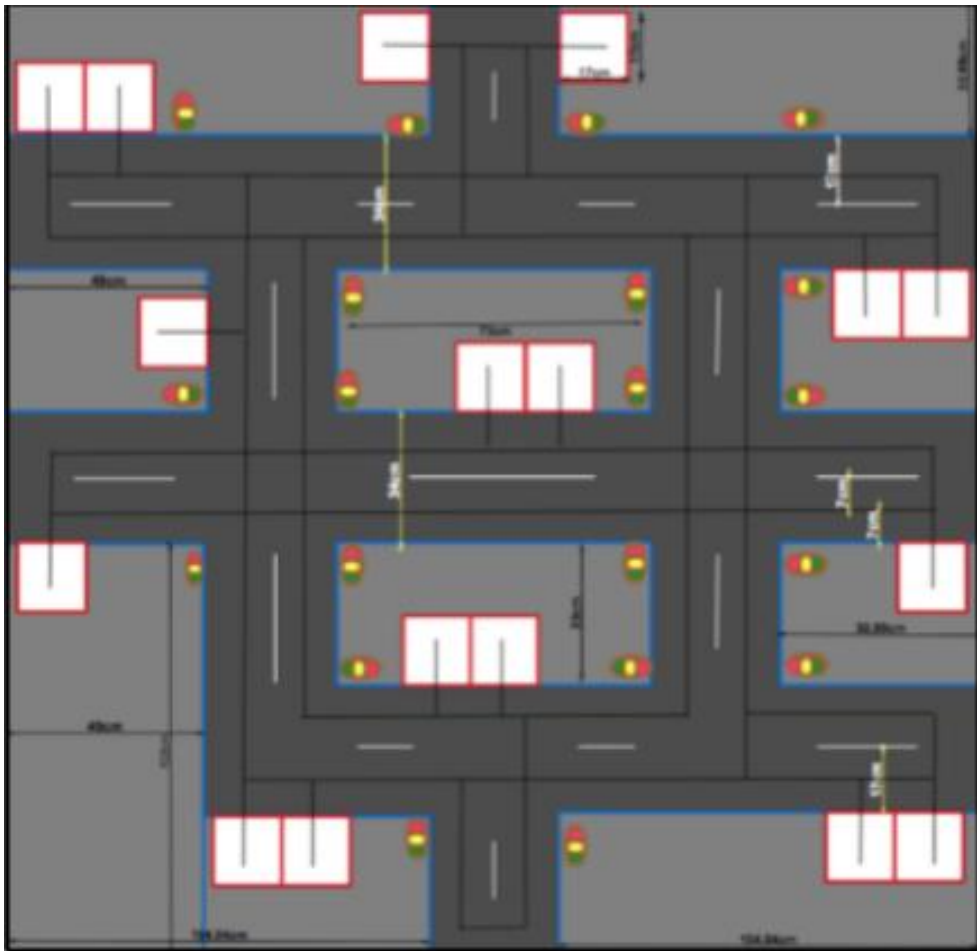


Figure 6.1: The urban like Environment

6.1. The State space

Our robots are called Khepera IV. They represent real moveable cars as depicted in figure 6.2 and we have a platform representing an urban like environment where the robots can move. The platform has crossing roads, traffic lights and parking gates.



Figure 6.2 Robot Khepera IV

The Khepera IV has two wheels to make a transition or a rotation. If it is a simple transition the two wheels turn at the same time in the same direction. If it is a rotation each wheel will turn in the opposite direction of the other. The Khepera IV has also a laser to detect a given line given its color. For that purpose, our robot can move from one state to another by following the line. The high-level model of the platform looks like in the figure 6.3. This high-level model shows the possible transition between the states, and in our controller synthesis we will also provide SCOTS with this model as state space.

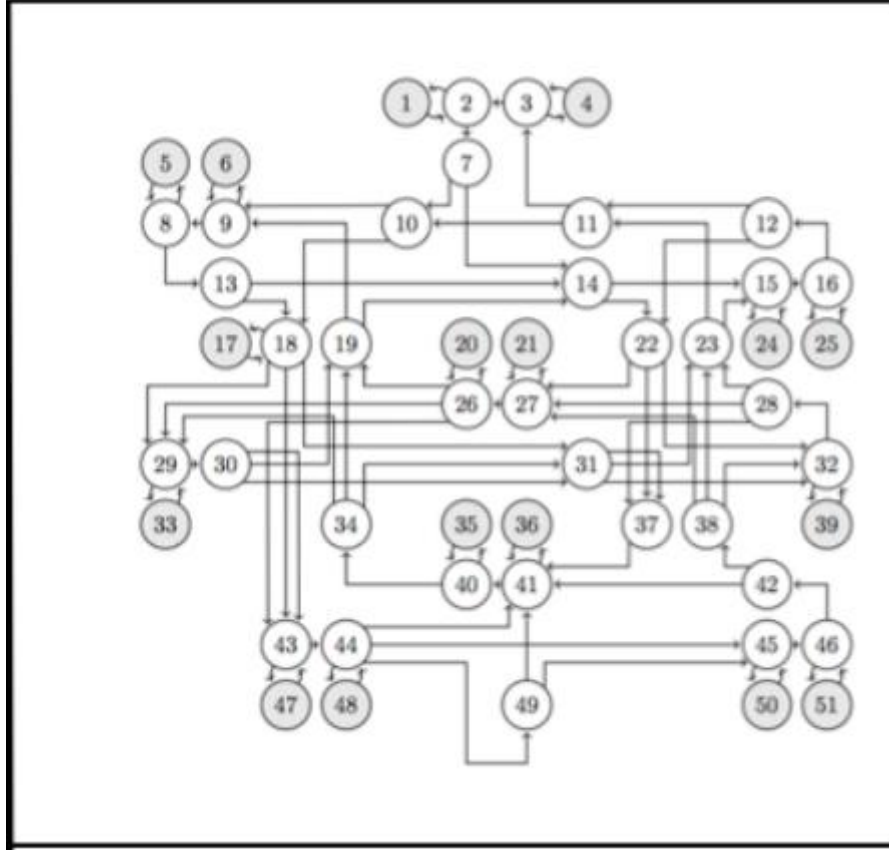


Figure 6.3 The state-space

6.2. The Controller synthesis

The controller synthesis in this test case is not the same as before because we already have a simple and discretized system. So, in our case we will not specify any sampling time and any ode_solver. We will also not have a growth bound. The dynamical computation will be in form of if statements, where each transition in the high-level model will be represented through these if statements like it is depicted in figure 6.4 where we defined that if the current state is 1 and that if we have input 0 the next state would be 2.

```

If (x[0] == 1, && u[0] == 0)
{
  xx[0] = 2 ;
}

```

Figure 6.4 Declaration of a transition from the high-level model

Concerning the state space, we declared the lower bound and the upper bound given also this high-level model with a quantization step of 1. Practically we create a one-dimensional hyper-rectangle that goes from state 1 till 69 incremented by 1 each time so that each state will be represented by a cube of dimension 1. The corresponding code is depicted in figure 6.5.

```
scots::SymbolicSet vehicleCreateStateSpace(Cudd &mgr) {

    /* setup the workspace of the synthesis problem and the uniform grid */
    /* lower bounds of the hyper rectangle */
    double lb[sDIM]={1};
    /* upper bounds of the hyper rectangle */
    double ub[sDIM]={69};
    /* grid node distance diameter */
    double eta[sDIM]={1};

    scots::SymbolicSet ss(mgr,sDIM,lb,ub,eta);

    /* add the grid points to the SymbolicSet ss */
    ss.addGridPoints();

    return ss;
}
```

Figure 6.5 State space declaration with SCOTS

For the input, the maximum transition from one state is 4, so we declared the input as going from 0 to 3 with a quantization step equal to 1 as shown in figure 6.6.

```
scots::SymbolicSet vehicleCreateInputSpace(Cudd &mgr) {

    /* lower bounds of the hyper rectangle */
    double lb[iDIM]={0};
    /* upper bounds of the hyper rectangle */
    double ub[iDIM]={3};
    /* grid node distance diameter */
    double eta[iDIM]={1};

    scots::SymbolicSet is(mgr,iDIM,lb,ub,eta);
    is.addGridPoints();

    return is;
}
```

Figure 6.6 Input space declaration with SCOTS

For the obstacles and the target declaration, and as shown in figure 6.7, we declare polytopes using the matrix

$$H=\{-1,1\},$$

and the vector h to be of the form:

$$h=\{-s, s\},$$

where s is the state that we want to omit as an obstacle.

```

/*****
/* we define the target set */
/*****
/* first make a copy of the state space so that we obtain the grid
 * information in the new symbolic set */
scots::SymbolicSet ts(ss);
/* define the target set as a symbolic set */
double H[2*sDIM]={-1,
                  1};
/* compute inner approximation of P={ x | H x<= h1 } */
double h[2] = {-12,12};
ts.addPolytope(2,H,h, scots::OUTER);
ts.writeToFile("vehicle_target.bdd");

```

Figure 6.7 Target declaration

Finally, we declare our abstraction and we compute our fixed point. The whole process leads to the synthesis of the symbolic controller. We can now process to the header file implementation using the `bdd2implement` tool.

6.3. The header file implementation

BDD2Implement is a C++ tool to generate hardware/software implementations of BDD-based symbolic controllers. Having the tools SCOTS that generate BDD-based symbolic controllers of general nonlinear dynamical systems, BDD2Implement completes missing ring in the automatic synthesis technique. BDD2Implement can generate codes in the following formats VHDL/Verilog module or in C/C++.

Practically, the tool begins by reading the `bddfile` then, given the state and its action, we generate a Boolean function representing each transition of the system to be verified. The `bdd2implement` use a template file as support to generate the code which include these Boolean functions. The resulting header file is depicted in the appendix 5.

6.4. One robot control

In this test case the main goal is to steer one robot to its target while avoiding obstacles with the help of the FMLAB interface. Each time we are going to detect the robot position with a camera then we are going to generate a controller then implement it using a C++ code. Then using the robot position or state we get the inputs from the controller then we fit this input into the dynamic to move the robot from one state to the next one considering that it has to follow a path that take it to the target.

As explained before the dynamics takes the form of IF statement then using the FMLAB function GoTo we can move the robot to the next state, as depicted in figure 6.8.

```
if (s== 1 && u[0][0] == 0)
{
    c.goTo( 2);
}
```

Figure 6.8 Declaration of a transition

With s is the current state u is the input and c the instantiation of the robot. During this test scenario, we figure out that the header file synthesized by the tool Bdd2implement has a bug in it, as it allows our robot to reach it's target but it doesn't avoid the obstacles. For that reason, we got back to the setValuedMap function as used in the raspberry PI. Whenever there is an obstacle on one path to the target, the robot changes to another path that contains no obstacles and leads to the target.

Here is an example that explains the control of one robot:

We choose as start point a state that is close to the target. From this state, the robot has two possible paths. One path is small and would take the robot toward the target after 2 steps and one path is very long. We decided to put an obstacle on the smallest path in order to steer the robot to the longest path. This explicative image (figure 6.9) shows how the controlled robot did behave on the real platform.

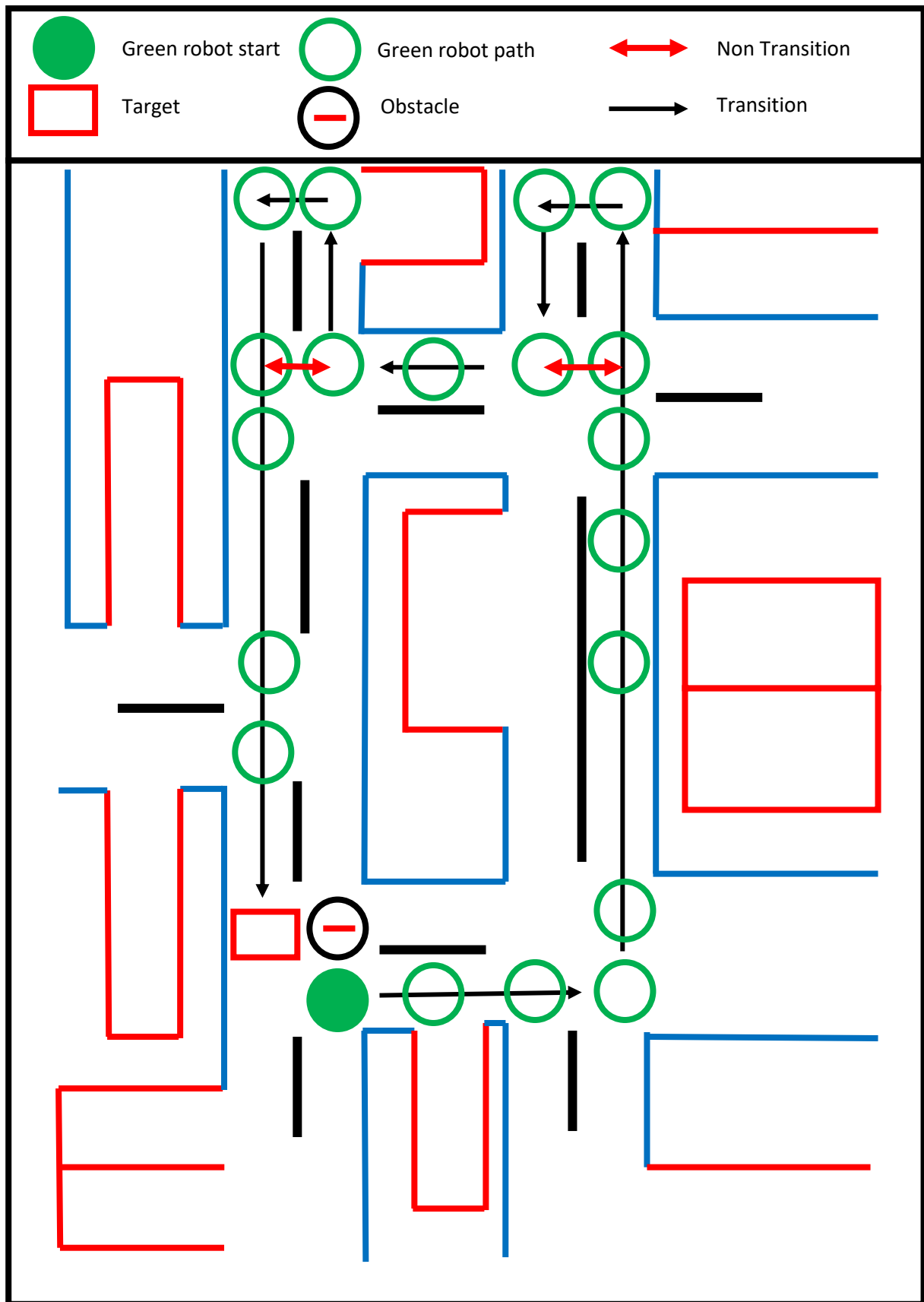
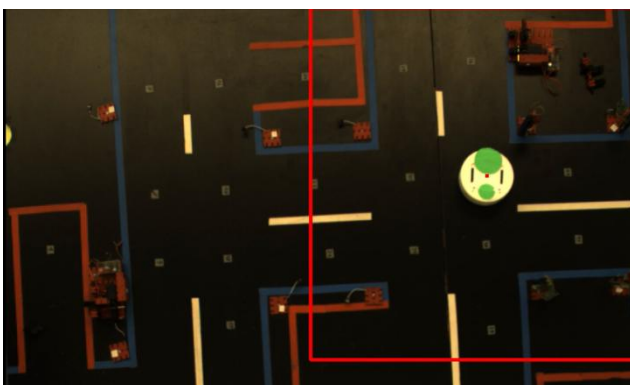
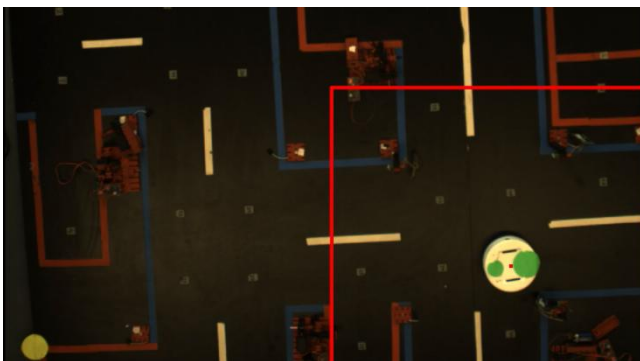
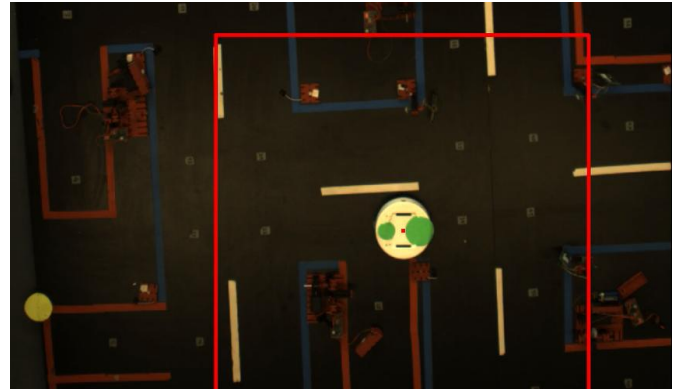
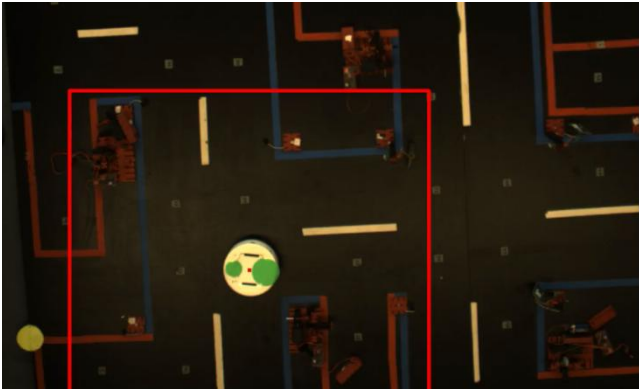


Figure 6.9 Path followed by the controlled robot to reach the target while avoiding the obstacle

As you can see the green robot could just go to the target after two steps however as the obstacle is on this path. The controller steers the robot to another path leading to the target.

In the following photos (figure 6.10), captured from the camera frame that allows us to detect the robot, represent a real overview of the followed path of the controlled robot step by step from links to right and from up to down.



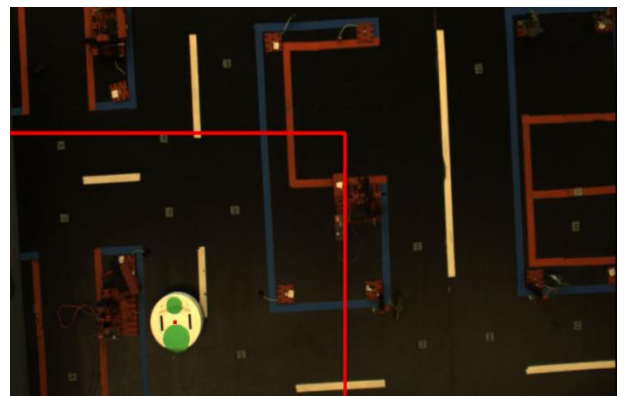
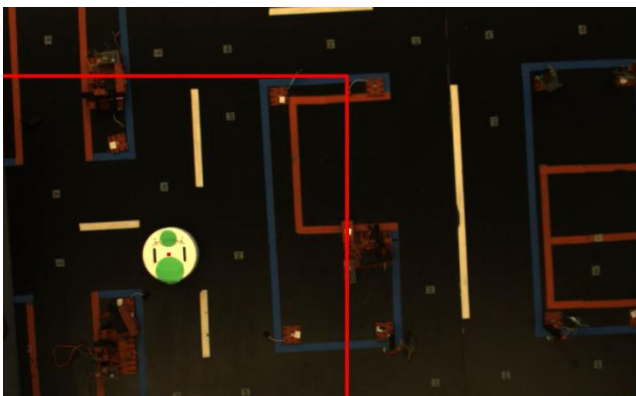
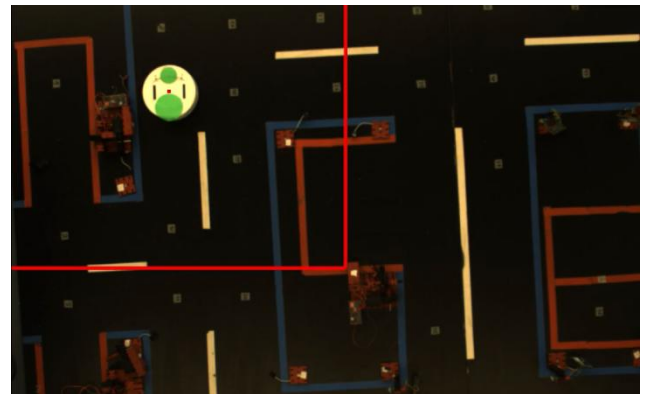
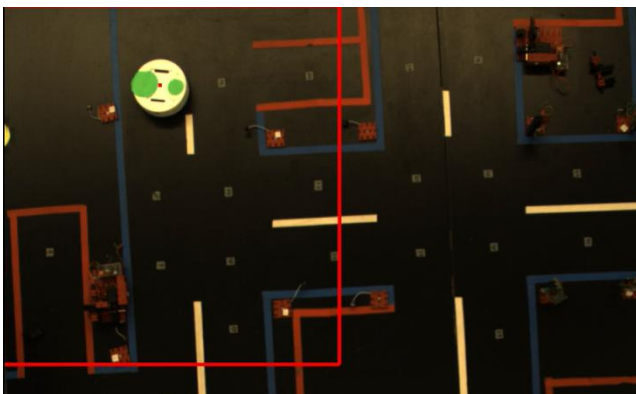
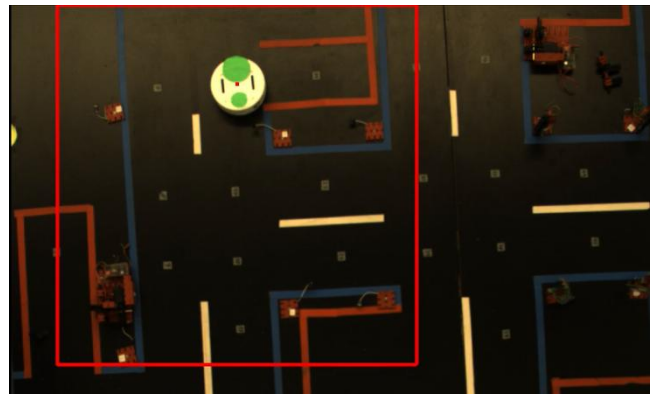
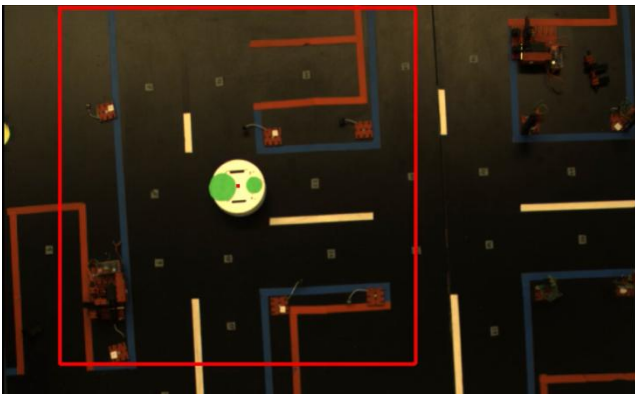
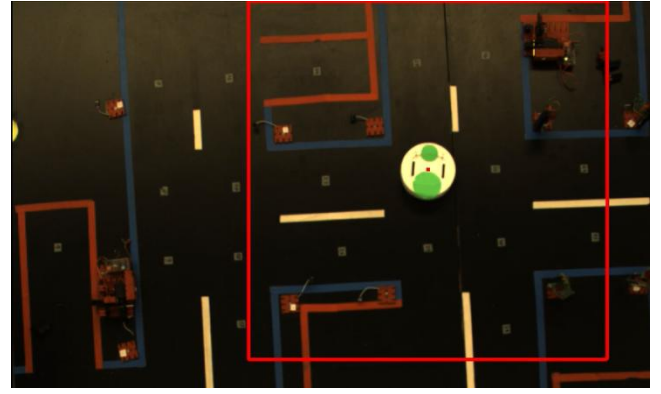
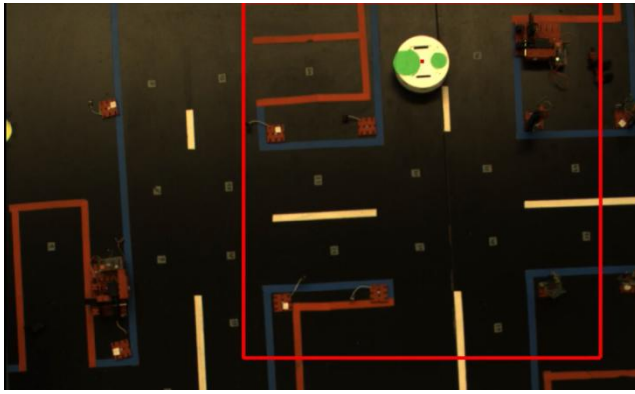


Figure 6.10. Some steps of one robot path

6.5. Control of two robots

In this test case the main goal is to control two robots simultaneously. The two robots have to reach their respective goals while avoiding each other. The whole process is online. Each time a controller is synthesized given the new state of the robot and the new obstacles.

The inputs are fed to the robots in the same way as described in the previous section and the controller synthesis is also the same as before. The only difference here is that we synthesize two different controllers given the robots positions and store them in two different bdd file.

The next step is then to load the respective controller and to read the inputs from it and write them in a text file. Finally, we read these inputs from the text file and move the robots accordingly.

A main program uses threads in order to run two sub-programs simultaneously. Each sub-program begins by detecting the respective robot position and writes it into a text file. Then this sub-program calls the controller, that given the position will synthesize a controller, generating this way a bdd file.

This bdd file is then given back to the sub-programs that reads it and then get the control input for each robot. Finally given the control input, the sub-program moves the robot to the new state. This process repeats infinitely often through a while loop inside the main program. The structure of the whole program is depicted in figure 6.11.

In this case scenario, the first robot (Green robot), start at state 2 while the second robot (blue robot), start at state 9. After two steps the blue robot is on the path of the green robot. As a result the green robot waits for the blue to get out of his path toward the target 12. After that the green robot restart moving while the blue one goes to its initial state. The green robot finally reaches its target 12 and thus has a new goal which is state 2. As the blue robot is on its path another time the controller decided to steer it to another free path. After two other steps the blue robot is on the path of the green robot, and thus the green robot stops another time to wait the path to be free as it has no other path to go in. This process is depicted in figure 6.12 and the steps were taken from the camera frame that is above the platform and is presented in figure 6.13.

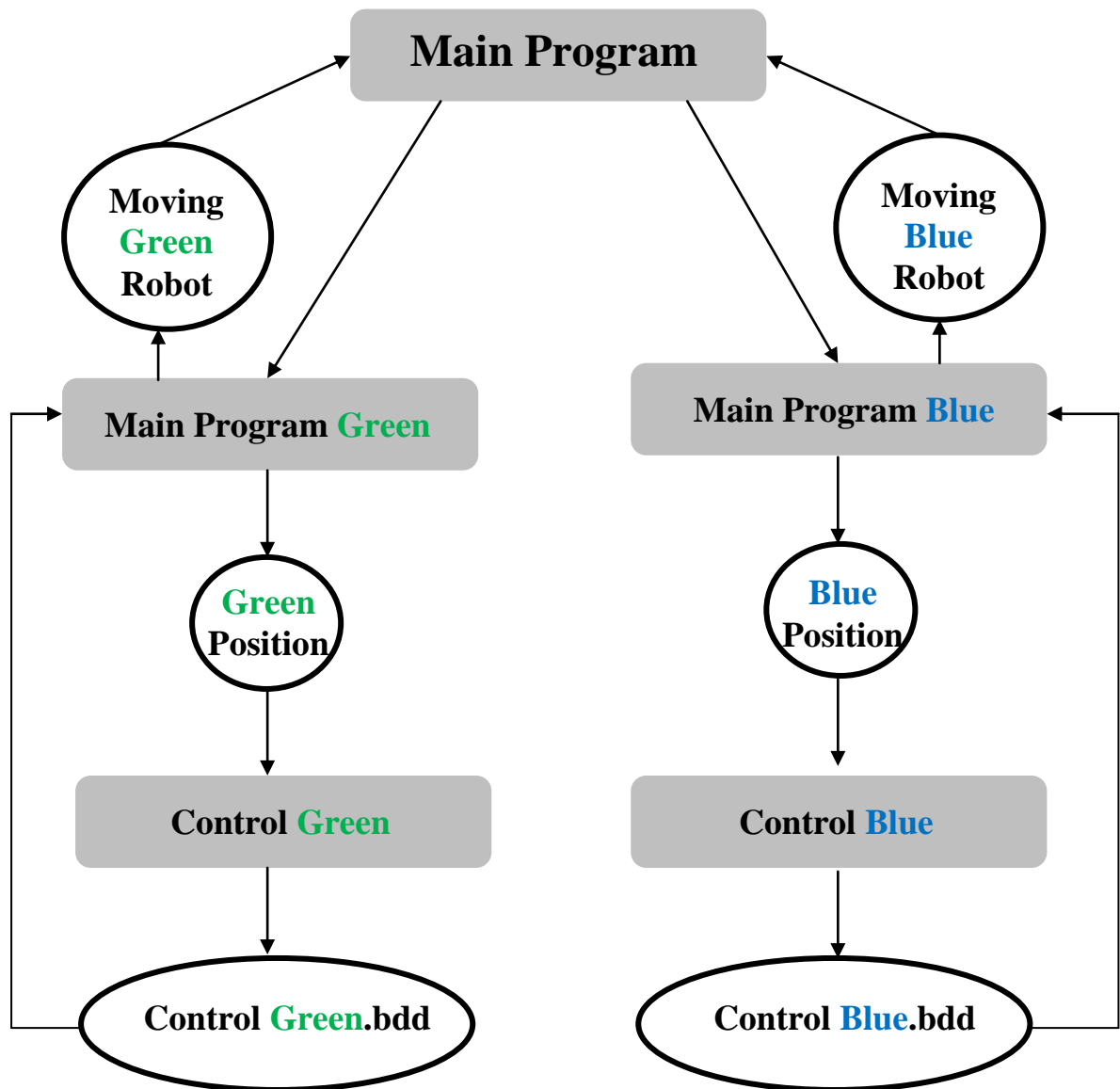


Figure 6.11. Structure of the program

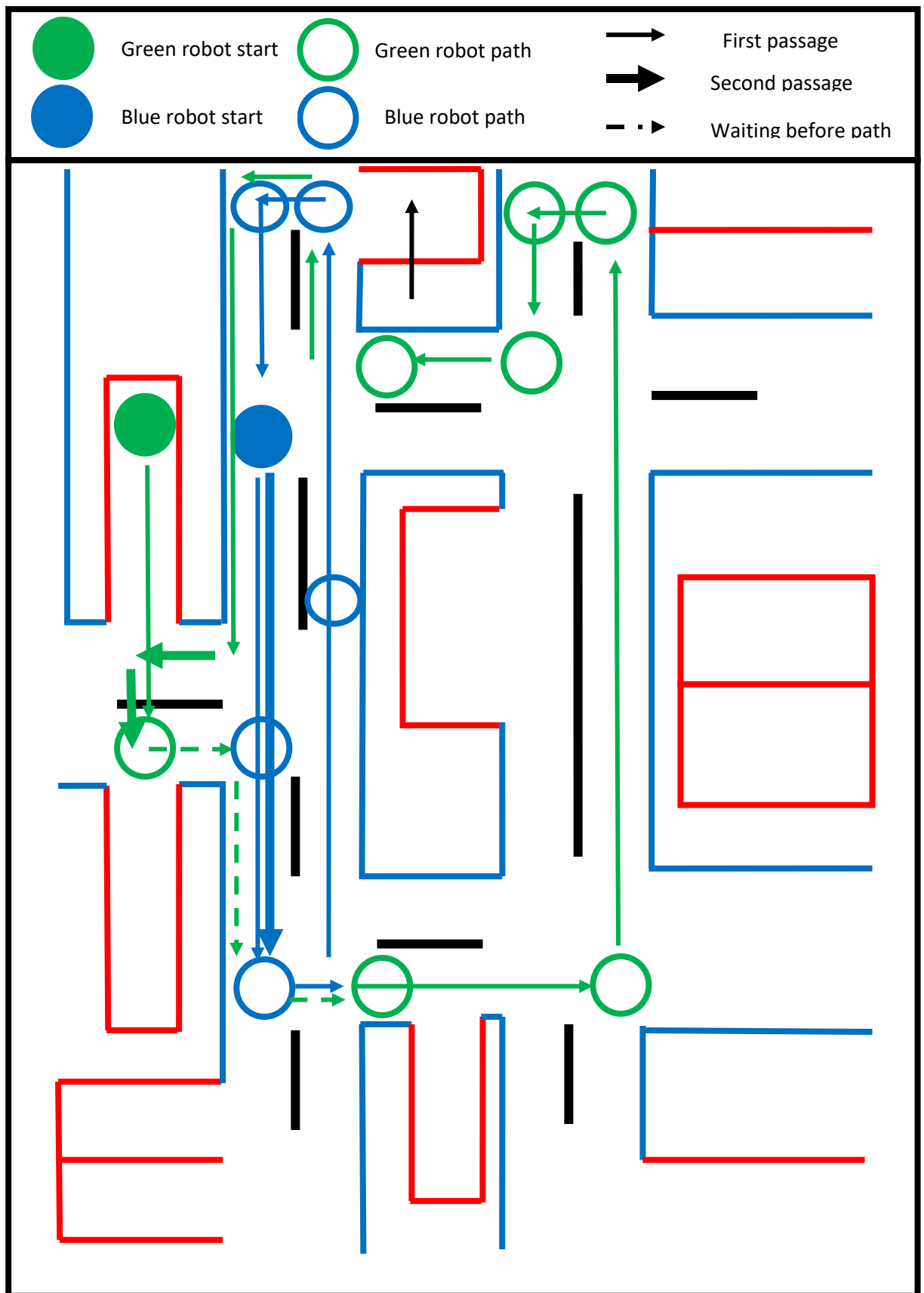
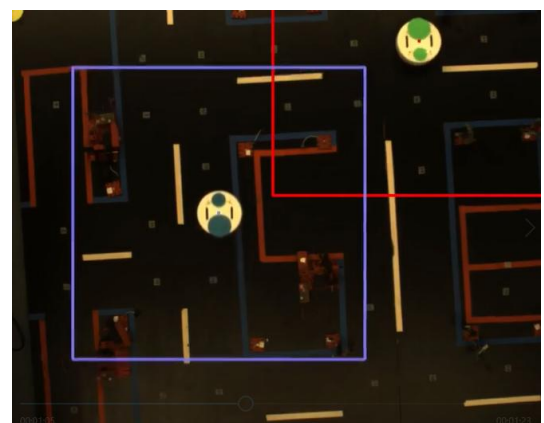
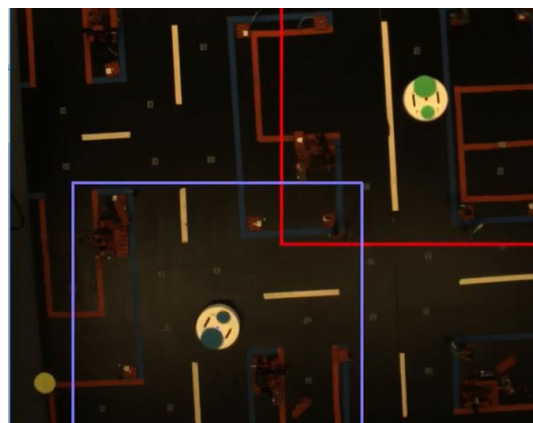
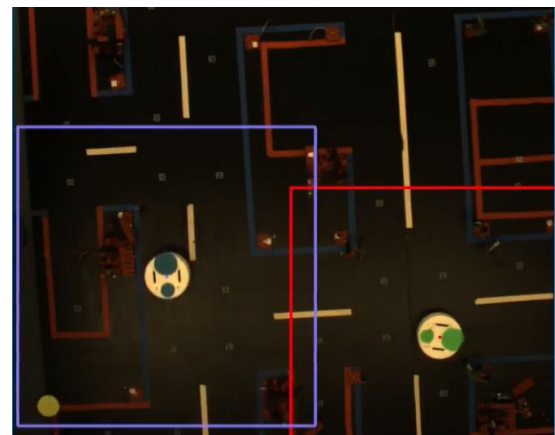
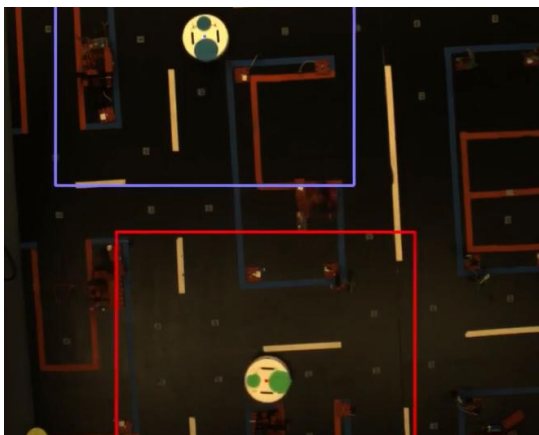
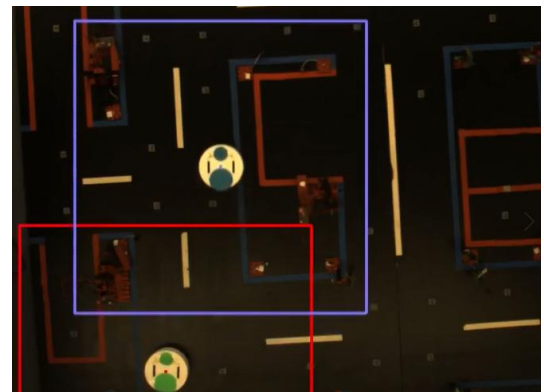
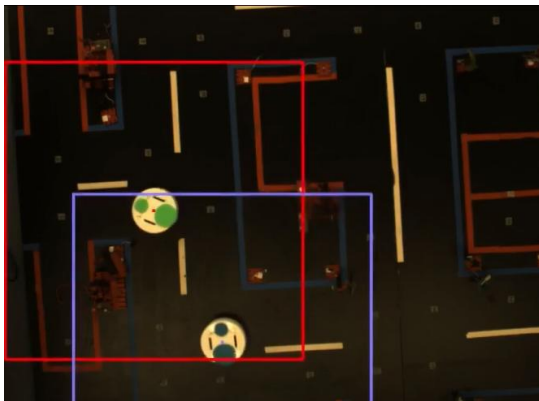
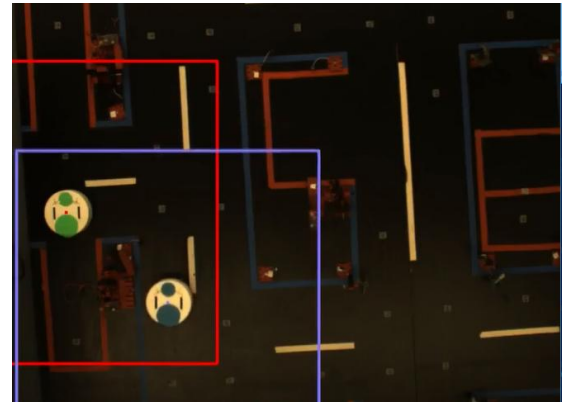
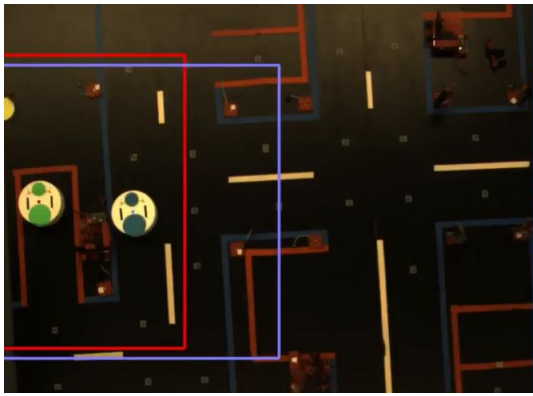


Figure 6.12. Path followed by the controlled robot to reach the target while avoiding the obstacle



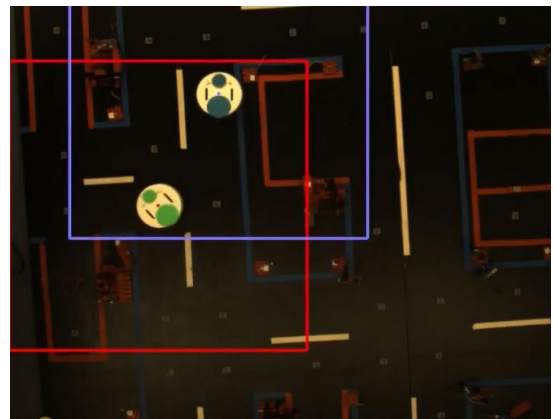
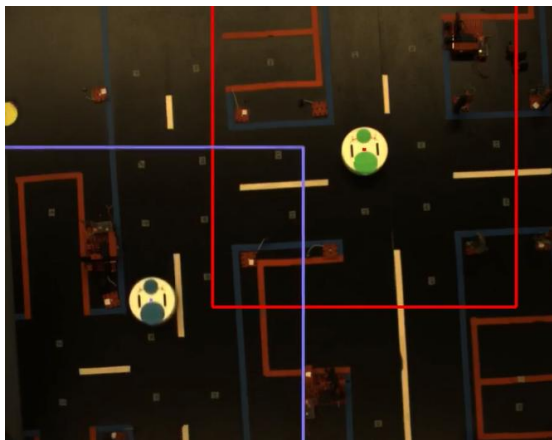
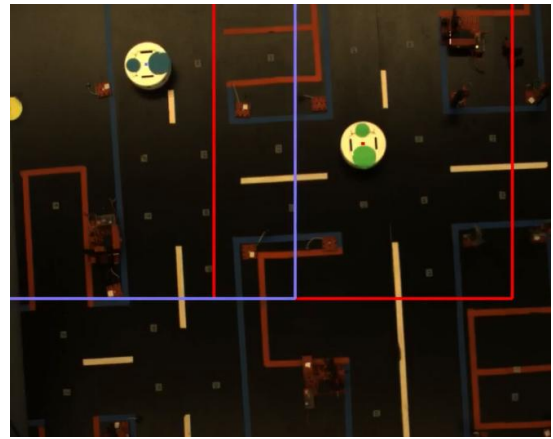


Figure 6.13. Some steps of two robots path

Chapter 7.

Conclusion and Future Work

7.1. Conclusion

The purpose of this thesis is to present implementations of a symbolic controller, for unicycle dynamics on FPGAs and a Raspberry PI and its test on robots.

Scots is here used to solve the abstract control problem using fixed point computation and finally the synthesized controller is refined to match the concrete system.

About the implementation on FPGA:

- First and using mainly software in the loop technique, the controller was tested.
- Second the controller inputs were obtained from the function `getInputs`, and then were fed to the dynamics that were simulated using the hardware simulator.
- Finally, the controller was implemented on the FPGA, where we substituted the controller into a VHDL code and then integrated it in an AXI peripheral. This AXI serves as interface between the computer and the FPGA to get the control input given the state.

About the implementation over the Raspberry Pi 3, the interaction with the synthesized controller is done through the C code functions using the SymbolicSet library.

Finally, our controller is tested on a real robot given an urban like environment. As a first part, the controller is synthesized and then implemented in the form of a C code. This C code gave the specification and the state space generate inputs to the robot in order to steer it toward the goal while avoiding obstacles. As a second part, the controller is synthesized online while moving the robots giving a dynamically changing obstacle. The two robots reach their respective target while avoiding each other.

7.2. Future Work

For the vehicle example, a multi layered abstraction could be done in order to solve the control problem, where for example three layer of abstraction with different quantization step and sampling time could be realized, and each time it would solve the problem using the layer with the biggest quantization step whenever this is feasible, and change to a finer one whenever it is needed.

List of Figures

Figure 3.1. Representation of a polytope on the left of the figure and an ellipsoid on the right

Figure 3.2. Sample-and-hold implementation of a controller synthesized with SCOTS

Figure 4.1. OP 4200

Figure 4.2. Top level representation of the RT-Lab simulink model

Figure 4.3. Comsol Subsystem

Figure 4.4. Schematic of the BDD implementation tool

Figure 4.5. Path of the vehicle, modeled by a unicycle model, inside the state space

Figure 5.1. Connexion and expansion

Figure 6.1 The urban like Environment

Figure 6.2 Robot Khepera IV

Figure 6.3 The state-space

Figure 6.4 Declaration of a transition from the high level model

Figure 6.5 State space declaration with SCOTS

Figure 6.6 Input space declaration with SCOTS

Figure 6.7 Target declaration

Figure 6.8 Declaration of a transition

Figure 6.9 Path followed by the controlled robot to reach the target while avoiding the obstacle

Figure 6.10. Some steps of one robot path

Figure 6.11. Structure of the program

Figure 6.12. Path followed by the controlled robot to reach the target while avoiding the obstacle

Figure 6.13. Some steps of two robots path

Bibliography

- [1]
<https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-design-tools/archive.html>
- [2] F. Somenzi. “CUDD: CU decision diagram package release”. In: University of Colorado at Boulder (1998).
- [3] 2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC), Cyber Physical Systems: Design Challenges
Year: 2008, Volume: 1, Pages: 363-369. DOI Bookmark:10.1109/ISORC.2008.25
Authors: Edward A. Lee
- [4] https://en.wikipedia.org/wiki/Hardware-in-the-loop_simulation
- [5] <https://www.opal-rt.com/op4200/>
- [6] <https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html>
- [7] <https://alchitry.com/blogs/tutorials/what-is-an-fpga>
- [8] <http://zedboard.org/product/zedboard>
- [9] <http://zedboard.org/sites/default/files/documentations/GS-AES-Z7EV-7Z020-G-V7-1.pdf>
- [10] https://en.wikipedia.org/wiki/Formal_verification
- [11]
https://www.tutorialspoint.com/modelling_and_simulation/modelling_and_simulation_verification_validation.htm
- [12] <https://gitlab.lrz.de/hcs/sense/tree/master/tools/bdd2implement>

Appendix 1 vehicle_simul.m

```
function vehicle_simul
clear set
close all

%% simulation

% target set
lb=[9 0];
ub=lb+0.5;
v=[9 0; 9.5 0; 9 0.5; 9.5 .5];
% initial state
x0=[0.4 0.4 0.0];
t=[];

controller=SymbolicSet('vehicle_controller.bdd','projection',[1 2 3]); %read th controller bdd-
file
target=SymbolicSet('vehicle_target.bdd');%read th target bdd-file

y=x0;
v=[];
while(1)

    u=controller.getInputs(y(end,:));%get controller output
    v=[v; u(1,:)];
    if (target.isElement(y(end,:)))%check with target
        break;
    end
    [t x]=ode(1,y(end,:),u(1,:));%ode solver with simulink

    y=[y;x(end,:)];
end

%% plot the vehicle domain
% colors
colors=get(groot,'DefaultAxesColorOrder');

% load the symbolic set containig the abstract state space
set=SymbolicSet('vehicle_ss.bdd','projection',[1 2]);
plotCells(set,'facecolor','none','edgec',[0.8 0.8 0.8],'linew',.1)
hold on

% load the symbolic set containig obstacles
set=SymbolicSet('vehicle_obst.bdd','projection',[1 2]);
plotCells(set,'facecolor',colors(1,:)*0.5+0.5,'edgec',colors(1,:), 'linew',.1)

% plot the real obstacles and target set
plot_domain

% load the symbolic set containig target set
set=SymbolicSet('vehicle_target.bdd','projection',[1 2]);
plotCells(set,'facecolor',colors(2,:)*0.5+0.5,'edgec',colors(2,:), 'linew',.1)

% plot initial state and trajectory
```

```

plot(y(:,1),y(:,2),'k.-')
plot(y(1,1),y(1,1),'.','color',colors(5,:),'markersize',20)

box on
axis([-0.5 10.5 -0.5 10.5])

end
%function dxdt = unicycle_ode(t,x,u)
function [t x]= ode(t,Y,U)
X=[];
t= [];
U=U;
Y=Y;
model='test_vehicle_dynamics';
open_system(model);
%Apply changes to the model specified through a SimulationInput object, in
in = Simulink.SimulationInput(model);

t = (0:0.01:10)';%time
%define the model paramters
U1 =U(1,1);
U2 = U(1,2);
y1 = Y(1,1);
y2 = Y(1,2);
y3 = Y(1,3);

%set model parameters
in = in.setVariable('U1',U1);
in = in.setVariable('U2',U2);
in = in.setVariable('y1',y1);
in = in.setVariable('y2',y2);
in = in.setVariable('y3',y3);

% configure the simulation
Set_param(model, 'Abstol', '1e-6', 'StopTime','0.3')
outputs=simOut.get('yout');%get simulation output
x1=(outputs.get('X1').Values);
x2=(outputs.get('X2').Values);
x3=(outputs.get('X3').Values);
t=x1.Time;
x1=x1.Data;
x2=x2.Data;
x3=x3.Data;
x=[x1';x2';x3'];
%
% dxdt = zeros(3,1);
% c=atan(tan(u(2))/2);
%
% dxdt(1)=u(1)*cos(c+x(3))/cos(c);
% dxdt(2)=u(1)*sin(c+x(3))/cos(c);
% dxdt(3)=u(1)*tan(u(2));
%
%
end

function plot_domain

```

```

colors=get(groot,'DefaultAxesColorOrder');

v=[9 0; 9.5 0; 9 0.5; 9.5 .5];
patch('vertices',v,'faces',[1 2 4 3],'facec',colors(2,:), 'edgec',colors(2,:));

v=[1 0 ;1.2 0 ; 1 9 ; 1.2 9 ];
patch('vertices',v,'faces',[1 2 4 3],'facec',colors(1,:), 'edgec',colors(1,:));
v=[2.2 0 ;2.4 0 ; 2.2 5 ; 2.4 5 ];
patch('vertices',v,'faces',[1 2 4 3],'facec',colors(1,:), 'edgec',colors(1,:));
v=[2.2 6 ;2.4 6 ; 2.2 10 ; 2.4 10 ];
patch('vertices',v,'faces',[1 2 4 3],'facec',colors(1,:), 'edgec',colors(1,:));
v=[3.4 0 ;3.6 0 ; 3.4 9 ; 3.6 9 ];
patch('vertices',v,'faces',[1 2 4 3],'facec',colors(1,:), 'edgec',colors(1,:));
v=[4.6 1 ;4.8 1 ; 4.6 10 ; 4.8 10 ];
patch('vertices',v,'faces',[1 2 4 3],'facec',colors(1,:), 'edgec',colors(1,:));
v=[5.8 0 ;6 0 ; 5.8 6 ; 6 6 ];
patch('vertices',v,'faces',[1 2 4 3],'facec',colors(1,:), 'edgec',colors(1,:));
v=[5.8 7 ;6 7 ; 5.8 10 ; 6 10 ];
patch('vertices',v,'faces',[1 2 4 3],'facec',colors(1,:), 'edgec',colors(1,:));
v=[7 1 ;7.2 1 ; 7 10 ; 7.2 10 ];
patch('vertices',v,'faces',[1 2 4 3],'facec',colors(1,:), 'edgec',colors(1,:));
v=[8.2 0 ;8.4 0 ; 8.2 8.5 ; 8.4 8.5 ];
patch('vertices',v,'faces',[1 2 4 3],'facec',colors(1,:), 'edgec',colors(1,:));
v=[8.4 8.3;9.3 8.3 ; 8.4 8.5 ; 9.3 8.5 ];
patch('vertices',v,'faces',[1 2 4 3],'facec',colors(1,:), 'edgec',colors(1,:));
v=[9.3 7.1;10 7.1 ; 9.3 7.3 ; 10 7.3 ];
patch('vertices',v,'faces',[1 2 4 3],'facec',colors(1,:), 'edgec',colors(1,:));
v=[8.4 5.9;9.3 5.9 ; 8.4 6.1 ; 9.3 6.1 ];
patch('vertices',v,'faces',[1 2 4 3],'facec',colors(1,:), 'edgec',colors(1,:));
v=[9.3 4.7;10 4.7 ; 9.3 4.9 ; 10 4.9 ];
patch('vertices',v,'faces',[1 2 4 3],'facec',colors(1,:), 'edgec',colors(1,:));
v=[8.4 3.5;9.3 3.5 ; 8.4 3.7 ; 9.3 3.7 ];
patch('vertices',v,'faces',[1 2 4 3],'facec',colors(1,:), 'edgec',colors(1,:));
v=[9.3 2.3;10 2.3 ; 9.3 2.5 ; 10 2.5 ];
patch('vertices',v,'faces',[1 2 4 3],'facec',colors(1,:), 'edgec',colors(1,:));

end

```


Appendix 2 BDD.m

The different codes will be here presented

```
function bddtomatfile()
    lw0=0;%min x
    lw1=0;
    lw2=-3.4;
    eta0=0.2;%quantization x
    eta1=0.2;
    eta2=0.2;
    controller=SymbolicSet('vehicle_controller.bdd');%reading controller
    conpoint=controller.points;% controller matrice
    target=SymbolicSet('vehicle_target.bdd'); %reading Target
    targetpoints=target.points;% Target matrice
    A=ones(51^3,2)*100;% ones matrice
    c=conpoint(:,1);
    c=(c-lw0)/eta0;% Abstraction Controller
    c1=conpoint(:,2);
    c1=(c1-lw1)/eta1;
    c2=conpoint(:,3);
    c2=(c2-lw2)/eta2;
    c3=conpoint(:,4);
    c4=conpoint(:,5);
    constate=[c c1 c2 c3 c4];
    for m=1:size(constate)% indexing Controller
        index=constate(m,1)*50^2+constate(m,2)*50^1+constate(m,3)*50^0;
        index=int64(index);
        A(index,1)=constate(m,4);
        A(index,2)=constate(m,5);
    end
    c=targetpoints(:,1);% Abstraction Target
    c=(c-lw0)/eta0;
    c1=targetpoints(:,2);
    c1=(c1-lw1)/eta1;
    c2=targetpoints(:,3);
    c2=(c2-lw2)/eta2;
    targetstate=[c c1 c2];

    for m=1:size(targetstate)% Indexing Controller
        index=targetstate(m,1)*50^2+targetstate(m,2)*50^1+targetstate(m,3)*50^0;
        index=int64(index);
        targetindex(m)=index;
    end
    save('controllernew.mat','conpoint')% Save Controller and target
    save('target.mat','targetpoints')
    save('controlsymbloic.mat','A')
    save('targetindex.mat','targetindex')
    A1=A(:,1)';
    A2=round(A(:,2),3);
    csvwrite('controller1.dat',A1)
    csvwrite('controller2.dat',A2)
end
function combs = nmultichoosek(values, k)
%// Return number of multisubsets or actual multisubsets.
if numel(values)==1
```

```
    n = values;  
    combs = nchoosek(n+k-1,k);  
else  
    n = numel(values);  
    combs = bsxfun(@minus, nchoosek(1:n+k-1,k), 0:k-1);  
    combs = reshape(values(combs),[],k);  
end
```

Appendix 3 bddhil.c

```
/*
 * Placeholder PetaLinux user application.
 *
 * Replace this with your application code
 */
#include <stdio.h>
#include <stdlib.h>
#include "bddReader.h"

int
set_interface_attribs (int fd, int speed, int parity)
{
    struct termios tty;
    memset (&tty, 0, sizeof tty);
    if (tcgetattr (fd, &tty) != 0)
    {
        printf ("Error %d from tcgetattr", errno);
        return -1;
    }

    cfsetospeed (&tty, speed);
    cfsetispeed (&tty, speed);

    tty.c_cflag = (tty.c_cflag & ~CSIZE) | CS8;    // 8-bit chars
    // disable IGNBRK for mismatched speed tests; otherwise receive
break
    // as \000 chars
    tty.c_iflag &= ~IGNBRK;                        // disable break processing
    tty.c_lflag = 0;                                // no signaling chars, no echo,
                                                    // no canonical processing
    tty.c_oflag = 0;                                // no remapping, no delays
    tty.c_cc[VMIN] = 0;                             // read doesn't block
    tty.c_cc[VTIME] = 5;                            // 0.5 seconds read timeout

    tty.c_iflag &= ~(IXON | IXOFF | IXANY); // shut off xon/xoff

ctrl

    tty.c_cflag |= (CLOCAL | CREAD); // ignore modem controls,
                                    // enable reading
    tty.c_cflag &= ~(PARENB | PARODD); // shut off parity
    tty.c_cflag |= parity;
    tty.c_cflag &= ~CSTOPB;
    tty.c_cflag &= ~CRTSCTS;

    if (tcsetattr (fd, TCSANOW, &tty) != 0)
    {
        printf ("Error %d from tcsetattr", errno);
        return -1;
    }
    return 0;
}

void
set_blocking (int fd, int should_block)
{
    struct termios tty;
    memset (&tty, 0, sizeof tty);
```

```

    if (tcgetattr (fd, &tty) != 0)
    {
        printf("Error %d from tcgetattr", errno);
        return;
    }

    tty.c_cc[VMIN]  = should_block ? 1 : 0;
    tty.c_cc[VTIME] = 5;           // 0.5 seconds read timeout

    if (tcsetattr (fd, TCSANOW, &tty) != 0)
        printf ("Error %d setting term attributes", errno);
}

int readLine(int fd, char* lineBuffer, int max){

    char ch;
    int count = 0;

    read(fd, &ch, 1);
    do
    {
        while(read(fd, &ch, 1) < 1);

        if(ch != '\n' && ch != '\r'){
            lineBuffer[count] = ch;
            count++;
        }
        if(ch == '\n'){
            write (fd, "\r\n", 2);
            usleep (200);
        }
        else{
            write (fd, &ch, 1);
            usleep (100);
        }

    }while ((ch != '\n') && (count != max-1));

    lineBuffer[count] = '\0';
    return count++;
}

char *portname = "/dev/ttyPS0";
int main(int argc, char *argv[])
{
    unsigned long exit=0,fd,state;
    ssize_t n;
    char buf[100], action[50];

    fd = open (portname, O_RDWR | O_NOCTTY | O_SYNC);
    if (fd < 0){
        printf("Error: %d opening %s: %s", errno, portname,
strerror (errno));
        return 0;
    }

    set_interface_attribs (fd, B115200, 0);    // set speed to
115,200 bps, 8n1 (no parity)
    set_blocking (fd, 0);                      // set no
blocking

```

```

        write (fd, "HIL started v1.1 !\n", 12);           // send 7
character greeting

do{
    n = readLine(fd, buf, sizeof buf);
    if(n > 0){
        if(strcmp(buf,"exit")==0){
            exit = 1;
        }
        else if(strcmp(buf,"ping")==0){
            write (fd, "pong\r\n", 6);
            usleep (6 * 100);
        }
        else{
            state = atol(buf);
            n =
sprintf(action,"%d",bddGetAction(state));
            write (fd, action, n);
            write (fd, "\r\n", 2);
            usleep ((n+2) * 100);
        }
    }
}while(!exit);
return 0;
}

```


Appendix 4 bddReader.h

```
#include <stdint.h>
#include <assert.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <unistd.h>
#include <stddef.h>
#include <errno.h>
#include <termios.h>
#include <unistd.h>

#define BDD_PHY_ADDRESS 0x43C00000
#define BDD_INP_OFFSET 0
#define BDD_OUT_OFFSET 1

// You need to be in root to run this!!
#define MAP_SIZE 4096UL // our 4K range of the AXI bdd
#define MAP_MASK (MAP_SIZE-1)
void* get_v_addr(int phys_addr){
    void* mapped_base;
    int memfd;
    void* mapped_dev_base;
    off_t dev_base = phys_addr;
    memfd = open("/dev/mem", O_RDWR | O_SYNC);
    if(memfd == -1){
        printf("Cant open /dev/mem.\n");
        exit(0);
    }

    mapped_base = mmap(0, MAP_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, memfd, dev_base & ~MAP_MASK);
    if(mapped_base == (void*)-1){
        printf("Cant map the physial address to user-space
!.\n");
        exit(0);
    }

    mapped_dev_base = mapped_base + (dev_base & MAP_MASK);
    return mapped_dev_base;
}

unsigned long bddGetAction(unsigned long state)
{
    unsigned long* bdd_addr = get_v_addr(BDD_PHY_ADDRESS);

    unsigned long* bdd_inp = bdd_addr + BDD_INP_OFFSET;
    unsigned long* bdd_out = bdd_addr + BDD_OUT_OFFSET;
    *bdd_inp = state;
    usleep (100);
    unsigned long out = *bdd_out;
    return out;
}
```


Appendix 5 bdd_vehicle.h

```
#define BDD_INPUT_BITS 7
#define BDD_OUTPUT_BITS 2

bool f0(bool x0, bool x1, bool x2, bool x3, bool x4, bool x5, bool x6){
    return (!((x0 && ((x1 && ((x2 && ((x3 && ((x4 &&
((x5) || ((x6)))))) || (! (x4) && ((x6)))))) || (! (x3) &&
((x6)) || (! (x5)))))) || (! (x2) && ((x3 && (((x6)) ||
(! (x5))) || (! (x4)))) || (! (x3) && ((x4) || (((x5) ||
(! ((x6)))))))))) || (! (x1) && ((x2 && ((x3 && ((x4 &&
((x6)) || (! (x4) && ((x5) || (((x6)))))) || (! (x3) && ((x4
&& (((x6)) || (! (x5)))) || (! (x4) && ((x5) || (((x6))))))))))
|| (! (x2) && ((x3 && (((x6)) || (! (x4)))) || (! (x3) && ((x4)
|| (((x5) || (((x6)))))))))) || (! (x0) && ((x1 && ((x2 &&
((x3 && ((x4 && (((x6)) || (! (x5)))) || (! (x4) && ((x5) ||
(((x6)))))) || (! (x3) && (((x6)) || (! (x5)))) || (! (x4))))))
|| (! (x2) && ((x3 && ((x4) || (((x5) || (((x6)))))))) ||
(! (x3) && ((x4 && ((x6)) || (! (x4) && ((x5) ||
(! ((x6)))))))))) || (! (x1) && ((x2 && ((x3 && ((x4 &&
((x6)) || (! (x5)))) || (! (x4) && ((x6)))) || (! (x3) && ((x4
&& ((x5) || (((x6)))) || (! (x4) && ((x6)))))) || (! (x2) &&
((x3 && (((x6)) || (! (x5)))) || (! (x4)))) || (! (x3) && ((x4
&& ((x6)) || (! (x4) && ((x6)) || (! (x5))))))))))))));
}

bool f1(bool x0, bool x1, bool x2, bool x3, bool x4, bool x5, bool x6){
    return (!((x0 && ((x1 && ((x2 && ((x3 && ((x4 &&
((x6)) || (! (x5)))) || (! (x4) && ((x5) || (((x6)))))) ||
(! (x3) && (((x6)) || (! (x4)))))) || (! (x2) && ((x4 && ((x6))
|| (! (x4) && ((x6)) || (! (x5)))))) || (! (x1) && ((x2 &&
((x3 && ((x6)) || (! (x3) && ((x4) || (((x6)))))) || (! (x2)
&& ((x3 && ((x4 && ((x6)) || (! (x4) && ((x6)) ||
(! (x5)))))) || (! (x3) && ((x4 && ((x5) || (((x6)))) ||
(! (x4) && ((x5) || (! ((x6)))))))))) || (! (x0) && ((x1 &&
((x2 && ((x3 && (((x6)) || (! (x4)))) || (! (x3) && ((x4 &&
((x6)) || (! (x5)))) || (! (x4) && ((x6)))))) || (! (x2) &&
((x3 && ((x4) || (((x6)) || (! (x5)))))) || (! (x3) && ((x4)
|| (((x5 && ((x6)))))))))) || (! (x1) && ((x2 && ((x3 &&
((x4 && ((x6)) || (! (x4) && ((x6)) || (! (x5)))))) || (! (x3)
&& (((x6)) || (! (x5)))) || (! (x4)))))) || (! (x2) && ((x3 &&
((x6)) || (! (x4)))) || (! (x3) && ((x4 && ((x6)) || (! (x4)
&& ((x5))))))))))))));
}

typedef bool (*bool_func_ptr)(bool x0, bool x1, bool x2, bool x3, bool
x4, bool x5, bool x6);
bool_func_ptr bddBoolFunctions[BDD_OUTPUT_BITS];

void interfaceBooleanFunctions(bool* inputValues, bool* outValues){
    bddBoolFunctions[0] = f0;
    bddBoolFunctions[1] = f1;

    unsigned long int i;
    for(i=0; i<BDD_OUTPUT_BITS; i++){
```

```

        outValues[i]=bddBoolFunctions[i]( inputValues[0],
inputValues[1], inputValues[2], inputValues[3], inputValues[4],
inputValues[5], inputValues[6] );
    }
}

void int_to_bool(unsigned long int in, int count, bool* out){
    unsigned long int mask = 1L;
    int i;
    for (i = 0; i < count; i++) {
        out[i] = (in & mask) ? true : false;
        in >>= 1;
    }
}

unsigned long int bool_to_int(int count, bool* in){
    int i;
    unsigned long int val=0;
    unsigned long int weight=1;
    for (i = 0; i < count; i++) {
        val += weight*in[i];
        weight <<= 1;
    }
    return val;
}

unsigned long int getControlAction(unsigned long int state){
    bool stateBits[BDD_INPUT_BITS];
    bool actionBits[BDD_OUTPUT_BITS];
    int_to_bool(state, BDD_INPUT_BITS, stateBits);
    interfaceBooleanFunctions(stateBits, actionBits);
    return bool_to_int(BDD_OUTPUT_BITS, actionBits);
}

```

