Parametric Reachability Synthesis using the tool SCOTS

Bachelor Thesis

Scientific work to obtain the degree
B.Sc. Electrical Engineering and Information Technology

Department of Electrical and Computer Engineering
Technische Universität München

Supervised by Prof. Dr. Majid Zamani & Mr Mahmoud Khaled
Assistant Professorship of Hybrid Control Systems
TUM Department of Electrical and Computer Engineering

Submitted by Nur Hanis Bin Samad
213 Pasir Ris Street 21
#02-208 Singapore 510213
+65 97718770

Filed München, on 18 June 2018

Singapore Institute of Technology - Technische Universität München

Parametric Reachability Synthesis using the tool SCOTS

Bachelor Thesis

scientific work for the degree

Bachelor of Science

in

Electrical Engineering and Information Technology

Nur Hanis Bin Samad

April 2nd 2018 to June 22nd 2018

Abstract

SCOTS is C++ tool developed by Dr. Matthias Rungger and Dr. Majid Zamani to perform the synthesis of symbolic controllers [1]. A control or mechanical system can be defined via differential equations, this then becomes the input for computation of a symbolic model. The software can be accessed via this site `https://www.hcs.ei.tum.de/en/software/scots/`

SCOTS is built upon the concept of Binary Decision Diagrams, where transitional states are created leading to atomic propositions [1]. The target controller in this case computes its path and trajectory based on a set of obstacles drafted within the state space via polytopes and ellipsoids. Similar to binary search trees, the target would navigate accordingly based on the objects placement. With parametric reachability in mind, we are tasked to encode multiple obstacle configurations as different states/possibilities within the model.

We would now access the capabilities of SCOTS to perform parametric reachability synthesis, which refers to the formal verification of 2 things in particular namely the reachability computation and parameter synthesis.[2] The objective is to store multiple configurations of obstacles into one Binary Decision Diagram where we would then synthesise a controller whose overall capability is to navigate to specified target within the state space regardless of the obstacle configuration.

The implementation of this task is via C++ programming of which SCOTS was created and we simulate various cases via MATLAB [1].

Acknowledgement

I would like to extend my love and gratitude to my **Family** back in Singapore. Attempting a Bachelor Thesis in Germany for a period of 3 months, including performing a month fasting during Ramadhan has been a trying and challenging experience, where it is also a period which led to some self-discovery and independence. Thank you all for your love, support and prayers whilst I'm away. I promise to return home safe.

My girlfriend, **Hazelinda**. Thank you for believing in me and being a constant motivation for me to complete my studies. I am eternally grateful for all the support you have given me be it small or big, I'm sure you know what they are. Life in Germany was certainly made a lot easier with all the provisions you and your family prepared for me before the trip. May you and your family always be in good health.

Mr Mahmoud Khaled, thank you for your counsel and supervision in explaining and aiding my comprehension on the aforementioned topic. The task at hand has certainly challenged me technically and would like to seek your apology for any slowness in comprehension and execution of tasks. Thank you for your patience in guiding me through the entirety of the project. Eid Mubarak to you and your family back in Egypt and that you are rewarded with the PhD you thoroughly deserve.

List of Figures

List of Equations

List of Tables

Contents

Listings

Chapter 1

Introduction

1.1 Background

The synthesis of controllers for any system has become a necessary undertaking for anyone in the control field [8]. Its importance allows an engineer to comprehend the dynamism of a systems response to changing inputs or other random interference both internally and externally. Feedback control systems theory allows the engineer to develop solutions to rectify any design deficiencies.

Control synthesis has also extended towards symbolic model theory [1]. The need to assess reactive systems is crucial to the design of many modern day mechanical and electronic systems. With the concept of Binary Decision Diagrams [3], SCOTS was created to alleviate the difficulties of performing symbolic controller synthesis by instrinscally providing the required algorithms to synthesise the controller by looking at its reachability parametrisations.

1.2 Objectives

- Familiarity with SCOTS as a tool to aid in the synthesis of symbolic controllers with respect to various vehicular examples.
- Analyse the Symbolic Set Class [1] within the SCOTS framework and use the defined constructors and functions in parameterising various configurations of obstacles for the vehicle to navigate through to reach its target.
- Define these obstacles as a Symbolic Set / Binary Decision Diagram, an obstacle space is created in tandem with 2 other BDDs namely the state-space (the grid or the play area where the synthesis takes place) and the target space (the ”goal” or targetted end point for the vehicle as it navigates through the various obstacles.
- As the the number of obstacles drafted into the space becomes greater, there is a need for programming optimization to reduce computation time and complexity.

Chapter 2

Literature Review

2.1 Binary Decision Diagrams

A **binary decision diagram (BDD)** created by R.Bryant in 1986 is an example of a data structure which is used in computer science to depict Boolean functions [4]. These functions takes input variables of the Boolean type and produces a corresponding output. Boolean functions takes in 2 values, either 1(true) or 0 (false). The data structure is inherently binary in nature making them useful in the design of logic circuits and in our case the formal verification of a created algorithm and access its capabilities.

When there are infinitely many boolean arguments in a function, we assume this value to be 'n', we therefore have 2^n inputs [3]. Given the above notion, this enables us to choose either '0' or '1' for the output for any given input, leading to 2^n number of functions. This idea would be used in the development of the different obstacle configurations which would be elaborated more in detail later on.

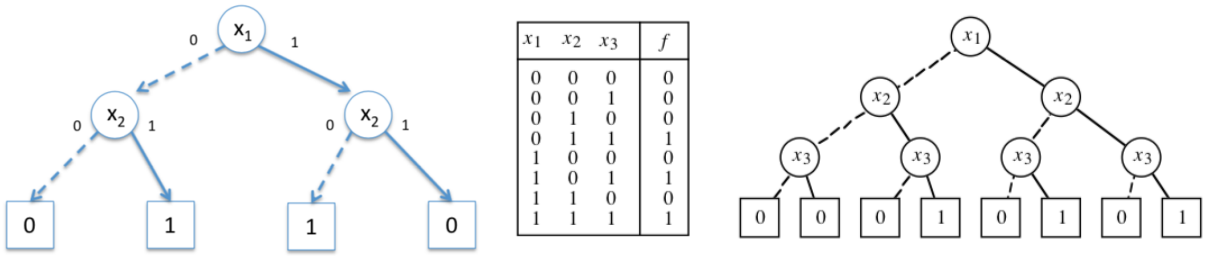


Figure 2.1: Left: Binary Decision Diagram [3]

Figure 2.2: Right: Binary tree with a Truth Table [4]

A BDD for a given variable request is an a-cyclic coordinated diagram fulfilling the accompanying properties. [9]

1. There exist only 1 unique root.
2. There exist either 1 or 2 nodes with no successor defined by 0 or 1, if there are 2 then there are defined differently.
3. Remaining nodes have a variable name and can have a maximum of 2 descendants depicted as 0-child or a 1-child. The leaves connected are also named 0 and 1 respectively.
4. A descendant of a node are also labeled either 0 or 1, or by a larger value then its parent based on the variable order.
5. All descendant-closed subgraphs of the graph are non-isomorphic.

Given the following properties, BDDs evidently are derived from binary decision trees via multiple processes of compression namely [4],

1. identical subtrees are shared
2. the removal of redundant nodes, based on the flow of the diagram.

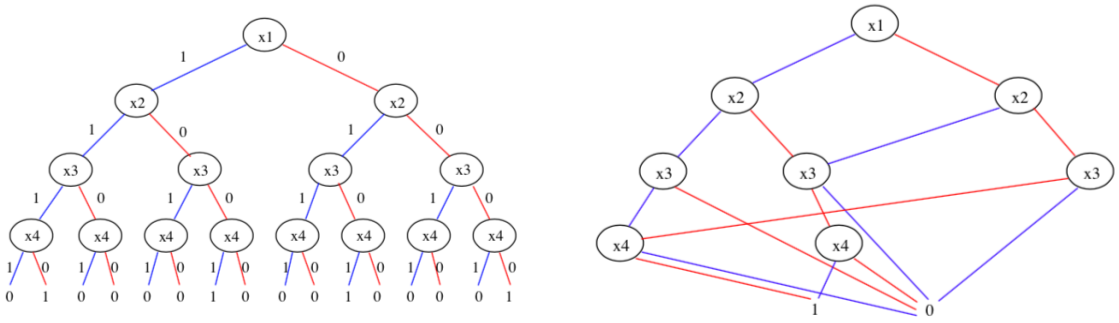


Figure 2.3: Before and after compression [3]

These properties assist in defining and lead to the creation of SCOTS [1]. Its purpose and functionality would be discussed later on.

SCOTS presents itself as an implementation of BDDs. A workflow is defined within the manual which can be found on `https://www.hcs.ei.tum.de/en/software/scots/`.

2.1.1 Application of Binary Decision Diagrams

Up till recently, Binary Decision Diagrams have been applied in the design of digital circuits, the formal verification of models and proving of model specifications [4]. SCOTS embodies this philosophy as it sets to allow for the synthesis of symbolic controllers by defining specifications such as [1] reachability, avoids and safety for example.

As Binary Decision Diagrams are a data structures which hold a number of boolean functions, they therefore can also perform operations similar regular boolean algebra [4] Such boolean operations which are well documented in many research and digital design include the **OR, AND, XOR, XNOR** operations. These may extend to the concept of Ordered Binary Decision Diagrams,[4] which in brief refers to the decision diagrams that are reduced as much as possible by following a specific order, this is done via patterns or manual manipulation based on the number of variables or inputs. This or other OBBDs can be used for formal verification of symbolic systems via sequential

analysis.

2.1.2 Boolean Functions Explained

$$f : \{0,1\}^n \rightarrow \{0,1\}[10] \tag{2.1}$$

Boolean functions is amongst the prioritised learning subjects for discrete mathematics, which extends to the concept of mathematical logic [11]. They assist in the development of algebraic propositions. It was later realised that boolean functions became useful in providing representations for control systems, as we see in SCOTS, via the application Symbolic Sets / BDDs.

Ryan O'Donnell quotes the domain of a boolean function for the definition of a hyper-cube/n-cube/Boolean Cube or Discrete Cube based on equation **2.2** which is known as a Hamming Cube [10] Hamming is a well known name amongst the field of information/coding theory as well as cryptography.

$$f : \{-1,1\}^n \rightarrow \{-1,1\}[10] \tag{2.2}$$

Boolean functions may be described using truth-tables, a formulae or via graphical/circuit based diagrams [12]. With respect to SCOTS, a graphical or flow diagram representation is used in the form of a directed acyclic graph, with vertices and nodes labeled either '0' or '1', which spured the idea of Binary Decision Diagrams [4].

2.2 Symbolic Model Theory

2.2.1 Definition

Symbolic models represents system in terms of the way they function, often through time and over a state space, these models are invariably mathematical [13]. Finite state models of developing systems expand exponentially when more components are introduced, making the formal verification more strenuous. **Symbolic model checking** [14] was introduced to deviate from building state graphs by utilising BDDs which constitute Boolean functions to describe sets and relations. It is necessary to recognise the model in some way forecast possible outcomes and results, which allows us to filter them as plausible or invalid based on the model's specification. A technique is needed to prove the validity of the outcomes.

An implementation similar to symbolic model checking is the use of **Ordered Binary Decision Diagrams** [14]. BDDs in this case can be utilised to solve equivalence problems arising from Boolean algebra/equations. The SCOTS tool employs these concepts to allow for quick controller synthesis based on the BDDs implemented within the system.

The existence of symbolic models has led to the research of various model checking methods, such as the **temporal logic** [1, 14]. Temporal logic is a concept of accounting for any changes in time. Within a system populated with atomic propositions or different transitional states, temporal logic proposes the rules and symbols for those states or propositions that are time dependent. SCOTS has the ability to synthesise controllers for differential equation based systems, by defining elaborate specifications for the **linear temporal logic** model. [1].

Linear Temporal Logic is model dependent on future time, it specifies a system as being dynamic, which is the case with SCOTS when we want to synthesis a controller where the obstacles that are created are randomly being added and remove with respect to time.

2.2.2 Symbolic Controller Synthesis

In the SCOTS environment, a system is depicted as a triple $S = (X, U, F)$, X defines the state alphabet, while U defines the input alphabet, which valued sets and the transition function $F : X \times U$. The procedure for the symbolic synthesis are listed below [1].

1. We let (S_1, Σ_1) be a control problem, defining S_2 a finite state system to replace S_1, along with an abstract specification Σ_1 is tabulated. S_1 and S_2 are defined as plant and symbolic model respectively.

2. C_2, a feedback composable controller with S_2, this provides a solution for the (S_2, Σ_2) control issue.

3. We assume that the controller C_2 is synthesised successfully, C_2 is then refined to C_1 to therefore solve (S_1, Σ_1).

The belief in this technique is theoretically valid where we relate the plant $S_1 = (X_1, U_1, F_1)$ and the symbolic set $S_2 = (X_2, U_2, F_2)$ using a feedback refinement relation $Q \subseteq X_1 \times X_2$.

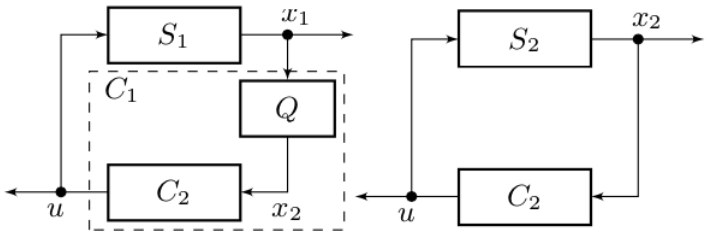


Figure 2.4: Block Diagram for the Controller Synthesis [1]

2.3 Discrete Abstractions in Symbolic Models

With the development of SCOTS, Dr. Rungger and Dr. Zamani aims to provide a solution to controller synthesis for non-linear control systems, which are derived from symbolic models or **discrete**

abstractions [1]. It is important to study abstractions in some detail as it is a topic amongst people in the computer science field, which has now seen its application in hybrid control systems.

2.3.1 Definition

Abstraction within software engineering can be viewed as the thought process when trying to solve a complicated stage of a described problem. At this roadblock, we would then return to the initial perceptive thoughts of the subject matter, and disregard irrelevant information [15]. This in turn enables the the developer to conjure concepts of how to solve the problem, rather then rely on the details at the appearance of the problem. We would want to regard different things or objects as similar, this may reduce the complexity of the problem analysis identifying common characteristics and differentiate the rest. The concept of abstraction is presented to the learning mind as an ability that can help not only in developing more in-depth/concise programming solutions, but to in solving problems of any nature [15].

2.3.2 Discrete Abstraction-based solutions

Symbolic control solutions involve discretized synthesis methods based on symbolic abstractions [16] i.e. States and input variables are symbolised as sets within the dynamical system. The existence of symbolic abstractions has led to numerous research into ways of computing them efficiently and accurately.

Jun Liu and Necmiye Ozay quoted that abstraction-based controller synthesis would naturally converge towards hybrid feedback controllers [17]. These methods of synthesis have become more apparent amongst those within the control field, and they are performed following this criteria,

1. Define a finite number of abstractions that represent the control systems dynamics.

2. Using control methods, begin solving the discrete synthesis problem with respect to its abstraction specifications to acquire a discretised control methodology.

3. Perform refinement analysis on the proposed synthesised controller a hybrid controller, with the hope of fulfilling the systems specifications.

SCOTS embodies these principles, where the synthesised controller is built upon feedback refinement relation [1], however it is different as it utilised quantized state information as compared to exact state information. Formal verifications are conducted to minimise the number of complexities so as to meet the project specifications [18]. The refinement relation as quoted is by allowing 'more 'input variables and producing less output variables'.

SCOTS also aims to enforce reachability specifications [1]. There has been extensive work using discrete abstractions with regards respect to reachability analysis, such as by Gieselmann et. al, whom utilises Piece-wise affine ODEs to aid in their analysis of genetic regulatory networks, which can hopes to solve reachability specifications within biological systems [19]. The computation can be detailed into these steps

1. A defined state-space is broken into hyper-rectangles where the differential of solution with respect to time produces the negative or positive sign.

2. By sectioning the state-space, it propels towards the creation of a discrete abstraction which converges towards to transitional system, which can offer greater qualitative detail with regards to the systems dynamics.

3. A set of regulations are given for the synthesis of the aforementioned discrete transitional system referencing to the constraints [19].

When SCOTS performs the controller synthesis, and **Abstract Domain** is drafted from the combination of key elements;

Table 2.1: Abstract Domain Elements

Domain	Grid Parameter	BDD Variable IDs with and instance of Symbolic Set

2.4 Parametric, Program Reachability and Synthesis

Parametric reachability by design is to deduce all path constraints within the system. As the name implies, along with the the symbolic synthesis, we are required to parameterise or gauge the the number of possible ways for a system to reach its target. In the case of SCOTS, we would want to synthesise a controller where the vehicle can reach its intended target despite the configuration/layout/or number of the obstacles in the state space. In turn, we also would also parameterise the obstacles into a BDD/Symbolic Set titled the Obstacle Space.

Program synthesis and program reachability may be generalised as verification problems [20]. The synthesis of controllers in SCOTS can be seen as a form of formal verification, by looking at a systems invariance and reachability specifications [1]. Another way of assessing reachability within a system is also by observing the order of functions when they terminate. [20].

Reachability synthesis seeks to find values for a set of program variables in the event the during execution, the program reaches a defined point or coordinate. With larger programs, synthesis can be more complicated as it require to perform more testing; every new measurement or test conducted will make it more difficult to acquire consistent results [21].

SCOTS allows the synthesis of a controller which implements reachability can be computed from fixed point computations [1]. There exist a pair of algorithms to perform the synthesis. A reachability problem is used to explain the methodology, in point 2.3: Synthesis via fixed point computations.

2.5 Recursive Functions

For computations which are iterative in nature, it is good habit and practice to utilise recursive functions [28]. Recursive functions is described as process where functions would call themselves again in a sequence until a prescribed condition is met. Each definition of a recursive function has what is defined as a **base case** for which the iteration begins.

An example of applying recursion is when performing the **factorial (!)** tabulation.

Listing 2.1: C++ Factorial

```
1  #include <iostream>
2  using namespace std;
3  //Factorial function
4  int f(int n){
5      if (n <= 1)
6           return 1;
7      else
8          return n*f(n-1);
9          }
10 int main(){
11     int num;
12     cout<<"Enter a number: ";
13     cin>>num;
14     cout<<"Factorial of entered number: "<<f(num);
15     return 0;
16 }
```

Referencing to the title of this thesis and its objectives, we are to develop obstacles within our vehicle arena, learning and applying the appropriate methods. We may use recursion to call the **addPolytope()** function to produce the desired number via the the 2^n formula as each obstacle in itself is boolean in nature, where it is either **'1' or '0'**.

2.6 SCOTS

2.6.1 Background - Workflow

Written by Dr. Matthias Rungger and Dr. Majid Zamani, SCOTS is an open source software tool developed to facilitate the synthesis of controllers for symbolic models [1]. The tool was contrived from C++ and has a MATLAB general user interface to aid in the visualisation of the controller in action [5]. Being an open source application, the tool encourages researchers to add more functionality to the tool to suit their needs respectively.

The tools within SCOTS utilise the CUDD Libary [22]. This library assist in the building various types of decision diagrams, one of which is BDDs, which are the data structures for SCOTS. [1]

Amongst the literature review, there is an inherent relation between control and program synthesis. The SCOTS tool is to allow for the formal verification of software based control systems [5] in the most elementary form possible. This eases the understanding of symbolic controller synthesis for users.

A controller synthesis begins with deciphering control problems, and SCOTS computes controllers based on the following mathematical relation.

$$\dot{\xi}(t) \in f(\xi(t), u) + [[-w, w]][5] \tag{2.3}$$

The aforementioned equation specifies system of non-linear nature. These control systems has a number of variables namely time t. This futher supplements the idea of temporal logic [21], where in practice, the robots final path of navigation is dependent on future time, where the obstacles

are parameterised via 2^n, giving the obstacle space configuration various possibilities, thus adjusting robot trajectory accordingly. With a constant value input u, these become parameters for the continuous function ξ. $[[-w,w]]$ indicates the hyper-rectangle, a generalisation of rectangles for larger dimensions, defined by the cartesian product of intervals (source: **SymbolicSet.hh**). This is one of the constructors within the **Symbolic Set Class**, which would be further discussed in the respective section.

Quoting from the robot example written by Mahmoud Khaled, the vehicle is represented with the following Ordinary Differential Equation. State space variable X and an input variable U is used here. (Source: **robot.cc**)

$$\begin{aligned} x_0 &= x_0 \\ x_0 &= x_0 - 1 \\ \dot{x}_0 &= u_0 \\ \dot{x}_1 &= u_1 \end{aligned} \tag{2.4}$$

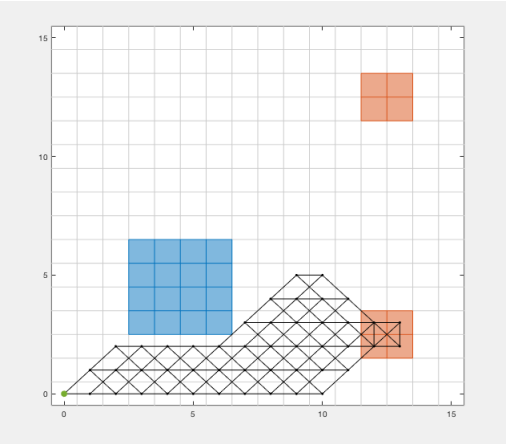


Figure 2.5: Visualisation of Synthesised Controller for robot example [1]

Referencing image 2.5 above, a grid parameter of 15x15 is defined as the working area or the state space for the controller synthesis. The red box represent the 'goal' of the vehicle and the blue box is an area as non-accessible, and the lines that branch out from the origin (0,0) details the path

of the robot which it can take, these grid points are classified as reachable. These functions would be further elaborated within the Symbolic Set section of this report.

2.6.2 The Symbolic Set Class

With the assignment of performing parametric reachability synthesis, a further study of the Symbolic Set Class is required as it is the program that facilitates the construction of the various Binary Decision Diagrams for the controller synthesis. The program written by Dr. Matthias Rungger can be found in the appendix.

BDDs are data structures within the SCOTS framework [3]. With this in mind, SCOTS is inherently Objective Oriented, this is clearly reflected in a number of constructors within the Symbolic Set Class in SymbolicSet.hh which would now be detailed below. These aid in the definition of new BDD within the framework.

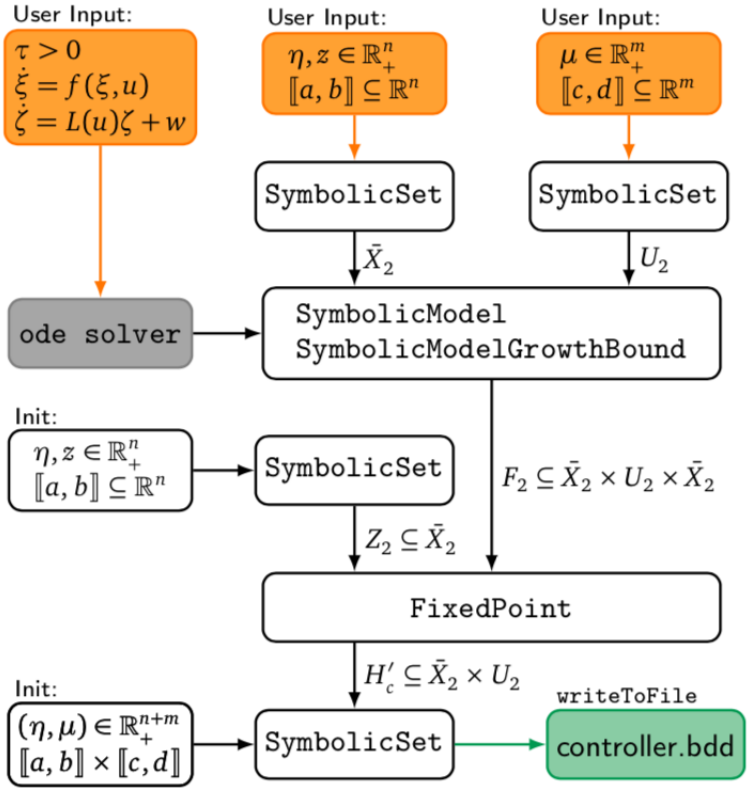


Figure 2.6: Flow Chart for SCOTS, Author: Matthias Rungger. [5]

1. **Constructor 1: Uniform domains for the hyper interval and the grid parameter** - When using SCOTS, it is important to define the dimensions of the working area or known mathematically, the state space[1], they are represented using the following mathematical statements. They refer to definition of input, output, and relevant state variables that is referenced to ordinary differential equations to define the system. The SymbolicSet.hh created by Dr. Matthias Rungger details the subsequent constructors in exhaustive detail [23].

$$\begin{aligned} \eta &\in \mathbb{R}^n_{>0} \\ [[a,b]] &\subseteq \mathbb{R}^n \\ \mu &\in \mathbb{R}^n_{>0} \\ [[c,d]] &\subseteq \mathbb{R}^n \end{aligned} \tag{2.5}$$

A state within a control system points towards defined state variables which characterise the system and describes its reaction to input variables. [24]. With reference to figure 2.4, we have a feedback control system. Common dynamical systems can be categorised with these mathematical equations [6]. The SCOTS tool requests the user to define the necessary input variables for synthesis of the controller.

$$\begin{aligned} \dot{x}(t) &= Ax(t) + Bu(t) \\ y(t) &= Cx(t) + Du(t) \end{aligned} \tag{2.6}$$

A and B are categorised as the dynamics and input matrix respectively. C and D are defined as output/sensor matrix and the feedthrough matrix [6]. **However, SCOTS is a tool to synthesise controllers of nonlinear systems, therefore the above equations are non-applicable.**

2. **Constructor 2: Read symbolic set from file** - An algorithm performs the comparsion of BDD variable IDs. When new variable IDs are created, the program would update the variable IDs to match the newly created IDs and would then load the file for example **file.bdd**.

3. **Constructor 3: Providing IDs to newly generated BBDs, retaining IDs if BDDs were drafted from other sources.**

4. **Constructor 4: The symbolic set is now displayed based on the dimension and grid parameters defined at the earlier stage [1].**

5. **Constructor 5: The cartesian product of 2 grids from 2 seperate Symbolic Sets** - At this stage of the compilation, a **controller.bdd** is created, which represents the synthesised controller to be visually depicted via the MATLAB interface [1].

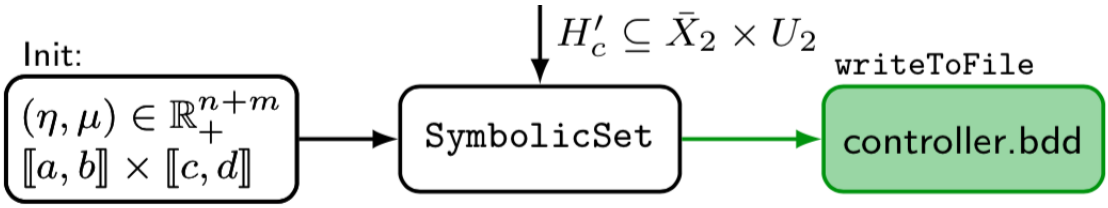


Figure 2.7: Cartesian product section, Author: Matthias Rungger. [5]

2.7 SCOTS Simulation

Within the SCOTS folder, there are various number of examples prepared by Dr. Matthias Rungger, which greatly facilitates the ease of comprehending the tool as a whole. A manual is drafted to allow the user to adopt the necessary files and libraries to run the software tool smoothly. We would now demonstrate the procedure of running SCOTS tool using 2 key examples namely the unicycle and vehicle1 example. These files may be obtained from the following URL `https://www.hcs.ei.tum.de/en/software/scots/` A general workflow could be summarised as follows.

1. Develop program in C++.
2. Open a terminal within your operating system where the C++ file lies, and compile using a makefile, which dictates the necessary file paths for where the CUDD Library [4] and MATLAB is installed.
3. Based on the program developed, there would be a number of bdd files created which would be then used to draft the (.m) file for MATLAB, which would then be used to visualise the controller in graphical format.

The tool welcomes expansion and manipulation, allowing the user to tweak to the needs of their project, or to spur on further development for future uses.

2.7.1 Examples of SCOTS Controller Synthesis

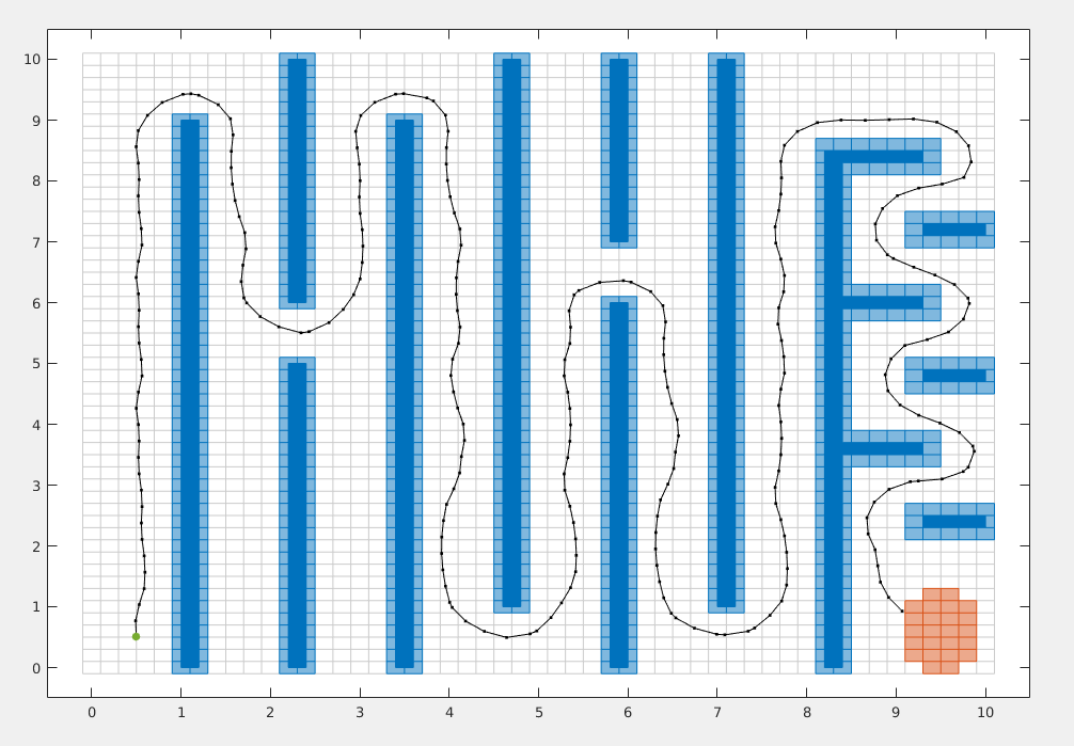


Figure 2.8: Unicycle Example from **unicycle.cc**, Author: Matthias Rungger. [1]

Dr. Matthias Rungger wrote a control synthesis problem for a unicycle vehicle. The program in C++ dictates a set of ordinary differential equations to represent the system **(Source: unicycle.cc)**. At the beginning of each program, it is important to include the necessary file libraries. There are a number of header files which are unique to SCOTS, and they can be seen in the following C++ snippet extracted from the **unicycle.cc**,

2.7.1.1 Unicycle Example

Listing 2.2: Header Files in unicycle example

```
1  /* unicycle.cc created on: 21.01.2016      author: rungger */
2  /* information about this example is given in the readme file*/
3  #include <array>
4  #include <iostream>
5  #include "cuddObj.hh"
6  #include "SymbolicSet.hh"
7  #include "SymbolicModelGrowthBound.hh"
8  #include "TicToc.hh"
9  #include "RungeKutta4.hh"
10 #include "FixedPoint.hh"
```

$$\begin{aligned} \dot{x}_0 &= u_0 \cos(x_2) \\ \dot{x}_1 &= u_0 \sin(x_2) \\ \dot{x}_0 &= u_1 \end{aligned} \tag{2.7}$$

Using the Symbolic Set Class written by Dr. Rungger, it is imperative to define a number of BDDs to aid in the visualisaton of the close loop within MATLAB, they predominantly are **(Source: unicycle.cc)**. A makefile is created to perform the compilation of the program, which produces a number of BDDs which are listed below. A readme file is prepared to assist new users of its functionality.

1. **State Space** - with reference to **SymbolicSet.hh** and the manual, it is imperative for the user to define the state space dimensions (state space variables), in particular the grid parameters and the hyper-interval for the synthesis problem, this defines the working area for the vehicle, the target, and as a whole, the area the synthesised controller is based on. A BDD is created and written to a file named **unicycle_ss.bdd**

Listing 2.3: **(Source: unicycle.cc)** State Space creation

```
1 /* construct SymbolicSet for the state space */
2 scots::SymbolicSet ss=unicycleCreateStateSpace(mgr);
3 ss.writeToFile("unicycle_ss.bdd");
4
5 scots::SymbolicSet unicycleCreateStateSpace(Cudd &mgr) {
6 double lb[sDIM]={0,0,-M_PI-0.4};
7 double ub[sDIM]={10,10,M_PI+0.4};
8 double eta[sDIM]={.2,.2,.1};
```

2. **Target Space/Set** - this refers to the 'goal' of the vehicle, this reflected in **Figure 2.8** by the red-coloured oval-shaped polygon. In similar fashion a BDD is written named **unicycle_target.bdd**

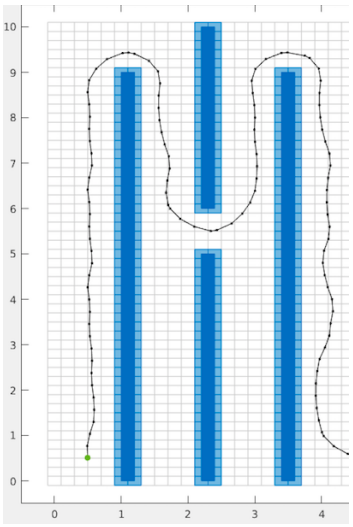
```
/*****************************************************************************/
/* the target set */
/*****************************************************************************/
/* first make a copy of the state space so that we obtain the grid
 * information in the new symbolic set */
scots::SymbolicSet ts = ss;
/* define the target set as a symbolic set */
double H[9]={ 2, 0, 0,
              0, 1, 0,
              0, 0, .1};
/* compute inner approximation of P={ x | H x<= h1 }  */
double c[3] = {9.5,0.6,0};
ts.addEllipsoid(H,c, scots::INNER);
ts.writeToFile("unicycle_target.bdd");
```

Figure 2.9: Target defined in **unicycle.cc**, Author: Matthias Rungger. [1]

The configuration of state space is mimicked for the the target space, as we want to retain the same working area. Within **SymbolicSet.hh**, there 2 functions responsible for adding or removing grid-points in the form of polygons within the state space and they are **addPolytope/remPolytope** and **addEllipsoid/remEllipsoid** [5]. These functions are based on the following mathematical relations.

$$P := \{x \in \mathbb{R}^n | Hx \leq h\}$$
$$E := \{x \in \mathbb{R}^n | |L(x-y)|_2 \leq 1\} \tag{2.8}$$

Referencing to the program above, the **addEllipsoid** function is employed to indicate in the state space where the 'goal' of the vehicle is. The blue maze is constructed using the **remPolytope** function, a figure below would demonstrate its use case.



```
/* remove outer approximation of P={ x | H x<= h1 } form state space */
double h1[4] = {-1,1.2,-0, 9};
ss.remPolytope(4,H,h1, scots::OUTER);
/* remove outer approximation of P={ x | H x<= h2 } form state space */
double h2[4] = {-2.2,2.4,-0,5};
ss.remPolytope(4,H,h2, scots::OUTER);
/* remove outer approximation of P={ x | H x<= h3 } form state space */
double h3[4] = {-2.2,2.4,-6,10};
ss.remPolytope(4,H,h3, scots::OUTER);
/* remove outer approximation of P={ x | H x<= h4 } form state space */
double h4[4] = {-3.4,3.6,-0,9};
ss.remPolytope(4,H,h4, scots::OUTER);
```

Figure 2.10: remPolytope function [5]

Remaining in the idea of the Symbolic Controller Synthesis, we would want to classify a controllers specifications based on it reachability. By removing the grid-points via the **remEllipsoid** function, the controller synthesis is made aware that those areas are unreachable, in abstract almost like creating an obstacle for the controller to avoid to finally reach its 'goal'.

In this project, we are to create the obstacles via using the addPolytope function, therefore grid-points are added to the symbolic set but make these grid points unreachable, more be discussed in the the implementation section.

2.7.1.2 Vehicle1 Example

Within the SCOTS tool, there are a functions that assist model checking and formal verification. During the compilation stage, the terminal can detail out the project description by using the **printInfo()**

function [5]. This function displays the control systems specifications/information, the figure below showcases a terminal output using the aforementioned method.

```
Unfiorm grid details:
First grid point: 0 0
Last grid point: 5 5
Grid node distance (eta)in each dimension: 1 1
Number of grid points in each dimension: 6 6
Number of grid points in domain: 36
Number of elements in the symbolic set: 36
Number of binary variables: 6
Bdd variable indices (=IDs) in dim 1: 0 1 2
Bdd variable indices (=IDs) in dim 2: 3 4 5
```

Figure 2.11: printInfo() function from **robot.cc** [1]

The program terminal can also display the overall time taken for program to synthesize the controller, this is achieved using the **TicToc.hh** program written by Dr Matthias Rungger. In formal verification of models, it is necessary to achieve a balance between speed and accuracy or computations, which ties in to the concept of programming optimization [25].

```
Nurs-Air:vehicle1 jedihanistar$ make
clang++ -c -Wall -Wextra -std=c++11 -O3 -DNDEBUG -I/opt/local/include -I../../../bdd -I../../../utils  vehicle.c
c -o vehicle.o
clang++ -Wall -Wextra -std=c++11 -O3 -DNDEBUG -o vehicle vehicle.o -L/opt/local/lib -lcudd
Nurs-Air:vehicle1 jedihanistar$ ./vehicle
Symbolic set saved to file: vehicle_ss.bdd
Symbolic set saved to file: vehicle_obst.bdd
Symbolic set saved to file: vehicle_target.bdd
0.........10.........20.........30.........40.........50.........60.........70.........80.........90.........100
Elapsed time is 130.653 seconds.

Number of elements in the transition relation: 3.73168e+07
Iterations: ..................................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
..................................................................... number: 473
Elapsed time is 535.511 seconds.
Symbolic set saved to file: vehicle_controller.bdd
Nurs-Air:vehicle1 jedihanistar$
```

Figure 2.12: TicToc.hh function from **vehicle.cc** [1]

Amongst the objectives of this paper is to introduce obstacles within the space of the controller, but ultimately, we are required to derive ways to reduce the computation complexities as the number of obstacles within the space grows, this could reduce controller synthesis times and increase output efficiency.

In similar fashion, the vehicle1 example begins by detailing Ordinary Differential Equations to describe the system, authored by Dr. Mathhias Rungger in **vehicle.cc**

$$\begin{aligned} \dot{x}_0 &= u_0 \frac{\cos(\alpha + x_2)}{\cos(\alpha)} \\ \dot{x}_1 &= u_0 \frac{\sin(\alpha + x_2)}{\cos(\alpha)} \\ \dot{x}_2 &= u_0 tan(u_1) \end{aligned} \tag{2.9}$$

These equations are written into a C++ program, which is then computed using function named ode_solver. The makefile is drafted to allow compilation via a terminal, which then produces a number of BDD files namely **vehicle_ss.bdd**, **vehicle_target.bdd**, **vehicle _obst.bdd** and the synthesized controller vehicle _**controller.bdd (Source: vehicle1 folder after running make and ./vehicle in terminal)** We then proceed to MATLAB to run **vehicle.m** file to visualise the controller in graphical format.

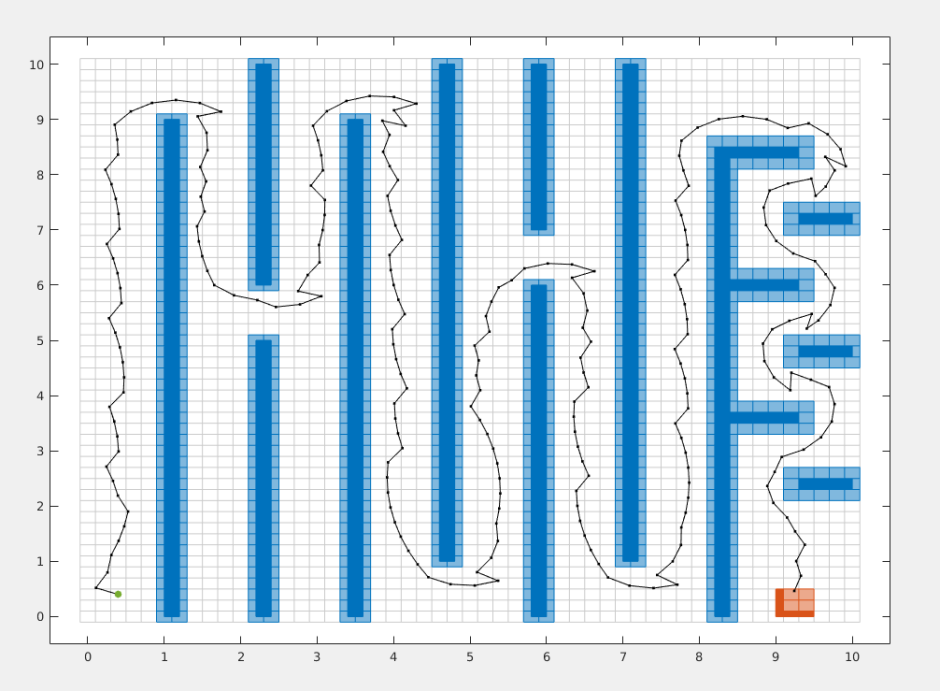


Figure 2.13: Vehicle 1 projection**vehicle.cc** [1]

Via visual comparison between unicycle and vehicle1, the obstacles placements are identical however, the difference is **vehicle.cc** uses **addPolytope** whilst **unicycle.cc** uses **remPolytope**. **Using the remPolytope essentially removes Grid-Points from the state space making those areas formed by the Polytopes as unreachable.** The movement of the vehicles are different due to definitions of their respective ODEs. The computation times are also different, the table below illustrates them.

Table 2.2: Computation Time

Example	Vehicle1	Unicycle
Initial Stage Elapsed-Time (seconds)	155.979	523.166
Transition-Relation Elapsed-Time (seconds)	586.952	529.995

It is important for software engineers to balance between speed and accuracy when creating algorithms for computers [26]. These are in direct reference on the capabilities of the computers with respect to their processing power etc. We are encouraged to control the number/values of input variables within a system as to minimise as much errors at outputs, usually surfacing from quick computation.

Chapter 3

Implementation

Based on what we have learnt thus far, we now assess the capabilities of SCOTS as per the thesis statement, which says **"Parametric Reachability Synthesis using the tool SCOTS"**. We first begin by exploring SCOTS and its functionality. We then work on modifying SCOTS to allow for parametric obstacles in the arena (i.e., encoding the obstacles as a parameter in the synthesis). Then, SCOTS will be modified to show existence of a parameter valuation, such that there is a strategy for the controller to reach the "goal". Since, the dimension of the problem grows due to the parameterization of obstacles, we might need to employ the techniques that benefit from the sparsity of the system to reduce computation complexity.

In this implementation, we perform programming additions to a number of created example authored by the Dr. Matthias Rungger, Mr Mahmoud Khaled. The process encapsulates a 'learning on the job' phase where we analyse the programs written and understand the functions/class/constructors used to aid in the development process. The end objective is to access the SCOTS tool's capability in synthesizing a controller that can work regardless of the number/position/dimension of parameterized obstacles within the synthesis work area. The vehicle or robot in question would still be able to reach its 'goal' or the target space. An analogy to describe the work is '1 to govern all possibilities'.

3.1 Beginning

We work with 3 distinct spaces, namely the **State Space (ss)**, **Target Space (ts)** and **Obstacle Space (obst)**. These spaces are also Binary Decision Diagrams or defined as Symbolic Sets. They are initialised using the following commands;

Listing 3.1: Creating Symbolic Sets / BDDs

```
1 scots::SymbolicSet vehicleCreateStateSpace(Cudd &mgr);
2 scots::SymbolicSet ss=vehicleCreateStateSpace(mgr);
3 ss.writeToFile("vehicle_ss.bdd");
4 scots::SymbolicSet obst(ss);
5 obst.writeToFile("vehicle_obst.bdd");
6 scots::SymbolicSet ts(ss);
7 ts.writeToFile("vehicle_target.bdd");
```

Both target and obstacle spaces created is based of the state space definition, we would want to retain the the configurations as it would help in performing the synthesis more uniformly. They are all saved as **bdd** files, which would then be projected via MATLAB visually. The original program had a square grid of size 10units. For the initial phase, we begin by reducing the grid dimensions to allow for quicker synthesis of controllers. Based on this, the tool synthesis time has a linear relationship with a number of variables such as the state dimension, the space between each grid-point, the number of parameterise obstacles among others. We begin to decrease the grid size in the effort to understand the tool better.

Listing 3.2: Space Definition

```
1 scots::SymbolicSet vehicleCreateStateSpace(Cudd &mgr) {
2   double lb[sDIM]={0,0,-M_PI-0.4};
3   double ub[sDIM]={5,5,M_PI+0.4};
4     double eta[sDIM]={.2,.2,.2};
5     scots::SymbolicSet ss(mgr,sDIM,lb,ub,eta);
6     ss.addGridPoints();
```

```
7    return ss;
8    }
```

The grid area are defined by boundaries, **upper(ub)** and **lower(lb)**, the distance between each grid point is stored in the variable eta. The **Cudd** refers to a package developed in C to assist in computing decision diagrams. Information can be attained via the following url `https://github.com/ivmai/cudd`.

As mentioned in the literature review, we now begin by using the addPolytope function to build the arena up with obstacles; we also define the 'goal' with the same function. The initial stages, we define a small number of obstacles and an end point to ensure the tool is functioning. The array h1[4] holds 4 values which represent minimum and maximum values of x-coordinates and y-coordinates. Together with h2 and h3, they form the 3 obstacles in this instance.

Listing 3.3: Obstacle creation

```
1  scots::SymbolicSet ts(ss);
2  double h[4] = {-4,5,-4,5};
3  ts.addPolytope(4,H,h, scots::OUTER);
4  ts.writeToFile("vehicle_target.bdd");
5  void vehicleCreateObstacles(scots::SymbolicSet &obst) {
6    double H[4*sDIM]={-1, 0, 0,
7                       1, 0, 0,
8                       0,-1, 0,
9                       0, 1, 0};
10   double h1[4] = {-0,1,-1, 2};
11   obst.addPolytope(4,H,h1, scots::OUTER);
12   double h2[4] = {-2,3,-0,2};
13   obst.addPolytope(4,H,h2, scots::OUTER);
14   double h3[4] = {-2,3,-4,5};
15   obst.addPolytope(4,H,h3, scots::OUTER);
16   }
```

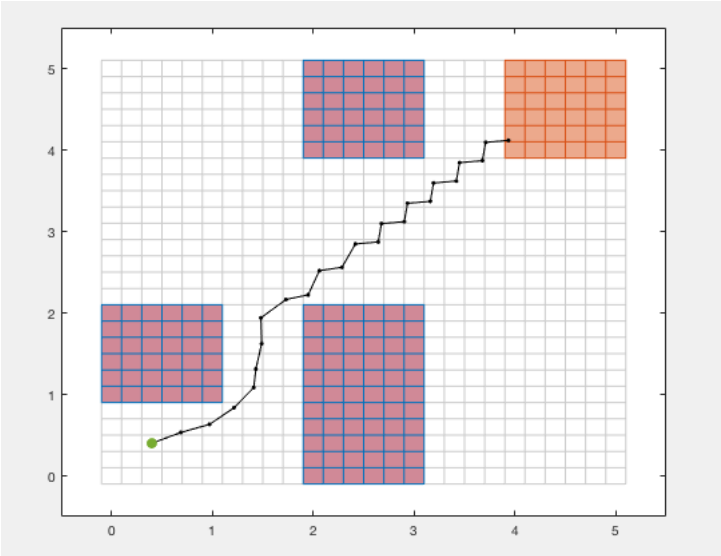


Figure 3.1: MATLAB projection of above specification **vehicle.cc** [1]

SCOTS performs controller synthesis by fixed-point computation [1].The SCOTS tool has a defined class named **FixedPoint**, which allows for controller synthesis based on reachability specifications [5]. This class is based on the **FixedPoint.hh** file written by Dr Matthias Rungger. With this class, the tool is able to distinguish which Polytopes are obstacles and which is the target. There is function named **reachAvoid** which takes both the Target Space and Obstacle BDDs as parameters, which goes into the development of the the **vehicle_controller.bdd** file. The **reachAvoid** function is defined within **FixedPoint.hh**, is to administer reachability via their invariance specifications [1]. From this, we can attempt at describing an algorithm which encodes obstacles as a parameter within the synthesis, where the number, position, and layout of obstacles are random, and assess SCOTS ability to synthesise a controller based on those parameterized obstacles.

Listing 3.4: Reachability Controller

```
1  scots::FixedPoint fp(&abstraction);
2   BDD T = ts.getSymbolicSet();
3   BDD O = obst.getSymbolicSet();
4    tt.tic();
5   BDD C=fp.reachAvoid(T,O,1);
6    tt.toc();
```

3.2 Algorithm - Learning Phase

A pseudocode is drafted in the beginning to aid in explaining the idea and methodology of this process. As stated above, the **addPolytope** function is used in the development of obstacles. In order for the obstacles to be produced at a random locations, random numbers, we could opt to encapsulate the function within a loop. The loop exits if the parameterization of obstacles exceeds the known defined state space.

Result: Parameterise the obstacles as part of the Synthesis
intial definition;
while *Still within Upper/Lower bounds of Synthesis State space* **do**
    obstacle generation;
    if *Obstacles fully populate state space* **then**
        Vehicle Movement stops/Controller Stops;
    else
        produce obstacles at random as parameter within synthesis;
    end
    using reachAvoid to synthesize controller;
end

Algorithm 1: Pseudocode for Methodology

As discussed within the introduction, SCOTS employs binary decision diagrams, which in turn describes the tool as a whole as a set of boolean-type variables and functions, hence the combination of obstacles within the state space may be in the form of 2^n [3]. Each grid point has extending vertices which could reflect the a boolean function relation with either '0' or '1'. With an increase of the hyper-rectangles dimensions, the resultant number of obstacles would therefore be greater. A discussion of boolean combinatorics is required here, to aid in deriving an algorithm that can hopefully allow for a more dynamic obstacle parameterization in the controller synthesis.

In order to allow for randomness, we may start by including a pair of libraries that allow for random number generation, and a function that calls for random numbers to be produced.

Listing 3.5: Library for random number generator [7]

```
#include <ctime>
#include <cstdlib>
srand(time(0));
```

We can encase the addPolytope function within a loop, and indicate how many such obstacles we would want to create via indicating the number of iterations via the variable **i**. It is important to begin working with the obstacles parameters with respect to their x-and-y values/coordinates, hence we label the variables for ease of comprehension. We may make the following program alterations/additions on the **vehicle.cc** example, and assess its feasibility.

Listing 3.6: Create Obstacle Function

```
void vehicleCreateObstacles(scots::SymbolicSet &obst) [
  double H[4*sDIM]={-1, 0, 0,1, 0, 0,0,-1, 0, 0, 1, 0};
        srand(time(0));
          for(int i=0; i<10; i++)
           {
              int minX = (rand()%7);
              int maxX = (rand()%7);
              int minY = (rand()%7);
              int maxY = (rand()%7);

          cout<<minX<<maxX<<minY<<maxY<<endl;

          double hTemp[4]={-minX,maxX,-minY,maxY};
            obst.addPolytope(4,H,hTemp, scots::OUTER);
           }
    }
```

Listing 3.7: State Space Definition **author: Dr Matthias Rungger**

```
1 scots::SymbolicSet vehicleCreateStateSpace(Cudd &mgr) {
2       double lb[sDIM]={0,0,-M_PI-0.4};
3       double ub[sDIM]={7,7,M_PI+0.4};
4       double eta[sDIM]={.2,.2,.2};
5        scots::SymbolicSet ss(mgr,sDIM,lb,ub,eta);
6        ss.addGridPoints();
7  return ss;
8 }
```

We may make adjustments to the upper and lower bounds of the state space, were we set a lower-bound of 0 and an upper-bound of 7, hence the numbers within each **rand()** function can output numbers between 0 and 6, which would still remain within the limits of the state space bounds. the **cout** is to allow us to print to console and ensure each stream of numbers produced are random. We can now run a few examples with these configurations and analyse the tool at computing the path for vehicle. In the first run, we iterated 10 times, and produced the following set of x-and-y values. This would then be the coordinates for the addPolytope function to create the shapes within the arena.

Table 3.1: Values Generated

6541	5015	2054	3420	5366
4616	**2525**	6632	5162	2124

When we look upon the entries **2525** and **4616** in this case, they would be traditionally defined via the addPolytope function. Based on the definition of the addPolytope, the 1st and 3rd entries represent Minimum x-and-y values, so the values have a negative sign attached to it, so in the graph, the x-component of the shape is from 2 to 5 etc.

Listing 3.8: addPolytope **source: SymbolicSet.hh**

```
1 double h1[4]={-2,5,-2,5};
2 obst.addPolytope(4,H,h1, scots::OUTER);
3 double h2[4]={-4,6,-1,6};
4 obst.addPolytope(4,H,h2, scots::OUTER);
```

The MATLAB visualization of the synthesised controller is as follows;

Figure 3.2: Random obstacle generated **vehicle.cc**

Only these 2 obstacles are valid as they concur to how the h_n**[4]** array is defined for the addPolytope function. The remaining x-and-y coordinates generated are of reverse order, therefore unable to parameterise the obstacles properly. The 2 obstacles produced are also over-lapping therefore creating 1 large obstacle in this case. With the **TicToc.hh**, it is possible to measure how long the tool takes to synthesis the controller based on the specifications given. The console quoted that the synthesis took **157.701 seconds**. With different formulas attached to the rand() function, we may be able to develop more obstacles within the space at smaller intervals, as the **eta[sDIM]**, which indicate the distance

between each grid point is **0.2**.

With every execution, different combinations of values would be produced, we can run 2nd simulation and analyse the output.

Table 3.2: 2nd set of values generated

5043	0033	6215	6542	**2534**
5202	4106	**0633**	5224	2126

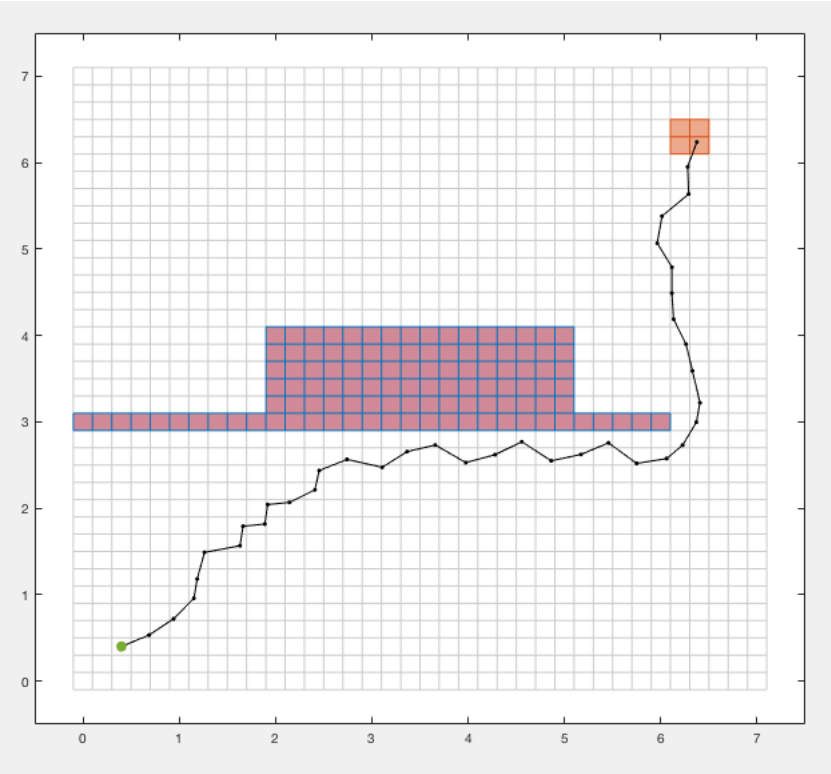


Figure 3.3: 2nd obstacle generated **vehicle.cc**

With the following parameterization of obstacles, SCOTS is still able to synthesize controller by using the **reachAvoid** function, but we are still limited to only 2 obstacles as they are the only valid options.

Below we propose a possible improvement where we ensure that the minX and min Y values would always be smaller or atleast equal to their max values, ensuring a well defined rectangle can be

established. We code the following below.

Listing 3.9: Edition to random number generator

```
1    int minX = (rand()%5);
2    int maxX = minX+(rand()%5);
3    int minY = (rand()%5);
4    int maxY = minY+(rand()%5);
```

With alteration, we can generate a greater number of obstacles as they fit within the definition of the **addPolytope** function, below is an example of the randomly generated values.

Table 3.3: Values generated in increasing order

0123	3711	0300	1411	3304

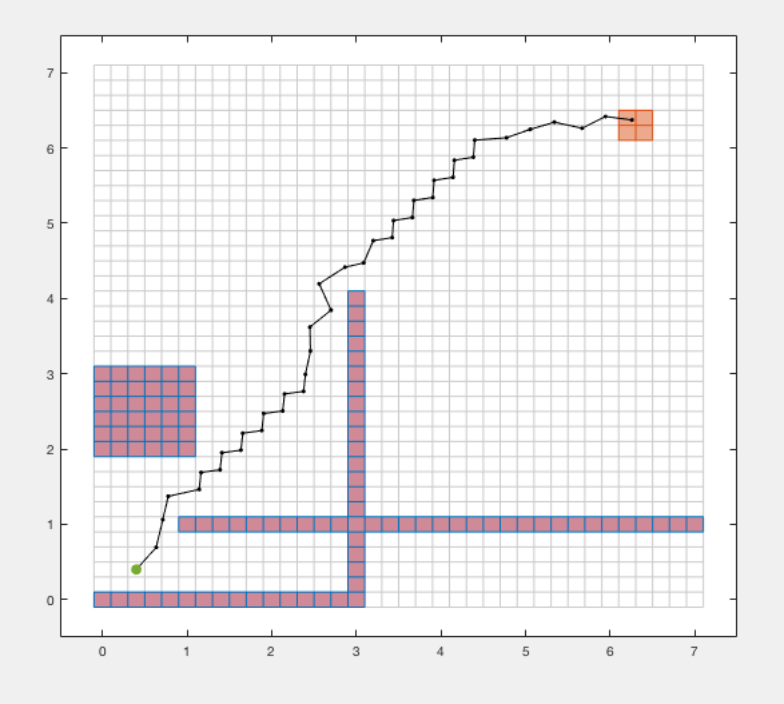


Figure 3.4: More obstacles generated **vehicle.cc**

3.2.1 Evaluation from Supervisor

Upon consultation, it is realised that the randomness factor is a 2nd phase component after we have constructed a set of Obstacles, and the paramaterization of these obstacles are generated using the 2^n relation, making them boolean functions as either they exist or not, therefore we are now tasked at creating 2^n combination of obstacles and save each configuration as their own seperate BDD file i.e. vehicle_obst1.bdd. vehicle_obst2.bdd, etc. A revised algorithm is required to carry out the process and it can be detailed below. When n is large, thus the combinations grow larger. Below details an algorithm for the aformentioned problem.

Result: Produce Obstacles files using 2^n

intial definition;

while *n* **do**

> create obstacle BDDs, save to file;
>
> synthesize a controller for each obstacle BDD;
>
> save each controller file;

end

Each obstacle in this case becomes a binary variable, where it is either present '1' or absent '0'. The value n represent the number of of obstacles, so when *n* is large, we then have 2^n possible number of obstacle combinations. If we take *n* to be 3, we will then have 8 possible combinations, increasing like binary numbers.

Table 3.4: 0 to 7 in binary

000	001	010	011	100	101	110	111

We will now use these concept for re-implementation phase. We would do our best to program as close as the specification requires and hopefully meet the project objectives.

3.3 Post-Review Implementation Phase

We will approach this implementation using the reviews from the supervisor and make improvements. Below details the workflow of this procedure, where we paramaterize the obstacles as boolean variables and meet closer to the problem specifiactions.

1. Work within the same arena dimensions i.e.Upper and Lower bounds specified in the initial learning phase.
2. Encode obstacles as binary variables more appropriately
3. Efficiently encode obstacles using the appropriate programming techniques.
4. Generate a controller for each obstacle parameterization, which would then be used at the next phase where all controllers would be consolidated as one global controller

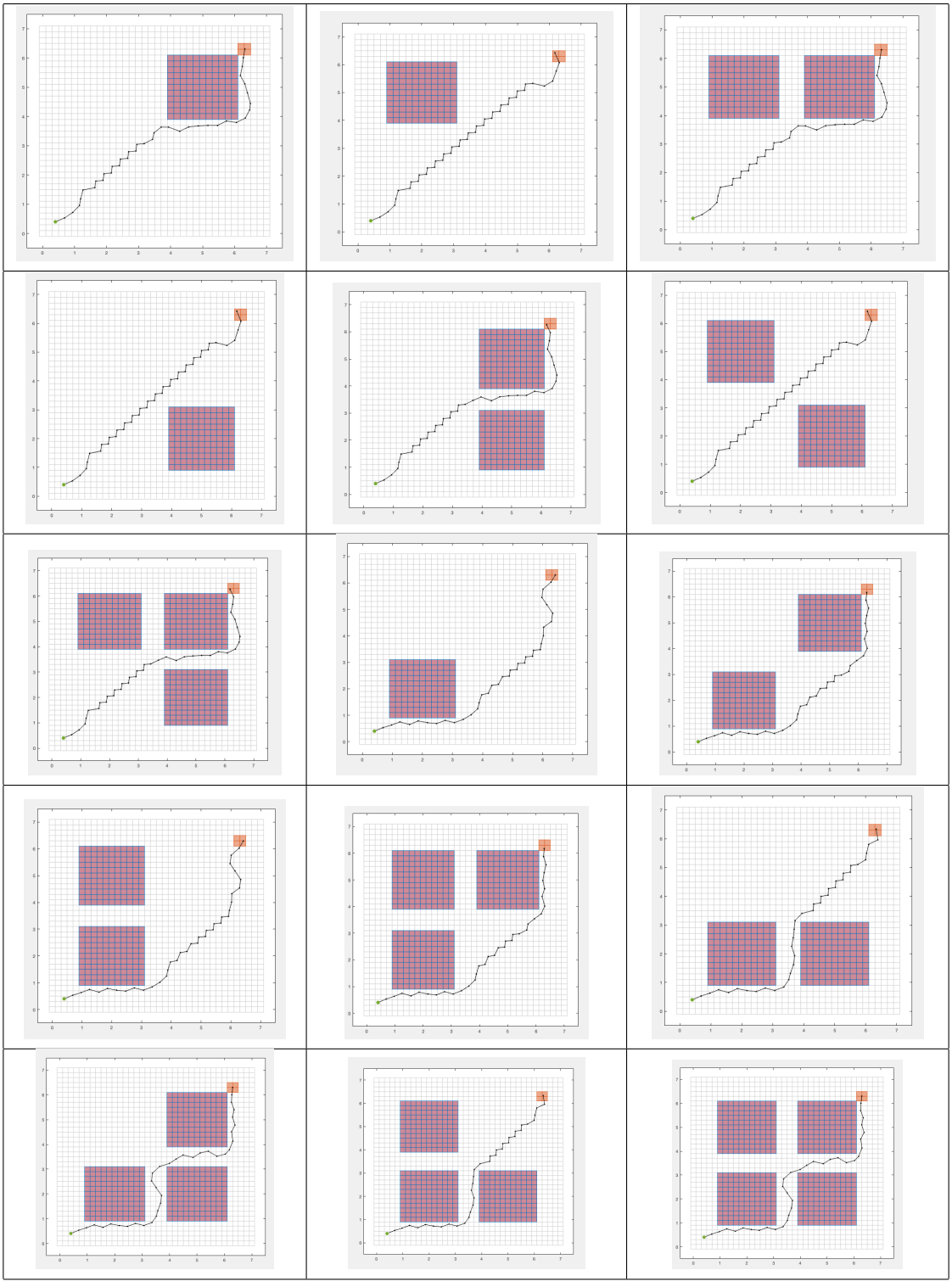
We let $n = 4$ for this instance, which would result in $2^4 = 16$ possible combinations. We have 4 bits which we use to represent each obstacle where '0' indicates off and '1' indicates as on.

Table 3.5: 0 to 15 in binary

0000	0001	0010	0011	0100	0101	0110	0111
1000	1001	1010	1011	1100	1101	1110	1111

With the following binary table as a guide, we would now visually represent the different obstacle configurations within the MATLAB interface to dictate more clearly. Each obstacle blocks are squares of dimensions 2×2 grid-points. We keep to this size and shape for simplicity before moving along to more complex polygons. We also visually display the controller created for each use case. These controllers are unique as each are synthesized based on the configuration of obstacles within the state space. The target is defined as a small polytope in the upper-right hand corner of the graph. Making the target small ensures the vehicle would only apprehend its movement only at the defined target.

Table 3.6: 2^4 Combinations of Obstacles with Synthesized Controllers



The generation of the obstacles can be done via calling the following function and defining the polytopes that make up the combinations as per binary increments. The 4 bit sequence begins from the bottom left block, and ends at the top right, which means, for 1111, the Bit 4 is the bottom left, Bit 3 bottom right, Bit 2 is Top left, and Bit 1 being top right, making an easier explanation of the binary sequencing.

Listing 3.10: Function to create obstacle configuration **1111**

```
1  void vehicleCreateObstacles15(scots::SymbolicSet &obst15) {
2  double H[4*sDIM]={-1, 0, 0,
3                     1, 0, 0,
4                     0,-1, 0,
5                     0, 1, 0};
6
7                  double h1[4] = {-1,3,-1,3}; //4th BIT
8                  obst15.addPolytope(4,H,h1, scots::OUTER);
9                  double h2[4] = {-4,6,-1,3}; //3rd BIT
10                 obst15.addPolytope(4,H,h2, scots::OUTER);
11                 double h3[4] = {-1,3,-4,6}; //2nd BIT
12                 obst15.addPolytope(4,H,h3, scots::OUTER);
13                 double h4[4] = {-4,6,-4,6}; //1st BIT
14                 obst15.addPolytope(4,H,h4, scots::OUTER);
15             }
```

However, if n increases, the number of times we call the function **void vehicleCreateObstacles()** would be 2^n times, increasing the number of lines of code, which may not be a good practice for programming. We can employ a for loop which to iterate over the function where the variable **int i** refers to the value of 2^n, so for 2^4 we will have and $i < 16$. As the value of i increases, they represent the decimal value of the binary sequence, so when $i = 12$, it would employ the necessary number addPolytope functions to produce the 1100 pattern. A program snippet is detailed below explaining this methodology.

Listing 3.11: for-loop to create obstacle configuration **1100**

```
1  void vehicleCreateObstacles(scots::SymbolicSet &obst)
2      {
3              double H[4*sDIM]={-1, 0, 0,
4                                 1, 0, 0,
5                                 0,-1, 0,
6                                 0, 1, 0};
7
8              for(int i = 0; i < 16; i++) {
9
10         ... //cases 0 to 11
11
12         else if(i == 12)
13             {
14              double h4[4] = {-1,3,-1,3}; //4th BIT
15              obst.addPolytope(4,H,h4, scots::OUTER);
16              double h3[4] = {-4,6,-1,3}; //3rd BIT
17              obst.addPolytope(4,H,h3, scots::OUTER);
18              obst.writeToFile ("vehicle_obst12.bdd");
19              break;
20             }
21
22          ... //cases 13 to 16
```

With this in place, we would be able to generate the obstacle configurations and store them as their own BDDs i.e. **vehicle_obst0.bdd to vehicle_obst15.bdd**. We would then proceed to interface with the **reachAvoid()** function to synthesise a controller for each obstacle configuration, which would then produce the follow files named **vehicle_controller0.bdd to vehicle_controller15.bdd**. The projection of both the obstacles and the controller is done via MATLAB which you may refer to Figure 3.6 as a guide. We are currently working within a small state space dimension, of a grid size 7×7, an a obstacle size of 2×2.

3.3.1 Evaluation of Post Review Implementation

With a larger state space and smaller obstacle sizes, the aforementioned implementation therefore only reflects a small scenario, and may not scale as well at higher dimensions. The parameterization of obstacles is also done via characterising them as predefined blocks, with fixed sizes. A more appropriate way is to represent them more numerically via their x-and-y coordinates. The development of the obstacle and controller files are also not created effectively. The above implementation is static in this case, may not scale well when $n \to \infty$. Using feedback from Mr Mahmoud Khaled, we progress onto the following specification for the obstacles, and assess ways on how to produce the necessary outputs in the best possible way, following this list;

1. Within the **vehicle.m** file, include in all the obstacle BDDs and controller BDDs generated for the previous case and perform the simulation. The goal of this section is to make the obstacle space dynamic, where they appear randomly, and the controller/vehicle adapts and avoids the obstacles and reach its target.

2. The initial definition of the obstacles was 2×2. It is known that the distance of each grid point or the **eta** is defined as 0.2, referencing to the code here in Listing 3.2. We were instructed to now create a 3×3 grid-point obstacle, which in turn refers to obstacle squares of size 0.6×0.6. The distance between each obstacle is also to be set to 0.6 points. With these specifications, The obstacle combination of 2^n may be much larger.

3. Following the pointers above, we aim to employ a better application of **for-loops** for the BDD generation of both obstacles and their synthesized controllers.

An alteration to the algorithm may be necessary, ensuring we can encode the parameterized obstacles within the controller synthesis more appropriately and accurately.

3.4 2nd Post-Review Implementation Phase

We make adjustments to the algorithm to incorporate the positional calculation of x-and-y values for the obstacles.

1: **for** $i = 0$ to 2^n **do**
2: **for** $x = 0$ to $x \leq UpperBound$ AND $y = 0$ to $y \leq UpperBound$ **do**
3: CreateObstacle() function call
4: Obstacle BDD , Squares of 0.6×0.6 grid-points, via addPolytope function
5: x++ and y++
6: **end for**
7: Get SymbolicSet of Obstacle
8: Get SymbolicSet of Target Set
9: reachAvoid() function call
10: Produce Controller
11: i++
12: **end for**

Table 3.7: Algorithm adjusted

The algorithm detailed above is allow us to comprehend the concept of nested for-loops [27]. It is important that we can perform correct numerical iterations to aid in generating the obstacles and their respective controller files. It is important to indicate the upper bounds as the limit, and keep the obstacles generated only within the specified state space definition referenced from Listing 3.2, which indicate a space of 7×7. Having obstacle of size 0.6×0.6, the number of obstacle combinations is definitely larger, leading to an increased computation time to synthesize controllers. It is also possible that the number of obstacles would fill up the entire state-space, which if we refer to algorithm 1, the vehicle would then stop moving. This algorithmic approach an application of recursive functions [28].

Chapter 4

Reflections

SCOTS is a versatile tool that encourages extension of functionality with respect to learning such as performing analysis; The user is allowed to create their own control problem and synthesis controllers according to their defined ODEs and State Space settings. The unicycle and vehicle examples demonstrate the use of the remPolytope and addPolytope functions defined in SymbolicSet.hh respectively. Each offers their own sets of merits in an attempt to increase efficiency and reduce computational time.

With regards to the project, we managed in some capacity to paramaterize obstacles via a boolean characterisation of 2^n. With smaller obstacle sizes whilst maintaining the same grid-parameters and dimensions, 2^n can be infinitely large. Difficulties arose at this juncture, where developing obstacles has to be done via an algorithm which would generate them systematically and they are positioned equidistant from each other, ensuring a uniform grid-like pattern, giving SCOTS more time to perform computation via fixed abstractions [1] using the **reachAvoid** function.

The development phase of this implementation involved a study and appreciation of for-loops and nested for loops, which would aid in the understanding and application of recursive functions [28]. It is key to comprehend programming techniques such as recursion, which would allow us to create the obstacles more efficiently. The method we have employed in developing the obstacles is

predetermined via inputing exact coordinates via x-and-y. As the obstacles are to be equidistant each coordinate value is offset on all sides accordingly. If recursion is employed, we may be able to create the obstacles more simply. Each x-and-y coordinate can be stored in an array as the iteration is running they are updated and a value of 0.6 grid-points is added in the positive x-and-y directions, ensuring equidistant obstacles.

1: **for** i=0 to i=2^n **do**
2: **for** x=0 to x=max, and y=0 to y=max **do**
3: xArray[i]=x+0.6;
4: yArray[i]=y+0.6;
5: x++ and y++
6: **end for**
7: i++
8: **end for**

Table 4.1: Proposed improvement

The above algorithm could be seen as abstract and non-conclusive, it is more of an illustration of an idea to incorporated recursion in the computation. It is amongst the aims of the project to always optimize programs and reduce computation complexity to increase performance in synthesizing the controllers for each Obstacle BDD created in the process. **Due to inexperience at applying recursive functions, the implementation came to a halt at this juncture within the project.**

Chapter 5

Conclusion

The aim of this project was to encode obstacle BDDs/SymbolicSets configurations as a parameter within the synthesis of the Symbolic Controller. As it is specified that the obstacles themselves are of the boolean nature, they are either to be set to the **ON** or **OF**. Quoting from the earlier section, we defined obstacles in increasing decimal value numerically, aligning the position the obstacles in a 4-bit binary sequence, to produce up to 16 obstacle configurations, which you refer to table 3.6.

We learn boolean functions are the primary data structures for Binary Decision Diagrams founded R.Bryant in 1986 [4], which in turn went into the creation of SCOTS [1]. The Symbolic Set class created by Dr Rungger and Dr Majid allows for a systematic way of generation of BDDs which help in symbolic controller synthesis. Users of SCOTS can characterise their own controller problem and input the necessary domains and differential equations to solve them.

With the concept of parametric reachability in mind, we managed in some capacity to produce the various obstacle BDDs via the 2^n relation. However the generation of this obstacles may not be able to alter dynamically as $n \rightarrow \infty$. Each obstacle has a pre-determined position and size which limits its flexibility. On a comprehension level it is a straight-forward approach but it may not be feasible when generating smaller and greater number of obstacles, hence improvements need to be made, where

recursion should be used.

As an evaluation of the proposed implementations above, it is of great importance to be adopt better programming theory and techniques so as to create more efficient and reliable solutions for larger scale obstacle parameterisations. This would go a long way in minimising computation complexity and execution times. The overall performance of the controller synthesis is dependant on the structure and flow of the program.

Finally from the point of view as a learner, there is greater understanding of how a number of aspect of engineering tie in to the thesis such as

1. **Computer Science** - Programming in C++; classes and objects, constructor(overloaded), recursive functions, memory allocation etc.
2. **Control Engineering** - SCOTS employs a feedback control loop, where the output takes part in the next iteration.
3. **Symbolic Theory** - Comprehend the synthesis of discrete/real-time controllers.

The thesis has enabled me to learn new things, which for me is certainly a positive point, with that i would like to express gratitude for the opportunity given.

Bibliography

[1] M. Rungger and M. Zamani, "Scots: A tool for the synthesis of symbolic controllers," April 2016.

[2] T. Dreossi, "Sapo: Reachability computation and parameter synthesis of polynomial dynamical systems," July 2016.

[3] F. Pfenning, "Lecture notes on binary decision diagrams," p. 15, October 2010.

[4] R. E.Bryant, "Symbolic boolean manipulation with ordered binary decision diagrams symbolic boolean manipulation with ordered binary decision diagrams," p. 35, July 1992.

[5] M. Rungger, "Scots - user manual," pp. 1–22.

[6] M. I. of Technology, "Topic 5 : Feedback control systems," *Lecture Notes*, p. 3, 2010.

[7] (2011, April). [Online]. Available: https://youtu.be/naXUIEAIt4U

[8] P. Antsaklis and Z. Gao, "Control system design," *The Electronics Engineers' Handbook, 5th Edition McGraw-Hill, Section 19, pp. 19.1-19.30, 2005.*, vol. 5th, July 2005.

[9] B. Bollig, M. Sauerhoff, D. Sieling, and I. Wegener, "Binary decision diagrams," June 1996.

[10] R. O'Donnell, *Analysis of Boolean Functions*, 1st ed. Cambridge University Press, 4 October 2014.

[11] (2014, Semptember). [Online]. Available: https://www.encyclopediaofmath.org/index.php/Boolean_function

[12] T. B. Sørensen, "Representations of boolean functions," Slides.

[13] G. Pola, A. Firrard, and P. Tabuada, "Approximately bisimilar symbolic models for nonlinear control systems," p. 16, 14 Jan 2008.

[14] K. McMillan, "Symbolic model checking: An approach to the state explosion problem," April 30 1992.

[15] O. Hazzan and J. Kramer, "Abstraction in computer science and software engineering."

[16] P.-J. Meyer, A. Girard, and E. Witrant, "Compositional abstraction and safety synthesis using overlapping symbolic models," vol. 2, pp. 1–8, 19 July 2017.

[17] J. Liu and N. Ozay, "Abstraction, discretization, and robustness in temporal logic control of dynamical systems," *Conference Paper*, pp. 293–302, April 2014.

[18] G. Reissig, A. Weber, and M. Rungger, "Feedback refinement relations for the synthesis of symbolic controllers," vol. 3, pp. 1–27, 2 Jan 2017.

[19] G. Batt, H. de Jong, M. Page, and J. Geiselmann, "Symbolic reachability analysis of genetic regulatory networks using discrete abstractions."

[20] S. Srivastava, S. Gulwani, and J. S. Foster, "From program verification to program synthesis," 17-23 January 2010.

[21] *Program Synthesis "is" Program Reachability*.

[22] F. Somenzi, "Cudd: Cu decision diagram package". in: University of colorado at boulder," 1998.

[23] M. Rungger, "Symbolicset.hh," SCOTS header file, September 2015.

[24] D. Rowell, "State-space representation of lti systems," *Analysis and Design of Feedback Control Systems*, October 2002.

[25] S. Prajna, P. A. Parrilo, and A. Rantzer, "Nonlinear control synthesis by convex optimization," *IEEE Transactions on Automatic Control*, vol. 49, no. 2, pp. 310–314, February 2004.

[26] [Online]. Available: http://www-personal.umich.edu/~mejn/cp/chapters/errors.pdf

[27] [Online]. Available: https://www.tutorialspoint.com/cplusplus/cpp_nested_loops.htm

[28] [Online]. Available: https://www.programiz.com/cpp-programming/recursion