Towards Automated System Synthesis Using SCIDUCTION

Susmit Kumar Jha

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-118
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-118.html

November 18, 2011

Acknowledgement

This research was supported by NSF grants CNS-0644436 and CNS-0627734, the FCRP/MARCO Multi-Scale Systems Center (MuSyC), Microsoft Research, Intel, and the Berkeley Fellowship for Graduate Studies from UC Berkeley.

Towards Automated System Synthesis Using SCIDUCTION

by

Susmit Kumar Jha

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering - Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Sanjit A. Seshia, Chair
Professor Claire Tomlin
Professor Dorit S. Hochbaum

Fall 2011

Towards Automated System Synthesis Using SCIDUCTION

Abstract

Towards Automated System Synthesis Using SCIDUCTION

by

Susmit Kumar Jha

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Science

University of California, Berkeley

Professor Sanjit A. Seshia, Chair

Automated synthesis of systems that are correct by construction has been a long-standing goal of computer science. Synthesis is a creative task and requires human intuition and skill. Its complete automation is currently beyond the capacity of programs that do automated reasoning. However, there is a pressing need for tools and techniques that can automate non-intuitive and error-prone synthesis tasks. This thesis proposes a novel synthesis approach to solve such tasks in the synthesis of programs as well as the synthesis of switching logic for cyberphysical systems.

The common underlying theme of the proposed synthesis techniques is a novel combination of deductive reasoning, inductive reasoning and structure hypotheses on the system under synthesis. We call this combined reasoning technique SCIDUCTION that stands for 'Structurally Constrained Induction and Deduction'. SCIDUCTION constrains *inductive* and *deductive* reasoning using *structure hypotheses*, and actively combines *inductive* and *deductive* reasoning: for instance, deductive techniques generate examples for learning, and inductive techniques generate generalizations as candidate designs to be proved or disproved by deduction.

We use the proposed synthesis approach for automated synthesis of loop-free programs from black-box oracle specifications using functions from a library of component functions, synthesizing optimal cost fixed-point code with specified accuracy from floating-point code, and synthesizing switching logic of hybrid systems for safety and performance properties. We illustrate that our approach can be used to automate system synthesis, and thus, can prove to be an effective aid to designers and developers.

To

Sumit, Mum and Papa,

The Source of Never-ending Eternal Hope and Assurance

Contents

Acknowledgments

First, and foremost, I would like to thank my parents and my elder brother, Sumit Jha, for their love and understanding. I am ever indebted to my family and friends who I have always taken for granted and whose undying faith and well wishes have never abandoned my side. Your unconditional affection shall always be my inspiration; and *sneh*, my guiding star.

I am also grateful to my advisor, Professor Sanjit A. Seshia, for making my tenure as a graduate student productive as well as filled with fun. His kind guidance and continued encouragement has made this thesis possible. I am thankful to members of my dissertation committee: Professor Claire Tomlin and Professor Dorit Hochbaum for their insightful comments and suggestions. I am thankful to all my instructors at UC Berkeley and Indian Institute of Technology (IIT) Kharagpur. In particular, I am grateful to Professor Alberto Sangiovanni-Vincentelli and the graduate student instructor, Dr. Alessandro Pinto, who introduced me to the area of embedded systems. I am also grateful to Professor Somesh Jha, Professor Michael Jordan, Professor George Necula, Professor Dawn Song, Professor Phil Spector and Professor David Wagner for stirring my interest in programming languages, formal methods, computer security and machine learning.

I am also thankful to my internship mentors at Tata Institute of Fundamental Research (TIFR), Bombay, Ecole Polytechnique Federale de Lausanne (EPFL) and SRI International: Professor R.K. Shyamasundar, Professor Thomas Henzinger, Dr. Nir Piterman, Dr. Jasmin Fisher and Dr. Ashish Tiwari. In particular, I thank Professor Shyamasundar for making me fall in love with computer science. I am also grateful to Dr. Tiwari who is a continued source of inspiration for research as well as trekking. I am thankful to Dr. John Rushby, Dr. Shankar Natarajan and other researchers at SRI International for two wonderful and productive summers at Menlo Park. I am also grateful to my collaborators: Prateek Bhanshali, Dr. Bryan Brady, Luigi Di Guglielmo, Dr. Sumit Gulwani, Dan Holcomb, Professor Trent Jaeger, Wenchao Li, Rhishi Limaye, Dr. David Molnar, Divya Muthukumaran, Dr. David King, John Kotker, Dorsa Sadigh, and Cynthia Sturton. I am thankful to Professor S. Ramesh and Dr. Swarup Mohalik at General Motors Research, Dr. Sriram Rajamani at Microsoft Research India, Dr. Mike Kishinevsky at Intel, and Dr. Aarti Gupta at NEC Research for their continued encouragement.

Last but not the least, I am thankful to the squirrels of Berkeley for inspiring me with their contagious enthusiasm, unbridled curiosity and complete absence of fear.

Chapter 1

Introduction

Automated synthesis of systems that are correct by construction has being a long standing goal of computer science and engineering. But, even today, the design and analysis of systems across different domains of hardware and software development remains a predominantly human endeavor. Developers and designers still play a central role in both hardware design and software development from the inception of the design and its specification to the final testing, deployment, and system maintenance [93]. The creation of tools that can aid the designers and developers of engineered systems remains an important goal in different areas of computer science including programming languages [19], software engineering [80], formal verification [16] and electronic design automation [80].

The complete automation of the synthesis process would be an unrealistic expectation. Software development and hardware design are inherently creative tasks which require a lot of human ingenuity and insight. Building systems often involves discovering new algorithms, designing new architectures, and orchestrating details required for building larger systems as a composition of its smaller components. The complete automation of these creative tasks seems improbable in the near future, but there is definitely a pressing need for complementing human intelligence with automated synthesis techniques in domains where manual design is tedious and error-prone.

In the recent past, a lot of progress has been made in automating the verification and testing of engineered systems, and, to a lesser extent, towards automating the synthesis of such systems. Automated verification techniques enable developers and designers to quickly verify their hypoth-

esized designs, and to discover and fix errors, if any. This undoubtedly lightens their burden by freeing them from identifying corner case scenarios, and checking their design with respect to the expected behavior manually. Automated synthesis techniques have also being proposed in niche application areas to automate mechanical aspects of the design process. But, the absence of complete specifications at the start of the design process and the computational hardness of synthesis problems, even when specifications are available, are two main reasons that have prevented a wider adoption of automated synthesis techniques.

Thus, the design and development of engineered systems still remains a *trial and error* process, especially in domains where human intuition and insight are limited due to the inherent complexity of the design problem, and the characteristic combinatorial nature of design space. We make an effort to develop an automated approach to guide the tedious *trial and error* approach to synthesis. We focus on automated synthesis for problems that are particularly challenging to human designers. Our goal is not to replace human designers and developers; rather, we seek to investigate the development of tools and techniques that can serve as an effective computational aid in their work.

Our work on automated synthesis is motivated, in part, by the recent success in automated verification of systems [108], which is chiefly driven by the advancements in deductive reasoning techniques such as constraint solving [13, 37] and automated theorem proving [72, 111]. Deduction techniques are useful in discovering corner-cases and overlooked scenarios on which a candidate design can fail with respect to a specification provided by the user. Hence, they have proved to be valuable work-horses for automated debugging to software developers as well as hardware designers. However, purely deductive methods cannot handle many complex synthesis tasks, including those considered in this thesis. Therefore, this thesis makes an effort to propose a unified theme for automated synthesis by proposing a new approach that combines deductive reasoning with inductive inference (algorithmic learning). A central idea is to use deduction to discover example behaviors of the desired system, and then use induction to synthesize the system as generalization of the discovered behaviors. Hypotheses on the structure of the system can be provided by users to aid the induction engine in the generalization step, and to constrain the search space of deduction engines.

1.1 Thesis Statement

The thesis we explore in this dissertation is the following:

The systematic combination of induction, deduction, and structure hypotheses is effective in automating tricky and tedious synthesis tasks in system design.

We call this combined reasoning technique SCIDUCTION, standing for 'Structurally Constrained Induction and Deduction' [112]. SCIDUCTION constraint *inductive* and *deductive* reasoning using *structure hypotheses*, and actively combines *inductive* and *deductive* reasoning: for instance, deductive techniques generate examples for learning, and inductive techniques generate generalizations as candidate designs to be proved or disproved by deduction.

In the rest of the chapter, we first motivate the automated synthesis problem by identifying particular applications where human insight and intuition would greatly benefit from automated reasoning support. Then, we identify the common underlying challenge in developing an automated synthesis technique for such applications. We propose SCIDUCTION as an effective solution to the challenge, and discuss how it builds on existing techniques. Then, we summarize the contributions of the thesis, and conclude this chapter by presenting the organization of the thesis.

1.2 Challenges in Automated Synthesis

In this section, we briefly present some design and programming problems that illustrate both the need for automation, and the limitations of human ingenuity. These problems are from diverse domains including software engineering, computer security, hardware design and cyber-physical systems. The common thread connecting all these problems is their non-intuitive nature. They clearly provide an opportunity to build techniques that can aid human designers and developers in practice. Our synthesis approach based on SCIDUCTION is motivated by these applications. While we detail our approach and show how they can be used to solve these problems later in this thesis, in this section we introduce these problems and mention how human experts solve these problems.

1.2.1 Motivating Examples

We enumerate the motivating applications along with a simple example to illustrate the need for automated synthesis.

Bitvector Programs

Bitvector programs manipulate bitvectors (strings of bits) using arithmetic operators such as addition and subtraction, and bitwise operators such as bitwise and, bitwise or, and right shift. These programs are usually small loop-free code, but often involve very intricate and unintuitive tricks. In fact, the upcoming 4th volume of the classic series "Art of Computer Programming" by Knuth has a special chapter on bitwise tricks and techniques [70]. The use of both arithmetic and logical operations makes these programs difficult for programmers, and they often rely on a compendium of these algorithms published by expert programmers such as Hacker's Delight [132]. But, if a particular scenario requires a new solution which has not being previously published, programmers are forced to design these bitvector programs on their own. Such a design process is both time-consuming and error-prone. Further, these bitvector programs are used in embedded systems such as network routers and other systems where performance is important. Thus, it is desirable to use as few operations as possible to accomplish a particular task.

The following is a simple example of a bitvector program from the Hacker's Delight [132].

> Given a bit-vector integer x, of finite but arbitrary length, construct a new bit-vector y that corresponds to x with the rightmost string of contiguous 1s turned off, i.e., reset to 0s.

Let us consider writing some sample input-output pairs, or examples, for the problem. For any input x, it is easy to provide the corresponding output y. Some example (x, y) pairs are $(01101100, 01100000), (10100111, 10100000), (00000000, 00000000)$ and so on.

A straightforward, but inefficient, implementation is a loop that iterates through the bits of x and zeroes out the rightmost contiguous string of 1s. We first present this simple implementation with a *while* loop in Procedure 1. The loopy implementation has $O(n)$ comparisons and $O(n)$ arithmetic operations for bitvectors of size n. Such bitvector manipulation programs are

Procedure 1 Code for turning off rightmost contiguous 1s

Input: Bitvector x

Output: Bitvector y

```
currentIndex = length(x) − 1
while x[currentIndex] ≠ 1 do
  y[currentIndex] = x[currentIndex]
  currentIndex = currentIndex − 1
end while
while x[currentIndex] ≠ 0 do
  y[currentIndex] = 1 − (x[currentIndex])
  currentIndex = currentIndex − 1
end while
return y
```

Procedure 2 Turning off rightmost contiguous 1s

Input: Bitvector x

Output: Bitvector y

```
t1 = x − 1
t2 = t1 | x
t3 = t2 + 1
y = t3 & x
return y
```

often used in embedded devices where they operate on large bitstrings such as network packets. Further, branching in the program in the form of the *while* loops makes this code less efficient than straight line code. It is well known that straight line code often shows better cache performance. Can we synthesize a shorter and more efficient implementation? It is difficult to answer this, but it is easy to speculate that the elementary operations that may be used inside such an efficient implementation will be the standard bit-vector operators: bit-wise logical operations (OR $|$, AND $\&$, XOR $\oplus$, NEGATION $\neg$), and basic arithmetic operations (ADD $+$, SUBTRACT $-$, MULTIPLY $*$, DIVIDE $/$). The program presented in Procedure 2 can solve this problem with

a straight line program using only 4 operations. Further, the solution is independent of the wordlength of the input bitvector.

A programmer will require considerable familiarity with bit-level manipulations to come up with such an implementation. In order to better appreciate the ingenuity required to come up with the program, we briefly discuss the intuition behind different operations in this program. In the first step, subtracting 1 from the bitvector turns all the trailing $0s$ to 1s. For instance, on input 0111001100, subtracting 1 returns 0111001011. In the next step, we do a bitwise OR with the initial input to get a bitvector which is the same except for the trailing 0 bits turned on. In our example, we obtain 0111001111. Adding 1 turns off all the rightmost-ones as well as switches back trailing 0s. The only difference from the final required result is a 1 after the rightmost string of contiguous 1s in the given input. In the case of our example, we obtain 0111010000. In the fourth and last-step, we do a bitwise AND with the initial input to obtain the final desired output bitvector which has the rightmost contiguous 1s turned off. For our example, we get 0111000000.

This example illustrates the level of experience and insight into bitwise operations required to write such bitvector programs. The insight required to write such bitvector programs becomes more and more involved with the length of the code which can be as long as 15 operations. It is difficult to expect common programmers to be able to come up with such non-intuitive programs on their own in a timely manner and without errors. Automatic synthesis of such bitvector programs is one of the applications of our synthesis approach, and is discussed in detail in Chapter 3.

Program Understanding and Deobfuscation

The next application that we consider is that of program understanding and deobfuscation. A very important component of software engineering is understanding programs. This is of even greater importance to security experts working on tasks, such as auditing third-party code, or understanding decompiled malware code.

The dependence on human experts for understanding code in order to release patches for vulnerable systems delays the response to new malware, and is often responsible for heavy economic losses. Use of obfuscation techniques by malware writers [135] to make programs really difficult to understand by humans coupled with inefficiencies of decompilers used to obtain source code from malware binaries, makes program understanding a very challenging task. An effective aid for

program understanding would be the ability to select a particularly puzzling snippet of code and have an automated synthesis engine *resynthesize* its simpler but functionally equivalent version.

Procedure 3 Obfuscated Code

Input: Bitvector x

Output: Bitvector y

```
a = 1, b = 0, z = 1, c = 0
while 1 do
  if a == 0 then
    if b == 0 then
      y = z + y, a = ¬a, b = ¬b, c = ¬c
      if ¬c then
        break
      end if
    else
      z = z + y, a = ¬a, b = ¬b, c = ¬c
      if ¬c then
        break
      end if
    end if
  else
    if b == 0 then
      z = y << 2, a = ¬a
    else
      z = y << 3, a = ¬a, b = ¬b
    end if
  end if
end while
return y
```

Procedure 4 Multiply by 45

Input: Bitvector x

Output: Bitvector y

```
z = y << 2
y = z + y
z = y << 3
y = z + y
return y
```

We illustrate this problem with a small example snippet of obfuscated code in Procedure 3 from the decompiled code of the Conficker worm. This piece of code has an infinite loop with non-intuitive branching in the control flow of the code. But, functionally, it is equivalent to the code in Procedure 4 which multiplies a number with 45. This simpler version of the program is synthesized using our synthesis technique presented in Chapter 3. It uses operators: `shift-left`($<<$) and `add`($+$), present in the obfuscated code as the component functions. It does not use `multiply`($*$) since it is not present in the obfuscated code.

Automatically inferring functionally equivalent but simpler version of obfuscated code is a difficult and time-consuming task, and it would be useful to build an automated technique to assist software engineers in this task. We discuss this application in Chapter 3 of the thesis, and show how our synthesis approach can aid programmers in deobfuscation.

Fixed-point Implementation of Floating-point Code

Programs written in the domains of digital signal processing and embedded systems have two important characteristics. First, they commonly contain procedures that compute functions of their inputs, where these functions are mathematically specified as operating on the reals; examples include filters used in signal conditioning and the computation of control inputs. Second, they must run in resource-constrained environments and/or at high performance, requiring their optimization for low resource cost (e.g., low power, low area) as well as for performance.

Specifically, at the high-level design stage (which could involve manually writing a "reference" program or using model-based design environments such as Simulink/Stateflow and LabVIEW),

the reals are approximated with floating-point arithmetic. Designers create signal processing or control algorithms as programs based on floating-point arithmetic. However, when these algorithms must be implemented in software, they must be optimized for power and performance. It is common for embedded platforms to have processors without floating-point units due to their added cost and performance penalty. Such platforms increasingly include hardware such as field-programmable gate arrays (FPGAs), on which fixed-point arithmetic can be efficiently implemented. The signal processing/control engineer must thus redesign her floating-point program to instead use *fixed-point arithmetic*. Each floating-point variable and operation in the original program is simply replaced by a corresponding fixed-point variable and operation, so the basic structure of the program does not change. The tricky part of the redesign process is to find the *optimal fixed-point types*, viz., the optimal bit-widths of fixed-point variables, so that the implementation on the platform is optimal — lowest cost and highest performance — *and* the resulting fixed-point program is sufficiently accurate.

We present an example in Procedure 5 that illustrates this problem and the difficulty that programmers face in doing the floating-point to fixed-point translation manually. The floating-point code in this example takes `radius` of a circle as the input, and computes the `area` of the circle.

Procedure 5 Floating-point code to compute area of circle

Input: `radius`

Output: `area`

double `mypi, radius, t, area`

`mypi` $=$ `3.14159265358979323846`

`t` $=$ `radius` $\times$ `radius`

`area` $=$ `mypi` $\times$ `t`

return `area`

In addition to the inputs of the floating-point code, the fixed-point version of the code takes the fixed-point type of the variables as an input. The fixed-point type is a 3-tuple $\langle \mathtt{s_j}, \mathtt{iwl_j}, \mathtt{fwl_j} \rangle$ for j-th variable where $\mathtt{s_j}$ denotes the signed-ness of the variable, $\mathtt{iwl_j}$ denotes the integer wordlength and $\mathtt{fwl_j}$ denotes the fraction wordlength. For example $\langle 1, 3, 4 \rangle$ denotes a fixed-point variable that is signed and has a total wordlength of 7 bits, where the first 3-bits denote the integers and the next 4 bits denote the fractional part. So, a constant 0010010 of the fixed-point type $\langle 1, 3, 4 \rangle$ denotes $\frac{16+2}{2^4} = 1.125$ in decimal.

Procedure 6 Fixed-point code to compute area of circle

Input: `radius`, $\langle \mathtt{s_j}, \mathtt{iwl_j}, \mathtt{fwl_j} \rangle$ for $\mathtt{j} = 1, 2, 3, 4$

Output: `area`

$\mathbf{fx}\langle \mathbf{s_1}, \mathbf{iwl_1}, \mathbf{fwl_1} \rangle$ `mypi`

$\mathbf{fx}\langle \mathbf{s_2}, \mathbf{iwl_2}, \mathbf{fwl_2} \rangle$ `radius`

$\mathbf{fx}\langle \mathbf{s_3}, \mathbf{iwl_3}, \mathbf{fwl_3} \rangle$ `t`

$\mathbf{fx}\langle \mathbf{s_4}, \mathbf{iwl_4}, \mathbf{fwl_4} \rangle$ `area`

`mypi` $=$ 3.14159265358979323846

`t` $=$ `radius` $\times$ `radius`

`area` $=$ `mypi` $\times$ `t`

return `area`

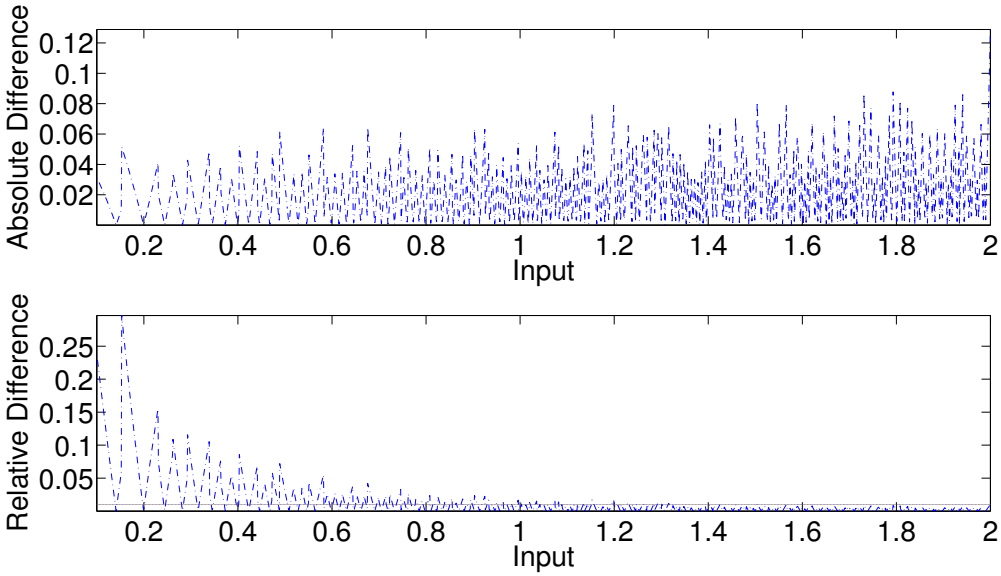


Figure 1.1: Example of Fixed-point Code with Wordlength 8

Different choice of the fixed-point types leads to different costs of the implementation as well as different accuracy of the fixed-point program with respect to the floating-point program. Typically, designers have a minimum threshold for accuracy. For example, they might require that the output produced by the fixed-point code might not differ by more than 1% from that produced by the floating-point code. While increasing the wordlength of the fixed-point variables will lead to higher accuracy, it will also lead to higher implementation cost depending on the cost model.

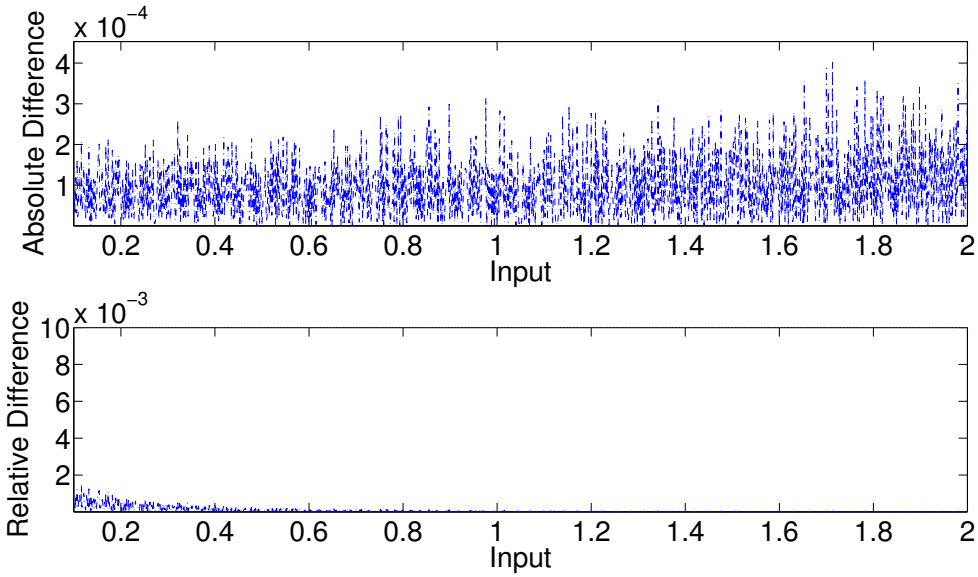


Figure 1.2: Example of Fixed-point Code with Wordlength 16

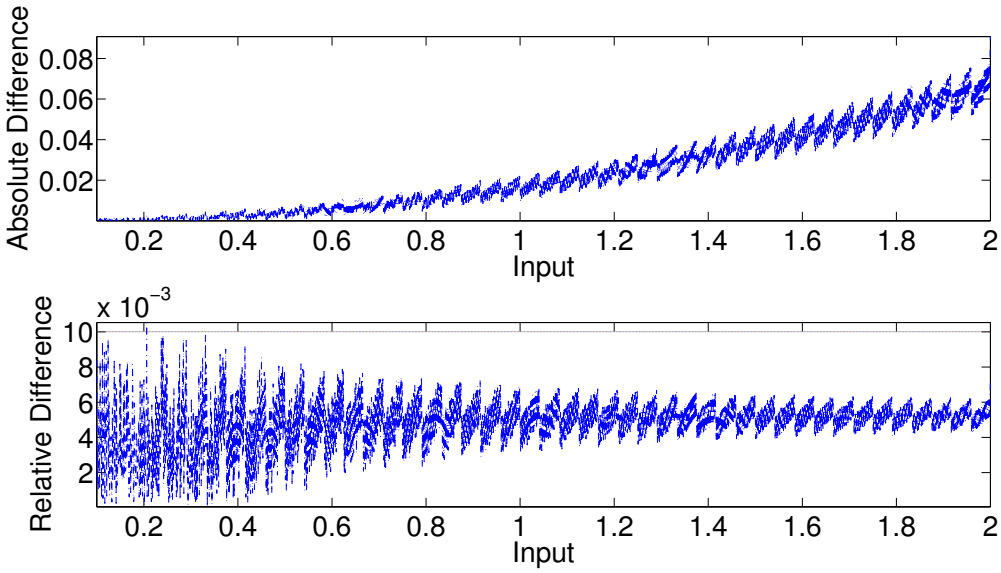


Figure 1.3: Example of Fixed-point Code Synthesized Using SCIDUCTION based Approach

Using a standard cost model that models area cost of hardware implementation [38], we observe the accuracy and the cost of the example at two extremes in Figure 1.1 and Figure 1.2. The figures plot the relative and absolute difference between the floating-point and fixed-point program with

given wordlengths for different values of radius from 0.1 to 2, which is domain of the input. The relative difference is the ratio between the absolute difference and the output of the floating-point program. The horizontal line indicates the maximum relative error threshold suggested by the user.

The wordlengths for all the four variables is 8 in the first case and 16 in the second case. We assign sufficient integer wordlength to avoid overflow, and leave the remaining bits for fractions. The fixed-point types in the first case are $\langle 0, 2, 6\rangle, \langle 0, 1, 7\rangle, \langle 0, 2, 6\rangle, \langle 0, 4, 4\rangle$. The cost for this implementation is 81.80 units. In the second case, the fixed-point types are $\langle 0, 2, 14\rangle, \langle 0, 1, 15\rangle, \langle 0, 2, 14\rangle, \langle 0, 4, 12\rangle$ and the cost of implementation increases to 316.20. The first case clearly violates the expected accuracy for a number of inputs, while the second one satisfies the accuracy requirement but has a very high cost. The optimal cost fixed-point code with acceptable accuracy lies in between these two extremes and a manual designer has to do a trial and error based search to discover this optimal design.

In Chapter 4, we describe how our synthesis approach can be used to automatically discover the optimal design with respect to a given cost model and expected accuracy constraint. For the example presented above, the optimal wordlengths for the four variables are $5, 10, 13, 14$, that is, the optimal fixed-point types for `mypi` is $\langle 0, 2, 3\rangle$, `radius` is $\langle 0, 1, 9\rangle$, `t` is $\langle 0, 2, 11\rangle$ and `area` is $\langle 0, 4, 10\rangle$. The cost for this implementation is 104.65. This was discovered using our synthesis techniques based on SCIDUCTION. The error plot for this implementation is shown in Figure 1.3.

Switching Logic Synthesis for Safety Properties

Cyber-physical systems, in which software interfaces with the physical world through sensors and actuators, are increasingly becoming ubiquitous. These systems are modeled as hybrid systems, that consist of multiple modes of operation of the physical plant. Controllers, often designed as software, control the switching between different modes of operation. Each mode of operation is a continuous dynamical system, and is relatively well-studied in literature on control theory [119, 110, 67]. But, the discrete part that accomplishes switching between the mode has often to be designed manually with very little computational support. These systems are used in transportation, health-care, and other societal-scale applications where safe operation of systems is very critical. Safety specifications of the system are often available as invariants on the continu-

ous variables that model the system state of the physical world. Synthesis of switching logic such that the system stays safe is a challenging task, and often designers need to resort to simulations with different switching conditions before identifying a safe switching logic.

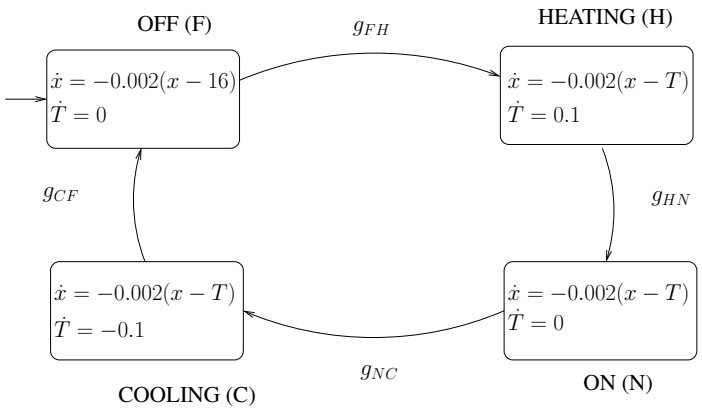


Figure 1.4: 4-mode Thermostat Controller

We illustrate it with a simple example of a 4-mode thermostat controller. It is presented in Figure 1.4. The room temperature is represented by x and the temperature of the heater is represented by T. The initial condition I is given by $T = 20$ degrees Celsius and $x = 19$ degrees Celsius. The safety property ϕ_S to be enforced is that the room temperature lies between 18 and 20 degrees Celsius, that is, ϕ_S is $18 \leq x \leq 20$. (We omit the units in the sequel, for brevity.)

In the OFF mode, the temperature falls at a rate proportional to the difference between the room temperature x and the temperature outside the room, which is assumed to be constant at 16. In the HEATING mode, the heater heats up from 20 to 22. The heater cools down from 22 to 20 in the COOLING mode. In the ON mode, the heater is at a constant temperature of 22. In the HEATING, ON and COOLING mode, the temperature of the room changes in proportion to the difference between the room temperature and the heater temperature. We need to synthesize the four guards: g_{FH}, g_{HN}, g_{NC} and g_{CF} such that the system must respect the safety property on the room temperature x.

We discuss how our SCIDUCTION based synthesis approach can be used to solve this problem in Chapter 6. The final guards synthesized by our technique are as follows.

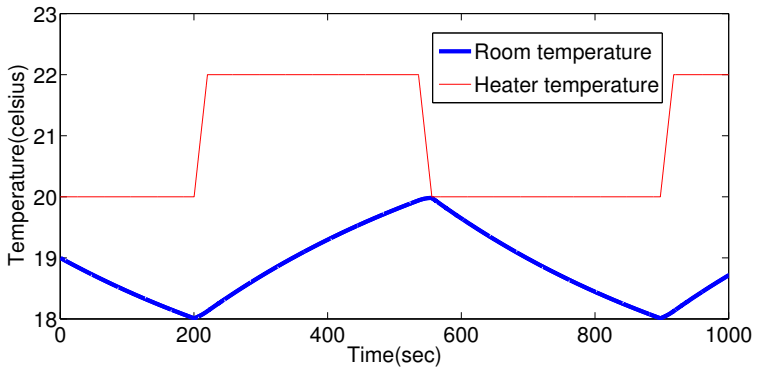


Figure 1.5: Behavior of Thermostat Controller Synthesized for Safety Specifications

$$
\begin{aligned}
g_{FH} &: 18.00 \leq x \leq 18.01 \\
g_{HN} &: 18.00 \leq x \leq 18.26 \\
g_{NC} &: 19.94 \leq x \leq 19.95 \\
g_{CF} &: 19.65 \leq x \leq 20.00
\end{aligned}
$$

The behavior of the synthesized thermostat for the first 1000 seconds from the initial state is shown in Figure 1.5. Our technique can aid designers by synthesizing switching logic for a given safety specification.

Switching Logic Synthesis for Optimal Performance

Besides the safety properties, designers often want to synthesize switching logic for controlling the switching through different modes of operations of cyber-physical systems such that the resulting system has optimal behavior with respect to a given cost metric. The cost metric may be provided as a function over the values of the continuous variables used to model the continuous plant that the controller interacts with. Further, the optimality may be desired over the *long-term* behavior of the system as opposed to a finite horizon behavior of the system upto a fixed time.

We briefly explain this using a thermostat controller example similar to the one considered in Section 1.2.1. This three mode dynamical system is presented in Figure 1.6. The temperature is

recorded in Celsius, and time is measured in minutes.

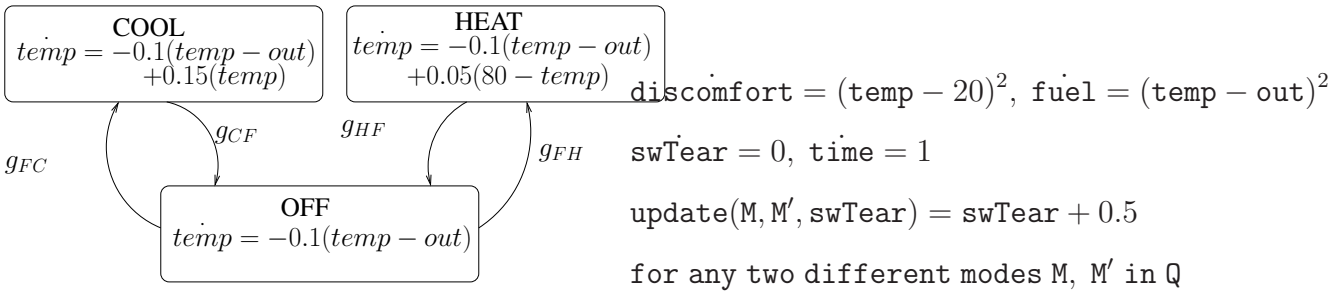


Figure 1.6: 3-Mode Thermostat Controller

The performance requirement is to keep the temperature as close as possible to the target temperature 20, *and* to consume as little fuel as possible in the long run. We also want to minimize the wear and tear of the heater caused by switching. The performance metric provided by the user is given by the tuple $\langle \texttt{PR}, f_{\texttt{PR}}, \texttt{update} \rangle$, where penalty variables $P = \{\texttt{discomfort}, \texttt{fuel}, \texttt{swTear}\}$ denote the discomfort, fuel and wear-tear due to switching and reward variables $R = \{\texttt{time}\}$ denote the time spent. The evolution ($f_{\texttt{PR}}$) and update functions ($\texttt{update}$) for the penalty and reward variables is shown in Figure 1.6. We need to synthesize the guards such that the following cost metric is minimized.

$$\lim_{t \to \infty} \frac{10 \times \texttt{discomfort}(t) + \texttt{fuel}(t) + \texttt{swTear}(t)}{\texttt{time}(t)}$$

Since the reward variable is the time spent, minimizing this metric means minimizing the *average* discomfort, fuel cost and wear-tear of the heater. We give a higher weight (10) to discomfort than fuel cost and wear-tear. The behavior of room-temperature in the synthesized optimal system from the initial room temperature of 20 is shown in Figure 1.7.

Human designers would require running a huge number of simulations followed by manual reasoning from candidate switching conditions and their observed performance to synthesize a switching logic which is optimal with respect to the given performance requirement. In Chapter 7, we discuss how our automated synthesis approach based on SCIDUCTION can be used to automatically infer a switching logic that is optimal with respect to the provided performance metric.

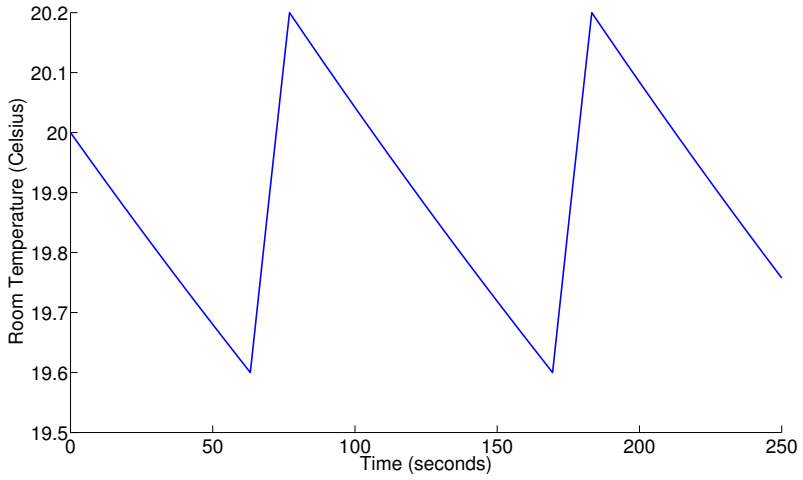


Figure 1.7: Behavior of Thermostat Controller Synthesized for Performance Specifications

1.2.2 Common Features and Challenges

The examples presented in the previous section illustrate that a number of design problems across different domains can benefit from automation. The common features and challenges for all these examples are as follows:

1. *Non-intuitive Design*: Design problems such as those presented in this section are not well-structured for human reasoning. Manual insight and intuition is of very limited use for these design problems. Designers and developers have to resort to trial and error. This process is both time-consuming as well as error-prone. This makes automated synthesis techniques pertinent for such applications.

2. *Intricate but Small Designs*: The synthesis problems are intricate but are relatively small. While complete automated synthesis of operating systems and compilers appear to be infeasible, problems such as those presented in this section illustrate that automation is possible for a number of synthesis problems, that are relatively small but difficult to solve manually. Further, the fact that these small synthesis problems are often components of large complex engineered system underscores the importance of correctly and optimally solving these small but challenging synthesis problems.

3. *Difficult to Verify*: Verification of a candidate system against the user-provided specification is either computationally expensive, making it practically infeasible such as equivalence checking of obfuscated code with simplified version, or even theoretically undecidable such as verification of multi-modal dynamical systems.

4. *Easy Query on Particular System Behavior*: It is easy to query about some specific behavior of the target system. For example, it is easy to obtain output of the target bit-vector code for a given input bitvector and it is easy to check whether a particular system trajectory of a multi-mode dynamical system with a given switching logic satisfies the safety property or not.

1.3 Thesis Contribution

The central contribution of this thesis is a new synthesis paradigm combing induction, deduction, and structure hypotheses, along with four demonstrations. We call this paradigm SCIDUCTION [112] based synthesis. SCIDUCTION can be used for synthesis as well as verification [112]. It builds on successful deductive techniques, such as satisfiability solving [13, 37], verification techniques [31] and numerical optimization [74], with inductive techniques such as algorithmic concept learning [6, 94]. Different instantiations of this paradigm by combining different inductive and deductive reasoning techniques were used to solve synthesis problems in different application domains. One of our main focus was on selecting applications where automated synthesis would be of most help to designers and developers.

SCIDUCTION can be seen as a *lens* through which we view the key ideas amongst the synthesis techniques presented in this thesis. SCIDUCTION uses structure hypothesis to integrate induction and deduction. *Induction* is the process of inferring a general principle from observed instances. Machine learning algorithms are typically inductive and learn a concept of classifier through generalization of examples [94, 7]. *Deduction*, on the other hand, involves the use of general rules and axioms to infer conclusions about particular problem instances [108, 16, 39]. Traditional formal verification and synthesis techniques, such as model checking or theorem proving, are deductive. It is natural since synthesis and verification are inherently deductive processes. On the other hand, inductive reasoning only ensures that the truth of its premises make it probable that its conclusion

is also true. The key is the use of structure hypothesis to combine inductive and deductive reasoning to obtain the kind of guarantees obtained in deductive synthesis and verification techniques. These structure hypotheses are mathematical assumptions used to define the class of artifacts to be synthesized.

Before describing the approach, it would be useful to reflect on the combined use of induction and deduction. Looking back at manual design and development, it is easy to notice that we often employ a combination of inductive and deductive reasoning while performing synthesis. For example, while trying to synthesize a proof for a theorem, we start by working out some examples and then try to find a pattern in the properties satisfied by those examples. This step of generalizing to patterns is an inductive process. Very often, human intuition and experience implicitly provides some guiding hypotheses for discovering these patterns. These patterns yield lemma or background facts that then guide a deductive process of proving the statement of the theorem from known facts and previously established theorems. Similarly, while synthesizing a new design or developing a new program manually, one often starts by enumerating sample behaviors that the design must satisfy and hypothesizing components that might be useful in the design process. The next step is to systematically combine these components using some design rules to obtain a candidate design or program. Thus, manual synthesis process usually iterates between inductive and deductive reasoning. We attempt to formalize this combination of induction and deduction using the notion of SCIDUCTION.

1.3.1 Elements of SCIDUCTION

Formally, a synthesis problem is a pair $\langle \mathcal{C}_S, \mathcal{C}_\phi \rangle$ where

- $\mathcal{C}_S$ is the class of systems from which we need to synthesize the system, and
- the synthesized system must satisfy a given specification from the class of specifications $\mathcal{C}_\phi$.

An instance of SCIDUCTION can be described using a triple $\langle \mathcal{H}, \mathcal{I}, \mathcal{D} \rangle$ where the three elements are as follows:

1. *Structure Hypothesis* $\mathcal{H}$: This encodes our hypothesis about the form of the design to be synthesized. Formally, $\mathcal{H}$ encodes a hypothesis that the system to be synthesized falls in a

subclass $\mathcal{C}_{\mathcal{H}}$ of $\mathcal{C}_S$, that is, $\mathcal{C}_{\mathcal{H}} \subseteq \mathcal{C}_S$. For example, consider the class of systems $\mathcal{C}_S$ to be the set of all finite automata over some set of variables and satisfying a specification ϕ. A structure hypothesis $\mathcal{H}$ could restrict the finite automata to be the synchronous composition of automata from a finite library $\mathcal{L}$. Each possible system in $\mathcal{C}_{\mathcal{H}}$ is some composition of automata from $\mathcal{L}$.

2. *Inductive Inference Engine* $\mathcal{I}$: This is an algorithm for learning from examples an artifact h defined by $\mathcal{H}$. The exact learning algorithm depends on the synthesis problem. It can be any online learning algorithm that can infer concepts from examples generated by one or more *oracles*. The oracles could be implemented using deductive procedures or a light-weight procedure such as execution of concrete model or evaluation of a black-box specification.

3. *Deductive Engine* $\mathcal{D}$: This is a lightweight decision procedure that applies deductive reasoning to answer queries generated in the synthesis process. The exact deduction engine depends on the synthesis problem. Deductive engine $\mathcal{D}$ is used to answer queries generated by inductive inference engine. Each query is typically formulated as a decision problem or an optimization problem to be solved by $\mathcal{D}$.

The combination of inductive and deductive reasoning and their connections have being long studied in artificial intelligence [109]. Inductive inference has also being previously formulated as deduction problem where inductive bias is provided as an additional input to the deductive engine. Inductive logic programming [99], an approach to machine learning, blends induction and deduction by performing inference in first-order theories using examples and background knowledge. Combination of inductive and deductive reasoning have also been explored for synthesizing plans in artificial intelligence; for example, the SSGP approach [35] generates plans by sampling examples, generalizing from those examples, and then proving the correctness of the generalization. The important distinction between the past work and our approach is the use of combined inductive and deductive approach for automated synthesis, and we accomplish this using structure hypothesis that the user can provide to the synthesis engine.

1.3.2 Soundness Guarantee of SCIDUCTION based Synthesis

A synthesis technique is sound if, given an arbitary synthesis instance $\langle \mathcal{C}_S, \phi \rangle$, if it outputs S, then $S \models \phi$. The SCIDUCTION based synthesis approach must be proved *sound* if the structure hypothesis $\mathcal{H}$ is true. Let $\phi \in \mathcal{C}_\phi$ be the specification and $\mathcal{C}_S$ be the class of systems from which we need to synthesize the system that satisfies ϕ using a SCIDUCTION based synthesis technique denoted by $\texttt{synth}_{\langle \mathcal{H},\mathcal{I},\mathcal{D} \rangle}$. The structure hypothesis $\mathcal{H}$ assumes that if a system is synthesizable in the class $\mathcal{C}_S$ for the given specification ϕ, the synthesized system falls in a subclass $\mathcal{C}_H \subseteq \mathcal{C}_S$. Formally, "the structure hypothesis is valid" $(s.h.i.v)$ is defined as

$$(s.h.i.v) \triangleq \ (\exists c\ c \in \mathcal{C}_S\ .\ c \models \phi) \Rightarrow (\exists c\ c \in \mathcal{C}_H\ .\ c \models \phi)$$

The soundness guarantee is as follows:

$$(s.h.i.v) \Rightarrow \texttt{synth}_{\langle \mathcal{H},\mathcal{I},\mathcal{D} \rangle}(\phi, \mathcal{C}_S) \models \phi$$

1.3.3 Applications of SCIDUCTION

SCIDUCTION provides a common framework to our synthesis techniques and makes it easy to identify and apply our techniques to new problems domains. The novel synthesis techniques presented in this thesis are as follows:

- *Synthesis of Loop-free Programs from Oracle Specification*: Given a specification as an input/output Oracle which produces desired output for a given input, we present a synthesis technique to automatically synthesize programs using components functions from a given library of primitive functions. We use this to automatically synthesize a number of bit-vector programs. We also use it to automatically deobfuscate snippets of code by using the obfuscated code as input/output oracle, and resynthesizing functionally equivalent but easy-to-understand code. The novelty of our approach lies in using black-box specification available only as an input/output oracle. Our approach does not rely on the availability of a complete functional specification, and is more widely applicable. This enables us to handle applications such as bit-vector programs and deobfuscation. We note that reasoning with a complete specification is often difficult, as in the case of obfuscated programs. In many

other cases, a complete specification is not readily available, as in the case of bit-vector programs.

- *Synthesis of Fixed-point Code from Floating-Point Code*: Given a floating-point code for a numerical computation routine, an accuracy specification, and the cost model of the hardware on which fixed-point code will be implemented, we present a synthesis technique to automatically discover the correct length of fixed-point variables in the corresponding fixed-point code. The synthesized fixed-point code meets the accuracy specification, and is of optimal cost with respect to the given cost model. The novelty of our approach lies identifying a small number of sufficient example executions of the floating-point code and using them to discover minimal cost fixed-point code with an accuracy above the specificed threshold.

- *Synthesis of Switching Logic for Safety*: Given a multi-mode dynamical system describing the continuous plant representing the physical world, and a safety specification on the variables modeling the behavior of the continuous plant, we propose an automated technique to synthesize the discrete controller that controls the switching between different modes of the system. Our synthesized controller ensures that the hybrid system, consisting of the discrete controllers and the continuous plant, remains safe. The novelty of our work lies in combining numerical simulation, fixed-point algorithm, and algorithmic learning for synthesis that enables us to consider systems with arbitrary continuous behavior as long as they can be simulated.

- *Synthesis of Switching Logic for Performance*: Given a mult-mode dynamical system describing the continuous plant and a cost function on the variables modeling a given performance objective, we propose an automated technique to synthesize the discrete controller for controlling the switching between different modes of the system such that the long-term cost is minimized. The novelty of our work lies in long-term performance objectives, and in combining numerical optimization and algorithmic learning to synthesize switching conditions between different modes of the multi-mode dynamical system.

1.4 Thesis Overview

The thesis is divided into two parts: the first part discusses techniques for automated synthesis of programs and the second part discusses techniques for automated synthesis of switching logic for cyber-physical systems. In Chapter 2, we discuss some preliminary material and notation along with previous work on automated program synthesis. We present the technique for synthesizing loop-free programs in Chapter 3. This is based on joint work with Sumit Gulwani, Sanjit A. Seshia and Ashish Tiwari [58]. We illustrate how our technique can be used to synthesize bitvector programs as well as perform program deobfuscation. We also contrast our work with related work and highlight the novel contributions. In Chapter 4, we present the technique for synthesizing fixed-point code from floating-point code, and demonstrate it over a set of case-studies. This is based on joint work with Sanjit A. Seshia [60]. In Chapter 5, we discuss some preliminary material and notations on automated synthesis of switching logic. We also discuss related work on automated control synthesis for multi-mode dynamical systems (hybrid systems). We present an automated synthesis technique to synthesize switching logic for hybrid systems with respect to safety specifications in Chapter 6. We also present experimental results to demonstrate its effectiveness. This is based on collaborations with Sumit Gulwani, Sanjit A. Seshia and Ashish Tiwari [59]. In Chapter 7, we present an automated approach to synthesize switching logic for hybrid systems with quantitative performance objectives. This is based on joint work with Sanjit A. Seshia and Ashish Tiwari [61]. We highlight the novelty of our work, and present case studies to demonstrate how it can be used in practice. We summarize the contributions made by the thesis in Chapter 8, and conclude the thesis by identifying future research directions.

This research was supported by NSF grants CNS-0644436 and CNS-0627734, the FCRP/-MARCO Multi-Scale Systems Center (MuSyC), Microsoft Research, Intel, and the Berkeley Fellowship for Graduate Studies from UC Berkeley.

Part I

Synthesis of Programs

Chapter 2

Background

In this chapter, we present relevant background on programs that will be helpful in explaining our technique for automated synthesis of loop-free programs from component functions in Chapter 3 as well as fixed-point numerical computation code from its floating-point version in Chapter 4.

2.1 Formalism and Notations

In this section, we formally present the syntax and semantics of programs. We begin by introducing a simple programming language *IMP* which manipulates integers using some basic arithmetic operations. The syntactic set associated with *IMP* are as follows.

- numbers $\mathbf{N}$, consisting of positive and negative integers with zero,
- truth values $\mathbf{T} = \{false, true\}$,
- variables $\mathbf{Var}$,
- arithmetic expressions $\mathbf{Aexp}$,
- boolean expressions $\mathbf{Bexp}$,
- commands $\mathbf{Com}$.

We assume that the syntactic structure of numbers and locations is given. For instance, numbers $\mathbf{N}$ might be the set of signed decimal numerals for positive and negative whole numbers, and the variables $\mathbf{Var}$ might consist of non-empty strings of letters or letters followed by digits. The syntactic structure of the arithmetic expressions, boolean expressions and commands is given using Backus-Naur form (BNF). The convention in using metavariables over the syntactic categories is as follows:

- n, m range over numbers $\mathbf{N}$,
- X, Y range over variables $\mathbf{Var}$,
- a range over arithmetic expressions $\mathbf{Aexp}$,
- b range over boolean expressions $\mathbf{Bexp}$,
- c range over commands $\mathbf{Com}$.

These variables can be sub-scripted or primed to make reading the rules of syntactic structure more convenient.

The arithmetic expressions $\mathbf{Aexp}$ are formed by the following expression rules.

$$a ::= n \mid X \mid a_o \circ a_1$$
$$\circ ::= + \mid - \mid \times \mid$$

The symbol "::=" can be read as "can be" and the symbol "|" as "or". Thus, an arithmetic expression in *IMP* can be a number n or a variable X or $a_0 + a_1$ or $a_0 - a_1$ or $a_0 \times a_1$, built from arithmetic expressions a_0 and a_1.

Similarly, the boolean expressions $\mathbf{Bexp}$ are formed by the following expression rules.

$$b ::= true \mid false \mid a_o = a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \wedge b_1 \mid b_0 \vee b_1$$

The commands $\mathbf{Com}$ are formed by the following expression rules.

$$b ::= \mathbf{skip} \mid X := a \mid \mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1 \mid \mathbf{while}\ b\ \ c$$

Evaluation of numbers:	$\langle n, \sigma \rangle \rightarrow n$
Evaluation of variables:	$\langle X, \sigma \rangle \rightarrow X$
Evaluation of sums:	$\dfrac{\langle a_0, \sigma \rangle \rightarrow n_0 \qquad \langle a_1, \sigma \rangle \rightarrow n_1}{\langle a_0 + a_1, \sigma \rangle \rightarrow n}$ where n is the sum of n_0 and n_1
Evaluation of subtractions:	$\dfrac{\langle a_0, \sigma \rangle \rightarrow n_0 \qquad \langle a_1, \sigma \rangle \rightarrow n_1}{\langle a_0 + a_1, \sigma \rangle \rightarrow n}$ where n is the result of subtraction n_1 from n_0
Evaluation of products:	$\dfrac{\langle a_0, \sigma \rangle \rightarrow n_0 \qquad \langle a_1, \sigma \rangle \rightarrow n_1}{\langle a_0 \times a_1, \sigma \rangle \rightarrow n}$ where n is the product of n_0 and n_1
Evaluation of division:	$\dfrac{\langle a_0, \sigma \rangle \rightarrow n_0 \qquad \langle a_1, \sigma \rangle \rightarrow n_1}{\langle a_0 / a_1, \sigma \rangle \rightarrow n}$ where n is the quotient on dividing n_0 by n_1

Table 2.1: Inductive Evaluation Rules for $\mathbf{Aexp}$

We now describe the semantics of the language *IMP*. Underlying a programming language is an idea of state determined by the contents of the variables $\mathbf{Var}$. With respect to a state, an arithmetic expression in *IMP* evaluates to an integer and a boolean expression evaluates to a truth value. Commands change the state using the evaluated values of the arithmetic and boolean expressions in the commands. Thus, formally defining semantics of a language involves definition of *state* and then, the *evaluation* of the arithmetic and boolean expressions, and finally the *execution* of the commands.

A *state* is a function $\sigma : \mathbf{Var} \rightarrow \mathbf{N}$ from variables to numbers. The set of *states* is denoted by Σ. Thus, $\sigma(X)$ is the value of the variable X in state σ. The evaluation of an expression a in a state σ is denoted by $\langle a, \sigma \rangle$. The inductive syntax-directed evaluation rules specify how an arithmetic expression is evaluated in a state is presented in Table 2.1.

The rules can be read as consisting of a premise above a solid line and the conclusion below the line. When there is no premise, the solid line is omitted and only the conclusion is given. For example, the rule for sum can be read as: if $\langle a_0, \sigma \rangle$ evaluates to n_0 and $\langle a_1, \sigma \rangle$ evaluates to n_1, then $\langle a_0 + a_1, \sigma \rangle$ evaluates to n where n is the sum of n_0 and n_1.

Similarly, the evaluation of boolean expressions is specified using the set of tables in Table 2.2.

true:	$\langle true, \sigma \rangle \to true$
false:	$\langle false, \sigma \rangle \to false$
Evaluation of equality:	$\dfrac{\langle a_0, \sigma \rangle \to n \qquad \langle a_1, \sigma \rangle \to m}{\langle a_0 = a_1, \sigma \rangle \to true}$ if n and m are equal $\dfrac{\langle a_0, \sigma \rangle \to n \qquad \langle a_1, \sigma \rangle \to m}{\langle a_0 = a_1, \sigma \rangle \to false}$ if n and m are unequal
Evaluation of inequality $\leq$:	$\dfrac{\langle a_0, \sigma \rangle \to n \qquad \langle a_1, \sigma \rangle \to m}{\langle a_0 \leq a_1, \sigma \rangle \to true}$ if n is less than or equal to m $\dfrac{\langle a_0, \sigma \rangle \to n \qquad \langle a_1, \sigma \rangle \to m}{\langle a_0 \leq a_1, \sigma \rangle \to false}$ if n is not less than or equal to m
Evaluation of negation:	$\dfrac{\langle b, \sigma \rangle \to true}{\langle \neg b, \sigma \rangle \to false} \quad \dfrac{\langle b, \sigma \rangle \to false}{\langle \neg b, \sigma \rangle \to true}$
Evaluation of conjunction:	$\dfrac{\langle b_0, \sigma \rangle \to t_0 \qquad \langle b_1, \sigma \rangle \to t_1}{\langle b_0 \wedge b_1, \sigma \rangle \to t}$ where t is $true$ if t_0 is $true$ and t_1 is $true$, and is $false$ otherwise.
Evaluation of disjunction:	$\dfrac{\langle b_0, \sigma \rangle \to t_0 \qquad \langle \neg b_1, \sigma \rangle \to t_1}{\langle b_0 \vee b_1, \sigma \rangle \to t}$ where t is $true$ if t_0 is $true$ or t_1 is $true$, and is $false$ otherwise

Table 2.2: Inductive Evaluation Rules for $\mathbf{Bexp}$

Atomic Commands	$\langle \textbf{skip}, \sigma \rangle \to \sigma$ $\dfrac{\langle a, \sigma \rangle \to m}{\langle X := a, \sigma \rangle \to \sigma[m/X]}$
Sequencing	$\dfrac{\langle c_0, \sigma \rangle \to \sigma'' \qquad \langle c_1, \sigma'' \rangle \to \sigma'}{\langle c_0; c_1, \sigma \rangle \to \sigma'}$
Conditionals	$\dfrac{\langle b, \sigma \rangle \to true \qquad \langle c_0, \sigma \rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1 \ , \sigma \rangle \to \sigma'}$ $\dfrac{\langle b, \sigma \rangle \to false \qquad \langle c_1, \sigma \rangle \to \sigma'}{\langle \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1 \ , \sigma \rangle \to \sigma'}$
While-loop	$\dfrac{\langle b, \sigma \rangle \to false}{\langle \textbf{while } b \ \ c, \sigma \rangle \to \sigma}$ $\dfrac{\langle b, \sigma \rangle \to true \qquad \langle c, \sigma \rangle \to \sigma'' \qquad \langle \textbf{while } b \ \ c \rangle, \sigma'' \to \sigma}{\langle \textbf{while } b \ \ c, \sigma \rangle \to \sigma}$

Table 2.3: Inductive Execution Rules for commands $\mathbf{Com}$

The rules for execution of commands is given in Table 2.3. Similar to evaluation of arithmetic and boolean expressions, $\langle c, \sigma \rangle \to \sigma'$ denotes that executing command c from state σ yields a new program state σ'. For any number $m \in \mathbf{N}$ and $X \in \mathbf{Var}$, we write $\sigma[m/X]$ for the state obtained from σ by replacing the contents in the variable X by m, that is,

$$\sigma[m/X](Y) = \begin{cases} m & \text{if } Y = X \\ \sigma(Y) & \text{if } Y \neq X \end{cases}$$

This completes the formal description of the simple *IMP* language. When the **while** command is not used in a program, it is said to be a *loop-free* program. This simple *IMP* language needs to be augmented in order to describe bit-vector programs, floating-point programs and fixed-point programs. In the rest of this section, we discuss these augmentations.

2.1.1 Bitvector Programs

In digital computers, data in a variable are often stored in computer memory as a bitvector, that is, a sequence of bits of some finite length. Sometimes, programs directly manipulate bitvectors instead of treating them abstractly as integers. Motivation for this was presented earlier in Section 1.2.1. Such programs that manipulate bitvectors are called bitvector programs. In order

to support bitvector programs, we need to extend *IMP* language with bitvector datatypes in addition to integers and also have bitvector operations in addition to integer arithmetic operations. The program state σ is now a mapping from variables to numbers as well as bitvector variables to bitvectors. We add bitvector constants $\mathbf{bv}$, bitvector variables $\mathbf{BV}$ and bitvector expressions $\mathbf{BVexp}$ to the syntax of *IMP*. bv ranges over the bitvector constants, BV ranges over the bitvector variables and p ranges over bitvector expressions. We also their use subscripted and primed versions for clarity of the rules. The bitvector expressions $\mathbf{BVexp}$ are formed by the following expression rules. We overload the arithmetic operations since their meaning can be inferred from the used context.

$$
\begin{aligned}
p &::= bv \mid BV \mid p_o \circ p_1 \mid \star\ p_0 \mid \mathbf{concat}\ p_0\ p_1 \mid \mathbf{extract}\ n\ m\ p_0 \\
\circ &::= \&\& \mid || \mid \oplus \mid + \mid - \mid * \mid / \mid >> \mid << \\
\star &::= - \mid \neg
\end{aligned}
$$

The boolean expressions for *IMP* are expanded to include the comparison between bitvectors.

$$b ::= true \mid false \mid a_o = a_1 \mid a_0 \leq a_1 \mid \neg b \mid b_0 \wedge b_1 \mid b_0 \vee b_1 \mid p_0 = p_1 \mid p_0 \leq p_1$$

We now give the semantics of bitvector operations. A bitvector of length m is a function whose domain is the initial segment of the naturals $[0 \ldots m)$ and the co-domain is $\{0, 1\}$. The i-the bit of a bitvector bv is denoted by $bv[i]$ and the length of the bitvector is denoted by $L(bv)$. In order to define the semantics, we need the following additional helper functions.

- $\mathbf{bv2nat}$: This takes a bitvector bv of length m and returns an integer in the range $[0 \ldots 2^m)$ and is defined as follows:

$$\mathbf{bv2nat}(bv) = bv(m-1) \times 2^{m-1} + bv(m-2) \times 2^{m-2} + \ldots bv(0) \times 2^0$$

- $\mathbf{nat2bv[m]}$: This takes a non-negative integer n and returns the unique bitvector $bv : [0 \ldots m) \rightarrow \{0, 1\}$ of length m such that

$$bv(m-1) \times 2^{m-1} + ... + bv(0) \times 2^0 = n \mod 2^m$$

The semantic evaluation rules for bitvector expressions $\mathbf{BVexp}$ in a program state σ is given by inductive rules in Table 2.4.

Evaluation of bitvector constants:	$\langle bv, \sigma \rangle \rightarrow bv$
Evaluation of bitvector variables:	$\langle BV, \sigma \rangle \rightarrow BV$
Evaluation of concat:	$\dfrac{\langle p_0, \sigma \rangle \rightarrow bv_0 \qquad \langle p_1, \sigma \rangle \rightarrow bv_1}{\langle \mathbf{concat}\ p_0\ p_1, \sigma \rangle \rightarrow bv}$ where $bv[i]$ is $bv_0[i]$ for $0 \leq i < L(bv_0)$ and $bv_1[i - L(bv_0)]$ for $L(bv_0) \leq i < L(bv_0) + L(bv_1)$
Evaluation of extract:	$\dfrac{\langle p_0, \sigma \rangle \rightarrow bv_0}{\langle \mathbf{extract}\ n_0\ n_1\ p_0, \sigma \rangle \rightarrow bv}$ where $bv[i]$ is $bv_0[i + n_0]$ for $0 \leq i < n_1 - n_0$
Evaluation of bvnot:	$\dfrac{\langle p_0, \sigma \rangle \rightarrow bv_0}{\langle \neg p_0, \sigma \rangle \rightarrow bv}$ where $bv[i]$ is $\neg bv_0[i]$ for $0 \leq i < L(bv_0)$
Evaluation of bvand:	$\dfrac{\langle p_0, \sigma \rangle \rightarrow bv_0 \qquad \langle p_1, \sigma \rangle \rightarrow bv_1}{\langle (p_0 \&\& p_1), \sigma \rangle \rightarrow bv}$ where $bv[i]$ is 0 if $bv_0[i]$ is 0 else $bv_1[i]$
Evaluation of bvor:	$\dfrac{\langle p_0, \sigma \rangle \rightarrow bv_0 \qquad \langle p_1, \sigma \rangle \rightarrow bv_1}{\langle (p_0 \text{——} p_1), \sigma \rangle \rightarrow bv}$ where $bv[i]$ is 1 if $bv_0[i]$ is 1 else $bv_1[i]$
Evaluation of bvxor:	$\dfrac{\langle p_0, \sigma \rangle \rightarrow bv_0 \qquad \langle p_1, \sigma \rangle \rightarrow bv_1}{\langle p_0 \oplus p_1, \sigma \rangle \rightarrow bv}$ where $bv[i]$ is 1 if $bv_0[i] \neq bv_1[i]$, 0 otherwise
Evaluation of bvneg:	$\dfrac{\langle p_0, \sigma \rangle \rightarrow bv_0}{\langle -p_0, \sigma \rangle \rightarrow bv}$ where bv is $\mathbf{nat2bv}[L(bv_0)](2^m - \mathbf{bv2nat}(bv_0)$
Evaluation of bvadd:	$\dfrac{\langle p_0, \sigma \rangle \rightarrow bv_0 \qquad \langle p_1, \sigma \rangle \rightarrow bv_1}{\langle p_0 + p_1, \sigma \rangle \rightarrow bv}$ where bv_0 and bv_1 are of length m and $bv = \mathbf{nat2bv}[m](\mathbf{bv2nat}(bv_0) + \mathbf{bv2nat}(bv_1))$

Evaluation of bvsub:	$\dfrac{\langle p_0, \sigma\rangle \rightarrow bv_0 \qquad \langle p_1, \sigma\rangle \rightarrow bv_1}{\langle p_0 - p_1, \sigma\rangle \rightarrow bv}$ where bv_0 and bv_1 are of length m and $bv = bv_0 + (-bv_1)$
Evaluation of bvmul:	$\dfrac{\langle p_0, \sigma\rangle \rightarrow bv_0 \qquad \langle p_1, \sigma\rangle \rightarrow bv_1}{\langle p_0 \times p_1, \sigma\rangle \rightarrow bv}$ where bv_0 and bv_1 are of length m and $bv = \mathbf{nat2bv}[m](\mathbf{bv2nat}(bv_0) \times \mathbf{bv2nat}(bv_1))$
Evaluation of bvdiv:	$\dfrac{\langle p_0, \sigma\rangle \rightarrow bv_0 \qquad \langle p_1, \sigma\rangle \rightarrow bv_1}{\langle p_0 / p_1, \sigma\rangle \rightarrow bv}$ where bv_0 and bv_1 are of length m and $bv = \mathbf{nat2bv}[m](\mathbf{bv2nat}(bv_0) / \mathbf{bv2nat}(bv_1))$
Evaluation of shift-left:	$\dfrac{\langle p_0, \sigma\rangle \rightarrow bv_0}{\langle p_0 >> n, \sigma\rangle \rightarrow bv}$ where $bv = \mathbf{nat2bv}[L(bv_0)](\mathbf{bv2nat}(bv_0) \times 2^n$
Evaluation of shift-right:	$\dfrac{\langle p_0, \sigma\rangle \rightarrow bv_0}{\langle p_0 << n, \sigma\rangle \rightarrow bv}$ where $bv = \mathbf{nat2bv}[L(bv_0)](\mathbf{bv2nat}(bv_0)/2^n$

Table 2.4: Inductive Evaluation Rules for $\mathbf{BVexp}$

The evaluation of extended boolean expressions is done by adding more rules given in Table 2.5.

In addition to the discussed operators, we can also use $bv_0 < bv_1$ as syntactic sugar when $bv_0 \leq bv_1$ but not $bv_0 = bv_1$. Similarly, $bv_0 > bv_1$ is syntactic sugar for "$bv_0 \leq bv_1$ is not true", and $bv_0 \geq bv_1$ is syntactic sugar for "$bv_0 < bv_1$ is not true". This completes the formal specification of the bitvector programs.

Bitvector programs are more difficult to write because they allow bit-wise logical operations along with arithmetic operations. These programs can "sometimes stall programmers for hours or

Evaluation of bitvector equality:	$\dfrac{\langle BV_0, \sigma\rangle \rightarrow bv_0 \qquad \langle BV_1, \sigma\rangle \rightarrow bv_1}{\langle bv_0 = bv_1, \sigma\rangle \rightarrow true \text{ if } \mathbf{bv2nat}(bv_0) = \mathbf{bv2nat}(bv_1)}$ $\dfrac{\langle BV_0, \sigma\rangle \rightarrow bv_0 \qquad \langle BV_1, \sigma\rangle \rightarrow bv_1}{\langle bv_0 = bv_1, \sigma\rangle \rightarrow false \text{ if } \mathbf{bv2nat}(bv_0) \neq \mathbf{bv2nat}(bv_1)}$
Evaluation of bitvector inequality $\leq$:	$\dfrac{\langle BV_0, \sigma\rangle \rightarrow bv_0 \qquad \langle BV_1, \sigma\rangle \rightarrow bv_1}{\langle bv_0 \leq bv_1, \sigma\rangle \rightarrow true \text{ if } \mathbf{bv2nat}(bv_0) \leq \mathbf{bv2nat}(bv_1)}$ $\dfrac{\langle BV_0, \sigma\rangle \rightarrow bv_0 \qquad \langle BV_1, \sigma\rangle \rightarrow bv_1}{\langle bv_0 \leq bv_1, \sigma\rangle \rightarrow false \text{ if } \mathbf{bv2nat}(bv_0) \not\leq \mathbf{bv2nat}(bv_1)}$

Table 2.5: Additional Bitvector Evaluation Rules for $\mathbf{Bexp}$

days if they really want to understand why things work" [132]. These programs "typically describe some plausible yet unusual operation on integers or bit-strings that can be programmed using either longish fixed sequence of machine instructions or a loop, but the same thing can be done much more cleverly using just four or three or two carefully chosen instruction whose interactions are not at all obvious until explained or fathomed" [132]. Efficient bitvector code-fragments are of great significance for people who write optimizing compilers or higher-performance code in applications such as graphics, hardware programming, encryption, networking and databases [104].

Apart from manual difficulty of synthesizing bitvector programs, there are two other characteristics of these programs which make them an interesting target for automated synthesis. These characteristics are based on a large compendium of programs presented in Hacker's Delight book [132] which is said to be the Bible of bitvector programs.

- These programs are usually small up to about 25 lines of code and do not have loops in them. Hence, these programs are tricky but small loop-free programs.
- The operations used in these programs belong to a finite library of bit-wise operations and arithmetic operations and thus, these programs are amenable to a component-based synthesis technique.

We discuss an automated approach to synthesize bitvector program in Chapter 3.

2.1.2 Floating-point and Fixed-point Programs

Floating-point [40] is a system for approximately representing real numbers which supports a wide range of values. It allows the use of a fixed number of significant digits which is scaled using an exponent to represent real numbers approximately. The floating-point system is so called because the radix point can *float* anywhere relative to the significant digits of the number. In contrast to fixed-point numbers, floating-point representation can be used to support a much wider range of values with the same number of digits. The most common floating-point representation used in computers is that defined by IEEE 754 Standard [2].

In order to support floating-point programs, the *IMP* programming language needs to be augmented with single-precision and double-precision floating types as defined in IEEE 754 Standard. The storage layout of the floating-point numbers consist of three basic components: the sign, the exponent, and the mantissa. The storage layout of the single-precision and double-precision floating point numbers is presented in Table 2.6

- The *sign bit* is 0 for a positive number and 1 for a negative number. Flipping the value of this bit flips the sign of the number.

- The *mantissa*, also known as the significand, represents the precision bits of the number. It is composed of an implicit leading bit and the fraction bits. In order to maximize the quantity of representable numbers, floating-point numbers are typically stored with the radix point after the first non-zero digit. In base 2, the only possible non-zero digit is 1. Thus, we can just assume a leading digit of 1, and don't need to represent it explicitly. As a result, the mantissa has effectively 24 bits of resolution, by way of 23 fraction bits in single-precision floating-point numbers, and 53 bits of resolution, by way of 52 fractional bits in double-precision.

- The *exponent* field needs to represent both positive and negative exponents. To do this, a bias is added to the actual exponent in order to get the stored exponent. For IEEE single-precision floats, this value is 127. Thus, an exponent of 0 means that 127 is stored in the exponent field. A stored value of 200 indicates an exponent of $(200-127)$, or 73. Exponents of -127 (all 0s) and $+128$ (all 1s) are reserved for special numbers. For double precision, the exponent field is 11 bits, and has a bias of 1023.

	Sign	Exponent	Fraction	Bias
Single Precision	1 [31]	8 [30 − 23]	23 [22 − 00]	127
Double Precision	1 [63]	11 [62 − 52]	52 [51 − 00]	1023

Table 2.6: Floating-point Number Layout

Precision	Denormalized	Normalized	Approximate Decimal
Single	$\pm\ 2^{-149}$ to $(1 - 2^{-23}) \times 2^{-126}$	$\pm\ 2^{-126}$ to $(2 - 2^{-23}) \times 2^{127}$	$\pm\ 10^{-44.85}$ to $10^{38.53}$
Double	$\pm\ 2^{-1074}$ to $(1 - 2^{-52}) \times 2^{-1022}$	$\pm\ 2^{-1022}$ to $(2 - 2^{-52}) \times 2^{1023}$	$\pm\ 10^{-323.3}$ to $10^{308.3}$

Table 2.7: Range of floating-point numbers

Floating-point solves a number of representation problems. Fixed-point has a fixed window of representation, which limits it from representing very large or very small numbers. Floating-point, on the other hand, employs a sort of "sliding window" of precision appropriate to the scale of the number. The range of positive floating-point numbers can be split into normalized numbers (which preserve the full precision of the mantissa), and denormalized numbers. The denormalized numbers do not have an implicit leading bit of 1 and allow representation of really small numbers but with only a portion of the fraction's precision. The exponent of all 0s (-127) and all 1s (128) are reserved for denormalized numbers and representing infinity respectively. A complete discussion on the semantics of floating-point operations can be found in the IEEE 754 Standard [2].

A floating-point unit (FPU) is used to carry out operations on floating-point numbers such as addition, subtraction, multiplication, division and square root. Some floating-point operations such as exponential and trigonometric calculation could be emulated in a software library routine or supported explicitly by FPUs. The latter is more common with modern FPUs. FPUs are integrated with CPUs in computers but most embedded processors do not have hardware support for floating-point operations. Emulation of floating-point operations without hardware support can be very slow. The high cost of FPUs make it prohibitively expensive to include them in many low cost embedded systems. Inclusion of FPUs also increases the power consumption of the processors. This has made the use of fixed-point arithmetic very common in embedded systems.

Fixed-point [134] is a system for representing real numbers in which there are fixed number of digits and the radix point is also fixed. Fixed-point data consists of a sign mode bit, an integer part and a fractional part. We denote the fixed-point type of a variable x by $\mathbf{fx}\tau(x)$. A fixed-point

Number systems with $\texttt{WL} = 32$	Range	Precision
Single-precision Floating-point	$\pm\ 10^{-44.85}$ to $10^{38.53}$	*Adaptive*
Fixed-point type: $\langle 1, 8, 24\rangle$	$-10^{2.11}$ to $10^{2.11}$	$10^{-7.22}$
Fixed-point type: $\langle 1, 16, 16\rangle$	$-10^{4.52}$ to $10^{4.52}$	$10^{-4.82}$
Fixed-point type: $\langle 1, 24, 8\rangle$	$-10^{6.93}$ to $10^{6.93}$	$10^{-2.41}$

Table 2.8: Range of 32 bit fixed-point and floating-point numbers

type is a 3-tuple.

$$\langle \texttt{Signedness}, \texttt{IWL}, \texttt{FWL}\rangle$$

The sign mode bit Signedness is 0 if the data is unsigned and is 1 if the data is signed. The number of bits used for representing integers is called the integer wordlength (IWL) and the number of bits used for representing fraction is called the fractional wordlength (FWL). The fixed-point wordlength (WL) is the sum of the integer wordlength and fractional wordlength, that is, $\texttt{WL} = \texttt{IWL} + \texttt{FWL}$. We denote the wordlength of a variable x by $\texttt{WL}(x)$, its integer wordlength by $\texttt{IWL}(x)$, and its fractional wordlength by $\texttt{FWL}(x)$. A fixed-point number with fractional word length (FWL) is scaled by a factor of $\frac{1}{2^{\texttt{FWL}}}$. For example, a fixed point number 01110 with 0 as sign mode bit, integer wordlength of 2 and fractional wordlength of 2 represents $14 \times \frac{1}{2^2}$, that is, 3.5. Converting a fixed-point number with scaling factor R to another type with scaling factor S, requires multiplying the underlying integer by R and dividing by S; that is, multiplying by the ratio R/S. For example, converting 01110 with 0 as sign-bit, integer wordlength of 2 and fractional wordlength of 2 into a fixed-point number with 0 as sign-bit, integer wordlength of 2 and fractional wordlength of 3 requires multiplying with $\frac{2^3}{2^2}$ to obtain 011100. If the scaling factor is to be reduced, the new integer will have to be rounded. For example, converting the same fixed-point number 01110 to a fixed-point number with fractional wordlength of 0 and integer wordlength of 2 yields 011, that is, 3 which is obtained by rounding down from 3.5. The range of the fixed-point number is much smaller compared to the range of floating-point numbers for the same number of bits since the radix point is fixed and no dynamic adjustment of precision is possible. For example, let us consider fixed-point numbers with 32 bits. Table 2.8 shows the comparison of range of fixed-point numbers with different integer wordlengths and fractional wordlengths, and the range of the normalized single-precision floating-point numbers.

Thus, translating a floating-point code into fixed-point code is non-trivial and requires careful consideration of loss of precision and range. The integer wordlengths and fractional wordlengths

of the fixed-point variables need to be carefully selected to ensure computation does not loose accuracy beyond the specified threshold. In Chapter 4, we present an automated synthesis technique to derive fixed-point code from floating-point code. Fixed-point arithmetic is performed on custom hardware and FPGA is often a natural choice for implementing these algorithms. The operations supported by fixed-point arithmetic are the same as floating-point arithmetic standard [2] but the semantics might be different on custom hardware. For example, the rounding mode for arithmetic operations could be different, and the result could be specified to saturate or overflow/underflow in case the wordlength of a variable is not sufficient to store a computed result. A complete semantics of fixed-point operation is provided with the Fixed-point Toolbox in Maltab [1].

2.2 Related Work

In this section, we describe some of the different approaches used for program synthesis. Both deductive and inductive techniques have been used for program synthesis. Automated program synthesis has also been studied with slightly different goals such as automated completion of partial programs and automated optimization of programs.

2.2.1 Deductive Program Synthesis

In deductive program synthesis [87, 122], a program is synthesized by constructively proving a theorem which states that for all inputs in a given set, there exists an output, such that a given functional specification predicate holds. Deductive program synthesis assumes that a full functional specification is given. Moreover, it requires advanced deduction technology that is hard to automate. In contrast, we require only an input/output oracle specification and use fully automated SMT solvers [13] for deduction.

2.2.2 Inductive Program Synthesis

In inductive program synthesis [123, 69], recursive programs are generated from input-output examples in two steps. In the first step, a set of I/O examples are written as one large conditional

expression. In the second step, this initial program is generalized into a recursive program by searching for syntactic regularities in the initial program. In contrast, we do not require a "good" set of I/O examples be given a-priori. Moreover, we do not explicitly generalize – generalization happens implicitly from synthesizing a function using only a given set of components. Shapiro's Algorithmic Debugging System [114] performs synthesis by repeatedly using the oracle to identify errors in the current (incorrect) program and then *fixing* those errors. We do not fix incorrect programs. We use the incorrect program to identify a distinguishing input and then re-synthesize a new program that works on the new input-output pair as well. In programming by demonstration [75, 77, 29], the user demonstrates how to perform a task and the system learns an appropriate representation of the task procedure. Another related work is the ADATE system [102] for automated functional programming. It uses specifications that contain few constraints on the programs to be synthesized and that allows a wide range of correct programs. Successive better programs are developed using incremental program transformations. A key to the success of ADATE is the exact design of these transformations and how to systematically search for appropriate transformation sequences. Unlike our method, these approaches do not make active oracle queries, but rely on the demonstrations or transformation rules which the user chooses. Making active queries is important for efficiency and terminating quickly (so that user is not overwhelmed with queries).

2.2.3 Synthesis from Functional Specification

Synthesis from functional specification has being widely studied in literature. Brahma[46] uses SMT solving technology to synthesize a straight-line sequence of instructions from functional description of the desired code sequence. `DIPACS` [62] uses an AI planner to implement a programmer-defined abstract algorithm using a sequence of library calls. The behavior of the library procedures and the abstract algorithm is specified using high-level *abstractions*, e.g., predicates *sorted* and *permutation*. It uses interaction with the programmer to prune undesirable compositions. Jungloid mining tool [86] synthesizes code-fragments (over a given set of API methods annotated with their type signatures) given a simple query that describes the desired code using input and output types.

2.2.4 Automated Synthesis for Program Completion

Another line of related work is on automated completion of partially written programs. The Sketch [117, 116] system takes as input a sketch, that is, a program with holes of missing constants, and synthesizes programs by correctly filling these holes such that the synthesized program is consistent with the user-given complete specification. In contrast, we consider the component-based synthesis problem in which we are provided with a finite set of component functions and we need to synthesize a correct program using these components. Unlike Sketch, we are not provided with a complete program specification and instead, we only have an input/output oracle specification. The approach used in Sketch for synthesis is also based on constraint solving. Sketch internally generates Boolean constraints, which are solved using Boolean satisfiability solvers. Our technique generates formulas in a richer logic, which are solved using Satisfiability Modulo Theory (SMT) solvers [13]. The SKETCH approach uses a counterexample-guided loop that constantly interacts with a verifier to check candidate implementations against a complete specification, where the verifier provides counterexamples until a correct solution has been found. In contrast, we do not use counterexamples for synthesis. Further, we require a validation oracle only when the specification cannot be realized using the provided components. This verifier is not required to return a counter-example. Sketch relies on the developer to come up with the algorithmic insight and uses the sketch compiler to fill in the missing details using counterexample guided inductive synthesis. In contrast, our tool seeks to discover algorithmic insights, albeit at the cost of being more suited for a special class of programs. We chose bitvector programs as our main application domain since coming up with algorithmic insight is the hard part here.

2.2.5 Program Optimization

Another line of related work relevant to program synthesis is that of program optimization. Super-optimization is the task of finding an optimal code sequence for a straight-line target sequence of instructions, and it is used in optimizing performance-critical inner loops. One approach to super-optimization has been to simply enumerate sequences of increasing length or cost, testing each for equality with the target specification [90]. Another approach has been to constrain the search space to a set of equality-preserving transformations expressed by the system designer [63] and then select the one with the lowest cost. Recent work has used super-optimization [11, 12] to

automatically generate general purpose peephole optimizers by optimizing a small set of instructions in the code. In these approaches, the exhaustive state space search is quite expensive making them amenable to only discovering optimal instructions of length four or less in reasonable amount of time.

2.2.6 Fixed-point Program Synthesis

Previous techniques for optimizing fixed-point types are based on statistical sampling of the input space. These methods sample a large number of inputs and heuristically solve an optimization problem such as Equation 4.1 that minimizes implementation cost while ensuring that some correctness specification is met over the sampled inputs. The techniques differ in in the heuristic search method employed, in the measure of cost, or in how accuracy of fixed-point implementation is determined. Sung and Kum [124] use a heuristic search technique which starts with the minimum wordlength implementation as the initial guess. The wordlengths are increased one by one till the error falls below an acceptable threshold. Han et al. [50, 51] use a gradient-based sequential search method which starts with the minimum wordlength implementation as the initial guess. The gradient (ratio of increase in accuracy and increase in wordlengths) is computed for a set of wordlength changes at each step and the search moves in the direction with maximum gradient. Shi et al. [115] propose a floating-point to fixed-point conversion methodology for digital VLSI signal processing systems. Their approach is based on a perturbation theory which shows that the change to the first order is a linear combination of all the first- and second-order statistics of the quantization noise sources. Their technique works with general specification critera, as long as these can be represented as large ensemble averages of functions of the signal outputs. For example, they use mean-squared error (MSE) as the specification function. The cost of the implementation is a quadratic function. Monte Carlo simulation of a large number of input examples is used to formulate a quadratic optimization problem based on perturbation theory. In contrast, our specification requires that the accuracy condition holds for all inputs and not just on an average. Further, the cost function can be any arbitrary function for our technique and need not be quadratic. Perhaps most importantly, our technique does not rely on apriori random sampling of a large number of input values, instead using optimization to discover a small set of *interesting* examples which suffice to discover optimal fixed-point implementation.

Purely analytical methods [121, 68] based on dataflow analysis have also been proposed for synthesizing fixed-point programs based on forward and backward propagation in the program's dataflow graph. The advantages of these techniques are that they do not rely on picking the right inputs for simulation, can handle arbitrary programs (with approximation), and can provide correctness guarantees. However, they tend to produce very conservative wordlength results.

Inductive synthesis based on satisfiability solving has been previously used for synthesizing programs from functional specifications. These approaches [118, 58, 47] rely on constraint solving in much the same way as we rely on optimization routines. However, these approaches only seek to find a correct program, without any notion of cost and optimization. In contrast, our technique is used to find a fixed-point program which is not only correct with respect to a condition on accuracy but is also of minimal cost.

2.2.7 Dimensions

In summary, the related work on program synthesis can be broadly categorized along the following dimensions. We also identify how the techniques proposed in this thesis are different from existing work on program synthesis with respect to each dimension.

1. The first dimension is the *specification* required for program synthesis.Deductive program techniques [87, 122] as well as the more recently developed SAT based techniques for program completion [117, 116, 46] require a complete specification which can be an unoptimal code or a logical formula. Inductive techniques, such as program synthesis through demonstration [123, 69, 75, 77, 29] or inference of API sequences from input-output examples [86], require a set of input/output pairs for the target program. The correctness of the synthesized program depends on whether this set of input/output pairs is representative of all possible input/outputs. In contrast, our method discovers a small but sufficient number of input/output pairs to identify a correct program.

2. The second dimension is the *user input* regarding the target program. The Sketch system [117, 116] requires user to provide the target program with missing constants as holes which can be filled by the synthesis technique. The synthesis techniques for discovering API sequence calls [86] are provided with a library of possible APIs and their type

signature. Deductive program synthesis techniques [87, 122] also require specification of operations and commands in the programming language and their semantics. Superoptimizers [90, 63, 11, 12] rely on user to provide a functionally correct program which can then be optimized to synthesize a more efficient program. In our approach for automated synthesis of bit-vector programs, we require a finite library of bit-vector operations to be used in the program. In order to synthesize fixed-point code from the floating-point code, we use the structure of the floating-point code and use the synthesis technique to discover the correct wordlengths of fixed-point variables.

3. The third dimension is the *approach* used to solve the synthesis problem. Deductive program synthesis techniques [87, 122] depend on manual or partially automated theorem proving techniques to infer the correct program. Inductive program synthesis techniques [123, 69, 75, 77, 29] use learning techniques to identify syntactic regularities and synthesize programs through generalization of these regularities. Shapiro's Algorithmic Debugging System [114] and Sketch [117, 116] uses incremental identification and fixing of errors in a program to synthesize the correct program. We use a combination of inductive and deductive techniques for program synthesis. The bit-vector programs are synthesized by using the SMT solvers [13] to discover a set of inputs which identify a semantically unique combination of component functions which is consistent with the input/output oracle. The queries to the oracle are made using the idea of distinguishing input from algorithmic learning which is discussed in detail in Chapter 3. In the automated synthesis of fixed-point code from floating-point code described in Chapter 4, we use numerical optimization to discover input/output pairs which are then generalized to find a low cost fixed-point implementation.

Chapter 3

Oracle Based Synthesis of Loop-free Program

In this chapter, we present an approach to automatic synthesis of loop-free programs using SCIDUCTION. Our approach is based on a combination of oracle-guided learning from examples, and constraint-based synthesis from components using satisfiability modulo theories (SMT) solvers. Our approach is suitable for many applications, including as an aid to program understanding tasks such as reverse engineering malware. We demonstrate the efficiency and effectiveness of our approach by synthesizing bit-manipulating programs and by deobfuscating programs. We begin by introducing the problem in Section 3.1 and presenting our solution in Section 3.2. We discuss the performance and correctness guarantees of our approach in Section 3.3 and present experimental results in Section 3.4.

3.1 Introduction

Automatic synthesis of programs has long been one of the holy grails of software engineering. It has found many practical applications: generating optimal code sequences [90, 63], optimizing performance-critical inner loops, generating general-purpose peephole optimizers [11, 12], automating repetitive programming tasks [75], and filling in low-level details after the higher-level intent has been expressed [117]. Two applications of synthesis are of particular interest in this

chapter. The first is that of automating the discovery of non-intuitive algorithms (e.g., [46]). The second application, as we show in this chapter, is *program understanding*, and more specifically, *program deobfuscation*. The need for deobfuscation techniques has arisen in recent years, especially due to an increase in the amount of malicious code that is often obfuscated [125]. Currently, human experts use decompilers and manually deobfuscate the resulting code (e.g., see [105, 132]). Clearly, this is a tedious task that could benefit from automated tool support.

A traditional view of program synthesis is that of synthesis from complete specifications. One approach is to give a specification as a formula in a suitable logic [87, 122, 62, 46]. Another is to write the specification as a simpler, but possibly far less efficient program [90, 63, 117]. While these approaches have the advantage of completeness of specification, such specifications are often unavailable, difficult to write, or expensive to check against using automated verification techniques. Writing a logical specification for bitvector programs is difficult for users. In the case of reverse engineering malware, complete specification is available in principle as the obfuscated malware but automatically verifying against this code is practically infeasible due to obfuscation techniques which hinder static analysis. In this chapter, we propose a novel *oracle-guided* approach to program synthesis, where an *I/O oracle* that maps a given program input to the desired output is used as an alternative to having a complete specification. The key idea of our algorithm is to query the I/O oracle on an input that can distinguish between non-equivalent programs that are consistent with the past interaction with the I/O oracle. The process is repeated until a semantically unique program is obtained. Our experimental results show that only few rounds of interaction are needed.

We apply the oracle-guided approach to automated synthesis of *loop-free programs*, those that compute functions of their input and terminate. Such programs arise in a variety of application contexts, such as low-level bit-manipulating code, scientific computing kernels, parts of control software in graphical languages such as LabVIEW, and even applications in high-level scripting languages such as Javascript and Ruby that are formed by chaining multiple high-level operators. A key characteristic of our method is that it is *component-based*, meaning that we synthesize a program by performing a circuit-style, loop-free composition of components drawn from a given component library. We can also address the challenge of identifying whether the given set of components is insufficient to synthesize the desired program. For this purpose, we additionally require making only one query to a more expensive *validation oracle* that checks whether the

program is correct or not.

Our synthesis algorithm is based on a novel *constraint-based* approach that reduces the synthesis problem to that of solving two kinds of constraints: the *I/O-behavioral constraint* whose solution yields a candidate program consistent with the interaction with the I/O oracle, and the *distinguishing constraint* whose solution provides the input that distinguishes between non-equivalent candidate programs. These constraints can be solved using off-the-shelf SMT (Satisfiability Modulo Theory) solvers. Traditional synthesis algorithms perform a expensive combinatorial search over the space of all possible programs. In contrast, our technique leaves the inherent exponential nature of the problem to the underlying SMT solver, whose engineering advances over the years allow them to effectively deal with problem instances that arise in practice, which are usually not hard, and hence end up not requiring exponential reasoning.

3.1.1 Contributions

- We propose a novel oracle-guided approach to synthesis, where an I/O oracle obviates the need for complete specifications.
- We present an efficient SMT encoding of the space of possible programs formed using functions from a finite library of component functions.
- We present an instantiation of the oracle-guided approach to synthesis of loop-free programs over a given set of components (see problem definition in Section 3.1.2). This is enabled by a novel constraint-based technique that involves an interaction between SMT solvers and the I/O oracle (Section 3.2).
- We demonstrate the utility of our synthesis technique to discovery of bit-manipulating programs [132], which are often needed for optimizing performance (Section 3.4.2). These programs are quite unintuitive and can be difficult for even expert programmers to discover.
- We propose a novel application of program synthesis to program understanding. We demonstrate this in the context of malware deobfuscation by deobfuscating examples drawn from and inspired by the Conficker and MyDoom viruses using our synthesis technique (Section 3.4.2).

In the rest of the section, we first present a formal problem definition in Section 3.1.2 and then illustrate it with a simple example in Section 3.1.3.

3.1.2 Problem Definition

The goal is to synthesize a loop-free program using a given set of base components and using input-ouput examples. We assume the presence of an I/O oracle that can be queried on any input. The I/O oracle, when given an input, returns the output of the desired program (that we wish to synthesize) on that input. We also assume the presence of a validation oracle that validates the correctness of a candidate program. Finally, we assume that we are given a set of (base) components that should be used as building blocks in the synthesized program. Each component is given in the form of its input-output specification, which is written as a logical formula relating the inputs and the outputs of that component. For ease of presentation, we assume that all components have exactly one output. We also assume that all inputs and outputs have the same *type*. These restrictions are easily removed.

Formally, the synthesis problem in our proposed programming methodology requires the following:

- A validation oracle $\mathcal{V}$ that, given any candidate program (constructed from base components), returns a Boolean answer indicating whether the candidate program is the desired one or not.
- An I/O oracle $\mathcal{I}$ that, given any program input, returns the output of the desired program on that input.
- A set of specifications $\{\langle \vec{I}_i, O_i, \phi_i(\vec{I}_i, O_i)\rangle \mid i = 1, \ldots, N\}$, called a library, where $(\vec{I}_i, O_i, \phi_i(\vec{I}_i, O_i)\rangle$ is the specification for the base component f_i, which includes
  - a tuple of input variables $\vec{I}_i$ and an output variable O_i
  - an expression $\phi_i(\vec{I}_i, O_i)$ over variables $\vec{I}_i$ and O_i that specifies the input-output relationship of the i-th component.

  All variables $\vec{I}_i, O_i$ are assumed distinct.

The goal of the synthesis problem is to synthesize a program P that can be validated by the validation oracle $\mathcal{V}$, i.e., $\mathcal{V}(P) = \texttt{true}$. Furthermore, program P should be constructed using only the set of base components in the library, i.e.,, Program P should take $\vec{I}$ as its inputs and use the set $\{O_1, \ldots, O_N\}$ as temporary variables in the following form:

$$\underline{P(\vec{I})\texttt{:}}$$
$$O_{\pi_1} := f_{\pi_1}(X_{\pi_1})\texttt{;} \quad \ldots \texttt{;} \quad O_{\pi_N} := f_{\pi_N}(X_{\pi_N})\texttt{;}$$
$$\texttt{return } O_{\pi_N}\texttt{;}$$

where

C1. each variable in X_{π_i} is either an input variable from $\vec{I}$, or a temporary variable O_{π_j} such that $j < i$, and

C2. $\pi_1, \ldots, \pi_N$ is a permutation of $1, \ldots, N$.

Program P above appears to be a straight-line program, but, in fact, it can be more complex because the base components f_i's can be complex. In particular, base components can be "if-then-else" functions, and using these components, Program P can describe arbitrary loop-free programs.

We note that the program P above is using *all* components from the library. We can assume this without any loss of generality. Even when there is a correct Program P using fewer components, that program can always be extended to a program that uses all components by adding dead code. Dead code can be easily statically identified and removed in a post-processing step.

We also note that program P above is using each base component only once. We can assume this without any loss of generality. If there is a Program P using *multiple* copies of the same base component, we assume that the user provides multiple copies explicitly in the library. Such a restriction of using each base component only once is interesting in two regards. First, it can be used to enforce efficient or minimal programs. Second, it prunes the search space of possible programs making the synthesis problem finite and tractable.

Informally, the synthesis problem is to come up with a program – using only the base components in the given library – that is accepted by the validation oracle.

3.1.3 Running Example

We present one example in this section to introduce the synthesis problem and motivate our approach.

Consider the following programming problem: Given a bit-vector integer x, of finite but arbitrary length, construct a new bit-vector y that corresponds to x with the rightmost string of contiguous 1s turned off, i.e., reset to 0s. Such programming problems often arise while developing low level embedded code, network applications or in other domains where bit-level manipulation is needed.

Let us contemplate writing a formal specification for this problem. The most natural and easiest specification involves the use of alternating quantifiers, where n is the length of x:

$$\begin{aligned}\exists i, j.\ \{\ 0 \leq i, j < n \wedge (\forall k.\ j \leq k \leq i \implies x[k] = 1) \\ \wedge\ (\forall k.\ 0 \leq k < j \implies x[k] = 0) \\ \wedge\ (x[i+1] = 0 \vee i = n-1) \\ \wedge\ (\forall k.\ i < k < n \implies x[k] = y[k]) \\ \wedge\ (\forall k.\ 0 \leq k \leq i \implies y[k] = 0)\ \}\end{aligned}$$

The above specification is not easy to write for a common programmer. Moreover, verifying any candidate implementation against the above specification is challenging using current tools due to the presence of quantifiers in the formula. For example, there is no sound and complete procedure for first-order logic formulas of linear arithmetic with uninterpreted functions [49].

Let us consider writing some sample input-output pairs, or examples, for the problem. For any input x, it is easy to provide the corresponding output y. Some example (x, y) pairs are $(0110, 0000), (0101, 0100), (110110, 110000)$.

Finally, let us contemplate writing a program for the above problem. A straightforward, but inefficient, implementation is a loop that iterates through the bits of x and zeroes out the rightmost contiguous string of 1s. Can we synthesize a shorter and more efficient implementation? It is difficult to answer this, but it is easy to speculate that the elementary operations that may be used inside such an efficient implementation will be the standard bit-vector operators: bit-wise logical operations ($|$, $\&$, $\oplus$, $\sim$), and basic arithmetic operations ($+, -, *, /, \%$).

Given a set of possible elementary operations, and an ability to generate outputs for given inputs, our oracle-guided synthesis tool BRAHMA will synthesize the following nontrivial and tricky procedure for solving the above problem.

```
turnOffRightMostOneBitString (x)
{ t1 = x-1;
  t2 = (x || t1);
  t3 = t2+1;
  t4 = (t3 && x);
  return t4; }
```

A programmer will require considerable familiarity with bit-level manipulations to come up with such an implementation. Hence, automated synthesis of such difficult-to-write programs would be helpful to programmers.

3.2 SCIDUCTIVE Approach

In this section, we provide our solution for the program synthesis problem formally described above. Our solution is based on encoding the space of all possible programs by a formula (Section 3.2.1). Given a set of input-output pairs, we then constrain this formula further so that it encodes only those programs that work correctly on the given input-output pairs (Section 3.2.1). By solving this constraint, we generate a candidate solution. If the candidate solution is not the desired program, we provide a way to generate a new input-output pair (Section 3.2.1). The overall procedure that combines these parts to solve the program synthesis problem is presented in Section 3.2.2. We present enhancements to our basic procedure in Section 3.2.4.

3.2.1 Encoding Programs

We present an encoding of the space of *well-formed candidate programs*, that is, of programs P satisfying constraints C1 and C2, as formulas. Note, however, that the material in subsequent sections depends only on the *existence* of such an encoding. Our proposed approach can be used with alternative encodings as well.

Intuitively, the encoding we use involves viewing the space of candidate programs as all ways of *connecting* components from the library that satisfy syntactic and semantic well-formedness constraints. Each connection is encoded using an integer-valued *location variable*. Put another

way, the value of a location variable determines which component goes on which location (line-number), and from which location (line-number or circuit input) it gets its input arguments.

The main property of the encoding that our approach relies upon is distilled into the following theorem. This theorem states the existence of two formulas (encodings): the first formula $\psi_{\texttt{wfp}}$ represents the set of all *syntactically* well-formed programs; whereas the second formula $\phi_{\texttt{func}}$ represents the set of all *semantic* input-output behaviors of a well-formed program.

Theorem 1. *There exists a set of integer-valued location variables L, a* well-formedness constraint $\psi_{\texttt{wfp}}(L)$ *over L, a mapping* `Lval2Prog`*, and a* functional constraint $\phi_{\texttt{func}}(L, \vec{I}, O)$ *over $L \cup \{\vec{I}, O\}$ such that the following properties hold:*

- `Lval2Prog` *is a bijective mapping from the set of values L that satisfy the constraint $\psi_{\texttt{wfp}}(L)$ to the set of programs that satisfy constraints C1 and C2.*
- *Let L_0 be a satisfying assignment to the formula $\psi_{\texttt{wfp}}$. If α and β are any candidate input and output values, then the formula $\phi_{\texttt{func}}(L_0, \alpha, \beta)$ is true iff the program* `Lval2Prog`(L_0) *returns the value β on the input α.*

The proof of Theorem 1 follows from the results stated in [46].

We now describe the encoding more formally. Let $\mathbf{P}$ and $\mathbf{R}$ denote the union of all formal inputs (parameters) and formal outputs (return variables) of the components respectively, that is,

$\mathbf{P} := \bigcup_{i=1}^{N} \vec{I}_i \qquad \mathbf{R} := \bigcup_{i=1}^{N} \{O_i\} = \{O_1, \ldots, O_N\}$

Any straight-line program constructed using N components can be described by a set of *location variables* L

$$L := \{l_x \mid x \in \mathbf{P} \cup \mathbf{R}\}$$

that contains one new variable l_x for each variable x in $\mathbf{P} \cup \mathbf{R}$ with the following interpretation associated with each of these variables.

- If x is the output variable O_i of the component f_i, then l_x is the line number in the program where the component f_i is used.
- If x is the j^{th} input parameter of the component f_i, then l_x is the line number "from where component f_i gets its j^{th} input".

In the above description, line number refers to either a line of the program, or to some input. For uniformity, each input in $\vec{I}$ is assigned a line number from $0, \ldots, |\vec{I}| - 1$ and the program line

numbers then take values from $|\vec{I}|, \ldots, |\vec{I}|+N-1$. Let $M = |\vec{I}|+N$. The variables L take values in the range $0, \ldots, M-1$ and these new line numbers have the following interpretation.

- For $0 \leq j < |\vec{I}|$, line number j is blank; it takes the value of the j^{th} input of the program.
- For $|\vec{I}| \leq j < M$, line number j contains the $(j - |\vec{I}| + 1)$-th assignment statement of the original program P.

The well-formedness constraint $\psi_{\texttt{wfp}}(L)$, defined below, encodes the interpretation of the location variables l_x along with syntactic well-formedness constraints, such as consistency and acyclicity constraints.

$$\begin{aligned}
\psi_{\texttt{wfp}}(L) &\stackrel{\text{def}}{=} \bigwedge_{x \in \mathbf{P}} (0 \leq l_x < M) \ \wedge \ \bigwedge_{x \in \mathbf{R}} (|\vec{I}| \leq l_x < M) \\
&\qquad \wedge \ \psi_{\texttt{cons}}(L) \ \wedge \ \psi_{\texttt{acyc}}(L) \\
\psi_{\texttt{cons}} &\stackrel{\text{def}}{=} \bigwedge_{x,y \in \mathbf{R}, x \neq y} (l_x \neq l_y) \\
\psi_{\texttt{acyc}} &\stackrel{\text{def}}{=} \bigwedge_{i=1}^{N} \bigwedge_{x \in \vec{I}_i, y \equiv O_i} l_x < l_y
\end{aligned}$$

The consistency constraint $\psi_{\texttt{cons}}$ encodes that every line in the program should have at most one component, while the acyclicity constraint $\psi_{\texttt{acyc}}$ encodes that every variable should be initialized *before* it is used.

The function `Lval2Prog` returns the program corresponding to a given valuation L as follows: in the i^{th} line of `Lval2Prog`(L), we have the assignment $O_j := f_j(O_{\sigma(1)}, \ldots, O_{\sigma(t)})$ if $l_{O_j} = i$, $l_{I_j^k} = l_{O_{\sigma(k)}}$ for $k = 1, \ldots, t$, where t is the arity of component f_j, and $(I_j^1, \ldots, I_j^t)$ is the tuple of input variables $\vec{I}_j$ of f_j. The well-formedness constraint describes syntactically correct programs, but it does not describe the semantics of these programs.

The functional constraint $\phi_{\texttt{func}}(L, \vec{I}, O)$ is obtained by taking $\psi_{\texttt{wfp}}(L)$ and adding to it constraints capturing the dataflow semantics and semantics of components.

$$\begin{aligned}
\phi_{\texttt{func}}(L, \vec{I}, O) &\stackrel{\text{def}}{=} \exists \mathbf{P}, \mathbf{R} \ \psi_{\texttt{wfp}}(L) \ \wedge \ \phi_{\texttt{lib}}(\mathbf{P}, \mathbf{R}) \\
&\qquad \wedge \ \psi_{\texttt{conn}}(L, \vec{I}, O, \mathbf{P}, \mathbf{R}) \\
\phi_{\texttt{lib}}(\mathbf{P}, \mathbf{R}) &\stackrel{\text{def}}{=} (\bigwedge_{i=1}^{N} \phi_i(\vec{I}_i, O_i))
\end{aligned}$$

$$\psi_{\texttt{conn}}(L, \vec{I}, O, \mathbf{P}, \mathbf{R}) \stackrel{\text{def}}{=} \bigwedge_{x,y \in \mathbf{P} \cup \mathbf{R} \cup \vec{I} \cup \{O\}} (l_x = l_y \;\Rightarrow\; x = y)$$

where $\phi_{\texttt{lib}}$ represents the semantics of the base components (that relates the inputs and outputs of each component), and $\psi_{\texttt{conn}}$ represents the dataflow semantics (that matches the inputs and output of the different components and the inputs and output of the overall program with each other, in accordance with values of location variables).

The formula $\phi_{\texttt{func}}(L, \vec{I}, O)$ represents the class of all syntactically well-formed programs P, constructed using only the N base components, that on input $\vec{I}$ return output O. Hence, we can solve the program synthesis problem by finding appropriate values for the L variables. We need to find values for L such that the input-output behavior of the resulting program matches the input-output behavior specified by the I/O oracle.

A key step in our solution of the program synthesis problem is to *synthesize programs that work for finitely many input-output pairs*. We discuss this next.

I/O-behavioral Constraint

In this section, we show how to generate a constraint whose solution provides a candidate program whose input-output behavior matches a given *finite* set of input-output examples.

Given a set E of input-output examples $\{(\alpha_j, \beta_j)\}_j$, we use the notation $\texttt{Behave}_E$ to denote the following constraint, which we refer to as *I/O-behavioral constraint*.

$$\texttt{Behave}_E(L) \stackrel{\text{def}}{=} \bigwedge_{(\alpha_j, \beta_j) \in E} \phi_{\texttt{func}}(L, \alpha_j, \beta_j)$$

Let L_0 be a set of values such that $\texttt{Behave}_E(L_0)$ is true. It follows from the definition of the I/O-behavioral constraint that the program encoded by L_0 will give output β_j, whenever it is given an input α_j, for all pairs (α_j, β_j) in E. This property of the I/O-behavioral constraint is stated below.

Theorem 2 (I/O-behavioral Constraint)**.** *For any satisfying solution L_0 to the I/O-behavioral constraint, the input-output behavior of the program* $\texttt{Lval2Prog}(L_0)$ *matches all the input-output examples in the set E.*

The proof of the above theorem is immediate from the definition of an I/O-behavioral contraint and Theorem 1.

We next check if the program, which is synthesized by considering finitely many input-output pairs, is the desired program. We want to avoid the use of the validation oracle, since it is expensive. Here we use what is perhaps the central idea of our approach: *generate a "distinguishing" input that differentiates this program from another candidate program.*

Distinguishing Constraint

In this section, we show how to generate a constraint whose solution provides an input that distinguishes a given candidate program from another non-equivalent candidate program, both of which have a given set of input-output pairs in their respective input-output behavior.

Let E be a set of input-output pairs. Let P be a candidate program, defined by values L, whose input-output behavior matches the set E. Suppose P is not the desired program. Then, there should be some input $\vec{I}$ such that P gives incorrect output on $\vec{I}$. But, how do we find such an $\vec{I}$?

If P is not the desired program, then let us assume that there is a correct program P'. Clearly, for all input-output pairs (α_j, β_j) in E, the program P' should return β_j when it is given input α_j. But since P is not the desired program, whereas P' is the desired program, P and P' should give different outputs on some new input.

We say $\vec{I}$ is a distinguishing input if there is another program P' whose input-output behavior also matches E, but P and P' give different outputs on the input $\vec{I}$. The constraint $\texttt{Distinct}_{E,P}(\vec{I})$, defined below, represents the set of all distinguishing inputs $\vec{I}$ and we refer to it as *distinguishing constraint*.

$$\texttt{Distinct}_{E,L}(\vec{I}) \stackrel{\text{def}}{=} \exists L', O, O' \; \texttt{Behave}_E(L') \wedge \phi_{\texttt{func}}(L, \vec{I}, O) \\ \wedge \; \phi_{\texttt{func}}(L', \vec{I}, O') \; \wedge \; O \neq O'$$

Theorem 3 (Distinguishing Constraint)**.** *If α is a satisfying solution to the distinguishing constraint*
$\texttt{Distinct}_{E,P}(\vec{I})$*, then there exists a program P' such that P and P' have different behaviors on input α, but have the same behavior on all the inputs in the set E.*

The proof of Theorem 3 follows from the definition of the distinguishing constraint, Theorem 2 and Theorem 1. We now have all the ingredients for describing our overall procedure for solving the synthesis problem.

3.2.2 Oracle-Guided Synthesis

In this section, we describe our oracle-guided iterative synthesis procedure. The description uses the I/O-behavioral constraint and the distinguishing constraint described above.

The procedure works by iteratively synthesizing new programs that work correctly on more and more inputs. It starts with a set containing just one arbitrarily chosen input. In each iteration, the procedure synthesizes a program that works correctly on the current finite set of inputs. If such a program is found, then the procedure attempts to find a distinguishing input. If a distinguishing input is found, then it is added into the set of inputs for subsequent iterations. In all other cases, the procedure terminates. It either returns the correct program, or it notes that the components provided are insufficient for synthesizing the correct program.

For solving the I/O-behavioral constraint and the distinguishing constraint, the procedure makes use of a function $\texttt{T-SAT}$. Given a formula $\phi(A)$, the function $\texttt{T-SAT}(\phi(A))$ searches for values for A that will make the formula ϕ true. If successful, then $\texttt{T-SAT}(\phi(A))$ returns one such specific value for A. Otherwise, it returns $\perp$. The function $\texttt{T-SAT}$ is implemented as a call to a Satisfiability Modulo Theory (SMT) solver. SMT solvers check for satisfiability of a first-order formula with respect to underlying background theories [13].

The pseudo-code for the procedure is given in Figure 8. The procedure maintains a set E of input-output examples constructed by querying the I/O oracle $\mathcal{I}$ on a new input at the start of the while loop (Line 1) and in each iteration of the while loop (Line 16). In each iteration of the while loop, the procedure attempts to synthesize a candidate program P (represented by L) that satisfies the set E of input-output examples (Line 3). If it fails, then it returns failure (Line 5). Otherwise, it checks (in Line 8) whether the candidate program P is the semantically unique program that satisfies the given set of input-output examples. A program is semantically unique if any other program that satisfies the given set of input-output examples produces the same output as the program for any other input. If P is the semantically unique program, then the procedure either

returns P (Line 11) or failure (Line 14) depending on whether the validation oracle $\mathcal{V}$ validates P or not. If the candidate program is not semantically unique, then an input α is obtained that is added to E to help narrow down the choice of candidate programs (Line 16).

3.2.3 Illustration on Running Example

We illustrate the oracle-guided synthesis approach on the running example presented in Section 3.1.3. The problem was, given a bit-vector integer x, of finite but arbitrary length, to construct a new bit-vector y that corresponds to x with the rightmost string of contiguous 1s turned off.

Our technique starts with a random input 01011 and the I/O oracle $\mathcal{I}$ (the user) is used to obtain the corresponding expected output 01000. This step corresponds to Line 1 of the algorithm presented in Figure 8.

Given the input/output pair $(01011, 01000)$, our technique generates the following candidate program (Line 3): (we give only the expression returned)

$$(x + 1) \,\&\, (x - 1)$$

Then, it checks whether a semantically different program can be generated in Line 7. In this case, our technique generates the following alternative program and the distinguishing input 00000:

$$(x + 1) \,\&\, x$$

The I/O oracle is used to obtain the output 00000 for this input (Line 16). This is added to the set of input/output pairs E. Note that the newly added pair rules out one of the candidate programs, namely, $(x + 1) \,\&\, (x - 1)$.

In the next iteration, with the updated set E, the technique finds the program

$$-(\neg x) \,\&\, x$$

and the check in Line 7 generates the alternate program

$$(((x\& - x) \mid -(x - 1))\&x) \oplus x$$

and the input 00101. Hence, we add $(00101, 00100)$ to E. This rules out $(((x\& - x) \mid -(x - 1))\&x) \oplus x$.

Note that at this stage, the program $(x+1)\&x$ remains a candidate, since it was not ruled out in the earlier iterations. In next four iterations, BRAHMA generates $(01111, 00000)$, $(00110, 00000)$, $(01100, 00000)$ and $(01010, 01000)$ as input-output examples and adds them to E. The semanti-

cally unique program generated from the resulting set E is the desired program:

$$(((x-1)|x)+1)\&x.$$

3.2.4 Optimization

The basic procedure described above can be improved by using alternate ways to generate the inputs that are used by the procedure for synthesis.

`IterativeSynthesis` uses an SMT solver in two ways:
(a) First, an SMT solver is used to generate a candidate program that works for the current set of inputs.
(b) Second, an SMT solver is used to generate a new distinguishing input on which the currently synthesized program and the desired program potentially differ.

Although SMT solvers are fast and capable of handling very large formulas, using them in every iteration compromises efficiency. It is tempting to speculate that the use of SMT solvers for generating a distinguishing input (case (b) above) can be avoided; for example, by replacing it by a function that finds new inputs by sampling the input space in some way. We explore two alternative ways for sampling the input space.

Sampling Uniformly at Random

Let $\mathcal{I}$`nputs` be the set of all possible valuations for the input variables. Let `sample`($\mathcal{I}$`nputs`) be a function that returns a particular input from the input space $\mathcal{I}$`nputs` by sampling the set $\mathcal{I}$`nputs` uniformly at random. The function `sample`($\mathcal{I}$`nputs`) can be used to find a new input, in place of the call to the SMT solver, in Line 7 of Procedure `IterativeSynthesis`. We will call this new variant as `Random`.

Sampling With Bias

The second approach we consider is based on biasing the search for inputs towards a certain part of the input space. Not all inputs in the input space are equally important. For example, a

program may take an integer input i, but have the same behavior for all $i > 5$ and have interesting behaviors only on values $0 \leq i \leq 5$. For many applications, the user knows a-priori which inputs are more crucial in defining the overall program. The idea behind the sampling with bias strategy is to search for distinguishing inputs by biasing the search to this part of the input space.

In the bitvector benchmarks, the input space consists of all (tuples of) bitvectors of a certain bit width. It is well-known that, for a very large class of commonly-used bitvector functions, the rightmost bits influence the output more than the leftmost bits.

Property 1 (See [132], Chapter 2)**.** *A function mapping bitvectors to bitvectors can be implemented with add, subtract, bitwise and, bitwise or, and bitwise not instructions if and only if each bit of the output depends only on bits at and to the right of that bit in each input operand.*

This suggests that we should bias the sampling so that we get more variety on the rightmost bits.

Procedure 7 `ConstrainedRandomInput`: Strategy for generating a new input based on sampling from the input space with an application-dependent bias.

$\{cnt$ is a global variable initialized to $0\}$
$\{$ K is a parameter (number of rightmost bits to set) $\}$
if $cnt < 2^K$ **then**
  $\alpha :=$ `sample`$(\mathcal{I}$`nputs`$)$;
  $\alpha :=$Set rightmost K bits of α to cnt;
  $cnt := cnt + 1$;
else
  $\alpha :=$ `T-SAT`$($`Distinct`$_{E,L}(\vec{I}))$;
end if

The code `ConstrainedRandomInput` in Procedure 7 uses a constrained random strategy for generating a new input. It starts with an input α that is sampled uniformly at random, but then it sets its rightmost K bits to the (rightmost K bits in the) number cnt. Since cnt is incremented each time, we get a new combination in each time. Specifically, if $K = 2$, then in four calls to the Function `ConstrainedRandomInput`, we will get all four combinations – 00, 01, 10 and 11 – in the rightmost 2 digits of I. The code `ConstrainedRandomInput` finds the first 2^K

inputs this way. If more are needed, then it goes back to using the SMT solver. The new variant of `IterativeSynthesis` – obtained by replacing the call to the SMT solver in Line 7 by the code `ConstrainedRandomInput` – will be called `Constrained Random`.

We will compare the performance of `IterativeSynthesis`, `Random`, and `Constrained Random` in Section 3.4.

3.3 Discussion

3.3.1 Choosing Base Components

It is reasonable to ask how base components are chosen in our approach and what happens when the given set of base components is either insufficient or very large.

The choice of base components is made by the user and is guided by the application domain. This allows the user to use his/her knowledge to guide the synthesis and influence success. It is not unreasonable to expect users to provide this information. In several application domains, there is a natural choice for the set of base components. For example, a natural set of base components for synthesizing bitvector algorithms will contain components that perform bitwise and, or, not, xor, negation, increment and decrement operations. In our experiments on synthesizing bitvector programs (Section 3.4), we started with such a set of base components, referred to as the *standard library*. If the synthesis procedure found that this set of components was insufficient, the standard library was augmented with a set of new components suggested by the user and the synthesis procedure was re-run with this *extended library*.

Nevertheless, choosing a reasonable set of base components is crucial for the feasibility of our synthesis approach. The search space of candidate programs grows exponentially with the number of base components. The strategy of starting with a small set of base components, and then incrementally adding components, can partly avoid the need to deal with very large set of base components. However, it can be successful only if the synthesis engine not only synthesizes correct programs quickly, but also reports infeasibility of the synthesis problem quickly. In our experiments, we show that our technique can detect infeasibility efficiently.

When using our program synthesis approach for performing program deobfuscation, the base components are picked from the assignment and conditional statements in the obfuscated code. For the deobfuscation examples (reported in Section 3.4), the base components used for synthesis contain only operators (such as left-shift and bitwise-xor) that appear explicitly in the obfuscated code. For example, the deobfuscated program in $P24$ (Figure 3.4) multiplies the input by 45. It uses operators: `shift-left`$(<<)$ and `add`$(+)$, present in the obfuscated code as the component functions. It does not use `multiply`$(*)$ since it is not present in the obfuscated code.

3.3.2 Connections to Learning

Our oracle-guided synthesis framework has close connections to certain fundamental results in computational learning theory. We explore these connections in this section.

Our oracle-based model is similar to the query-based learning model proposed by Angluin [6], but with some important distinctions. In Angluin's model, a learner interacts with an oracle through the use of *membership* and *equivalence* queries in order to learn a *target concept*. In our setting, the *target concept* is the program we seek to synthesize. A membership query is similar to the query we make to an I/O oracle, except that the former returns a binary answer whereas the I/O oracle returns an output value. An equivalence query is similar to a query to the validation oracle, except that, in Angluin's model, if the candidate concept is not equivalent to the target concept; the oracle returns a counterexample as evidence for this non-equivalence. In our context, since the validation oracle is called only at the end, when we are left with a semantically unique program consistent with the set of examples, such a counterexample is not needed. Moreover, Angluin's model treats both kinds of queries as equally expensive. We make a distinction between the cheaper queries to the I/O oracle and the more expensive queries to the validation oracle, which allows us to optimize our implementation. Finally, our algorithm iterates by finding distinguishing inputs, which is not an operation supported by Angluin's model.

Two other results from learning theory also shed light on why our oracle-based approach is effective in practice.

First, note that our focus on loop-free programs that compute functions of finite-precision bit-vector inputs indicates a connection to the work on learning Boolean circuits. In particular, the classic result on learning constant-depth Boolean (AC^0) circuits from a few test inputs [78]

provides a partial explanation for the effectiveness of this strategy. The result relies on a theorem stating that AC^0 circuits can be approximated well by low-degree polynomials, which in turn are known to be identifiable by their behavior on few inputs.

The second relevant result relates to the notion of *teaching dimension* introduced by Goldman and Kearns [41]. Informally, the teaching dimension of a concept class is the minimum number of examples a teacher (oracle) must reveal to uniquely identify *any* target concept from that class. As our experiments show, we need very few examples to synthesize our target programs in practice, indicating that these programs form a concept class with a low teaching dimension. Moreover, our algorithm fits closely with a result by Goldman and Kearns [41], showing that the generation of an *optimal teaching sequence* of examples is equivalent to a minimum set cover problem. In the set cover problem for a given target concept, the universe of elements is the set of all incorrect concepts (programs) and each set S_i, corresponding to example x_i, contains concepts that are differentiated from the target concept by this example x_i. We can see that our Procedure `IterativeSynthesis` computes such a distinguishing example in each iteration, and terminates when it has computed a "set cover" that distinguishes the target concept from all other candidate concepts (the "universe"). Given this close connection, it does seem that the classes of functions corresponding to the bit-manipulating and deobfuscation examples we consider have small teaching dimension, and also Procedure `IterativeSynthesis` is effective at generating a sequence of examples close to the optimal teaching sequence.

3.4 Results and Experiments

3.4.1 Correctness Guarantee

The following theorem states the correctness of Procedure `IterativeSynthesis`. Note that if the inputs $\vec{I}$ take values from a finite domain, then the number of iterations of the loop in the procedure is bounded by the total number of different inputs; and hence, in such cases the procedure is guaranteed to terminate.

Theorem 4. *If Procedure* `IterativeSynthesis`*, given in Procedure 8, returns a program P, then $\mathcal{V}(P)$ is true. If Procedure* `IterativeSynthesis` *returns "Components insufficient",*

then there does not exist any program P constructed from the set of base-components such that $\mathcal{V}(P)$ is true. Furthermore, Procedure `IterativeSynthesis` *is guaranteed to terminate when the inputs $\vec{I}$ take values from a finite domain.*

Proof. The proof of the correctness theorem follows immediately from the description of the procedure in Procedure 8, combined with the properties stated in Theorem 1, Theorem 2 and Theorem 3. We also illustrate in Figure 3.1 all three cases in which Procedure `IterativeSynthesis` terminates. The first case corresponds to step 7 and the second and third cases correspond to step 11 and step 12 respectively. □

Procedure 8 `IterativeSynthesis`(): Oracle-guided Synthesis Procedure

Input: Set of base components used in construction of $\texttt{Behave}_E$ and $\texttt{Distinct}_{E,L}$

Output: Candidate Program

$E := \{(\alpha_0, \mathcal{I}(\alpha_0))\}$ // α_0 is an arbitrary value for $\vec{I}$
while 1 **do**
  $L := \texttt{T-SAT}(\texttt{Behave}_E(L))$;
  if $(L == \perp)$ **then**
    return "Components insufficient";
  end if
  $\alpha := \texttt{T-SAT}(\texttt{Distinct}_{E,L}(\vec{I}))$;
  if $(\alpha == \perp)$ **then**
    $P := \texttt{Lval2Prog}(L)$;
    if $(\mathcal{V}(P))$ **then**
      return P;
    end if
  else
    return "Components insufficient";
  end if
  $E := E \cup \{\alpha, \mathcal{I}(\alpha)\}$;
end while

From Figure 3.1, we observe that the program synthesized by our SCIDUCTION based synthesis technique is guaranteed to be correct if the structure hypothesis that "a correct program can by

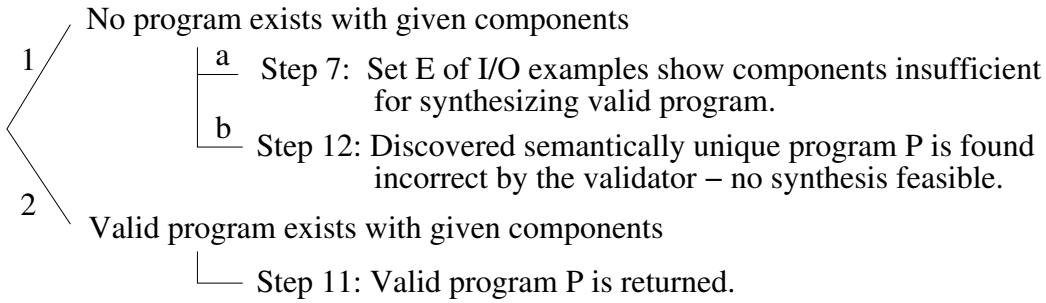


Figure 3.1: Termination cases of Synthesis Procedure. The validation oracle is needed only to ensure correctness in case 1b.

synthesized using components in the library" is correct.

3.4.2 Experiments

We present experimental evaluation of our technique and compare different approaches namely `IterativeSynthesis`, `Random`, and `Constrained Random` discussed in Section 3.2.

Setup and Benchmarks We have implemented `IterativeSynthesis` in a tool called BRAHMA. It uses Yices 1.0.21 [120] as the underlying SMT solver. We ran our experiments on 8x Intel(R) Xeon(R) CPU 1.86GHz with 4GB of RAM. BRAHMA was able to synthesize the desired circuit for each of the benchmark examples. Semi-biased BRAHMA implements `Constrained Random` with the parameter $K = 2$. Thus, it differs only in first 4 steps from BRAHMA. As mentioned in Section 3.2, this is specially targetted towards synthesis of bitvector programs.

We selected a set of 25 benchmark examples to evaluate our technique. 22 benchmarks (P1-P22) are bit-manipulation programs from the book *Hacker's Delight*, commonly referred to as the Bible of bit twiddling hacks [132]. 3 benchmarks were used as examples to illustrate the use of our technique for deobfuscation. These benchmarks reflect obfuscation strategies from literature on obfuscation techniques [27] (P23) and Internet worms - Conficker [105] (presented in Section 1.2.1) and MyDoom [100] (P25).

P1(x) : Turn-off rightmost 1 bit.

1 o_1=(x - 1)

2 res=(x && o_1)

P2(x) : Test whether an unsigned integer is of the form $2^n - 1$

1 o_1=(x + 1)

2 res=(x && o_1)

P3(x) : Isolate the rightmost 1-bit

1 o_1=(- x)

2 res=(x && o_1)

P4(x) : Form a mask that identifies the rightmost 1 bit and trailing 0s

1 o_1=(x - 1)

2 res=(x $\oplus$ o_1)

P5(x) : Right propagate rightmost 1-bit

1 o_1=(x - 1)

2 res=(x || o_1)

P6(x) : Turn on the rightmost 0-bit

1 o_1=(x + 1)

2 res=(x || o_1)

P7(x) : Isolate the rightmost 0-bit

1 o_1=($\neg$ x)

2 o_2=(x + 1)

3 res=(o_1 && o_2)

P8(x) : Form a mask that identifies the trailing 0's

1 o_1=(x - 1)

2 o_2=($\neg$ x)

3 res=(o_1 && o_2)

P9(x) : Absolute Value Function

1 o_1=(x >> 31)

2 o_2=(x $\oplus$ o_1)

3 res=(o_2 - o_1)

P10(x, y) : Test if nlz(x) == nlz(y) where nlz is number of leading zeroes

1 o_1=(x && y)

2 o_2=(x $\oplus$ y)

3 res=($o_2 \leq_u o_1$)

P11(x, y) : Test if nlz(x) $<$ nlz(y)

1 o_1=($\neg$ y)

2 o_2=(x && o_1)

3 res=($o_2 >_u$y)

P12(x, y) : Test if nlz(x) <= nlz(y) where nlz is number of leading zeroes

1 o_1=($\neg$ y)
2 o_2=(x && o_1)
3 res=($o_2 \leq_u$ y)

P13(x) : Sign Function

1 o_1=(x >> 31)
2 o_2=(- x)
3 o_3=(o_2 >> 31)
4 res=(o_1 || o_3)

P14 (x, k) : Round up x to a multiple of k-th power of 2

1 o_1=(-1 >> k)
2 o_2=(o_1 + 1)
3 o_3=(x - o_2)
4 res=(o_3 && o_1)

P15 (x, y) : Floor of average of two integers without over-flowing

1 o_1=(x && y)
2 o_2=(x $\oplus$ y)
3 o_3=(o_2 >> 1)
4 res=(o_1 + o_3)

P16 (x, y) : Compute max of two integers

1 o_1=(x $\oplus$ y)
2 o_2=(- (x $\geq_u$ y))
3 o_3=(o_1 && o_2)
4 res=(o_3 $\oplus$ y)

P17 (x, y) : Compute min of two integers

1 o_1=(x $\oplus$ y)
2 o_2=(- (x $\leq_u$ y))
3 o_3=(o_1 && o_2)
4 res=(o_3 $\oplus$ y)

P18 (x, y) : Ceil of average of two integers without over-flowing

1 o_1=(x || y)
2 o_2=(x $\oplus$ y)
3 o_3=(o_2 >> 1)
4 res=(o_1 - o_3)

P19(x) : Turn-off the rightmost contiguous string of 1 bits

1 o_1=(x - 1)
2 o_2=(x || o_1)
3 o_3=(o_2 + 1)
4 res=(o_3 && x)

P20(x) : Determine if an integer is a power of 2 or not

1 o_1=(x - 1)
2 o_2=(o_1 && x)
3 o_3=bvredor (x)
4 o_4=bvredor (o_2)
5 o_5=!(o_4)
6 res=(o_5 && o_4)

P21(x) : Next higher unsigned number with same number of 1 bits

1 o_1=(- x)
2 o_2=(x && o_1)
3 o_3=(x + o_2)
4 o_4=(x $\oplus$ o_2)
5 o_5=(o_4 >> 2)
6 o_6=(o_5 / o_2)
7 res=(o_6 || o_3)

P22(x) : Round up to the next highest power of 2

1 o_1=(x - 1)
2 o_2=(o_1 >> 1)
3 o_3=(o_1 || o_2)
4 o_4=(o_3 >> 2)
5 o_5=(o_3 || o_4)
6 o_6=(o_5 >> 4)
7 o_7=(o_5 || o_6)
8 o_8=(o_7 >> 8)
9 o_9=(o_7 || o_8)
10 o_{10}=(o_9 >> 16)
11 o_{11}=(o_9 || o_{10})
12 res=(o_{10} + 1)

Figure 3.2: Bit-vector Benchmarks

P23: Interchange the source and destination addresses.

```
1  interchangeObs(IPaddress* src , IPadress* dest )
2  { *src = *src ⊕ *dest ;
3    if (*src == *src ⊕ *dest )
4       { *src = *src ⊕ *dest ;
5       if (*src == *src ⊕ *dest )
6          { *dest = *src ⊕ *dest ;
7          if (*dest == *src ⊕ *dest )
8             { *src = *dest ⊕ *src ;
9             return; }
10         else
11           { *src = *src ⊕ *dest ;
12           *dest = *src ⊕ *dest ;
13           return;} }
14      else
15        *src = *src ⊕ *dest ; }
16   *dest = *src ⊕ *dest ; *src = *src ⊕ *dest ; return;
17   }
```

Deobfuscated Version

```
1  interchange(IPaddress* src , IPadress* dest )
2  {
3  *dest = *src ⊕ *dest ;
4  *src = *src ⊕ *dest ;
5  *dest = *src ⊕ *dest ;
6  return;
7  }
```

Figure 3.3: Deobfuscation Benchmark P23

P24: Multiply with 45.

```
1  mul45Obs(Bitvector x)
2  a = 1, b = 0, z = 1, c = 0
3  while {1}
4        if {a == 0} {
5           if b == 0 {
6              y = z + y; a = ¬a; b = ¬b; c = ¬c;
7              if {¬c} break;
8              }
9           else {
10                z = z + y; a = ¬a; b = ¬b; c = ¬c;
11                if {¬c} break;
12                }
13          }
14       else {
15             if{b == 0} {z = y << 2; a = ¬a;}
16             else {z = y << 3, a = ¬a, b = ¬b}
17             }
18       }
19 return y;
```

Deobfuscated Version

```
1  mul45(Bitvector x)
2  z = y << 2;
3  y = z + y;
4  z = y << 3;
5  y = z + y;
6  return y
```

Figure 3.4: Deobfuscation Benchmark P24

P25: SMTP example from MyDoom

```
1  genStringObs(int input)
2  {
3    a1 = 1; a2 = 0;b1 = 1; b2 = 0;c1 = 0; c2 = 0;;
4    if (input == 0) {
5       a1 = 0; a2 = 0;b1 = 0; b2 = 0; }
6    else if (input == 1) {
7         c1 = 0; c2 = 1; }
8    else if (input == 2) {
9         a1 = 1; a2 = 0;c1 = 1; c2 = 1; }
10   else if (input == 3) {
11        b1 = 0; b2 = 0;c1 = 1; c2 = 1; }
12   else return NULL;
13   c = 2 * c1 + c2;
14   if(c == 1) {
15   return rot13("EPCGGB, 7"); }
16   else
17   if (c == 2) {
18      if (input * (input − 1) mod 2 == 0)
19         return rot13("EPCGGB", 7);
20      else
21        return rot13("RUYB", 4); }
22   else {
23        if (b1 ⊕ b2)
24        return rot13("ZNVYSEBZ", 9)
25        else if ((a1 ⊕ a2) ≠ (b1 ⊕ b2))
26             return rot13("RUYB", 4);
27        else return rot13("QNGN", 4); } }
```

```
1 rot13(char *buf, int sz)
2 {
3   char *buf1 = malloc((sz + 1) * sizeof(char));
4   char a;
5   while (a =~ *buf)
6   {
7     *buf1 = (~a-1/(~((a || 32))/13*2-11)*13);
8     buf++; buf1++;
9   }
10  return buf1;
11 }
```

Deobfuscated Version

```
1 genString(int input)
2 { if(input == 0)
3   return "EHLO";
4   else if (input == 1)
5   return "RCPTTO";
6   else if (input == 2)
7   return "MAILFROM";
8   else if (input == 3)
9   return "DATA";
10  else return NULL;
11 }
```

Figure 3.5: Deobfuscation Benchmark P25

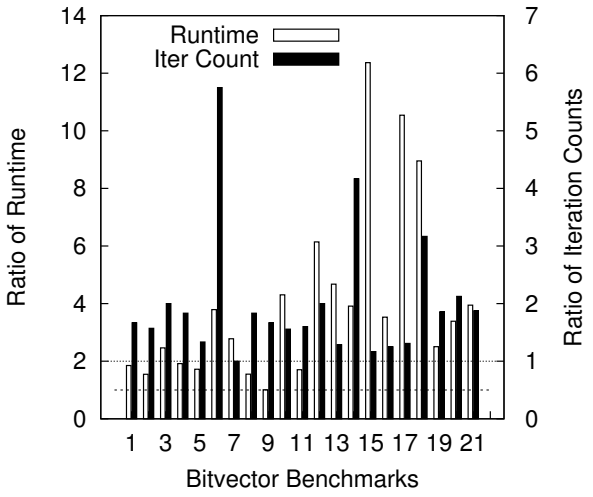


Figure 3.6: Ratio of Runtime for Random Input Generation to SemiBiased BRAHMA

Bit-Manipulating Programs

The bitvector benchmarks were run using a *standard library* of base components, and if necessary, an *extended library* as discussed in Section 3.3.1. In Table 3.1, we report the runtime when using the standard library (col. 4) and when using the user-augmented extended library (col. 5), in case the standard library was not sufficient. Note that our tool quickly terminates when the given library is insufficient.

For bitvector benchmarks, the user plays the role of the I/O oracle as well as the validation oracle. If the user guarantees that the provided set of base components is sufficient to encode the desired solution, then we do not require the validation oracle. Otherwise, it is theoretically impossible to know whether or not the generated solution is the correct one without a validation oracle. However, in practice, our algorithm detects insufficiency of the base components by discovering inconsistency, and not by a query to the validation oracle. This suggests that in the absence of any validation oracle, we can consider the semantically unique candidate program returned by our algorithm to be the correct program for all practical purposes.

We now compare the three approaches on bit-vector benchmarks using two metrics - the total runtime and the number of iterations. We present the ratio of runtimes of random input generation (col 2 of Table 3.1) and semi-biased BRAHMA (col 5 of Table 3.1) in Figure 3.6. Semi-biased BRAHMA is 1.5 times to 12 times faster than random technique. For P22, the random technique

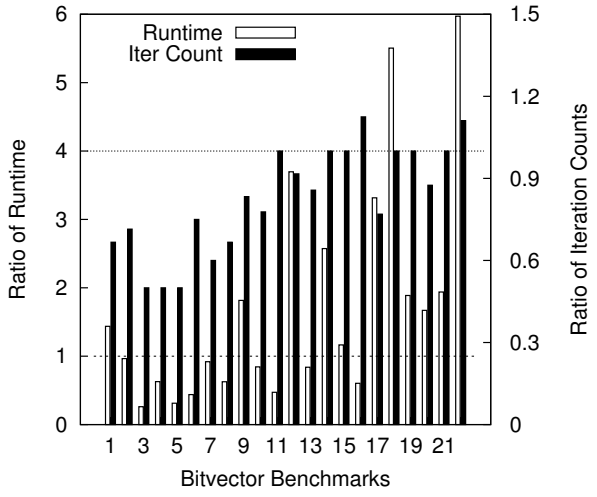


Figure 3.7: Ratio of Runtime for BRAHMA to SemiBiased BRAHMA

did not finish in 1 hour while semi-biased BRAHMA was able to synthesize it in 186 seconds. Also, the number of iterations required to synthesize a program is also reduced significantly as shown in Table 3.1. BRAHMA and semi-biased BRAHMA is compared in Figure 3.7. While the number of iterations is more for semi-biased BRAHMA, it is faster than the BRAHMA on larger benchmarks. It reduces the runtime for P18 from 140.65 seconds to 25.55 seconds, P21 from 527.91 seconds to 272.28 seconds and P22 from 1108.15 seconds to 187.17 seconds. This validates the optimization proposed in Section 3.2.

Deobfuscation

The I/O oracle involves simply evaluating the obfuscated program on the given input. The validation oracle can be a program equivalence checking tool or the user.

An additional challenge that this domain offers is the presence of arbitrary string constants. Our synthesis framework can be easily extended to discovering such constants. For this purpose, we introduce a generic base component f_c that simply outputs some arbitrary constant c. The component f_c takes no input and returns one output O and its functional specification is written as $O = c$. The only change to the framework is that since c is allowed to be arbitrary, we existentially quantify over c in the functional constraint $\phi_{\texttt{func}}$ described in Section 3.2.1.

For the three examples that we used in experiments, observe that BRAHMA gives the best

Bench	Random Inputs		Semibiased BRAHMA		
Names	Runtime	Iter	Runtime Standard Lib	Runtime Extended Lib	Iter
1	2	3	4	5	6
P1	1.48	5	0.80*	0.80	3
P2	7.35	11	4.75*	4.75	7
P3	1.60	8	0.65*	0.65	4
P4	1.65	11	0.86*	0.86	6
P5	3.92	8	2.28*	2.28	6
P6	6.22	23	1.64*	1.64	4
P7	1.39	5	0.50*	0.50	5
P8	2.20	11	1.42*	1.42	6
P9	4.95	10	3.85	4.90	6
P10	13.99	14	4.57	3.25	9
P11	24.31	16	2.86	14.27	10
P12	279.49	24	2.64	45.52	12
P13	32.50	9	3.02	6.95	7
P14	14.32	25	3.00	3.66	6
P15	167.84	7	4.50	13.57	6
P16	66.93	10	4.95	18.97	8
P17	217.34	17	5.89	20.62	13
P18	228.78	19	7.98	25.55	6
P19	163.82	13	65.45*	65.45	7
P20	214.14	17	19.30	63.23	8
P21	1074.04	15	13.28	272.28	8
P22	timeout	NA	187.17	185.57	9

Table 3.1: Random input generation and Semi-biased BRAHMA on Bitvector Examples. NA denotes not applicable. * denotes that the extended library was same as standard library. Runtimes in sec.

Bench	BRAHMA		Random		Semibiased BRAHMA	
Names	Runtime (sec)	Iter	Runtime (sec)	Iter	Runtime (sec)	Iter
P23	1.380	3	24.28	9	12.12	5
P24	5.28	2	11.96	4	2.94	2
P25	0.50	5	timeout	NA	0.86	9

Table 3.2: Deobfuscation Examples

performance. The key observation from the experiments is that random input generation does not work well for examples such as P25 where randomly generating integers has a rare chance of 1 in 2^{32} to pick an input which produces any of the first 4 possible outputs. BRAHMA takes exactly 5 iterations to query the I/O oracle with inputs that generate all the 5 possible outputs.

The experimental results indicate that adding a distinguishing input is better than adding a random input to E because it guarantees that at least one candidate program is definitely removed from the search space. Thus, it guarantees progress. Moreover, it possibly also removes a set of other similar designs from the search space.

3.5 Conclusion

We have presented a novel approach to program synthesis based on oracle-guided learning from examples and SMT solvers. Applications to synthesis of bit-vector programs and deobfuscation have been demonstrated. Experiments indicate that our approach can be efficient and effective for discovering unintuitive code and for program understanding.

Chapter 4

Synthesis of Optimal Fixed-Point Code

Programs written in the domains of digital signal processing and embedded systems have two important characteristics. First, they commonly contain procedures that compute functions of their inputs, where these functions are mathematically specified as operating on the reals; examples include filters used in signal conditioning and the computation of control inputs. Second, they must run in resource-constrained environments and/or at high performance, requiring their optimization for low resource cost (e.g., low power) as well as for performance. This chapter addresses a problem arising from the interaction of these two characteristics.

Specifically, at the high-level design stage (which could involve manually writing a "reference" program or using model-based design environments such as Simulink/Stateflow and LabVIEW), the reals are approximated with floating-point arithmetic. Designers create signal processing or control algorithms as programs based on floating-point arithmetic. However, when these algorithms must be implemented in software, they must be optimized for power and performance. It is common for embedded platforms to have processors without floating-point units due to their added cost and performance penalty. Such platforms increasingly include hardware such as field-programmable gate arrays (FPGAs), on which fixed-point arithmetic can be efficiently implemented. The signal processing/control engineer must, thus, redesign her floating-point program to instead use *fixed-point arithmetic*. The tricky part of the redesign process is to find the *optimal fixed-point types*, viz., the optimal bit-widths of fixed-point variables, so that the implementation on the platform is optimal — lowest cost and highest performance — *and* the resulting fixed-point program is sufficiently accurate. Accuracy is particularly important for safety-critical embedded

systems where the control input to actuators is the result of such a fixed-point computation. Thus, to summarize, the conversion from floating-point to fixed-point is subject to two opposing constraints: (i) the width of fixed-point types must be minimized, and (ii) the outputs of the fixed-point program must be accurate, lying within a specified distance of those of the floating-point version.

In this chapter, we propose a new approach to compute a fixed-point version of a floating-point function based on inductive synthesis. Our technique , named SCIDUCTION, takes the floating-point program, specified accuracy, and an implementation cost model as input, and generates the fixed-point program with specified accuracy and optimal implementation cost. The core idea in SCIDUCTIONis to perform inductive synthesis from input-output examples using optimization oracles. The optimization oracles are iteratively invoked to generate candidate programs that are optimal for a set of examples as well as to find new input-output examples to drive the synthesis process. The following novel contributions are made:

- We present a new approach for inductive synthesis of fixed-point programs from floating-point versions. The novelty stems in part from our use of optimization: we not only use optimization routines to minimize fixed-point types (bit-widths of fixed-point variables), as previous approaches have, but also show how to use an optimization oracle to systematically generate input-output examples for inductive synthesis.
- We present theoretical guarantees on when our approach finds minimum fixed-point types at specified accuracy: using optimization oracles that find globally-optimal solutions guarantees that we will find the optimal fixed-point program meeting the specified accuracy.
- We illustrate the practical effectiveness of our technique on programs drawn from the domains of digital signal processing and control theory. For the control theory examples, we not only exhibit the synthesized fixed-point programs, but also show that these programs, when integrated in a feedback loop with the rest of the system, perform as accurately as the original floating-point versions.

The chapter is organized as follows. Sec. 4.1 presents a formal definition of the problem and a running example. Our approach is described in Sec. 4.2. We present our experimental results in Sec. 4.3 and conclude in Sec. 4.4.

4.1 Problem Definition

Numerical computation with specified accuracy is of fundamental importance in many applications such as digital signal processing and control systems. Algorithms for these applications are often designed ignoring the finite precision of computer arithmetic. Quantization errors due to finite precision are analyzed later. Floating-point environments are very useful in providing a convenient design and validation environment to designers who can use it to explore the algorithm space and also quickly simulate the design using standard floating-point units available on computers. When these algorithms need to be implemented on hardware, floating-point data types are converted to fixed-point data types in order to reduce the cost of hardware needed as well as to decrease the power requirement and increase the speed of computation. The conversion from floating-point to fixed-point often results into decreased computation accuracy. An optimal selection of word-lengths of fixed-point variables to ensure accuracy remains above the specified threshold and the implementation cost is low, is a challenging task.

Along with a floating-point implementation of numerical computation, the user can provide additional specification regarding the desired fixed-point implementation. The floating-point program takes inputs from a given *input domain*. This input domain can be provided as a logical condition on the inputs. Numerical computation is expected to have some *specified threshold of numerical accuracy* which can be provided as a correctness condition for the fixed-point program. Further, the fixed-point implementation has a cost associated with it and an optimal implementation is expected to take minimal cost. The user can provide a cost model for the implementation. We use a simple illustrative example to explain the problem of synthesizing optimal fixed-point program from floating-point program, and then present the formal problem definition.

4.1.1 Floating-point Implementation

Floating-point implementation of the numerical computation provides the scaffold of the fixed-point program. Each floating-point variable needs to be translated to fixed-point datatype. and hence, the synthesis task is that of discovering the appropriate wordlength of the fixed-point variables such that the correctness criteria is met for all inputs satisfying the input condition and the implementation cost is minimized.

Example 4.1: We present an example in Procedure 9 that illustrates this problem and the difficulty that programmers face in doing the floating-point to fixed-point translation manually. The floating-point program in this example takes `radius` as the input, and computes the `area` of the circle with this `radius`.

Procedure 9 Floating-point program to compute area of circle

Input: `radius`

Output: `area`

double `mypi, radius, t, area`

`mypi` $=$ `3.14159265358979323846`

`t` $=$ `radius` $\times$ `radius`

`area` $=$ `mypi` $\times$ `t`

return `area`

From the provided floating-point program, it is easy to extract the scaffold of the fixed-point program. The only unknowns are the fixed-point type of the variables `mypi`, `radius`, `t` and `area`. So, the scaffold fixed-point program takes as input both the radius and the wordlengths for the fixed-point variables. Recall that the fixed-point type is a 3-tuple $\langle \mathtt{s_j}, \mathtt{iwl_j}, \mathtt{fwl_j} \rangle$ for j-th variable where $\mathtt{s_j}$ denotes the signed-ness of the variable, $\mathtt{iwl_j}$ denotes the integer wordlength and $\mathtt{fwl_j}$ denotes the fraction wordlength.

Procedure 10 Fixed-point program to compute area of circle

Input: `radius`, $\langle \mathtt{s_j}, \mathtt{iwl_j}, \mathtt{fwl_j} \rangle$ for $\mathtt{j} = 1, 2, 3, 4$

Output: `area`

$\mathbf{fx}\langle \mathbf{s_1}, \mathbf{iwl_1}, \mathbf{fwl_1} \rangle$ `mypi`

$\mathbf{fx}\langle \mathbf{s_2}, \mathbf{iwl_2}, \mathbf{fwl_2} \rangle$ `radius`

$\mathbf{fx}\langle \mathbf{s_3}, \mathbf{iwl_3}, \mathbf{fwl_3} \rangle$ `t`

$\mathbf{fx}\langle \mathbf{s_4}, \mathbf{iwl_4}, \mathbf{fwl_4} \rangle$ `area`

`mypi` $=$ `3.14159265358979323846`

`t` $=$ `radius` $\times$ `radius`

`area` $=$ `mypi` $\times$ `t`

return `area`

We use $F_{fl}(X)$ to denote the floating-point program with inputs $X = x_1, x_2, \ldots, x_n$ and $F_{fx}(X, \mathbf{fx}\tau)$ to denote the fixed-point version of the program with fixed-point type $\mathbf{fx}\tau$.

4.1.2 Input Domain

The context in which a numerical fixed-point program $F_{fx}(X, \mathbf{fx}\tau)$ is executed often provides a precondition that must be satisfied by valid inputs $(x_1, x_2, \ldots, x_n)$. This defines the input domain denoted by $Dom(X)$.

Example 4.2: In our example of computing area of a circle, our interest is in radius in the interval $[0.1, 2]$ and so the input domain $Dom(\texttt{radius})$ is

$$\texttt{radius} \geq 0.1 \wedge \texttt{radius} < 2$$

4.1.3 Correctness Condition for Accuracy

Correctness condition for accuracy involves specification of a suitable error function and a maximum threshold of error that the fixed-point program can have with respect to floating-point program. Let $F_{fl}(X)$ be floating-point and $F_{fx}(X, \mathbf{fx}\tau)$ be the corresponding fixed-point function with fixed-point type $\mathbf{fx}\tau$. The error function $Err(F_{fl}(X), F_{fx}(X, \mathbf{fx}\tau))$ is provided by the user along with the maximum error threshold `maxError`. Some common error functions can be:

- Absolute difference between the floating-point function and fixed-point function, that is,

$$|F_{fl}(X) - F_{fx}(X, \mathbf{fx}\tau)|$$

.

- Relative difference between the floating-point function and fixed-point function, that is,

$$\left| \frac{F_{fl}(X) - F_{fx}(X, \mathbf{fx}\tau)}{F_{fl}(X)} \right|$$

.

- Moderated relative difference, that is,

$$\left| \frac{F_{fl}(X) - F_{fx}(X, \mathbf{fx}\tau)}{F_{fl}(X) + \delta} \right|$$

  which approaches relative difference for $F_{fl}(X) >> \delta$ and approaches weighted absolute difference for $F_{fl}(X) << \delta$. When $F_{fl}(X)$ can be zero for some values of X, the moderated relative difference remains bounded unlike the relative difference which becomes unbounded.

The *correctness condition for accuracy* requires that for all inputs in the provided *input domain* $Dom(X)$, the error function $Err(F_{fl}(X), F_{fx}(X, \mathbf{fx}\tau))$ is below the specified threshold `maxError`, that is,

$$\forall X \in Dom(X) \; . \; Err(F_{fl}(X), F_{fx}(X, \mathbf{fx}\tau)) \leq \texttt{maxError}$$

Example 4.3: In our running example of computing area of a circle, the error function is chosen to be relative difference, and the correctness condition for accuracy is

$$\forall \texttt{radius}, \; such\ that, \; \texttt{radius} \geq 0.1 \wedge \texttt{radius} < 2$$

$$\frac{F_{fl}(\texttt{radius}) - F_{fx}(\texttt{radius}, \mathbf{fx}\tau)}{F_{fl}(\texttt{radius})} \leq 0.1$$

4.1.4 Implementation Cost Model

The scaffold for the floating-point program is obtained from the fixed-point program. The synthesis process discovers the fixed-point types of the variables used in the program. So, the cost of different possible implementations differs only in the fixed-point types of the variables. Recall that fixed-point type is 3-tuple $\langle \mathtt{s_j}, \mathtt{iwl_j}, \mathtt{fwl_j} \rangle$ for j-th variable where $\mathtt{s_j}$ denotes the signed-ness of the variable, $\mathtt{iwl_j}$ denotes the integer wordlength and $\mathtt{fwl_j}$ denotes the fraction wordlength. The *cost model* of the implementation is a function of the chosen fixed-point types of the variables. In practice, it is often just a function of the total wordlengths ($\mathtt{WL} = \mathtt{IWL} + \mathtt{FWL}$) of the variables. The cost model can incorporate implementation area, power and other metrics of interest. For a given fixed-point program $F_{fx}(X, \mathbf{fx}\tau)$, let $T = \{t_1, t_2, \ldots, t_k\}$ be the set of fixed-point program variables with corresponding types $\{\mathbf{fx}\tau(t_1), \mathbf{fx}\tau(t_2), \ldots, \mathbf{fx}\tau(t_k)\}$. The cost of the fixed-point

implementation is a function from the fixed-point types of the variables to a real value, that is, cost of the fixed-point program F_{fx} is

$$\texttt{cost} : (\mathbf{fx}\tau(t_1), \mathbf{fx}\tau(t_2), \ldots, \mathbf{fx}\tau(t_k)) \rightarrow \mathbb{R}$$

A number of area-cost and power-cost models [76, 85, 26, 38] of hardware implementing fixed-point program can be used as cost function in our technique.

Example 4.4: The area model proposed by Constantinides et al [38] for the running example yields the following cost function. We use this cost model in all our examples.

$$\texttt{cost}(\mathbf{fx}\tau(\texttt{mypi}), \mathbf{fx}\tau(\texttt{radius}), \mathbf{fx}\tau(\texttt{t}), \mathbf{fx}\tau(\texttt{area})) =$$

$$\texttt{costdelay}(\texttt{IWL}(\texttt{mypi}))$$

$$+\texttt{costmul}(\texttt{IWL}(\texttt{radius}), \texttt{IWL}(\texttt{radius}), \texttt{IWL}(\texttt{t}))$$

$$+\texttt{costmul}(\texttt{IWL}(\texttt{mypi}), \texttt{IWL}(\texttt{t}), \texttt{IWL}(\texttt{area}))$$

where

$$\texttt{costdelay}(l) = l + 1$$

and

$$\texttt{costmul}(l_1, l_2, l) = 0.6 \times (l_1 + 1) * l_2 - 0.85 * (l_1 + l_2 - l)$$

.

Our technique can be used with any cost model. In all our experiments as well as the running example, we use the Constantinides model [38].

4.1.5 Problem Definition

Definition 1 (Optimal Fixed-point Program Synthesis)**.** *The optimal fixed-point program synthesis problem is as follows. Given*

1. *a floating-point program $F_{fx}(X, \mathbf{fx}\tau(T))$ with fixed-point variables T,*
2. *an input domain $Dom(X)$*

3. *a correctness condition* $Err(F_{fl}(X), F_{fx}(X, \mathbf{fx}\tau(T))) \leq \texttt{maxError}$

4. *a cost model* $\texttt{cost}(\mathbf{fx}\tau(t_1), \mathbf{fx}\tau(t_2), \ldots, \mathbf{fx}\tau(t_k))$

optimal fixed-point program synthesis problem is to discover fixed-point types

$$\mathbf{fx}\tau(T)^* = \{\mathbf{fx}\tau(t_1), \mathbf{fx}\tau(t_2), \ldots, \mathbf{fx}\tau(t_k)\}$$

such that the fixed-point program $F_{fl}(X)$ *with the above types for fixed-point variables*

- *(a) is correct with respect to the correctness condition for accuracy, that is,*

$$\forall X \in Dom(X)\ .\ Err(F_{fl}(X), F_{fx}(X, \mathbf{fx}\tau(T)^*)) \leq \texttt{maxError}$$

- *(b) has minimal cost with respect to the given cost function among all fixed-point types that satisfy condition (a), that is,*

$$\mathbf{fx}\tau(T)^* = \underset{\mathbf{fx}\tau(T)\ \texttt{satisfies (a)}}{\operatorname{argmin}} \texttt{cost}(\mathbf{fx}\tau(T))$$

Our goal is to automated this search for fixed-point types in order to synthesize fixed-point program that is firstly, correct with respect to the correctness condition for accuracy and then, optimal with respect to the given cost model. We illustrate this problem using the running example below.

Example 4.5: In our running example of computing the *area of a circle*, we need to discover $\mathbf{fx}\tau(\texttt{mypi})$, $\mathbf{fx}\tau(\texttt{radius})$, $\mathbf{fx}\tau(\texttt{t})$ and $\mathbf{fx}\tau(\texttt{area})$ such that the fixed-point program satisfies the correctness condition

$$(a)\ \ \forall \texttt{radius},\ \textit{such that},\ \texttt{radius} \geq 0.1 \wedge \texttt{radius} < 2$$

$$\frac{F_{fl}(\texttt{radius}) - F_{fx}(\texttt{radius}, \mathbf{fx}\tau)}{F_{fl}(\texttt{radius})} \leq 0.1$$

and the cost is minimized, that is,

$$(b) \quad \underset{\mathbf{fx}\tau\ \texttt{satisfies (a)}}{\operatorname{argmin}} \texttt{cost}(\mathbf{fx}\tau(\texttt{mypi}, \texttt{radius}, \texttt{t}, \texttt{area}))$$

We use this example to illustrate the trade-off between cost and error and how, a human would use trial and error to discover the correct wordlengths.

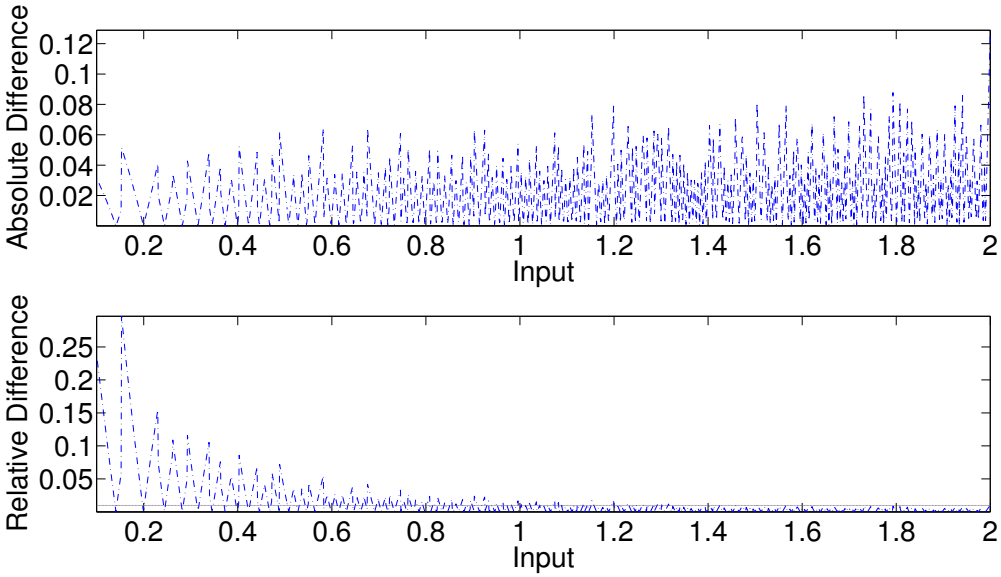


Figure 4.1: Wordlength 8

Case 1: In the first case, all wordlengths (WL) are fixed to 8. Integer wordlengths (IWL) are selected to avoid overflow and the remaining bits are used for fractional wordlength (FWL). Since all variables will always take positive values, we set the signed-ness bit to 0. Thus, the fixed-point types for the variables are: $\mathbf{fx}\tau(\texttt{mypi}) = \langle 0, 2, 6\rangle, \mathbf{fx}\tau(\texttt{radius}) = \langle 0, 1, 7\rangle, \mathbf{fx}\tau(\texttt{t}) = \langle 0, 2, 6\rangle, \mathbf{fx}\tau(\texttt{area}) = \langle 0, 4, 4\rangle$ The cost of the implementation is 81.80 . Figure 4.1 illustrates both the relative and absolute difference between the fixed-point and floating-point program with this wordlength. It is obtained by simulating all radius in the given domain at intervals of 0.0001. The horizontal line in the plots of relative error shows the maximum threshold of 0.01.

Case 2: Increasing the wordlengths of all the fixed-point variables to 12, increases the cost of implementation to 179.80. The new fixed-point types for the variables are: $\mathbf{fx}\tau(\texttt{mypi}) = \langle 0, 2, 10\rangle, \mathbf{fx}\tau(\texttt{radius}) = \langle 0, 1, 11\rangle, \mathbf{fx}\tau(\texttt{t}) = \langle 0, 2, 10\rangle, \mathbf{fx}\tau(\texttt{area}) = \langle 0, 4, 8\rangle$ but the relative and absolute difference between the fixed-point and floating-point program is reduced as illustrated in Figure 4.2. The correctness condition for accuracy is still not satisfied.

Case 3: Further increasing the wordlength to 16 decreases the error to satisfy the correctness criteria as illustrated in Figure 4.3 but the cost of implementation rises to 316.20. The fixed-point types for the variables are: $\mathbf{fx}\tau(\texttt{mypi}) = \langle 0, 2, 6\rangle$, $\mathbf{fx}\tau(\texttt{radius}) = \langle 0, 1, 7\rangle$, $\mathbf{fx}\tau(\texttt{t}) = \langle 0, 2, 6\rangle$, $\mathbf{fx}\tau(\texttt{area}) = \langle 0, 4, 4\rangle$.

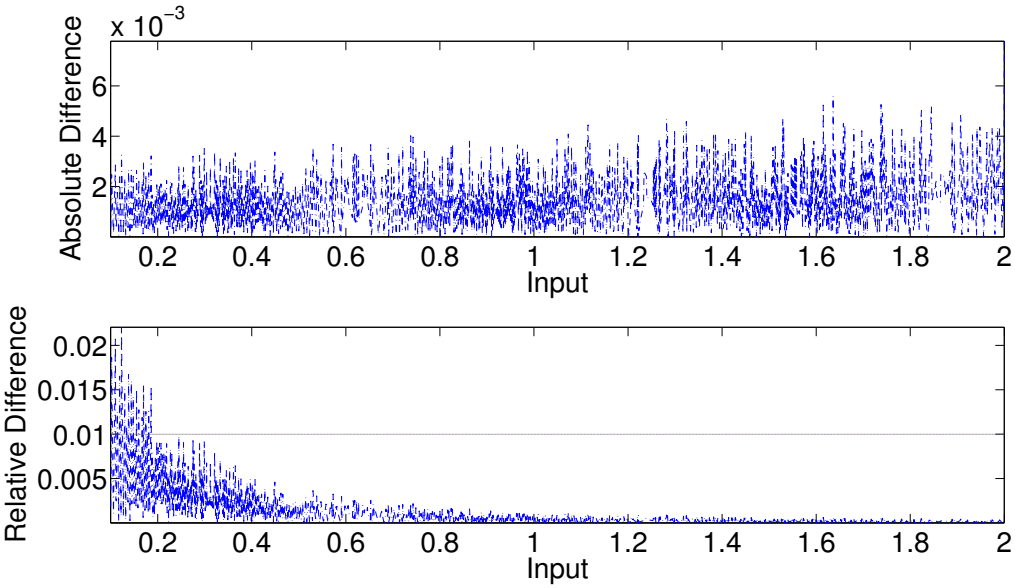


Figure 4.2: Wordlength 12

Manual synthesis of fixed-point program would require the developer to try simulations of this program with varying fixed-point types for different variables and identify the cases where the correctness condition is satisfied. The next step would be to find the fixed-point type that minimizes the cost from among all the types that satisfy the correctness condition. Such a manual trial and error process is time-consuming and error prone. Another dimension of the problem is finding the set of inputs for which the program needs to be simulated to check for the correctness condition. Since the input domain can be very large, it is not possible to simulate the program for all possible inputs. Intelligent choice of the inputs to use in simulating the program is important for a practical synthesis approach. In the following section, we present an automated approach to solve this problem.

4.2 SCIDUCTIVE Approach

A central idea behind our approach is to identify a small set of *interesting* inputs $S(X)$ from the input domain $Dom(X)$ such that the optimal implementation satisfying the correctness condition for the inputs in $S(X)$ will be optimal and correct for all inputs in the given input domain

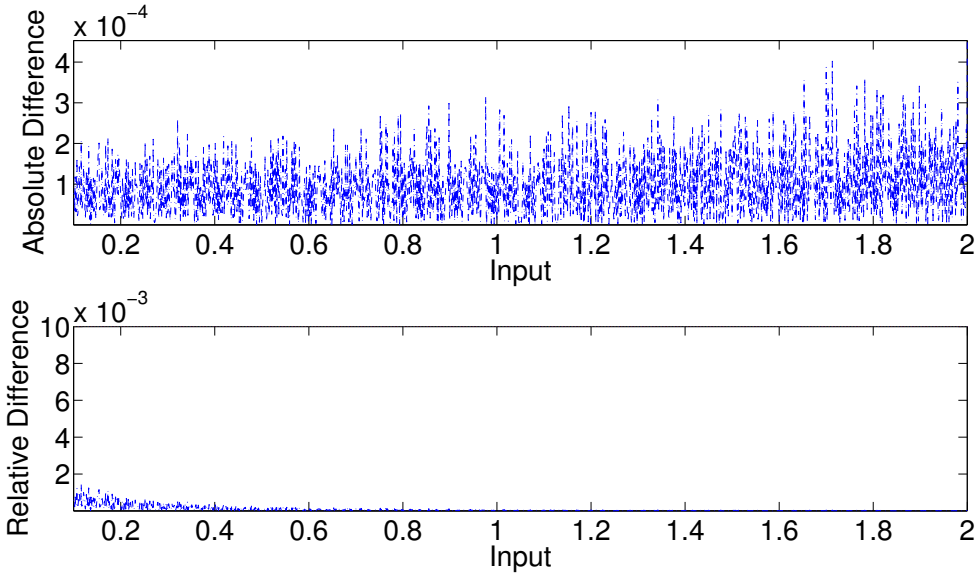


Figure 4.3: Wordlength 16

$Dom(X)$.[1] An optimization oracle $\mathcal{O}_V$ is used to discover the elements of $S(X)$. A different optimization oracle $\mathcal{O}_S$ is used to compute the minimum cost implementation for inputs of $S(X)$. If the oracles $\mathcal{O}_V$ and $\mathcal{O}_S$ find globally-optimal solutions, we show that our procedure will compute the minimum cost implementation for *all inputs* in $Dom(X)$.

The top-level synthesis algorithm is presented in Procedure 11. We assume that there is a (possibly very large) upper bound on wordlengths $\texttt{WL}_{max}$ beyond which it is non-optimal to use the fixed-point version. The algorithm starts with a randomly selected set of examples S^0 from the given input domain. Then, a fixed-point implementation that satisfies the accuracy condition for each of these inputs and is of minimal cost is synthesized using the routine `optimize` (which invokes $\mathcal{O}_S$). If no such implementation is found, the algorithm reports INFEASIBLE . Otherwise, the testing routine `getErr` checks whether the implementation fails the correctness condition for any input (using $\mathcal{O}_V$). If so, a set of inputs Bad^i on which the implementation violates the correctness condition are added to the set S^i used for synthesis, and the process is repeated. If the correctness condition is satisfied, the resulting fixed-point types are output.

In the rest of this section, we describe the main components of our approach in detail, including the theoretical result.

[1] In this section, we will sometimes simply write S or Dom for brevity.

Procedure 11 Overall Synthesis Algorithm: `syn`

Input: Floating-point program F_{fp},

Fixed-point program F_{fx} with fixed-point variables T,

Domain of inputs Dom, Error function Err,

maximum error threshold `maxError`, Cost Model `cost`,

maximum wordlengths $\texttt{WL}_{max}$

Output: Fixed-point type $\textbf{fx}\tau$ for variables T

or INFEASIBLE

S^0 = random sample from Dom, $Bad^0 = S^0$, $i = 0$

while $Bad^i \neq \emptyset$ **do**

$i = i + 1$

$S^i = S^{i-1} \cup Bad^{i-1}$

$\textbf{fx}\tau^i = \texttt{optimize}(F_{fp}, F_{fx}, Dom, Err, \texttt{maxError}, \texttt{cost}, \texttt{WL}_{max}, S^i)$

if $\textbf{fx}\tau^i = \perp$ **then**

return INFEASIBLE

end if

$Bad^i = \texttt{getErr}(F_{fp}, F_{fx}, \textbf{fx}\tau^i, Dom, Err, \texttt{maxError})$

end while

return $\textbf{fx}\tau^* = \textbf{fx}\tau^i$

4.2.1 Synthesizing Optimal Types for a Finite Input Set

The `optimize` function (see Procedure 12) is used to obtain optimum fixed-point types such that the fixed-point program with these types satisfies the correctness condition for a finite input set S and has minimal cost. First, the floating-point program F_{fl} is executed for all the inputs in the sample S and the range of each variable t_i as well as its `Signedness` is recorded by the functions `getRange` and `isSigned` respectively. Then, the integer wordlength `IWL` sufficient to represent the computed range is assigned to each variable t_i and the `Signedness` is 1 if the variable takes both positive and negative values, and 0 otherwise. If the fixed-point program with maximum wordlengths $\texttt{WL}_{max}$ fails the correctness condition, we conclude that the synthesis is not feasible and return $\perp$. If not, we search for the wordlength with minimum cost satisfying the correctness condition using our optimization oracle $\mathcal{O}_S$. The result is used to compute the fractional wordlengths, and the resulting fixed-point types are returned.

More precisely, $\mathcal{O}_S$ solves the following optimization problem over $\mathbf{fx}\tau$:

$$\texttt{Minimize cost}(\mathbf{fx}\tau)\ \texttt{s.t.}$$
$$\bigwedge_{x \in S} Err(F_{fx}(x, \mathbf{fx}\tau), F_{fl}(x)) \leq \texttt{maxError} \tag{4.1}$$

Let us reflect on the nature of the above optimization problem. It is a discrete optimization problem with a non-convex constraint space, a problem class that is hard to solve efficiently [34]. In spite of this, since floating-point and fixed-point programs can ultimately be encoded to Boolean satisfiability (SAT), due to the dramatic practical progress in SAT solving, one might be hopeful of an efficient SAT-based pseudo-Boolean optimization procedure to solve this problem (e.g.,see [33, 22]). However, we note two main factors going against such a SAT-based approach. First, the optimization is over wordlengths, rather than bits encoding variable values; thus, one must do an explicit case-split over the large space of possible wordlengths for each vector of variables X. Second, SAT solving is known to be notoriously difficult for reasoning about programs with arbitrary floating-point arithmetic operations involving non-linear arithmetic. Additionally, the size of the constraint that encodes correctness grows linearly in the number of input examples. While a SAT-based approach may work for programs with a small number of variables and no non-linear floating-point operators, for the general case, these factors effectively rule out such an approach.

Hence, we implement $\mathcal{O}_S$ using a greedy procedure `getMinCostWL` (see Procedure 13). The wordlength of each variable is increased or decreased by 1 independently to construct a set of candidate wordlength assignments $cand\texttt{WL}$. We select a set of valid candidate wordlength assignments $valcand\texttt{WL}$ from $cand\texttt{WL}$ which satisfy the correctness condition. From these, the least-cost assignment is then selected as our next greedy choice. This process is continued till no further reduction in cost is possible and we have reached a local minimum.

This method can be used for any floating-point program because it relies only on executing the program and does not require any explicit modeling of the operations. However, it only guarantees finding a wordlength with a locally-minimum cost and not globally-minimum cost.

Procedure 12 Optimal Fixed-Point Types Synthesis: `optimize`

Input: Floating-point program F_{fp},
Fixed-point program F_{fx} with fixed-point variables T, Domain of inputs Dom, Error function Err,
maximum error threshold `maxError`, Cost Model `cost`,
max wordlengths $\texttt{WL}_{max}$, Input S

Output: Optimal wordlengths `WL` for inputs S or $\perp$

for all fixed-point variable t_i in F_{fx} **do**
$\texttt{IWL}(t_i) = \lceil \log(\texttt{getRange}(t_i, F_{fl}, S) + 1) \rceil$
$\texttt{Signedness}(t_i) = \texttt{isSigned}(t_i, F_{fl}, S)$
end for
if $\texttt{WL}_{max} < \texttt{IWL}$ **then**
return $\perp$
end if
$\text{fx}\tau = \langle \texttt{Signedness}, \texttt{IWL}, \texttt{WL}_{max} - \texttt{IWL} \rangle$
if $Err(F_{fp}(x), F_{fx}(x, \text{fx}\tau)) > \texttt{maxError}$ **then**
return $\perp$
end if
$\texttt{WL} = \texttt{getMinCostWL}(F_{fp}, F_{fx}, Dom, Err, \texttt{maxError}, \texttt{cost}, \texttt{WL}_{max}, S^i, \texttt{IWL}, \texttt{Signedness})$
return $\text{fx}\tau = \langle \texttt{Signedness}, \texttt{IWL}, \texttt{WL} - \texttt{IWL} \rangle$

Procedure 13 `getMinCostWL`

Input: Floating-point program F_{fp},

Fixed-point program F_{fx} with fixed-point variables T,

Domain of inputs Dom, Error function Err,

maximum error threshold $maxErr$, Cost Model `cost`,

max wordlengths $\texttt{WL}_{max}$, Input S

Output: Optimal wordlengths `WL`

$valcand\texttt{WL} = \{\texttt{WL}_{max}\}$

while $valcand\texttt{WL}$ is not empty **do**

$\texttt{WL} = \underset{vc\texttt{WL} \in valcand\texttt{WL}}{\operatorname{argmin}} \; cost(vc\texttt{WL})$

$\text{fx}\tau = \langle \texttt{Signedness}, \texttt{IWL}, \texttt{WL} - \texttt{IWL} \rangle$

$cand\texttt{WL} = \emptyset$, $valcand\texttt{WL} = \emptyset$

for all fixed-point variable t_i in F_{fx} **do**

$\texttt{WL}^{i-}(j) = \texttt{WL}(j) \; \forall j \neq i, \texttt{WL}^{i-}(i) = \texttt{WL}(i) - 1$

$\texttt{WL}^{i+}(j) = \texttt{WL}(j) \; \forall j \neq i, \texttt{WL}^{i+}(i) = \texttt{WL}(i) + 1$

$cand\texttt{WL} = cand\texttt{WL} \cup \{\texttt{WL}^{i-}, \texttt{WL}^{i+}\}$

end for

for all $cand$ in $cand\texttt{WL}$ **do**

$cand\text{fx}\tau = \langle \texttt{Signedness}, \texttt{IWL}, cand\texttt{WL} - \texttt{IWL} \rangle$

if $Err(F_{fp}(x), F_{fx}(x, cand)) \leq maxErr \; \forall x \in S$

and $\texttt{cost}(cand\text{fx}\tau) < \texttt{cost}(\text{fx}\tau)$ **then**

$valcand\texttt{WL} = valcand\texttt{WL} \cup \{cand\}$

end if

end for

end while

return $\text{fx}\tau$

4.2.2 Verifying a Candidate Fixed-Point Program

In order to verify that the fixed-point program $F_{fx}(X, \mathbf{fx}\tau)$ satisfies the correctness condition, we need to check if the following logical formula is satisfiable.

$$\exists X \in Dom(X) \;\; Err(F_{fx}(X, \mathbf{fx}\tau), F_{fp}(X)) > \texttt{maxError} \tag{4.2}$$

If the formula is unsatisfiable, there is no input on which the fixed-point program violates the correctness condition.

Once again, in principle one can use a SAT-based approach to solve the above problem. However, for arbitrary floating-point and fixed-point arithmetic operations, it is extremely difficult to solve such a problem in practice with current SAT solvers. Instead, we use a novel optimization-based approach to verify the candidate fixed-point program. An optimization oracle $\mathcal{O}_V$ is used to maximize the error function $Err(F_{fx}(X, \mathbf{fx}\tau), F_{fp}(X))$ over the domain $Dom(X)$. If there is no input $X \in Dom(X)$ for which the error function exceeds `maxError`, the fixed-point program is correct and we terminate. Otherwise, we obtain an example input on which the fixed-point program violates the correctness condition. Multiple inputs can also be generated where they exist.

In practice, with the current state-of-the-art optimization routines, it is difficult to implement $\mathcal{O}_V$ to find a global optimum. Instead, we use a numerical optimization routine based on the Nelder-Mead method [101] which can handle arbitrary non-linear functions and generates local optima. Procedure 14 defines `getErr` which invokes the Nelder-Mead routine (indicated by "argmaxlocal"). This routine requires one to supply a starting value of X, which we generate randomly. To find multiple inputs, we invoke the routine from from different random initial points and record all example inputs on which the fixed-point program violates the correctness condition. Since a global optimum is not guaranteed, we repeat this search `maxAttempts` times before declaring that the fixed-point program is correct. Using an optimization oracle that finds a global optimum would guarantee the soundness of our approach (see Sec. 4.2.4), but Procedure 14 works well in practice. Importantly, this approach can be used for any floating-point program.

Procedure 14 Verification Routine `getErr`

Input: Floating-point program F_{fp},
Fixed-point program F_{fx}, Fixed-point type $\mathbf{fx}\tau$,
Domain of inputs Dom, Error function Err,
maximum error threshold `maxError`

Output: Inputs Bad on which F_{fx} violates correctness condition

$Bad = \emptyset$
while $i \leq$ `maxAttempts` **do**
$i = i + 1$, $X_0 =$ random sample from Dom
$X_{cand} = \underset{X}{\text{argmaxlocal}}(Err(F_{fp}(X), F_{fx}(X, \mathbf{fx}\tau)), X_0)$
if $Err(F_{fp}(X_{cand}), F_{fx}(X_{cand}, \mathbf{fx}\tau)) >$ `maxError` and $X \in Dom$ **then**
$Bad = Bad \cup \{X\}$
end if
end while

4.2.3 Illustration on Running Example

We illustrate the synthesis approach presented in Section 4.2 using the running example. Our algorithm took 4 iterations. The (cumulative) number of samples used in iteration $2, 3, 4$ after adding examples discovered by `getErr` procedure was $18, 22$ and 34 respectively. To evaluate our approach, we exhaustively simulated the generated fixed-point program on the given domain ($0.1 \leq$ `radius` < 2) at intervals of 0.0001. The results after iteration $2, 3, 4$ are presented in Figure 4.5 and Figure 4.6. As a point of comparison, we also show the result of synthesizing a fixed-point program using the `optimize` routine with 100 inputs (3 times as many as our approach) selected uniformly at random in Figure 4.4. The horizontal line in the plots denotes the maximum error threshold of 0.01 on the relative difference error function. The cost of the fixed-point program synthesized with random sampling is 89.65, and the fixed-point types of the variables are $\mathbf{fx}\tau(\texttt{mypi}) = \langle 0, 2, 3\rangle$, $\mathbf{fx}\tau(\texttt{radius}) = \langle 0, 1, 8\rangle$, $\mathbf{fx}\tau(\texttt{t}) = \langle 0, 2, 10\rangle$ and $\mathbf{fx}\tau(\texttt{area}) = \langle 0, 4, 8\rangle$. Notice, however, that it is incorrect for a large number of inputs. In contrast, the cost of the implementation produced using our technique is 104.65, and the fixed-point types of the variables are $\mathbf{fx}\tau(\texttt{mypi}) = \langle 0, 2, 3\rangle$, $\mathbf{fx}\tau(\texttt{radius}) = \langle 0, 1, 9\rangle$, $\mathbf{fx}\tau(\texttt{t}) = \langle 0, 2, 11\rangle$ and $\mathbf{fx}\tau(\texttt{area}) = \langle 0, 4, 10\rangle$. All outputs produced by our technique are within the error threshold.

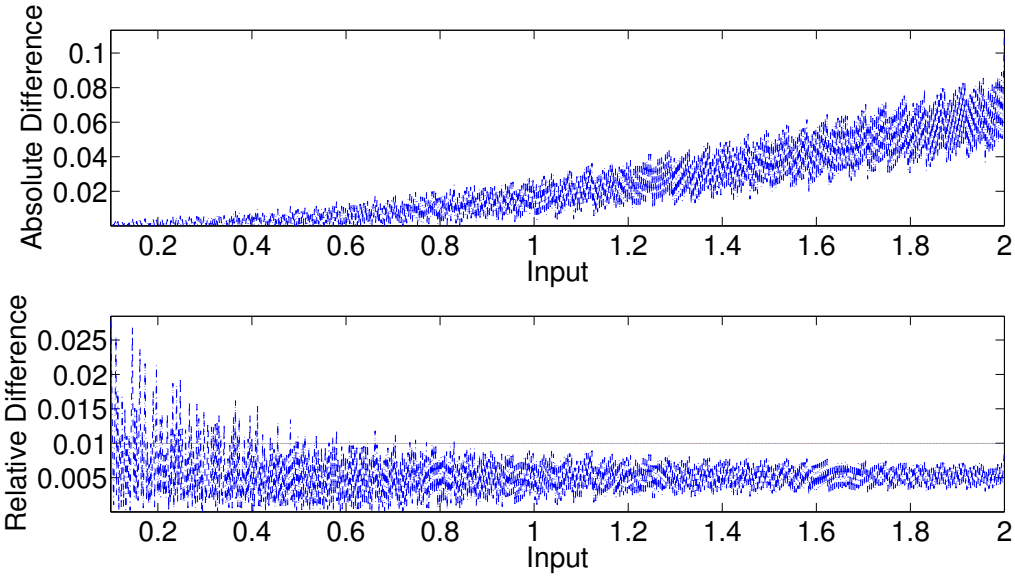


Figure 4.4: Running Example Using Random Inputs. The horizontal line indicates the error threshold

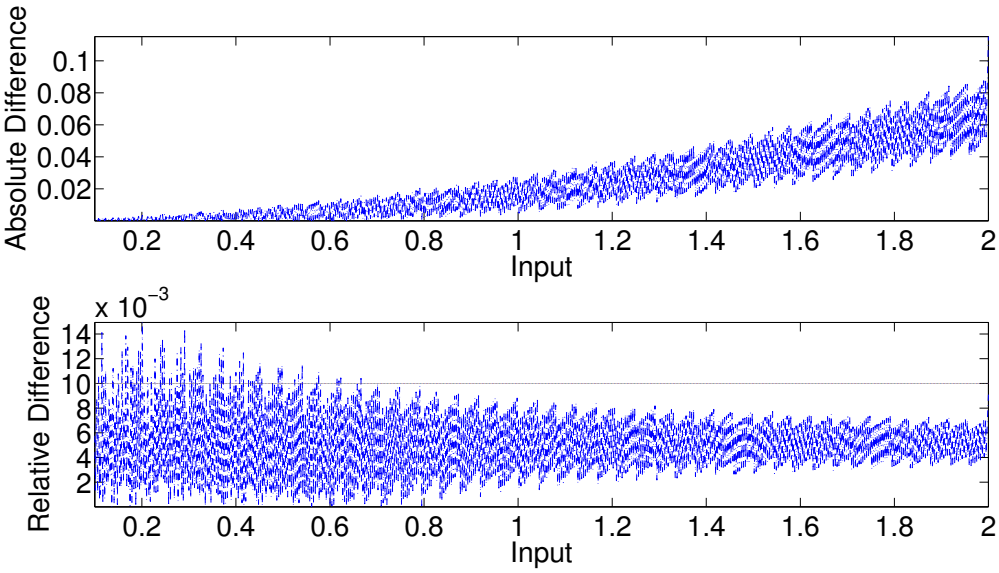
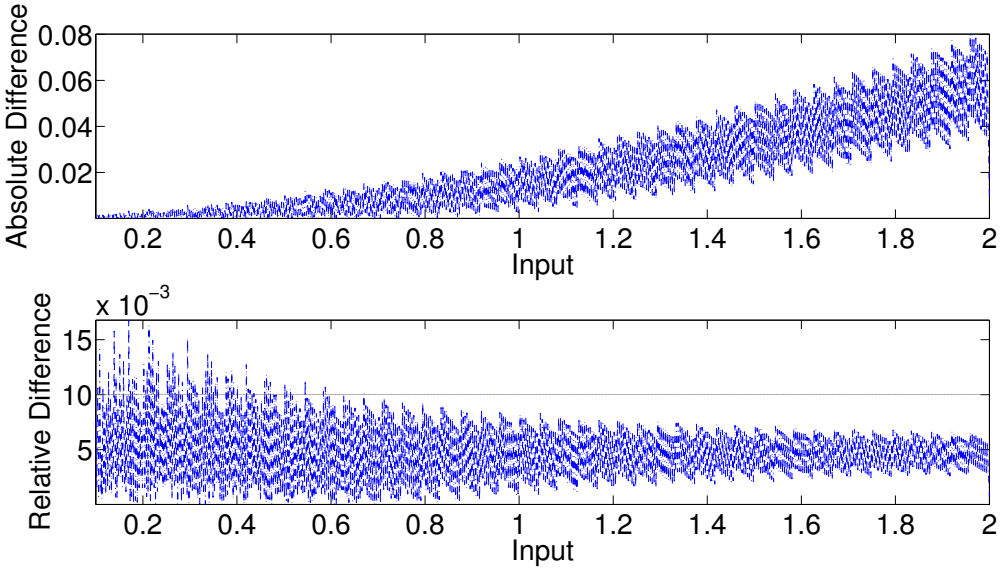


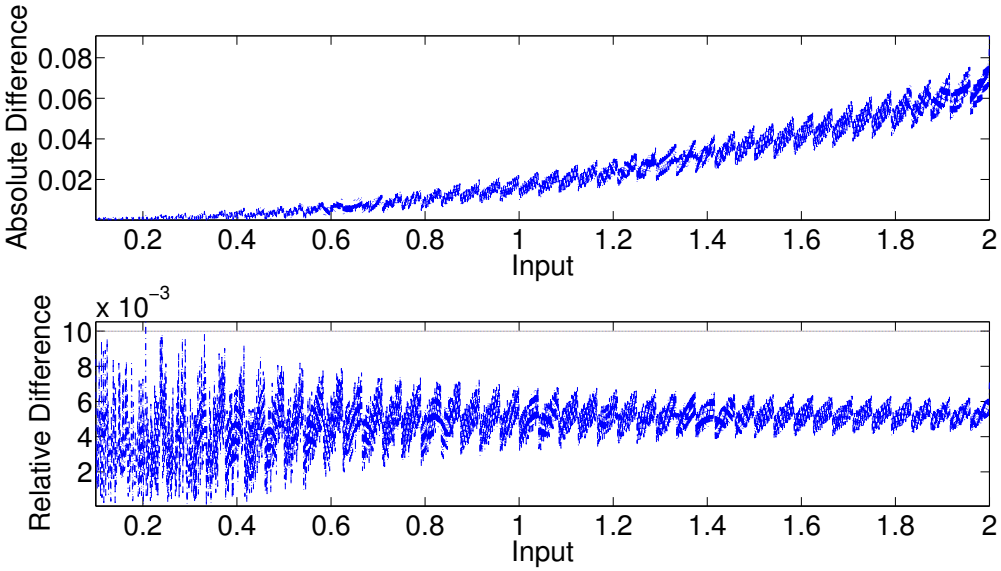
Figure 4.5: Our Approach on Running Example: Iteration 2

4.2.4 Theoretical Results

We now present the soundness guarantee of our SCIDUCTION based synthesis technique. The following theorem summarizes the correctness and optimality guarantees of our approach.



(a) Iteration 3 of Procedure `syn`



(b) Iteration 4 of Procedure `syn`

Figure 4.6: Our Approach on Running Example: Iteration 3 and 4

Theorem 5. *The synthesis procedure presented in Procedure 11 is guaranteed to synthesize the fixed-point program which is of minimal cost and satisfies the correctness condition for accuracy if optimization oracles $\mathcal{O}_S$ and $\mathcal{O}_V$ find globally-optimal solutions (when they exist).*

Proof. We first prove correctness and then optimality of the obtained solution. Consider Equation 4.2:

$$\exists X \in Dom(X) \;\; Err(F_{fx}(X, \mathbf{fx}\tau), F_{fp}(X)) > \texttt{maxError}$$

If $\mathcal{O}_V$ finds globally-optimal solutions, it will find the $X \in Dom(X)$ that maximizes $Err(F_{fx}(X, \mathbf{fx}\tau), F_{fp}(X))$, and hence determine where or not Equation 4.2 is satisfiable. Thus, the correctness condition for accuracy is satisfied.

Next, let $\mathbf{fx}\tau^*$ ($\neq \perp$) be the fixed-point type returned by Procedure 11. Let us assume that there exists a fixed-point type $\mathbf{fx}\tau'$ with a cost lower than $\mathbf{fx}\tau^*$ which also satisfies the correctness condition:

$$\begin{aligned} & \forall X \in Dom(X) \;\; Err(F_{fx}(X, \mathbf{fx}\tau'), F_{fp}(X)) \leq \texttt{maxError} \\ \wedge \quad & \texttt{cost}(\mathbf{fx}\tau') < \texttt{cost}(\mathbf{fx}\tau^*) \end{aligned}$$

Hence, for any $D \subseteq Dom(X)$,

$$\begin{aligned} & \forall X \in D \;\; Err(F_{fx}(X, \mathbf{fx}\tau'), F_{fp}(X)) \leq \texttt{maxError} \\ \wedge \quad & \texttt{cost}(\mathbf{fx}\tau') < \texttt{cost}(\mathbf{fx}\tau^*) \end{aligned}$$

But $\mathbf{fx}\tau^*$ is the solution generated by applying $\mathcal{O}_S$ to the optimization problem of Equation 4.1:

$$\begin{aligned} & \texttt{Minimize cost}(\mathbf{fx}\tau) \texttt{ s.t.} \\ & \bigwedge_{x \in S} Err(F_{fx}(x, \mathbf{fx}\tau), F_{fl}(x)) \leq \texttt{maxError} \end{aligned} \tag{4.3}$$

Since $\mathcal{O}_S$ is guaranteed to generate globally-optimal solutions, setting $D = S$, we obtain a contradiction. Hence, there exists no fixed-point type $\mathbf{fx}\tau'$ with a cost lower than $\mathbf{fx}\tau^*$ which also satisfies the correctness condition. Hence, $\mathbf{fx}\tau^*$ is the optimal correct solution. □

As noted earlier, it is difficult to implement ideal $\mathcal{O}_S$ and $\mathcal{O}_V$ (that find global optima) with current SAT and optimization methods for arbitrary floating-point programs. Nonetheless, our experience with heuristic methods that find local optima has been very good. Also, improvements

in optimization/SAT methods can directly be leveraged with our inductive synthesis approach. In contrast, the current techniques for synthesizing fixed-point versions of floating-point programs perform heuristic optimization over a randomly selected set of inputs. Such techniques do not provide any correctness guarantees and the number of inputs needed could be much larger, as illustrated in Section 4.2.3. Our approach systematically discovers a small number of example inputs such that the optimal fixed-point program for this set yields that for the entire input domain.

4.3 Experiments

We present case studies from DSP and control systems to illustrate the utility of the presented synthesis approach. Our technique was implemented in Matlab, and Nelder-Mead implementation available in Matlab as `fminsearch` function was used for numerical optimization. We use the Constantinides et al [38] cost model.

4.3.1 Infinite Impulse Response (IIR) Filter

The first case study is an IIR filter which is used in digital signal processing applications. It is a first-order direct form-II IIR filter with the schematic shown in Figure 4.7. The constants are $a_1 = -0.5$, $b0 = 0.9$ and $b1 = 0.9$. The fixed-point variables are identified in the schematic. We use our synthesis technique to discover the appropriate fixed-point types of these variables. The input domain used in synthesis is $-2 < input < 2$. The correctness condition for accuracy is to ensure that the relative error between the floating-point and fixed-point program is less than 0.1.

In order to test the correctness of our implementation, we feed a common input signal to both the IIR filter implementations: floating-point version and the fixed-point version obtained by our synthesis technique. The input signal is a linear chirp from 0 to $\frac{Fs}{2}$ Hz in 1 second.

$$input = (1 - 2^{-15}) \times sin(\pi \times \frac{Fs}{2} \times t^2)$$

where $Fs = 256$ and $t = 0$ to $1 - \frac{1}{Fs}$ and is sampled at intervals of $\frac{1}{Fs}$. Figure 4.8 shows the input, outputs of both implementations and the relative error between the two outputs. We observe that the implementation satisfies the correctness condition and the relative error remains below 0.1 throughout the simulation.

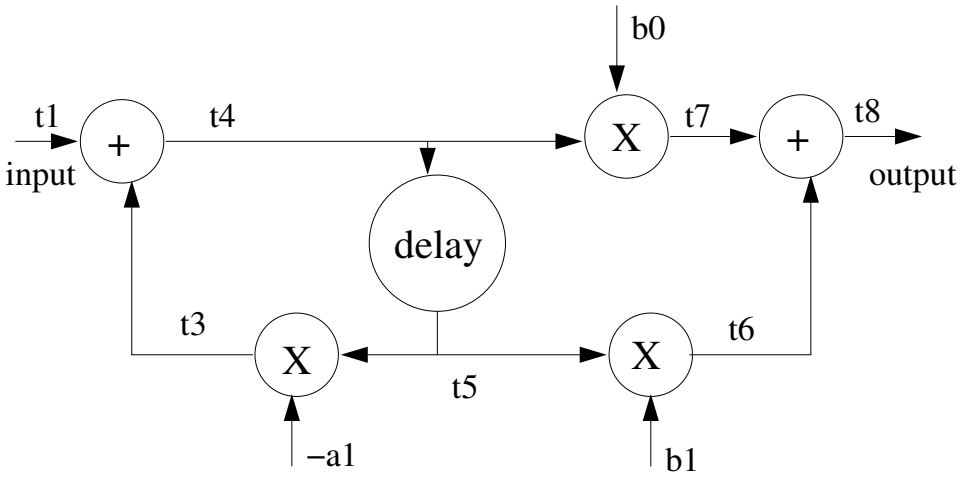


Figure 4.7: IIR Filter Schematic

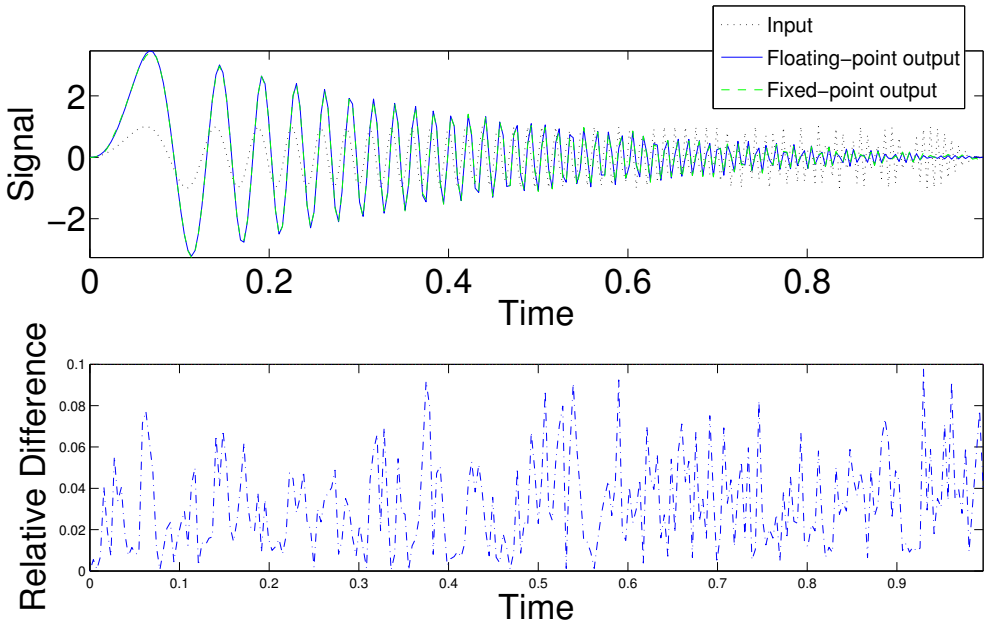


Figure 4.8: IIR Filter Using Floating-point and Fixed-point. In the top plot, the floating-point and fixed-point outputs are virtually superimposed on each other.

4.3.2 Finite Impulse Response (FIR) Filter

The second case study is a low pass FIR filter of order 4 with tap coefficients $0.0346, 0.2405, 0.4499, 0.2405$ and 0.0346. The input domain, correctness condition and input signal to test the floating-point implementation and synthesized fixed-point program are same as the previous case

study. Figure 4.9 shows the input, outputs of both implementations and the relative error between the two outputs. We observe that the implementation satisfies the correctness condition and the relative error remains below 0.1 throughout the simulation.

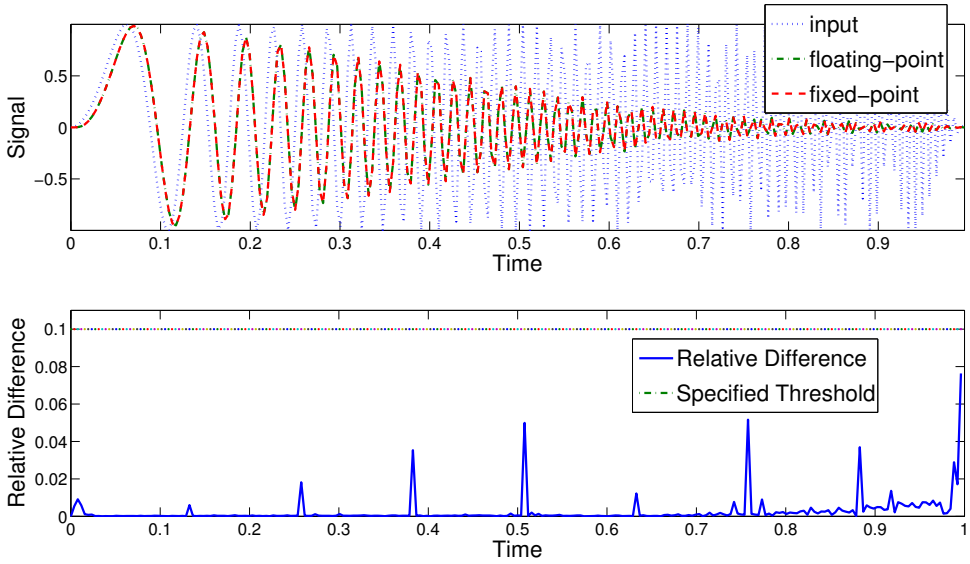


Figure 4.9: FIR Filter Using Floating-point and Fixed-point. The floating-point/fixed-point outputs are virtually superimposed on each other.

4.3.3 Field Controlled DC Motor

The next case study is a field controlled DC Motor. It is a classic non-linear control example from Khalil [67]. The example used here is an adaptation of Khalil's example presented in [129]. The system dynamics is described by the following ordinary differential equations.

$$\begin{aligned} \dot{v}_f &= R_f i_f + L_f \dot{i}_f \\ \dot{v}_a &= c_1 i_f \omega + L_a \dot{i}_a + R_a i_a \\ J\dot{\omega} &= c_2 i_f i_a - c_3 \omega \end{aligned}$$

The first equation is for the field circuit with v_f, i_f, R_f, L_f being its voltage, current, resistance, and inductance. The variables v_a, i_a, R_a, L_a are the corresponding voltage, current, resistance, and inductance of the armature circuit described by the second equation. The third equation is a torque

equation for the shaft, with J as the rotor inertia and c_3 as a damping coefficient. The term $c_1 i_f \omega$ is the back electromotive force induced in the armature circuit, and $c_2 i_f i_a$ is the torque produced by the interaction of the armature current with the field circuit flux. In the field controlled DC motor, field voltage v_f is the control input while v_a is held constant. The purpose of the control is to drive the system to the desired set point for the angular velocity ω.

We can now rewrite the system dynamics in the following normal form where $a = \frac{R_f}{L_f}$, $u = \frac{v_f}{L_f}$, $b = \frac{R_a}{L_a}$, $\rho = \frac{v_a}{L_a}$, $c = \frac{c_1}{L_a}$, $\theta = \frac{c_2}{J}$, $d = \frac{c_3}{J}$.

$$\begin{aligned} \dot{i}_f &= -a i_f + u \\ \dot{i}_a &= -b i_a + \rho - c i_f \omega \\ \dot{\omega} &= \theta i_f i_a - d\omega \end{aligned}$$

We assume no damping, that is, $c_3 = 0$ and set all the other constants a, b, c, θ, ρ to 1 [129]. The state feedback law to control the system is given by

$$u = \frac{\theta(a+b) i_f i_a + \theta \rho i_f - c\theta i_f^2 \omega}{\theta i_a}$$

The corresponding floating-point code is shown below.

Input: $\mathtt{i_f}, \mathtt{i_a}, \omega, \theta, \rho, \mathtt{c}, \epsilon$

Output: $\mathtt{u}$

$\mathtt{t1} = \theta \times \mathtt{i_a}$; $\mathtt{t2} = \epsilon + \mathtt{t1}$; $\mathtt{t3} = 1/\mathtt{t2}$; $\mathtt{t31} = \mathtt{a} + \mathtt{b}$;

$\mathtt{t32} = \mathtt{i_f} \times \mathtt{i_a}$; $\mathtt{t33} = \mathtt{t31} \times \mathtt{t32}$; $\mathtt{t4} = \theta \times \mathtt{t33}$; $\mathtt{t41} = \rho \times \mathtt{i_f}$;

$\mathtt{t5} = \theta \times \mathtt{t41}$; $\mathtt{t6} = \mathtt{i_f} \times \mathtt{i_f}$; $\mathtt{t61} = \mathtt{t6} \times \omega$; $\mathtt{t62} = \mathtt{t61} \times \theta$;

$\mathtt{t7} = \mathtt{c} \times \mathtt{t62}$; $\mathtt{t8} = \mathtt{t4} + \mathtt{t5}$; $\mathtt{t9} = \mathtt{t8} - \mathtt{t7}$; $\mathtt{u} = \mathtt{t3} \times \mathtt{t9}$;

return $\mathtt{u}$

The system is initialized with field current $i_f = 1$, armature current $i_a = 1$ and angular velocity $\omega = 1$.

The computed control law can be mathematically shown to be correct by designers who are more comfortable in reasoning with real arithmetic but not with finite precision arithmetic. Its implementation using floating-point computation also closely mimics the arithmetic in reals but the control algorithms are often implemented using fixed-point computation on embedded platforms. We use our synthesis technique to automatically derive a low cost fixed-point implementation of the control law computing u. The input domain is $0 \leq i_a, i_f, \omega \leq 1.5$. The correctness condition

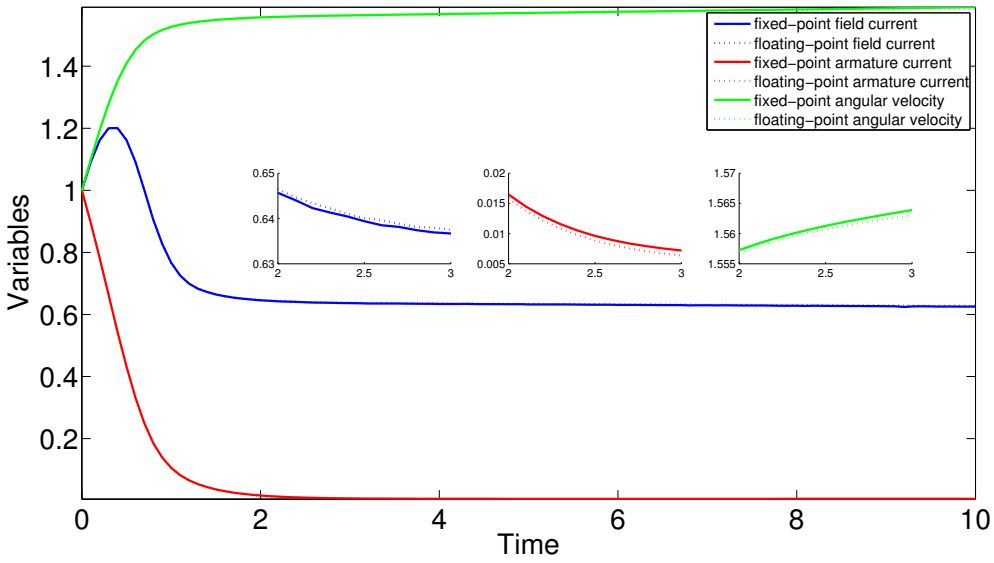
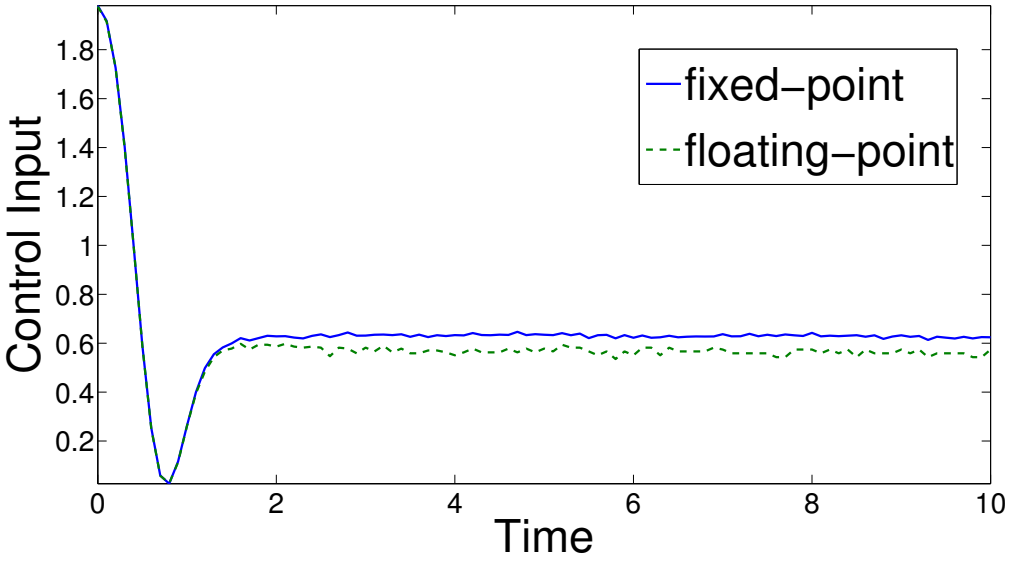


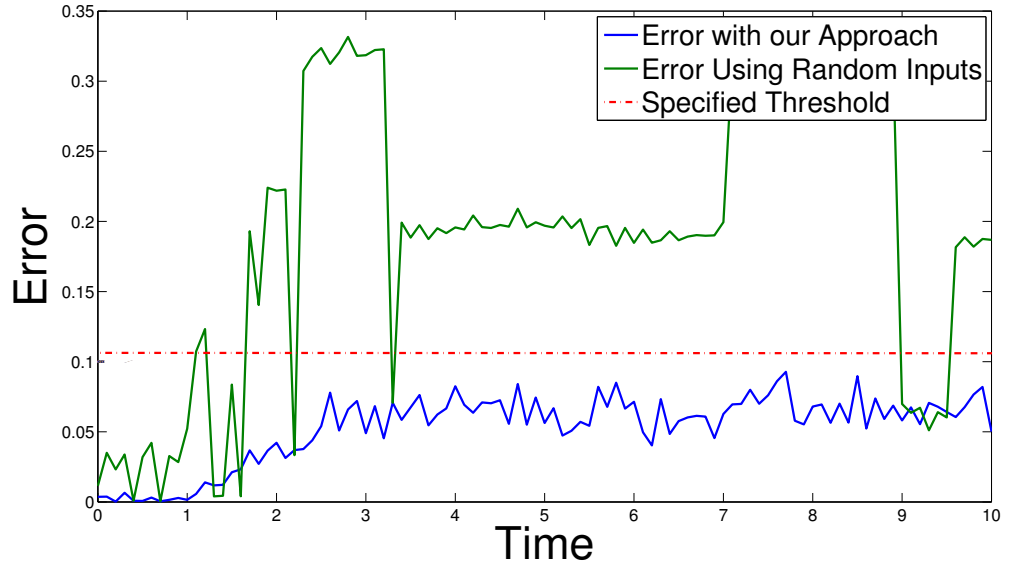
Figure 4.10: DC Motor Using Floating-point and Fixed-point Controller with zoomed-in view for 2 to 3 seconds.

for accuracy is that the absolute difference between the u computed by fixed-point program and the floating-point program is less than 0.1.

Figure 4.10 shows the simulation of the system using the fixed-point implementation of the controller and the floating-point implementation. This end-to-end simulation shows that fixed-point program generated by our technique can be used to control the system as effectively as the floating-point program. This illustrates the practical utility of our technique. Figure 4.11(b) plots the difference between the control input computed by the fixed-point program and the floating-point program. It shows that the fixed-point types synthesized using our approach satisfy the correctness condition, and the difference between the control input computed by the fixed-point and floating-point program is within the specified maximum error threshold of 0.1. The number of inputs needed in our approach was 127. In contrast, the fixed-point types found using 635(5X our approach) randomly selected inputs violate the correctness condition for a large number of inputs.



(a) Computed Control Input



(b) Error

Figure 4.11: Error in Control Input Using Fixed-point and Floating-point Program

4.3.4 Two-Wheeled Welding Mobile Robot

The next case study is a nonlinear controller for a two-wheeled welding mobile robot (WMR) [20]. The robot consists of two wheels and a robotic arm. The wheels can roll and there is no slipping. (x, y) represents the Cartesian coordinate of the WMR's center point and ϕ is the heading angle of the WMR. v and ω are the straight and angular velocities of the WMR at its center point. The welding point coordinates (x_w, y_w) and the heading angle ϕ_w can be calculated from the WMR's center point:

$$\begin{aligned} x_w &= x - l\sin\phi \\ y_w &= y + l\cos\phi \\ \phi_w &= \phi \end{aligned}$$

So, the equation of motion for the welding point is as follows:

$$\begin{aligned} \dot{x}_w &= v\cos\phi - l\omega\cos\phi - \dot{l}\sin\phi \\ \dot{y}_w &= v\sin\phi - l\omega\sin\phi + \dot{l}\cos\phi \\ \dot{\phi}_w &= \omega \end{aligned}$$

The objective of the WMR controller is to ensure that the robot tracks a reference point R. The reference point R moving with a constant velocity of v_r on the reference path has coordinates (x_r, y_r) and the heading angle ϕ_r. The tracking error is the difference between the location of the robot and the reference point.

$$\begin{bmatrix} e_1 \\ e_2 \\ e_3 \end{bmatrix} = \begin{bmatrix} \cos\phi & \sin\phi & 0 \\ -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_r - x_w \\ y_r - y_w \\ \phi_r - \phi_w \end{bmatrix}$$

The two control parameters in the model are v and ω. In order to ensure that the error quickly converges to 0, a nonlinear controller based on Lyapunov stability is as follows:

$$\begin{aligned} v &= l(\omega_r + k_2 e_2 v_r + k_3\sin e_3) + v_r\cos e_3 + k_1 e_1 \\ \omega &= \omega_r + k_2 e_2 v_r + k_3\sin e_3 \end{aligned}$$

where k_1, k_2 and k_3 are positive constants. Table 4.1 provides the numerical values of constants and initial values of the state variables from Bui et al [20]. All lengths are in meters, angle in radians and time in seconds.

Table 4.1: Numerical and Constant Values

Parameters	Values	Parameters	Values
k_1	4.2	l	0.15
k_2	5000	$\dot{l}$	0
k_3	1	v_r	$7.5e-3$
x_r	0.280	x_w	0.270
y_r	0.400	y_w	0.390
ϕ	0	ϕ_w	15

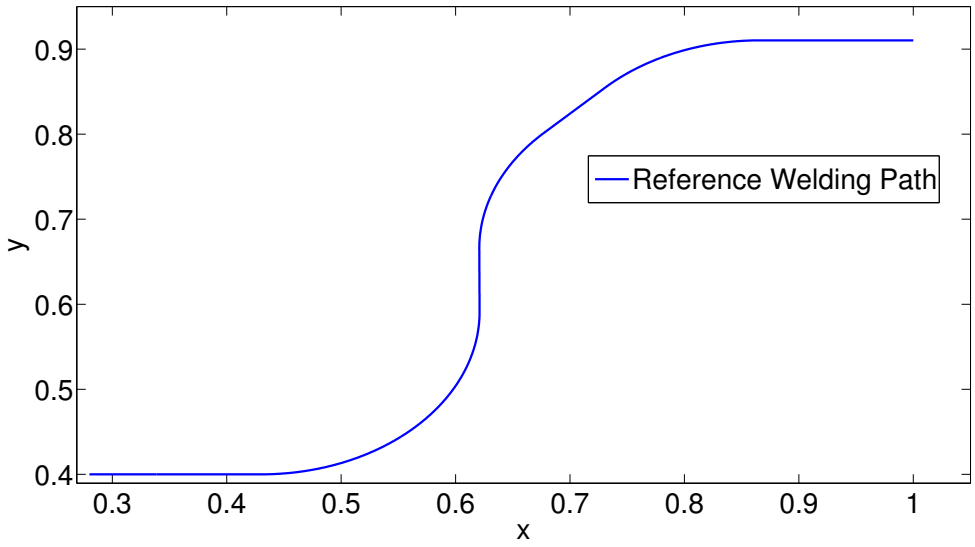


Figure 4.12: Reference Welding Line

We use our synthesis technique to automatically synthesize fixed-point program computing both control inputs: v and ω. The error function used for v is the relative difference

$$\frac{v_{floating-point} - v_{fixed-point}}{v_{floating-point}}$$

and the error function used for ω is the moderated relative difference

$$\frac{\omega_{floating-point} - \omega_{fixed-point}}{\omega_{floating-point} + \delta}$$

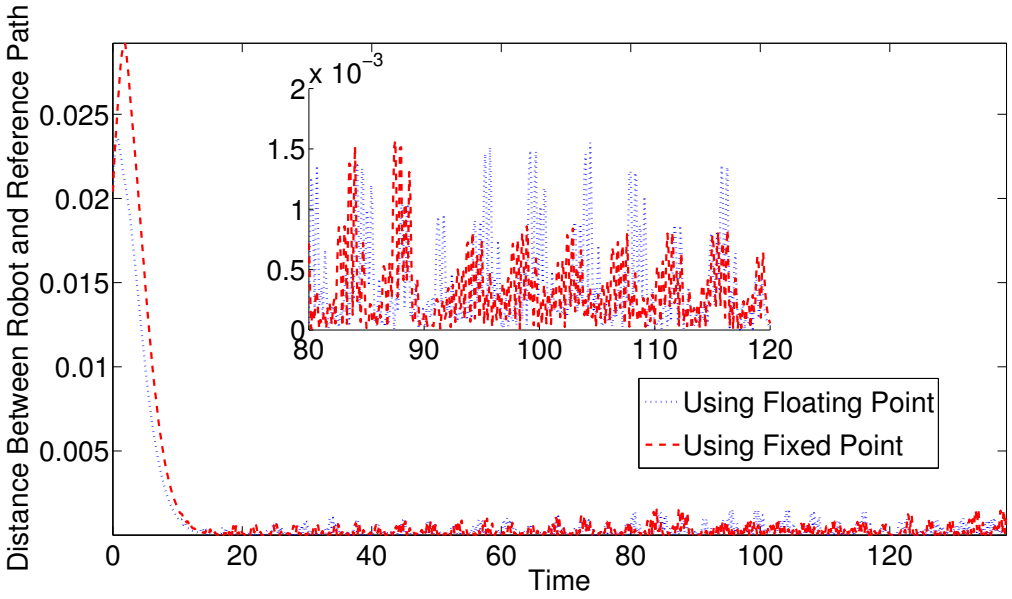


Figure 4.13: Distance of WMR from Reference Line with zoomed-in view for 80 to 120 seconds.

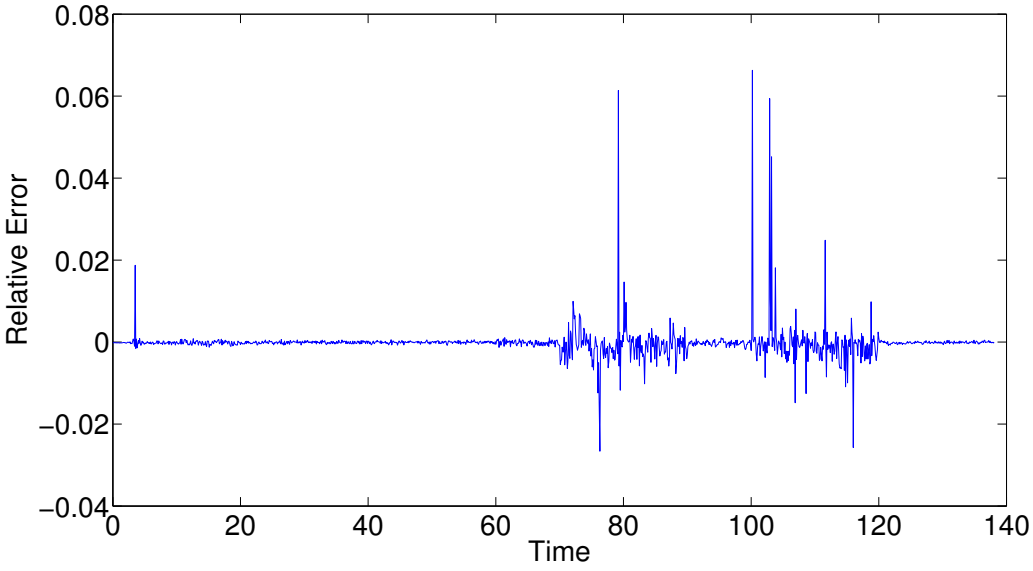


Figure 4.14: Error in computing v

where $\delta = 0.001$. The moderated relative difference is useful here since $\omega_{floating-point}$ can be 0. We require that the difference values for both controllers are less than 0.1. Figure 4.12 shows the reference line for welding and Figure 4.13 shows the distance of the WMR from the reference line

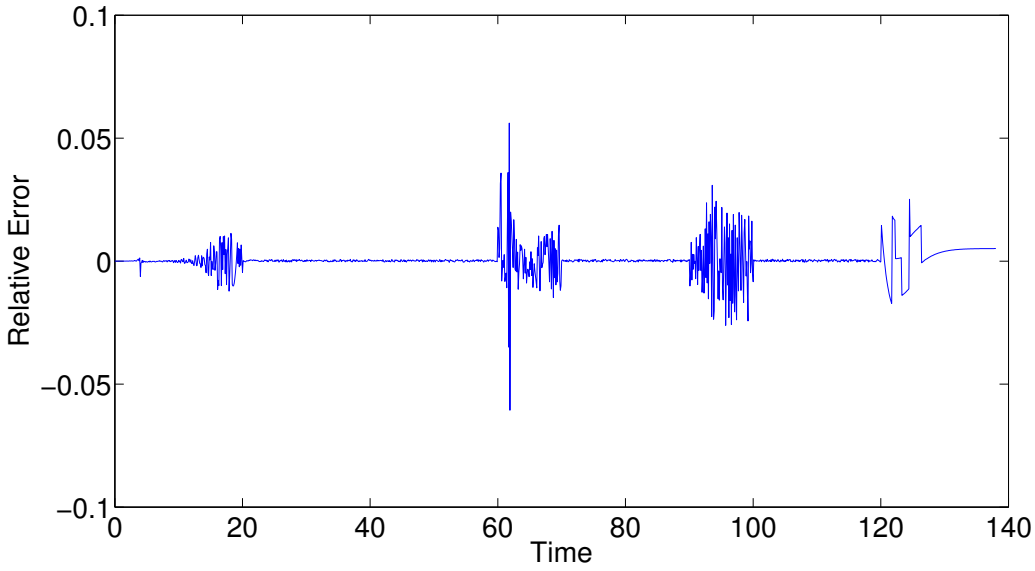


Figure 4.15: Error in computing ω

as a function of time for both cases: firstly, when the controller is implemented as a floating-point program and secondly, when the controller is implemented as a fixed-point program synthesized using our technique. The robot starts a little away from the reference line but quickly starts tracking the line in both cases. Figure 4.14 and Figure 4.15 show the error between the floating-point controller and fixed-point controller for both control inputs: v and ω, respectively.

Table 4.2 summarizes the performance of our technique in the four case-studies.

Table 4.2: Performance

Case-study	Runtime (seconds)	# of Iterations of `syn`
IIR Filter	268	5
FIR Filter	379	4
DC Motor u	4436	8
WMR v	2218	7
ω	1720	4

4.4 Conclusion

In this chapter, we presented a SCIDUCTION based novel approach to automated synthesis of fixed-point program from floating-point program by discovering the fixed-point types of the variables. The program is synthesized to satisfy the provided correctness condition for accuracy and to have optimal cost with respect to the provided cost model. We illustrated our approach on a set of case studies from digital signal processing and control systems.

Part II

Synthesis of Switching Logic

Chapter 5

Background

In this chapter, we present relevant background on cyber-physical systems that will be helpful in explaining our technique for automated synthesis of switching logic. Cyber-physical systems [32, 4, 81] are dynamical systems with interacting continuous and discrete dynamics. This mixed nature of the dynamics of these hybrid systems make their synthesis and analysis challenging. While continuous dynamics of a hybrid system is usually modeled using differential equations, discrete dynamics is modeled using finite automata. The model of the continuous dynamics is also often referred to as the *plant* and the finite automata is often called the *switching logic*. Hybrid systems arise in modeling a number of applications in different domains such as electronic design and automation, analog and mixed signal circuits, real-time software, robotics and automation, mechatronics, aeronautics, air and ground transportation systems, and process control.

As cyber-physical systems (CPS) are increasingly deployed in transportation, health-care, and other societal-scale applications, there is a pressing need for automated tool support to ensure dependability while enabling designers to meet shortening time-to-market constraints. Model-based design tools enable designers to work at a high level of abstraction, but there is still a need to assist the designer in creating *correct* and *efficient* systems.

A holy grail for the design of cyber-physical systems is to automatically synthesize models from safety and performance specifications. In its most general form, automated synthesis is very difficult to achieve, in part because synthesis often involves human insight and intuition, and in

part because of system complexity – the tight integration of complex continuous dynamics with discrete switching behavior can be tricky to get correct. Nevertheless, in some contexts, it may be possible for automated tools to complete partial designs generated by a human designer, thus enabling the designer to efficiently explore the space of design choices whilst ensuring that the synthesized system remains safe.

5.1 Formalism and Notations

Just as purely dynamical systems are often modeled using ordinary differential equations, hybrid systems are formally modeled using hybrid automata [81]. Hybrid automata combine the expressiveness of differential equations with that of finite automata, and can, hence, express a richer class of multi-modal dynamical systems. We discuss the associated formalism and notation in this section.

5.1.1 Hybrid Automata

A *hybrid system* is obtained by composing a multi-modal dynamical system with a switching logic governing how the system switches through its different modes. Each mode of the hybrid automata models both the continuous physical world as well as the continuous control algorithms. While design of control for continuous dynamical systems corresponding to each mode are relatively well-understood [110, 67], the discovery of the switching logic between different modes is a major challenge in design of hybrid systems. Hybrid automata is formally defined in Definition 2.

Definition 2. *A hybrid automaton is a collection $H = (Q, X, Init, f, E, G, R)$, where*

- $Q = \{q_0, q_1, q_2, \ldots, q_m\}$ *is a set of discrete states;*
- X *is a set of real-valued continuous variables and* $|X| = n$*;*
- $Init \subseteq Q \times \mathbb{R}^n$ *is a set of initial states;*
- $f : Q \times \mathbb{R}^n \to \mathbb{R}^n$ *is a vector field;*
- $E : Q \times Q$ *is a set of edges;*

- $G : E \to \mathcal{P}(\mathbb{R}^n)$ *is a guard condition,*
  where $\mathcal{P}(\mathbb{R}^n)$ *denotes the power-set of* $\mathbb{R}^n$.

The following definition of hybrid automata, that is equivalent to Definition 2, decomposes it into a multi-modal dynamical system and a switching logic that controls the dynamical system switching through these modes.

Definition 3. *A hybrid automaton is a collection* $H = (MDS,\ SwL)$*, where*

- MDS *is a multimode dynamical system* $(Q, X, Init, f)$
  - $Q = \{q_0, q_1, q_2, \ldots, q_m\}$ *is a set of discrete states;*
  - X *is a set of real-valued continuous variables and* $|X| = n$*;*
  - $Init \subseteq Q \times \mathbb{R}^n$ *is a set of initial states;*
  - $f : Q \times \mathbb{R}^n \to \mathbb{R}^n$ *is a vector field;*
- $SwL : Q \times Q \to \mathcal{P}(\mathbb{R}^n)$ *is a guard condition,*
  where $\mathcal{P}(\mathbb{R}^n)$ *denotes the power-set of* $\mathbb{R}^n$.

For any given hybrid system $(Q, X, Init, f, E, G, R)$, the multimode dynamical system MDS is $(Q, X, Init, f)$ and the switching logic is given by

$$SwL(q_i, q_j) = \begin{cases} \emptyset & \text{if } (q_i, q_j) \notin E \\ G(q_i, q_j) & \text{if } (q_i, q_j) \in E \end{cases}$$

The state of the hybrid automata (q_i, x) is a tuple consisting of the discrete state $q_i \in Q$ and the continuous state $x \in X$. Starting from an initial value $(q_0, x_0) \in Init$, the continuous dynamics of the state x in the initial mode q_0 is given by the differential equation,

$$\dot{x} = f(q_0, x), x(0) = x_0,$$

while the discrete state q remains constant, i.e.,

$$q(t) = q_0.$$

Continuous evolution of the state x continues until it reaches a guard $G(q_0, q_j) \subseteq \mathbb{R}^n$ of some edge $(q_0, q_i) \in E$. When the continuous dynamics reaches the guard, the discrete state may change to

q_j. After this discrete change, the continuous evolution resumes in the new mode q_j. This process of continuous evolution and discrete jumps continues. Formally, a *trajectory* of a hybrid system is a continuous function $\tau(t) : [0, \infty) \mapsto \mathbb{R}^n$ if there is an increasing sequence $t_0 := 0 < t_1 < t_2 \ldots$, and a sequence of modes $q_0, q_1, \ldots$, such that

- $\tau(0) \in I$,
- for each interval $[t_i, t_{i+1})$, $\frac{d\tau}{dt}(t) = f_i(\tau(t))$ for all $t_i \leq t < t_{i+1}$,
- $\tau(t_{i+1}) \in G(q_i, q_{i+1})$ for all $i = 0, 1, 2 \ldots$

We use a simple example of a two tank system [5] to illustrate a hybrid automaton.

Example 1 (Two Tank System)**.** *The example is depicted in Figure 5.1. For $i \in \{1, 2\}$, x_i denotes the volume of water in Tank i and $v_i > 0$ denotes the constant flow of water out of Tank i. w denotes the constant flow of water into the system, dedicated exclusively to either Tank 1 or Tank 2 at any time instant. The objective is to keep the water volumes in Tank 1 and Tank 2 above r_1 and r_2 respectively, assuming that the water volumes are above r_1 and r_2 initially. This is to be achieved by a controller that switches the inflow to Tank 1 whenever $x_1 \leq r_1$ and to Tank 2 whenever $x_2 \leq r_2$.*

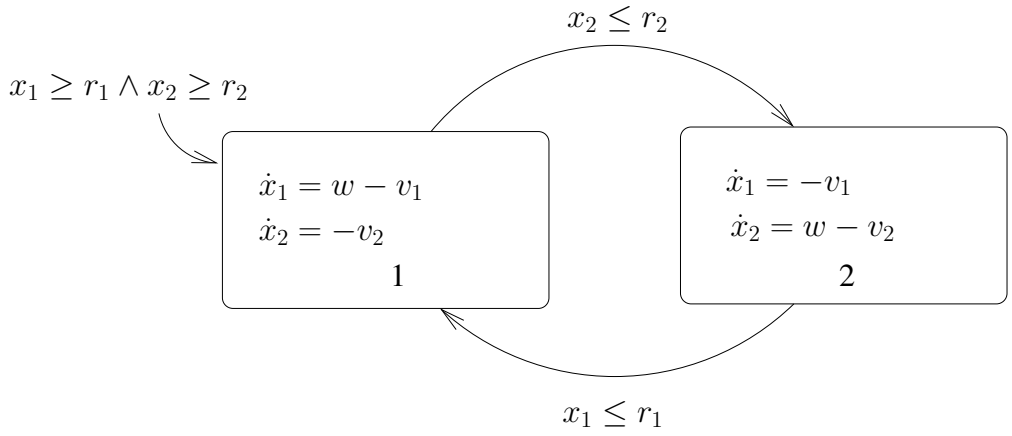


Figure 5.1: Two-Tank System

The hybrid automata description of the two-tank system is as follows:

$$H_{\texttt{2tank}} = (MDS_{\texttt{2tank}}, SwL_{\texttt{2tank}}),$$

where

- $MDS_{\texttt{2tank}}$ *is a multi-modal dynamical system* $(Q_{\texttt{2tank}}, X_{\texttt{2tank}}, Init_{\texttt{2tank}}, f_{\texttt{2tank}})$
  - $Q_{\texttt{2tank}} = \{1, 2\}$ *is the set of discrete states;*
  - $X = \{x_1, x_2\}$ *is the set of continuous variables that takes values in* $\mathbb{R}^2$*;*
  - $Init_{\texttt{2tank}} = (1, \{(x_1, x_2) | x_1 \geq r_1 \wedge x_2 \geq r_2\})$ *is the set of initial states;*
  - f *is the vector field with*
    - $f(1, x_1) = \dot{x}_1 = w - v_1$
    - $f(1, x_2) = \dot{x}_2 = -v_2$
    - $f(2, x_1) = \dot{x}_1 = -v_1$
    - $f(2, x_2) = \dot{x}_2 = w - v_2$
- SwL *is the switching logic with*
  - $SwL(1, 2) = \{(x_1, x_2) | x_2 \leq r_2\}$
  - $SwL(2, 1) = \{(x_1, x_2) | x_1 \leq r_1\}$

□

5.1.2 Boolean Properties

We can define temporal properties [88, 25, 57] on the system trajectories. These properties are either satisfied by the trajectories, or they are not satisfied. Hence, they are called Boolean properties. We now discuss the formal definition of temporal formula and the evaluation of a temporal formula on a given dynamical system.

A *state formula* ϕ is a predicate on state variables or a Boolean combination of predicates. A state formula is evaluated over a state. The formula ϕ evaluates to *true* on a state $\mathbf{x}$ if $\mathbf{x} \in \phi$. A state formula represents the set of states on which the formula evaluates to *true*. If ϕ, ϕ' are sets of states, then $\mathbf{G}\phi$, $\mathbf{F}\phi$, $\phi\mathbf{W}\phi'$ and $\phi\mathbf{U}\phi'$ are *temporal formulas*. A temporal formula is evaluated over a given trajectory τ.

- The formula $\mathbf{G}\phi$ evaluates to *true* on a trajectory τ if and only if

$$\forall t \geq 0 : \tau(t) \in \phi \tag{5.1}$$

Informally, the temporal formula $\mathbf{G}\phi$ is true if and only if ϕ is true on all states in the trajectory τ.

- The formula $\mathbf{F}\phi$ evaluates to true on a trajectory τ if and only if

$$\exists t \geq 0 : \tau(t) \in \phi \tag{5.2}$$

Informally, the temporal formula $\mathbf{F}\phi$ is true if and only if ϕ is true on at least some state in the trajectory τ.

- The formula $\phi \mathbf{U} \phi'$ evaluates to true on a trajectory τ if and only if

$$\exists t_0 : \tau(t_0) \in \phi' \ \wedge \ (\forall 0 \leq t < t_0 : \tau(t) \in \phi) \tag{5.3}$$

Informally, the temporal formula $\phi \mathbf{U} \phi'$ is true if and only if ϕ' becomes true eventually and until it becomes true, ϕ is true.

- The weak until operator, $\mathbf{W}$, is a weaker specification than the $\mathbf{U}$ operator, and does not require that ϕ' necessarily becomes true. If ϕ, ϕ' are sets of states, then the temporal formula $\phi \mathbf{W} \phi'$ evaluates to true over a given trajectory τ if and only if

$$\begin{aligned}(\exists t_0 : \tau(t_0) \in \phi' \ \wedge \ (\forall 0 \leq t < t_0 : \tau(t) \in \phi)) \ \vee \\ (\forall t \geq 0 : \tau(t) \in \phi)\end{aligned} \tag{5.4}$$

A state formula can be evaluated on a trajectory as follows: a state formula ϕ evaluates to *true* on a trajectory τ if and only if $\tau(0) \in \phi$. We can combine state and temporal formulas using Boolean connectives and evaluate them over trajectories using the natural interpretation of the Boolean connectives. If Φ is a state or temporal formula, then we write

$$Mode_i, \ I \ \models \ \Phi$$

to denote that the formula Φ evaluates to true on *all* trajectories of the system in mode i that start from a state in I.

Hybrid systems are often required to satisfy some safety invariant `safe` which is a predicate over the continuous variables. For all evolutions of the system starting from any initial state in the set $Init$, all the reachable system states are required to lie in the safe region. These safety

invariants are written as a temporal property $\mathbf{G}(\texttt{safe})$, and are of particular interest to us. To familiarize our readers with the kind of requirements that can be stated as a safety property, we present a few example of safety properties below:

- Temperature (temp) of a room is always between 18 and 20 degree Celsius. The safety property is

$$18 \leq \texttt{temp} \leq 20$$

.

- If the distance between two air-crafts ($dist$) is below a certain threshold (THRES), the air-crafts move away from each other, that is their relative velocity ($relV$) is positive. The corresponding safety property is

$$dist < \texttt{THRES} \Rightarrow relV > 0$$

.

- Either of the two fuel tanks has a fuel level ($level_1, level_2$) above a minimum threshold (TMIN) but the level in both fuels are below the maximum threshold (TMAX). The safety property is

$$(level_1 \geq \texttt{TMIN} \vee level_2 \geq \texttt{TMIN}) \wedge (level_1 \leq \texttt{TMAX} \wedge level_2 \leq \texttt{TMAX})$$

.

5.1.3 Quantitative Properties

Designers often require that the hybrid systems satisfy certain quantitative performance properties. In our approach, quantitative performance metrics are defined using new continuous state variables, that compute "rewards" or "penalties" accumulated over the course of a hybrid trajectory. We also allow the new variables to be updated during discrete transitions. Such updates enable us to penalize or reward discrete mode switches.

Definition 4. Performance Metric. *A* performance metric *for a given multi-modal system* $\texttt{MDS} := \langle Q, X, \texttt{Init}, f \rangle$ *is a tuple* $\langle \texttt{PR}, f_{\texttt{PR}}, \texttt{update} \rangle$*, where* $\texttt{PR} := P \cup R$ *is a finite set of continuous*

variables (disjoint from X), partitioned into penalty variables P and reward variables R, $f_{\mathtt{PR}} : Q \times \mathbb{R}^X \mapsto \mathbb{R}^{\mathtt{PR}}$ *defines the vector field that determines the evolution of the variables* PR*, and* $\mathtt{update} : Q \times Q \times \mathbb{R}^{\mathtt{PR}} \mapsto \mathbb{R}^{\mathtt{PR}}$ *defines the updates to the variables* PR *at mode switches.*

Given a trajectory $\mathbf{qx} : [0, \infty) \mapsto (Q \times \mathbb{R}^X)$ of a multi-modal or hybrid system with mode-switching time sequence $t_1, t_2, \ldots$, and given a performance metric, we define the *extended trajectory* $\mathbf{qx}^e : \mathbb{R}^+ \mapsto (Q \times \mathbb{R}^X \times \mathbb{R}^{\mathtt{PR}})$ with respect to the same mode-switching time sequence as a function that satisfies $\mathbf{qx}^e(0) = (\mathbf{q}(0), \mathbf{x}(0), \vec{0})$ and $\mathbf{qx}^e(t) = (\mathbf{q}(t), \mathbf{x}(t), \mathbf{PR}(t))$, where $\mathbf{PR}$ satisfies:

$$\frac{d\mathbf{PR}(t)}{dt} = f_{\mathtt{PR}}(\mathbf{qx}(t)) \text{ for all } t : t_i < t < t_{i+1} \text{ and,}$$
$$\mathbf{PR}(t_i) = \mathtt{update}(\mathbf{q}(t_{i-1}), \mathbf{q}(t_i), \lim_{t \to t_i^-} \mathbf{PR}(t))$$

where t_i^- denotes the left-hand limit of t_i.

In our approach, the cost of a trajectory $\mathbf{qx}$ is defined using its corresponding extended trajectory $\mathbf{qx}^e$ as

$$\mathtt{cost}(\mathbf{qx}) := \lim_{t \to \infty} \sum_{i=1}^{|P|} \frac{\mathbf{P}_i(t)}{\mathbf{R}_i(t)} \tag{5.5}$$

where $\mathbf{P}_i$ and $\mathbf{R}_i$ are the projection of $\mathbf{qx}^e$ onto the i-th penalty variable and i-th reward variable, and $|P| = |R|$.

We are only interested in trajectories where the above limit exists and is finite. As the definition of cost indicates, we are interested in the *long-run average* (penalty per unit reward) cost rather than (penalty or reward) cost over some bounded/finite time horizon. Some examples of auxiliary performance variables (PR) and cost function are described below:

- the *number of switches* that take place in a trajectory can be tracked by defining an auxiliary variable p_1 that has dynamics $\frac{dp_1}{dt} = 0$ at all points in the state space, and that is incremented by 1 at every mode switch, that is,

$$p_1(t_i) = \mathtt{update}(q, q', p_1(t_i^-)) = p_1(t_i^-) + 1$$

- the *time elapsed since start* can be tracked by defining an auxiliary variable r_1 that has dynamics $\frac{dr_1}{dt} = 1$ at all points and that is left unchanged at discrete transitions; that is,

$$r_1(t_i) = \mathtt{update}(q, q', r_1(t_i^-)) = r_1(t_i^-)$$

- the *average switchings* (per unit time) can be observed to be $\frac{p_1}{r_1}$. If this cost becomes unbounded as the time duration of a trajectory increases, then this indicates *zeno* behavior [54]. Thus, if we use p_1 and r_1 as the penalty and reward variables in the performance metric, then we are guaranteed that non-zeno systems will have "smaller" cost and thus be "better".
- the *power consumed* could change in different modes of a multi-modal system and an auxiliary (penalty) variable p_2 that has dynamics $\frac{dp_2}{dt} = c_j$ in mode j, where c_j is the rate of power consumption in mode j, can track the power consumed in a particular trajectory. p_2 is left unchanged at discrete transitions; that is,

$$p_2(t_i) = \texttt{update}(q, q', p_2(t_i^-)) = p_2(t_i^-)$$

- the *distance from unsafe region* can be tracked by an auxiliary reward variable r_2 that evolves based on the distance of the current state from the closest unsafe state. Let the unsafe region be `unsafe` and the distance metric be D, the evolution of r_2 is given by

$$\frac{dr_2}{dt} = \min_{y \in \texttt{unsafe}} D(y, x(t))$$

where $x(t)$ is the state of the system at time t and y is some unsafe state. r_2 is left unchanged at discrete transitions; that is,

$$r_2(t_i) = \texttt{update}(q, q', r_2(t_i^-)) = r_2(t_i^-)$$

Synthesis of switching logic for a given safety or performance specification is the focus of this part of the thesis. For various choices of the switching logic, the hybrid automata will show different behaviors. Because of the interactions between the discrete switching and the continuous dynamics, it is difficult to compute the switching logic that enables the hybrid automaton to satisfy a given safety or performance specification. We formally define the switching logic synthesis problem below.

Definition 5 (Switching logic synthesis problem)**.** *Given a multi-modal continuous dynamical system ($MDS := (Q, X, Init, f)$), a safety specification ϕ_S, and/or a performance metric $\langle \texttt{PR}, f_{\texttt{PR}}, \texttt{update} \rangle$, over the penalty and reward variables $\texttt{PR} := P \cup R$, the switching logic synthesis problem seeks to synthesize a switching logic* `SwL` *such that the hybrid system* $\texttt{HS} := (\texttt{MDS}, \texttt{SwL}')$ *is safe with respect to the safety property, and/or optimal with respect to the performance metric.*

5.2 Related Work

Automated control of continuous systems has being widely studied. A detailed study of existing techniques in control theory can be found in textbooks [119], [110] and [67]. We focus on the research literature on the automated synthesis of switching logic of hybrid systems and describe the salient features of the related work.

5.2.1 Synthesis for Boolean Safety Properties

Control synthesis of hybrid systems for Boolean properties has been previously investigated in literature.

Techniques Based on Reachability Analysis

One of the first automated synthesis approaches for hybrid systems is presented by Lygeros et al. [82]. They consider the problem of controlling hybrid systems with controllable continuous inputs as well as uncontrollable noises (disturbances) against two objectives having different priorities. The first is a safety property, and has the highest priority. The second is an efficiency objective, and has a lower priority. A class of least restrictive safe controllers can be identified which satisfies the safety objective. The optimal controller, with respect to efficiency, can then be found within this class of least restrictive safe controllers. The technique is used for discrete systems, continuous systems, as well as hybrid systems. The hybrid system considered is the industrial steam boiler benchmark problem [53]. Motivated by aircraft conflict resolution benchmark, Tomlin et al. [130, 128] presented a technique to synthesize controllers for nonlinear hybrid systems. They showed a conceptual equivalence between a two player game on finite automata in which the goal of one player is to keep the system inside a certain *good* subset of the state space and the goal of the other player is force the system out of this *good* subset, to the Hamilton-Jacobi-Bellman equation for differential games on continuous systems. Their technique is based on the computation of the maximal safe set such that the controllable actions and continuous input keep the system safe irrespective of the uncontrollable disturbances. The technique requires numerical computation of the solution of Hamilton-Jacobi-Bellman partial differential equation (PDE) and,

hence, is limited in scalability by the number of dimensions. Also, numerical error needs to be bounded to ensure that the synthesized controller guarantees safety. For hybrid systems, where the continuous dynamics are defined by linear differential equations, Asarin et al. [8] present a synthesis technique to discover the conditions upon which the controller should switch the behavior of the system from one mode to another in order to avoid a set of bad states. Their approach is based on iterative computation of reachable states. They use approximate reachability analysis in linear systems to compute reachable states [9]. Raffard et al. [106] consider stochastic hybrid systems where the continuous dynamics is governed by stochastic differential equations and the discrete mode evolves according to a continuous time Markov chain. They synthesize feedback control that keeps the state of the stochastic hybrid system within a prescribed region for at least an expected amount of time. Their approach is based on replacing the Hamilton-Jacobi based technique with a PDE constrained optimization problem. Optimization is done using adjoint-based gradient descent methods [79]. A constraint-based technique for synthesizing switching logic is also proposed by Taly et al. [126, 127]. It involves generating and solving an $\exists - \forall$ constraint. However, the size of the constraint increases as the number of modes increase, and hence, the technique is useful only with a small number of modes.

Techniques Based on Approximation and Abstraction

Approximation based techniques [98, 28] to synthesize control systems for hybrid systems have also been proposed in literature. Moor et al. [98] suggest an approach for synthesizing supervisory control for switched linear systems based on two conservative approximation techniques. The switched linear system is approximated by sampling and state space partitioning. Then, the resultant finite state machine is translated into a past induced finite state machine using l-completion [97]. l-completion approximates a super set of the original behavior, and, hence, the estimate of reachable states based on an l-completion is conservative, that is, the exact set of reachable states is guaranteed to be contained in the estimate. Discrete techniques can then be used to solve the supervisory control problem on the approximation level. Since the approximation is conservative, the desired closed loop properties are retained if the supervisor is connected to the underlying switched linear system. Cury et al. [28] also describe an approximation based approach to synthesizing discrete controllers for hybrid systems. Their approach uses finite-state approximations of the hybrid system dynamics. Their synthesize a supervisory control that selects

the plant input signal based on the observed discrete events to assure the closed-loop system satisfies the design specifications. They also present a technique to refine the coarse approximations, and develop better solutions to the control synthesis problem.

5.2.2 Synthesis for Quantitative Performance Properties

Another line of related work is the optimal control of hybrid systems with respect to some quantitative metric stemming from the seminal work of Branicky et al. [18]. They established a necessary condition for the optimal trajectory under a general cost function in terms of quasi-variational inequalities. But, they did not present an algorithm to compute the desired control.

Optimal Trajectory from Initial to Final State

Manon et al. [89] propose a method to synthesize a trajectory to drive a hybrid system with linear vector fields from an initial state to a final state. Kamgarpour et al [64] and Gonzalez et al. [43, 42] consider the problem of synthesizing optimal control of switched dynamical systems with nonlinear continuous dynamics. The control inputs considered are both the discrete component corresponding to the sequence of mode switches as well as the continuous component corresponding to the time spent in each mode, and the continuous input in each mode. The goal is to synthesize an optimal control that drives the system from an initial state to a target final state. The cost function includes both the running cost of the trajectory and the cost of the final state reached. They propose a bi-level hierarchical algorithm that divides the problem into two nonlinear constrained optimization problems. The lower level corresponds to keeping the mode sequence fixed, and discovering the optimal mode duration and the optimal continuous input. The higher level corresponds to the discovery of the optimal mode sequence using single mode insertion technique. The approach was extended by the authors [43] to allow multiple objectives in the cost function, and penalize the number of hybrid jumps. Another gradient descent based approach to the synthesis of hybrid controllers is presented by Axelsson et al. [10]. They also use a bi-level hierarchical algorithm. At the lower level, their algorithm considers a fixed mode sequence, and minimizes the cost function with respect to the modes durations. At the higher level, it updates the mode-sequence by using a gradient technique.

Shortest Mode Switching Sequence

Koo et al. [71] present a synthesis technique for determining a switching sequence through low level control modes to accomplish some high-level task. The continuous controllers are assumed to have been pre-designed, and only mode switching conditions need to be synthesized. Their approach exploits the structure of the output tracking controllers in order to extract a finite graph where the mode switching problem is solved. The solution can then be implemented using the continuous controllers. Thus, they decouple the problem of synthesizing discrete and continuous controllers. Given an initial mode and the target mode, they discover the shortest consistent mode switching sequence that can later be enforced by the synthesis of suitable continuous control inputs.

Optimal Switching Times

Synthesis of optimal switching times for a given mode sequence has also been investigated in literature by Shaikh et al. [113] and Kamgarpour et al [64]. Here, the mode switching sequence is fixed but the optimal dwell times in each mode are discovered. The method can be combined with combinatorial search methods to discover the most optimal switching sequence which is at most k-Hamming distance from an initial given sequence. Xu et al. [133] also consider a similar formulation where the sequence of mode switches is available. Their goal is to synthesize the continuous control input that would result into the most optimal trajectory.

Long-run Cost

Notions of long-run cost similar to that presented in Section 5.1.3 have appeared in other areas. The notion of long-run average cost is used in economics to describe the cost per unit output (reward) in the long-run. In computer science, long-run costs have been studied for graph optimization problems [66]. Long-run average objectives have also been studied for Markov decision processes (MDPs) [24, 73]. However, MDPs do not have any continuous dynamics. Another related work is optimal scheduling using the priced timed automata [107], in which timed automata is extended by associating a fixed cost to each transition and a fixed cost rate per time unit in a location. Game-theoretic synthesis techniques for discrete systems with quantitative objectives have

also being studied in literature where the long-run cost is expressed using lexicographic mean-payoff conditions [14]. We consider multi-modal dynamical systems with possibly non-linear dynamics, and our cost rates are functions of the continuous variables.

Nonlinear Optimization

In order to synthesize optimal switching logic, we need to solve possibly nonlinear optimization problems where the function to be optimized is not available analytically but is provided only as a black-box oracle. The oracle accepts finite precision arguments and returns the value of the function with finite precision, that is, with a limited number of digits. Hence, our interest in solving these optimization problems is in finding approximate solutions with some prescribed accuracy ϵ. This concept of ϵ-accurate optimization was introduced by Hochbaum and Shantikumar [56], and is discussed in detail in a recent survey paper [55]. A solution is said to be ϵ-accurate if it is at most at a distance of ϵ from an optimal solution, that is, the solution is identical to the optimum in $O(log\frac{1}{\epsilon})$ decimal digits. It has been shown that any convex separable [23] optimization problem on totally unimodular constraints (or problems with constraint matrices that have bounded subdeterminants) can be solved in polynomial time [56]. The polynomiality is in the input size and the log of the accuracy required in the solution $(1/\epsilon)$. The optimization problem generated by our technique is not guaranteed to be convex and hence, we rely on numerical optimization techniques which work well in practice. We use Nelder-Mead method [101] which is available as `fminsearch` function [91] in Matlab to solve the optimization problems. But the choice of optimization routine is orthogonal to our proposed synthesis approach, and any optimization technique can be used as the backend of our approach. This allows us to levarage the progress in the field of constraint optimization for further scaling our synthesis approach, and making it more efficient.

5.2.3 Dimensions

The related work on synthesis of controllers for hybrid systems presented above can be broadly classified along several different dimensions: (1) property of interest which the synthesized system is expected to satisfy, (2) control inputs which need to be synthesized and (3) the approach used for synthesis.

1. First, based on the *property of interest*, synthesis work broadly falls into one of the two categories. The first category finds controllers that meet some safety specification [130, 128, 8]. The second category finds controllers that meet some *liveness* specifications, such as synthesizing a trajectory to drive a hybrid system from an initial state to a desired final state [18, 43, 42, 89, 71], while also minimizing some cost metric. Purely constraint based approaches for solving switching logic synthesis problem have also being used for reachability specifications [126, 127].

2. The second dimension that differentiates work on controller synthesis for hybrid systems is the space of *control inputs* considered; that is, what is assumed to be controllable. The space of controllable inputs could consist of any combination of continuous control inputs, the mode sequence, and the dwell times within each mode. Gonzales et al. [43, 42] consider all the three control parameters, whereas some other works either assume the mode sequence is not controllable [133, 113] or there are no continuous control inputs [10, 71].

3. The third dimension for placing work on controller synthesis of hybrid systems is the *approach used for solving the synthesis problem*. There are direct approaches for synthesis that compute the controlled reachable states in the style of solving a game [8, 128], and abstraction-based approaches that do the same, but on an abstraction or approximation of the system [98, 28]. The other class of approaches are based on using nonlinear optimization techniques and gradient descent [42, 10].

With respect to related work, the categorization of our contributions using the three dimensions of synthesis identified above is as follows:

1. *Property of interest:* Previous work on automated control synthesis for hybrid systems have been targeted towards safety specifications [130, 128, 8] as well as reachability specifications [18, 43, 42, 89, 71]. In Chapter 6, we present an automated synthesis technique targeted towards safety properties. In Chapter 7, we present an automated synthesis technique target-ted towards performance objectives where the goal is to minimize a specified long-run cost metric.

2. *Control Input:* Previous work on synthesis of hybrid systems synthesize either all or some of the following components of a hybrid system: continuous control inputs, dwell times in

each mode, and mode sequence for some high level tasks [43, 42, 133, 113, 10, 71]. In our work in Chapter 6 and Chapter 7, we assume that there are no continuous control inputs which need to be synthesized, and both the mode sequence and the dwell time within each mode are controllable entities.

3. *Approach:* Previous approaches have used game-theoretic techniques either directly [8, 128] or using abstraction or approximation [98, 28] as well as nonlinear optimization techniques [42, 10]. We present a SCIDUCTIVEreasoning based approach to automatically synthesize switching logic of hybrid systems which combines deductive reasoning techniques such as SMT solving and numerical optimization with inductive techniques from algorithmic learning. Our approach is discussed in detail in Chapter 6 and Chapter 7.

Chapter 6

Synthesis of Switching Logic for Safety Specifications

In this chapter, we present a new approach to assist designers of cyber-physical systems by synthesizing the switching logic, given a partial system model, using a combination of fixpoint computation, numerical simulation, and machine learning. Our technique begins with an over-approximation of the guards on transitions between modes. In successive iterations, the over-approximations are refined by eliminating points that will cause the system to reach unsafe states, and such refinement is performed using numerical simulation and machine learning. In addition to safety requirements, we synthesize models to satisfy dwell-time constraints, which impose upper and/or lower bounds on the amount of time spent within a mode. We demonstrate using case studies that our technique quickly generates intuitive system models and that dwell-time constraints can help to tune the performance of a design.

6.1 Introduction

As discussed in Chapter 5, a multi-modal dynamical system (MDS) is a physical system (plant) that can operate in different modes. The dynamics of the plant in each mode is known. However, to achieve safe and efficient operation, it is often necessary to switch between the different operating modes. Designing correct switching logic can be tricky and tedious. We consider the problem

of automatically synthesizing switching logic, given the intra-mode dynamics, so as to preserve safety in MDS. The human designer can guide the synthesis process by providing initial approximations of the switching guards and a library of expressions (components) using which the guards can be synthesized.

Our synthesis approach performs reasoning within each mode and reasoning across modes in two different ways. Within each mode, reasoning is based entirely on using *numerical simulations*. While this can lead to potential unsoundness, it allows us to handle complex and nonlinear dynamics that are difficult to reason about in any other way. Across modes, reasoning is performed using *fixpoint computation techniques*. Similar to abstract interpretation, computation of the fixpoint is performed over an "abstract domain," which is specified by the user in the form of a component library for the switching guards. Each step of the fixpoint computation involves the use of *machine learning* to learn improved approximations of the switching guards based on the results of numerical simulations.

6.1.1 Contributions

We make the following novel contributions in this chapter.

- The key contribution of our chapter is a *new approach for synthesizing safe switching logic based on integrating numerical simulation, machine learning, and fixpoint iterations*.
- In addition to safety, our approach also extends to handling dwell-time requirements, which impose upper and/or lower bounds on the amount of time spent within a mode.
- While numerical simulations have been used to perform formal verification (e.g., [65, 39, 30]), to our knowledge our approach is the first to use simulations to perform synthesis with safety guarantees.
- We demonstrate using case studies (Sec. 6.1.3 and 6.3) that our technique generates intuitive system models and that dwell-time constraints can help to tune the performance of a design.

6.1.2 Problem Definition

In this section, we describe the problem of synthesizing switching logic for a multi-modal continuous dynamical system. We present two versions of the problem. In the first version, we ask for a switching logic that only preserves safety. In the second version, we also require that the synthesized system satisfy some dwell-time requirements in each mode. We begin with some definitions.

A *continuous dynamical system* (CDS) is a tuple $\langle X, f \rangle$ where X is a finite set of $|X| = n$ real-valued variables that define the state space $\mathbb{R}^n$ of the continuous dynamical system, and $f : \mathbb{R}^n \mapsto \mathbb{R}^n$ is a vector field that specifies the continuous dynamics as $\frac{d\mathbf{x}}{dt} = f(\mathbf{x})$. The vector field f is assumed to be locally Lipschitz at all points, which guarantees the existence and uniqueness of solutions to the ordinary differential equations.

Often, a system has multiple modes and in each mode, its dynamics is different. Such a multi-modal system behaves as a different continuous dynamical system in each mode.

Definition 6 (Multi-modal CDS (MDS))**.** *An MDS is a tuple* $\langle X, I, f_1, f_2, \ldots, f_k \rangle$ *where*

- $\langle X, f_i \rangle$ *is a continuous dynamical system (representing the* i*-th mode)*
- $I \subseteq \mathbb{R}^n$ *is the set of initial states*

We will use $Q = \{1, 2, \ldots, k\}$ as the set of indices of the modes. A *trajectory* for MDS is a continuous function $\tau(t) : [0, \infty) \mapsto \mathbb{R}^n$ if there is an increasing sequence $t_0 := 0 < t_1 < t_2 \ldots$ such that

- $\tau(0) \in I$,
- for each interval $[t_i, t_{i+1})$, there is some mode $j \in Q$ such that $\frac{d\tau}{dt}(t) = f_j(\tau(t))$ for all $t_i \leq t < t_{i+1}$, and
- $j = 1$ when $t_i = 0$ (that is, we start in Mode 1.).

A multi-modal system can nondeterministically switch between its modes. The goal is to control the switching between different modes to achieve safe operation.

Definition 7 (Switching logic (SwL))**.** *A switching logic* SwL *for an MDS* $\langle X, I, (f_i)_{i \in Q} \rangle$ *is a tuple* $\langle (\mathsf{g}_{ij})_{i \neq j; i,j \in Q} \rangle$*, containing guards* $\mathsf{g}_{ij} \subseteq \mathbb{R}^n$.

A multi-modal system MDS can be combined with a switching logic SwL to create a hybrid system HS := (MDS, SwL) in the following natural way: the hybrid system HS has k modes with dynamics given by $\frac{dX}{dt} = f_i$ in mode i, and with g_{ij} being the guard on the discrete transition from mode i to mode j. The initial states of HS are I in Mode 1, where I is the set of initial states of the MDS. The discrete transitions in HS have identity reset maps, that is, the continuous variables do not change values during discrete jumps. The state invariant Inv_i for a mode $i \in I$ is the (topological) closure of the complement of the union of all guards on outgoing transitions; in other words $Inv_i := Closure(\mathbb{R}^n - \bigcup_{j \in I} g_{ij})$. Note that we are assuming here that a discrete transition is taken as soon as it is enabled.This completes the definition of the hybrid system. The semantics of hybrid systems that defines the set of *reachable states* of hybrid systems is standard [3].

A safety property is a set $\phi_S \subseteq \mathbb{R}^n$ of states. We will overload ϕ_S to also denote the predicate $\phi_S(X)$. A state $\mathbf{x}$ is said to be *safe* if and only if $\mathbf{x} \in \phi_S$ (or equivalently, if $\phi_S(\mathbf{x})$ is true). A hybrid system HS is safe with respect to ϕ_S if and only if all the reachable states in HS are safe.

Coming up with the correct guards for the mode switches such that all reachable states are safe is challenging and our proposed technique aims at automating this task. While controller synthesis has been widely studied, what differentiates our work is that we provide the designer an option to provide some initial partial design. Specifically, we assume that the designer can provide an over-approximations for the guards. In the extreme case, if transition from mode i to mode j is disallowed, then the designer can set $g_{ij} = \emptyset$, and if the designer knows nothing about the possibility of a transition from mode i to mode j, then she can set $g_{ij} = \mathbb{R}^n$. The designer can specify partial information by picking an intermediate set as the initial guard.

If SwL := $\langle (g_{ij})_{i,j \in Q} \rangle$ and SwL′ := $\langle (g'_{ij})_{i,j \in Q} \rangle$ are two switching logics, then we use the notation SwL′ $\subseteq$ SwL to denote that $g'_{ij} \subseteq g_{ij}$ for all $i, j \in Q$.

We provide two variants of the problem definition.

Definition 8 (Switching logic synthesis problem v1)**.** *Given a multi-modal continuous dynamical system (MDS), a switching logic* SwL*, and the safety specification ϕ_S, the switching logic synthesis problem seeks to synthesize a new switching logic* SwL′ *such that*
(1) SwL′ $\subseteq$ SwL *and*
(2) the hybrid system HS := (MDS, SwL′) *is safe with respect to ϕ_S.*

Consider the case when the designer provides no information and sets all guards to $\mathbb{R}^n$. In this case, it is trivial to synthesize a safe hybrid system by just setting all switching guards to be ϕ_S. The reader can check that this is a solution for the switching logic synthesis problem defined above. This solution is, however, undesirable since the resulting hybrid system has only *zeno behaviors*, i.e., an infinite number of transitions can be made in finite time (as we are assuming that a transition is taken as soon as it is enabled).

In order to rule out systems with *zeno behaviors*, we use dwell time specifications. Dwell time is a well-known concept in hybrid systems [52, 95, 96], where it has been used for verification. We use dwell time as a requirement for synthesis. The user can use it to guarantee synthesis of non-zeno and desirable systems.

The second problem definition below gives the designer a way to explicitly rule out solutions that have zeno behavious using dwell time specifications. Specifically, the user can specify (both lower and upper) bounds on the amount of time every trajectory should spend in a mode.

Definition 9 (Switching logic synthesis problem v2)**.** *Given a multi-modal continuous dynamical system (MDS), a switching logic* $\mathtt{SwL}$*, a sequence* $\langle te_1, \ldots, te_k \rangle$ *of non-negative minimum-dwell time requirements, a sequence* $\langle tx_1, \ldots, tx_k \rangle$ *of non-negative maximum-dwell time requirements, and a safety specification* ϕ_S*, the switching logic synthesis problem seeks to synthesize a new switching logic* $\mathtt{SwL}'$ *such that*
(1) $\mathtt{SwL}' \subseteq \mathtt{SwL}$*,*
(2) the hybrid system $\mathtt{HS} := (\mathtt{MDS}, \mathtt{SwL}')$ *is safe with respect to* ϕ_S*, and*
(3) whenever any trajectory of $\mathtt{HS}$ *enters mode* i*, it stays in mode* i *for atleast* te_i *and atmost* tx_i *time units.*

The designer can now force the synthesis of only nonzeno systems by setting te_i to a strict positive number for selected modes. Note that if the designer sets te_i to zero and tx_i to ∞ for all modes, then the second problem is the same as the first problem.

We can defined state and temporal formulas over the trajectories as discussed in Section 5.1. A state formula can be evaluated on a trajectory as follows: a state formula ϕ evaluates to true on a trajectory τ if $\tau(0) \in \phi$. We can combine state and temporal formulas using Boolean connectives and evaluate them over trajectories using the natural interpretation of the Boolean connectives.

If Φ is a state or temporal formula, then we write

$$Mode_i,\ I \ \models \ \Phi$$

to denote that the formula Φ evaluates to true on *all* trajectories of the CDS in mode i that start from a state in I.

6.1.3 Running Example

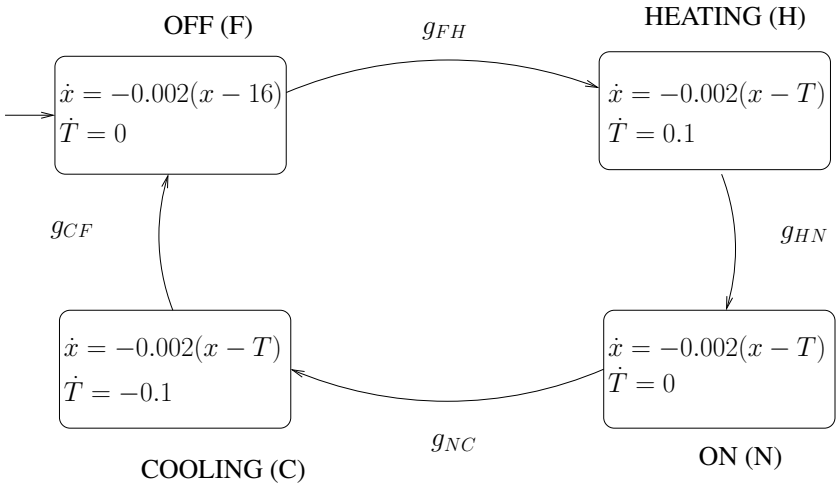


Figure 6.1: Thermostat

In this section, we present an overview of our approach using a thermostat controller [81] as an example. The 4-mode thermostat controller is presented in Figure 6.1. The room temperature is represented by x and the temperature of the heater is represented by T. The initial condition I is given by $T = 20°\mathrm{C}$ and $x = 19°\mathrm{C}$. The safety requirement ϕ_S is that the room temperature lies between $18°\mathrm{C}$ and $20°\mathrm{C}$, that is, ϕ_S is $18 \leq x \leq 20$. (We omit the units in the sequel, for brevity.)

In the OFF mode, the temperature falls at a rate proportional to the difference between the room temperature x and the temperature outside the room which is assumed to be constant at 16. In the HEATING mode, the heater heats up from 20 to 22 and in the COOLING mode, the heater cools down from 22 to 20. In the ON mode, the heater is at a constant temperature of 22. In the HEATING, ON and COOLING mode, the temperature of the room changes in proportion to the

difference between the room temperature and the heater temperature. We need to synthesize the four guards: g_{FH}, g_{HN}, g_{NC} and g_{CF}.

The guards must respect the safety property on the room temperature x as well as the specification on the heater temperature T in HEATING and COOLING mode. So, from the given specifications, we know that

$$\begin{aligned} g_{FH} &\subseteq 18 \leq x \leq 20 \wedge T = 20 \\ g_{HN} &\subseteq 18 \leq x \leq 20 \wedge T = 22 \\ g_{NC} &\subseteq 18 \leq x \leq 20 \wedge T = 22 \\ g_{CF} &\subseteq 18 \leq x \leq 20 \wedge T = 20 \end{aligned} \tag{6.1}$$

In order that the MDS remains safe, we need to ensure that all states reachable within each mode are safe. Consider the OFF mode. We need to ensure that all traces starting from some point in the initial condition I or g_{CF} do not reach an unsafe state before reaching some state in g_{FH}. Reaching some state in g_{FH} enables a transition out of the OFF mode. In other words, the first two temporal properties in Equation 6.2 must be satisfied by all traces in the OFF mode. Similarly, for HEATING mode, all traces starting from some state in $x \in g_{FH}$ must not reach an unsafe state before reaching an exit state in g_{HN}, as indicated by the third property below. For the other two modes, similar temporal properties on the traces need to be enforced. Overall, the following temporal assertions can be written for the four guards.

$$\begin{aligned} F, I &\models \phi_S \, \mathbf{W} \, g_{FH} \\ F, g_{CF} &\models \phi_S \, \mathbf{W} \, g_{FH} \\ H, g_{FH} &\models \phi_S \, \mathbf{W} \, g_{HN} \\ N, g_{HN} &\models \phi_S \, \mathbf{W} \, g_{NC} \\ C, g_{NC} &\models \phi_S \, \mathbf{W} \, g_{CF} \end{aligned} \tag{6.2}$$

Switching Logic Synthesis Problem v1: We can synthesize a safe switching logic by computing the fixpoint of the above 5 assertions in Equation 6.2. We initialize using the equations in Equation 6.1 obtained from the safety and other user provided specifications which put an upper bound on the guards. We then perform a *greatest fixpoint computation*: in each iteration, we remove states from the guards which would lead to some unsafe state in a mode. Fixpoint computation leads to the following guards which ensure that all states reachable are safe. We compute only till the second

place of decimal.

$$g_{FH} : 18.00 \leq x \leq 19.90 \wedge T = 20$$
$$g_{HN} : 18.00 \leq x \leq 19.95 \wedge T = 22$$
$$g_{NC} : 18.00 \leq x \leq 19.95 \wedge T = 22$$
$$g_{CF} : 18.00 \leq x \leq 20.00 \wedge T = 20$$

The behavior of the synthesized thermostat for the first 1000 seconds from the initial state is shown in Figure 6.2. The room temperature gradually rises from its initial value of 19 and then stays between 19.90 and 20.

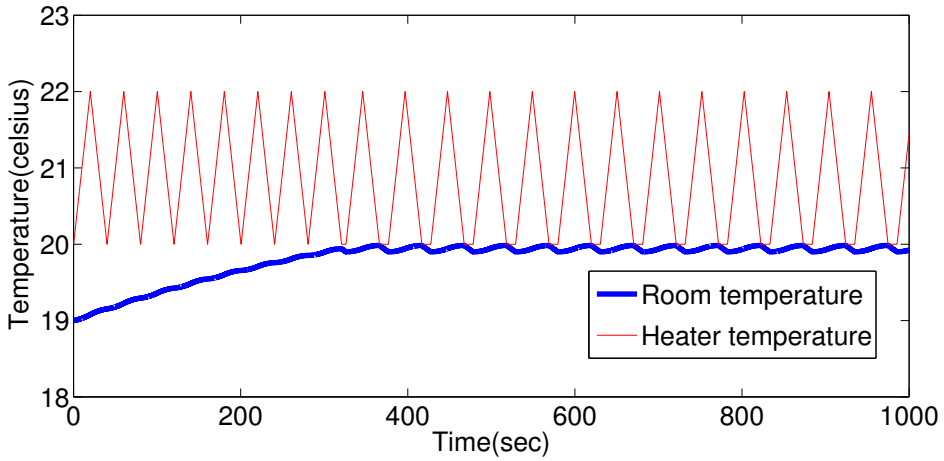


Figure 6.2: Behavior of Synthesized Thermostat

Switching Logic Synthesis Problem v2: Though the system synthesized above satisfies the safety specification, it has the undesirable behavior of switching frequently. It keeps the room temperature in the narrow interval of $19.90 \leq x \leq 20$, even though the safety condition only required it to be in $18 \leq x \leq 20$. Ideally, designers are interested not only in safe systems but in systems with good performance. The dwell time specification provides a mechanism to the designer to guide our synthesis technique to solutions with good performance.

Minimum dwell-time of 100 seconds in OFF mode (case A): We add an extra constraint in the specification of our synthesis problem that the system must spend atleast 100 seconds in the OFF mode. This would lead to less frequent switching as well as minimize energy consumption since heater remains off in the OFF mode.

Let us add a timer variable t with dynamics $\dot{t} = 1$ in every mode. Assume that t is reset to 0 during every discrete transition. To enforce the minimum dwell-time, the following constraint must also be satisfied in addition to the fixpoint constraints in Equation 6.2.

$$\begin{aligned} F, I &\models \phi_S \ \mathbf{W} \ (g_{FH} \wedge t \geq 100) \\ F, g_{CF} &\models \phi_S \ \mathbf{W} \ (g_{FH} \wedge t \geq 100) \end{aligned} \tag{6.3}$$

The guards obtained by computing the fixpoint of equations in (6.2) and (6.3) are as follows.

$$\begin{aligned} g_{FH} &: 18.00 \leq x \leq 19.90 \wedge T = 20 \wedge t \geq 100 \\ g_{HN} &: 18.00 \leq x \leq 19.95 \wedge T = 22 \\ g_{NC} &: 18.35 \leq x \leq 19.95 \wedge T = 22 \\ g_{CF} &: 18.45 \leq x \leq 20.00 \wedge T = 20 \end{aligned}$$

Since t was a timer variable we had introduced, we next eliminate it from g_{FH}. We do so by removing states from g_{FH} which are reachable from any state in g_{CF} in less than 100 seconds. These set of states are $18.01 < x \leq 20 \wedge T = 20$. Hence, the final guards that respect the safety property as well as enforce a minimum dwell-time of 100 seconds in OFF mode are as follows.

$$\begin{aligned} g_{FH} &: 18.00 \leq x \leq 18.01 \wedge T = 20 \\ g_{HN} &: 18.00 \leq x \leq 19.95 \wedge T = 22 \\ g_{NC} &: 18.00 \leq x \leq 19.95 \wedge T = 22 \\ g_{CF} &: 18.00 \leq x \leq 20.00 \wedge T = 20 \end{aligned}$$

The behavior of the synthesized thermostat for the first 1000 seconds from the initial state is shown in Figure 6.3. We observe that the number of switches has gone down from 21 to 5 and the room temperature now stays between 18.01 and 18.45.

Minimum dwell-time of 300 seconds in both OFF and ON mode (case B): We observe that the design synthesized with minimum dwell-time of 100 seconds in OFF mode has relatively less switching but still, we would like to reduce its switching frequency. Also, the room temperature can safely lie between 18 and 20 but in the above synthesized system, it is restricted to a narrow interval of 18.01 and 18.45. So, we increase the minimum dwell-time in OFF mode to 300 seconds. We also enforce a minimum dwell-time of 300 seconds in ON mode to ensure room heats up to a higher temperature within the safe interval.

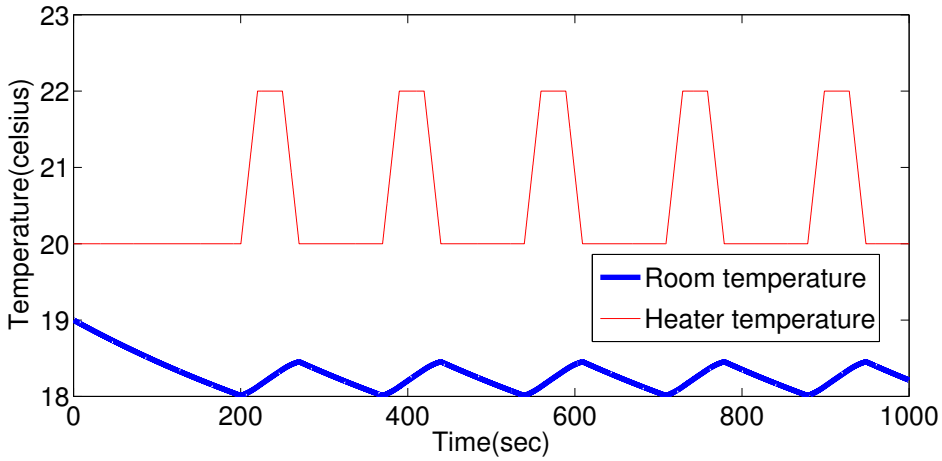


Figure 6.3: Behavior of Synthesized Thermostat with Dwell Time Specification: Minimum dwell time of 100s in OFF mode.

We now get the following fixpoint equations.

$$\begin{aligned} F, I &\models \phi_S \ \mathbf{W}\ g_{FH} \wedge (t \geq 300) \\ F, g_{CF} &\models \phi_S \ \mathbf{W}\ g_{FH} \wedge (t \geq 300) \\ H, g_{FH} &\models \phi_S \ \mathbf{W}\ g_{HN} \\ N, g_{HN} &\models \phi_S \ \mathbf{W}\ g_{NC} \wedge (t \geq 300) \\ C, g_{NC} &\models \phi_S \ \mathbf{W}\ g_{CF} \end{aligned}$$

Fixpoint computation yields the following guards.

$$\begin{aligned} g_{FH} &: 18.00 \leq x \leq 18.14 \wedge T = 20 \wedge t \geq 300 \\ g_{HN} &: 18.00 \leq x \leq 18.26 \wedge T = 22 \\ g_{NC} &: 19.60 \leq x \leq 19.95 \wedge T = 22 \wedge t \geq 300 \\ g_{CF} &: 19.65 \leq x \leq 20.00 \wedge T = 20 \end{aligned}$$

We restrict g_{NC} and g_{FH} in the same way as (Case A) by computing the set of states reachable from g_{HN} and g_{CF} in less than 300 seconds respectively. The final synthesized guards are as follows.

$$\begin{aligned} g_{FH} &: 18.00 \leq x \leq 18.01 \wedge T = 20 \\ g_{HN} &: 18.00 \leq x \leq 18.26 \wedge T = 22 \\ g_{NC} &: 19.94 \leq x \leq 19.95 \wedge T = 22 \\ g_{CF} &: 19.65 \leq x \leq 20.00 \wedge T = 20 \end{aligned}$$

The behavior of the synthesized thermostat for the first 1000 seconds from the initial state is shown in Figure 6.4. We observe that the number of switches has gone down to 1 and the room temperature is still within the safe interval of 18 and 20. This example shows how our synthesis

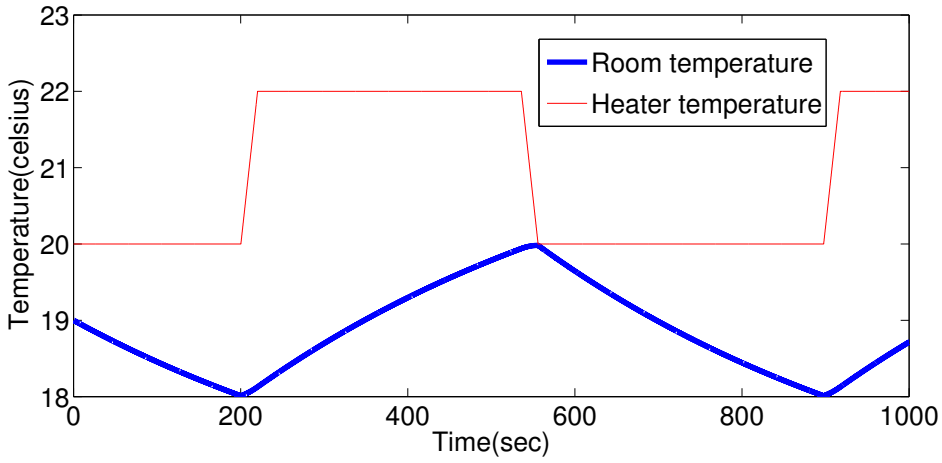


Figure 6.4: Behavior of Synthesized Thermostat with Dwell Time Specification: Minimum dwell time of 300s in OFF and ON modes.

approach can be used to synthesize not only safe systems but also systems with desired performance. Dwell-time properties can be used by the user to explore designs with better performance.

6.2 SCIDUCTIVE Approach

We are now ready to describe the procedure for solving the switching logic synthesis problem in Definition 8.

Assume that we are given an MDS MDS, a safety property ϕ_S, and an over-approximation of the guards SwL.

$$\texttt{MDS} := \langle X, I, f_1, \ldots, f_k \rangle, \quad \phi_S \subseteq \mathbb{R}^n, \quad \texttt{SwL} := \langle (\mathrm{g}_{ij})_{i,j \in Q} \rangle$$

We wish to solve the problem in Def. 8 for these inputs.

Let us say we find guards g'_{ij}'s such that they have the following property: for every mode i, if a trajectory enters mode i (via any of the incoming transitions with guard g'_{ji}), then it remains

safe *until* one of the exit guards g_{ik} becomes true. This property can be written formally using the *weak until* operator.

$$\begin{aligned} Mode_1,\ I &\models \phi_S \mathbf{W}(\bigvee_{k\in Q} \mathsf{g}'_{1k}) \\ Mode_i,\ \bigvee_{j\in Q} \mathsf{g}'_{ji} &\models \phi_S \mathbf{W}(\bigvee_{k\in Q} \mathsf{g}'_{ik}) \text{ for } i = 1..k \end{aligned} \tag{6.4}$$

If the guards in the switching logic $\mathtt{SwL}'$ satisfy the collection of assertions in Equation 6.4, then the resulting hybrid system is safe. The converse is also true.

Lemma 1. *Given an* $\mathtt{MDS} := \langle X, I, f_1, \ldots, f_k \rangle$*, and a safety property* ϕ_S*, if* $\mathtt{SwL}' = \langle (\mathsf{g}'_{ij})_{i,j\in Q} \rangle$ *is a switching logic that satisfies all assertions in Equation 6.4, then the hybrid system* $\mathtt{HS} := (\mathtt{MDS}, \mathtt{SwL}')$ *is safe with respect to* ϕ_S*.*

Conversely, if there exists a switching logic $\mathtt{SwL}'$ *such that the hybrid system* $\mathtt{HS} := (\mathtt{MDS}, \mathtt{SwL}')$ *is safe with respect to* ϕ_S*, then there is a switching logic* $\mathtt{SwL}'' \subseteq \mathtt{SwL}'$ *that satisfies the assertions in Equation 6.4.*

Proof. The first part follows directly from the definition of the semantics of the temporal operators and our assumption that discrete transitions are taken as soon as they are enabled.

For the converse part, the desired $\mathtt{SwL}'' := \langle (\mathsf{g}''_{ij})_{i,j\in Q} \rangle$ is obtained by intersecting the set $Reach$ of reachable states of $\mathtt{HS}$ with $\mathtt{SwL}'$; that is, $\mathsf{g}''_{ij} := \mathsf{g}'_{ij} \cap Reach$. The reader can verify that $\mathtt{SwL}''$ will satisfy the assertions in Equation 6.4. □

6.2.1 Switching Logic Synthesis for Safety

At a semantic level, we can solve the problem in Definition 8 by computing the fixpoint of the assertions in Equation 6.4. This procedure is presented in Figure 6.5. The fixpoint iterations start by picking the most liberal guards possible (which is the intersection of the safety property and the user-specified bounds). In each successive step, the guards are made smaller by removing certain "bad" states. Specifically, we remove from g'_{ji} any state that reaches an unsafe state following the dynamics of Mode i, *before* it reaches any exit guard. Thus, we reason locally about only one mode at a time in each iteration. We stop when we reach a fixpoint.

```
SwitchSyn1(MDS, φ_S, SwL):
1  // Input MDS := ⟨X, I, f_1, ..., f_k⟩,
2  // Input φ_S ⊆ R^n,
3  // Input SwL := ⟨(g_ij)_{i,j∈Q}⟩,
4  // Output synthesis successful/failed
5  for all i, j ∈ Q do g'_ij := g_ij ∩ φ_S
6  repeat {
7   for all i ∈ Q do {
8     bad := {x | Mode_i, {x} ⊨ ¬(⋁_k g'_ik) U ¬φ_S}
9     for all j ∈ Q do g'_ji := g'_ji − bad
10    if (i == 1 and I ∩ bad ≠ ∅)
11       return "synthesis failed"
12  }
13 } until (g'_ij's do not change)
14 if (I ⇒ φ_S)
15    return "synthesis successful"
16 else return "synthesis failed"
```

Figure 6.5: Procedure for solving the switching logic synthesis problem v1.

We state the soundness and completeness of the fixpoint algorithm for solving the switching logic synthesis problem.

Lemma 2. *If Procedure* SwitchSyn1 *terminates with "synthesis successful" and* g'_{ij} *are the discovered guards, then these guards satisfy all the assertions in Equation 6.4.*

Proof. If Procedure SwitchSyn1 returns "synthesis successful", then the condition in Step 10 must be false, that is, $I \cap bad = \emptyset$. So, from the definition of bad, there does not exist $\mathbf{x} \in I$ such that $Mode_1, \{\mathbf{x}\} \models \neg(\bigvee_k \mathsf{g}'_{1k})\mathbf{U}\neg\phi_S$. For all states $\mathbf{x} \in I$,

$$Mode_1, \{\mathbf{x}\} \models \phi_S \mathbf{W}(\bigvee_{k\in Q} \mathsf{g}'_{1k})$$

that is,

$$Mode_1, I \models \phi_S \mathbf{W}(\bigvee_{k\in Q} \mathsf{g}'_{1k})$$

Also, if Procedure SWITCHSYN1 returns "synthesis successful", the termination condition for *repeat* loop at Step 13 must be true. So, for all $i \in Q$ the $\mathrm{g}'_{ji} = \mathrm{g}'_{ji} - bad$ for all $j \in Q$, that is $\mathrm{g}'_{ji} \cap bad$ is empty. So, for all $i \in Q$, there does not exist any $\mathbf{x} \in \mathrm{g}'_{ji}$ for any j such that $Mode_i, \{\mathbf{x}\} \models \neg(\bigvee_k \mathrm{g}'_{ik})\mathbf{U}\neg\phi_S$. So, for all $i \in Q$, for any state $\mathbf{x}$ in $\bigvee_{j\in Q} \mathrm{g}'_{ji}$,

$$Mode_i, \{\mathbf{x}\} \models \phi_S \mathbf{W}(\bigvee_{k\in Q} \mathrm{g}'_{ik})$$

that is,

$$Mode_i, \bigvee_{j\in Q} \mathrm{g}'_{ji} \models \phi_S \mathbf{W}(\bigvee_{k\in Q} \mathrm{g}'_{ik})$$

Thus, the discovered guards g'_{ij} satisfy all the assertions in Equation 6.4.

□

Theorem 6 (Soundness of Procedure SWITCHSYN1)**.** *If Procedure* SWITCHSYN1 *terminates with "synthesis successful" and* g'_{ij} *are the discovered guards, then the hybrid system* $\mathtt{HS} := (\mathtt{MDS}, \langle(\mathrm{g}'_{ij})_{i,j\in Q}\rangle)$ *is safe for* ϕ_S*.*

Proof. Using Lemma 1 and Lemma 2, we conclude that the hybrid system $\mathtt{HS} := (\mathtt{MDS}, \mathtt{SwL}')$ is safe with respect to ϕ_S. □

Even when it terminates with success, note that the Procedure SWITCHSYN1 does not guarantee that the synthesized hybrid system HS has nonzeno behaviors. In a post-processing step, one can perform sufficient checks to guarantee the absence of zeno behaviors.

We can also show that our procedure is complete.

Theorem 7 (Completeness of Procedure SWITCHSYN1)**.** *If Procedure* SWITCHSYN1 *terminates with "synthesis failed", then there is no* $\mathtt{SwL}' \subseteq \mathtt{SwL}$ *such that the hybrid system* $\mathtt{HS} := (\mathtt{MDS}, \mathtt{SwL}')$ *is safe.*

Proof. Assume that the claim is false and there is a switching logic $\mathtt{SwL}' \subseteq \mathtt{SwL}$ such that $\mathtt{HS} := (\mathtt{MDS}, \mathtt{SwL}')$ is safe. By Lemma 1, there is a switching logic $\mathtt{SwL}'' := Reach \cap \mathtt{SwL}'$ that satisfies Equation 6.4. Recall that $Reach$ is the set of reachable states of HS. Let $\mathtt{SwL}'_i, i = 0, 1, \ldots$, be the intermediate switching logics computed by Procedure SWITCHSYN1. Clearly, $\mathtt{SwL}'_0 \supseteq \mathtt{SwL}'_1 \supseteq$

$\mathtt{SwL}'_2 \supseteq \cdots$ and $\mathtt{SwL}'_0 := \mathtt{SwL} \cap \phi_S$. Since $Reach \subseteq \phi_S$ by assumption, we can easily verify that $\mathtt{SwL}'' \subseteq \mathtt{SwL}'_0$. We will inductively show that $\mathtt{SwL}'' \subseteq \mathtt{SwL}'_i$ for all i.

Suppose $\mathtt{SwL}'' \subseteq \mathtt{SwL}'_N$. Suppose we go from $\mathtt{SwL}'_N$ to $\mathtt{SwL}'_{N+1}$ by deleting the set bad from $\mathtt{g}'_{ji}$. We need to show that $\mathtt{SwL}'' \subseteq \mathtt{SwL}'_{N+1}$. Let $\mathtt{SwL}'_N := \langle(\mathtt{g}'_{Nij})_{i,j\in Q}\rangle$ and let $\mathtt{SwL}'' := \langle(\mathtt{g}''_{ij})_{i,j\in Q}\rangle$.

$$
\begin{aligned}
&\mathbf{x} \in bad \\
&\quad\Rightarrow\quad Mode_i, \{\mathbf{x}\} \models \neg(\bigvee_k \mathtt{g}'_{Nik})\mathbf{U}\neg\phi_S \\
&\quad\Rightarrow\quad Mode_i, \{\mathbf{x}\} \models \neg(\bigvee_k \mathtt{g}''_{ik})\mathbf{U}\neg\phi_S, \because \mathtt{g}''_{ik} \subseteq \mathtt{g}'_{Nik} \\
&\quad\Rightarrow\quad Mode_i, \{\mathbf{x}\} \models \neg(\phi_S \mathbf{W} \bigvee_k \mathtt{g}''_{ik}) \\
&\quad\Rightarrow\quad Mode_i, \{\mathbf{x}\} \not\models \phi_S \mathbf{W} \bigvee_k \mathtt{g}''_{ik} \\
&\quad\Rightarrow\quad \mathbf{x} \notin \mathtt{g}''_{ji}, \because \mathtt{SwL}'' \text{ satisfies Equation 6.4} \\
&\qquad\quad\ \mathbf{x} \notin I \text{ if } i == 1, \because \mathtt{SwL}'' \text{ satisfies Equation 6.4}
\end{aligned}
$$

This shows that $\mathtt{SwL}'' \subseteq \mathtt{SwL}'_{N+1}$ and Procedure SWITCHSYN1 cannot return at Line 11. Since HS is assumed to be safe, $I \Rightarrow \phi_S$ and hence Procedure SWITCHSYN1 cannot return at Line 16. Hence, Procedure SWITCHSYN1 can only return "synthesis successful" contradicting our assumption. □

6.2.2 Switching Logic Synthesis for Safety and Dwell-time

We now consider the switching logic synthesis problem in Definition 9. Recall that apart from the bounds on the guards, the user can provide minimum and maximum dwell time requirements for each mode. The goal is to synthesize a switching logic where the guards satisfy the specified bounds and the trajectories of the resulting hybrid system satisfy the minimum and maximum dwell time requirements.

Procedure SWITCHSYN2 for solving the problem in Definition 9 is outlined in Figure 6.6. Procedure SWITCHSYN2 runs in three phases. In the first step, the new problem is transformed to the old problem. In the second step, Procedure SWITCHSYN1 is used to solve the generated problem. In the third step, the result is transformed back to get a result of the given problem.

Suppose that we are given

$$\texttt{MDS} := \langle X, I, f_1, \ldots, f_k \rangle, \;\; \phi_S \subseteq \mathbb{R}^n, \;\; \texttt{SwL} := \langle (\mathsf{g}_{ij})_{i,j \in Q} \rangle,$$
$$\texttt{Te} := \{te_1, \ldots, te_k\}, \;\; \texttt{Tx} := \{tx_1, \ldots, tx_k\}$$

In the first step, the problem in Definition 9 is reduced to the previous problem. This reduction is achieved by introducing a new state variable t such that
(1) the dynamics of t is given by $\dot{t} = 1$ in each mode
(2) the variable t is reset to 0 in each discrete transition
These two steps are performed by the function `Add_timer_t`. Now, the dwell time requirements can be specified as bounds on the variable t. Specifically, the over-approximation `SwL` of the guards can be updated as follows:

$$\mathsf{g}_{ij} \;\; := \;\; \mathsf{g}_{ij} \wedge (te_i \leq t \leq tx_i)$$

In the second step, a call to Procedure SWITCHSYN1 is made, but with the updated `SwL`. Recall that Procedure SWITCHSYN1 essentially performs an iterative fixpoint computation to solve Equation 6.4. Equation 6.4 assumes that discrete transitions do not reset any continuous variables. Since we now have discrete transitions that reset t to 0, we need a slightly modified Procedure SWITCHSYN1 that solves the modified equations below:

$$\begin{aligned} Mode_1, R(I) \;\; &\models \;\; \phi_S \mathbf{W} \bigvee_{k \in Q} \mathsf{g}'_{1k} \\ Mode_i, \bigvee_{j \in Q} R(\mathsf{g}'_{ji}) \;\; &\models \;\; \phi_S \mathbf{W} \bigvee_{k \in Q} \mathsf{g}'_{ik} \;\; \text{for } i = 1..k \end{aligned} \tag{6.5}$$

where $R(S)$ is the set of states obtained by resetting the t-component of every state in the set S to 0. If ϕ is a formula denoting the set S, then $R(\phi)$ is $\exists s(\phi[s/t] \wedge t = 0)$ (the notation $\phi[s/t]$ means replace t by s in ϕ). Informally, $R(\phi)$ can be computed by first removing facts about t from ϕ and then adding the new fact $t = 0$ to it.

The guards synthesized by Procedure SWITCHSYN1 will use the new state variable t. However t was not part of our original problem specification. In the third step, the variable t is eliminated from the guards synthesized by Procedure SWITCHSYN1. Suppose $\texttt{SwL}' := \langle (\mathsf{g}'_{ij})_{i,j \in Q} \rangle$ is the switching logic synthesized by Procedure SWITCHSYN1. We first project out the t-component from $\texttt{SwL}'$ to get our first guess for the desired `SwL`. Then, for every mode i, and for each entry

guard, say $\mathtt{g}'_{ji}$, and for each exit guard $\mathtt{g}'_{ik}$, we compute pairs of states $(\mathbf{x}, \mathbf{x}')$ such that $\mathbf{x} \in \mathtt{g}_{ji}$, $\mathbf{x}' \in \mathtt{g}_{ik}$, there is a trajectory in mode i that starts from state $\langle \mathbf{x}, t = 0 \rangle$ and reaches $\langle \mathbf{x}', t' \rangle$ in time t', and $\langle \mathbf{x}', t' \rangle$ is not in $\mathtt{g}'_{ik}$. A behavior where mode i in entered in state $\mathbf{x}$ and exited in $\mathbf{x}'$ was disallowed in $\mathtt{SwL}'$, but it is allowed in $\mathtt{SwL}$ (since $\mathtt{SwL}$ ignores t). Hence, we need to either remove $\mathbf{x}$ from $\mathtt{g}_{ji}$, or remove $\mathbf{x}'$ from $\mathtt{g}_{ik}$. Procedure SWITCHSYN2 procedure non-deterministically makes this choice.

Removal of states from the guards can potentially cause the modified switching logic to become unsafe. Hence, in the final step, we need to verify that the updated guards still satisfy Equation 6.4. This is performed by the function `Verify`. The function `Verify` can be implemented by calling Procedure SWITCHSYN1 and checking its return value.

We can now state the soundness and completeness of Procedure SWITCHSYN2 for solving the switching logic synthesis problem in Def. 9.

Theorem 8 (Soundness of Procedure SWITCHSYN2)**.** *If Procedure* SWITCHSYN2 *terminates with "synthesis successful" and* $\mathtt{g}'_{ij}$ *are the discovered guards, then the hybrid system* $\mathtt{HS} := (\mathtt{MDS}, \langle (\mathtt{g}'_{ij})_{i,j \in Q} \rangle)$ *is safe for* ϕ_S *and it satisfies the dwell time requirements specified by* `Te` *and* `Tx`*.*

Proof. The final `Verify` check guarantees that `HS` is safe with respect to ϕ_S. The over-approximation defined in Step 6 of Procedure SWITCHSYN2 ensures that the switching logic $\mathtt{SwL}'$ synthesized by Procedure SWITCHSYN1 on Line 8 satisfies the dwell time requirements. From Theorem 6, the guards also satisfy the following.

$$M_i, \bigvee_{j \in Q} \mathtt{g}'_{ji} \models \phi_S \mathbf{W} (\bigvee_{k \in Q} \mathtt{g}'_{ik})$$

From Step 16, it follows that

$$M_i, \bigvee_{j \in Q} \mathtt{g}'_{ji} \models (\neg \bigvee_{k \in Q} \mathtt{g}'_{ik}) \mathbf{W} (t \geq te_i)$$

So, the guards g'_{ij}'s synthesized by Procedure SWITCHSYN2 satisfy the following assertions

$$M_i, \bigvee_{j \in Q} \mathtt{g}'_{ji} \models (\phi_S \wedge \neg \bigvee_{k \in Q} \mathtt{g}'_{ik}) \mathbf{W} (t \geq te_i) \quad \text{for all } i \in Q \tag{6.6}$$

So, the synthesized guards define a switching logic that satisfies the requirements in Problem Definition 9. □

SwitchSyn2(MDS, ϕ_S, SwL, Te, Tx):

1 // Input MDS, ϕ_S, SwL: As in Figure 6.5
2 // Input Te := $\langle te_1, \ldots, te_k \rangle$
3 // Input Tx := $\langle tx_1, \ldots, tx_k \rangle$
4 // Output synth. successful/failed
5 MDS^e := Add_timer_t(MDS)
6 SwL^e := $\langle (\mathtt{g}_{ij} \wedge (te_i \leq t \leq tx_i))_{i,j \in Q} \rangle$
7 // Call SwitchSyn1 with the updated SwL
8 *res* := SwitchSyn1(MDS^e, ϕ_S, SwL^e)
9 if *res* == "synthesis failed"
10 return "synthesis failed"
11 else let SwL$'$ be the synthesized guards
12 // post processing step
13 for all $i, j \in Q$ do
14 $\mathtt{g}_{ij} := \{\mathbf{x} \mid \langle \mathbf{x}, t \rangle \in \mathtt{g}'_{ij}\}$
15 for all $i, j, k \in Q$ do {
16 *bad* := $\{\langle \mathbf{x}, \mathbf{x}' \rangle \mid \mathbf{x} \in \mathtt{g}_{ji} \wedge \mathbf{x}' \in \mathtt{g}_{ik} \wedge \langle \mathbf{x}', t' \rangle \notin \mathtt{g}'_{ik}$
17 $\wedge\, M_i, \{\langle \mathbf{x}, t = 0 \rangle\} \models \mathit{true}\mathbf{U}\{\langle \mathbf{x}', t' \rangle\}\}$
18 Guess B_1, B_2 s.t. $B_1 \times B_2 \supseteq \mathit{bad}$
19 $g_{ji} := g_{ji} - B_1$; $g_{ik} := g_{ik} - B_2$
20 }
21 if (Verify(MDS, ϕ_S, $\langle (\mathtt{g}_{ij})_{i,j \in Q} \rangle$))
22 return "synthesis successful"
23 else return "synthesis failed"

Figure 6.6: Procedure for solving the switching logic synthesis problem v2.

We can also state and prove completeness of Procedure SwitchSyn2.

Theorem 9 (Completeness of Procedure SwitchSyn2)**.** *If, for every possible guess on Line 18, the Procedure* SwitchSyn2 *terminates with "synthesis failed", then there is no* SwL$' \subseteq$ SwL *such that the hybrid system* HS := (MDS, SwL$'$) *is safe and it satisfies the dwell time requirements.*

Proof. The completeness of the algorithm follows from the non-deterministic guesses in Step 18. The Procedure SWITCHSYN2 first transforms the problem to an extended MDS and uses Procedure SWITCHSYN1 to compute the guards (Step 8). By Theorem 7, we know that Procedure SWITCHSYN1 is complete. So, if there is a switching logic that produces a MDS which is safe and satisfies dwell-time properties, then guards computed in Step 8 will contain this switching logic. Hence, the only place where completeness might be compromised is in the post-processing step. However, we make non-deterministic guesses for removing states from the guards computed by Procedure SWITCHSYN1 and hence, if a solution exists, we can always guess the correct sets to be substracted from the computed guards in Step 18 such that we obtain the desired solution. This gives us the desired completeness result. □

Procedure SWITCHSYN2 is nondeterministic and involves making the correct guesses in the postprocessing stage. We can get a deterministic version of the procedure by making arbitrary guesses at each point. This deterministic version will be sound: whenever the procedure outputs "synthesis successful", the synthesis problem in Definition 9 indeed has a positive answer. However, it will not be complete: even when there is a positive answer for the synthesis problem, the deterministic variant can fail to find the appropriate guards because it can make the wrong choices. Some form of backtracking appears to be required. In practice, our implementation's heuristically-guided choices have always obtained a positive answer.

6.2.3 Guards from Simulations

A key step in the implementation of Algorithms SWITCHSYN1 and SWITCHSYN2 is the computation of the bad state sets. In general, since the mode dynamics can be non-linear and quite complex, exactly computing the bad sets through analytical means is computationally infeasible. However, it is easier to perform numerical simulation of even complex, non-linear dynamics from individual points. In particular, in many cases, numerical simulation can be used to check whether a point $\mathbf{x}$ is a member of bad. Given such a membership check, our approach uses machine learning to compute an over-approximation of bad. While such over-approximation can result in a loss of completeness, it is guaranteed to generate safe switching logic.

Machine Learning

Our procedure assumes the availability of a machine learning algorithm $\mathcal{L}$ that can learn any target set from a *concept class* $\mathcal{C}$. $\mathcal{L}$ uses an oracle that can label points $\mathbf{x}$ as being in the target concept (i.e., $\mathbf{x} \in bad$) or not in it (i.e. $\mathbf{x} \notin bad$).

$\mathcal{L}$ is parameterized by $\mathcal{C}$, a point we sometimes make explicit by writing $\mathcal{L}_{\mathcal{C}}$ rather than $\mathcal{L}$.

Formally, given the following three inputs: (i) an over-approximation $\overline{c} \in \mathcal{C}$ of the set bad; (ii) a simulation oracle that can label a point $\mathbf{x}$ as $\mathbf{x} \in bad$ or $\mathbf{x} \notin bad$; and (iii) (optionally) a sample of examples $P \subseteq bad$ (if they exist), $\mathcal{L}_{\mathcal{C}}$ must generate as output a set $out_{\mathcal{L}} \in \mathcal{C}$ with the following properties: if $bad \in \mathcal{C}$, then $out_{\mathcal{L}} = bad$; otherwise, $out_{\mathcal{L}} \supseteq bad$.

For simplicity, we first describe below how $\mathcal{L}$ can be implemented when bad is an interval constraint on a single variable. It is also possible to extend this method to conjunctions of interval constraints on multiple variables. Conjunctions of interval constraints define hyperboxes.

Simulation Oracles

We assume the availability of the following two kinds of simulation-based oracles:

- *Oracle $\mathcal{SO}_A$:* This is an oracle that, given a state $\mathbf{x}$, the dynamics of a mode $Mode_i$, and state sets ϕ_1 and ϕ_2, returns a Boolean answer indicating whether the following property holds:

  $$Mode_i, \{\mathbf{x}\} \models (\phi_1 \mathbf{U} \phi_2)$$

  Note that definition of $\mathcal{SO}_A$ is motivated by the need to compute bad in Line 8 of Procedure SWITCHSYN1.
- *Oracle $\mathcal{SO}_B$:* This is an oracle that, given a state pair $\langle \mathbf{x}, \mathbf{x}' \rangle$, the dynamics of a mode $Mode_i$, extended-state set ψ, and state sets ϕ_1 and ϕ_2, returns a Boolean answer indicating whether the following property holds:

  $$\begin{aligned} & \mathbf{x} \in \phi_1 \wedge \mathbf{x}' \in \phi_2 \wedge \langle \mathbf{x}', t' \rangle \notin \psi \\ \wedge \quad & Mode_i, \{\langle \mathbf{x}, 0 \rangle\} \models (true \mathbf{U} \langle \mathbf{x}', t' \rangle) \end{aligned}$$

  The definition of $\mathcal{SO}_B$ is motivated by the need to compute bad in Line 17 of Procedure SWITCHSYN2.

Implementing these oracles involves performing a simulation from state $\mathbf{x}$ according to the (deterministic) dynamics in $Mode_i$, checking whether the condition on the RHS of the $\mathbf{U}$ operator has become true, and if not, checking that the LHS condition remains true. We assume the presence of a numerical simulator that can, for the mode dynamics of interest, select an appropriate discretization of time so as to check the above formulas with the $\mathbf{U}$ operator.

Learning Interval Constraints

We now describe how one can implement $\mathcal{L}$ for learning an interval constraint over a single variable $x \in X$. We give conditions under which the algorithm presented here satisfies the conditions required of $\mathcal{L}$ as stated above.

An interval constraint is of the form $x \in [l_i, u_i]$ where $l_i, u_i \in \mathbb{Q}$. This constraint can also be expressed using inequalities as $l_i \leq x \leq u_i$.

Thus, the concept class $\mathcal{I}$ is the set of all constraints of the form $x \in [l_i, u_i]$ for any $l_i, u_i \in \mathbb{Q}$ and for any $x \in X$. The initial over-approximation $\overline{c}$ and the set $out_{\mathcal{L}}$ generated by $\mathcal{L}$ are both representable as an interval constraint. Algorithm $\mathcal{L}_{\mathcal{I}}$ to learn interval constraints is given below in Procedure 15.

Algorithm $\mathcal{L}_{\mathcal{I}}$ begins by checking the end-points of $\overline{c} = [\overline{l}, \overline{u}]$ for membership in bad. If both $\overline{l}$ and $\overline{u}$ are in bad, it simply outputs $out_{\mathcal{L}} = \overline{c}$. Otherwise, it selects the minimum and maximum elements $x_{\min}$ and $x_{\max}$ in the set of examples $P \in bad$. (If P is not provided as input, $\mathcal{L}$ will randomly sample elements of $out_{\mathcal{L}}$ until an example $P \in bad$ is found).
We assume that the interval $[\overline{l}, \overline{u}]$ can be suitably discretized so that the extreme points of bad are members of this discretized set of points. Since guards are implemented using finite-precision software, this assumption is not restrictive. $\mathcal{L}$ then performs binary search in the ranges $[\overline{l}, x_{\min}]$ and $[x_{\max}, \overline{u}]$ until it finds two examples $x_l \in [\overline{l}, x_{\min}]$ and $x_u \in [x_{\max}, \overline{u}]$ such that $x_l, x_u \in bad$ where x_l is the smallest such point and x_u is the largest. It then outputs $out_{\mathcal{L}} = [x_l, x_u]$.

It is easy to see that if $bad \in \mathcal{C}$, then $out_{\mathcal{L}} = bad$.

However, if $bad \notin \mathcal{C}$, then bad must be a disjoint union of intervals. Under the condition that P contains one point from each interval in this union, we obtain $out_{\mathcal{L}} \supset bad$.

Alternatively, suppose that the dynamics within each mode i is such that each state variable

Procedure 15 Learning Interval Constraints

Input: An over-approximation $\overline{c} = [\overline{l}, \overline{u}]$ of the set bad,

A simulation oracle that can label a point $\mathbf{x}$ as $\mathbf{x} \in bad$ or $\mathbf{x} \notin bad$,

$P \subseteq bad$

Precision ϵ

Output: The target concept bad which belongs to the class $\mathcal{I}$

$l = min(P)$

$u = max(P)$

while $l - \overline{l} < \epsilon$ **do**

$m = \frac{\overline{l}+l}{2}$

if $m \in bad$ **then**

$l = m$

else

$\overline{l} = m$

end if

end while

while $\overline{u} - u < \epsilon$ **do**

$m = \frac{\overline{l}+u}{2}$

if $m \in bad$ **then**

$u = m$

else

$\overline{u} = m$

end if

end while

evolves monotonically with time – i.e., its value within that mode either increases with time or it decreases, but not both. In this case, bad cannot be a disjoint union of intervals, and so $out_{\mathcal{L}} = bad$.

Learning Hyperboxes

It is possible to extend the above procedure to learn interval constraints to learn a *conjunction of interval constraints*, viz., where $\mathcal{C}$ is the set of all n-dimensional boxes in $\mathbb{R}^n$. The extension is based on identifying diagonally-opposite corners of the n-dimensional hyperbox. The diagonally opposite corners of the hyperbox can be identified using binary search from the corners of the starting overapproximate hyperbox, assuming points in the hyperbox can be labeled as safe/unsafe (positive/negative) using the simulation oracle. The search terminates when we have found the lower and upper diagonal corners as positive examples with their *immediate outer neighbours* as negative examples; for further details, see the hyperbox learning problem discussed by Goldman and Kearns [41]. In the case that bad is not of this form, an over-approximation is obtained by applying the Procedure $\mathcal{L}_{\mathcal{I}}$ to each $x \in X$ separately and taking the disjunction of the generated intervals. The same technique can be used to directly learn $good$ guards if $good$ is known to be an hyperbox.

Example of Learning Guards

We illustrate this with an example from our experiments. For the thermostat example in Figure 6.1, the room temperature x varies monotonically in the heating mode. We also start with an over-approximation for the guard from off to heating mode g_{FH} that $g_{FH} \subseteq 18 \leq x \leq 20 \wedge T = 20$. We query the simulation oracle $\mathcal{SO}_A$ at $x = 18.00$ and $x = 20.00$. $\mathcal{SO}_A$ returns 'yes' for $x = 18$ indicating that the evolution from $x = 18$ is safe, but it returns 'no' for $x = 20$. We can then perform a binary search for the revised end point of the interval. $\mathcal{SO}_A$ also returns 'no' for $x = 19.96$ and 'yes' for $x = 19.95$, then we know (e.g., by monotonicity) that it will return 'no' for all $x \in [19.96, 20]$. So, we revise the over-approximation of the guard to $g_{FH} \subseteq 18 \leq x \leq 19.95 \wedge T = 20$ at the end of the first iteration.

Iter.	l_{FH}, u_{FH}	l_{HN}, u_{HN}	l_{NC}, u_{NC}	l_{CF}, u_{CF}
0	$18.00, 20.00$	$18.00, 20.00$	$18.00, 20.00$	$18.00, 20.00$
1	$18.00, 19.95$	$18.00, 20.00$	$18.00, 19.95$	$18.00, 20.00$
2	$18.00, 19.95$	$18.00, 19.95$	$18.00, 19.95$	$18.00, 20.00$
3	$18.00, 19.90$	$18.00, 19.95$	$18.00, 19.95$	$18.00, 20.00$
4	$18.00, 19.90$	$18.00, 19.95$	$18.00, 19.95$	$18.00, 20.00$

Table 6.1: Steps of Fixpoint Computation for Thermostat v1

6.3 Results and Experiments

We have implemented our technique using a Matlab-based numerical simulator. Here we present three case studies to illustrate how our technique can be used in practice to synthesize switching logic for multi-modal continuous dynamical systems. For the *Thermostat Controller* described earlier in Section 6.1.3, we give only the intermediate steps of our approach. For two other case studies, we describe synthesis problems and present its solution obtained by our technique.

6.3.1 Thermostat Controller

This example is described in Section 6.1.3 with the results we obtained. Here, we only briefly explain how the final guards were obtained.

Table 6.1 shows the intermediate steps of the fixpoint computation, indicating how guards shrink in each iteration of the algorithm. In the first iteration, the reduction of u_{FH} and u_{NC} to 19.95 occurs as the system must spend some time in the HEATING mode as T goes from 20 to 22, and during that period x cannot increase beyond 20. Thus, a simulation from $x = 19.96$ for example, would reach an unsafe state. The subsequent iterations propagate the restrictions on the exit guards of modes (e.g., u_{NC} for ON) to apply to the entry guards to those modes (e.g., u_{HN}).

Similarly, for the synthesis problems with dwell-time constraints, we show intermediate steps of the fixpoint computation in Tables 6.2 and 6.3. Consider Table 6.2. One can observe the impact of the min-dwell-time constraint in the value of l_{NC} and l_{CF} in iteration 1, where the need to spend at least 100 sec. in the OFF mode causes the controller to switch to COOLING or OFF only when

Iter.	l_{FH}, u_{FH}	l_{HN}, u_{HN}	l_{NC}, u_{NC}	l_{CF}, u_{CF}
0	$18.00, 20.00$	$18.00, 20.00$	$18.00, 20.00$	$18.00, 20.00$
1	$18.00, 19.95$	$18.00, 20.00$	$18.35, 19.95$	$18.45, 20.00$
2	$18.00, 19.95$	$18.00, 19.95$	$18.35, 19.95$	$18.45, 20.00$
3	$18.00, 19.90$	$18.00, 19.95$	$18.35, 19.95$	$18.45, 20.00$
4	$18.00, 19.90$	$18.00, 19.95$	$18.35, 19.95$	$18.45, 20.00$

Table 6.2: Steps of Fixpoint Computation for Thermostat v2 Case A

Iter.	l_{FH}, u_{FH}	l_{HN}, u_{HN}	l_{NC}, u_{NC}	l_{CF}, u_{CF}
0	$18.00, 20.00$	$18.00, 20.00$	$18.00, 20.00$	$18.00, 20.00$
1	$18.00, 19.95$	$18.00, 18.35$	$19.60, 19.95$	$19.65, 20.00$
2	$18.00, 18.14$	$18.00, 18.26$	$19.60, 19.95$	$19.65, 20.00$
3	$18.00, 18.14$	$18.00, 18.26$	$19.60, 19.95$	$19.65, 20.00$

Table 6.3: Steps of Fixpoint Computation for Thermostat v2 Case B

the temperature is higher than 18.35. Similarly, for the last problem (see Table 6.3), imposing the min-dwell-time constraint on the ON mode causes the lower bound l_{NC} to be higher.

For the problem Thermostat v2 Case B, we can additionally restrict g_{NC} and g_{FH} using the post-processing step in the algorithm described in Figure 6.6. The final synthesized guards are then as follows.

$$g_{FH} : 18.00 \leq y \leq 18.01 \wedge T = 20$$

$$g_{HN} : 18.00 \leq y \leq 18.26 \wedge T = 22$$

$$g_{NC} : 19.94 \leq y \leq 19.95 \wedge T = 22$$

$$g_{CF} : 19.65 \leq y \leq 20.00 \wedge T = 20$$

6.3.2 Traffic Collision and Avoidance System

Consider a simplified version of the Traffic Collision and Avoidance System (TCAS) [103], which seeks to ensure that two planes flying in opposite directions do not collide and maintain a specified safe distance (200 meters in our example). It operates by guiding the planes through a turn-left/fly-straight/turn-right maneuver as shown in the Figure 6.7. The three recovery maneuvers are indicated by corresponding mode names. We need to synthesize switching logic between

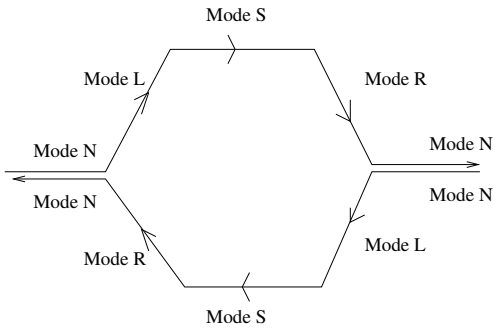


Figure 6.7: Simplified Traffic Collision and Avoidance System

the modes such that the planes are always at least 200 meters apart at all times.

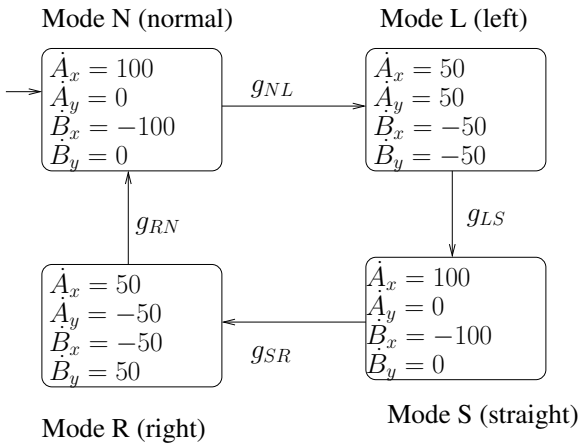


Figure 6.8: Simplified Traffic Collision and Avoidance System

The dynamics of the four modes of TCAS are given in Figure 6.7. We limit the movement of the plane in 2 dimensions $(X - Y)$ to simplify the example. Let $(\dot{A}_x, \dot{A}_y)$, and $(\dot{B}_x, \dot{B}_y)$ denote the (X, Y) velocities of the two planes A and B. Let $d(A, B)$ denote the Euclidean distance between the two planes, that is, $d(A, B) = \sqrt{(A_x - B_x)^2 + (A_y - B_y)^2}$. Hence we have the following safety property: $d(A, B) \geq 200$. In addition to this safety property, we also require that the planes at the end of the maneuver must regain their original orientation, that is, along the X-axis. So, $A_y = 0$ and $B_y = 0$ when returning to the normal mode at the end of the maneuver. Further, we would like to switch away from the straight mode only after the planes have crossed each other, that is, $A_x - B_x > 0$. We initialize the guards as given in Equation 6.7 using the safety property

and the other specifications mentioned above.

$$\begin{aligned} g^0_{NL} &: d(A,B) \geq 200 \\ g^0_{LS} &: d(A,B) \geq 200 \\ g^0_{SR} &: d(A,B) \geq 200 \wedge A_x - B_x > 0 \\ g^0_{RN} &: d(A,B) \geq 200 \wedge A_y = 0 \wedge B_y = 0 \end{aligned} \tag{6.7}$$

Consider two cases for the synthesis problem - one with just the minimum dwell-time constraint and the second with both the minimum and the maximum dwell-time constraint. This example illustrates how designers can use maximum dwell-time constraints to synthesize systems with desired behavior and not just safe behavior.

Case A: Only a minimum dwell-time requirement of 1 second in the straight mode is provided, ensuring that the planes spend some time in the straight mode before turning again. The final guards synthesized by computing fixpoint are as follows.

$$\begin{aligned} g_{NL} &: g^0_{NL} \wedge B_x - A_x \geq 283 \\ g_{LS} &: g^0_{LS} \wedge A_y - B_y \geq 200 \\ g_{SR} &: g^0_{SR} \wedge A_x - B_x \geq 117 \\ g_{RN} &: g^0_{RN} \wedge (A_x - B_x \geq 0 \vee B_x - A_x \geq 283) \end{aligned} \tag{6.8}$$

The behavior of the system synthesized above is illustrated in Figure 6.9. The initial state is $A_x = 0, A_y = 0, B_x = 600, B_y = 0$. X and Y denote the distance between the planes in X and Y co-ordinates and D denotes the distance between the planes. The minimum value of D is 200m. The synthesized system is safe and satisfies the minimum dwell-time requirement but it has the undesirable behavior of switching from normal mode to maneuver modes immediately at the initial state. The planes could have delayed their entry into the maneuver mode.

Case B: In this case, we also provide a maximum dwell-time requirement of 1.1 second in the straight mode. This ensures that the planes fly towards each other till it is necessary to switch to maneuver modes. By specifying the maximum dwell-time requirement on the straight mode, we effectively limit the time spend in maneuver and hence, force the system to stay in the normal

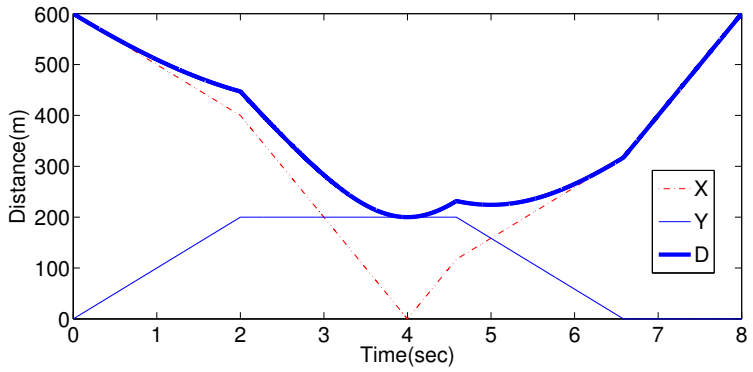


Figure 6.9: Sample Behavior of Synthesized TCAS

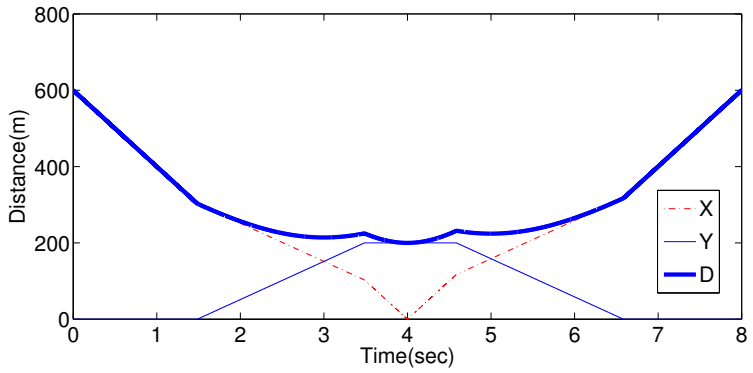


Figure 6.10: Sample Behavior of Synthesized TCAS with Max Dwell-time

mode for a longer time. The final guards synthesized by computing the fixpoint are as follows.

$$\begin{aligned} g_{NL} &: g^0_{NL} \wedge 303 \geq B_x - A_x \geq 283 \\ g_{LS} &: g^0_{LS} \wedge A_y - B_y \geq 200 \wedge B_x - A_x \leq 103 \\ g_{SR} &: g^0_{SR} \wedge A_x - B_x \geq 117 \\ g_{RN} &: g^0_{RN} \wedge (A_x - B_x \geq 0 \vee B_x - A_x \geq 283) \end{aligned} \tag{6.9}$$

We again plot the behavior of the synthesized system with the same initial state as Case A in Figure 6.10. The time spent in maneuver is now limited and we stay in normal mode till the planes are 303 meters far from each other and then switch to the collision avoidance maneuver.

6.3.3 Automatic Transmission

Our next example is a 3-gear *automated transmission* system [81]. The transmission system is illustrated in Figure 6.11; notice that *the mode dynamics are non-linear*. u and d denote the throttle in accelerating and deaccelerating mode. The transmission efficiency η is η_i when the system is in mode i.

$$\eta_i = 0.99e^{-(\omega - a_i)^2/64} + 0.01$$

where $a_1 = 10, a_2 = 20, a_3 = 30$ and ω is the speed. The distance covered is denoted by θ. The acceleration in mode i is given by the product of the throttle and transmission efficiency. For simplicity, we fix $u = 1$ and $d = -1$. From an initial state of $\theta = 0, \omega = 0$, the system must reach $\theta = \theta_{max} = 1700$ with $\omega = 0$. The synthesis problem is to find the guards between the modes such that the efficiency η is high for speeds greater than some threshold, that is, $\omega \geq 5 \Rightarrow \eta \geq 0.5$. Also, ω must be less than an upper limit of 60. So, the safety property ϕ_S to be enforced would be

$$(\omega \geq 5 \Rightarrow \eta \geq 0.5) \wedge (0 \leq \omega \leq 60)$$

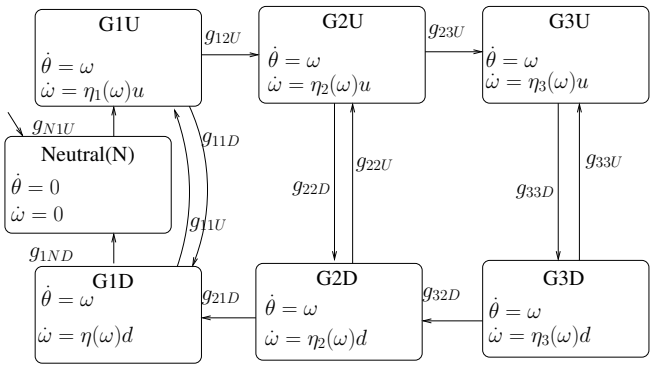


Figure 6.11: Automatic Transmission System

Since the speed must reduce to 0 on reaching θ_{max}, the guard g_{1ND} is initialized to $\phi_S \wedge \theta = \theta_{max} \wedge \omega = 0$. All the other guards are initialized to ϕ_S.

The final set of guards obtained after fixpoint computation are as follows.

$$\begin{aligned} g_{N1U}, g_{11U} &: 0 \leq \omega \leq 16.70 \\ g_{12U}, g_{22U} &: 13.29 \leq \omega \leq 26.70 \\ g_{23U}, g_{33U} &: 23.29 \leq \omega \leq 36.70 \ , \ g_{33D} : 23.29 \leq \omega \leq 36.70 \\ g_{32D}, g_{22D} &: 13.29 \leq \omega \leq 26.70 \\ g_{21D}, g_{11D} &: 0 \leq \omega \leq 16.70 \ , \ ; g_{1ND} : \theta = \theta_{max} \wedge \omega = 0 \end{aligned} \tag{6.10}$$

We now impose a minimum dwell-time of 5 seconds on all the six gear modes. The guards obtained by computing the fixpoint are as follows.

$$\begin{aligned} g_{N1U} &: \omega = 0 \ , \ g_{11U} : \omega = 0 \\ g_{1ND} &: \theta = \theta_{max} \wedge \omega = 0 \ , \ g_{12U} : 13.29 \leq \omega \leq 23.42 \\ g_{11D} &: 1.31 \leq \omega \leq 16.70 \ , \ g_{23U} : 26.70 \leq \omega \leq 33.42 \\ g_{22D} &: \omega = 26.70 \ , \ g_{33D} : \omega = 36.70 \\ g_{32D} &: 16.58 \leq \omega \leq 26.70 \ , \ g_{33U} : 23.29 \leq \omega \leq 33.42 \\ g_{21D} &: 1.31 \leq \omega \leq 16.70 \ , \ g_{22U} : 13.29 \leq \omega = 23.42 \end{aligned} \tag{6.11}$$

The plot of the behavior of the transmission system when it is made to switch from Neutral mode through the six gear modes and back to the Neutral mode is shown in Figure 6.12. The efficiency η is always greater than 0.5 when the speed is higher than 5 and we spend atleast 5 seconds in the six gear modes. Starting from $\theta = 0, \omega = 0$, the synthesized system reaches $\theta = \theta_{max}$ with $\omega = 0$.

6.3.4 Train Gate Controller

The example is a four mode train-gate controller system illustrated in Figure 6.13. The purpose of the train gate controller is to close the gate when the train approaches and to open the gate when the train has passed.

The system has two variables $\{d, a\}$ where d is the distance of the train from the gate and a is the angle of the gate. The distance d is negative when the train is approaching the train and is positive when the train has passed. The speed of the train is constant, $\dot{d} = 40$ m/s. The gate

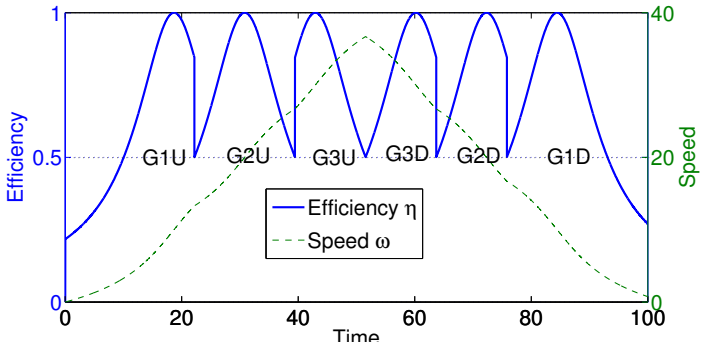


Figure 6.12: Transmission efficiency and speed with changing gears

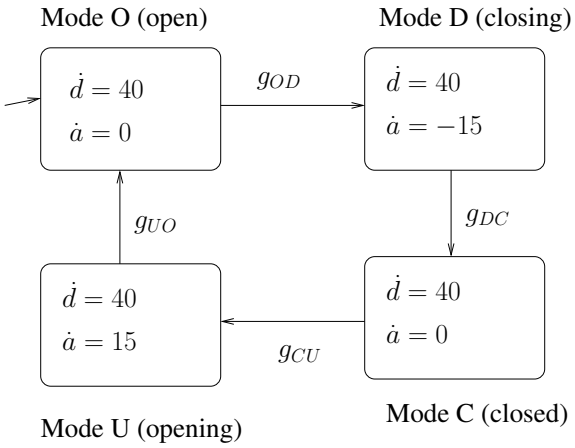


Figure 6.13: Train Gate Control System

closes or opens at a constant rate of $\dot{a} = 15$ degrees/sec. The controller has four modes - open (O), closing (D), closed (C) and opening (U), that is, $Q = \{O, D, C, U\}$. The continuous dynamical system in each mode are described by a set of ordinary differential equations as illustrated in Figure 6.13.

Starting from an open gate mode, the controller will eventually start closing the gate when the train approaches and eventually train would be closed. The gate would start opening after the train has passed and would reach the open mode. The gate closes at 0 degree and the gate opens to 90 degrees. So, closing mode takes a from 90 degrees to 0 degree and the opening mode takes a from 0 degree to 90 degrees. The mode switch from mode C happens only when the train has atleast reached the gate, that is, $d \geq 0$.

In order for the above system to be safe, we would like to enforce the following safety property ϕ_S

$$-50 < d < 50 \iff a = 0$$

that is, when the train is within 50 metres from the gate, the gate remains closed. We need to synthesize the switching logic `SwL` such that the above property is ensured in all reachable states.

Case A: We initialize the guards using the safety property and other constraints on mode switches mentioned above.

$$\begin{aligned} g_{OD} : \phi_S \ , \ g_{DC} : \phi_S \wedge (a = 0) \\ g_{CU} : \phi_S \wedge d \geq 0 \ , \ g_{UO} : \phi_S \wedge (a = 90) \end{aligned} \tag{6.12}$$

Solving the fixpoint equations yields the following guards.

$$\begin{aligned} g_{OD} : d \leq -290 \ , \ g_{DC} : d \leq -50 \wedge (a = 0) \\ g_{CU} : d \geq 50 \ , \ g_{UO} : (d \geq 50 \vee d \leq -290) \wedge (a = 90) \end{aligned} \tag{6.13}$$

The behavior of the system is shown in Figure 6.14.

Case B: We consider a max-dwell time of 5 seconds in the close mode C. The

$$\begin{aligned} g_{OD} : \phi_S \ , \ g_{DC} : \phi_S \wedge (a = 0) \\ g_{CU} : \phi_S \wedge t \leq 5 \wedge d \geq 0 \ , \ g_{UO} : \phi_S \wedge (a = 90) \end{aligned} \tag{6.14}$$

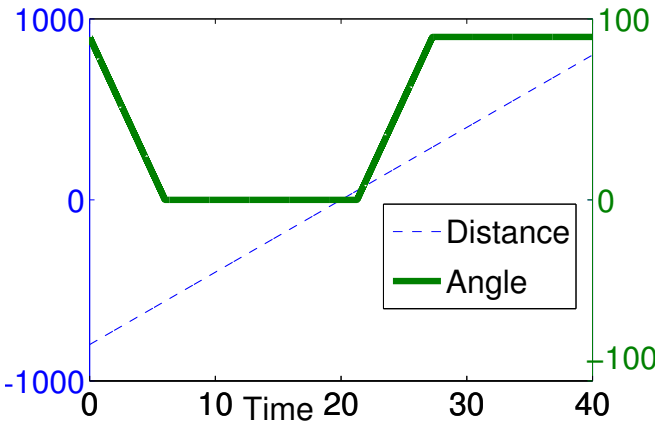


Figure 6.14: Sample Behavior of Synthesized Gate Controller

where t denotes the time spent in the closed mode C. The guards synthesized using fixpoint computation are as follows. The behavior of the system is shown in Figure 6.15.

$$\begin{aligned} g_{OD} &: (d \leq -290 \wedge d \geq -390) \\ g_{DC} &: (d \leq -50 \wedge d \geq -150) \wedge (a = 0) \\ g_{CU} &: d \geq 50 \\ g_{UO} &: (d \geq 50 \vee d \leq -290) \wedge (a = 90) \end{aligned} \tag{6.15}$$

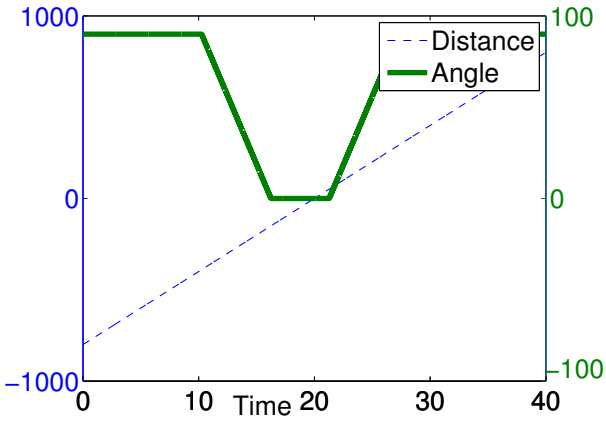


Figure 6.15: Sample Behavior of Synthesized Gate Controller with Max-dwell Time

6.3.5 Performance

We summarize the number of iterations needed to reach the fixed point and total runtime in Table below. The total runtime includes the time to obtain simulation traces from different initial states, time to label these traces as good or bad and the time to synthesize the new guards cumulative over all the iterations.

Example	# of Iterations	Runtime (seconds)
Thermostat Controller		
v1	5	21.6
v2 Case A	6	26.2
v2 Case B	6	25.7
TCAS		
Case A	4	55.3
Case B	5	59.1
Automatic Transmission	6	83.6
Train Gate Controller		
Case A	3	22.5
Case B	4	28.3

6.4 Discussion

We make some remarks on the synthesis procedure. First, note that restricting $out_{\mathcal{L}}$ to be an interval constraint does not require the final guards to also be of this form, since the designer is free to specify a *starting* switching logic using arbitrary expression syntax. The restriction only means that the set of points *removed* from the guards at each iteration of the fixpoint computation must be representable as an interval constraint to avoid losing completeness by removing too many points. As we demonstrate in our experimental results, we are able to synthesize interesting and non-trivial switching logic in spite of this restriction to the guard syntax.

We also observe that to employ the binary search procedure, we need to discretize the domains of variables in X. In general, such discretization is induced by a corresponding discretization of time chosen by the numerical simulator. Since controllers are in any case implemented using finite-precision computer arithmetic, we believe this finitization of intervals is not a restriction in practice.

6.5 Conclusion

We presented a new approach for synthesizing safe hybrid systems that uses numerical simulations and fixpoint computation. The user can guide synthesis by specifying dwell time requirements and the form of the guards. In the following chapter, we extend our SCIDUCTION based approach to automated synthesis of switching logic for quantitative performance properties.

Chapter 7

Synthesis of Switching Logic for Performance Specifications

Given a multi-modal dynamical system, optimal switching logic synthesis involves generating conditions for switching between the system modes such that the resulting hybrid system satisfies a quantitative specification. In this chapter, we formalize and solve the problem of optimal switching logic synthesis for quantitative specifications over long run behavior. We present an approach for specifying quantitative measures using reward and penalty functions, and illustrate its effectiveness using several examples. Each trajectory of the system, and each state of the system, is associated with a cost. Our goal is to synthesize a system that minimizes this cost from each initial state. We present an automated technique to synthesize switching logic for such quantitative measures. Our algorithm works in two steps. For a single initial state, we reduce the synthesis problem to an unconstrained numerical optimization problem which can be solved by any off-the-shelf numerical optimization engine. In the next step, optimal switching condition is learnt as a generalization of the optimal switching states discovered for each initial state. We prove the correctness of our technique and demonstrate the effectiveness of this approach with experimental results.

7.1 Introduction

As discussed in Chapter 5, multi-modal dynamical systems (MDS) is a physical system (plant) that can operate in different modes. The dynamics of the plant in each mode is known. In order to achieve safe and efficient operation, one needs to design the controller for the plant (typically implemented in software) that switches between the different operating modes. Designing correct and optimal switching logic can be tricky and tedious for a human designer.

In this chapter, we consider the problem of synthesizing the switching logic for an MDS so that the resulting system is *optimal*. Optimality is formalized as minimizing a quantitative cost measure over the long-run behavior of the system. Specifically, we formulate cost as penalty per unit reward motivated by similar cost measure in Economics. For a given initial state, the optimal long-term behavior corresponds to a trajectory of infinite length with infinite number of mode switches which has minimum cost. So, discovering the optimal long-term behavior requires

- discovering this infinite chain of mode switches, and
- the switching states from one mode to another.

Thus, this problem would seem to involve optimization over an infinitely-long trajectory, involving an unbounded set of parameters. However, we reduce this problem to optimization over bounded set of parameters representing the *repetitive long-term behavior*. The key insight is that the long-term cost is essentially the cost of the repetitive part of the behavior. We only require the user to provide a guess of a number of switches which could suffice to reach the repetitive behavior from an initial state. The system stays in repetitive behavior after reaching it and hence, the user can pick any large enough bound. We consider the super-sequence of all possible mode sequences with the given number of mode switches and use the times spent in each mode in this super-sequence as the parameters for optimization. If the time spent in a particular mode is zero, the mode is removed from the optimum mode sequence. The optimization problem is then formulated as an unconstrained numerical optimization problem which can be solved by off-the-shelf tools. Solving this optimization problem yields the time spent in each mode which in turn gives us the optimum mode switching sequence. So, to summarize, for a given initial state, we obtain a sequence of *switching states* at which mode transitions must occur so as to minimize the long-run cost. The final step involves generalizing from a sample of switching states to a *switching condition*, or

guard. Given an assumption on the structure of guards, an inductive learning algorithm is used to combine switching states for different initial states to yield the optimum switching logic for the entire hybrid system. This combination of structure assumption on guards, deduction and inductive learning is another instance of the SCIDUCTION approach proposed in the thesis.

7.1.1 Contributions

To summarize, the novel contributions of this chapter are as follows:

- We formalize the problem of synthesizing optimal switching logic by introducing the notion of long-run cost which needs to be minimized for optimality (Section 7.1.2).
- The synthesis problem requires optimization over infinite trajectories and not just a finite time horizon. We show how to reduce optimization over an infinite trajectory to an equivalent optimization over a bounded set of parameters representing the *limit* behavior. (Section 7.2);
- We present an algorithm to solve this optimization problem for a single initial state based on unconstrained numerical optimization (Section 7.2.2). Our algorithm makes *no assumptions on the intra-mode continuous dynamics* other than locally-Lipschitz continuity and relies only on the ability to accurately simulate the dynamics, making it applicable even for nonlinear dynamics, and
- An inductive learning algorithm based on randomly sampling initial states is used to generalize from optimal switching states for individual initial states to an optimal switching guard for the set of all initial states. This generated switching logic is guaranteed to be the true optimal switching logic with high probability (Section 7.2.3).

Experimental results demonstrate our approach on a range of examples drawn from embedded systems design (Section 7.3).

7.1.2 Problem Definition

As in the previous chapter, we model a hybrid system as a combination of a multimodal dynamical system and a switching logic.

Definition 10. Multimodal Dynamical System (MDS). *A* multimodal dynamical system *is a tuple* $\langle Q, X, f, \texttt{Init}\rangle$*, where* $Q := \{1, \ldots, N\}$ *is a set of modes,* $X := \{x_1, \ldots, x_n\}$ *is a set of continuous variables,* $f : Q \times \mathbb{R}^{|X|} \mapsto \mathbb{R}^{|X|}$ *defines a vector field for each mode in* Q*, and* $\texttt{Init} \subseteq Q \times \mathbb{R}^X$ *is a set of initial states. The* state space *of such an MDS is* $Q \times \mathbb{R}^X$*. A function* $\mathbf{qx} : \mathbb{R}^+ \mapsto (Q \times \mathbb{R}^X)$ *is said to be a* trajectory *of this MDS with respect to a sequence* $t_1, t_2, \ldots$ *of* switching times *if*
(i) $\mathbf{qx}(0) \in \texttt{Init}$ *and*
(ii) for all i *and for all* t *such that* $t_i < t$ *and* $t < t_{i+1}$*, it is the case that* $\mathbf{q}(t) = \mathbf{q}(t_i)$ *and*

$$\frac{d\mathbf{x}(t)}{dt} = f(\mathbf{q}(t), \mathbf{x}(t)), \tag{7.1}$$

where $\mathbf{q}$ *and* $\mathbf{x}$ *denote the projection of* $\mathbf{qx}$ *into the mode and continuous state components. The function* $\mathbf{x}$ *is continuous. The* switching sequence is the sequence of modes $\mathbf{q}(t_1), \mathbf{q}(t_2), \ldots$.

If MDS is a multimodal dynamical system, then its semantics, denoted $[\![\texttt{MDS}]\!]$, is the set of its trajectories with respect to all possible switching time sequences.

Definition 11. Switching Logic (SwL). *A* switching logic *for a multimodal system* $\texttt{MDS} := \langle Q, X, f, \texttt{Init}\rangle$ *is a tuple* $\langle (\texttt{g}_{q_1 q_2})_{q_1, q_2 \in Q}\rangle$ *where* $\texttt{g}_{q_1 q_2} \subseteq \mathbb{R}^X$ *is the* guard *defining the switch from mode* q_1 *to mode* q_2*.*

Given a multimodal system and a switching logic, we can now define a hybrid system by considering only those trajectories of the multimodal system that are consistent with the switching logic.

Definition 12. Hybrid System (HS). *A* hybrid system *is a tuple* $\langle \texttt{MDS}, \texttt{SwL}\rangle$ *consisting of a multimodal system* $\texttt{MDS} := \langle Q, X, f, \texttt{Init}\rangle$ *and a switching logic* $\texttt{SwL} := \langle (\texttt{g}_{q_1 q_2})_{q_1, q_2 \in Q}\rangle$*. The* state space *of the hybrid system is the same as the state space of* MDS*. A function* $\mathbf{qx} : \mathbb{R}^+ \mapsto (Q \times \mathbb{R}^X)$ *is said to be a* trajectory *of this hybrid system if there is a a sequence* $t_1, t_2, \ldots$ *of* switching times *such that*
(a) $\mathbf{qx}$ *is a trajectory of* MDS *with respect to this switching time sequence and*
(b) setting $t_0 = 0$*, for all* t_i *in the switching time sequence with* $i \geq 1$*,* $\mathbf{x}(t_i) \in \texttt{g}_{\mathbf{q}(t_{i-1})\mathbf{q}(t_i)}$ *and for all* t *such that* $t_{i-1} < t < t_i$*,* $\mathbf{x}(t) \notin \cup_{q \in Q} \texttt{g}_{\mathbf{q}(t_{i-1})q}$*.*

Discrete jumps are taken as soon as they are enabled and they do not change the continuous variables. For the notion of a trajectory to be well-defined, guards are required to be closed sets.

The semantics of a hybrid system HS, denoted $[\![\texttt{HS}]\!]$, is the collection of all its trajectories as defined above.

Quantitative Measures for Hybrid Systems

Our interest is in automatically synthesizing hybrid systems which are *optimal* in the long-run. Quantitative measure for a hybrid system HS is defined in Section 5.1. To recapitulate, a quantitative measure on a hybrid system HS is defined by extending HS with new continuous state variables. The new continuous variables compute "rewards" or "penalties" that are accumulated over the course of a hybrid trajectory. We also allow the new variables to be updated during discrete transitions, which enables us to penalize or reward discrete mode switches.

Definition 13. Performance Metric. *A* performance metric *for a given multimodal system* $\texttt{MDS} := \langle Q, X, f, \texttt{Init}\rangle$ *is a tuple* $\langle \texttt{PR}, f_{\texttt{PR}}, \texttt{update}\rangle$*, where* $\texttt{PR} := P \cup R$ *is a finite set of continuous variables (disjoint from* X*), partitioned into penalty variables* P *and reward variables* R*,* $f_{\texttt{PR}} : Q \times \mathbb{R}^X \mapsto \mathbb{R}^{\texttt{PR}}$ *defines the vector field that determines the evolution of the variables* PR*, and* $\texttt{update} : Q \times Q \times \mathbb{R}^{\texttt{PR}} \mapsto \mathbb{R}^{\texttt{PR}}$ *defines the updates to the variables* PR *at mode switches.*

Given a trajectory $\mathbf{qx} : \mathbb{R}^+ \mapsto (Q \times \mathbb{R}^X)$ of a multimodal or hybrid system with mode-switching time sequence $t_1, t_2, \ldots$, and given a performance metric, we define the *extended trajectory* $\mathbf{qx}^e : \mathbb{R}^+ \mapsto (Q \times \mathbb{R}^X \times \mathbb{R}^{\texttt{PR}})$ with respect to the same mode-switching time sequence as any function that satisfies $\mathbf{qx}^e(0) = (\mathbf{q}(0), \mathbf{x}(0), \vec{0})$ and $\mathbf{qx}^e(t) = (\mathbf{q}(t), \mathbf{x}(t), \mathbf{PR}(t))$, where $\mathbf{PR}$ satisfies:

$$\frac{d\mathbf{PR}(t)}{dt} = f_{\texttt{PR}}(\mathbf{qx}(t)) \text{ for all } t : t_i < t < t_{i+1} \text{ and,}$$
$$\mathbf{PR}(t_i) = \texttt{update}(\mathbf{q}(t_{i-1}), \mathbf{q}(t_i), \lim_{t \to t_i^-} \mathbf{PR}(t))$$

.

The cost of a trajectory $\mathbf{qx}$ is defined using its corresponding extended trajectory $\mathbf{qx}^e$ as

$$\texttt{cost}(\mathbf{qx}) := \lim_{t \to \infty} \sum_{i=1}^{|P|} \frac{\mathbf{P}_i(t)}{\mathbf{R}_i(t)} \tag{7.2}$$

where $\mathbf{P}_i$ and $\mathbf{R}_i$ are the projection of $\mathbf{qx}^e$ onto the i-th penalty variable and i-th reward variable, and $|P| = |R|$.

We are only interested in trajectories where the above limit exists and is finite. We will further define $\texttt{cost}$ of a part of a trajectory from time instant t_1 to a time instant t_2 ($t_2 > t_1$) as follows:

$$\texttt{cost}(\mathbf{qx}, t_1, t_2) := \sum_{i=1}^{|P|} \frac{\mathbf{P}_i(t_2) - \mathbf{P}_i(t_1)}{\mathbf{R}_i(t_2) - \mathbf{R}_i(t_1)} \tag{7.3}$$

where $\mathbf{P}_i$ and $\mathbf{R}_i$ are components of $\mathbf{PR}$ as before.

As the definition of $\texttt{cost}$ indicates, we are interested in the *long-run average* (penalty per unit reward) cost rather than (penalty or reward) cost over some bounded/finite time horizon.

Optimal Switching Logic Synthesis

Definition 14. Optimal Switching Synthesis Problem. *Given a multimodal system* $\texttt{MDS} = \langle Q, X, f, \texttt{Init}\rangle$ *, and a performance metric, the* optimal switching logic synthesis problem *seeks to find a switching logic* $\texttt{SwL}^*$ *such that the cost of a trajectory from any initial state in the resulting hybrid system* $\texttt{HS}^* := \texttt{HS}(\texttt{MDS}, \texttt{SwL}^*)$ *is no more than the cost of corresponding trajectory from the same initial state in an arbitrary hybrid system* $\texttt{HS} := \texttt{HS}(\texttt{MDS}, \texttt{SwL})$ *obtained using an arbitrary switching logic* $\texttt{SwL}$*, that is,* $\forall(\mathbf{x}, q) \in \texttt{Init}\ .\ \texttt{cost}(\mathbf{qx}^*) \leq \texttt{cost}(\mathbf{qx})$ *where* $\mathbf{qx}^*(0) = \mathbf{qx}(0) = (\mathbf{x}, q), \mathbf{qx} \in [\![\texttt{HS}^*]\!], \mathbf{qx} \in [\![\texttt{HS}]\!]$

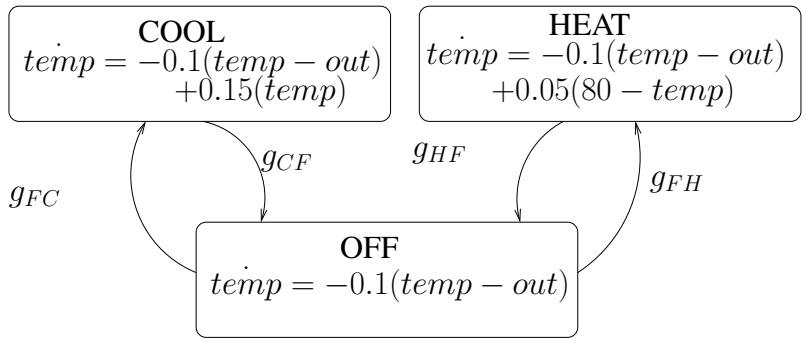
We will assume, without loss of any generality, that we are given an over-approximation of the switching logic $\texttt{SwL}^{over} := \langle(\texttt{g}^{over}_{qq'})_{q,q'\in Q}\rangle$. In this case, the optimal synthesis problem seeks to find a switching logic $\texttt{SwL}^* := \langle(\texttt{g}^*_{qq'})_{q,q'\in Q}\rangle$ that also satisfies the constraint that $\texttt{g}^*_{qq'} \subseteq \texttt{g}^{over}_{qq'}$ for all $q, q' \in Q$, which is also written in short as $\texttt{SwL}^* \subseteq \texttt{SwL}^{over}$.

The over-approximation $\texttt{SwL}^{over}$ of the switching set can be used to restrict the search space for switching conditions. The set $\texttt{g}^{over}_{qq'}$ can be an empty set if switches are disallowed from q to q'. The set $\texttt{g}^{over}_{qq'}$ can be $\mathbb{R}^X$ if there is no restriction on switching from q to q'.

7.1.3 Running Example

Let us consider a simple three mode thermostat controller as our running example. The multimode dynamical system describing this system is presented in Figure 7.1. The thermostat controller is described by the tuple $\langle Q, X, f, \texttt{Init}\rangle$ where $Q = \{\texttt{OFF}, \texttt{HEAT}, \texttt{COOL}\}$, $X =$

$\{\texttt{temp}, \texttt{out}\}$, f is $f_{\texttt{OFF}} : \dot{\texttt{temp}} = -0.1(\texttt{temp} - \texttt{out})$ in mode OFF, $f_{\texttt{HEAT}} : \dot{\texttt{temp}} = -0.1(\texttt{temp} - \texttt{out}) + 0.05(80 - \texttt{temp})$ in mode HEAT and $f_{\texttt{COOL}} : \dot{\texttt{temp}} = -0.1(\texttt{temp} - \texttt{out}) - 0.15(\texttt{temp})$ in mode COOL, and $\texttt{Init} = \texttt{OFF} \times [18, 20] \times [12, 26]$. For simplicity, we assume that the outside temperature out does not change.



$\dot{\texttt{discomfort}} = (\texttt{temp} - 20)^2,\ \dot{\texttt{fuel}} = (\texttt{temp} - \texttt{out})^2$

$\dot{\texttt{swTear}} = 0,\ \dot{\texttt{time}} = 1$

$\texttt{update}(\texttt{M}, \texttt{M}', \texttt{swTear}) = \texttt{swTear} + 0.5$

for any two different modes M, M′ in Q

Figure 7.1: Thermostat Controller

The performance requirement is to keep the temperature as close as possible to the target temperature 20 and to consume as little fuel as possible in the long run. We also want to minimize the wear and tear of the heater caused by switching. The performance metric is given by the tuple $\langle \texttt{PR}, f_{\texttt{PR}}, \texttt{update} \rangle$, where penalty variables $P = \{\texttt{discomfort}, \texttt{fuel}, \texttt{swTear}\}$ denote the discomfort, fuel and wear and tear due to switching and reward variables $R = \{\texttt{time}\}$ denote the time spent. The evolution and update functions for the penalty and reward variables is shown in Figure 7.1. We need to synthesize the guards such that the following cost metric is minimized. Since the reward variable is the time spent, minimizing this metric means minimizing the *average* discomfort, fuel cost and wear and tear of the heater. We give a higher weight (10) to discomfort than fuel cost and wear and tear.

$$\lim_{t \to \infty} \frac{10 \times \texttt{discomfort}(t) + \texttt{fuel}(t) + \texttt{swTear}(t)}{\texttt{time}(t)}$$

7.2 SCIDUCTIVE Approach

In this section, we present a SCIDUCTION based approach for automated synthesis of optimal switching logic which minimizes long-run cost. We use numerical optimization to discover optimal switching points for a single initial state and then, sample over the initial states to inductively infer the optimal switching logic.

Given a multi-modal system $\texttt{MDS} = \langle Q, X, f, d, Inv, Init\rangle$, an initial state $(q_0, \mathbf{x}_0) \in Init$ and the performance metric tuple $(Y, f_Y, \texttt{update})$, we need to find the switching times $t_1, t_2, \ldots$ and the mode switching sequence $\mathbf{q}$ such that the corresponding trajectory $\mathbf{qx}$ is of minimum cost.

$$\begin{aligned}
&\min_{\mathbf{q}, t_1, t_2, \ldots} \texttt{cost}(\mathbf{qx}) \quad \textit{subject to} \\
&(1)[\text{Init}] : \mathbf{qx}(0) = (q_0, \mathbf{x}_0) \;\; (2)[\text{Guards}] : \forall i \; \mathbf{x}(t_i) \in \mathrm{g}^{over}_{\mathbf{q}(i)\mathbf{q}(i+1)} \\
&(3)[\text{Time elapse}] : \forall t \;.\; t_i < t < t_{i+1} \;.\; \mathbf{q}(t) = \mathbf{q}(t_i), i = 1, 2, \ldots; \\
&(4)[\text{Flow}] : \forall t \; \frac{d\mathbf{x}(t)}{dt} = f(\mathbf{q}(t), \mathbf{x}(t))
\end{aligned} \tag{7.4}$$

Since the switching sequence $t_1, t_2, \ldots$ could be of infinite length, it is not a-priori evident how to solve the above problem. In the rest of the section, we formulate an equivalent optimization problem with finite number of switching times as variables.

7.2.1 Optimization over Finite Parameters

We now show how to reduce the above optimization problem with infinite number of switching times as parameters to an optimization problem with finite parameters. Let *trajectory segment* $\mathbf{qx}_{[tf,te]}$ of a trajectory $\mathbf{qx}$ of length $L = tf - te$ be the restriction of the *trajectory* to $tf \leq t \leq te$, that is, $\mathbf{qx}_{[tf,te]} : T \mapsto (Q \times \mathbb{R}^X)$ where $T = [tf, te] \subseteq \mathbb{R}^+$ and $\mathbf{qx}_{[tf,te]}(t) = \mathbf{qx}(t)$ for $tf \leq t \leq te$. The *switching times* of the trajectory segment is a finite sub-sequence $t_m, t_{m+1}, \ldots t_n$ of the switching times $t_1 \ldots, t_m, \ldots, t_n, \ldots$ of the trajectory $\mathbf{qx}$ and $t_{m-1} < tf \leq t_m$ and $t_n \leq te < t_{n+1}$. The special case of a trajectory segment is a *trajectory prefix* in which the trace starts at time $ts = 0$.

Our goal is to minimize the lifetime cost. The lifetime cost is dominated by the the cost of the *limit behavior* of the system. We are only interested in the following stable limit behaviors when the lifetime cost is defined by the limit in Equation 7.2.

- *asymptotic:* for any ϵ, there exists a time t_ϵ after which the trajectory gets asymptotically ϵ-close to some state $(q_T, \mathbf{x}_T)$, $||\mathbf{qx}(t) - (q_T, \mathbf{x}_T)||^2 < \epsilon$ for all $t \geq t_\epsilon$ where $||.,.||$ denotes the Euclidean norm, or

- *converging:* there exists a time t_{conv} after which the trajectory converges, $\mathbf{qx}(t) = \mathbf{qx}(t_{conv})$ for all $t \geq t_{conv}$, or

- *cyclic:* there exists a time t_{cyc} after which the trajectory enters a cycle with period L, $\mathbf{qx}(t) = \mathbf{qx}(t + kP)$ for all $t \geq t_{cyc}$ and $k \geq 1$.

In all these cases, we can reason about the long-run cost by considering some finite, but arbitrarily long, trajectory prefixes. Suppose the trajectory $\mathbf{qx}$ is asymptotic to some hybrid state $\mathbf{qx}^\infty = (q^\infty, \mathbf{x}^\infty)$. In this case, we assume that the penalty and reward variables $\mathtt{PR}$ also asymptotically approach some values $\mathtt{PR}^\infty = (P^\infty, R^\infty)$. Now consider the trajectory prefix $\mathbf{qx}_{[0,te]}$. We have

$$\mathtt{cost}(\mathbf{qx}) = \sum_i \frac{P_i^\infty}{R_i^\infty} \quad \text{and}$$

$$\mathtt{cost}(\mathbf{qx}_{[0,te]}) = \sum_i \frac{P_i(te)}{R_i(te)} < \sum_i \frac{P_i^\infty + \epsilon}{R_i^\infty - \epsilon} < \mathtt{cost}(\mathbf{qx}) + \delta_\epsilon$$

Hence, by choosing te appropriately, we can find a trajectory prefix whose cost is arbitrarily close to the cost of the asymptotic trajectory.

Any repetitive trajectory $\mathbf{qx}$ can be decomposed into a finite prefix $\mathbf{qx}_{pref} = \mathbf{qx}_{[0,tp]}$ followed by a trajectory segment $\mathbf{qx}_{rep} = \mathbf{qx}_{[tp,tP]}$ repeated infinitely. We say

$$\mathbf{qx} = \mathbf{qx}_{pref} \,.\, (\mathbf{qx}_{rep})^\omega$$

when $\forall t \leq tp \,.\, \mathbf{qx}(t) = \mathbf{qx}_{pref}(t)$ and $\forall t \geq tp \,.\, \mathbf{qx}(t) = \mathbf{qx}_{rep}(tp + r)$ where $r = (t - tp) \mod L$ and $L = tP - tp$. The case when the trajectory converges to some hybrid state can be treated in the same way as a repetitive trajectory.

In Lemma 3 and Theorem 10, we summarize how cost converges to a limit for trajectories with repetitive limit behavior.

Lemma 3. *For each repetition of the segment $\mathbf{qx}_{rep} = \mathbf{qx}_{[tp,tP]}$, the change in penalty and reward variables is constant, that is, for $P = tP - tp$.*

$$\begin{aligned}\forall k \geq 1 \,.\, P_i(tp + kP) - P_i(tp + (k-1)P) \\ = P_i(tp + P) - P_i(tp) = \Delta P_i \\ \forall k \geq 1 \,.\, R_i(tp + kP) - R_i(tp + (k-1)P) \\ = R_i(tp + P) - R_i(tp) = \Delta R_i\end{aligned}$$

Proof. The change in penalty and reward variables is given by the evolution function $f_{\texttt{PR}}$ and the update on switch function `update`. We know that $\mathbf{qx}_{rep}$ is repetitive and so, $\mathbf{qx}(tp + kP + t) = \mathbf{qx}(tp + t)$ for all $t < tP - tp$ and $k \geq 1$ and hence,

$$f_{\texttt{PR}}(\mathbf{qx}(tp + kP + t)) = f_{\texttt{PR}}(\mathbf{qx}(tp + t))$$

Also, for any mode switch time $tp \leq t_i \leq tP$, $t'_i = t_i + kP$ is also a switch time because hybrid states at t_i and t'_i are the same. Further,

$$\texttt{update}(q(t_{i-1}), q(t_i), \lim_{t \to t_i^-} \mathbf{PR}(t))$$

$$= \texttt{update}(q(t'_{i-1}), q(t'_i), \lim_{t \to t_i'^-} \mathbf{PR}(t))$$

So, integrating f_{PR} over continuous evolution and applying `update` function at mode switches, we observe that

$$\begin{aligned}P_i(tp + kP) - P_i(tp + (k-1)P) = P_i(tp + P) - P_i(tp) \\ = \Delta P_i \\ R_i(tp + kP) - R_i(tp + (k-1)P) = R_i(tp + P) - R_i(tp) \\ = \Delta R_i\end{aligned}$$

□

Theorem 10. *For a trajectory* $\mathbf{qx}$ *which can be decomposed into* $\mathbf{qx}_{pref} \,.\, (\mathbf{qx}_{rep})^{\omega}$*, the cost of the trajectory is equal to the cost of the repetitive segment* $\mathbf{qx}_{rep}$*, that is,* $\texttt{cost}(\mathbf{qx}) = \texttt{cost}(\mathbf{qx}_{rep})$.

Proof.

$$
\begin{aligned}
\texttt{cost}(\mathbf{qx}) &:= \lim_{t\to\infty} \sum_{i=1}^{|P|} \frac{\mathbf{P}_i(t)}{\mathbf{R}_i(t)} \ \text{[Equation 7.2]} \\
&= \lim_{t\to\infty} \sum_{i=1}^{|P|} \frac{\mathbf{P}_i(tp) + \mathbf{P}_i(t) - \mathbf{P}_i(tp)}{\mathbf{R}_i(tp) + \mathbf{R}_i(t) - \mathbf{R}_i(tp)} \\
&= \lim_{k\to\infty} \sum_{i=1}^{|P|} \frac{\mathbf{P}_i(tp) + k\Delta\mathbf{P}_i}{\mathbf{R}_i(tp) + k\Delta\mathbf{R}_i} \ \text{[Lemma 3]} \\
&= \frac{\Delta\mathbf{P}_i}{\Delta\mathbf{R}_i} \ [\mathbf{P}_{\mathrm{i}}(tp), \mathbf{R}_{\mathrm{i}}(tp) \text{ are finite}] \\
&= \sum_{i=1}^{|P|} \frac{\mathbf{P}_i(tP) - \mathbf{P}_i(tp)}{\mathbf{R}_i(tP) - \mathbf{R}_i(tp)} \\
&= \texttt{cost}(\mathbf{qx}, tp, tP) \ \text{[Equation 7.3]} \\
&= \texttt{cost}(\mathbf{qx}_{rep}) \ \text{[Definition of } \mathbf{qx}_{\mathrm{rep}}]
\end{aligned}
$$

□

Using Theorem 10, the optimization problem in Equation 7.4 is equivalent to the following optimization problem. Intuitively, if the repetitive part of the trajectory and the finite prefix before the repetitive part have finite cost, then the long run cost of a trajectory in the limit is the cost of the repetitive part of the trajectory. More generally, to also handle the case when the (optimal) trajectory is asymptotic, we can replace the cyclicity requirement, $\mathbf{qx}(tp) = \mathbf{qx}(tP)$, in the optimization problem by the weaker requirement that the state $\mathbf{qx}(tP)$ at time tP be very "close" to the state $\mathbf{qx}(tp)$ at time tp; see also Section 7.2.2.

$$
\begin{aligned}
&\min_{\mathbf{q}, t_1, t_2, \ldots} \ \texttt{cost}(\mathbf{qx}) \quad \textit{subject to} \qquad (7.5)\\
&(1)[\text{Init}] : \mathbf{qx}(0) = (q_0, \mathbf{x}_0) \ \ (2)[\text{Guards}] : \forall i \ \mathbf{x}(t_i) \in \mathrm{g}^{over}_{\mathbf{q}(i)\mathbf{q}(i+1)} \\
&(3)[\text{Time elapse}] : \forall t \ . \ t_i < t < t_{i+1} \ . \ \mathbf{q}(t) = \mathbf{q}(t_i), i = 1, 2, \ldots; \\
&(4)[\text{Flow}] : \forall t \ \frac{d\mathbf{x}(t)}{dt} = f(\mathbf{q}(t), \mathbf{x}(t)) \\
&(5)[\text{Repetitive Trajectory}] : \ \mathbf{qx} = \mathbf{qx}_{pref} \ . \ (\mathbf{qx}_{rep})^{\omega} \\
&(6)[\text{Repetitive Time}] : \ \mathbf{qx}_{pref} = \mathbf{qx}_{[0,tp]}, \ \mathbf{qx}_{rep} = \mathbf{qx}_{[tp,tP]} \\
&\text{where } 0 \leq t_1 \leq \ldots t_n \leq tP, \ 0 \leq tp < tP
\end{aligned}
$$

7.2.2 Numerical Optimization

In this section, we present an algorithm to solve the above optimization problem. The key idea is to construct a scalar function $F(\mathbf{q}, t_1, t_2, \ldots, t_n, tp, tP)$ where $\mathbf{q}$ is the switching mode sequence; $t_1, t_2, \ldots, t_n$ are the switching times, and tp, tP are the times denoting repetitive behavior, such that the minimum value of F is attained when the switching mode sequence and switching times correspond to the trajectory $\mathbf{qx}$ with minimum long-run cost, and $\mathbf{qx}_{[tp,tP]}$ is the repetitive part of the trajectory.

Once we have constructed F, we need to minimize F. Apart from $\mathbf{q}$, all arguments of F are real-valued. Suppose we fix $\mathbf{q}$ and let $F_{\mathbf{q}}(t_1, t_2, \ldots, t_n, tp, tP)$ denote the function F with fixed mode sequence $\mathbf{q}$. Now $F_{\mathbf{q}}$ is a function from multiple real variables to a real, and hence (approximate) minimization of F can be performed using *unconstrained nonlinear numerical optimization* techniques [17]. These techniques only require that we are able to evaluate F once its arguments are fixed. This we accomplish using numerical simulation of the multimodal system.

Defining F

The optimization problem in Equation 7.5 is a constrained optimization problem. The constraint $\mathbf{qx} = \mathbf{qx}_{pref} \ . \ (\mathbf{qx}_{rep})^{\omega}$ requires identifying a trajectory $\mathbf{qx}$ starting from the given initial state $(q_0, \mathbf{x}_0)$ such that it enters repetitive behavior at time tp, and $q(tp) = q(tP)$ and $\mathbf{x}(tp) = \mathbf{x}(tP)$ where $tp < tP$. We call this constraint the repetition constraint. A standard technique for solving some constrained optimization problems is to translate it into an unconstrained optimization problem by modifying the optimization objective such that optimization automatically enforces the constraint. This is done by quantifying the violation using some metric and then minimizing the sum of the earlier minimization objective and the weighted violation measure. In order to enforce the repetition constraint by suitably modifying the optimization objective, we introduce a distance function between the hybrid states. Let d be the distance function between two hybrid states such that

$$d((q_1, \mathbf{x}_1), (q_2, \mathbf{x}_2)) = ||\mathbf{x}_1 - \mathbf{x}_2||^2 \ \text{ if } q_1 = q_2 \quad \text{and} \ \infty \ \ o.w.$$

where $||\mathbf{x}_1 - \mathbf{x}_2||$ is the Euclidean norm. $(Q \times X, d)$ forms a metric space. So, the distance between the hybrid states is 0 if and only if $q_1 = q_2$ and $\mathbf{x}_1 = \mathbf{x}_2$.

$$\text{Let } F(\mathbf{q}, t_1, \ldots, t_n, tp, tP) = \begin{cases} \texttt{cost}(\mathbf{qx}_{[tp,tP]}) + M \times d(\mathbf{qx}(tp), \mathbf{qx}(tP)) \\ \texttt{if } (a)\ 0 \leq t_1 \leq \ldots t_n \leq tP,\ tp < tP \text{ and} \\ \quad (b)\ \forall i\ \mathbf{x}(t_i) \in \mathrm{g}^{over}_{\mathbf{q}(i)\mathbf{q}(i+1)} \\ \infty \quad \texttt{otherwise} \end{cases}$$

where M is any positive constant and $\mathbf{qx}$ is a trajectory starting from the given initial state, that is,

$$\mathbf{qx}(0) = (q_0, \mathbf{x}_0);\ \forall t\ t_i < t < t_{i+1}\ \mathbf{q}(t) = \mathbf{q}(t_i)\ \ ,\ i = 1, 2, \ldots$$
$$\text{and } \forall t\ \frac{d\mathbf{x}(t)}{dt} = f(\mathbf{q}(t), \mathbf{x}(t))$$

It is easy to see that the minimum value of the function F is attained when the hybrid states at time tp and tP are the same, that is, the trajectory segment $\mathbf{qx}_{[tp,tP]}$ is the repetitive part of the trajectory and the cost of this segment is minimum. Using Theorem 10, we conclude that the optimization problem in Equation 7.5 of Section 7.2 can be reduced to the following unconstrained multivariate numerical optimization problem

$$\min_{t_1,\ldots,t_n,tp,tP} \quad F(t_1, \ldots, t_n, tp, tP) \tag{7.6}$$

As remarked above, if the arguments of F are fixed, then F can be evaluated using a numerical simulator. Also, for a fixed $\mathbf{q}$, we can use a numerical nonlinear optimization engine to find the minimum value of the function $F_{\mathbf{q}}$.

Running Example

We illustrate our technique for the running example with a fixed sequence of modes say $\mathbf{q} =$ `OFF`, `HEAT`, `OFF` starting from the initial state

$$(\texttt{OFF}, \texttt{temp} = 22, \texttt{out} = 16)$$

The outside temperature `out` does not change with time and remains the same as the initial state. Only the room temperature `temp` changes with time. The switching time sequence is t_1, t_2. Let

tp denote the time when the thermostat enters the repetitive behavior and tP be the time such that $\texttt{temp}(tp) = \texttt{temp}(tP)$. When $t_1 \leq t_2 \leq tp \leq tP$ and $tp < tP$, the function

$$F_{\mathbf{q}}(t_1, t_2, tp, tP) = \texttt{cost}(\mathbf{qx}_{[tp,tP]}) + 1000(\texttt{temp}(tp) - \texttt{temp}(tP))^2$$

and it is set to 2000 otherwise (approximating infinity in the formulation with a very high constant). We use $ode45$ function in MATLAB [92] for numerically simulating the ordinary differential equations representing continuous dynamics in each mode. In order to find the minimum value of $F_{\mathbf{q}}$ and the corresponding arguments that minimize the function, we use the implementation of Nelder-Mead simplex algorithm [101]. The minimum value of $F_{\mathbf{q}}$ is obtained at

	t_0	t_1	t_2	tp	tP
t	0	5.02	5.24	3.54	5.24
$temp$	22.0	19.6	20.2	20.2	20.2

So, the switch states corresponding to the minimum long-run cost for the given initial state

$$(\texttt{OFF}, \texttt{temp} = 22, \texttt{out} = 16)$$

and given switching sequence of modes

$$\texttt{OFF}, \texttt{HEAT}, \texttt{OFF}$$

is $\mathsf{g}_{HF} = \{20.2\}$ and $\mathsf{g}_{FH} = \{19.6\}$.

We repeat the experiments with different initial states but with the same mode switching sequence. Even with different initial states $(\texttt{OFF}, \texttt{temp} = 20.5, \texttt{out} = 16)$, $(\texttt{OFF}, \texttt{temp} = 21, \texttt{out} = 16)$ and $(\texttt{OFF}, \texttt{temp} = 21.5, \texttt{out} = 16)$, we obtain the same switching states in this example: $\mathsf{g}_{HF} = \{20.2\}$ and $\mathsf{g}_{FH} = \{19.6\}$.

When we change the mode switching sequence to $\texttt{OFF}, \texttt{HEAT}, \texttt{OFF}, \texttt{HEAT}, \texttt{OFF}$, we discover the optimal switching sequence to be

	t_0	t_1	t_2	t_3	t_4	tp	tP
t	0	5.02	5.24	6.73	6.95	3.54	6.95
$temp$	22.0	19.6	20.2	19.6	20.2	20.2	20.2

$t_1 = 5.02, t_2 = 5.24, t_3 = 6.73, t_4 = 6.95, tp = 3.54, tP = 6.95$ which again yields the same optimal switching states $\mathrm{g}_{HF} = \{20.2\}$ and $\mathrm{g}_{FH} = \{19.6\}$.

We observe that the optimal behavior with respect to the given cost metric would be to switch from `OFF` mode to `HEAT` mode at $\texttt{temp} = 19.6$ and then switch from `HEAT` to `OFF` mode at $\texttt{temp} = 20.2$ regardless of the initial room temperature as long as the outside temperature $\texttt{out} = 16$. The optimal mode cycle is between `OFF` and `HEAT` modes.

For an initial state with outside temperature higher than the outside room temperature $\texttt{out} > 20$, the optimal cycle would be between `OFF` and `COOL` modes. With the mode sequence `OFF`, `COOL`, `OFF` and the initial state $(\texttt{OFF}, \texttt{temp} = 20.5, \texttt{out} = 26)$, we discover the optimal switching states to be $\mathrm{g}_{CF} = \{20\}$ and $\mathrm{g}_{FC} = \{20.3\}$.

Finding Optimal Mode Sequence

The algorithm above assumed that the switching mode sequence $\mathbf{q}$ was fixed. It can be easily adapted to also automatically discover the optimal switching mode sequence. Any mode sequence starting in mode 1 and with at most k switches in a system with N modes $Q = \{1, 2, \ldots N\}$ is a subsequence of $1(2 \ \ldots \ N \ 1)^k$, that is, mode 1 followed by $(2 \ \ldots \ N \ 1)$ repeated k times. Let dwell-time of a mode i be the time spent in the mode $t_{i+1} - t_i$. Given the switching times $t_1, t_2, ...t_{Nk}$ and tp, tP, we define the NZ function which removes the switch times and modes from the switching sequence with zero dwell-times, that is,

$$
\begin{aligned}
&NZ(\overline{\mathbf{q}}, t_1, t_2, \ldots, t_{Nk}, tp, tP) = (\mathbf{q}, t_{i_1}, t_{i_2}, \ldots, t_{i_K}, tp, tP) \\
&\text{where } \mathbf{q} = q_{i_1}, q_{i_2}, \ldots, q_{i_K}, 0 < t_{i_1} < t_{i_2} < \ldots < t_{i_K} < tP \\
&\text{and } t_m = t_{i_j} \text{ for all } i_j < m < i_{j+1}
\end{aligned}
$$

For example, given the sequence of switching times $5, 6, 6, 11, 12, 12$ and $tp = 6.5, tP = 12.5$ with the switching mode sequence $\overline{\mathbf{q}} = 1, 2, 3, 1, 2, 3, 1$,

$$NZ(\overline{\mathbf{q}}, 5, 6, 6, 11, 12, 12, 6.5, 12.5) = (\mathbf{q}, 5, 6, 11, 12, 6.5, 12.5)$$

where $\mathbf{q} = 1, 2, 1, 2, 1$.

Given a guess on the number of mode switches k such that k or less switches are needed to reach the optimal repetitive behavior, we can use $\overline{\mathbf{q}} = 1(2 \ \ldots \ N \ 1)^k$ as the over-approximate

switching mode sequence and then find the optimal switching subsequence corresponding to the minimal long-run cost behavior using the following modified optimization formulation.

$$\min_{t_1,\ldots,t_{Nk},tp,tP} \quad F(NZ(\overline{\mathbf{q}}, t_1, \ldots, t_{Nk}, tp, tP)) \tag{7.7}$$

If the optimal value returned by minimizing the above function is attained with the arguments $t_1^*, \ldots, t_{Nk}^*, tp^*, tP^*$, then the optimal switching sequence $\mathbf{q}$ and the optimal switching time sequence is given by

$$(\mathbf{q}, t_{i_1}, \ldots, t_{i_k}, tp, tP) = NZ(\overline{\mathbf{q}}, t_1^*, \ldots, t_{Nk}^*, tp^*, tP^*)$$

Running Example

We illustrate the above technique on the running example below. Let us guess that reaching the optimal repetitive behavior from the initial state $\texttt{OFF}, \texttt{temp} = 22, \texttt{out} = 16$ takes at most 2 switches. We consider the mode sequence $\texttt{OFF}, \texttt{HEAT}, \texttt{COOL}, \texttt{OFF}, \texttt{HEAT}, \texttt{COOL}, \texttt{OFF}$ which would contain all mode sequences with 2 switches (it also contains some mode sequences with more than 2 switches). We try to minimize the corresponding function $F(NZ(t_1, t_2, \ldots, t_6, tp, tP))$.

The minimum value obtained for the function F with the starting state $(\texttt{OFF}, \texttt{temp} = 22, \texttt{out} = 16)$ by our optimization engine corresponds to the following trajectory.

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	tp	tP
t	0	5.08	5.32	5.32	6.97	7.23	7.23	4.87	8.66
$temp$	22.0	19.6	20.2	20.2	19.6	20.2	20.2	19.7	19.7

The optimal mode sequence and the switching times points are obtained as

$$NZ(\overline{\mathbf{q}}, 5.08, 5.32, 5.32, 6.97, 7.23, 7.23, 4.87, 8.66) =$$
$$(\texttt{OFF}, \texttt{HEAT}, \texttt{OFF}, \texttt{HEAT}, \texttt{OFF}, 5.08, 5.32, 6.97, 7.23, 4.87, 8.66)$$

Since $tp = 4.87$ and $tP = 8.66$, the repetitive part of the mode sequence is $\texttt{HEAT}, \texttt{OFF}$. The switch from mode $\texttt{OFF}$ to $\texttt{HEAT}$ occurs at times t_1 and t_4. We observe that $\texttt{temp}(t_1) = \texttt{temp}(t_4) = 19.6$. So, the optimal trajectory switches from $\texttt{OFF}$ to $\texttt{HEAT}$ at $\texttt{temp} = 19.6$. The switches from $\texttt{HEAT}$

to COOL and then to OFF occur at the same times: $t_2 = t_3$ and $t_5 = t_6$. So, the dwell-time in the mode COOL is 0 and it needs to be removed from the optimal switching sequence. The switch into mode OFF occurs at times t_3 and t_6 with $\texttt{temp}(t_3) = \texttt{temp}(t_6) = 20.2$. Thus, the optimal mode sequence is $\texttt{OFF}, (\texttt{HEAT}, \texttt{OFF})^{\omega}$ and the guards discovered from this trajectory are $\mathsf{g}_{FH} = 19.6$ and $\mathsf{g}_{HF} = 20.2$. □

Thus, the approach presented so far can be used to synthesize switching conditions for minimum cost long-run behavior for a given initial state. We need a guess on the number of switches k such that the optimal behavior has at most k switches.

We summarize the guarantee of our approach for a single initial state in the following theorem

Theorem 11. *For a single initial state, our technique discovers the switching states corresponding to the optimal trajectory with minimum long-run cost if numerical optimization engine can discover global minimum of the numerical function F.*

The proof of the above theorem follows from the definition of F. If numerical optimization engines are guaranteed to only find local minima of F, our technique will find trajectories of minimal cost. We employ the Nelder-Mead simplex algorithm as described by Lagarias et al [74, 101] for minimizing F since it is a derivative-free method and it can better handle discontinuities in function F. We use its implementation available as the $fminsearch$ [91] function in MATLAB.

7.2.3 Guard Inference Using Learning

In this section, we present an approach to synthesize optimal switching logic from the discovered optimal switching states for sampled initial states. The numerical optimization based approach presented in Section 7.2.2 can find the switching state for each mode switch along the trajectory corresponding to optimal long-run behavior *for a given initial state*. However, since systems are generally designed to operate in more than one initial state, we need to synthesize the *guard condition* for mode switches such that the trajectory from each initial state has optimal long-run cost. In this section, we present a technique for synthesizing guard conditions in a probabilistic setting, where we assume the ability to sample initial states from their (arbitrary) probability distribution. Our technique samples initial states and obtains corresponding optimal switching states for each mode switch. From the individual optimal switching states, we gener-

alize to obtain the guard condition for the mode switch using *inductive learning* (learning from examples). In order to employ learning, we make a *structure assumption* on the form of guards and use concept learning algorithms to efficiently learn the guards from sampled switching states. We assume that the guard condition is a halfspace, that is, a linear inequality over the continuous variables X.

In the rest of the section, we discuss how the existing results from algorithmic concept learning can be used efficiently to learn a halfspace representing the guard condition from the discovered switching states for each mode-switch. We first mention results which prove the efficient learnability of halfspaces and then present an algorithm which can be used to learn halfspaces in the probabilistically approximately correct (PAC) learning framework [131]. In this framework, the learner receives samples marked as positive or negative for points lying inside and outside the concept respectively, and the goal is to select a generalization concept from a certain class of possible concepts such that the selected concept has low generalization error with very high probability. In our case, the concept class is the set of all possible halfspaces in $\mathbb{R}^n$ and the concept to be learnt is the halfspace that is the correct guard in the optimal switching logic. The points in a concept to be learnt are the states in the guard and the points outside the concept are the states outside the guard.

A halfspace can be learnt with a very high accuracy using polynomial-sized sample [15]. We briefly summarize the relevant results from learning theory that establish the efficient learnability of halfspaces in the PAC learning framework. A concept class is said to *shatter* a set of points if for any classification of the points as positive and negative, there is some concept in the class which would correctly classify the points. Any concept class is associated with a combinatorial parameter of the class, namely, the Vapnik-Chervonenkis (VC) dimension defined as the cardinality of the largest set of points (arbitrarily labeled as positive or negative) that the algorithm can shatter. For example, consider the concept class to be partitions in $\mathbb{R}^2$ using straight lines, that is, halfspaces in $\mathbb{R}^2$. The straight line should separate positive points in the true concept and negative points outside the concept. There exist sets of 3 points that can indeed be shattered using this model; in fact, any 3 points that are not collinear can be shattered, no matter how one labels them as positive or negative. However, it can be shown using Radon's theorem that no set of 4 points can be shattered [44]. Thus, the VC dimension of straight lines is 3. In general, the VC dimension for halfspaces in $\mathbb{R}^n$ is known to be $n + 1$ [15]. The following theorem from Blumer et al [15] establishes the relation

between efficient learnability of a concept class in the PAC learning framework and VC dimension of the concept class.

Theorem 12. *Let* $\mathbf{C}$ *be a concept class with a finite VC dimension* d*. Then, any concept in* $\mathbf{C}$ *can be learnt in the following sense: with probability at least* $1-\delta$ *a concept* C *is learnt which incorrectly labels a point with a probability of at most* ϵ*, where* C *is generated using a random sample of labeled points of size at least*

$$\max(\frac{4}{\epsilon}\log\frac{2}{\delta}, \frac{8d}{\epsilon}\log\frac{13}{\epsilon})$$

.

Since the VC dimension for the class of halfspaces in $\mathbb{R}^n$ is $n+1$, a halfspace can be learnt in PAC learning framework using a sample of size at least

$$\max(\frac{4}{\epsilon}\log\frac{2}{\delta}, \frac{8n+8}{\epsilon}\log\frac{13}{\epsilon})$$

This, learning halfspaces in $\mathbb{R}^n$ requires samples polynomial in $n, \frac{1}{\epsilon}$ and $\frac{1}{\delta}$ and by increasing the probabilistic accuracy of the learnt halfspace requires polynomial increase in the number of samples. This is critical for efficiently learning guards in our algorithm.

We first discuss *Halfspace learning algorithm:* `HSinfer` which, as the name suggests, can be used to learn halfspaces in the PAC framework from a given sample and then, describe the switching logic synthesis algorithm. In $\mathbb{R}^n$, a halfspace is given by $\theta \cdot X + \theta_0 \geq 0$ where $\theta \in \mathbb{R}^n, \theta \in \mathbb{R}$ and X is any point in $\mathbb{R}^n$ which satisfied the above inequality if and only if the point is in the concept halfspace to be learnt. For any point X_i, let Y_i be 1 if X_i is in the concept and -1 if it is outside the concept. The algorithm below is the standard *Perceptron Learning* algorithm and is known to converge after k iterations where $k \leq (max_i||X_i||)/(\min_i \frac{Y_i(\theta X_i+\theta_0)}{||\theta||})^2$ [36].

We now describe the algorithm to *learn guards* for multiple initial states using the technique presented in Section 7.2.2 and the halfspace learning algorithm `HSinfer`. The algorithm simply involves finding optimal switching points for each mode-switch and then using halfspace learning to infer the guards. The key idea is to use the optimal switching states as *positive* points for the concept learning problem and the non-optimal states explored during optimization (which preceded the optimal switching states along any trajectory) as *negative* points since these states cannot be in the guard for an optimal switching logic.

Procedure 16 Halfspace learning algorithm `HSinfer` [36]

Input: Set of labeled sample points $\{(X_i, Y_i)\}$

Output: θ, θ_0 such that $\theta X + \theta_0 \geq 0$ is the halfspace Set $\theta^0 = 0, \theta_0^0 = 0, t = 0$

for each i **do**
  if $\theta^0 X + \theta_0^0 \geq 0$ **then**
    Predicted $y_i = 1$
  else
    Predicted $y_i = -1$
  end if
end for
while some i has $Y_i \neq y_i$ **do**
  pick some i with $Y_i \neq y_i$
  $\theta^{t+1} := \theta^t + Y_i X_i$
  $\theta_0^{t+1} := \theta_0^t + Y_1$
  $t := t + 1$
  for each i **do**
    if $\theta^t X + \theta_0^t \geq 0$ **then**
      Predicted $y_i = 1$
    else
      Predicted $y_i = -1$
    end if
  end for
end while
return θ, θ_0

Procedure 17 Finding optimal switching logic SL^{opt}

Input: $MDS(X, Q)$, initial states I, tolerance of generalization error δ and maximum probability of error ϵ

Output: Optimal Switching Logic SL^{opt}

1. Sample initial states from I for provided δ, ϵ.
2. For each initial state, obtain optimal trajectory in $MDS(X, Q)$ and switching states for the mode switches along the trajectory.
3. Label the obtained switching states as positive points.
4. Label the states preceding switching states along any trajectory to be negative points.
5. Using obtained sample of positive and negative states, learn the guard for the mode switch as generalization of these states using `HSinfer`.
6. Output these guards as synthesized optimal switching logic SL^{opt}.

We now discuss the guarantees provided by our technique. Under the structure assumption that guards are halfspaces, our PAC learning algorithm algorithm computes guards with probability at least $1-\delta$ such that the probability that a guard contains any state which is not a switching-state or misses any switching-state is at most ϵ. Further, the guards inferred by the above algorithm can be made probabilistically more and more accurate by choosing suitable values of ϵ, δ and considering correspondingly larger and larger samples of initial states as given by Theorem 12. For a trajectory to be a non-optimal trajectory, any one switching point along the trajectory needs to be classified correctly. Thus, the following theorem establishes the probabilistic guarantees of our switching logic synthesis algorithm.

Theorem 13. *Given a* $\texttt{MDS}(X, Q)$*, using random sampling from the set of initial states which has a sample size polynomial in* $n, \frac{1}{\epsilon}$ *and* $\frac{1}{\delta}$*, Algorithm 17 synthesizes a switching logic* `SwL` *with probability at least* $1-\delta$ *such that any trajectory in the synthesized hybrid system* $\texttt{HS}(\texttt{MDS}, \texttt{SwL})$ *is not optimal with probability at most* $m\epsilon$*, where* m *is the number of guards in the switching logic, that is,* $m = |\texttt{SwL}| \leq |Q|^2$ *and* n *is the number of variables, that is,* $n = |X|$.

Running Example

Given the set of initial states $16 \leq \texttt{temp} \leq 26$ and $\texttt{out} \in \{16, 26\}$. The set of initial states is partitioned into subsets where each subset is a 0.1 interval of room temperature `temp` and the

outside temperature is 16 or 26. The guards discovered are: $\mathtt{g}_{HF} : \mathtt{temp} \geq 20.2 \wedge \mathtt{out} = 16$, $\mathtt{g}_{FH} : \mathtt{temp} \leq 19.6 \wedge \mathtt{out} = 16$, $\mathtt{g}_{CF} : \mathtt{temp} \leq 20.0 \wedge \mathtt{out} = 26$, $\mathtt{g}_{FC} : \mathtt{temp} \geq 20.3 \wedge \mathtt{out} = 26$.

7.3 Results and Experiments

Apart from the running example of Thermostat controller, we applied our technique to three other case studies: (i) an Oil Pump Controller, which is an industrial case study from [21], and (ii) a DC-DC Buck-Boost Converter, motivated by the problem of minimizing voltage fluctuation in a distributed aircraft power system, and (iii) an air-handling unit in buildings. We employ the implementation of Nelder-Mead simplex algorithm as described by Lagarias et al [74, 101] and available as the $fminsearch$ [91] function in MATLAB for numerical optimization. All experiments were performed on a dual-core 1.66GHz processor with 4GB memory.

7.3.1 Thermostat Controller

If we change the cost metric in the thermostat controller to $\lim_{t\to\infty} \frac{\mathtt{discomfort}(t)+\mathtt{fuel}(t)+\mathtt{swTear}(t)}{\mathtt{time}(t)}$ giving equal weight to all the three penalties (instead of $10 : 1 : 1$ weight ratio used earlier) the optimal switching logic discovered with this cost metric are: $\mathtt{g}_{HF} : \mathtt{temp} \geq 20.0 \wedge \mathtt{out} = 16$, $\mathtt{g}_{FH} : \mathtt{temp} \leq 18.8 \wedge \mathtt{out} = 16$, $\mathtt{g}_{CF} : \mathtt{temp} \leq 21.9 \wedge \mathtt{out} = 26$, $\mathtt{g}_{FC} : \mathtt{temp} \geq 22.7 \wedge \mathtt{out} = 26$. We observe that the room temperature oscillates closer to the target temperature when the discomfort penalty is given relatively higher weight in the cost metric. This case study illustrates that a designer can suitably define a cost metric which reflects their priorities and, then, our technique can be used to automatically synthesize switching logic for the given cost metric. The runtime for simulation and numerical optimization is 284 seconds.

7.3.2 Oil Pump Controller

Our second case study is an Oil Pump Controller, adapted from the industrial case study in [21]. The example consists of three components - a machine which consumes oil in a periodic

manner, a reservoir containing oil, an accumulator containing oil and a fixed amount of gas in order to put the oil under pressure, and a pump. The simplification we make is to use a periodic function to model the machine's oil consumption and we do not model any noise (stochastic variance) in oil consumption.

The state variable is the volume V of oil in the accumulator. The system has two modes: mode `ON` when the pump is ON and mode `OFF` when the pump is OFF. Let the rate of consumption of oil by the machine be given by $m = 3 * (cos(t) + 1)$ where t is the time. The rate at which oil gets filled in the accumulator is p. $p = 4$ when the pump is on and $p = 0$ when the pump is off. The change in volume of oil in the accumulator is given by the following equation $\dot{V} = p - m$ where p and m take different values depending on the mode of operation of the pump. For synthesis, we consider two different sets of requirements [21].

In the first set of requirements, the volume of oil in the tank must be within some safe limit, that is, $1 \leq V \leq 8$ and the average volume of oil in the accumulator should be minimized. We model these requirements using our cost definition by defining one penalty variable p_1 and one reward variable r_1. Let the evolution of penalty p_1 be $\dot{p}_1 = V$ if $1 \leq V \leq 8, \quad M$ otherwise where M is a very large ($M \geq 10^5 p_1$) constant (10^6 in our experiments) and that of reward r_1 be $\dot{r}_1 = 1$. Minimizing the cost function $cost1 = \lim_{t\to\infty} \frac{p_1(t)}{r_1(t)}$ minimizes the average volume $\lim_{t\to\infty} \frac{\int_0^t V(t)}{t}$ and also enforces the safety requirement $\forall t \ . \ 1 \leq V(t) \leq 8$.

In the second set of requirements, we add an additional requirement to those in the first set. We require that the the oil volume is mostly below some threshold $V_{high} = 4.5$ in the long run. We model this requirement by adding an additional penalty and an additional reward variable p_2 and r_2 with evolution functions: $\dot{p}_2 = 1$ if $V > V_{high}, \quad 0$ otherwise and $\dot{r}_2 = 1$ if $V < V_{high}, \quad 0$ otherwise. The new cost function is $cost2 = \lim_{t\to\infty}(\frac{p_1(t)}{r_1(t)} + \frac{p_2(t)}{r_2(t)})$. Let t_{high} be the total duration when the volume is above V_{high} and t_{low} be the duration that it is below V_{high}. Minimizing $p_2/r_2 = t_{high}/t_{low}$ would ensure that we spend more time with volume less than V_{high} in the accumulator.

The guards: g_{FN} from `OFF` to `ON` and g_{NF} from `ON` to `OFF` obtained for the above $cost1$ objective are $\mathsf{g}_{FN} : V \leq 3.71 \quad \mathsf{g}_{NF} : V \geq 4.62$ and for $cost2$ objective are $\mathsf{g}_{FN} : V \leq 4.07 \quad \mathsf{g}_{NF} : V \geq 4.71$. The runtimes are 438 seconds and 479 seconds respectively.

We simulate from an initial state $V = 4$ and the behavior for both objectives is presented in

Figure 7.3.2. In both cases, the behavior satisfies the safety property that the volume is within 1 and 8. Since, we minimize oil volume, the volume is close to the lower limit of 1. We also observe that using the second cost metric causes decrease in duration of time when oil volume is higher than the 4.5 but the average volume of oil increases. This illustrates how designers can use different cost metrics to better reflect their requirements.

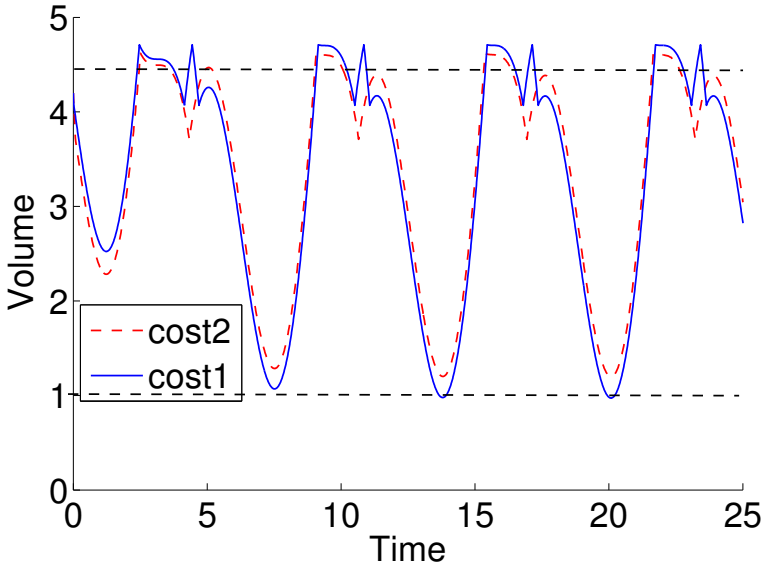


Figure 7.2: Oil Pump Controller: Volume in accumulator

7.3.3 DC-DC Buck-Boost Converter

In this case study, we synthesize switching logic for controlling DC-DC buck-boost converter circuits described in [48]. The goal is to maintain the output voltage V_R across a varying load R at some target voltage V_d. The converter can be modeled as a hybrid system with three modes of operation. The state space of the system is $X = \langle i_L \ u_C \rangle$ where i_L is the current through the inductor and u_C is the load voltage. The dynamics in the three modes are given by the state space equation $\dot{X} = A_k X + B_k E$ where $k = 1, 2, 3$ is the mode and E is the input voltage. The

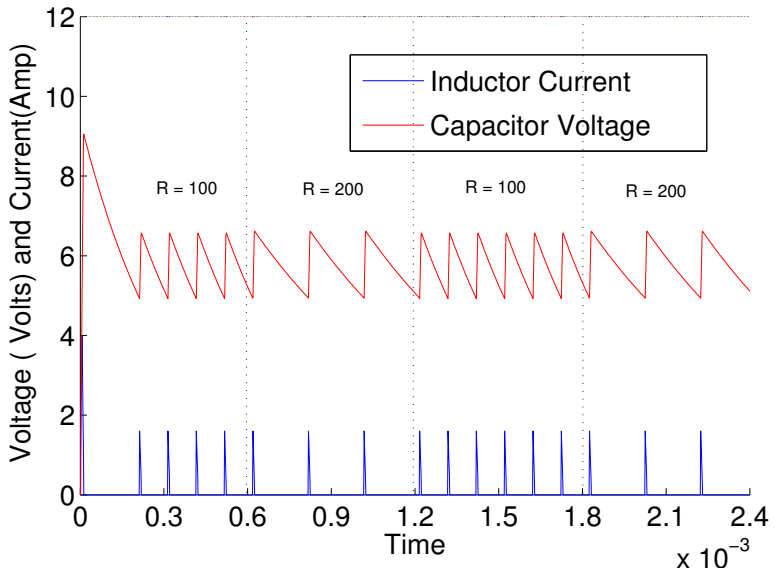


Figure 7.3: DC-DC Boost Converter

coefficients of the equations are

$$A_1 = \begin{bmatrix} \frac{-rL-rs}{L} & 0 \\ 0 & \frac{-1}{C*(R+rC)} \end{bmatrix}, \quad B_1 = \begin{bmatrix} \frac{1}{L} \\ 0 \end{bmatrix},$$

$$A_2 = \begin{bmatrix} \frac{-rL-rd}{L} & \frac{-1}{L} \\ \frac{R}{R+rC} * (\frac{1}{C} - \frac{rC*(rL+rd)}{L}) & \frac{-R}{R+rC} * (\frac{rC}{L} + \frac{1}{R*C}) \end{bmatrix},$$

$$B_2 = \begin{bmatrix} \frac{1}{L} \\ \frac{R}{(R+rC)*\frac{rC}{L}} \end{bmatrix}, \quad A_3 = \begin{bmatrix} 0 & 0 \\ 0 & \frac{-1}{(R+rC)*C} \end{bmatrix}, \quad B_3 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

We mention two key performance requirements of the DC-DC Boost Converter described in [48]. The first requirement is that the converter be resilient to load variations. The second requirement is to minimize the variance of the voltage across the load V_R from the target voltage. This variance is called the ripple voltage. We define penalty variable p_1 with the following evolution functions: $\dot{p_1} = (V_R - V_d)^2$. We want to minimize the average deviation from the target voltage. So, we define the reward variable r_1 with $\dot{r}_1 = 1$. The cost function is $\texttt{cost} = \lim_{t\to\infty} \frac{p_1(t)}{r_1(t)}$. This minimizes the average variance of V_R from the target voltage V_d. This corresponds to minimizing the ripple voltage. Since the load R also changes periodically, it also minimizes the transient variance in voltage.

Given the dynamics in each of the three modes and the cost function, the synthesis problem is to automatically synthesize the guards $\mathrm{g}_{12}, \mathrm{g}_{23}, \mathrm{g}_{31}$ which minimizes the cost. We are given the over-approximation of the guard $\mathrm{g}_{23}^{over} : i_l = 0$. The guards obtained are as follows: $\mathrm{g}_{12} : i_L > 1.9$, $\mathrm{g}_{23} : i_L = 0$ and $\mathrm{g}_{23} : v_C > 4.6$. The runtime is 394 seconds. The system remains in the first mode until the inductor current reaches the reference current I_{ref}. The system remains in the second mode until the inductor current becomes 0. Then, the system switches to the third mode where it remains as long as the capacitor voltage remains over the reference voltage V_{ref}. We simulate the synthesized system and the behavior is shown in Figure 7.3.

7.3.4 Air Handling Unit in Buildings

We consider a switched model of air handling unit in a building presented by Haghighi et al [84]. The original model inferred from real data on the campus of University of California at Berkeley, USA is non-deterministic. We choose unique value of parameters in the model and eliminate noise to make the model deterministic. The model consists of a set of thermal zones and an air handling unit supplying air to these zones. Air handling unit (AHU) serves different thermal zones (called "rooms" for brevity) with a mixture of outside air and return air which is heated or cooled depending on target temperature and outside temperature. Each room j is modeled as a two-mass system: C_1^j is the fast-dynamic mass (e.g. air around the diffusers of AHU) that has low heat capacity, and C_2^j represents the slow-dynamic mass (e.g. the solid part which includes floors, walls and furniture) that has higher heat capacity. The phenomenon of fast and slow dynamics has been presented previously by Shavit et al [45]. The temperature of low heat capacity mass is T_1^j and the temperature of high heat capacity mass is T_2^j. The perceived temperature T_j of room j is assumed to be equal to the temperature of the mass of lower heat capacity, that is, $T_j = T_2^j$. The air enters the room j with a mass flow rate $\dot{m}^j$ and temperature $T_s^j = T_{oa} - \Delta$, where T_{oa} is the outside temperature and Δ is the temperature of the AHU cooling/heating coil. $\mathcal{N}^j$ is the set of all rooms neighboring room j. Figure 7.4 illustrates the model for two rooms.

The thermodynamic model of room j is given by the following ordinary differential equations.

$$C_1^j \dot{T}_1^j = \dot{m}^j C_p T_s^j + \frac{T_2^j - T_1^j}{R^j} + \sum_{i \in \mathcal{N}^j} \frac{T_1^i - T_1^j}{R^{ij}} + \frac{T_{oa} - T_1^j}{R_{oa}^j}$$

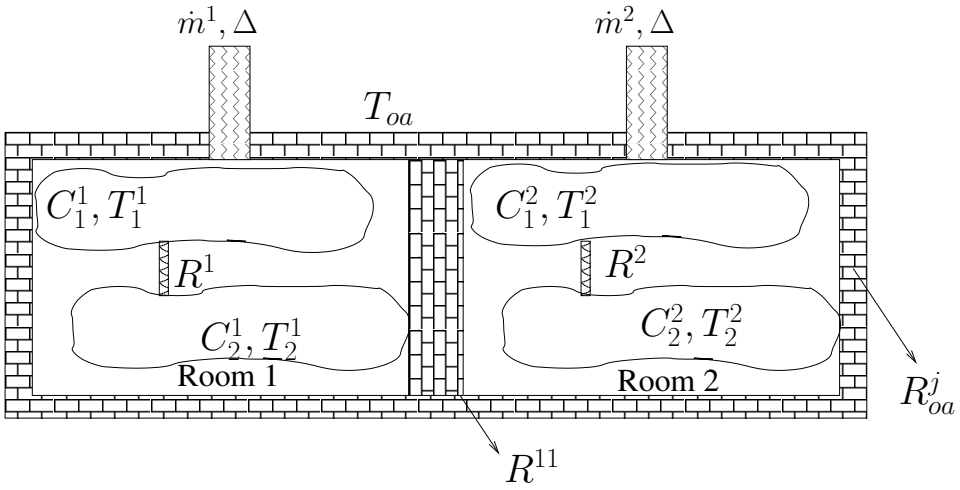


Figure 7.4: Two Rooms

$$C_2^j \dot{T}_2^j = \frac{T_1^j - T_2^j}{R^j}$$

The outside temperature T_{oa} is $300K$ and the target temperature T_g for all rooms is $290K$. The initial room temperature is also $290K$ for both low and high heat capacity masses. Each room can be either in cooling mode or the heating mode. $\Delta = 6.3$ in the cooling mode and 0 in the heating mode. The other parameters used in the model are given in the table below. $C_1^j, C_2^j, R^j, R_{oa}^j, R^{ij}$ are the same as the original model [84, 83], and $\dot{m}^j$ is the mid-value of the interval in the original model.

Parameter	Value	Parameter	Value
C_1^j	9.163×10^3 kJ/K	C_2^j	1.694×10^5 kJ/K
R^j	1.700×10^{-3} K/W	R_{oa}^j	5.700×10^{-2} K/W
R^{ij}	2.000×10^{-3} K/W	$\dot{m}^j$	1.250 kg/s

The goal of the switching logic controller is to ensure that the temperature in each room is at the target temperature. We associate the following penalty for each room j:

$$\texttt{penalty}^{\texttt{j}} = \alpha\ \texttt{discomfort}^{\texttt{j}} + \texttt{fuel}^{\texttt{j}}$$

where $\texttt{discomfort}^{\texttt{j}}$ represents the penalty of the temperature in room j being away from the target temperature, and $\texttt{fuel}$ is the fuel cost of cooling room j, that is, $\dot{\texttt{discomfort}}^{\texttt{j}} = 2|T^j - T_g|$

and $\dot{\texttt{fuel}}^{\texttt{j}} = \dot{m}^j\Delta$. α is a *weighing factor* used to give different weights to discomfort and fuel cost. The total penalty is the sum of penalty for each room, that is, $\texttt{penalty} = \sum_j \texttt{penalty}^{\texttt{j}}$. The reward $\texttt{reward}^{\texttt{j}}$ is the time, that is, $\dot{\texttt{reward}}^{\texttt{j}} = 1$. The total reward is the sum of rewards for each room. Thus, the cost represents the average weighted discomfort and fuel consumption per room. We use different weighing parameter α in our experiments and illustrate how the optimal switching logic varies with different choices of α.

In general, each room can switch from heating to cooling mode and back at different temperatures. We call the model where each room independently switches from heating to cooling mode and vice versa as the *independent model*. Secondly, we consider a *symmetric model* in which all rooms switch modes at the same temperature. In the first case, the space of switching parameters grows linearly with the number of rooms in the building. In the second case, the space of switching parameters is only 2, independent of the number of rooms in the building. For both these models, we consider two scenarios: a building with 3 rooms and 5 rooms. We denote the temperature at which room j switches from heating to cooling mode by T_c^j and the one at which it switches from cooling mode to heating by T_h^j. In case of *symmetric model*, since all the rooms switch into cooling or heating mode at same temperatures, we drop the superscripts and denote the respective switching temperatures by T_c and T_h.

Three Rooms

First, we consider a model with three rooms which are all connected to each other with respect to heat-transfer. The results of our synthesis approach is presented in the table below.

α	Model	Switch	Cost	Runtime(s)
2.5	Independent	$290.26, 289.79, 291.01, 289.90, 295.51, 286.36$	16.32	860.74
	Symmetric	$T_c = 290.06, T_h = 289.96$	12.71	721.65
1	Independent	$293.83, 289.84, 293.81, 289.82, 293.84, 282.55$	14.20	892.54
	Symmetric	$T_c = 293.20, T_h = 289.85$	13.25	547.20
1/2.5	Independent	$299.05, 289.63, 299.04, 280.20, 297.93, 281.40$	24.72	3437.24
	Symmetric	$T_c = 298.11, T_h = 298.01$	17.16	223.60

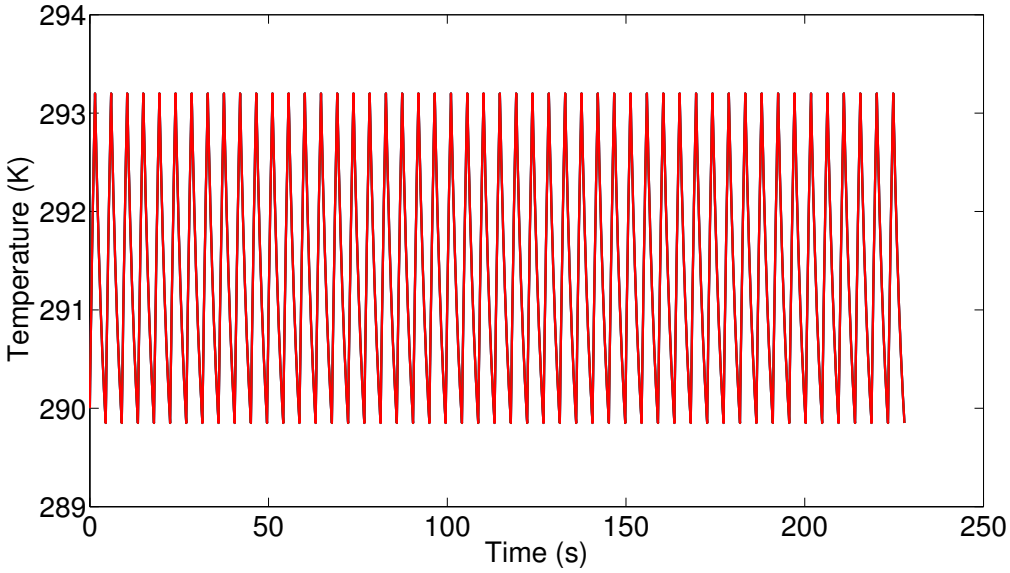
We make the following observations about the synthesized switching logic using both the independent and the symmetric model. We observe that using the symmetric model helps the

minimization engine to discover switching logic of lesser cost that the independent model in all the three cases. It also takes less time to discover switching logic of minimal cost using the symmetric model. The increase in the number of parameters for minimization in the independent model results into the Nelder-Mead method [74, 101] getting stuck at a local minimum which is of higher cost than the switching logic discovered using the symmetric model. The behavior of the system and the cost of the behavior for the case where $\alpha = 1$ is also shown in Figure 7.5 and Figure 7.6. We observe that in the independent model for $\alpha = 1$, one of the rooms is never cooled externally and instead, relies on the heat transfer with other two rooms to keep the temperature of the room close to the target temperature. Such behavior is possible only in the independent model. Thus, independent model allows greater dexterity in the choice of switching logic than the symmetric model. But the limitation of minimization engine leads to the discovery of higher cost switching logic for independent model than the one for symmetric model. Further, we observe that the synthesized switching logic results into a system where the room temperature stays close to the target temperature (290) when the cost of discomfort is much higher than that of the cost of the fuel ($\alpha = 2.5$), and it stays close to the outside temperature when the cost of fuel is much higher than discomfort ($\alpha = 1/2.5$). The room temperature oscillates close to the target temperature when the cost of discomfort and fuel is given equal weight ($\alpha = 1$).

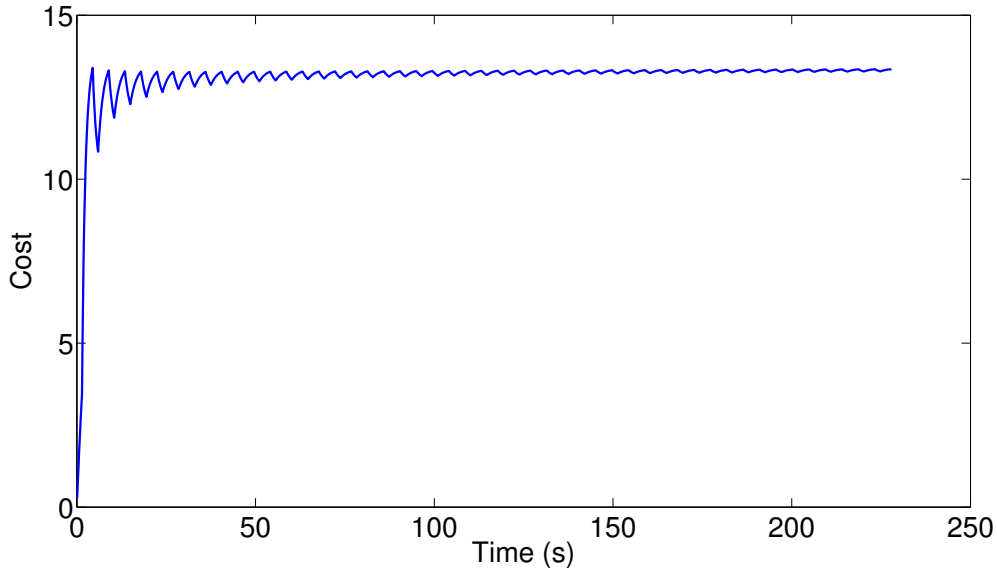
Five Rooms

Next, we consider a model with five rooms which are all connected to each other with respect to heat-transfer. The results of our synthesis approach is presented in the table below.

α	Model	Switch	Cost	Runtime(s)
2.5	Independent	$293.20, 289.33, 292.52, 289.32, 293.26, 289.34$ $297.96, 287.70, 294.46, 286.04$	18.61	1752.70
	Symmetric	$T_c = 290.06, T_h = 289.96$	12.71	1443.2
1	Independent	$293.56, 289.51, 293.56, 289.51, 293.55, 289.50$ $298.48, 288.39, 293.19, 283.76$	29.21	1850.21
	Symmetric	$T_c = 293.20, T_h = 289.85$	13.25	862.56
1/2.5	Independent	$298.93, 294.71, 299.12, 291.84, 297.70, 271.30$ $297.55, 283.59, 296.69, 281.26$	25.68	2909.35
	Symmetric	$T_c = 298.11, T_h = 298.01$	17.16	290.70

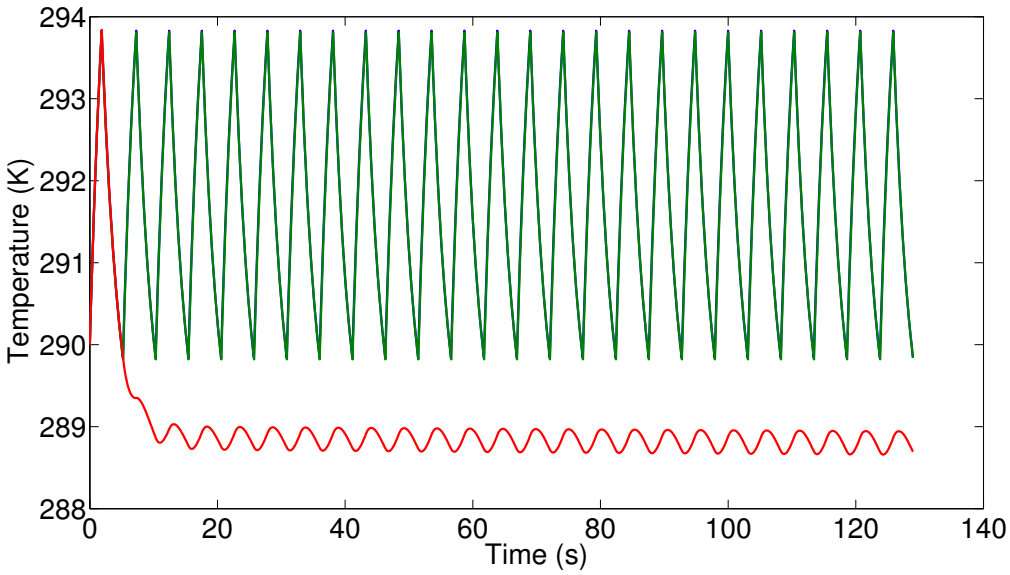


(a) Temperature of 3 Rooms: Temperature of each room is shown in different color

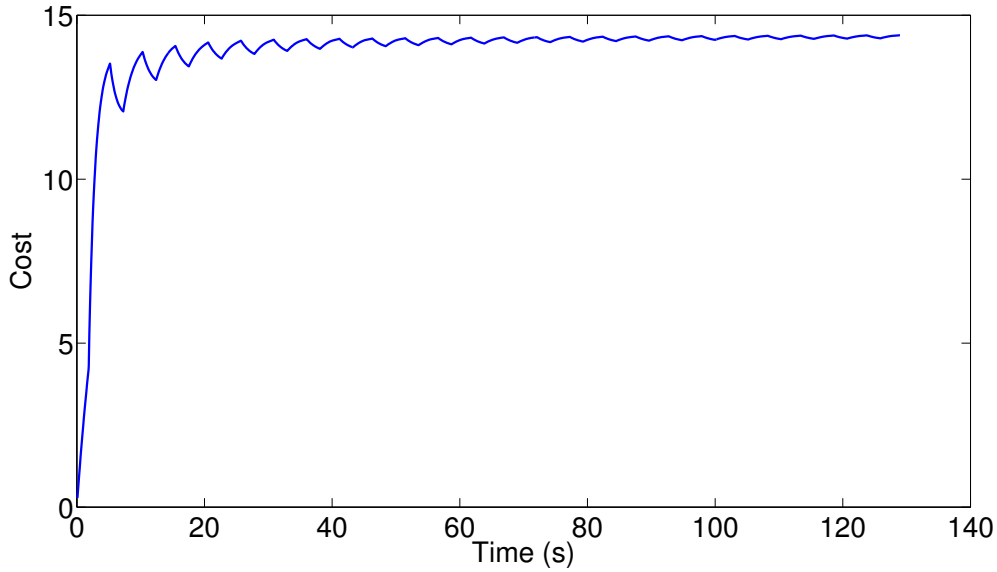


(b) Cost

Figure 7.5: 3 Rooms in Symmetric Model

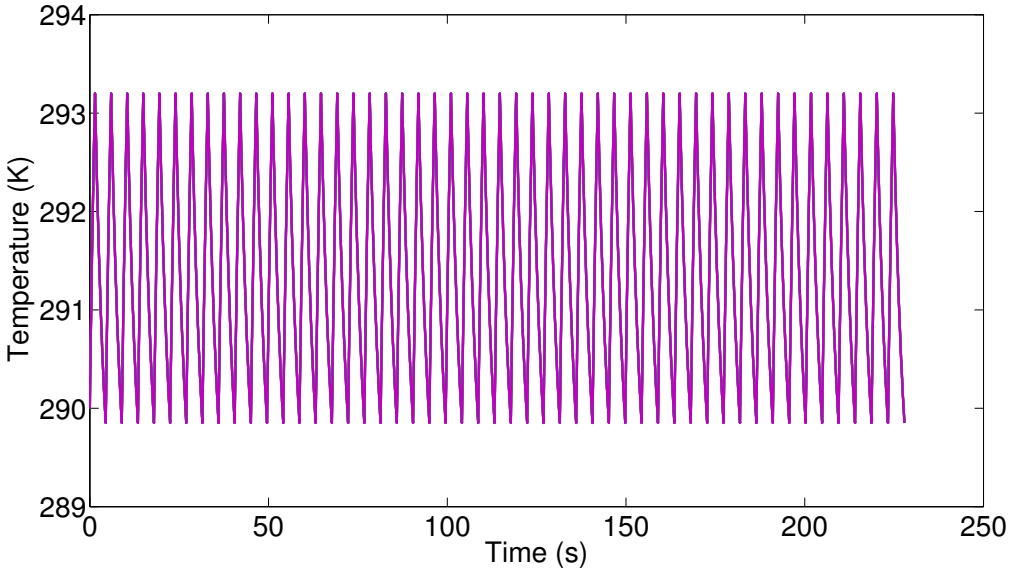


(a) Temperature of 3 Rooms: Temperature of each room is shown in different color

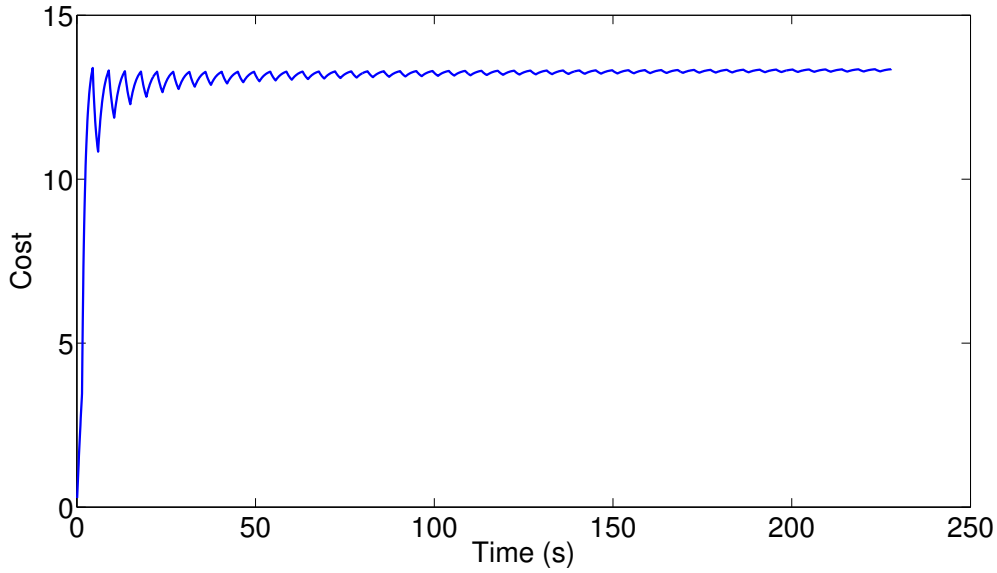


(b) Cost

Figure 7.6: 3 Rooms in Independent Model

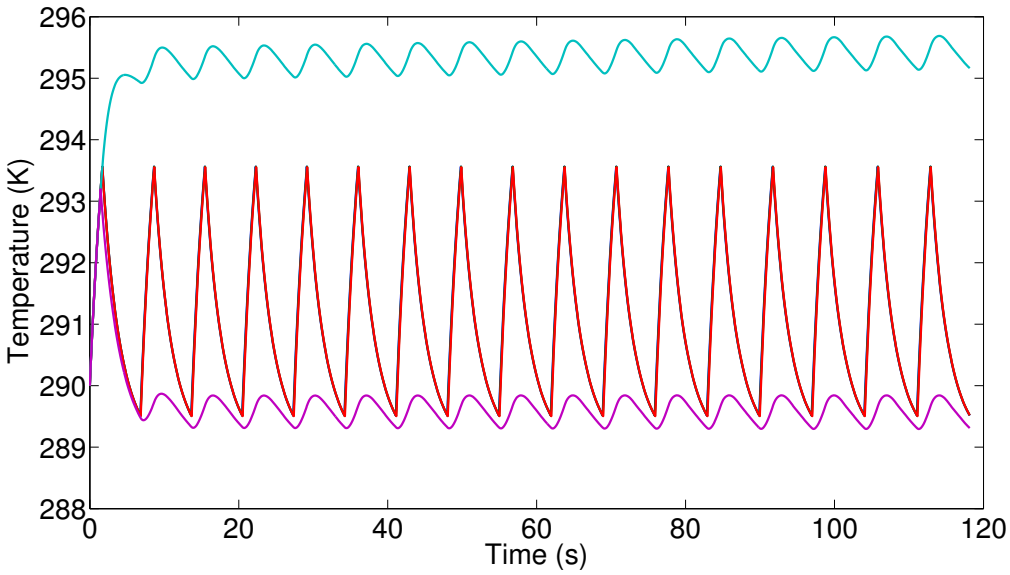


(a) Temperature of 5 Rooms: Temperature of each room is shown in different color

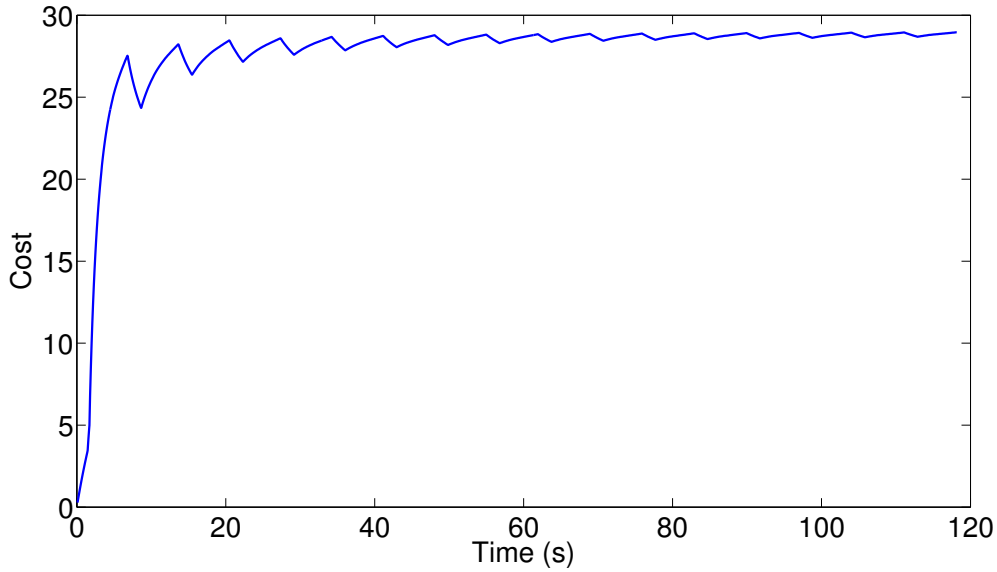


(b) Cost

Figure 7.7: 5 Rooms in Symmetric Model



(a) Temperature of 5 Rooms: Temperature of each room is shown in different color



(b) Cost

Figure 7.8: 5 Rooms in Independent Model

The observations about the synthesized switching logic using both the independent and the symmetric model is similar to the case with 3 rooms. We observe that using the symmetric model again helps the minimization engine to discover switching logic of lesser cost than the independent model in all the three cases. It also takes less time to discover switching logic of minimal cost using the symmetric model. This was also the case with 3 rooms. The behavior of the system and the cost of the behavior for the case where $\alpha = 1$ is also shown in Figure 7.7 and Figure 7.8. We observe that in the independent model for $\alpha = 1$, two of the rooms are never cooled externally. They rely on the heat transfer with other three rooms to keep the temperature of the rooms close to the target temperature. This is a consequence of the minimization engine failing to further minimize the cost of the switching logic and getting stuck in the local minimum. Further, we observe that the synthesized switching logic results into a system where the room temperature stays close to the target temperature (290) when the cost of discomfort is much higher than that of the cost of the fuel ($\alpha = 2.5$), and it stays close to the outside temperature when the cost of fuel is much higher than discomfort ($\alpha = 1/2.5$). The room temperature oscillates close to the target temperature when the cost of discomfort and fuel is given equal weight ($\alpha = 1$). This observation is again similar to the case with three rooms.

7.4 Conclusion

In this chapter, we presented an algorithm for automated synthesis of switching logic in order to achieve minimum long-run cost. Our algorithm is based on reducing the switching logic synthesis problem to an unconstrained numerical optimization problem which can then be solved by existing optimization techniques. We also give a learning-based approach to generalize from a sample of switching states to a switching condition, where the learnt condition is optimal with high probability.

Chapter 8

Conclusion

This chapter summarizes the main results presented in this thesis and suggests avenues of future work.

8.1 Summary

This thesis proposes a unified theme for automated synthesis that combines deductive reasoning techniques with inductive inference techniques based on algorithmic learning. The central idea is to use deduction to discover example behaviors of the desired system, and then use induction to synthesize the system as generalization of the discovered behaviors. Hypotheses on the structure of the system can be provided by users to aid the induction engine in the generalization step, and to constrain the search space of deduction engines. This systematic combination of induction, deduction, and structure hypotheses, termed SCIDUCTION, is used for automating tricky and tedious tasks in system design. This paradigm of SCIDUCTION builds on successful deductive techniques, such as satisfiability solving [13, 37], verification techniques [31] and numerical optimization [74], with inductive techniques such as algorithmic concept learning [6, 94]. Different instantiations of this paradigm by combining different inductive and deductive reasoning techniques were used to solve synthesis problems in different application domains. We use our technique for automated synthesis of loop-free programs from black-box oracle specifications using functions from a library of component functions, synthesizing optimal cost floating-point

code with specified accuracy from floating-point code, and synthesizing switching logic of hybrid systems for safety and performance properties. These applications illustrate the practical utility of the proposed synthesis approach based on SCIDUCTION.

8.2 Future Work

In this section, we identify present avenues for future work in automated program synthesis as well as automated synthesis of switching logic.

8.2.1 Program Synthesis

1. **Discovering composition of high-level API calls:**
   In practice, software development often involves building new applications using existing code packaged as application program interface (API). For example, scientific code relies on nonlinear algebra and statistical packages and web-services are built using composition of web-modules. Programming languages often provide large library of functions for different tasks in addition to the core language. Languages such as Python, C++ and Java have gained widespread use because of their support for libraries. Automated program synthesis techniques that can compose API functions using function summaries would be very useful to programmers and developers. This will require improvement in deductive reasoning techniques to support theories in which API function summaries can be expressed. For example, web-modules require effective decision procedures for the theory of strings.

2. **Synthesis of parallel implementation from sequential code:**
   One of the dreams of parallel and concurrent computing has being to build a compiler which takes sequential code as an input and produces a semantically equivalent but efficient parallel program. The increasing use of multi-core processors has further fueled this quest. While it is hard to exactly characterize the efficiency of a parallel program for all inputs, it is relatively easy to run the program on a sample input and observe its performance. Hence, use of a technique that combines induction from examples and deductive program synthesis could provide an interesting approach to this long-standing problem.

3. **Interactive synthesis of user-interface design:**
   Computational devices are now ubiquitous and this has made the design of user-interface become more important. It is often difficult to have complete user-interface specifications. A black-box oracle based approach that interacts with user to resolve different design choices of user-interface would be useful in creating more user friendly user-interface designs. Such a technique could be used to personalize websites and cell-phone applications for different users. This requires identification of different design choices in user-interface design and formal expression of the relation between different design choices. Since an user might make different choice for same query at different time instants, the synthesis technique must support an *imperfect* oracle specification which is not guaranteed to make similar choices for same queries. Hence, the goal of the synthesis algorithm would be to construct an user-interface which is most likely going to satisfy the user.

8.2.2 Synthesis of Switching Logic

1. **Synthesis in presence of uncertainty:**
   We assume in our work on switching logic synthesis that the dynamics in each mode of operation of the system is deterministic and known to us. In practice, the dynamics might be stochastic due to inherent uncertainty in the system parameters or due to our limited knowledge about them. Biological systems and market models in computational finance are two examples of such scenarios. Automated synthesis of switching logic for such systems would require reachability analysis for stochastic hybrid systems. Further, the guarantee of safety would only be probabilistic, and the synthesis task would be to construct a system which is *likely* to remain safe.

2. **Exploiting symmetry in presence of large number of modes:**
   Many systems have inherent symmetry in them. For example, consider a building with a large number of identical rooms, and with each room having an identical heater with a switch which can be on or off. If there are k such rooms, 2^k configurations of the building are possible depending on whether heater is on or off in different rooms. But since the rooms are identical, the control law for governing each of the heaters in the room is expected to be similar. A switching logic synthesis approach that can exploit such symmetry would enable

automated synthesis for systems with very large number of modes.

3. **Actuation and sensing delays:**
   Using hybrid systems as model of cyber-physical systems assumes that the system variables are immediately observed and the controlled variables can be changed instantly. In practice, observation of system variables takes place through a network of sensors and there is a non-zero latency in observing the variables and reporting it to the controller. The control decision also takes time to be implemented. Another dimension to extend our work is to take into account these actuation and sensing delays during synthesis. These delays would be inherently non-deterministic and the synthesis techniques must also address the uncertainty in these delays.

In conclusion, we presented a novel automated synthesis paradigm in this thesis and illustrated its utility on diverse applications. Further progress on automated synthesis can be achieved by pursuing this combination of induction and deduction using the structure hypothesis on the system under synthesis.

Bibliography

[1] Matlab: Fixed-point toolbox. `http://www.mathworks.com/help/toolbox/fixedpoint`.

[2] IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Std 754-1985*, 1985.

[3] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, February 1995.

[4] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, pages 209–229, 1993.

[5] Rajeev Alur and Thomas A. Henzinger. Modularity for timed and hybrid systems. In *CONCUR*, pages 74–88. Springer-Verlag, 1997.

[6] Dana Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1987.

[7] Dana Angluin and Carl H. Smith. Inductive inference: Theory and methods. *ACM Comput. Surv.*, 15:237–269, September 1983.

[8] E. Asarin, O. Bournez, T. Dang, O. Maler, and A. Pnueli. Effective synthesis of switching controllers for linear systems. *Proc. of the IEEE*, 88(7):1011–1025, 2000.

[9] Eugene Asarin, Thao Dang, Oded Maler, and Olivier Bournez. Approximate reachability analysis of piecewise-linear dynamical systems. In *Proceedings of the Third International Workshop on Hybrid Systems: Computation and Control*, HSCC, pages 20–31, London, UK, 2000. Springer-Verlag.

[10] H. Axelsson, Y. Wardi, M. Egerstedt, and E. Verriest. Gradient descent approach to optimal mode scheduling in hybrid dynamical systems. *Journal of Optimization Theory and Applications*, 136:167–186, 2008.

[11] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In *ASPLOS*, 2006.

[12] Sorav Bansal and Alex Aiken. Binary translation using peephole superoptimizers. In *OSDI*, 2008.

[13] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 4, chapter 8. IOS Press, 2009.

[14] Roderick Bloem, Krishnendu Chatterjee, Thomas A. Henzinger, and Barbara Jobstmann. Better quality in synthesis through quantitative objectives. In *CAV*, pages 140–156, 2009.

[15] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K. Warmuth. Learnability and the Vapnik-Chervonenkis dimension. *J. ACM*, 36(4):929–965, 1989.

[16] P.P. Boca, J.P. Bowen, and J.I. Siddiqi. *Formal Methods: State of the Art and New Directions*. Springer, 2009.

[17] J.F. Bonnans, J.Ch. Gilbert, C. Lemaréchal, and C. Sagastizábal. *Numerical Optimization – Theoretical and Practical Aspects, Second Edition*. 2006.

[18] M. Branicky, V. Borkar, and S. Mitter. A unified framework for hybrid control: Model and optimal control theory. *IEEE Trans. on Aut. Cntrl.*, 43(1):31–45, 1998.

[19] J.R. Buchanan. *A study in automatic programming*. Memo (Stanford Artificial Intelligence Laboratory). Stanford University, 1974.

[20] T.H. Bui, T.T. Nguyen, T.L. Chung, and S.B. Kim. A simple nonlinear control of a two-wheeled welding mobile robot. *INTERNATIONAL JOURNAL OF CONTROL, AUTOMATION AND SYSTEMS*, 1, 2003.

[21] Franck Cassez, Jan Jakob Jessen, Kim Guldstrand Larsen, Jean-François Raskin, and Pierre-Alain Reynier. Automatic synthesis of robust and optimal controllers - an industrial case study. In *HSCC*, pages 90–104, 2009.

[22] Donald Chai and Andreas Kuehlmann. A fast pseudo-boolean constraint solver. In *Proceedings of the 40th annual Design Automation Conference*, DAC '03, pages 830–835. ACM, 2003.

[23] A. Charnes and C. E. Lemke. Minimization of non-linear separable convex functionals. *Naval Research Logistics Quarterly*, 1(4):301–312, 1954.

[24] K. Chatterjee. Markov decision processes with multiple long-run average objectives. In *FSTTCS*, pages 473–484, 2007.

[25] E.M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 1999.

[26] Jonathan A. Clarke, Altaf Abdul Gaffar, George A. Constantinides, and Peter Y. K. Cheung. Fast word-level power models for synthesis of FPGA-based arithmetic. In *ISCAS*, 2006.

[27] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, Dept. Comp. Sci., The Univ. of Auckland, July 1997.

[28] J. Cury, B. Brogh, and T. Niinomi. Supervisory controllers for hybrid systems based on approximating automata. *IEEE Trans. Aut. Cntrl.*, 43:564–568, 1998.

[29] A. Cypher, editor. *Watch what I do: Programming by demonstration*. MIT Press, 1993.

[30] A. Donze and O. Maler. Systematic simulation using sensitivity analysis. In *HSCC*, volume 4416 of *LNCS*, pages 174–189, 2007.

[31] A. Engel. *Verification, Validation and Testing of Engineered Systems*. Wiley Series in Systems Engineering and Management. John Wiley & Sons, 2010.

[32] S. Engell, G. Frehse, and E. Schnieder. *Modelling, analysis, and design of hybrid systems*. Lecture notes in control and information sciences. Springer, 2002.

[33] Niklas En and Niklas Srensson. Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–26, 2006.

[34] R. Fletcher. *Practical Methods of Optimization*. J. Wiley, 1986.

[35] Harold Fox. *Agent problem solving by inductive and deductive program synthesis*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2008. AAI0821530.

[36] Yoav Freund and Robert E. Schapire. Large Margin Classification Using the Perceptron Algorithm. In *Machine Learning*, pages 277–296, 1998.

[37] M. Ganai, A. Gupta, and A. Gupta. *SAT-based scalable formal verification solutions*. Series on integrated circuits and systems. Springer Science+Business Media, 2007.

[38] Peter Y. K. Cheung George A. Constantinides and Wayne Luk. Wordlength optimization for linear digital signal processing. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol 22, No. 10*, pages 1432–1443, 2003.

[39] A. Girard and G. J. Pappas. Verification by simulation. In *HSCC*, volume 3927 of *LNCS*, pages 272–286, 2006.

[40] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23:5–48, 1991.

[41] Sally A. Goldman and Michael J. Kearns. On the complexity of teaching. *Journal of Computer and System Sciences*, 50:20–31, 1995.

[42] H. Gonzalez, R. Vasudevan, M. Kamgarpour, S. S. Sastry, R. Bajcsy, and C. J. Tomlin. A descent algorithm for the optimal control of constrained nonlinear switched dynamical systems. In *HSCC*, pages 51–60, 2010.

[43] Humberto González, Ramanarayan Vasudevan, Maryam Kamgarpour, Shankar Sastry, Ruzena Bajcsy, and Claire Tomlin. A numerical method for the optimal control of switched systems. In *CDC*, pages 7519–7526, 2010.

[44] P. M. Gruber and J. M. Wills, editors. *Handbook of Convex Geometry : Two-Volume Set*. North Holland, 1993.

[45] G.Shaavit and S.G. Brandt. The dynamic performance of a discharge air temperature system with a pi controller, technical report. Technical report, Honeywell Inc., commercial division., Arlington heights,IL, 1982.

[46] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Component based synthesis applied to bitvector circuits. Technical Report MSR-TR-2010-12, Microsoft Research, Feb 2010.

[47] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *PLDI*, pages 62–73, 2011.

[48] P. Gupta and A. Patra. Super-stable energy based switching control scheme for DC-DC buck converter circuits. In *ISCAS (4)*, pages 3063–3066, 2005.

[49] Joseph Y. Halpern. Presburger Arithmetic with Unary Predicates is Π^1_1 Complete. *The Journal of Symbolic Logic*, 56(2):637–642, 1991.

[50] Kyungtae Han, Iksu Eo, Kyungsu Kim, and Hanjin Cho. Numerical word-length optimization for cdma demodulator. In *Circuits and Systems, 2001. ISCAS 2001. The 2001 IEEE International Symposium on*, volume 4, pages 290 –293 vol. 4, may 2001.

[51] Kyungtae Han and B.L. Evans. Wordlength optimization with complexity-and-distortion measure and its application to broadband wireless demodulator design. In *Acoustics, Speech, and Signal Processing, 2004. Proceedings. (ICASSP '04). IEEE International Conference on*, volume 5, pages V – 37–40 vol.5, may 2004.

[52] T. Henzinger, P.-H. Ho, and H. Wong-Toi. Algorithmic analysis of nonlinear hybrid systems. *IEEE Trans. on Automatic Control*, 43:540–554, 1998.

[53] Thomas A. Henzinger and Howard Wong-Toi. Using HyTech to synthesize control parameters for a steam boiler. In *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control (Dagstuhl Seminar, June 1995).*, pages 265–282, London, UK, 1996. Springer-Verlag.

[54] Michael Heymann, George Meyer, and Stefan Resmerita. Analysis of zeno behaviors in hybrid systems. In *Proceedings of the 41st IEEE Conference on Decision and Control*, pages 2379–2384, 2002.

[55] Dorit Hochbaum. Complexity and algorithms for nonlinear optimizationproblems. *Annals of Operations Research*, 153:257–296, 2007.

[56] Dorit S. Hochbaum and J. George Shanthikumar. The complexity of nonlinear separable optimization. In *ICALP*, pages 461–472, 1989.

[57] Michael Huth and Mark Dermot Ryan. *Logic in computer science - modelling and reasoning about systems (2. ed.)*. Cambridge University Press, 2004.

[58] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, pages 215–224, 2010.

[59] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Synthesizing switching logic for safety and dwell-time requirements. In *ICCPS*, pages 22–31, 2010.

[60] Susmit Jha and Sanjit A. Seshia. Automated synthesis of fixed-point code. In *Under submission*, 2011.

[61] Susmit Jha, Sanjit A. Seshia, and Ashish Tiwari. Synthesizing switching logic to minimize long-run cost. In *EMSOFT*, pages to–appear, 2011.

[62] T. A. Johnson and R. Eigenmann. Context-sensitive domain-independent algorithm composition and selection. In *PLDI*, 2006.

[63] Rajeev Joshi, Greg Nelson, and Keith H. Randall. Denali: A goal-directed superoptimizer. In *PLDI*, 2002.

[64] M. Kamgarpour and C. Tomlin. On optimal control of non-autonomous switched systems under a fixed switching sequence. *Automatica*, 2011. To appear.

[65] J. Kapinski, B. H. Krogh, O. Maler, and O. Stursberg. On systematic simulation of open continuous systems. In *HSCC*, volume 2623 of *LNCS*. Springer, 2003.

[66] R. Karp. A characterization of the minimum cycle mean in a digraph. *Dis. Math.*, pages 309–311, 1978.

[67] H.K. Khalil. *Nonlinear systems*. Macmillan Pub. Co., 1992.

[68] Seehyun Kim, Ki-Il Kum, and Wonyong Sung. Fixed-point optimization utility for c and c++ based digital signal processing programs. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, 45(11):1455 –1464, nov 1998.

[69] Emanuel Kitzelmann and Ute Schmid. Inductive synthesis of functional programs: An explanation based generalization approach. *J. Machine Learning Res.*, 7:429–454, 2006.

[70] Donald E. Knuth. The art of computer programming. `http://www-cs-faculty.stanford.edu/~knuth/taocp.html`.

[71] T. Koo and S. Sastry. Mode switching synthesis for reachability specification. In *Proc. HSCC 2001*, LNCS 2034, pages 333–346, 2001.

[72] D. Kroening and O. Strichman. *Decision procedures: an algorithmic point of view*. Texts in theoretical computer science. Springer, 2008.

[73] A. Kucera and O. Strazovsky. On the controller synthesis for finite-state markov decision processes. *Fundamenta Informaticae*, 82(1-2):141–153, 2008.

[74] Jeffrey C. Lagarias, James A. Reeds, Margaret H. Wright, and Paul E. Wright. Convergence properties of the Nelder-Mead simplex method in low dimensions. *SIAM Journal of Optimization*, 9:112–147, 1998.

[75] Tessa Lau, Pedro Domingos, and Daniel S. Weld. Version space algebra and its application to programming by demonstration. In *ICML*, pages 527–534, 2000.

[76] Mike Tien-Chien Lee, V. Tiwari, S. Malik, and M. Fujita. Power analysis and minimization techniques for embedded DSP software. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 5(1):123 –135, march 1997.

[77] H. Lieberman, editor. *Your wish is my command: Giving users the power to instruct their software*. Morgan Kaufmann, 2001.

[78] Nathan Linial, Yishay Mansour, and Noam Nisan. Constant depth circuits, fourier transform, and learnability. In *FOCS*, pages 574–579, 1989.

[79] J.L. Lions. *Optimal Control of Systems Governed by Partial Differential Equations. (translated by S.K.Mitter)*. Springer-Verlag New York, Inc.

[80] M.R. Lowry and R.D. McCartney. *Automating software design*. AAAI Press, 1991.

[81] J. Lygeros. Lecture notes on hybrid systems. 2004.

[82] John Lygeros, Claire Tomlin, and Shankar Sastry. Multiobjective hybrid controller synthesis. In Oded Maler, editor, *Hybrid and Real-Time Systems*, volume 1201 of *Lecture Notes in Computer Science*, pages 109–123. Springer Berlin / Heidelberg, 1997.

[83] Yudong Ma and Mehdi Maasoumy. Optimal control of the operation of building cooling systems with vav boxes. Technical report, UC Berkeley, 2011.

[84] Mehdi Maasoumy Haghighi. Master's thesis, modeling and optimal control algorithm design for hvac systems in energy efficient buildings. Master's thesis, EECS Department, University of California, Berkeley, Feb 2011.

[85] E. Macii, M. Pedram, and F. Somenzi. High-level power modeling, estimation, and optimization. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(11):1061 –1079, nov 1998.

[86] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: Helping to navigate the API jungle. In *PLDI*, pages 48–61, 2005.

[87] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM TOPLAS*, 2(1):90–121, 1980.

[88] Zohar Manna and Amir Pnueli. A hierarchy of temporal properties. In *PODC*, pages 377–410, 1990.

[89] P. Manon and C. Valentin-Roubinet. Controller synthesis for hybrid systems with linear vector fields. In *Proc. IEEE Intl. Symp. on Intelligent Control*, pages 17–22, 1999.

[90] Henry Massalin. Superoptimizer - a look at the smallest program. In *ASPLOS*, pages 122–126, 1987.

[91] MathWorks. Minimum of unconstrained multivariable function using derivative-free method. `http://www.mathworks.com/help/techdoc/ref/fminsearch.html`.

[92] MathWorks. Solve initial value problems for ordinary differential equations. `http://www.mathworks.com/help/techdoc/ref/ode23.html`.

[93] D. Meister and T.P. Enderwick. *Human factors in system design, development, and testing*. Human factors and ergonomics. L. Erlbaum, 2002.

[94] T. M. Mitchell. *Machine learning*. McGraw Hill, New York, 1997.

[95] S. Mitra, D. Liberzon, and N. Lynch. Verifying average dwell time of hybrid systems. *ACM Trans. Embedded Comput. Syst.*, 8(1), 2008.

[96] C. Mitrohin, A. Podelski, and S. Wagner. Dwell time refinement, 2009. Personal communication.

[97] T. Moor and J. Raisch. Estimating reachable states of hybrid systems via l-complete approximations. In *SSCC'98*, 1998.

[98] T. Moor and J. Raisch. Discrete control of switched linear systems. In *ECC'99*, 1999.

[99] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *JOURNAL OF LOGIC PROGRAMMING*, 19(20):629–679, 1994.

[100] MyDoom Wikipedia Article. `http://en.wikipedia.org/wiki/Mydoom`, URL accessed Sep. 2009.

[101] J. A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, January 1965.

[102] Roland Olsson. Inductive functional programming using incremental program transformation. *Artif. Intell.*, 74:55–81, March 1995.

[103] G. J. Pappas, C. Tomlin, and S. Sastry. Conflict resolution for multi-agent hybrid systems. In *IEEE Control and Decision Conference*, pages 1184–1189, 1996.

[104] Vreda Pieterse, Derrick G. Kourie, Loek Cleophas, and Bruce W. Watson. Performance of c++ bit-vector implementations. In *SAICSIT '10*, pages 242–250, 2010.

[105] Phillip Porras, Hassen Saidi, and Vinod Yegneswaran. An analysis of Conficker's logic and rendezvous points. Technical report, SRI International, March 2009.

[106] Robin L. Raffard, Jianghai Hu, and Claire J. Tomlin. Adjoint-based optimal control of the expected exit time for stochastic hybrid systems. In *Hybrid Systems: Computation and Control, 8th Int. Workshop (HSCC 2005), volume 3414 of Lecture Notes in Computer Science*, pages 557–572. Springer Verlag, 2005.

[107] J. I. Rasmussen, K. G. Larsen, and K. Subramani. On using priced timed automata to achieve optimal scheduling. *FMSD*, 29:97–114, July 2006.

[108] S. Ray. *Scalable Techniques for Formal Verification*. Springer, 2010.

[109] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. 2010.

[110] S. S. Sastry. *Nonlinear Systems: Analysis, Stability, and Control*. Springer, 1999.

[111] J.M. Schumann. *Automated theorem proving in software engineering*. Springer, 2001.

[112] Sanjit A. Seshia. Sciduction: Combining induction, deduction, and structure for verification and synthesis. Technical Report UCB/EECS-2011-68, EECS Department, University of California, Berkeley, May 2011.

[113] M. Shaikh and P. Caines. On the optimal control of hybrid systems: Optimization of trajectories, switching times, and location schedules. In *HSCC*, 2003.

[114] Ehud Y. Shapiro. *Algorithmic Program DeBugging*. MIT Press, Cambridge, MA, USA, 1983.

[115] Changchun Shi and Robert W. Brodersen. An automated floating-point to fixed-point conversion methodology. In *Proc. IEEE Int. Conf. on Acoust., Speech, and Signal Processing*, pages 529–532, 2003.

[116] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodík, Vijay A. Saraswat, and Sanjit A. Seshia. Sketching stencils. In *PLDI*, pages 167–178, 2007.

[117] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.

[118] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.

[119] Eduardo D. Sontag. *Mathematical control theory: deterministic finite dimensional systems (2nd ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1998.

[120] SRI Intl. *Yices: An SMT solver*.

[121] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bidwidth analysis with application to silicon compilation. *SIGPLAN Not.*, 35:108–120, May 2000.

[122] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood. Deductive composition of astronomical software from subroutine libraries. In *CADE*, 1994.

[123] Phillip D. Summers. A methodology for lisp program construction from examples. *J. ACM*, 24(1), 1977.

[124] Wonyong Sung and Ki-Il Kum. Simulation-based word-length optimization method for fixed-point digital signal processing systems. *Signal Processing, IEEE Transactions on*, 43(12):3087 –3090, dec 1995.

[125] Symantec Corporation. Internet security threat report volume XIV. `http://www.symantec.com/business/theme.jsp?themeid=threatreport`, April 2009.

[126] Ankur Taly, Sumit Gulwani, and Ashish Tiwari. Synthesizing switching logic using constraint solving. In *VMCAI*, pages 305–319, 2009.

[127] Ankur Taly and Ashish Tiwari. Switching logic synthesis for reachability. In *EMSOFT*, pages 19–28, 2010.

[128] C. Tomlin, L. Lygeros, and S. Sastry. A game-theoretic approach to controller design for hybridsystems. *Proc. of the IEEE*, 88(7):949–970, 2000.

[129] Claire Tomlin. Field-controlled DC motor. In *Lecture Notes from Graduate Course in Non-linear Control Theory (EE222 Spring, 2011) at UC Berkeley*.

[130] Claire Tomlin, John Lygeros, and Shankar Sastry. Synthesizing controllers for nonlinear hybrid systems. In *HSCC*, pages 360–373, 1998.

[131] Leslie G. Valiant. A theory of the learnable. *Commun. ACM*, 27(11):1134–1142, 1984.

[132] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Longman, Boston, MA, USA, 2002.

[133] X. Xu and P. Antsaklis. Optimal control of switched systems via nonlinear optimization based on direct differentiation of value functions. *Intl. journal of control*, 75(16):1406–1426, 2002.

[134] Randy Yates. Fixed-point arithmetic: An introduction. In *Technical Reference Digital Signal Labs*, 2009.

[135] Ilsun You and Kangbin Yim. Malware obfuscation techniques: A brief survey. In *International Conference on Broadband, Wireless Computing, Communication and Applications (BWCCA)*, pages 297 –300, 2010.