TIMED REFINEMENT FOR VERIFICATION OF REAL-TIME OBJECT CODE PROGRAMS

A Dissertation
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Mohana Asha Latha Dubasi

In Partial Fulfillment of the Requirements
for the Degree of
DOCTOR OF PHILOSOPHY

Major Department:
Electrical and Computer Engineering

November 2018

Fargo, North Dakota

NORTH DAKOTA STATE UNIVERSITY

Graduate School

Title

TIMED REFINEMENT FOR VERIFICATION OF REAL-TIME OBJECT CODE PROGRAMS

By

Mohana Asha Latha Dubasi

The supervisory committee certifies that this dissertation complies with North Dakota State University's regulations and meets the accepted standards for the degree of

DOCTOR OF PHILOSOPHY

SUPERVISORY COMMITTEE:

Dr Sudarshan K. Srinivasan
Chair

Dr Scott C. Smith

Dr Dharmakeerthi Nawarathna

Dr Kenneth Magel

Approved:

November 1, 2018
Date

Dr Benjamin Braaten
Department Chair

ABSTRACT

Real-time systems such as medical devices, surgical robots, and microprocessors are safety-critical applications that have hard timing constraint. The correctness of real-time systems is important as the failure may result in severe consequences such as loss of money, time and human life. These real-time systems have software to control their behavior. Typically, this software has source code which is converted to object code and then executed in safety-critical embedded devices. Therefore, it is important to ensure that both source code and object code are error-free. When dealing with safety-critical systems, formal verification techniques have laid the foundation for ensuring software correctness.

Refinement based technique in formal verification can be used for the verification of real-time interrupt-driven object code. This dissertation presents an automated tool that verifies the functional and timing correctness of real-time interrupt-driven object code programs. The tool has been developed in three stages. In the first stage, a novel timed refinement procedure that checks for timing properties has been developed and applied on six case studies. The required model and an abstraction technique were generated manually. The results indicate that the proposed abstraction technique reduces the size of the implementation model by at least four orders of magnitude. In the second stage, the proposed abstraction technique has been automated. This technique has been applied to thirty different case studies. The results indicate that the automated abstraction technique can easily reduce the model size, which would in turn significantly reduce the verification time. In the final stage, two new automated algorithms are proposed which would check the functional properties through safety and liveness. These algorithms were applied to the same thirty case studies. The results indicate that the functional verification can be performed in less than a second for the reduced model.

The benefits of automating the verification process for real-time interrupt-driven object code include: 1) the overall size of the implementation model has reduced significantly; 2) the verification is within a reasonable time; 3) can be applied multiple times in the system development process.

ACKNOWLEDGEMENTS

Today, as I fulfill my dad's dream, I would like to remember and acknowledge all the people who have contributed to my success directly and indirectly. Each one of you has contributed to helping me become the person I am today.

Firstly, I would like to thank Dr. Sudarshan K. Srinivasan, Dr. Scott C. Smith, Dr. Dharmakeerthi Nawarathna and Dr. Kenneth Magel for their time, inputs, comments and for serving on my graduate committee. I would also like to thank Dr. Debasis Dawn for serving on my graduate committee while he was at North Dakota State University (NDSU).

I am would like to express my sincere gratitude to my Ph.D. advisor, Dr. Sudarshan K. Srinivasan, who had faith in me as a researcher and an instructor. He has always provided support and encouragement in my abilities. His guidance is always very valuable for me in terms of research, career, and teaching. His guidance has helped me grow as a researcher. I have learned so much from all the discussions that I have had with him both inside and outside the classroom.

I am grateful to the United States Agency for International Development (USAID) Government and the Electrical and Computer Engineering Department at North Dakota State University for the financial support that I have received as a graduate student. Getting funding for the most part of my Ph.D. journey has helped reduced the constant worry of supporting myself financially.

I would like to thank all the organizers (especially Dr. Natarajan Shankar, Dr. Aarti Gupta) and students whom I have met at various conferences and workshops, for the valuable discussions I have had and for their friendship. Attending conferences, workshops and summer schools has helped me in developing my research skills. Once again, I would like to thank my advisor for being supportive and always encouraging in learning outside the walls of the department. I am also thankful to the National Science Foundation (NSF) for providing the funds for traveling to various conferences, workshops and summer schools. Throughout my time as a student at NDSU, I have remained fair-minded due to the numerous discussions and arguments that I have had with my friends and colleagues. Hence I take this opportunity to thank all of them, who have been with me throughout this journey. I would also like to thank the staff of my department (Jeff, Pris, Laura,

Eric and Anne) for helping me with all the departmental procedures that would otherwise have been very difficult to accomplish.

A special thanks to my husband who has been through the various ups and downs of our life together during my Ph.D. journey. He has indeed taught me a different aspect of seeing life. He has been through all my tantrums and still supported me in achieving this goal. He has been a brave outsider to proofread this dissertation and give valuable suggestions on how it can be presented in a better way for the non-technical audience. Thank you for coming into my life.

Apart from my academic studies, I have also started my spiritual journey during this time. This has happened because of my husband and his favorite book "Bhagavad-Gita". Even though I resented it initially, my teacher Shymala Aunty has helped me to see the goodness in the teachings of Lord Krishna to Arjuna. This new journey, even though had a bitter starting, had helped me to see the goodness even in the darkest moments of life. I will always be truly grateful to the teachings of Bhagavad-Gita.

Finally, I would like to thank my mom (Sai Leela Dubasi), my dad (Govardhana Rao Dubasi), my husband (Vinay Gonela), my sister (Priyanka Dubasi) and everyone in my family who have supported me financially and emotionally throughout this challenging time. Your motivational words have helped me to survive this big journey of Ph.D. Thank you each and everyone for your encouraging words whenever I needed them the most. Without all of your support, I would not have come this far and achieved all this.

"PhD made me poorer, without money, but richer in thoughts.
The pursuit of PhD is enduring daring adventure."

– Lailah Gifty Akita

DEDICATION

To GOD,

(for giving me the strength to complete this phase of life)

&

To my dad and my advisor,

(for believing in my abilities when no one did)

&

To my husband

(for coming into my life)

TABLE OF CONTENTS

ABSTRACT iii

ACKNOWLEDGEMENTS iv

DEDICATION vi

LIST OF TABLES xi

LIST OF FIGURES xii

LIST OF ABBREVIATIONS xiv

1. INTRODUCTION 1

2. BACKGROUND 8

2.1. Set Theory 8

2.2. Formal Verification 8

2.3. Stepper Motor Control 11

2.3.1. Specification of Stepper Motor Control 12

2.3.2. Implementation of Stepper Motor Control 13

2.4. Theory of Refinement 14

2.4.1. WFS Refinement 16

2.4.2. WEB Refinement 18

3. CASE STUDIES 20

3.1. Stepper Motor Control Program 20

3.1.1. Case Study A - Interrupt Driven Full Stepping Stepper Motor Control in Clockwise Direction 21

3.1.2. Case Study B - Interrupt Driven Full Stepping Stepper Motor Control in Anti-Clockwise Direction 22

3.1.3. Case Study C - Interrupt Driven Double Stepping Stepper Motor Control in Clockwise Direction 23

3.1.4. Case Study D - Interrupt Driven Double Stepping Stepper Motor Control in Anti-Clockwise Direction 23

3.1.5. Case Study E - Interrupt Driven Half Stepping Stepper Motor Control in Clockwise Direction . . . 24

3.1.6. Case Study F - Interrupt Driven Half Stepping Stepper Motor Control in Anti-Clockwise Direction . . . 24

3.1.7. Case Study G - Full Stepping Stepper Motor Control without Interrupts in clockwise direction . . . 25

3.1.8. Case Study H - Full Stepping Stepper Motor Control without Interrupts in Anti-Clockwise Direction . . . 25

3.1.9. Case Study I - Double Stepping Stepper Motor Control without Interrupts in Clockwise Direction . . . 26

3.1.10. Case Study J - Double Stepping Stepper Motor Control without Interrupts in Anti-Clockwise Direction . . . 26

3.1.11. Case Study K - Half Stepping Stepper Motor Control without Interrupts in Clockwise Direction . . . 26

3.1.12. Case Study L - Interrupt Driven Half Stepping Stepper Motor Control in Anti-Clockwise Direction . . . 27

3.1.13. Case Study M - Interrupt Driven Variable Speed Full Stepping Stepper Motor Control in Clockwise direction . . . 27

3.1.14. Case Study N - Interrupt Driven Variable Half Full Stepping Stepper Motor Control in Clockwise directon . . . 28

3.2. Infusion Pump Control Program . . . 28

3.2.1. Pulse Width Modulation (PWM) . . . 28

4. TIMED REFINEMENT . . . 30

4.1. Specification TTS of Stepper Motor Control . . . 30

4.2. Implementation TTS of Stepper Motor Control . . . 31

4.3. Timed Refinement . . . 32

4.3.1. Timed Well-Founded Simulation Refinement . . . 37

4.3.2. Timed Well-Founded Equivalence Bisimulation Refinement . . . 39

4.4. Checking Timed Refinement . . . 42

4.5. Case Studies and Results . . . 46

5. STUTTERING ABSTRACTION . . . 51

5.1. Stuttering Abstraction for TS . . . 51
5.2. Timed Stuttering Abstraction (TSA) for TTS . . . 53
5.2.1. Timed Stuttering Abstraction (TSA) . . . 53
5.2.2. Correctness of TSA . . . 55
5.2.3. Procedure for Dynamic TSA . . . 56
5.3. Case Studies and Results . . . 59
6. WFS REFINEMENT CHECKER . . . 63
6.1. Automated WFS Refinement for Object-Code . . . 63
6.1.1. Procedure for Checking WFS Refinement for Object-Code . . . 64
6.1.2. Time Complexity for Refinement Checker . . . 66
6.2. Case Studies and Results . . . 66
7. LIVENESS . . . 70
7.1. Liveness Detection for Object-Code . . . 70
7.1.1. Stuttering Abstraction for TS . . . 72
7.1.2. Liveness Detection Procedure . . . 74
7.1.3. Time Complexity for Liveness Detection . . . 76
7.2. Case Studies and Results . . . 77
8. RELATED WORK . . . 79
8.1. Refinement Based Verification for Real-Time Systems . . . 79
8.2. Timed Stuttering Abstraction . . . 81
8.3. Refinement Checker and Liveness Detection . . . 83
9. CONCLUSIONS . . . 84
9.1. Future Works . . . 86
REFERENCES . . . 88
APPENDIX. A: LIST OF PUBLICATIONS . . . 97
APPENDIX. B: SOURCE CODE EXAMPLE . . . 99

APPENDIX. C: OBJECT CODE EXAMPLE 101

LIST OF TABLES

Table Page

1.1. Application Areas for Safety-Critical Devices (source: google) 2

4.1. Verification Statistics for TWEB 49

5.1. Verification Statistics for TSA 61

5.2. Verification Statistics for TSA for Infusion Pump Controller (IPC) Case Study 62

6.1. Verification Statistics for WFS Refinement Checker 67

LIST OF FIGURES

Figure Page

1.1. This Research Area Applications 5

2.1. Pictorial Representation of the Traffic Light Transition System 10

2.2. Different Types of Transition Systems 11

2.3. Stepper Motor Control Specification TS with Termination State 12

2.4. Stepper Motor Control Specification TS 13

2.5. Source Code to Object Code 14

2.6. Stepper Motor Control Implementation TS 15

3.1. Stepper Motor Control Specification TS for Interrupt-Driven Full Stepping in Clockwise Direction 22

3.2. Stepper Motor Control Specification TS for Interrupt-Driven Full Stepping in Anti-Clockwise Direction 22

3.3. Stepper Motor Control Specification TS for Interrupt-Driven Double Stepping in Clockwise Direction 23

3.4. Stepper Motor Control Specification TS for Interrupt-Driven Double Stepping in Anti-Clockwise Direction 24

3.5. Stepper Motor Control Specification TS for Interrupt-Driven Half Stepping in Clockwise Direction 24

3.6. Stepper Motor Control Specification TS for Interrupt-Driven Half Stepping in Anti-Clockwise Direction 25

3.7. Pulse Width Modulation Signal for Infusion Pump Control 29

3.8. PWM Specification TS 29

4.1. Stepper Motor Control Specification TTS 32

4.2. Stepper Motor Control Implementation TTS 33

4.3. Example Comparing Implementation and Specification Transitions. 34

4.4. Stuttering Segment 36

5.1. Basic Block in Abstraction 52

5.2. Basic Block in Abstraction for a TTS in Figure 4.2 54

5.3. Abstraction on One Stuttering Transition 55

7.1. Buggy Stepper Motor Control Implementation TS 71

7.2. An Example Snippet of Implementation TS from Fig. 7.1 that Contains Stuttering Cycle 73

7.3. An Example Snippet of Implementation TS From Fig. 7.1 that Contains Stuttering Cycle which Cannot be Reduced to Self-Loop 74

7.4. An Example Snippet of Implementation TS from Fig. 7.1 that Contains Multiple Paths 75

LIST OF ABBREVIATIONS

MM^a	Abstracted model
M^a	Abstracted timed transition system
$M_I^{a^n}$	Abstracted implementation timed transition system obtained by applying timed stuttering abstraction on M_I n times
AP	Atomic proposition
$BCET$	Best case execution time
R	Binary relation
T	Binary relation
B	Bisimulation relation
BAT	Bit-level analysis tool
w	Current state of the implementation transition
s	Current state of the specification transition
CSP	Communicating sequential processes
$r; q$	Composition of relations r and q
R_{CE}	Counterexample set
Φ	Degree of rotation
d	Delay
DC	Direct current
DAG	Directed acyclic graph
t_d	Duty cycle
E	Edges in a graph
ECG	Electrocardiography
EEG	Electroencephalography
EKG	Electrocardiography
B	Equivalence relation

FDR	Failures-divergences refinement
$FPGA$	Field programmable gate array
match	Flag used in checking WFS refinement
path-complete	Flag to control symbolic simulation and dynamic TSA
termination-condition	Flag to exit from repeat loop in checking timed refinement
sseg-complete	Flag to keep track of the computation of the stuttering segment
flag-SCC	Flag to identify SCC
flag-self-loop	Flag to identify self-loops
skip-simulation	Flag to skip finding the successor of w
FDA	Food and Drug Administration
FV	Formal verification
M_I	Implementation model
s_0	Initial state in a set
w_0	Initial state of object code
IPC	Infusion pump control
ISR	Interrupt service routine
L	Labeling function
length-of-M_I	Length of the implementation model
sseg-length	Length of the stuttering segment
I	List of states that cannot be abstracted
R_C	List of successor states
R_U	List of successor states that are not visited
sseg-list	List of transitions

i	Loop variable
j	Loop variable
lb	Lower bound on the time delay of the transition
lb^n	Lower delay bound of transition n
lb_i	Lower delay bound for the transition in M_I
lb_s	Lower delay bound for the transition in M_S
$lb_s^{\langle a,b \rangle}$	Lower delay bound for the transition $\langle a, b \rangle$ of M_S
$M_a \approx_r M_b$	M_a is a WEB refinement of M_b w.r.t refinement map r
$M_a \sqsubseteq_r M_b$	M_a is a WFS refinement of M_b w.r.t refinement map r
$M_a \trianglerighteq_r M_b$	M_a is a TWEB refinement of M_b w.r.t refinement map r
$M_a \simeq_r M_b$	M_a is a TWFS refinement of M_b w.r.t refinement map r
MM_I	Marked implementation timed transition system
$MM_I^{c \leftarrow b}$	Marked M_I of M_c w.r.t M_b
MM_S	Marked specification timed transition system
n_t	Non-stuttering transition
σ	Path in a transition system
POR	Partial order reduction
PWM	Pulse width modulation
r	Refinement map
$r()$	Refinement map function
RIT	Repetitive Interrupt Timer in LPC1768
r	Rotational speed
SMT	Satisfiability modulo theories
sseg-set	Set of all stuttering segments
S	Set of states in a transition system
S_S	Set of states of the specification model

R	Simulation relation
M_S	Specification model
w_{abs}	Start state of an abstraction
$S_{IStuttering}$	States in $M_{IStuttering}$
SA	Stuttering abstraction
ϕ	Stuttering segment
STS	Stuttering simulation
s_t	Stuttering transition
SCC	Strongly connected component
$ss\text{-}length$	Stuttering segment length
v	Successor state of the implementation transition
u	Successor state of the specification transition
TTS	Timed transition system
$TWEB$	Timed Well-Founded Equivalence Bisimulation
$TWFS$	Timed Well-Founded Simulation
TSA	Timed stuttering abstraction
t_p	Time period of a pulse
$R_{IStuttering}$	Transitions in $M_{IStuttering}$
R_I	Transitions of abstracted implementation model
R_S	Transitions of specification model
T	Transition relation
TS	Transition system
$M_{IStuttering}$	Transition system with only stuttering transitions
m	Type of transition
ub	Upper bound on the time delay of the transition
ub^n	Upper delay bound of transition n

ub_i	Upper delay bound for the transition in M_I
ub_s	Upper delay bound for the transition in M_S
$ub_s^{\langle a,b \rangle}$	Upper delay bound for the transition $\langle a, b \rangle$ of M_S
V	Vertices in a graph
WEB	Well-Founded Equivalence Bisimulation
WFS	Well-Founded Simulation
$WCET$	Worst case execution time

1. INTRODUCTION

Safety-critical embedded devices are used in a number of application areas that include, but not limited to medical devices, surgical robots, avionic control systems. Table 1.1 presents exhaustive areas of application of safety-critical systems. Typically, a software is used as a control program in these applications. The execution of the software needs to be correct. Otherwise, it would result in severe consequences such as loss of time, money [5, 49] and human safety. Ensuring the correctness of the control software is still an on-going challenge that warrants attention.

For example, during the time period 2001-2017, the Food and Drug Administration (FDA) has issued 54 Class-I recalls on infusion pumps (medical devices used to deliver controlled doses of fluid medications to patients intravenously) due to software issues [64, 65]. Class-I recalls are applied when the use of a medical device is determined to cause adverse health consequences or even death. Numerous other medical devices also have similar problems [55, 62]. Recalls and problems due to software errors are not limited to the medical industry [7, 12, 14] alone. For example, in the early 2000s, Toyota Camry's electronic throttle system experienced some unintended acceleration problems [17, 57, 63]. This even resulted in the loss of a human life. Initially, it was assumed that the Toyota Camry's electronic throttle system was faulty due to software errors. But, later on, the problem was identified to be different. However, this incident infused fear not to trust the software control programs in the automobile industry. Over the years, the use of software codes in automobiles has increased to a greater extent. Nowadays, automobiles not only consist of mechanical components but also software components. The use of these software has grown significantly in recent years resulting in several risks such as the outsider's ability to control the automobile by hacking into its control systems [54]. New developments in the automobile industry that has gained popularity in recent years such as self-driving cars are threatened by similar problems. The reliability of these control software programs in the self-driven cars is critical not only to the safety of the person in the car but also others on the road. In the past, numerous testing-based methods [56] have been developed to test the reliability of this software. However, current industry's standard process of testing-based methods is inadequate [29, 33, 50].

Table 1.1. Application Areas for Safety-Critical Devices (source: google)

Area	Examples
Aviation	aircrew life support systems
	air traffic control system
	avionics
	drones
	engine control systems
	flight planning to determine fuel requirements for a flight
	radio navigation system
	rockets
Medical	defibrillator machines
	dialysis machines
	electocardiography (ECG or EKG)
	electroencephalography (EEG)
	heart-lung machines
	infusion pumps
	insulin pumps
	mechanical ventilation systems
	medical imaging devices
	radiation therapy machines
	surgical robots
Military	defense facilities
	weapons
Railway	automatic train stop
	platform detection to control train doors

Continued on next page

Table 1.1 – Application Areas for Safety-Critical Devices (source: google) (continued)

Area	Examples
	railway control systems
	railway signaling systems
	advanced driver-assistance systems
	airbag systems
	automated driver assistance system
	battery management system
	braking systems
Automotive	drive by wire systems
	electric park brake
	electronic throttle control
	park by wire
	power steering systems
	seat belts
	shift by wire systems
	crew rescue systems
	crew transfer systems
Spaceflight	human spaceflight vehicles
	launch vehicle safety
	rocket range launch safety systems
	burner control systems
	circuit beaker
	emergency services dispatch systems
	electricity generation, transmission and distribution
Infrastructure	fire alarm

Continued on next page

Table 1.1 – Application Areas for Safety-Critical Devices (source: google) (continued)

Area	Examples
	fire sprinkler
	fuse (electrical)
	fuse (hydraulic)
	telecommunications
Nuclear engineering	nuclear reactor control system
Recreation	amusement rides
	climbing equipment
	parachutes
	SCUBA equipment
Others	communication protocols
	microprocessors

Formal verification (FV) has become the foundation for ensuring software correctness when dealing with safety-critical systems. These methods have been well-established to provide safety-assurances. The use of formal verification has become an industry-standard when addressing software correctness of safety-critical devices. There are many success stories and commercial adoption of the formal verification process. Formal verification process has been widely adopted by industries such as Intel [25, 39, 53], ARM [61], Microsoft [4, 8], and Airbus [24].

Safety-critical embedded devices are programmed in high-level languages like C, Java, Python, and many more, known as source code. When the program is executed on these embedded devices, it is converted into object code, whose verification is imperative. Verification of device-object code (code or the set of instructions that is executed by microcontrollers embedded in the device) is a challenging task as the object code is of very low-level, real-time, and interrupt driven. Interrupt-driven real-time object code programs can often have behaviors that are

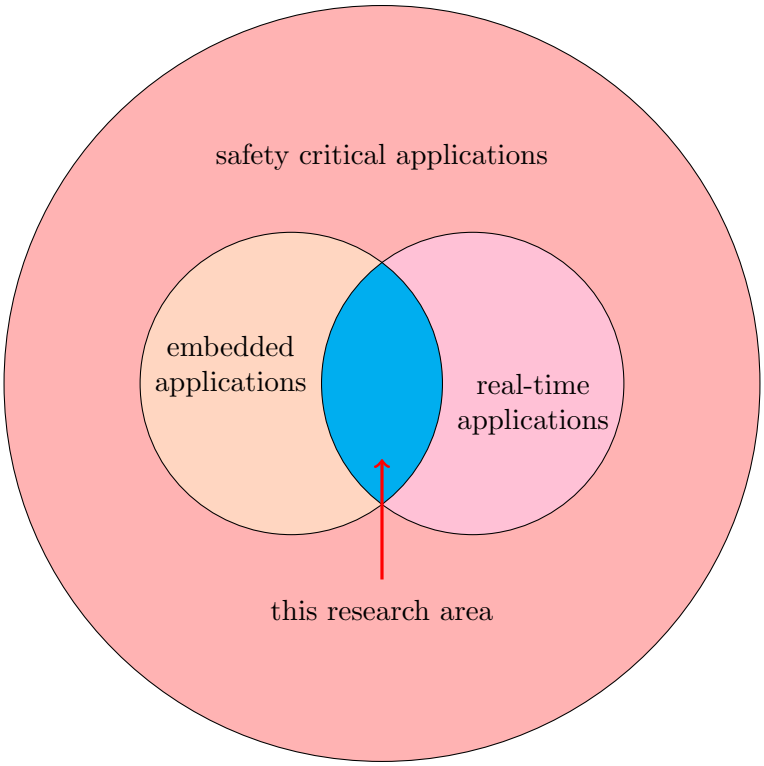


Figure 1.1. This Research Area Applications

very hard to emulate, capture, and analyze using only testing-based methods, that has been the primary method for verification in the industries for a long time. The inadequacy of industrial processes to address verification is evidenced by examples of buggy safety-critical embedded devices [33,43,50,68]. Even though several different problems exist, a critical gap in the field of formal verification is the lack of availability of efficient and scalable formal verification methods for the real-time device-object code. Figure 1.1 presents the research area tackled in this dissertation.

Formal verification consists of refinement-based verification [45] which has been demonstrated to be applicable to the verification of embedded control software at the object code level. Refinement-based verification consists of two models: 1) implementation model, and 2) specification model. Typically, the design artifact to be verified is called the implementation model. A specification model is a formal model that captures the correct functionality for the implementation. The goal of refinement-based verification is to mathematically check that the implementation model behaves correctly as defined by the specification model. Here, both the implementation model and specification model are represented as transition systems (TSs). Transition system is a

mathematical modeling framework for code that is based on states of the program and transitions between these states.

Refinement-based verification consists of Well-Founded Simulation (WFS) refinement and Well-Founded Equivalence Bisimulation (WEB) refinement [45]. WFS and WEB refinement explains how equivalence between two transition systems can be established and is applicable for functional verification. However, both of them do not consider timing requirements and properties, which are important elements in real-time interrupt-driven object code programs. Consequently, in this dissertation, the notion of refinement for the verification of real-time object code programs, called timed refinement, is first introduced. The concept of timed refinement is added to WFS and WEB refinements, and therefore called as Timed Well-Founded Simulation (TWFS) refinement and Timed Well-Founded Equivalence Bisimulation (TWEB) refinement. Timed refinement is a notion of equivalence between two timed transition systems (TTSs) that allows for stuttering between the implementation and specification, and also allows for the use of refinement maps. The incorporation of stuttering and refinement maps allows for timed refinement to be applicable to the verification of low-level interrupt-driven real-time object code against high-level specification models. In the definition of timed refinement, stuttering is the phenomenon where multiple but finite transitions of the implementation can match a single transition of the specification. Refinement maps allow low-level implementations to be verified against high-level specification models. Therefore, timed refinement verification for real-time interrupt-driven object code programs has been developed as an automated tool in three stages.

Firstly, an algorithm (it may be noted that in this dissertation, the term algorithm is interchangeably used with procedure) to verify timed refinement has been developed. It has been formalized and its correctness proof has been developed. The models were developed and WEB refinement has been established manually. Once the functional verification has been performed, timed refinement verification has been performed using the automated tool. In order to illustrate the effectiveness of the proposed technique, a set of six case studies has been used. The results indicate that the abstractions based on stuttering reduce the size of the implementation TTS by at least 4 orders of magnitude (Section 4.5).

Secondly, based on the results obtained in timed refinement, a novel abstraction technique that exploits the phenomenon of stuttering called timed stuttering abstraction (TSA) is intro-

duced. Timed stuttering abstraction is applied to timed transition systems. Specific additional contributions include: (1) a formalization of TSA; (2) correctness of TSA in the context of TWFS refinement; (3) dynamic timed stuttering abstraction, an algorithm to automatically apply TSA during symbolic simulation of the object code. A comprehensive set of thirty case studies is used to demonstrate the effectiveness of TSA. In this study, it has been shown that TSA provides a reduction in the complexity of real-time object-code verification by several-orders-of-magnitude. This, in turn, reduces the verification time of timed refinement.

In the previous stages of the studies, timed refinement was automated. However, WFS refinement verification was manually done, which is inadequate for large-scale programs. In addition, the study of literature indicates none of the up-to-date studies have automated WFS refinement verification for real-time interrupt-driven object code programs. In order to make a fully functional automated tool requires both WFS refinement verification and timed refinement verification to be automated. Therefore, in the final stage of this study, two algorithms for automatic WFS refinement checking optimized for object code verification are proposed. The purpose of the two algorithms is to check the following properties for timed transition systems: 1) safety and 2) liveness. Safety informally means that if the implementation makes progress, the result of that progress satisfies the specification requirements. Liveness verification checks for deadlock errors in the object code. The algorithms have been implemented and the automated tool flow has been applied to several object code control programs to demonstrate the effectiveness of the approach.

In summary, the unique contributions of this dissertation are as follows:

1. The notion of correctness called timed refinement for real-time interrupt-driven object code programs has been introduced for the first time.

2. A novel abstraction technique called stuttering abstraction for transition systems and timed stuttering abstraction (TSA) for timed transition systems.

3. New automated algorithms that check for safety and liveness properties for real-time interrupt-driven object code programs are developed.

2. BACKGROUND

This chapter presents information that could serve as the background to understand the notion of correctness presented later. Also, the various notations that are used throughout the dissertation are discussed. Most of these notations are pretty standard.

2.1. Set Theory

A *set* is defined as a collection of elements. An ordered pair is denoted as $\langle i, j \rangle$ whose first component is i and whose second component is j. Function application is denoted by an infix dot "." and is left associative. Sometimes, a function is represented using curried brackets *e.g.*, $f.x.y$ is written as $f(x, y)$.

For any binary relation R, sRw represents $\langle s, w \rangle \in R$ [45]. The *composition* of binary relations R and T is denoted by $R;T$ or $T \circ R$. A binary relation, $R \subseteq XxX$, is *reflective* if $\langle \forall x \in X :: xRx \rangle$. R is *symmetric* if $\langle \forall x, y \in X :: xRy \implies yRx \rangle$. R is *transitive* if $\langle \forall x, y, z \in X :: xRy \wedge yRz \implies xRz \rangle$. A binary relation is a *preorder* if it is reflective and transitive. A binary relation is an *equivalence relation* if it is reflective, transitive and symmetric.

A function from [0..n) for some natural number n is called a *finite sequence*. A *well-founded structure* is a pair $\langle W, \lessdot \rangle$ where W is a set and $\lessdot$ is a binary relation on W such that there are no infinitely decreasing sequences on W, with respect to $\lessdot$. $<$ is used to compare natural numbers and $\prec$ is used to compare ordinal numbers. *Ordinal* is a sequence in which something is in relation to others of its kind. An *ordinal number* tells the position of an item in an ordered sequence.

Atomic propositions (APs) are statements that are used to represent the *state* information, *i.e.*, information about the current state in the system. Atomic propositions can be evaluated to be either true or false. An invariant is a property that specifies which atomic propositions must hold at all times for an application.

2.2. Formal Verification

The traditional methods for checking the correctness were testing and simulation [15]. While simulation and testing explored some of the possible behaviors and scenarios of the system, it often left an open question, can there be fatal bugs in the unexplored cases? Simulation and testing are not sufficient for systems with a large number of possible states and can easily miss

errors. Formal verification conducts an exhaustive exploration of all possible behaviors and hence has become the foundation for ensuring software correctness. A system can be declared correct after applying formal verification method, this means that all behaviors have been explored and the question of adequate coverage or a missed behavior becomes irrelevant. Formal verification typically consists of checking the implementation (the artifact to be verified) against a specification. The specification is a high-level mathematical model that describes the correct behavior of the system.

Model checking and theorem proving are two leading verification paradigms in formal verification. In model checking, a desired behavioral property of a reactive system is verified over a given system (specification) through exhaustive enumeration of all the states reachable by the system and the behaviors that traverse through them [15]. A reactive system is described as non-terminating computing system that maintains an ongoing interaction with its environment. Model checking is a powerful method used extensively for hardware and software verification. It has been used widely for checking applications in computer science. In model checking, the specification is given as a formula in temporal logic, which can assert how the behavior of the system evolves over time. The major disadvantage of model checking is the state explosion that can occur if the system being verified has many components that can make transitions in parallel [15]. Theorem proving is a formal logic that uses states and proves theorems. The proofs of theorem proving are checked by computer programs. Theorem provers are based on set theory, higher-order logic, constructive type theory, first-order logic, etc. Theorem provers consist of refinement based verification.

Refinement based verification has been demonstrated [45] to be applied to the verification of embedded control software at the object code level. The goal of refinement-based verification is to mathematically prove that the implementation behaves correctly as defined by the specification. In refinement-based verification, both the specification and implementation are modeled as transition system (TS). Transition systems are used to describe the configuration of an application as states and transitions saying how and when to go from one state to another. For example, in a game of chess, the configurations are the positions in the game (*i.e.*, the placements of the chess pieces on the board), and transitions describe the movement of the chess pieces according to the rules of the game). A transition system may be used to describe a computer itself, networks, communication systems, algorithms, etc. When a system is described as a transition system, the system can be subjected to formal analysis, *i.e.*, it allows to talk about the system properties in a precise manner.

The formal definition of transition system is given below:

Definition 1. *[45] A transition system (TS) $M = \langle S, T, L\rangle$ is a three tuple in which S denotes the set of states, $T \subseteq SXS$ is left-total and is the transition relation that provides the transition between states, and L is a labeling function that describes what is visible at each state.*

S represents the states in the transition system which is also the state space of the system. Often, transition systems are drawn as directed graphs with states represented by vertices and transitions represented by directed edges. For example, consider the transition system shown in Figure 2.1. This is a simplified model of a traffic light, in which the light can be red or green. S = {R, G}, T = $\{\langle R, G\rangle, \langle G, R\rangle\}$ and the label $L(R)$ indicates the red light is ON and green light is OFF, and $L(G)$ indicates the green light is ON and the red light is OFF. Here it is also shown that R is the initial state for the traffic light system. Initial state (s_0) is a state in the set of states ($s_0 \in S$) indicates the state from which the application starts. Some applications may have multiple start states from which an application may start. For example, consider a football match as an example. In a football match, if each player is represented as a state then they would be a minimum of two states that are assigned as an initial state since a football game consists of a player from each team who could start the game. Depending on who starts the game the transition relation would be different. Similarly, depending on the safety-critical application, they could be multiple initial states and the corresponding transition relation could be different.

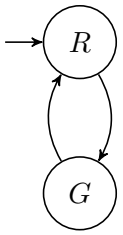
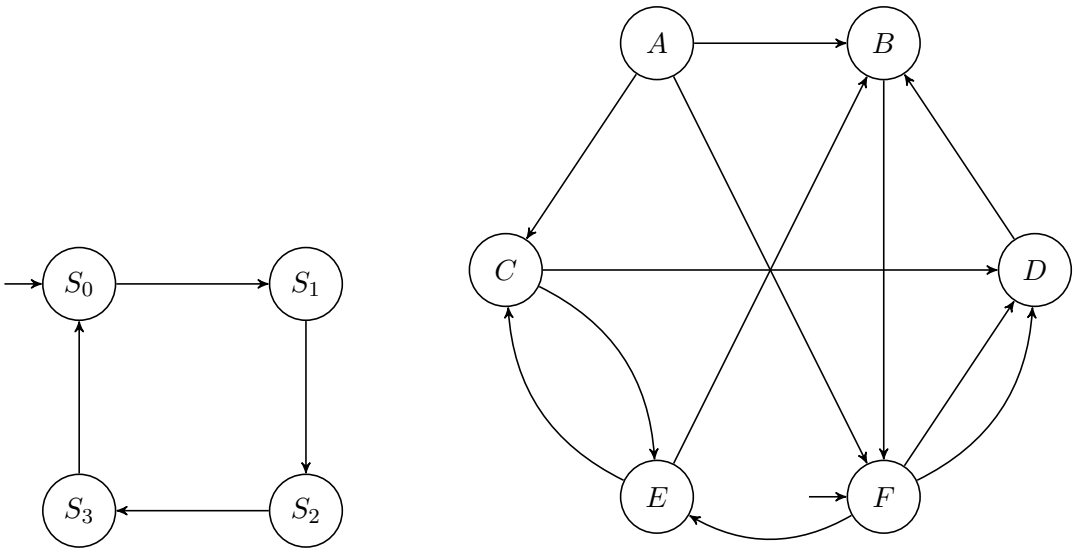


Figure 2.1. Pictorial Representation of the Traffic Light Transition System

Some of the properties of interest for a transition system are *safety properties* and *liveness properties*. Safety properties capture the intuitive notion that "nothing bad can happen in a process", *i.e.*, it is impossible to enter an unwanted configuration. Liveness properties capture the intuitive notion that "eventually something happens in a process", the safest way of avoiding errors

is to do nothing, but systems of that kind are rather boring. Another property of interest for any system is *termination*. Termination properties ensure that the system stops eventually at some point. Termination is an example of the liveness property and cannot be expressed in terms of invariant.



(a) Deterministic TS (b) Non-Deterministic TS

Figure 2.2. Different Types of Transition Systems

Transition systems can be classified as deterministic systems and non-deterministic systems (Figure 2.2). A transition system in which each state has only one successor state is known as *deterministic transition system*. In Figure 2.2a, each state in the transition system has one and only one successor. A transition system in which one or more states contain more than one successor state is known as *non-deterministic transition system*. For example in Figure 2.2b, states A, C, E, F contain multiple successor. Depending on the other parameters that describe the system, the successor state for each of the state can be determined.

In a transition system , a *path* σ is a sequence of states such that for adjacent states s and u, sTu where T is the transition relation.

2.3. Stepper Motor Control

Stepper motor control is used to describe the various algorithms presented in this dissertation for the verification of real-time interrupt-driven object code programs. A stepper motor is

a brushless DC electric motor. Stepper motors are widely used in commercial applications such as medical devices like infusion pump, computer peripherals, robotics like surgical robots, machine tools, and process control [16, 41], and many of these applications are safety-critical. The current pulse applied to the motor generates discrete rotation of the motor shaft. A stepper motor can have 4 or 6 leads. Consider a motor with 4 leads say a, b, c, and d. Then the following repeating sequence of values to the leads causes the motor to spin: $abcd$ = 1000, 0100, 0010, 0001, 1000 etc. There are other such repeating sequences that can cause a 4-lead or 6-lead stepper motor to spin. Every next value in the sequence causes the motor to rotate by a small angle. Thus a stepper motor can be controlled by software (that generates the above sequence of values to the leads), executing on a micro-controller that is interfaced with the motor. The time delay between when each value in the sequence is generated determines the speed of the motor. The speed also depends on the angle the motor rotates at each step.

2.3.1. Specification of Stepper Motor Control

Figure 2.3 shows only one type of specification transition system (TS) for stepper motor control with 4 leads. The states are represented as $S_0, S_1, S_2, S_3, S_4, S_5$. Here, S_0 is the initial state and S_5 is the stop state. The stepper motor can reach the stop/terminating state from any of the states once it has been turned on.

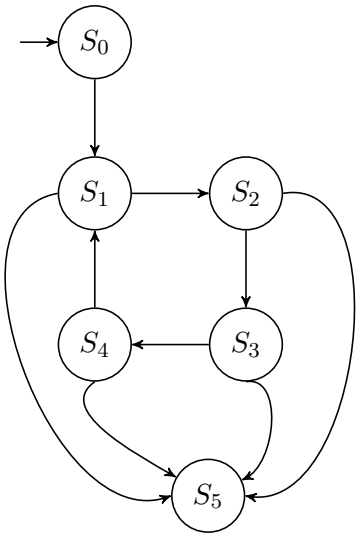


Figure 2.3. Stepper Motor Control Specification TS with Termination State

Throughout this dissertation, Figure 2.4 is considered as the specification transition system (TS) (M_S) for stepper motor control with 4-leads. The termination state is not considered for the rest of this dissertation since it is assumed the motor can stop/terminate from any state once it has been turned on. The states are represented as S_0, S_1, S_2, S_3, S_4. The transition relation determines the direction of the shaft. The labeling function gives the values of the leads, which determine the state. The transition system has 4 states and captures the repeating sequence of values the software controller must generate. The functionality of the controller is not fully described unless the speed of rotation is specified.

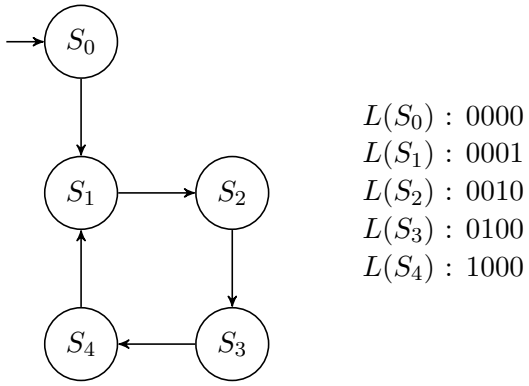


Figure 2.4. Stepper Motor Control Specification TS

2.3.2. Implementation of Stepper Motor Control

The target is to verify the implementation model for the stepper motor control which is the object code. The implementation model is obtained by generating a function for each instruction that describes the effect of the instruction on the state of the microcontroller. The state of the microcontroller is not as simple as the specification states (as shown in Figure 2.4) but consists of registers, flags, and memory of the microcontroller. The set of all such functions (one for each instruction) along with the initial state of the microcontroller defines the transition system model of the implementation. Note that this set includes the instructions in interrupt service routines of the interrupts that the program uses. The implementation model consists of millions of transitions because of the various possible values that the registers, flags, memory and special registers of the microcontroller can have during the execution of the object code program.

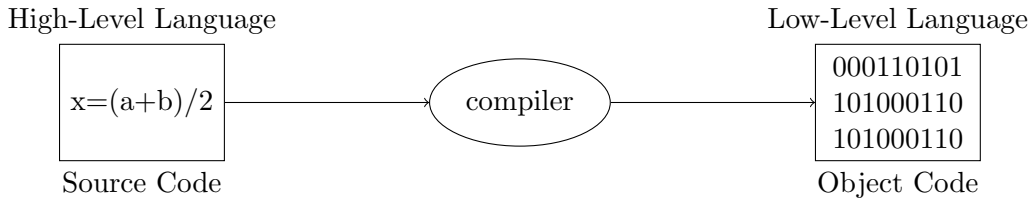


Figure 2.5. Source Code to Object Code

The choice of representing the implementation model is the object code program instead of the corresponding source code. Object code verification will find bugs that are introduced during compilation. Also, for interrupt driven programs, the implementation behavior is more accurately captured at the object level. For example, consider a C statement that is compiled into three instructions in the object code (Figure 2.5). An interrupt can occur after the execution of the first instruction and before the execution of the second instruction. This situation cannot be captured in a C model, as the instructions are bundled together in one C statement.

Figure 2.6 shows an example TS that is an implementation of the stepper motor specification TS shown in Figure 2.4. Looking at Figures 2.4 and 2.6, it can be seen that the specification TS and implementation TS look quite different.

2.4. Theory of Refinement

Theory of refinement is a task of modifying a computer program and simplifying it so that formal verification could be applied to it. Consider two systems S_1 and S_2 which are at different levels of hierarchy. In order to prove that S_1 and S_2 represent the same application, we need to define a relation between them. Simulation relations can be used to model the uniformity between two systems that are at different levels of hierarchy.

R is a *simulation relation* [45] on a transition system $M = \langle S, R, L \rangle$ if $R \subseteq S$ x S and for s, w such that sRw then

1. $L.s = L.w$

2. $\langle \forall u : sRu : \langle \exists v :: wRv \wedge uRv \rangle \rangle$

Let S_1 and S_2 be two transition systems which belong to different levels of hierarchy. S_1 simulates S_2 if every transition in S_1 can correspond to a similar transition in S_2. Here S_1 is the lower level model (implementation) and S_2 is the higher level (specification) model.

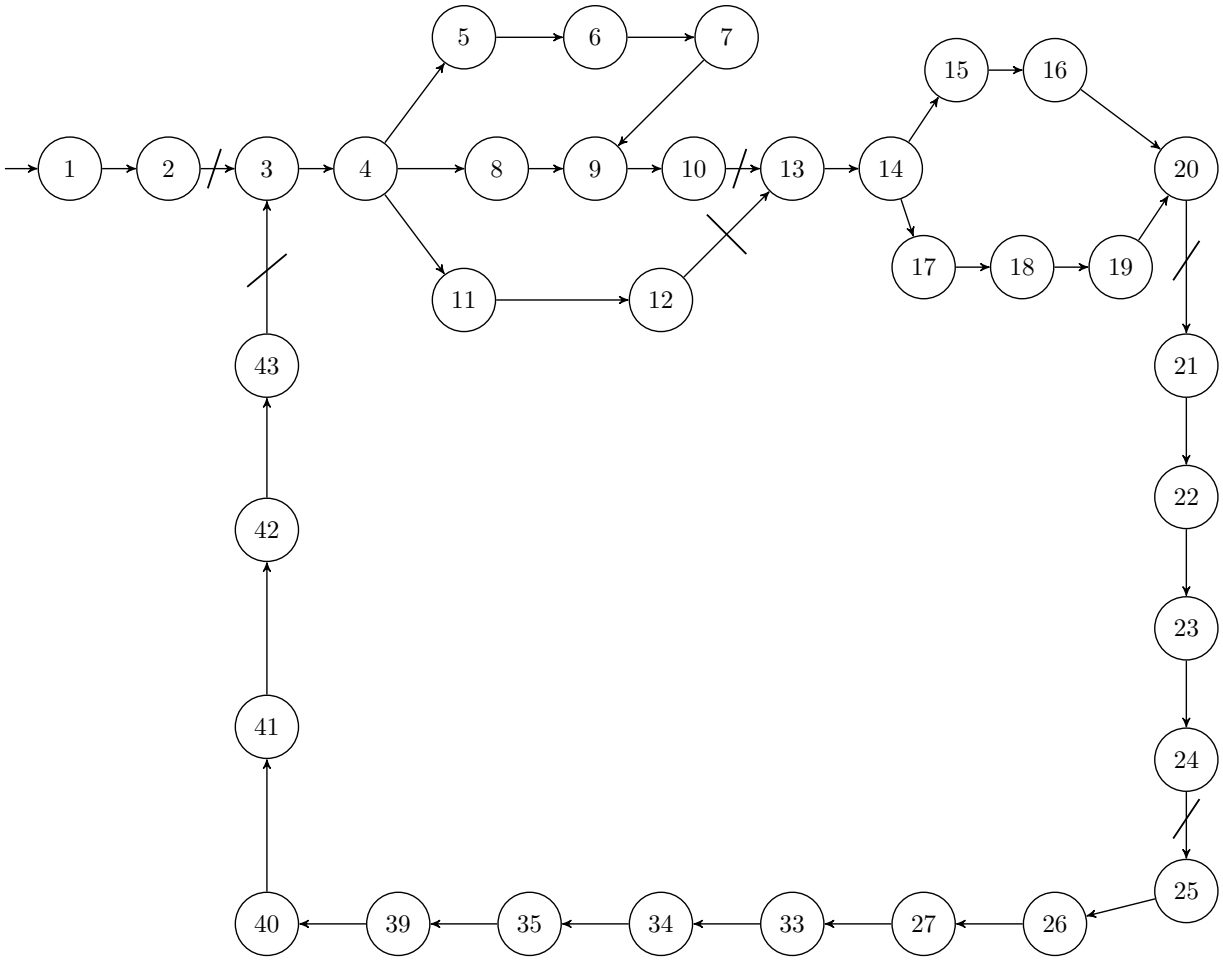


Figure 2.6. Stepper Motor Control Implementation TS

B is a *bisimulation relation* [45] on a transition system $M = \langle S, R, L \rangle$ if $B \subseteq S$ x S and for s, w such that sBw then

1. $L.s = L.w$
2. $\langle \forall u : sRu : \langle \exists v :: wRv \wedge uBv \rangle \rangle$
3. $\langle \forall v : wRv : \langle \exists u :: sRu \wedge uBv \rangle \rangle$

s is similar to w if there exists a simulation relation R such that sRw. Similarly s is bisimilar to w if there exists a bisimulation relation B such that sBw. It must be noted that if s is similar to w and w is similar to s, then s and w are not bisimilar. There exist a greatest simulation and a greatest bisimulation. The greatest simulation is preorder and the greatest bisimulation is an equivalence relation.

Manolios [45] has developed a notion of correctness for transition systems based on the simulation relation and bisimulation relation. His work provides reasoning about a single-step rather than reasoning about infinite computations. An implementation is correct with respect to a specification if the implementation is related to the specification as described by the notion of correctness. A step of the specification may translate to multiple steps of the implementation.

A relation $R \subseteq S$ x S is a *stuttering simulation* (STS) [45] on TS $M = \langle S, R, L\rangle$ iff for all s, w such that sBw:

1. $L.s = L.w$
2. $\langle \sigma : fp.\sigma.s : \langle \exists \delta : fp.\delta.w : match(B, \sigma, \delta)\rangle\rangle$

2.4.1. WFS Refinement

Well-founded simulation (WFS) refinement is a notion of correctness which describes how an implementation transition system is correctly implemented with respect to its specification transition system. WFS refinement deals with only functional correctness. The formal definitions and a detailed description of the WFS refinement are provided in [45]. Below is the definition of WFS

Definition 2. *[45] $B \subseteq S \times S$ is a WFS on TS $\mathcal{M} = \langle S, T, L\rangle$ iff:*

(1) $\langle \forall s, w \in S :: sBw \quad :: \quad L(s) = L(w)\rangle$; and

(2) There exist functions, $rankl : S \times S \times S \to \mathbb{N}$, $rankt : S \times S \to W$,
such that $\langle W, \prec\rangle$ is well-founded, and
$\langle \forall s, u, w \in S :: sBw \wedge sTu \quad ::$

(a) $\langle \exists v :: wTv \wedge uBv\rangle \vee$

(b) $(uBw \wedge rankt(u, w) \prec rankt(s, w)) \vee$

(c) $\langle \exists v :: wTv \wedge sBv \wedge rankl(v, s, u) < rankl(w, s, u)\rangle\rangle$

Here condition 1 states that a relation exists between states s and w and there have the same labels. In condition 2, case (a) denotes the non-stuttering transition on the implementation side, case (b) denotes the stuttering on the specification side and case (c) denotes stuttering on the implementation side. Progress on the model is denoted by a non-stuttering transition where the

states have different labels. Whereas in the case of stuttering transition the states have the same label. *rankt* and *rankl* are called as rank functions (discussed later).

When the specification and implementation systems are modeled as TSs, a step in the specification could be translated to multiple steps in the implementation. Hence, *stuttering is a phenomenon where multiple but finite transitions of implementation can match to the same specification transition.* WFS refinement has two key features: refinement-map and stuttering. Refinement-map, r, is a function that, given an implementation state, gives the corresponding specification state. A function like refinement-map is needed to bridge the abstraction gap between the implementation and specification systems.

The advantage of using WFS refinement is it is sufficient to reason about a single step of transition in the implementation and specification to check for correctness and find bugs. This makes WFS refinement applicable to deal with the complexity of object code. To avoid deadlock situations, in stuttering transitions, a witness function called rank is designed such that it decreases with each transition. In Definition 2, two rank functions *rankt* and *rankl* corresponds to the stuttering on specification and implementation side respectively.

Next is the definition of WFS refinement.

Definition 3. *[45] (WFS Refinement) Let* $M = \langle S, T, L\rangle$, $M' = \langle S', T', L'\rangle$, *and* $r : S \to S'$. *We say that* M *is a WFS refinement of* M' *with respect to refinement-map* r, *written* $M \sqsubseteq_r M'$, *if there exists a relation,* B, *such that* $\langle \forall s \in S :: sBr(s)\rangle$ *and* B *is a WFS on the TS* $\langle S \uplus S', T \uplus T', \mathcal{L}\rangle$, *where* $\mathcal{L}(s) = L'(s)$ *for* s *an* S' *state and* $\mathcal{L}(s) = L'(r(s))$ *otherwise.*

In the above definition, M and M' are the implementation and the specification TS respectively and r is the refinement-map function. The definition can be interpreted as every implementation transition should either be a stuttering or a non-stuttering transition. If it can be proved that an implementation TS is a WFS refinement of a specification TS, then every behavior of the implementation is guaranteed to match a behavior of the specification. If the implementation TS has a label that does not match to its specification TS or a transition that does not match in the specification TS, then these situations correspond to a bug in the implementation model (here $M \not\sqsubseteq_r M'$). If M is a WFS refinement of M' then, a state in M cannot be related to more than one state in M' (as the refinement-map is a function which is used to relate states of M to M').

WFS refinement satisfies composition property and is given by the theorem below.

Theorem 1. *[45] (Composition for WFS refinement) If* $M_a \sqsubseteq_r M_b$ *and* $M_b \sqsubseteq_q M_c$ *then* $M_a \sqsubseteq_{r;q} M_c$.

The above theorem states that the refinement is a compositional notion. Here, M_a is a WFS refinement of M_b w.r.t refinement map r is denoted as $M_a \sqsubseteq_r M_b$ and $r; q$ denotes composition, *i.e.*, $(r; q)(\mathrm{x}) = q(r(\mathrm{x}))$.

2.4.2. WEB Refinement

A formal and detailed description of Well-founded Equivalence Bisimulation (WEB) refinement is provided in [44, 45]. Here, a brief overview of the key features is given. As stated earlier, in the context of refinement, both the implementation and specification are treated as transition systems. Informally, the implementation behaves correctly as given by the specification, if every behavior of the implementation is matched by a behavior of the specification and vice versa. However, the implementation and specification may not have the same transition behavior.

Another issue is that to check equivalence, specification states and implementation states need to be compared. However, these states can look very different. In the stepper motor specification, each state is a four-bit value. However, the implementation state in this example includes registers and memory in the microcontroller. WEB refinement employs refinement maps, functions that map implementation states to specification states, to bridge this abstraction gap. Below is the definition of WEB.

Definition 4. *[45]* $B \subseteq S \times S$ *is a WEB on TS* $\mathcal{M} = \langle S, T, L\rangle$ *iff:*

(1) B is an equivalence relation on S; and

(2) $\langle \forall s, w \in S :: sBw \quad :: \; L(s) = L(w)\rangle$; and

(3) There exist functions $erankl : S \times S \to \mathbb{N}$, $erankt : S \to W$,

such that $\langle W, \lessdot\rangle$ is well-founded, and

$\langle \forall s, u, w \in S :: sBw \wedge sTu \quad ::$

(a) $\langle \exists v :: wTv \wedge uBv\rangle \vee$

(b) $(uBw \wedge erankt(u) \lessdot erankt(s)) \vee$

(c) $\langle \exists v :: wTv \wedge sBv \wedge erankl(v, u) < erankl(w, u)\rangle\rangle$

In the third condition, case (b) denotes stuttering on the specification side and case (c) denotes stuttering on the implementation side. In practice, stuttering rarely occurs on the specification side. To check WEB refinement, it is enough to reason about single transitions of the implementation and the specification.

Next is the definition of a WEB refinement.

Definition 5. *[45] (WEB Refinement) Let* $M = \langle S, T, L\rangle$, $M' = \langle S', T', L'\rangle$, *and* $r : S \to S'$. *We say that* M *is a WEB refinement of* M' *with respect to refinement map* r, *written* $M \approx_r M'$, *if there exists a relation,* B, *such that* $\langle \forall s \in S :: sBr(s)\rangle$ *and* B *is a WEB on the TS* $\langle S \uplus S', T \uplus T', \mathcal{L}\rangle$, *where* $\mathcal{L}(s) = L'(s)$ *for* s *an* S' *state and* $\mathcal{L}(s) = L'(r(s))$ *otherwise.*

Refinement is a compositional notion as given by the following theorem [46]. Below, $M_c \approx_r M_b$ denotes that M_c is a WEB refinement of M_b; and $r; q$ denotes composition, i.e. $(r; q)(s) = q(r(s))$.

Theorem 2. *[46] (Composition for WEB Refinement) If* $M_c \approx_r M_b$ *and* $M_b \approx_q M_a$ *then* $M_c \approx_{r;q} M_a$.

3. CASE STUDIES

This chapter details the two control programs used extensively to apply the proposed notion of refinement. The implementation model for these control programs is the device object code instead of its corresponding source code. Device object code verification will find bugs that are introduced during compilation. Also, for interrupt driven programs, the implementation behavior is more accurately captured at the object level (as explained in Section 2.3.2). In formal verification, the design artifact consists both implementation model and specification model. So, this chapter explains the specification model for all the benchmarks used in the rest of the dissertation. All the benchmarks were developed to run on an ARM Cortex-M3 based NXP LPC1768 [58] micro-controller.

3.1. Stepper Motor Control Program

A number of object code programs for stepper motor control were used as benchmarks to demonstrate the effectiveness of the proposed methodology. A stepper motor is a brushless DC electric motor. Current pulse applied to the motor generates discrete rotation of the motor shaft. A stepper motor can have 4 or 6 leads. Consider a motor with 4 leads say a, b, c, and d. It can be energized in various repeated sequences that cause it to rotate. A general specification model and implementation model for stepper motor are shown in Figure 2.4 and Figure 2.6, respectively. Three typically used sequences were used to develop the benchmarks.

1. ***Full stepping or single stepping or wave stepping*** **:** This sequence is expressed as $\langle 0001 \rangle$, $\langle 0010 \rangle$, $\langle 0100 \rangle$, $\langle 1000 \rangle$, $\langle 0001 \rangle$. A stepper motor has two pairs of stator windings. The two phases are energized in alternate fashion and in reverse polarity. Just as the rotor aligns with one of the stator poles, the second phase is energized. There are four steps and hence four discrete state transitions are required for one rotation of the stepper motor.

2. ***Double stepping*** **:** This sequence is expressed as $\langle 0011 \rangle$, $\langle 0110 \rangle$, $\langle 1100 \rangle$, $\langle 1001 \rangle$, $\langle 0011 \rangle$.

3. ***Half stepping*** **:** This sequence is expressed as $\langle 0001 \rangle$, $\langle 0011 \rangle$, $\langle 0010 \rangle$, $\langle 0110 \rangle$, $\langle 0100 \rangle$, $\langle 1100 \rangle$, $\langle 1000 \rangle$, $\langle 1001 \rangle$, $\langle 0001 \rangle$. The stepper motor has eight steps instead of four. This is cause because the second phase is turned on before the first phase is turned off. Hence, two phases

are energized at the same time. The rotor holds on between two full-step positions. There are eight discrete state transitions for one rotation of the stepper motor.

For the stepper motor, the values on the leads determine the state, the transition relation determines the speed and direction of the shaft. The direction of the shaft is classified as "***clock***" and "***anti***" which indicate that the motor was controlled in clockwise and anti-clockwise direction, respectively.

A stepper motor also has ***microstepping*** as a sequence. In microstepping technique, the stator flux is moved smoothly. The step angle in this technique is divided into multiple subdivisions to improve the control over the motor. This technique is used in applications where a refined motor work with greater resolution is needed. The stepper motor produces less vibration and makes noiseless stepping possible with no detectable "stepping". Hence it was not implemented as a benchmark.

The goal of this dissertation is verification of interrupt-driven real-time object code programs. The time delay is introduced in the source code by using of either a timer or a counter. "***RIT***" indicates that the interrupts were generated by Repetitive Interrupt Timer (RIT) to implement the timing delays for the motor control. "***noRIT***" indicates that instead of the RIT timer, code/counters was to implement timing delays.

3.1.1. Case Study A - Interrupt Driven Full Stepping Stepper Motor Control in Clockwise Direction

Stepper motor control with full stepping is implemented with the Repetitive Interrupt Timer (RIT), which is a timer present in the LPC1768. The controller microcode enables the RIT unit and also a register that RIT has to store a constant value. Then the code enters a while loop. The RIT has a counter which increments every clock cycle. When the counter reaches the value stored in the RIT register, an interrupt is generated. As soon as the interrupt is generated, the counter is reset to 0 and flow of control changes to the RIT interrupt service routine (ISR). In the RIT ISR, the FIOPIN register is updated to the next value of the leads required for full stepping, and then returns control to the main program. The source code for this case study is shown in Appendix B. The object code that is obtained from this source code is shown in Appendix C. It has to be noted that just the object code would not make any sense without knowing where to start from and also

where does the interrupt routine starts from. All this additional information is needed along with the object code.

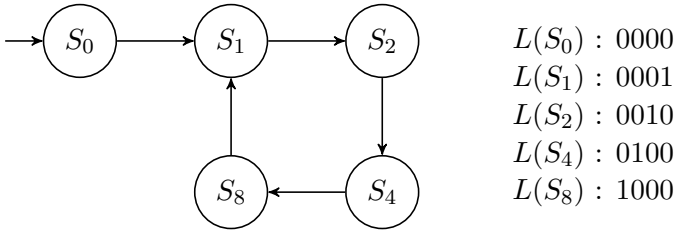


Figure 3.1. Stepper Motor Control Specification TS for Interrupt-Driven Full Stepping in Clockwise Direction

The specification model for interrupt driven full stepping stepper motor control in clockwise direction transition system is shown in Figure 3.1. The set of states $S = \{S_0, S_1, S_2, S_4, S_8\}$, with S_0 as the initial state, the transition relation $T = \{\langle S_0, S_1\rangle, \langle S_1, S_2\rangle, \langle S_2, S_4\rangle, \langle S_4, S_8\rangle, \langle S_8, S_1\rangle\}$ and the labeling function is shown on the left.

3.1.2. Case Study B - Interrupt Driven Full Stepping Stepper Motor Control in Anti-Clockwise Direction

This interrupt control mechanism is similar to case study A. The RIT unit is employed here. The ISR is modified to update the FIOPIN register based on full stepping in anti-clockwise direction.

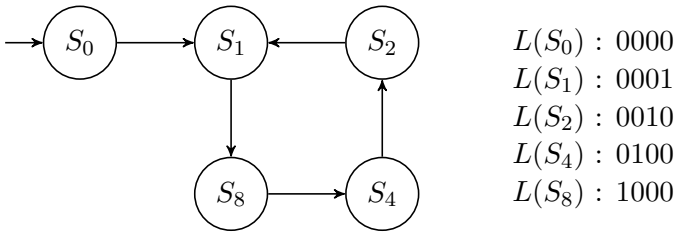


Figure 3.2. Stepper Motor Control Specification TS for Interrupt-Driven Full Stepping in Anti-Clockwise Direction

The specification model for interrupt driven full stepping stepper motor control in anti-clockwise direction transition system is shown in Figure 3.2. The set of states $S = \{S_0, S_1, S_2, S_4, S_8\}$,

with S_0 as the initial state, the transition relation $T = \{\langle S_0, S_1\rangle, \langle S_1, S_8\rangle, \langle S_8, S_4\rangle, \langle S_4, S_2\rangle, \langle S_2, S_1\rangle\}$ and the labeling function is shown on the left.

3.1.3. Case Study C - Interrupt Driven Double Stepping Stepper Motor Control in Clockwise Direction

This interrupt control mechanism is similar to case study A. The RIT unit is employed here. The ISR is modified to update the FIOPIN register based on double stepping in clockwise direction.

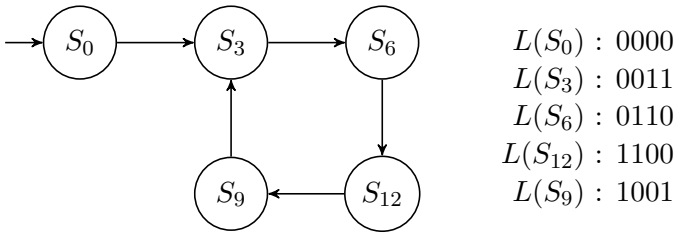


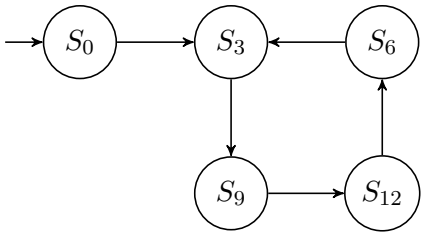
Figure 3.3. Stepper Motor Control Specification TS for Interrupt-Driven Double Stepping in Clockwise Direction

The specification model for interrupt driven double stepping stepper motor control in clockwise direction transition system is shown in Figure 3.3. The set of states $S = \{S_0, S_3, S_6, S_{12}, S_9\}$, with S_0 as the initial state, the transition relation $T = \{\langle S_0, S_3\rangle, \langle S_3, S_6\rangle, \langle S_6, S_{12}\rangle, \langle S_{12}, S_9\rangle, \langle S_9, S_3\rangle\}$ and the labeling function is shown on the left.

3.1.4. Case Study D - Interrupt Driven Double Stepping Stepper Motor Control in Anti-Clockwise Direction

This interrupt control mechanism is similar to case study A. The RIT unit is employed here. The ISR is modified to update the FIOPIN register based on double stepping in anti-clockwise direction.

The specification model for interrupt driven double stepping stepper motor control in anti-clockwise direction transition system is shown in Figure 3.4. The set of states $S = \{S_0, S_3, S_6, S_{12}, S_9\}$, with S_0 as the initial state, the transition relation $T = \{\langle S_0, S_3\rangle, \langle S_3, S_9\rangle, \langle S_9, S_{12}\rangle, \langle S_{12}, S_6\rangle, \langle S_6, S_3\rangle\}$ and the labeling function is shown on the left.

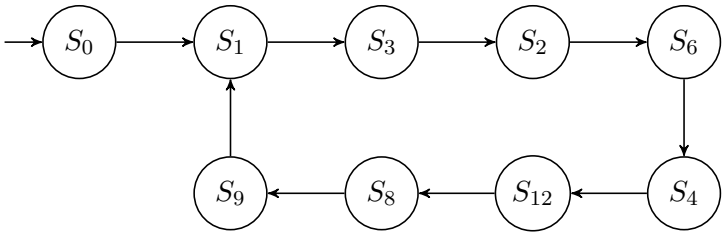


$L(S_0)$: 0000
$L(S_3)$: 0011
$L(S_6)$: 0110
$L(S_{12})$: 1100
$L(S_9)$: 1001

Figure 3.4. Stepper Motor Control Specification TS for Interrupt-Driven Double Stepping in Anti-Clockwise Direction

3.1.5. Case Study E - Interrupt Driven Half Stepping Stepper Motor Control in Clockwise Direction

This interrupt control mechanism is similar to case study A. The RIT unit is employed here. The ISR is modified to update the FIOPIN register based on half stepping in clockwise direction.



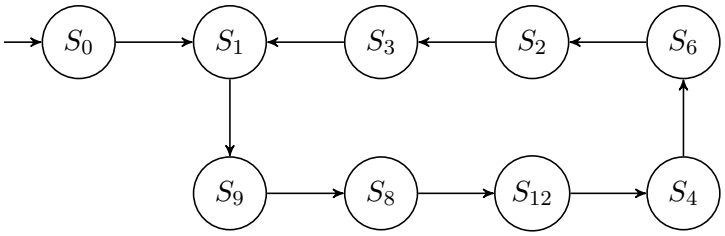
$L(S_0)$: 0000
$L(S_1)$: 0001
$L(S_3)$: 0011
$L(S_2)$: 0010
$L(S_6)$: 0110
$L(S_4)$: 0100
$L(S_{12})$: 1100
$L(S_8)$: 1000
$L(S_9)$: 1001

Figure 3.5. Stepper Motor Control Specification TS for Interrupt-Driven Half Stepping in Clockwise Direction

The specification model for interrupt driven half stepping stepper motor control in clockwise direction transition system is shown in Figure 3.5. The set of states $S = \{S_0, S_1, S_3, S_2, S_6, S_4, S_{12}, S_8, S_9\}$, with S_0 as the initial state, the transition relation $T = \{\langle S_0, S_1\rangle, \langle S_1, S_3\rangle, \langle S_3, S_2\rangle, \langle S_2, S_6\rangle, \langle S_6, S_4\rangle, \langle S_4, S_{12}\rangle, \langle S_{12}, S_8\rangle, \langle S_8, S_9\rangle, \langle S_9, S_1\rangle\}$ and the labeling function is shown on the left.

3.1.6. Case Study F - Interrupt Driven Half Stepping Stepper Motor Control in Anti-Clockwise Direction

This interrupt control mechanism is similar to case study A. The RIT unit is employed here. The ISR is modified to update the FIOPIN register based on half stepping in anti-clockwise direction.



$L(S_0) : 0000$
$L(S_1) : 0001$
$L(S_3) : 0011$
$L(S_2) : 0010$
$L(S_6) : 0110$
$L(S_4) : 0100$
$L(S_{12}) : 1100$
$L(S_8) : 1000$
$L(S_9) : 1001$

Figure 3.6. Stepper Motor Control Specification TS for Interrupt-Driven Half Stepping in Anti-Clockwise Direction

The specification model for interrupt driven half stepping stepper motor control in anti-clockwise direction transition system is shown in Figure 3.6. The set of states $S = \{S_0, S_1, S_3, S_2, S_6, S_4, S_{12}, S_8, S_9\}$, with S_0 as the initial state, the transition relation $T = \{\langle S_0, S_1\rangle, \langle S_1, S_9\rangle, \langle S_9, S_8\rangle, \langle S_8, S_{12}\rangle, \langle S_{12}, S_4\rangle, \langle S_4, S_6\rangle, \langle S_6, S_2\rangle, \langle S_2, S_3\rangle, \langle S_3, S_1\rangle\}$ and the labeling function is shown on the left.

3.1.7. Case Study G - Full Stepping Stepper Motor Control without Interrupts in clockwise direction

For this case study, full stepping control is implemented without using interrupts. The delay required between full stepping control states is achieved using for loops with a large number of iterations. The number of iterations of the for loop is determined so that the time required by the microcontroller to execute the for loop matches the delay required between full stepping control states. The drawback with this approach is that if the control program performs other functions and has enabled other interrupts, it may not be possible to guarantee accurate speed of the motor.

The specification model for full stepping stepper motor control without interrupts in clockwise direction transition system is similar to Figure 3.1. The set of states $S = \{S_0, S_1, S_2, S_4, S_8\}$, with S_0 as the initial state, the transition relation $T = \{\langle S_0, S_1\rangle, \langle S_1, S_2\rangle, \langle S_2, S_4\rangle, \langle S_4, S_8\rangle, \langle S_8, S_1\rangle\}$ and the labeling function is shown on the left. The specification transition system does not depend on how the delays are produced in the system.

3.1.8. Case Study H - Full Stepping Stepper Motor Control without Interrupts in Anti-Clockwise Direction

For this case study, the delay required between full stepping control states in anti-clockwise direction is achieved using for loops with a large number of iterations, instead of interrupts.

The specification model for interrupt driven full stepping stepper motor control in anti-clockwise direction transition system is similar to Figure 3.2. The set of states $S = \{S_0, S_1, S_2, S_4, S_8\}$, with S_0 as the initial state, the transition relation $T = \{\langle S_0, S_1\rangle, \langle S_1, S_8\rangle, \langle S_8, S_4\rangle, \langle S_4, S_2\rangle, \langle S_2, S_1\rangle\}$ and the labeling function is shown on the left. The specification transition system does not depend on how the delays are produced in the system.

3.1.9. Case Study I - Double Stepping Stepper Motor Control without Interrupts in Clockwise Direction

For this case study, the delay required between double stepping control states in clockwise direction is achieved using for loops with a large number of iterations, instead of interrupts.

The specification model for double stepping stepper motor control without interrupts in clockwise direction transition system is similar to Figure 3.3. The set of states $S = \{S_0, S_3, S_6, S_{12}, S_9\}$, with S_0 as the initial state, the transition relation $T = \{\langle S_0, S_3\rangle, \langle S_3, S_6\rangle, \langle S_6, S_{12}\rangle, \langle S_{12}, S_9\rangle, \langle S_9, S_3\rangle\}$ and the labeling function is shown on the left. The specification transition system does not depend on how the delays are produced in the system.

3.1.10. Case Study J - Double Stepping Stepper Motor Control without Interrupts in Anti-Clockwise Direction

For this case study, the delay required between double stepping control states in anti-clockwise direction is achieved using for loops with a large number of iterations, instead of interrupts.

The specification model for interrupt driven double stepping stepper motor control in anti-clockwise direction transition system is similar to Figure 3.4. The set of states $S = \{S_0, S_3, S_6, S_{12}, S_9\}$, with S_0 as the initial state, the transition relation $T = \{\langle S_0, S_3\rangle, \langle S_3, S_9\rangle, \langle S_9, S_{12}\rangle, \langle S_{12}, S_6\rangle, \langle S_6, S_3\rangle\}$ and the labeling function is shown on the left. The specification transition system does not depend on how the delays are produced in the system.

3.1.11. Case Study K - Half Stepping Stepper Motor Control without Interrupts in Clockwise Direction

For this case study, the delay required between half stepping control states in clockwise direction is achieved using for loops with a large number of iterations, instead of interrupts.

The specification model for interrupt driven half stepping stepper motor control in clockwise direction transition system is similar to Figure 3.5. The set of states $S = \{S_0, S_1, S_3, S_2, S_6, S_4, S_{12}, S_8,$

$S_9\}$, with S_0 as the initial state, the transition relation $T = \{\langle S_0, S_1\rangle, \langle S_1, S_3\rangle, \langle S_3, S_2\rangle, \langle S_2, S_6\rangle, \langle S_6, S_4\rangle, \langle S_4, S_{12}\rangle, \langle S_{12}, S_8\rangle, \langle S_8, S_9\rangle, \langle S_9, S_1\rangle\}$ and the labeling function is shown on the left. The specification transition system does not depend on how the delays are produced in the system.

3.1.12. Case Study L - Interrupt Driven Half Stepping Stepper Motor Control in Anti-Clockwise Direction

For this case study, the delay required between half stepping control states in anti-clockwise direction is achieved using for loops with a large number of iterations, instead of interrupts.

The specification model for interrupt driven half stepping stepper motor control in anti-clockwise direction transition system is similar to Figure 3.6. The set of states $S = \{S_0, S_1, S_3, S_2, S_6, S_4, S_{12}, S_8, S_9\}$, with S_0 as the initial state, the transition relation $T = \{\langle S_0, S_1\rangle, \langle S_1, S_9\rangle, \langle S_9, S_8\rangle, \langle S_8, S_{12}\rangle, \langle S_{12}, S_4\rangle, \langle S_4, S_6\rangle, \langle S_6, S_2\rangle, \langle S_2, S_3\rangle, \langle S_3, S_1\rangle\}$ and the labeling function is shown on the left. The specification transition system does not depend on how the delays are produced in the system.

3.1.13. Case Study M - Interrupt Driven Variable Speed Full Stepping Stepper Motor Control in Clockwise direction

The RIT unit is used to implement full stepping control. However, the motor has 3 speed modes and in each mode the motor runs at a different speed. The modes can be changed based on input from a keyboard, which acts as an external interrupt. When any key on the keyboard is pressed, an interrupt is generated (different from the RIT interrupt). The keyboard input is processed to change the speed of the motor. Note that since the control program supports 2 interrupts, the RIT interrupt is given the higher priority.

The specification model for interrupt driven variable speed full stepping stepper motor is not easy to depict as the state representation and state transition are no long represented with single bits. Two bits are needed to signify the system has received a signal in the form of an interrupt to change the speed of operation. "00" means that the system has normal operation, "01" indicates speed 2 as the operation speed and "10" indicates speed 3 as the operation speed. Here "11" is considered as bug in the model. Another bit is used to signify that the transition has happened from one speed to another speed. These bits have to be set and clear depending on the situation and mode of operation. Hence the specification TS is not shown as a figure.

3.1.14. Case Study N - Interrupt Driven Variable Half Full Stepping Stepper Motor Control in Clockwise directon

3 mode variable speed is implemented using half stepping. The input from keyboard (which acts as an external interrupt) and RIT interrupt are also employed here, with the RIT interrupt having higher priority.

The specification model for interrupt driven variable speed half stepping stepper motor is not easy to depict as the state representation and state transition are no long represented with single bits. Two bits are needed to signify the system has received a signal in the form of an interrupt to change the speed of operation. "00" means that the system has normal operation, "01" indicates speed 2 as the operation speed and "10" indicates speed 3 as the operation speed. Here "11" is considered as bug in the model. Another bit is used to signify that the transition has happened from one speed to another speed. These bits have to be set and clear depending on the situation and mode of operation. Hence the specification TS is not shown as a figure.

3.2. Infusion Pump Control Program

An infusion pump is a medical device that is used to deliver controlled dosages of medications or nutrients into the patient's circulatory system intravenously. Typical medications delivered include opioids, insulin, and chemotherapy drugs. Food and Drug Administration (FDA) has issued 54 Class-1 recalls on infusion pumps due to software errors from 2001 to 2017. The criticality of the pump functionality is because of incorrect dosage delivery due to software errors. We have used the Alaris Medley 8100 LVP module infusion pump [1,13] for our experiments. The Alaris Medley pump uses pulse width modulation for dosage control. We implemented the pulse width modulation control code for the Alaris pump on an ARM Cortex M3 based LPC 1768 micro-controller, which was interfaced with the pump so that our code implementation can control the pump. We also developed formal specifications for the pump control software based on the requirements in [67]. The transition system of the pump's control code had about 24.3 million transitions.

3.2.1. Pulse Width Modulation (PWM)

Pulse Width Modulation (PWM) is a control mechanism that controls the power supply to electrical device. PWM has a wide range of applications including servo control, telecommunication,

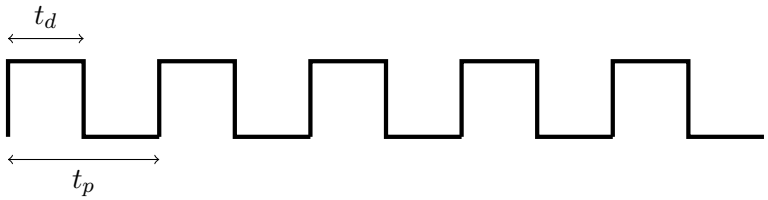


Figure 3.7. Pulse Width Modulation Signal for Infusion Pump Control

power delivered to a load, to regulate the voltage, for audio amplification and generation of electrical signals. Any of these techniques can be used to control a safety critical application.

The PWM waveform (as shown in Figure 3.7) can be generated by a controller to control the motor to which it is connected to. The average value of voltage (and current) that is fed to the motor can be controlled by turning the switch between the supply and motor on and off at a fast rate. Hence, PWM is a binary signal with two discrete values of 0 and 1. The width of the signal can be varied between 0 and the time period of the pulse (t_p). Duty cycle (t_d) is defined on the the time period during which the signal is high. The duty cycle determines the amount of energy delivered and hence controls the speed of the motor. The greater the duty cycle, faster is the speed of the motor. The time period for which s_{on} is active determines the speed of the motor.

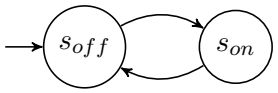


Figure 3.8. PWM Specification TS

The specification model for PWM as a transition system is shown in Figure 3.8. It has two discrete states s_{off} and s_{on}. Initially the signal is zero indicated by s_{off}. The signal becomes high indicated by s_{on}. Then the sequence keeps repeating. The set of states $S = \{s_{off}, s_{on}\}$, the transition relation $T = \{\langle s_{off}, s_{on}\rangle, \langle s_{on}, s_{off}\rangle\}$ and the labeling function $L(s_{off}) = 0$ and $L(s_{on}) = 1$.

4. TIMED REFINEMENT

The transitions in a transition system (TS) (described in Section 2.2) can be classified as *stuttering transitions* and *non-stuttering transitions* which are defined next.

Definition 6. *In a transition system (TS) $M = \langle S, T, L \rangle$, a transition of the form $\langle w, v \rangle$ is called a stuttering transition s_t, if:*

1. *The states $w, v \in S$*
2. *The transition $\langle w, v \rangle \in T$*
3. *$L(w) = L(v)$*

For a transition to be stuttering, it should exist in the transition relation T and also should have the same labelling function.

Definition 7. *In a transition system (TS) $M = \langle S, T, L \rangle$, a transition of the form $\langle w, v \rangle$ is called a non-stuttering transition n_t, if:*

1. *The states $w, v \in S$*
2. *The transition $\langle w, v \rangle \in T$*
3. *$L(w) \neq L(v)$*

For a transition to be non-stuttering, it should exist in the transition relation T and the states in the transition should not have the same labelling function.

4.1. Specification TTS of Stepper Motor Control

Now, consider the stepper motor controller that was discussed in Section 2.3. Microcontrollers can usually be used to control the working of a stepper motor. A software code can be programmed into a microcontroller which is interfaced with the motor. The microcontroller will generate a sequence of values which can be loaded onto the leads of a stepper motor. The functionality of the controller is not fully described unless the speed of rotation is specified. The speed of the motor depends on the angle the motor rotates at each step. The speed is determined by the

time delay of each transition. The delay (d) of each transition is given by: $d = \Phi/6r$, where, Φ is the degree of rotation for each step of the motor, and r is the rotational speed of the motor in rpm. In commercial applications, there is typically a tolerance in the speed of the motor. For example, the expected speed is 100rpm, but it is acceptable for the speed to vary between 96rpm and 104rpm. This tolerance results in a lower limit and an upper limit (lb and ub) on the delay of each transition. If $\Phi = 1.8^o$, then for this example, lb=2.884ms (corresponding to speed 104rpm) and ub=3.125ms (corresponding to speed 96rpm). However, these timing requirements on the transitions cannot be incorporated in a transition system specification. Therefore, timed transition systems is used to specify the specification.

Definition 8. *A Timed Transition System (TTS) M is a 3-tuple $\langle S, T, L\rangle$, where S is the set of states, T is the transition relation that defines the state transitions, and L is a labeling function that defines what is visible at each state. T is of the form $\langle w, v, lb, ub\rangle$, where $w, v \in S$ and lb, ub are non-negative integers that indicate the lower bound and the upper bound on the time delay of the transition, respectively.*

The above definition is based on the TTS model from Henzinger *et al.* [34], which gives a very detailed description of this TTS model. Note that this model is very amenable for modeling and refinement-based verification of real-time object code programs. The time interval is considered as lower and upper bound since tolerance is allowable in the speed of real-time applications. If the delay bounds are excluded from the above definition, it corresponds to a transition system (TS), which captures only the functional behavior and not the timing behavior (TS is defined in Definition 1).

Figure 4.1 shows a timed transition system (TTS) specification for the stepper motor control with transition system shown in Figure 2.4. The timing requirements are marked on the transitions with lb=2.884ms, and ub=3.125ms.

4.2. Implementation TTS of Stepper Motor Control

The target is to verify the stepper motor object code control program. The implementation model is obtained by generating a function for each instruction that describes the effect of the instruction on the state of the microcontroller. The state of a microcontroller includes the registers and memory of the microcontroller. The set of all such functions (one for each instruction) and the

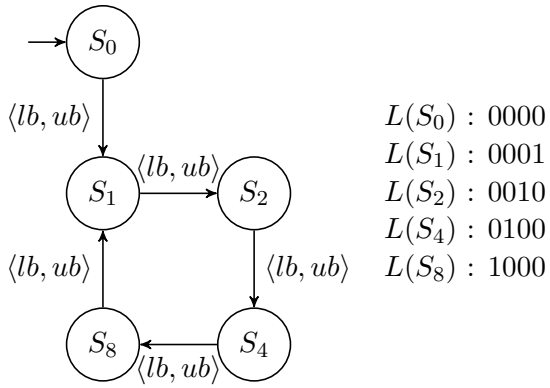


Figure 4.1. Stepper Motor Control Specification TTS

initial state of the microcontroller defines the TS model (Figure 2.6) of the implementation. This set includes the instructions in interrupt service routines of the interrupts that the program uses. Each instruction is also associated with a lower bound and an upper bound on the instruction execution time. A number of techniques and tools exist for timing analysis, and to determine worst case execution time (WCET) and best case execution time (BCET) [66]. The goal is not timing analysis, but functional and timing verification. Also, the lower and upper delay bounds on the specification side indicate requirements. On the implementation side, the delay bounds are an estimate of the lower limit and upper limit of the execution time of the instruction/transition. The delay bounds on the implementation side are used to verify if the implementation satisfies the timing requirements of the specification. For real-time object code, the timing properties are expressed in terms of the number of clock cycle needed to decode an instruction.

Figure 4.2 is an implementation TTS of the implementation TS (shown in Figure 2.6) which has been constructed for the stepper motor specification TTS shown in Figure 4.1. This example is used to introduce the notion of timed refinement.

4.3. Timed Refinement

The correct functioning of the stepper motor control program depends also on whether it meets the timing requirements of the specification. The notion of WFS refinement or WEB refinement does not consider time. The notion of timed refinement is introduced that accounts for timing requirements in the context of refinement. Timed refinement is defined in the context of timed transition systems (TTS).

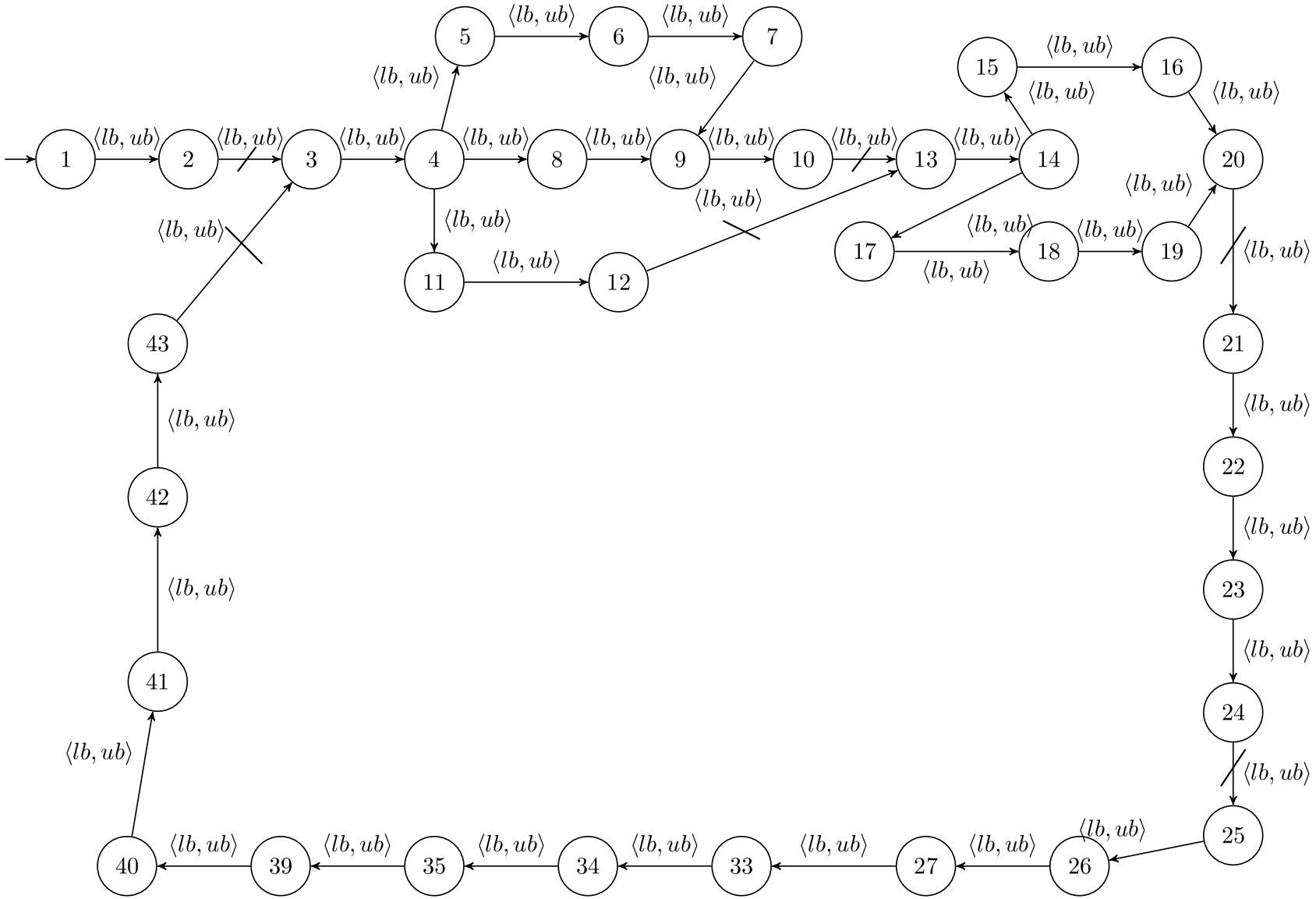


Figure 4.2. Stepper Motor Control Implementation TTS

If there is no stuttering between the implementation TTS (M_I) and the specification TTS (M_S), every step of M_I should match a step of M_S, and the delay of an implementation step should "match" the delay of the corresponding specification step. By "match" it means the following should be satisfied:

$$lb_s \leq lb_i \leq ub_i \leq ub_s$$

where, lb_s, ub_s, lb_i, and ub_i are the lower and upper delay bounds on corresponding specification and implementation steps, respectively. If stuttering is involved, which is the case for the stepper motor example (and would be the case for most real world examples), then the requirements on the relationship between M_I and M_S is more complicated. The reason being that with stuttering, multiple but finite steps of M_I can match a single step of M_S. Also, the number of stuttering steps in each situation is arbitrary and depends on the behavior of the implementation that needs to be verify.

Timed refinement is based on the idea that the implementation TTS M_I satisfies the timing requirements of its specification TTS M_S, if in every case, the delay between the previous time that M_I made progress w.r.t. M_S and the next time M_I makes progress w.r.t. M_S, matches the time delay required for M_S to make that progress.

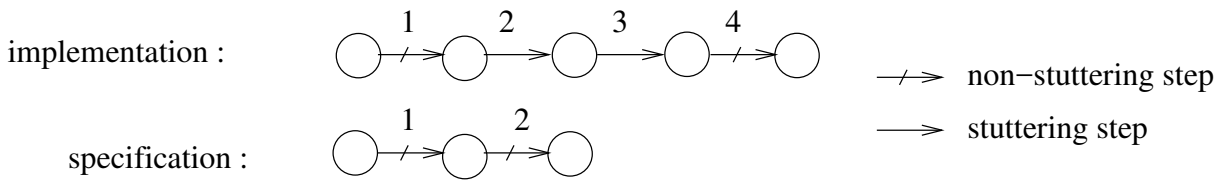


Figure 4.3. Example Comparing Implementation and Specification Transitions.

Timed refinement can be illustrated further with the example in Figure 4.3. In the figure, if step 1 of the implementation matches step 1 of the specification, and step 4 of the implementation matches step 2 of the specification, then steps 2 and 3 are stuttering steps of the implementation. Also, steps 1 and 4 of the implementation are non-stuttering steps. Progress on the implementation side corresponds to the non-stuttering steps. For this example, the following should be satisfied:

$$lb_s^2 \leq \sum_{n=2}^{4} lb_i^n \leq \sum_{n=2}^{4} ub_i^n \leq ub_s^2$$

In the above and in the following discussions, in lb^n and ub^n, superscript n indicates transition n. The idea of timed refinement can be formalized. In defining a timed refinement between an M_I and an M_S, it is assumed that a refinement relationship already exists between the two. In practice, what this means is that verification of timed refinement is preceded by verification of WFS refinement or WEB refinement. Since the refinement relationship has been established and a witness refinement map exists, the stuttering and non-stuttering transitions of the implementation TTS M_I can be identified. This information is captured in a Marked TTS, which is defined below,

Definition 9. *A Marked TTS MM is a TTS where every transition of the TTS is marked with a label s_t or n_t, indicating that the transition is a stuttering transition or a non-stuttering transition, respectively.*

Transitions of a marked TTS are of the form $\langle w, v, lb, ub, m\rangle$ where m $\in \{s_t, n_t\}$ (s_t and n_t indicating stuttering and non-stuttering transitions, respectively). The general theory of WFS refinement (or WEB refinement) allows for stuttering to occur on the implementation side and the specification side. In practice, situations in which the specification stutters is rare. Therefore, an assumption can be made that the implementation can have stuttering steps. Hence, for timed refinement, stuttering is ignored on the specification side. For object code verification, stuttering rarely occurs on the specification side as the implementation (object code) typically has a much larger number of transitions (millions) when compared with the specification. A marked specification TTS MM_S corresponding to a specification TTS M_S, is one in which every transition is a non-stuttering transition and is marked with n_t.

Definition 10. *A marked implementation TTS MM_I of TTS M_I w.r.t a marked specification TTS MM_S is a marked TTS where for every transition of MM_I of the form $\langle w, v, lb, ub, m\rangle$, if $r(w) = r(v)$, then m=s_t, else m=n_t. r is the refinement map used to establish that M_I is a WEB refinement of M_S.*

MM_I will satisfy the timing requirements of the corresponding MM_S, if every time the implementation makes progress (non-stuttering step), then the sum of the delay of the non-stuttering step and the delays of all preceding stuttering steps matches the delay of the corresponding specification step. Note that there may be many paths in the implementation that lead to a specific

non-stuttering step. For example in Figure 4.2, there exist multiple paths to reach state 13 from state 3. All these paths individually should satisfy the timing requirements of the corresponding specification step. These finite paths as called as stuttering segments, which are defined below. Note that in the following discussions, the delay bounds from T are omitted, if the delay bounds are not relevant for the discussion.

Definition 11. *A stuttering segment (ϕ) of a non-stuttering step $\langle w_a, w_b \rangle$ of an MM_I is a sequence of steps of MM_I $\{\langle w_{n-1}, w_{n-2} \rangle$, $\langle w_{n-2}, w_{n-3} \rangle$, ... , $\langle w_2, w_1 \rangle$, $\langle w_1, w_a \rangle$, $\langle w_a, w_b \rangle\}$, such that:*

1. *For all i such that $2 \leq i \leq n-1$, $\langle w_i, w_{i-1} \rangle$ is a stuttering step of MM_I.*
2. *$\langle w_n, w_{n-1} \rangle$ is a non-stuttering step of MM_I.*
3. *$\langle w_1, w_a \rangle$ is a stuttering step of MM_I.*

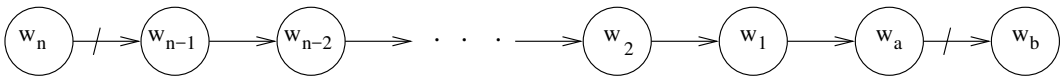


Figure 4.4. Stuttering Segment

The above definition is illustrated in Figure 4.4. Note that the least length of a stuttering segment is one. This occurs when a non-stuttering step is preceded by another non-stuttering step. The stuttering segment then only consists of one transition which is the non-stuttering step. Also, a non-stuttering step can have many associated stuttering segments. For the TTS shown in Figure 4.2, the stuttering segments of $\langle 10, 13 \rangle$ are:

1. $\{\langle 3, 4 \rangle, \langle 4, 8 \rangle, \langle 8, 9 \rangle, \langle 9, 10 \rangle, \langle 10, 13 \rangle\}$
2. $\{\langle 3, 4 \rangle, \langle 4, 5 \rangle, \langle 5, 6 \rangle, \langle 6, 7 \rangle, \langle 7, 9 \rangle, \langle 9, 10 \rangle, \langle 10, 13 \rangle\}$.

The idea of stuttering segments when combined with suitable abstractions for timed refinement verification significantly mitigates the path explosion problem that is often encountered in verification of interrupt driven control programs. The reason being that verification is reduced to analyzing stuttering segments.

4.3.1. Timed Well-Founded Simulation Refinement

Timed refinement can be defined with respect to Well Founded Simulation (WFS). A notion of correctness called Timed Well-Founded Simulation Refinement (TWFS) which is builds on WFS refinement describes how an implementation satisfies both functional and timing requirements of its specification. TWFS is defined below:

Definition 12. *M_I is a timed well-founded simulation (TWFS) refinement of M_S if:*

1. *M_I is a WFS refinement of M_S w.r.t. refinement map r.*
2. *Let MM_I be the marked TTS of M_I w.r.t. M_S. Then, for every non-stuttering transition of MM_I $\langle w_a, w_b \rangle$, and for every stuttering segment ϕ of $\langle w_a, w_b \rangle$, the following should be satisfied:*

$$lb_s^{\langle r(w_a), r(w_b) \rangle} \leq \sum_{p \in \phi} lb_i^p \leq \sum_{p \in \phi} ub_i^p \leq ub_s^{\langle r(w_a), r(w_b) \rangle}$$

In the above definition, $lb_s^{\langle a,b \rangle}$ and $ub_s^{\langle a,b \rangle}$ denote the lower and upper delay bounds for the transition $\langle a, b \rangle$ of M_S. lb_i and ub_i denote the lower and upper delay bounds for a transition in M_I which belongs to the stuttering segment ϕ of $\langle r(a), r(b) \rangle$. In defining the TWFS refinement between the implementation TTS (M_I) and specification TTS (M_S), it has been assumed that M_I is a WFS refinement of M_S that has already been established.

The notion of TWFS given above is bisimilar in nature, even though it is not defined in a symmetric manner. If the specification had a behavior that was not matched by the implementation, the implementation would not be a WFS refinement of the specification and hence would not be a TWFS of the specification. Note that for WFS refinement, an implementation state cannot be related to more than one specification state (as the refinement map is a function used to relate implementation states to specification states). So once WFS refinement has been established, it is not needed to check the other direction for TWFS.

WFS refinement is a compositional notion (see Section 2.4.1). A similar property for TWFS can be derived. Below, $M_c \simeq_r M_b$ denotes that M_c is a TWFS refinement of M_b using refinement map r; and $r; q$ denotes composition, i.e. $(r; q)(s) = q(r(s))$. Let $MM_I^{c \leftarrow b}$ denote the marked M_I of M_c w.r.t. M_b. For the following discussion, let $M_c \simeq_r M_b$ and $M_b \simeq_q M_a$.

Lemma 1. *If $\langle w, v\rangle$ is a non-stuttering transition of $MM_I^{c\leftarrow a}$, then $\langle w, v\rangle$ is a non-stuttering transition of $MM_I^{c\leftarrow b}$.*

Proof. If $\langle w, v\rangle$ is a non-stuttering transition of $MM_I^{c\leftarrow a}$, then $q(r(w)) \neq q(r(v))$. This implies that $r(w) \neq r(v)$, which implies that $\langle w, v\rangle$ is a non-stuttering transition of $MM_I^{c\leftarrow b}$. □

Lemma 2. *If $\langle w, v\rangle$ is a stuttering transition of $MM_I^{c\leftarrow a}$, then $\langle r(w), r(v)\rangle$ is a stuttering transition of $MM_I^{b\leftarrow a}$.*

Proof. We have that $M_b \simeq_q M_a$. Therefore, for $\langle r(w), r(v)\rangle$ to be a stuttering transition of $MM_I^{b\leftarrow a}$, we need $q(r(w)) = q(r(v))$. We have that $\langle w, v\rangle$ is a stuttering transition of $MM_I^{c\leftarrow a}$, which implies that $q(r(w)) = q(r(v))$. □

Lemma 3. *If ϕ is a stuttering segment of $MM_I^{c\leftarrow a}$, then ϕ can be partitioned into m ($m \geq 1$) segments $\phi_1, \phi_2, ..., \phi_m$, such that $\phi_1, ..., \phi_m$ are stuttering segments of $MM_I^{c\leftarrow b}$.*

Proof. From the definition of stuttering segments, we know that every stuttering segment ϕ is preceded by a non-stuttering transition (say t_p) and the last transition in ϕ is also a non-stuttering transition (say t_l). Since ϕ is a stuttering segment of $MM_I^{c\leftarrow a}$, from Lemma 1, we get that t_p and t_l are non-stuttering transitions of $MM_I^{c\leftarrow b}$. The other transitions of ϕ may or may not be non-stuttering transitions of $MM_I^{c\leftarrow b}$. If m of the transitions in ϕ are non-stuttering w.r.t. $MM_I^{c\leftarrow b}$, then these m non-stuttering transitions and the preceding stuttering transitions will result in m stuttering segments of $MM_I^{c\leftarrow b}$. □

Theorem 3. *(Composition for TWFS refinement) If $M_c \simeq_r M_b$ and $M_b \simeq_q M_a$ then $M_c \simeq_{r;q} M_a$.*

Proof. To show that $M_c \simeq_{r;q} M_a$, there are two conditions. First condition is that $M_c \sqsubseteq_{r;q} M_a$. Since $M_c \simeq_r M_b \rightarrow M_c \sqsubseteq_r M_b$ and $M_b \simeq_q M_a \rightarrow M_b \sqsubseteq_q M_a$, from Theorem 1 we get $M_c \sqsubseteq_{r;q} M_a$.

The proof of second condition is as follows (see Def. 12 for second condition). Without loss of generality, consider any non-stuttering transition of $MM_I^{c\leftarrow a}$ say $\langle w_a, w_b\rangle$ and the corresponding stuttering segment ϕ. Every stuttering segment is preceded by a non-stuttering transition (say $\langle w_c, w_d\rangle$). From Lemma 3, we have that ϕ can be partitioned into m stuttering segments of $MM_I^{c\leftarrow b}$: $\phi_1, ..., \phi_m$. Let $t_1, t_2, ..., t_m$ be the non-stuttering transitions of M_b corresponding to the m stuttering segments. Since $M_c \simeq_r M_b$, each of these m stuttering segments individually will

satisfy the timing requirements of the corresponding non-stuttering transitions of M_b. Therefore, ϕ will satisfy the sum of all the timing requirements of $t_1, t_2, ..., t_m$. Note that t_m is $\langle r(w_a), r(w_b)\rangle$. From Lemma 1, t_m:$\langle r(w_a), r(w_b)\rangle$ is a non-stuttering transition of $MM_I^{b\leftarrow a}$. The corresponding non-stuttering transition in M_a is $\langle q; r(w_a), q; r(w_b)\rangle$. Also, from Lemma 2, $t_1, t_2, ..., t_{m-1}$ are stuttering transitions of $MM_I^{b\leftarrow a}$ and $t_1, t_2, ..., t_{m-1}$ is preceded by the non-stuttering transition $\langle r(w_c), r(w_d)\rangle$ of $MM_I^{b\leftarrow a}$. Therefore, $t_1, t_2, ..., t_m$ is a stuttering segment of $MM_I^{b\leftarrow a}$. Since $M_b \simeq_q M_a$, $t_1, t_2, ..., t_m$ will satisfy the timing requirements of $\langle q; r(w_a), q; r(w_b)\rangle$. Therefore, stuttering segment ϕ of $MM_I^{c\leftarrow a}$ will satisfy timing requirements of $\langle q; r(w_a), q; r(w_b)\rangle$. Second condition is proved, we have: $M_c \simeq_{r;q} M_a$. □

TWFS refinement is based on the idea that M_I satisfies the timing requirements of its M_S, when in every case the delay between the previous time that M_I made progress w.r.t. M_S and the next time M_I makes progress w.r.t. M_S, matches the time delay required for M_S to make that progress. If there is no stuttering between the M_I and the M_S, every step of M_I should match a step of M_S, and the delay of an implementation step should "match" the delay of the corresponding specification step.

4.3.2. Timed Well-Founded Equivalence Bisimulation Refinement

Timed refinement can also be defined with respect to Well Founded Equivalence Bisimulation (WEB). In defining a timed refinement between an M_I and an M_S, it can be assumed that a refinement relationship already exists between the two. In practice, what this means is that verification of timed refinement is preceded by verification of WEB refinement. Since the refinement relationship has been established and a witness refinement map exists, stuttering and non-stuttering transitions of the implementation TTS M_I can be identified and captured in a Marked TTS (defined in Def. 9). The general theory of WEB refinement allows for stuttering to occur on the implementation side and the specification side. In practice, situations in which the specification stutters is rare. Therefore, an assumption can be made that the implementation can have stuttering steps. Hence a marked specification TTS MM_S corresponding to a specification TTS M_S, is one in which every transition is a non-stuttering transition and is marked with n_t. MM_I will satisfy the timing requirements of the corresponding MM_S, if every time the implementation makes progress (non-stuttering step), then the sum of the delay of the non-stuttering step and the delays of all

preceding stuttering steps matches the delay of the corresponding specification step. Note that there may be many paths in the implementation that lead to a specific non-stuttering step. All these paths individually should satisfy the timing requirements of the corresponding specification step. These finite paths as called as stuttering segments (defined in Def. 11).

Timed Well-Founded Equivalence Bisimulation (TWEB) is builds on WEB refinement and incorporates both functional and timing correctness. TWEB is defined below:

Definition 13. *M_I is a Timed Well-Founded Equivalence Bisimulation (TWEB) of M_S if:*

1. *M_I is a WEB refinement of M_S w.r.t. refinement map r.*

2. *Let MM_I be the marked TTS of M_I w.r.t. M_S. Then, for every non-stuttering transition of MM_I $\langle w_a, w_b \rangle$, and for every stuttering segment ϕ of $\langle w_a, w_b \rangle$, the following should be satisfied:*

$$lb_s^{\langle r(w_a), r(w_b) \rangle} \leq \sum_{p \in \phi} lb_i^p \leq \sum_{p \in \phi} ub_i^p \leq ub_s^{\langle r(w_a), r(w_b) \rangle}$$

In the above definition, $lb_s^{\langle a,b \rangle}$ and $ub_s^{\langle a,b \rangle}$ denote the lower and upper delay bounds for the transition $\langle a, b \rangle$ of M_S. lb_i and ub_i denote the lower and upper delay bounds for a transition in M_I which belongs to the stuttering segment ϕ of $\langle r(a), r(b) \rangle$.

The notion of TWEB given above is bisimilar in nature, even though it is not defined in a symmetric manner. If the specification had a behavior that was not matched by the implementation, the implementation would not be a WEB refinement of the specification and hence would not be a TWEB of the specification. Note that for WEB refinement, an implementation state cannot be related to more than one specification state (as the refinement map is a function used to relate implementation states to specification states). So once WEB refinement has been established, it is not needed to check the other direction for TWEB.

WEB refinement is a compositional notion (see Section 2.4.2). A similar property for TWEB is derived. Below, $M_c \trianglerighteq_r M_b$ denotes that M_c is a TWEB of M_b using refinement map r; and $r; q$ denotes composition, i.e. $(r; q)(s) = q(r(s))$. Let $MM_I^{c \leftarrow b}$ denote the marked M_I of M_c w.r.t. M_b. For the following discussion, let $M_c \trianglerighteq_r M_b$ and $M_b \trianglerighteq_q M_a$.

Lemma 4. *If $\langle w, v\rangle$ is a non-stuttering transition of $MM_I^{c\leftarrow a}$, then $\langle w, v\rangle$ is a non-stuttering transition of $MM_I^{c\leftarrow b}$.*

Proof. If $\langle w, v\rangle$ is a non-stuttering transition of $MM_I^{c\leftarrow a}$, then $q(r(w)) \neq q(r(v))$. This implies that $r(w) \neq r(v)$, which implies that $\langle w, v\rangle$ is a non-stuttering transition of $MM_I^{c\leftarrow b}$.

□

Lemma 5. *If $\langle w, v\rangle$ is a stuttering transition of $MM_I^{c\leftarrow a}$, then $\langle r(w), r(v)\rangle$ is a stuttering transition of $MM_I^{b\leftarrow a}$.*

Proof. We have that $M_b \trianglerighteq_q M_a$. Therefore, for $\langle r(w), r(v)\rangle$ to be a stuttering transition of $MM_I^{b\leftarrow a}$, we need $q(r(w)) = q(r(v))$. We have that $\langle w, v\rangle$ is a stuttering transition of $MM_I^{c\leftarrow a}$, which implies that $q(r(w)) = q(r(v))$. □

Lemma 6. *If ϕ is a stuttering segment of $MM_I^{c\leftarrow a}$, then ϕ can be partitioned into m ($m \geq 1$) segments $\phi_1, \phi_2, ..., \phi_m$, such that $\phi_1, ..., \phi_m$ are stuttering segments of $MM_I^{c\leftarrow b}$.*

Proof. From the definition of stuttering segments, we know that every stuttering segment ϕ is preceded by a non-stuttering transition (say t_p) and the last transition in ϕ is also a non-stuttering transition (say t_l). Since ϕ is a stuttering segment of $MM_I^{c\leftarrow a}$, from Lemma 4, we get that t_p and t_l are non-stuttering transitions of $MM_I^{c\leftarrow b}$. The other transitions of ϕ may or may not be non-stuttering transitions of $MM_I^{c\leftarrow b}$. If m of the transitions in ϕ are non-stuttering w.r.t. $MM_I^{c\leftarrow b}$, then these m non-stuttering transitions and the preceding stuttering transitions will result in m stuttering segments of $MM_I^{c\leftarrow b}$. □

Theorem 4. *(Composition for TWEB) If $M_c \trianglerighteq_r M_b$ and $M_b \trianglerighteq_q M_a$ then $M_c \trianglerighteq_{r;q} M_a$.*

Proof. To show that $M_c \trianglerighteq_{r;q} M_a$, there are two conditions. First condition is that $M_c \approx_{r;q} M_a$. Since $M_c \trianglerighteq_r M_b \rightarrow M_c \approx_r M_b$ and $M_b \trianglerighteq_q M_a \rightarrow M_b \approx_q M_a$, from Theorem 2 we get $M_c \approx_{r;q} M_a$.

Proof of second condition (see Def. 13 for second condition): Without loss of generality, consider any non-stuttering transition of $MM_I^{c\leftarrow a}$ say $\langle w_a, w_b\rangle$ and the corresponding stuttering segment ϕ. Every stuttering segment is preceded by a non-stuttering transition (say $\langle w_c, w_d\rangle$). From Lemma 5, we have that ϕ can be partitioned into m stuttering segments of $MM_I^{c\leftarrow b}$: $\phi_1, ..., \phi_m$.

Let $t_1, t_2, ..., t_m$ be the non-stuttering transitions of M_b corresponding to the m stuttering segments. Since $M_c \trianglerighteq_r M_b$, each of these m stuttering segments individually will satisfy the timing requirements of the corresponding non-stuttering transitions of M_b. Therefore, ϕ will satisfy the sum of all the timing requirements of $t_1, t_2, ..., t_m$. Note that t_m is $\langle r(w_a), r(w_b)\rangle$. From Lemma 4, t_m:$\langle r(w_a), r(w_b)\rangle$ is a non-stuttering transition of $MM_I^{b\leftarrow a}$. The corresponding non-stuttering transition in M_a is $\langle q; r(w_a), q; r(w_b)\rangle$. Also, from Lemma 5, $t_1, t_2, ..., t_{m-1}$ are stuttering transitions of $MM_I^{b\leftarrow a}$ and $t_1, t_2, ..., t_{m-1}$ is preceded by the non-stuttering transition $\langle r(w_c), r(w_d)\rangle$ of $MM_I^{b\leftarrow a}$. Therefore, $t_1, t_2, ..., t_m$ is a stuttering segment of $MM_I^{b\leftarrow a}$. Since $M_b \trianglerighteq_q M_a$, $t_1, t_2, ..., t_m$ will satisfy the timing requirements of $\langle q; r(w_a), q; r(w_b)\rangle$. Therefore, stuttering segment ϕ of $MM_I^{c\leftarrow a}$ will satisfy timing requirements of $\langle q; r(w_a), q; r(w_b)\rangle$. Second condition is proved, we have: $M_c \trianglerighteq_{r;q} M_a$. □

4.4. Checking Timed Refinement

Timed refinement (TWFS or TWEB depending on how the functional verification is performed) verification is performed in three steps. The first step is to verify that the implementation TTS (M_I) is a WFS refinement or WEB refinement of the specification TTS (M_S). The second step is to compute Marked M_S (MM_S) and Marked M_I (MM_I) using the information from the WFS or WEB refinement proof. The third step is to discharge the remaining proof obligations of timed refinement (TWFS or TWEB), which is to compute all the stuttering segments of MM_I and check that the stuttering segments satisfy the timing requirements of MM_S.

In this section, the proof obligations for the WEB refinement verification were generated manually and discharged using a decision procedure. The details are described in Section 4.5. The second step is straightforward, which is to identify the non-stuttering and stuttering transitions of M_I and mark them as such to get MM_I. However, when computing MM_I, the implementation TTS has also been abstracted. A brief overview of this abstraction is provided in Section 4.5. A detailed description of abstraction is provided in Chapter 5. The abstraction is required as otherwise, the number of states and transitions of the implementation TTS will explode.

A procedure has been developed for the third step (given in Algorithm 1), which checks the remaining timed refinement proof obligations. The input to the timed refinement verification procedure is a list of transitions of MM_I, a list of transitions MM_S, and the refinement map used for the WEB refinement proof (which is a list of implementation states and the specification states

Algorithm 1 Procedure For Checking Timed refinement

```
1: procedure CheckTimedRef(MM_I, MM_S, r)
2:     for all t : ⟨w, v, lb, ub, m⟩ ∈ R_I do
3:         if m = n_t then
4:             sseg-list[0] ← t;
5:             sseg-set ← {⟨FALSE, 0, sseg-list⟩};
6:             repeat
7:                 termination-condition ← TRUE; sseg-set' ← sseg-set;
8:                 for all sseg:⟨sseg-complete, sseg-length, sseg-list⟩ ∈ sseg-set' do
9:                     if ¬sseg-complete then
10:                        t^p : ⟨w^p, v^p, lb^p, ub^p, m^p⟩ ← sseg-list[sseg-length];
11:                        sseg-set ← sseg-set/sseg;
12:                        for all t^q : ⟨w^q, w^p, lb^q, ub^q, m^q⟩ ∈ R_I do
13:                            if m^q = n_t then
14:                                sseg-set ← sseg-set ∪{⟨TRUE, sseg-length, sseg-list⟩};
15:                            else
16:                                sseg-list[sseg-length + 1] ← t^q;
17:                                sseg-set ← sseg-set ∪{⟨FALSE, sseg-length+1, sseg-list⟩};
18:                                termination-condition ← FALSE;
19:            until termination-condition
20:            for all ⟨sseg-complete, sseg-length, sseg-list⟩ ∈ sseg-set do
21:                for all ⟨w^s, v^s, lb^s, ub^s, m^s⟩ ∈ R_{MM_S} do
22:                    if (w^s = r(w)) & (v^s = r(v)) then
23:                        if ¬(lb^s ≤ Σ_{i=0}^{sseg−length} lb^i ≤ Σ_{i=0}^{sseg−length} ub^i ≤ ub^s) then
24:                            return sseg-list;
```

they map to). Each transition will include information about the delay of the transition (lower bound and upper bound), and whether the transition is a stuttering or non-stuttering transition.

The procedure iterates through the non-stuttering transitions of MM_I (lines 2 and 3). The procedure computes all the stuttering segments for each non-stuttering transition. sseg-set is the set of all stuttering segments corresponding to transition t. A stuttering segment is recorded in the list of transitions sseg-list. The stuttering segments are stored in sseg-set as a three tuple: ⟨sseg-complete, sseg-length, sseg-list⟩. sseg-complete is a flag that keeps track of whether the computation of the stuttering segment is complete. sseg-length keeps track of the length of the stuttering segment as it is computed. Lines 7-18 are repeated until all the stuttering segments are computed. During the procedure, sseg-set stores the partially computed stuttering segments.

The procedure then iterates through the partially computed stuttering segments (lines 8-9). For each partially computed stuttering segment, the procedure looks at all incoming transitions to

the tail of the segment (t^p) in line 12. If there are n incoming transitions, the partially computed stuttering segment will split into n partially computed stuttering segments. Thus the partially computed stuttering segment is removed from sseg-set (line 11). If the incoming transition is a non-stuttering transition, then the stuttering segment is complete as it is (line 14). Then sseg-complete is set to TRUE and the stuttering segment is added to sseg-set. If the incoming transition is a stuttering transition, the transition is added to the tail of the partially computed stuttering segment and added to the sseg-set (lines 16-18).

As an example, the stuttering segments of $\langle 10, 13\rangle$ (in Figure 4.2) are $\{\langle 10, 13\rangle, \langle 9, 10\rangle, \langle 8, 9\rangle, \langle 4, 8\rangle, \langle 3, 4\rangle\}$ and $\{\langle 10, 13\rangle, \langle 9, 10\rangle, \langle 7, 9\rangle, \langle 6, 7\rangle, \langle 5, 6\rangle, \langle 4, 5\rangle, \langle 3, 4\rangle\}$. The procedure then computes the sum of the lower time delays and upper time delays for each of the stuttering segments in sseg-set. Based on the refinement map, every non-stuttering transition of the implementation maps to a transition of the specification. The procedure then checks that the total of the lower time delays and the total of the upper time delays for each stuttering segment of a non-stuttering transition lie within the lower bound delay and the upper bound delay of the corresponding specification transition (lines 21-22). For example, if the refinement map maps implementation state 10 to specification state S_1 and implementation state 13 to specification state S_2, then the non-stuttering transition $\langle 10, 13\rangle$ maps to the specification transition $\langle S_1, S_2\rangle$. The procedure will check that total delays of every stuttering segment of $\langle 10, 13\rangle$ lie within the delay bounds of $\langle S_1, S_2\rangle$. The stuttering segments that violate this requirement are counter examples. The procedure will output these stuttering segments (line 24). If no violations are found, timed refinement is verified. Next, completeness of the procedure is shown.

Lemma 7. *For all stuttering transitions $\langle w, v\rangle$ of MM_I, there exists a function rank : $S_I \to \mathbb{N}$ such that $rank(v) < rank(w)$ iff MM_I does not have stuttering cycles.*

Proof. $\Rightarrow$: This is proved by contradiction. Consider that MM_I has a stuttering cycle. Then, it is not possible to assign a natural number value to every state in the cycle such that the value decreases for every transition in the cycle. Therefore, there will be at least one transition in the stuttering cycle for which $rank(v) < rank(w)$ is not satisfied.

$\Leftarrow$: Consider the directed graph corresponding to MM_I, where states are the vertices and transitions are the directed edges. Now, remove all the non-stuttering transitions from R_I. Since

there are no cycles of stuttering transitions, the resulting graph should be a set of directed acyclic graphs (DAGs). Natural number values can now be assigned to all the nodes in each DAG such that the value decreases for every transition. This assignment of values is a witness *rank* function that satisfies $rank(v) < rank(w)$ for all stuttering transitions. □

Theorem 5. (*Completeness for TWFS) If* $M_I \approx_r M_S$*, then Procedure CheckTimedRef will complete for inputs* MM_I*,* MM_S*, and* r*.*

Proof. If $M_I \sqsubseteq_r M_S$, then from the definition of WFS refinement and the definition of Marked M_I (MM_I), there exists a function $rank : S_I \to \mathbb{N}$ such that $rank(v) < rank(w)$ for all stuttering transitions of MM_I. From Lemma 7, MM_I has no stuttering cycles. The repeat loop (lines 6-19) terminates only when all partially computed stuttering segments eventually hit a non-stuttering transition when tracing backward in R_I. Since R_I has no stuttering cycles and is left total, Procedure CheckTimedRef will complete. □

Theorem 6. (*Completeness for TWEB) If* $M_I \approx_r M_S$*, then Procedure CheckTimedRef will complete for inputs* MM_I*,* MM_S*, and* r*.*

Proof. If $M_I \approx_r M_S$, then from the definition of WEB refinement and the definition of Marked M_I (MM_I), there exists a function $rank : S_I \to \mathbb{N}$ such that $rank(v) < rank(w)$ for all stuttering transitions of MM_I. From Lemma 7, MM_I has no stuttering cycles. The repeat loop (lines 6-19) terminates only when all partially computed stuttering segments eventually hit a non-stuttering transition when tracing backward in R_I. Since R_I has no stuttering cycles and is left total, Procedure CheckTimedRef will complete. □

Theorem 7. *The time complexity of the CheckTimedRef procedure is* $\mathcal{O}(|R_I|^3)$*.*

Proof. Let n_i, n_s, n_{ss}, and n_{i-ns}, be the number of transitions of MM_I, number of transitions of MM_S, number of stuttering segments of MM_I, and number of non-stuttering transitions of MM_I. The outer for loop starting in line 2 has n_i passes. If initially the transitions of MM_I were classified as stuttering and non-stuttering transitions, the outer loop starting in line 2 can be reduced to n_{i-ns} passes. The initial classification would add an n_i to the running time.

Each run of the repeat loop (line 6) increases the length of the partially computed stuttering segments by 1. Therefore, the repeat loop has as many passes as the maximum length of all

stuttering segments of MM_I (max(ss-length)). The deletion in line 10 is (max(ss-length)). The for loop in line 8 has n_{ss} passes and the for loop in line 12 executes n_i times. So the running time of lines 6-19 is max(ss-length)$n_{ss}n_i$.

The loop starting in line 20 executes n_{ss} times and the inner loop in line 21 executes n_s times. Therefore, the complexity of lines 20-24 is $n_{ss}n_s$. Therefore, taking into consideration that the outer loop in line 2 executes n_{i-ns} times and adding up all the components we get the complexity of the procedure to be: $n_i + n_{i-ns}n_{ss}n_i$max(ss-length) + $n_{i-ns}n_{ss}n_s$. Since we consider only stuttering on the implementation side, the number of transitions of the implementation $n_i >$ the number of transitions of the specification n_s. Therefore, the complexity of the procedure reduces to $n_{i-ns}n_{ss}n_i$max(ss-length). In the worst case, if all transitions are non-stuttering transitions, $n_{i-ns} = n_i$. Also, there would not be any stuttering transitions. Therefore, max(ss-length) = 1. Also, number of stuttering segments would be equal to the number of transitions: $n_{ss} = n_i$. $n_i = |R_I|$, therefore, the time complexity of the CheckTimedRef procedure is $\mathcal{O}(|R_I|^3)$. □

4.5. Case Studies and Results

A stepper motor with 4 leads can be stepped in two different ways based on how the leads are energized. When the following four values are applied in a repeating sequence to the leads: ⟨0001⟩, ⟨0010⟩, ⟨0100⟩, ⟨1000⟩, ⟨0001⟩, ..., it is known as *full stepping*. Instead if the following eight values are applied in a repeating sequence to the leads: ⟨0001⟩, ⟨0011⟩, ⟨0010⟩, ⟨0110⟩, ⟨0100⟩, ⟨1100⟩, ⟨1000⟩, ⟨1001⟩, ⟨0001⟩, ..., it is known as *half stepping*.

A stepper motor can be controlled by a micro-controller. The ARM Cortex-M3 based NXP LPC1768 [58] micro-controller has been used for stepper motor control. Four pins from PORT 2 of the LPC1768 are connected to the stepper motor leads via an electronic circuit. The value of these 4 pins are determined by bits 28-31 of the FIOPIN register.

***Case Study 1–Interrupt Driven Full Stepping Stepper Motor Control*:** Stepper motor control with full stepping is implemented with the Repetitive Interrupt Timer (RIT), which is a timer present in the LPC1768. The controller microcode enables the RIT unit and also a register that RIT has to store a constant value. Then the code enters a while loop. The RIT has a counter which increments every clock cycle. When the counter reaches the value stored in the RIT register, an interrupt is generated. As soon as the interrupt is generated, the counter is

reset to 0 and flow of control changes to the RIT interrupt service routine (ISR). In the RIT ISR, the FIOPIN register is updated to the next value of the leads required for full stepping, and then returns control to the main program. The RIT constant register value is initialized such that the delays between consecutive interrupts generated by the RIT matches the delay required between full stepping control states. Also, this delay determines the speed at which the motor runs.

***Case Study 2–Interrupt Driven Half Stepping Stepper Motor Control*:** The control is similar to the mechanism used for case study 1. The RIT unit is employed here also. The ISR is modified to update the FIOPIN register based on half stepping control instead of full stepping control.

***Case Study 3–Full Stepping Stepper Motor Control without Interrupts*:** For this case study, full stepping control is implemented without using interrupts. The delay required between full stepping control states is achieved using for loops with a large number of iterations. The number of iterations of the for loop is determined so that the time required by the microcontroller to execute the for loop matches the delay required between full stepping control states. The drawback with this approach is that if the control program performs other functions and has enabled other interrupts, it may not be possible to guarantee accurate speed of the motor.

***Case Study 4–Half Stepping Stepper Motor Control without Interrupts*:** For this case study, the delay required between half stepping control states is achieved using for loops with a large number of iterations, instead of interrupts.

***Case Study 5–Interrupt Driven Variable Speed Full Stepping Stepper Motor Control*:** The RIT unit is used to implement full stepping control. However, the motor has 3 speed modes and in each mode the motor runs at a different speed. The modes can be changed based on input from a keyboard, which acts as an external interrupt. When any key on the keyboard is pressed, an interrupt is generated (different from the RIT interrupt). The keyboard input is processed to change the speed of the motor. Note that since the control program supports 2 interrupts, the RIT interrupt is given the higher priority.

***Case Study 6–Interrupt Driven Variable Half Full Stepping Stepper Motor Control*:** 3 mode variable speed is implemented using half stepping. The input from keyboard (which acts as an external interrupt) and RIT interrupt are also employed here, with the RIT interrupt having higher priority.

WEB Refinement Verification: For all the six case studies, WEB refinement verification was performed using the Bit-level Analysis Tool (BAT) [47, 48], which is a decision procedure for the theory of bit-vectors with arrays. Note that some of the proof obligations were encoded in SMT-LIB v2 language [6] and discharged using the z3 SMT solver [21]. For the WEB refinement verification, the specification TTS and the implementation TTS were encoded in the input language of the BAT tool. For the WEB refinement proof, timing information is not required and was not included in the descriptions of the implementation and specification TTS. The implementation TTS consisted of the *instruction functions* (see Section 2.3) and the initial state of the micro-controller registers and memory. The next step is to construct a refinement map, which is the function that maps implementation states to specification states. The refinement map for the case studies is the function that extracts bits 28-31 of the FIOPIN register, as these 4 bits are connected to the leads of the stepper motor and directly determine the state of the stepper motor.

Each instruction function corresponds to one or more transitions of the implementation TTS. It has been verified that each of the instruction functions satisfies the WEB refinement correctness formula (see Section 2.4.2). The proof obligations were encoded in the BAT language and checked using the BAT decision procedure. Many instructions corresponded to more than one transition. In most cases, all the transitions corresponding to an instruction were similar and could be verified together using symbolic states and symbolic simulation. For some instructions, there is more than one case. An example of this is instructions whose execution could be altered by the RIT or other interrupts. For such situations, we handled the cases separately. There were two verification obligations, one for the case where the interrupt occurs and one for the case where the interrupt does not occur. Note that the non-stuttering transitions corresponded to only those instructions that updated the FIOPIN register. All other instructions corresponded to stuttering transitions. For the proof, pre-conditions and post-conditions were used to propagate the required hypothesis for each of the proof obligations.

Timed Refinement Verification: The CheckTimedRef procedure (see Section 4.4) was implemented as a tool. The tool takes as input, transitions of abstracted MM_I, transitions of MM_S, and the refinement map. The marked implementation TTS of a real-time control program will have a very large number of states and transitions. Hence, this marked TTS cannot be input directly to the timed refinement verification procedure. We use a number of techniques to abstract MM_I.

The abstractions are based on the control flow graph of the object code program. Sets of states and sets of transitions corresponding to an instruction are abstracted as symbolic states and symbolic transitions. A basic block consisting of a sequence of stuttering transitions is abstracted as one stuttering transition with a delay which is the sum of the delays of the transitions in the sequence. Loops consisting of only stuttering transitions are replaced with one transition that mimics the delay of the loop. Use of these abstractions resulted in tractable and efficient verification of the stepper motor microcode control case studies.

Table 4.1. Verification Statistics for TWEB

Case Study	Model Size	Proof Size	Refinement Verif. Time [sec]	# of Transitions of MM_I	# of Transitions of Abstract MM_I
1	2,173	10,171	4.30	2.5 million	83
2	3,232	16,018	7.23	4.5 million	135
3	1,151	6,606	2.47	45 million	103
4	1,989	11,861	4.10	81 million	184
5	3,519	17,556	9.73	17.5 million	276
6	5,625	27,854	16.13	32 million	430

Table 4.1 shows the verification statistics for the 6 case studies. The verification experiments were performed on a Intel(R) Celeron(R) CPU 540 1.86GHz processor with an L2 cache of 1MB. The "Model Size" column gives the number of lines of the implementation model in the input language of the BAT tool. The "Proof Size" column gives the number of lines of BAT code for all the WEB refinement proof obligations (includes the implementation model and the specification model). The refinement verification time column gives the total time required to discharge all the WEB and timed refinement proof obligations. ***The table also gives the approximate number of transitions of the implementation TTS*** MM_I ***and the number of transitions of the abstracted*** MM_I***. As can be seen from the table, abstractions based on stuttering transitions and stuttering segments significantly reduce the number of transitions of the implementation TTS making object code verification feasible.*** Several bugs both functional and timing were found. Below one functional bug (found during WEB refinement verification) and

one timing bug (found during timed refinement verification) are described. *Any existing tools that can check timed refinement were not found. Approaches to encode timed refinement verification (when stuttering and refinement maps are involved) in UPPAAL/KRONOS like tools were also not found.* Therefore, results comparing our approach with other tools are not available.

Functional Bug: The bug was found for case study 1. Bits 28-31 of the 32-bit FIOPIN register control the motor leads. Other bits of FIOPIN can be used for other purposes and should not be updated. The FIOPIN register can only be updated as a whole and individual bits cannot be updated. Therefore, the FIOPIN is updated by using OR masking to set bits to '1' and AND masking to reset bits to '0'. This was accomplished by first performing the OR mask on the FIOPIN register and then the AND mask. Therefore, the motor state was transitioning from state 0001 to 0011, and then to 0010. This is incorrect as 0011 is not a correct state of the motor when full stepping. The bug was found during WEB refinement verification. If an external interrupt had occurred between the OR mask and the AND mask, then the motor would be stuck in a bad state.

Timing Bug: For both case studies 5 and 6, when switching from a lower speed to a higher speed, it is required that the transition take place smoothly with an upper bound for the delay of the transition as the delay for state transitions of the lower speed. However, this timing requirement was not satisfied by the object code in certain states. Specifically, if the value of the RIT counter was close to the compare value when the external interrupt occurred (forcing a change in speed), there was not enough time to update the counter value and the compare value, and still make the transition to the higher speed mode in time.

This chapter has presented timed refinement for real-time interrupt-drive object code programs. Timed refinement is defined as timed well-founded simulation (TWFS) refinement and timed well-founded equivalence bisimulation (TWEB) refinement, depending on which type of functional verification has been adopted. Next, a detailed description of stuttering abstraction will been discussed and how it can be automated for real-time interrupt-drive object code is also presented.

5. STUTTERING ABSTRACTION

Verification of device-object code (code or the set of instructions that is executed by microcontrollers embedded in the device) is very challenging because object code is very low-level, real-time, interrupt driven, and extensively exhibits the phenomenon of stuttering. Informally, stuttering means that the device executes millions of microprocessor instructions to account for only a single progression of a higher-level requirements/specification model. A novel abstraction technique that exploits the phenomenon of stuttering called stuttering abstraction (SA) was introduced in Chapter 4. This abstraction technique has been extended to timed transition systems (TTSs) called timed stuttering abstraction (TSA). This chapter extends and develops the idea of TSA. This chapter's contribution include: (1) a formalization of TSA; (2) correctness of TSA in the context of TWFS refinement; and (3) dynamic timed stuttering abstraction, an algorithm to automatically apply TSA during symbolic simulation of the object code.

5.1. Stuttering Abstraction for TS

The implementation model for real-time interrupt-driven object code consists of million of transitions. A large portion of these transitions are usually stuttering transitions (Def. 6). In Figure 2.6, it can be noticed that many paths in the implementation lead to a specific non-stuttering step. All these finite paths are termed as stuttering segments. *Stuttering segment* ϕ of $\langle w, v\rangle$ [26] where $\langle w, v\rangle$ is a non-stuttering transition can be described as a sequence of transitions in which $\langle w, v\rangle$ is preceded by zero to many stuttering transition(s) and another non-stuttering transition. The least length of a stuttering segment is one. This occurs when a non-stuttering step is preceded by another non-stuttering step. The stuttering segment then only consists of one transition which is the non-stuttering step. Also, a non-stuttering step can have many stuttering segments. For the TS shown in Figure 2.6, the stuttering segments of $\langle 10, 13\rangle$ are:

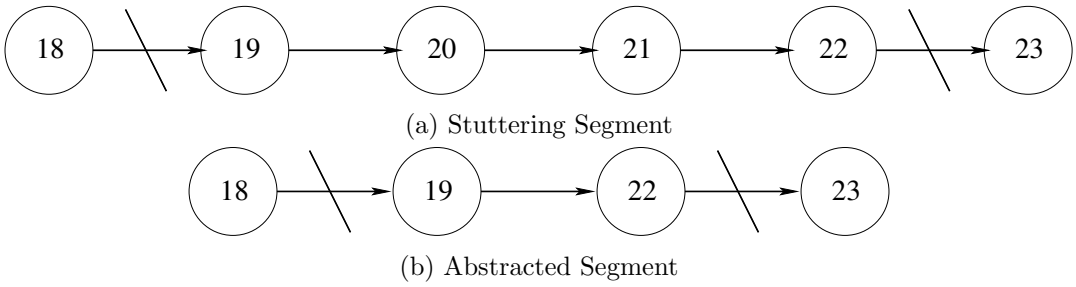
1. $\{\langle 2, 3\rangle, \langle 3, 4\rangle, \langle 4, 5\rangle, \langle 5, 6\rangle, \langle 6, 7\rangle, \langle 7, 9\rangle, \langle 9, 10\rangle, \langle 10, 13\rangle\}$
2. $\{\langle 2, 3\rangle, \langle 3, 4\rangle, \langle 4, 8\rangle, \langle 8, 9\rangle, \langle 9, 10\rangle, \langle 10, 13\rangle\}$

Object code contains millions of transitions, hence applying suitable abstraction techniques on the stuttering segments help to deal with path explosion problem. This reduces the verification

problem to analysis of stuttering segments. Abstraction techniques can be applied on these segments based on its control flow graph. Sets of states and sets of transition corresponding to these stuttering segments can be symbolic states and symbolic transitions. A basic block consisting of a sequence of stuttering transitions in a stuttering segment is abstracted as one stuttering transition. Loops consisting of only stuttering transitions can be replaced with one transition. First stuttering abstraction (SA) for transition systems (TS) is discussed. Below is the definition of stuttering abstraction:

Definition 14. *M is a TS which has a sequence of two stuttering transitions, $\langle w_1, w_2\rangle$ and $\langle w_2, w_3\rangle$ such that w_2 has only one incoming transition and only one outgoing transition, then the stuttering abstracted TS M^a of M is constructed by replacing $\langle w_1, w_2\rangle$ and $\langle w_2, w_3\rangle$ with $\langle w_1, w_3\rangle$*

Note that the definition above merges two transitions to one. Repeated application of the above definition can be used to merge large sequences of stuttering transitions. Consider a sequence shown in Figure 5.1a which is part of a stuttering segment. The idea with SA is that a finite sequence of stuttering transitions can be merged into one transition, while still preserving the functional behaviors of the original transition system. The sequence shown in Figure 5.1b is obtained by applying SA to the sequence in Figure 5.1a. More specifically, stuttering transitions $\langle 19, 20\rangle$, $\langle 20, 21\rangle$ and $\langle 21, 22\rangle$ are merged into one abstract transition $\langle 19, 22\rangle$. Since the object code TS typically has millions of stuttering transitions, SA will provide drastic reductions in the size of the TS, and will therefore significantly improve efficiency and scalability of verification.



(a) Stuttering Segment

(b) Abstracted Segment

Figure 5.1. Basic Block in Abstraction

A condition (to ensure that the abstraction is sound) is imposed on when stuttering transitions can be merged. When applying SA, intermediate states are removed. In the abstraction

example above, state 20 and 21 are removed. The condition is that states that are removed (or abstracted away) can only have one incoming transition and one outgoing transition. From Figure 5.1a, it can be seen that states 20 and 21 satisfy this requirement. Another example is as follows. In Figure 2.6, $\langle 3,4\rangle$, $\langle 4,5\rangle$, $\langle 5,6\rangle$, $\langle 6,7\rangle$, $\langle 7,9\rangle$, and $\langle 9,10\rangle$ form a sequence of stuttering transitions. Using SA, it would be ideal to merge all these stuttering transitions into one abstracted transition $\langle 3,10\rangle$. However, state 4 has multiple outgoing transitions and state 9 has multiple incoming transitions. Therefore, both states 4 and 9 cannot be abstracted. Thus, using SA, we can reduce the above stuttering sequence to $\langle 3,4\rangle$, $\langle 4,9\rangle$, and $\langle 9,10\rangle$. Hence, when applying stuttering abstraction, initial states are not removed.

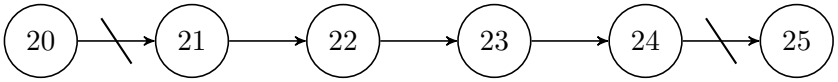
5.2. Timed Stuttering Abstraction (TSA) for TTS

In this dissertation, the implementation model is the real-time interrupt-driven object code. The implementation timed transition system (TTS) consists of *instruction functions* and the initial state of the micro-controller registers and memory. Each instruction function corresponds to one or more transitions of the implementation TTS. Hence, the size of the implementation TTS is in the range of millions. Manually developing proof methodology for this large range of transitions may introduce human errors. This section basically describes how abstraction can be applied on the stuttering transitions for timed transition systems. Timed stuttering abstraction (TSA) is described in terms of timed refinement.

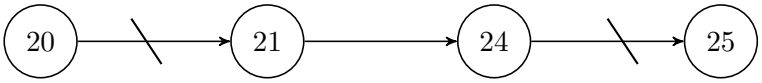
5.2.1. Timed Stuttering Abstraction (TSA)

The complexity of object code behavior presents itself in the size of the resulting TTS. For example, for the stepper motor case studies, the number of transitions of the object code TTS is in the order of millions. TSA is targeted at addressing this complexity. The sequence shown in Figure 5.2a is a stuttering segment of the M_I from Figure 4.2. The idea with TSA is that a finite sequence of stuttering transitions can be merged into one transition, while still preserving the functional and timing behaviours of the original M_I. The sequence shown in Figure 5.2b is obtained by applying TSA to the sequence in Figure 5.2a. More specifically, stuttering transitions $\langle 19,20\rangle$, $\langle 20,21\rangle$ and $\langle 21,22\rangle$ are merged into one abstract transition $\langle 19,22\rangle$. The upper bound and lower bound of $\langle 19,22\rangle$ is the sum of the upper bounds and lower bounds of the set of transitions abstracted, respectively. Since the object code TTS typically has millions of stuttering transitions,

TSA will provide drastic reductions in the size of this TTS, and will therefore significantly improve efficiency and scalability of verification.



(a) Stuttering Segment of TTS



(b) Abstracted Segment TTS

Figure 5.2. Basic Block in Abstraction for a TTS in Figure 4.2

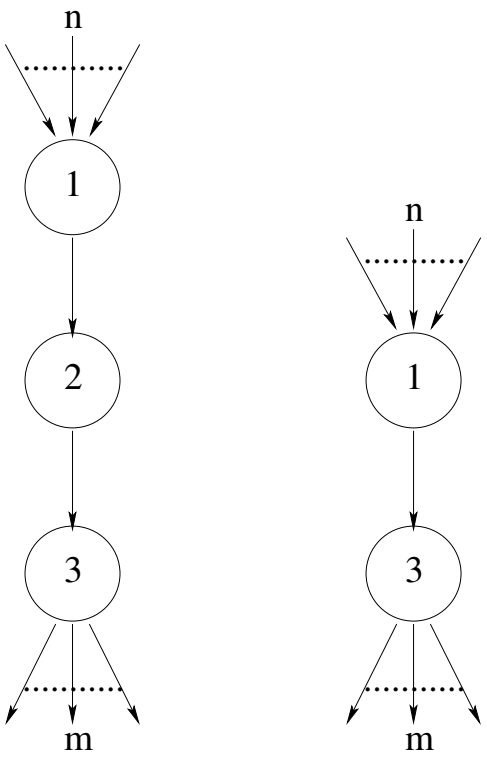
A condition (to ensure that the abstraction is sound) is imposed on when stuttering transitions can be merged. When applying TSA, intermediate states are removed. In the abstraction example above, state 22 and 23 are removed. The condition is that states that are removed (or abstracted away) can only have one incoming transition and one outgoing transition. From Figure 4.2, it can be seen that states 22 and 23 satisfy this requirement. Another example is as follows. In Figure 4.2, $\langle 3, 4\rangle$, $\langle 4, 5\rangle$, $\langle 5, 6\rangle$, $\langle 6, 7\rangle$, $\langle 7, 9\rangle$, and $\langle 9, 10\rangle$ form a sequence of stuttering transitions. Using TSA, it would be ideal to merge all these stuttering transitions into one abstracted transition $\langle 3, 10\rangle$. However, state 4 has multiple outgoing transitions and state 9 has multiple incoming transitions. Therefore, both states 4 and 9 cannot be abstracted. Thus, using TSA, we can reduce the above stuttering sequence to $\langle 3, 4\rangle$, $\langle 4, 9\rangle$, and $\langle 9, 10\rangle$. The formal definition of TSA is given below.

Definition 15. *M is a TTS which has a sequence of two stuttering transitions, $\langle w_1, w_2, lb_{12}, ub_{12}\rangle$ and $\langle w_2, w_3, lb_{23}, ub_{23}\rangle$ such that w_2 has only one incoming transition and only one outgoing transition, then the timed stuttering abstracted TTS M^a of M is constructed by replacing $\langle w_1, w_2, lb_{12}, ub_{12}\rangle$ and $\langle w_2, w_3, lb_{23}, ub_{23}\rangle$ with $\langle w_1, w_3, lb_{13}, ub_{13}\rangle$ in M, where $lb_{13} = lb_{12} + lb_{23}$ and $ub_{13} = ub_{12} + ub_{23}$*

Note that the definition above merges two transitions to one. Repeated application of the above definition can be used to merge large sequences of stuttering transitions.

5.2.2. Correctness of TSA

Timed well-founded simulation (TWFS) refinement and timed well-founded equivalence bisimulation (TWEB) refinement have compositional properties (Theorem 3, 4). Correctness of TSA is established next.



(a) M (b) M^a

Figure 5.3. Abstraction on One Stuttering Transition

Lemma 8. *If M is a TTS and M^a is a TSA (Def. 15) TTS of M, then $M \simeq_r M^a$.*

Proof. Consider Figure 5.3 in which the implementation TTS (M) is represented in Figure 5.3a and specification TTS (M^a) is given in Figure 5.3b. Condition 1 of TWFS refinement (Def. 12) requires that $M \sqsubseteq_r M^a$. Using the refinement-map (r), L(state 1 of M) = L(state 1 of M^a), where L is the labeling function. Each node of M can be assigned a natural number value such that the value decreases for each transition. This assignment of values is a witness rank function for stuttering transitions. From the refinement-map state 2 of M matches state 1 of M^a and rank(2)

<rank(1). State 3 of M matches to state 3 of M^a. Hence we have $M \sqsubseteq_r M^a$ (from the definition of WFS in [45]).

For the stuttering segments, if lb and ub represent the lower and upper bounds respectively, $lb_{13-of-M^a} \leq (lb_{12} + lb_{23}) \leq (ub_{12} + ub_{23}) \leq ub_{13-of-M^a}$. Thus, satisfying condition 2 of Def. 12. Since both the conditions of TWFS refinement have been satisfied, $M \simeq_r M^a$ □

Theorem 8. *(Correctness for TSA) Let M^{a^n} be the abstracted implementation TTS, obtained by applying TSA (Def. 15) on M_I n times, where n is a positive integer. Then $M^{a^n} \simeq_r M_S \rightarrow M_I \simeq_r M_S$*

Proof. A series of abstracted TTS M^{a^1}, M^{a^2}, ..., M^{a^n} can be constructed, where M^{a^1} is obtained by abstracting one stuttering transition from M_I, M^{a^2} is obtained by abstracting one stuttering transition from M^{a^1}, and so on until we get to M^{a^n}. From Lemma 8, we get that $M_I \simeq_r M^{a^1}$, $M^{a^1} \simeq_r M^{a^2}$, ..., $M^{a^{n-1}} \simeq_r M^{a^n}$. The hypothesis (as given above in the implication to be proved) is that $M^{a^n} \simeq_r M_S$. Theorem **??** can be repeatedly applied to string all the above TWFS refinement relations together to get that $M_I \simeq_r M_S$. □

5.2.3. Procedure for Dynamic TSA

Algorithm 2 presents a procedure that performs symbolic simulation of the object code to compute the TTS. The algorithm also performs dynamic TSA as it unfolds the object code TTS. The input to the procedure is the object code, the initial state of object code (w_0), a set of states that should not be abstracted (non-abs-states) and the refinement-map (r). non-abs-states contains a list of states that have multiple incoming transitions (for example, state 3, 9, 13, 20, etc in Figure 4.2). The non-abs-states are identified by their program counter values. They should be determined statically and they are typically states that correspond to targets of branch instructions. The procedure outputs the abstracted implementation TTS which is contained in R_I. The current state being processed is represented by 'w' which is initialized to w_0. 'v' represents the successor state and is initialized to null. The list of successors for a state are stored in R_C. The procedure explores a single path/trace at a time, hence R_U contains the successor states that have not been visited. R_I, R_C and R_U are of the form $\langle w, v \rangle$ and are initialized to null. For example, consider state 4 from Figure 4.2, the successor states are 5, 8 and 11. R_C will have the values: $\langle 4, 5 \rangle$, $\langle 4, 8 \rangle$ and $\langle 4, 11 \rangle$. If $\langle 4, 5 \rangle$ is the path that is being computed then R_U gets the values

Algorithm 2 Procedure for Symbolic Simulation and Dynamic Timed Stuttering Abstraction

```
1: procedure SymSimulation(w_0, object-code, non-abs-states, r)
2:     repeat
3:         repeat
4:             if ¬skip-simulation then
5:                 R_C ← simulate-object-code(w, object-code);
6:                 Choose any ⟨w, v⟩ ∈ R_C;
7:                 R_C ← R_C \ ⟨w, v⟩;
8:                 R_U ← R_U ∪ R_C;
9:             if v ∈ I then
10:                path-complete ← TRUE;
11:            if [(|R_C| = 0)∧ (ref-map(w) = ref-map(v))∧
12:           (v ∉ non-abs-states)] then
13:                ss-length + +;
14:            else
15:                if ref-map(w) ≠ ref-map(v) then
16:                    R_I ← R_I ∪ ⟨w, v, 0⟩;
17:                    I ← I ∪ v;
18:                if [(ref-map(w) = ref-map(v))∧
19:              (v ∈ non-abs-states)] then
20:                    ss-length + +;
21:                    I ← I ∪ v;
22:                    R_I ← R_I ∪ ⟨w_abs, v, ss-length⟩;
23:                else
24:                    R_I ← R_I ∪ ⟨w_abs, w, ss-length⟩;
25:                if |R_C| = 0 then
26:                    w_abs ← v;
27:                    ss-length ← 0;
28:                else
29:                    w_abs ← w;
30:                    ss-length ← 1;
31:            w ← v;
32:            if skip-simulation then
33:                skip-simulation ← FALSE;
34:        until ¬path-complete
35:        R_I ← R_I ∪ ⟨w_abs, v, ss-length⟩;
36:        if R_U ≠ ∅ then
37:            Choose any ⟨w, v⟩ ∈ R_U;
38:            R_U ← R_U \ ⟨w, v⟩;
39:            w_abs ← w;
40:            ss-length ← 0;
41:            path-complete ← FALSE;
42:            skip-simulation ← TRUE;
43:    until ¬(path-complete ∧ R_U = ∅)
44:    return (R_I);
45:    exit;
```

$\langle 4, 8\rangle$ and $\langle 4, 11\rangle$. Consider state 6 as another example. Here R_C will contain $\langle 6, 7\rangle$. Since $\langle 6, 7\rangle$ is part of the trace that being computed, there is nothing to include in R_U. As the states are being computed dynamically, some of them cannot be abstracted. These states are stored in 'I', which is initialized to w_0. ss-length keeps track of the no. of states that have been abstracted so far in the stuttering segment. It is initialized to zero. The procedure uses two control flags path-complete and skip-simulation, both initialized to false. In Figure 5.2a, consider the stuttering transitions $\langle 21, 22\rangle$, $\langle 22, 23\rangle$ and $\langle 23, 24\rangle$. Here if $\langle w, v\rangle = \langle 23, 24\rangle$ and since state 23 satisfies the abstraction criteria, the abstracted segment should be $\langle 21, 24\rangle$. To keep track of the abstraction, the start state of an abstraction (in this example state 21) is stored in w_{abs}, which is initialized to w_0. The initializations described above are not show in the algorithm.

The algorithm is described using the implementation TTS in Figure 4.2. The initial state w_0 is state 1, the successor states are computed using function simulate-object-code (line 5). Here state 1 has one outgoing transitions. $\langle w, v\rangle$ is assigned this transition where $w = 1$ and $v = 2$ (line 6). This transition is removed from R_C (line 7). Since state 1 has only one successor, once $\langle 1, 2\rangle$ transition is removed from R_C it becomes empty. In line 8, when R_C is concentrated to R_U, for state 1 R_U remains empty since state 1 has only one transition. As another example, consider state 4. It has multiple outgoing transitions. Consider one of the successor transitions, say $\langle 4, 5\rangle$. The remaining successors are stored in R_U (line 7-8). In order to abstract a transition, it should be a stuttering transition, 'w' should have only one out-going transition and 'v' should have only one incoming transition (line 11-13). $\langle 1, 2\rangle$ does satisfy this criteria and ss-length is incremented by 1 (line 13); To proceed, state 2 is assigned as the current state (line 31) and the entire process is repeated. The successor of state 2 is computed and assigned to $\langle w, v\rangle$. Since state 2 also has only one successor, it is selected and proceeded to line 11. The condition on line 11 fails since $\langle 2, 3\rangle$ is not a stuttering transition. The next step is to figure out which criteria has failed in order to proceed further. The first criterion to check is if the transition is a non-stuttering transition (line 15-17). This condition is satisfied for $\langle 2, 3\rangle$ and hence this transition is added to R_I (line 16), which contains the set of abstracted transition for the given M_I. The state 3 should not be abstracted since it is not part of a stuttering transition. Hence, this state is stored in 'I' (line 17). The second criterion to check is if 'v' has multiple incoming transitions and $\langle w, v\rangle$ is a stuttering transition (line 18-22). This condition fails for $\langle 2, 3\rangle$. In the else part of this condition, the stuttering transitions

that have been visited so far has to be saved, hence is appended to R_I (lines 23-24). w_0 is initialized to state 1, w is assigned state 2. Hence $\langle 1, 2, 1\rangle$ is added to R_I where the final part of the tuple says that so far, the algorithm has seen one stuttering transition (line 24). The third criterion to check the number of outgoing transitions for 'w' (line 25-30). State 2 has only one transition, so lines 26-27 are executed. Here, w_{abs} is assigned to state v since state 3 may have stuttering transitions in its successors (line 26). The ss-length has been assigned to 0 to start the new count of stuttering segment (line 27). State 3 is assigned to w to repeat the process again.

Consider $\langle 4, 5\rangle$ where w is state 4 and v is state 5. This transition fails the stuttering transition condition since state 4 has multiple outgoing transitions. In the else part, this transition fails criteria 1, and criteria 2. It satisfies the else part of criteria 3. State 4 will be marked as the beginning of the abstracted segment w_{abs} (line 29). $\langle 4, 5\rangle$ is part of the stuttering segment, hence ss-length has the value 1. To proceed, state 5 is assigned as the current state (line 31) and the entire process is repeated. This process is repeated until a state that has been visited previously is seen again (lines 9-10). This sets path-complete flag to TRUE, as this path has been fully explored. The procedure then checks if there are other unexplored paths in R_U (line 36-42). If so, a transition from R_U is chosen. For example say $\langle 4, 8\rangle$. Since state 8 is already computed, lines 5-8 should be skipped. This is achieved by assigning skip-simulation to TRUE (line 42). Also, as a new path is being explored, path-complete is set to FALSE (line 41). When state 8 becomes the current state then skip-simulation flag is set to FALSE (line 32-33) as the successor states of 8 is unknown. The algorithm (line 3-34) is repeated for $\langle 4, 8\rangle$ until a state that has been previously visited is reached. This could be state 9 for $\langle 4, 8\rangle$ since $\langle 4, 5\rangle$ trace has been computed previously. After finishing the computation, the next transition in R_U is chosen and the process is repeated until R_U is empty. The procedure terminates when R_U is empty and path-complete is TRUE.

Since the procedure is unwinding and exploring the implementation TTS, its complexity is linear in the number of transitions of the implementation TTS.

5.3. Case Studies and Results

***Control Program - 1*:** A number of object code programs for stepper motor control were used as benchmarks to demonstrate the effectiveness of the methodology. Table 5.1 shows the verification statistics for the benchmarks. A stepper motor can be energized in various repeated sequences that cause it to rotate. Three typically used sequences were used to develop the

benchmarks. Sequence ⟨0001⟩, ⟨0010⟩, ⟨0100⟩, ⟨1000⟩, ⟨0001⟩, ..., is known as *full stepping or single stepping or wave stepping*. Sequence ⟨0011⟩, ⟨0110⟩, ⟨1100⟩, ⟨1001⟩, ⟨0011⟩, ..., is known as *double stepping*. Sequence ⟨0001⟩, ⟨0011⟩, ⟨0010⟩, ⟨0110⟩, ⟨0100⟩, ⟨1100⟩, ⟨1000⟩, ⟨1001⟩, ⟨0001⟩, ..., is known as *half stepping*. The benchmark name indicates the type of control used. "Full", "Double", and "Half" indicate full stepping, double stepping, and half stepping were used, respectively. "RIT" indicates that the interrupts were generated by Repetitive Interrupt Timer (RIT) to implement the timing delays for the motor control. "noRIT" indicates that instead of the RIT timer, code was to implement timing delays. "clock" and "anti" indicate that the motor was controlled clockwise and anti-clockwise, respectively. Table 5.1 gives statistics for both correct and buggy versions of the controllers. "FuncBug" and "TimBug" indicate that the object code error was either a functional error or a timing error, respectively. The programs were developed to run on an ARM Cortex-M3 based NXP LPC1768 [58] micro-controller. Stepper motor leads were connected to four pins from PORT 2 of the LPC1768 via an electronic circuit. Table 5.1 also includes the time for abstraction, which is in the order of 100s of seconds.

***Control Program - 2*:** An infusion pump is a medical device that is used to deliver controlled dosages of medications or nutrients into the patient's circulatory system intravenously. Typical medications delivered include opioids, insulin, and chemotherapy drugs. From 2001 to 2017, the Food and Drug Administration (FDA) has issued 54 Class-1 recalls on infusion pumps due to software errors [64, 65]. Class-1 recalls are issued when the use of the device is determined to cause serious harm or death to patients. The criticality of the pump functionality is because of incorrect dosage delivery due to software errors. The Alaris Medley 8100 LVP module infusion pump [1, 13] was used for the experiments. The Alaris Medley pump uses pulse width modulation for dosage control. Pulse width modulation control code for the Alaris pump was implemented on an ARM Cortex M3 based LPC 1768 micro-controller, which was interfaced with the pump so that the code implemented can control the pump. The formal specifications were developed for the pump control software based on the requirements in [67]. Table 5.2 shows the verification statistics for the infusion pump control case study. The transition system of the pump's control code had about 24.3 million transitions. With the application of dynamic stuttering abstraction, the abstract transition system was computed in about 70 seconds and had less than 100 transitions. TWFS refinement verification was then completed within a few seconds. Several bugs were found in the

Table 5.1. Verification Statistics for TSA

S.No	Object Code Benchmarks	# of Trans. of MM_I [million]	# of Trans. of of Abstract MM^a	TSA Time [sec]
1	Full-RIT-clock	2.5	10	26.95
2	Full-RIT-anti	2.5	10	15.29
3	Double-RIT-clock	2.5	10	15.31
4	Double-RIT-anti	2.5	10	15.35
5	Half-RIT-clock	4.5	18	31.85
6	Half-RIT-anti	4.5	18	31.85
7	Full-noRIT-clock	82.5	10	171.88
8	Full-noRIT-anti	82.5	10	172.23
9	Double-noRIT-clock	82.5	10	171.25
10	Double-noRIT-anti	82.5	10	171.32
11	Half-noRIT-clock	148.5	18	321.58
12	Half-noRIT-anti	148.5	18	322.84
13	FuncBug-Full-RIT-clock	2.5	10	15.1
14	FuncBug-Full-RIT-anti	2.5	10	15.25
15	FuncBug-Double-RIT-clock	2.5	10	15.12
16	FuncBug-Double-RIT-anti	2.5	10	15.15
17	FuncBug-Half-RIT-clock	4.5	18	31.59
18	FuncBug-Half-RIT-anti	4.5	18	32.05
19	FuncBug-Full-noRIT-clock	99	20	211
20	FuncBug-Full-noRIT-anti	99	20	210.79
21	FuncBug-Double-noRIT-clock	82.5	10	176.24
22	FuncBug-Double-noRIT-anti	82.5	10	173.3
23	TimBug-Half-noRIT-clock	82.5	18	172.72
24	TimBug-Half-noRIT-anti	120	20	255.3

control code implementation which can categorized as functional and timing errors. The results for each of the buggy versions of the code and the correct code are summarized in Table 5.2. With this case study, it can be concluded that the dynamic stuttering abstraction is applicable to complex real-world control code verification.

In Table 5.1 and 5.2, the "# of Trans. of MM_I" column (column 3) given the total number of transitions in the implementation model TTS. The "# of Trans. of Abstract MM^a" column

Table 5.2. Verification Statistics for TSA for Infusion Pump Controller (IPC) Case Study

S.No	Object Code Benchmarks	# of Trans. of [million]	TSA Time [sec] of Abstract MM^a
1	IPC	24.3	70.28
2	IPC-FuncBug1	20.25	60.52
3	IPC-FuncBug2	24.3	75.15
4	IPC-FuncBug3	27	83.99
5	IPC-TimBug1	20.25	60.04
6	IPC-TimBug2	24.3	72.32

(column 4 of Table 5.1) gives the total number of transitions in the abstract implementation TTS (which is generated by applying dynamic TSA). Column 5 of Table 5.1 and Column 4 of Table 5.2 indicated as "TSA Time" gives the timed stuttering abstraction time in seconds to generate MM^a. The verification experiments were performed on a Intel Core i7 3.1 GHz processor with 8 GB memory. We also performed TWFS refinement checking on the abstracted TTS and were able to prove correctness or flag the functional/timing bug in the benchmarks. TWFS refinement checking was able to complete in less than a second for all the abstracted TTSs. Without abstraction, TWFS refinement checking was not possible because of memory issues due to the size of the implementation TTS.

This chapter has presented stuttering abstraction (SA) for transition systems (TSs), timed stuttering abstraction (TSA) for timed transition systems (TTSs). Dynamic timed stuttering abstraction has been fruitfully applied to real-time object code verification. Dynamic TSA has been demonstrated on two case studies, including stepper motor control and infusion pump control. Verification results demonstrate the effectiveness of dynamic TSA and its application reduces the number of transitions of the implementation TTS from millions to less than 100 for all benchmarks. As expected, most of the time is spent in performing dynamic TSA. But, once abstraction is applied, functional and timing verification times are very efficient.

6. WFS REFINEMENT CHECKER

In Chapter 4, a formal verification methodology for real-time interrupt-driven object code verification was described. Timed well-founded simulation refinement (TWFS) methodology is build on the theory of Well-Founded Simulation (WFS) refinement. In the context of WFS refinement, both the implementation and specification are modeled as transition systems (TSs). Transition system is a mathematical modeling framework for code that is based on states of the program and transitions between states. WFS refinement essentially defines what it means for an implementation TS to correctly implement a specification TS.

For TWFS refinement, WFS refinement checking was demonstrated by manually generating the required proof obligations for checking WFS refinement. However, this is insufficient for large programs. In this chapter, this gap is addressed by proposing an algorithm for automatic WFS refinement checking optimized for object code verification. The proposed algorithm checks for safety property which is based on WFS refinement. Safety informally means that if the implementation makes progress, the result of that progress satisfies the specification requirements. The algorithm has been implemented and the automated tool flow has been applied to a comprehensive set of thirty object code control programs to demonstrate the effectiveness of the approach.

6.1. Automated WFS Refinement for Object-Code

This section presents a procedure for automating WFS refinement for object code verification. According to Definition 2, a TS satisfies WFS when the two conditions are satisfied. Usually, in real-time object code verification, stuttering does not occur on the specification system. Hence the refinement-based correctness formula can be reduced to,

$$\begin{aligned}
&\langle \forall w \in \text{object-code} :: v = \text{object-code-step}(w) \wedge s = r(w) \\
&\wedge u = \text{SPEC-step(s)} \wedge \langle s, u \rangle \in \text{SPEC then} \\
&\text{(i)} \quad r(v) = s \text{ (for stuttering transition) or} \\
&\text{(ii)} \quad r(v) = u \text{ (for a non-stuttering transition) } \rangle
\end{aligned} \tag{6.1}$$

Here, once a refinement-map is constructed, in WFS refinement verification the idea is then to look at each transition. For example, consider $\langle w, v\rangle$ from the implementation transition system where w, v belong to the set of implementation states. To satisfy the refinement-based correctness formula 6.1, the transition should capture one of the options of being either a stuttering transition or a non-stuttering transition. If the implementation transition match to the same specification state i.e., $r(w) = r(v) = s$ where $r()$ is the refinement-map and s is a specification state, then it is called a stuttering implementation transition. If the implementation transition match to a specification transition i.e., $r(w) = s$ and $r(v) = u$ where $\langle s, u\rangle$ is a transition in specification, then it is called a non-stuttering implementation transition. If the implementation transition shows a behavior that is neither stuttering nor non-stuttering then it indicates the presence of an error or bug in the implementation transition system.

Typically object code (implementation TS (M_I)) consists of millions of transitions where a large portion of these transitions are of stuttering in nature. Hence, abstraction based on these stuttering transitions may be applied to the implementation TS (M_I). Applying stuttering abstraction heuristics on the TS (Section 7.1.1) makes the verification process faster and much more efficient. Using the specification TS (M_S), the set of states of the specification (S_S) can be constructed. If every behavior of M_I has a match in M_S then M_I is a WFS refinement of M_S with respect to refinement-map r and is denoted as $M_I \sqsubseteq_r M_S$. The refinement-map is a function that maps the symbolic states of M_I to the symbolic states of M_S.

6.1.1. Procedure for Checking WFS Refinement for Object-Code

Algorithm 3 presents a procedure that performs WFS refinement checking on the abstracted object code TS, which is the implementation TS. The inputs to the procedure include a list of transitions of M_I, a list of transitions of M_S, a set of states of the specification (S_S) and the refinement-map r. R_{CE} is the counterexample set, which the procedure will populate with implementation transitions that do not satisfy the WFS refinement correctness criteria (6.1). R_{CE} is initially empty (line 2). The procedure iterates through each transition in M_I (lines 3-19). The transitions of the implementation are of the form $\langle w, v\rangle$. State variables w and v are assigned to be the predecessor-state (zeroth column value of M_I[i]) and successor-state (first column value of M_I[i]) of transition M_I[i], respectively (lines 4-5). Variable 's' is assigned to be the value refinement-map (w) (line 6). Predicate *match* is used to keep track of whether the implementation transition has found

a matching specification transition, or is determined to be stuttering, or is neither. The case where it is neither is usually a counterexample or a bug.

Algorithm 3 Procedure for Checking WFS Refinement

```
1:  procedure CheckwfsRef(M_I, M_S, S_S, r)
2:      R_CE ← NULL;
3:      for i ← 1 to length-of-M_I do
4:          w ← M_I(0);
5:          v ← M_I(1);
6:          s ← r(w);
7:          match ← FALSE;
8:          if s ∈ S_S then
9:              if r(v) = r(w) then
10:                 match ← TRUE;
11:             else
12:                 for j ← 1 to length-of-M_S do
13:                     if s = M_S(0) then
14:                         u ← M_S(1);
15:                         if r(v) = u then
16:                             match ← TRUE;
17:                             break;
18:         if match = FALSE then
19:             R_CE ← R_CE ∪ ⟨w, v⟩;
20:     return R_CE;
```

A check is performed on 's' to see if it belongs to the set of states of specification (S_S) (line 8). If 's' does not belong to S_S, then the w state has no corresponding specification state and therefore points to an error and the transition is added to R_{CE} (lines 18-19). When 's' exists in S_S, a check has to be performed on the implementation transition to see if it is a stuttering implementation transition or a non-stuttering implementation transition. In case the transition is a stuttering transition (lines 9-10), the predicate *match* is set to true and the procedure proceeds to the next transition in the implementation. If the transition is a non-stuttering transition (lines 11-17), the procedure iterates through the specification transitions. In the specification transitions, only transitions where the predecessor-state is s is considered (lines 13-17). The corresponding successor-state is assigned to 'u' (line 14). If a specification successor-state u is found such that $r(v)$ is equal to u, then *match* is assigned true (lines 15-16). Once a match is found, the procedure exits iterating through the specification transitions and moves on to the next implementation transition

(line 17). If for a non-stuttering transition, $r(v)$ does not match with any u, then this point to an error in the implementation and the corresponding implementation transition is appended to R_{CE}. When all the transitions of the implementation have been checked, the procedure ends by returning the list R_{CE} (line 20).

6.1.2. Time Complexity for Refinement Checker

The time complexity of this algorithm is $\mathcal{O}(|R_I||R_S|)$. The outer for loop (line 3) has a length of R_I passes. The if condition on line 8 has a length of S_S passes. The inner for loop (line 12) has a length of R_S passes. The complexity of the algorithm is $|R_I| * (|S_S| + |R_S|)$. Usually, the number of transitions of the specification ($|R_S|$) is greater than equal to the number of states of specification ($|S_S|$) depending on the application. Hence, the overall time complexity is $\mathcal{O}(|R_I||R_S|)$.

6.2. Case Studies and Results

The effectiveness of the algorithm presented in this chapter was demonstrated on 30 different object code programs. A detailed description of the control programs has been explained in Chapter 3.

***Control Program - 1*:** A number of object code programs for stepper motor control were used as benchmarks to demonstrate the effectiveness of the methodology. Three sequences of stepper motor control that uses 4 leads are used to develop the benchmarks. Double stepping sequence an be described as $\langle 0011 \rangle$, $\langle 0110 \rangle$, $\langle 1100 \rangle$, $\langle 1001 \rangle$, $\langle 0011 \rangle$, so on. Full stepping sequence an be described as $\langle 0001 \rangle$, $\langle 0010 \rangle$, $\langle 0100 \rangle$, $\langle 1000 \rangle$, $\langle 0001 \rangle$, so on. Half stepping sequence an be described as $\langle 0001 \rangle$, $\langle 0011 \rangle$, $\langle 0010 \rangle$, $\langle 0110 \rangle$, $\langle 0100 \rangle$, $\langle 1100 \rangle$, $\langle 1000 \rangle$, $\langle 1001 \rangle$, $\langle 0001 \rangle$, so on. The benchmark name indicates the type of control used. "Full", "Double", and "Half" indicate full stepping, double stepping, and half stepping were used, respectively. "RIT" indicates that the interrupts were generated by Repetitive Interrupt Timer (RIT) to implement the timing delays for the motor control. "noRIT" indicates that instead of the RIT timer, the code was used to implement timing delays. "clock" and "anti" indicate that the motor was controlled clockwise and anti-clockwise, respectively. "FuncBug" and "TimBug" indicate that the object code error was either a functional error or a timing error, respectively. The programs were developed to run on an ARM Cortex-M3 based NXP LPC1768 [58] micro-controller. Stepper motor leads were connected to four pins from PORT 2 of the LPC1768 via an electronic circuit.

Table 6.1. Verification Statistics for WFS Refinement Checker

Case Study	Object Code Benchmarks	# of Trans. of MM_I [million]	# of Trans. of of Abstract MM^a	WFS refinement checker time [microsec]
1	Full-RIT-clock	2.5	10	3
2	Full-RIT-anti	2.5	10	3
3	Double-RIT-clock	2.5	10	4
4	Double-RIT-anti	2.5	10	3
5	Half-RIT-clock	4.5	18	4
6	Half-RIT-anti	4.5	18	3
7	Full-noRIT-clock	82.5	10	3
8	Full-noRIT-anti	82.5	10	4
9	Double-noRIT-clock	82.5	10	4
10	Double-noRIT-anti	82.5	10	6
11	Half-noRIT-clock	148.5	18	3
12	Half-noRIT-anti	148.5	18	5
13	FuncBug-Full-RIT-clock	2.5	10	-
14	FuncBug-Full-RIT-anti	2.5	10	-
15	FuncBug-Double-RIT-clock	2.5	10	-
16	FuncBug-Double-RIT-anti	2.5	10	-
17	FuncBug-Half-RIT-clock	4.5	18	-
18	FuncBug-Half-RIT-anti	4.5	18	-
19	FuncBug-Full-noRIT-clock	99	20	-
20	FuncBug-Full-noRIT-anti	99	20	-
21	FuncBug-Double-noRIT-clock	82.5	10	-
22	FuncBug-Double-noRIT-anti	82.5	10	-
23	TimBug-Half-noRIT-clock	82.5	18	3
24	TimBug-Half-noRIT-anti	120	20	4
25	IPC	24.3	6	3
26	IPC-FuncBug1	20.25	5	-
27	IPC-FuncBug2	24.3	8	-
28	IPC-FuncBug3	27	3	-
29	IPC-TimBug1	20.25	6	3
30	IPC-TimBug2	24.3	6	4

***Control Program - 2*:** An infusion pump is a medical device that is used to deliver controlled dosages of medications or nutrients into the patient's circulatory system intravenously. Typical medications delivered include opioids, insulin, and chemotherapy drugs. From 2001 - 2017, the Food and Drug Administration (FDA) has issued 54 Class-1 recalls on infusion pumps due to software errors [64,65]. Class-1 recalls are issued when the use of the device is determined to cause serious harm or death to patients. The criticality of the pump functionality is because of incorrect dosage delivery due to software errors. The Alaris Medley 8100 LVP module infusion pump [1,13] was used for the experiments. The Alaris Medley pump uses pulse width modulation for dosage control. Pulse width modulation control code for the Alaris pump was implemented on an ARM Cortex M3 based LPC 1768 micro-controller, which was interfaced with the pump so that the code implemented can control the pump. The formal specifications were developed for the pump control software based on the requirements in [67]. Table 6.1 also shows the verification statistics for the infusion pump control case study. The transition system of the pump's control code had about 24.3 million transitions. With the application of dynamic stuttering abstraction, the abstract transition system was computed in about 70 seconds and had less than 100 transitions. WFS refinement verification was then completed within a few seconds. Several bugs were found in the control code implementation which can be categorized as functional and timing errors.

Table 6.1 gives statistics for both correct and buggy versions of the benchmarks. The programs were developed to run on an ARM Cortex-M3 based NXP LPC1768 [58] microcontroller. The verification experiments were performed on an Intel Core i7 3.1 GHz processor with 8GB memory. Repetitive Interrupt Timer (RIT) was used to generate the interrupt at regular intervals of time. In Table 6.1, the "# of Trans. of MM_I" column (column 3) given the total number of transitions in the implementation model TS. The "# of Trans. of Abstract MM^a" column (column 4) gives the total number of transitions in the abstract implementation TS. Column 5 indicated as "WFS refinement checker time" gives the time taken to perform automated WFS refinement checking on the abstracted implementation TTS MM^a. The verification experiments were performed on an Intel Core i7 3.1 GHz processor with 8 GB memory. WFS refinement checking was able to complete in less than a second for all the abstracted TSs. Without abstraction, WFS refinement checking was not possible because of memory issues due to the size of the implementation TS. WFS refinement checking time is not shown for buggy models.

This chapter has presented automated well-founded simulation (WFS) refinement checker for large programs like real-time interrupt-driven object code programs. This algorithm checks for the safety feature in transition systems (TSs). The time complexity of this algorithm is $\mathcal{O}(|R_I||R_S|)$ which is linear in terms of the number of transitions of the implementation TS and specification TS. The refinement checker has been demonstrated on multiple benchmarks, where the verification time is very efficient making it possible to apply this procedure multiple times in practice.

7. LIVENESS

This chapter presents the algorithm for liveness verification of real-time interrupt-driven object code programs. Liveness verification checks for deadlock errors in the object code. Liveness property indicates that something good eventually happens. Deadlock in object code indicates that the execution of the instructions is in an infinite loop and from which execution might never come out from. This kind of situation usually indicates an error in the implementation transition system (TS) *i.e.*, the object code.

In graph theory, a strongly connected component (SCC) exists when every node in a graph is reachable from every other node. Deadlock in the transition system indicates the presence of a SCC. Tarjan's strongly connected component algorithm is an algorithm in graph theory that is usually used to find the SCC in a directed graph. Tarjan's SCC algorithm has linear time complexity.

Figure 7.1 shows a buggy implementation transition system for the specification transition system shown in Figure 2.4. This buggy model is considered as an example for the explanation in this chapter.

7.1. Liveness Detection for Object-Code

Deadlock errors in object code result from the software changing states but not making progress with respect to the specification. Liveness verification checks for deadlock errors. Deadlock shows up as a sequence of infinite stuttering steps because of progress w.r.t. the specification is captured by non-stuttering transitions. Since the transitions in the implementation transition system (TS) are finite, deadlock errors will show up as cycles of stuttering transitions in the implementation TS.

Well-Founded Simulation (WFS) refinement uses rank functions to check for liveness. Rank functions map implementation states to natural numbers. They are to be designed by the verification engineer so that the rank value decreases on every stuttering transition, which is a requirement for WFS refinement. If deadlock exists, the rank will eventually reach 0 or its minimum value and stop decreasing on stuttering transitions. Therefore, violations of the property that the rank de-

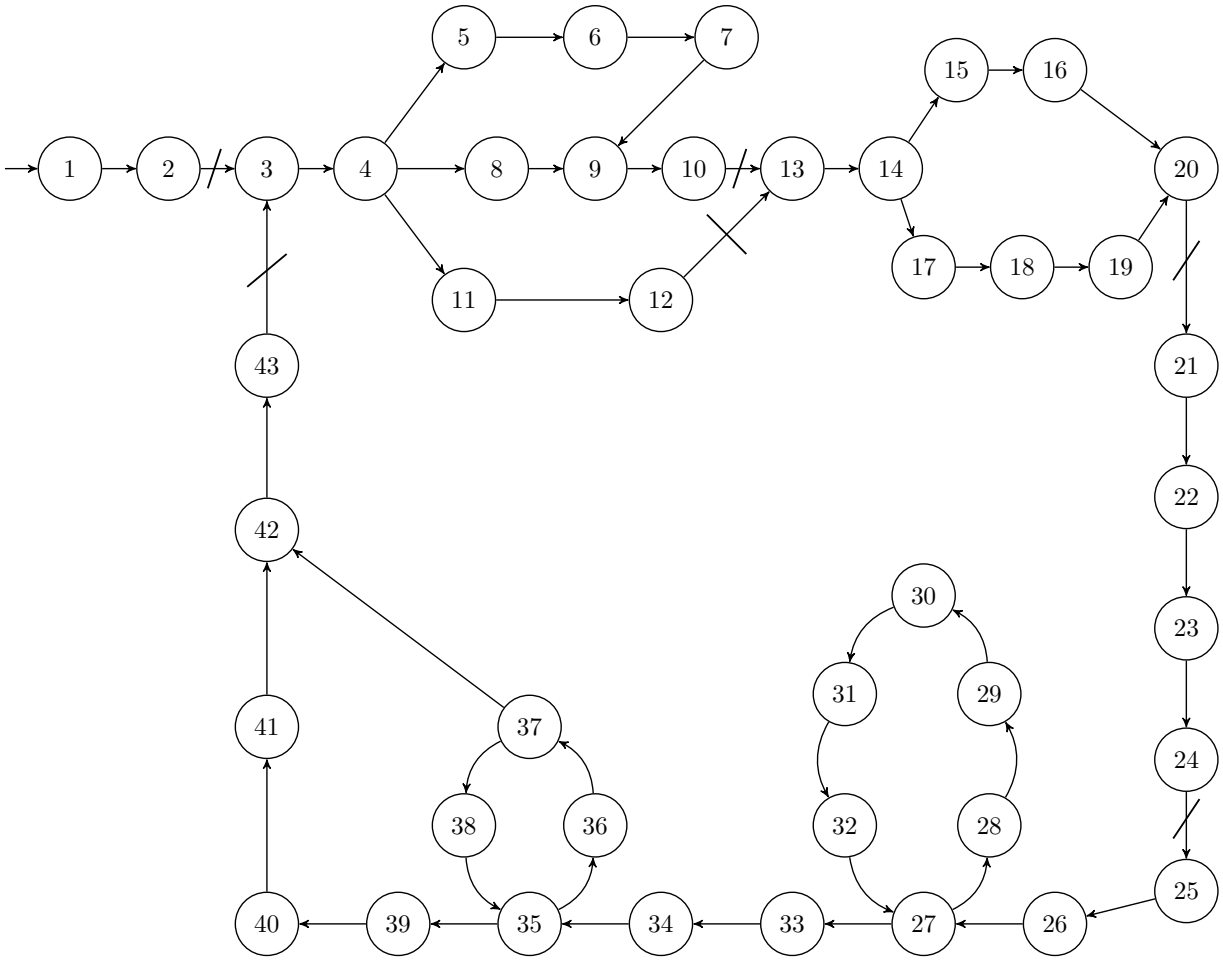


Figure 7.1. Buggy Stepper Motor Control Implementation TS

creases on every stuttering transition correspond to deadlock errors. Rank functions reduce liveness verification to reasoning about single transitions and therefore lead to effective verification.

An approach has been proposed for liveness checking that directly searches for stuttering cycles in the implementation TS. Rank functions are not used because they depend on the function of the code and are not easy to automate. Rank functions provide efficient verification when the implementation has a large number of transitions. While that is true for object code TSs, the fact that stuttering abstraction drastically reduces the size of the implementation TS can be exploited. Therefore, by checking for stuttering cycles after the implementation TS has been abstracted, both automation and efficiency can be attained. In this section, a procedure for detecting stuttering cycles is provided. Before that, the stuttering abstraction for transition system is revisited.

7.1.1. Stuttering Abstraction for TS

In Chapter 5, stuttering abstraction for transition system (TS) and timed transition system (TTS) has been discussed. From the definition of stuttering abstraction (Def. 14), two transitions that are stuttering can be merged into one. Repeated application of the definition can be used to merge large sequences of stuttering transitions. Also, the stuttering abstraction algorithm does not remove initial states. The lemma below gives the soundness of checking for deadlock errors by detecting stuttering cycles after applying stuttering abstraction to the implementation transition system (M_I).

Lemma 9. *Stuttering abstraction does not add or remove stuttering cycles from an implementation TS that are reachable from the initial states of the TS. The only impact that stuttering abstraction has on stuttering cycles is to reduce their size.*

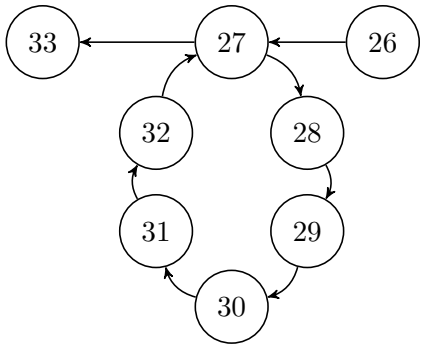
Proof. From Definition 14, stuttering abstraction merges two transition into one and the state that is removed due to this merger can only have one incoming and one outgoing transition. Such a merger will not create new cycles. Also, for a stuttering cycle to occur, there exists at least one state in the cycle with multiple incoming transitions (because the cycle has to be reachable from the initial state) unless the cycle includes an initial state. Since initial states are not abstracted and also states with multiple incoming transitions are not abstracted, stuttering abstraction will not remove stuttering cycles from the TS. In stuttering cycles, if and only if it contains states that have only one incoming and one outgoing transition, then those states can be abstracted which will reduce the overall size of the stuttering cycle. □

It can be seen in practice that stuttering abstraction often converts stuttering cycles to self-loops.

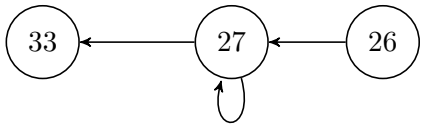
Definition 16. *A self-loop is a state that transitions to itself.*

In Figure 7.1, transitions $\langle 27, 28\rangle$, $\langle 28, 29\rangle$, $\langle 29, 30\rangle$,$\langle 20, 31\rangle$, $\langle 31, 32\rangle$ and $\langle 32, 27\rangle$ form a stuttering cycle. Repeated application of stuttering abstraction reduces this cycle to a self-loop on state 27 (as shown in Figure 7.2).

Lemma 10. *Stuttering abstraction can reduce stuttering cycles to self-loops.*



(a) Implementation TS Snippet With Stuttering Cycle



(b) Abstracted Implementation TS Snippet That Contains Self Loop

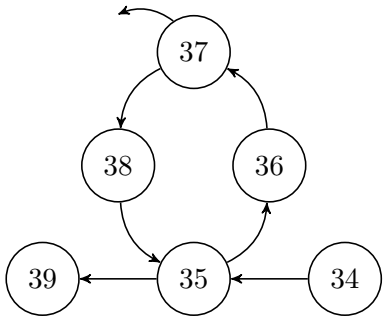
Figure 7.2. An Example Snippet of Implementation TS from Fig. 7.1 that Contains Stuttering Cycle

Proof. Stuttering abstraction will remove any state in a stuttering cycle that has only one incoming and one outgoing transition. Stuttering abstraction will not remove a state with multiple incoming or multiple outgoing transitions. Thus, if a stuttering cycle has only one state with multiple incoming and multiple outgoing transitions, then that will be the only state not to be removed from the cycle and such a stuttering cycle will reduce to a self-loop. □

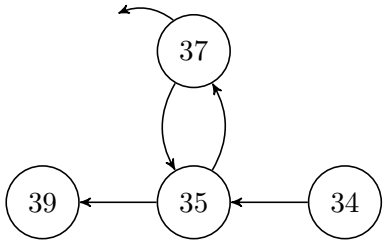
In Figure 7.1, transitions $\langle 35, 36\rangle$, $\langle 36, 37\rangle$, $\langle 37, 38\rangle$ and $\langle 38, 35\rangle$ form a stuttering cycle. Repeated application of stuttering abstraction does not reduce this cycle to a self-loop because state 37 contains two outgoing transitions (as shown in Figure 7.3).

Lemma 11. *Not every stuttering cycle can be reduced to self-loop.*

Proof. If a stuttering cycle has more than one state with multiple incoming or multiple outgoing transitions, then all such states will be preserved by stuttering abstraction. Such a stuttering cycle will not be reduced to a self-loop. □



(a) Implementation TS Snippet with Stuttering Cycle



(b) Abstracted Implementation TS Snippet That Does Not Reduce To Self Loop

Figure 7.3. An Example Snippet of Implementation TS From Fig. 7.1 that Contains Stuttering Cycle which Cannot be Reduced to Self-Loop

7.1.2. Liveness Detection Procedure

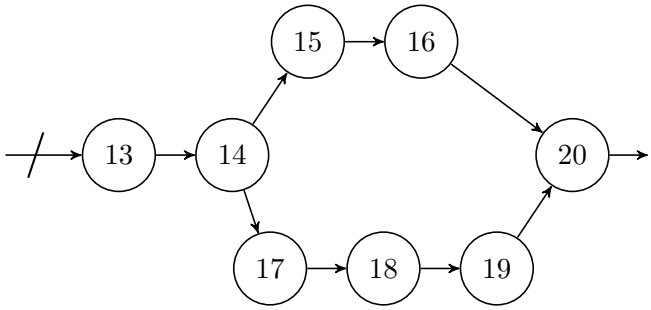
Algorithm 4 presents the procedure for detecting stuttering cycles that applies the well-known Tarjan's Strongly Connected Component (SCC) analysis algorithm, which is the most efficient algorithm for detecting cycles in a directed graph. The inputs to the procedure include the list of transitions of M_I. The procedure uses two boolean variables as flags called flag-self-loop and flag-SCC. Flag-self-loop helps to identify the presence of self-loops in the TS. Flag-SCC is used to identify the presence of strongly connected components (SCCs) in the TS. These variables are initialized to FALSE (lines 2-3). Since we are only interested in detecting stuttering cycles, the non-stuttering transitions are removed from M_I to get $M_{IStuttering}$ (line 4).

Once the non-stuttering transitions are removed, the resulting $M_{IStuttering}$ may have states that have no incoming and no outgoing transitions, called trivial SCCs. This can happen if, in M_I, there are states that have incoming and outgoing transitions that are only non-stuttering. Such trivial SCCs, however, will also be flagged as SCCs by Tarjan's algorithm and will, therefore, lead

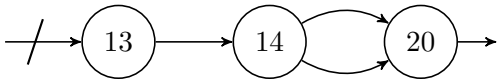
Algorithm 4 Procedure for checking stuttering cycles

```
1:  procedure CheckSC(M_I)
2:      flag-self-loop ← FALSE;
3:      flag-SCC ← FALSE;
4:      Remove the non-stuttering transitions in M_I to get M_IStuttering.
5:      Remove trivial SCC from M_IStuttering.
6:      Eliminate duplicate transitions in M_IStuttering.
7:      flag-self-loop ← Detect self-loops in M_IStuttering;
8:      if flag-self-loop then
9:          return counterexamples;
10:         exit;
11:     flag-SCC ← check-Tarjan-SCC;
12:     if flag-SCC then
13:         return SCCs;
14:         exit;
15:     return "No stuttering circles in the TS";
```

to spurious counterexamples. Such trivial SCCs are easily detected and removed from $M_{IStuttering}$ (line 5) with just one pass through the states of $M_{IStuttering}$.



(a) Implementation TS Snippet with Multiple Paths Between State 14 and State 20



(b) Abstracted Implementation TS Snippet that Contains Multiple Paths Between State 14 and State 20

Figure 7.4. An Example Snippet of Implementation TS from Fig. 7.1 that Contains Multiple Paths

In M_I, there may be multiple paths from one unabstractable state say w to another unabstractable state say v. If these paths only consist of stuttering transitions, when stuttering abstraction is applied, these paths will be reduced to single transitions from w to v that cannot be distinguished. Such transitions will essentially be duplicates in $M_{IStuttering}$ and will unnecessarily

reduce the efficiency of Tarjan's algorithm. For example, consider Figure 7.4 which is a snippet from Figure 7.1, the unabstractable state w is state 14 and the unabstractable state v is state 20. As can be seen in Figure 7.4a, there exist two paths from state 14 to state 20 in which all the transitions in both the paths are stuttering transitions. Since each of the stuttering transition in the multiple paths consists of only one incoming transition and one outgoing transition, these transitions can be abstracted. The abstracted transitions for these multiple paths can be seen in Figure 7.4b. Therefore, such duplicate transitions are detected and removed with just one pass through the states of $M_{IStuttering}$ (line 6). While the removal of trivial SCCs and the removal of duplicate transitions are shown separately, they can be combined into one pass in the implementation.

In practice, it can be seen that many of the stuttering cycles reduce to self-loops. Therefore, self-loops, which can also be checked using one pass through the states of $M_{IStuttering}$ by checking if a state has a transition to itself, are detected first (line 7). Lemma 10 guarantees that some stuttering cycles will reduce to self-loops. If self-loops are found, the procedure returns them as counterexamples and exits (lines 8 - 10). Only if $M_{IStuttering}$ does not have self-loops does the procedure employ Tarjan's algorithm (line 11). Lemma 11 states that all not all stuttering cycles will reduce to self-loops and therefore that Tarjan's algorithm be employed is a requirement for detecting all stuttering cycles. In practice, when detecting and correcting errors, the verification tool is often employed multiple times. Detecting self-loops directly, therefore, reduces the number of times Tarjan's algorithm is employed and therefore improves the efficiency of the verification process.

7.1.3. Time Complexity for Liveness Detection

The time complexity of this procedure is $\mathcal{O}(|R_{IStuttering}| + |S_{IStuttering}|)$, where $R_{IStuttering}$ denotes the transitions in $M_{IStuttering}$ and $S_{IStuttering}$ denotes the states in $M_{IStuttering}$. As described above, lines 4, 5, and 6, each require one pass through the states of $M_{IStuttering}$ and therefore each of these steps require $S_{IStuttering}$. The time complexity for Tarjan's SCC algorithm (line 11) for a graph with V vertices and E edges is $\mathcal{O}(|V|+|E|)$. For the implementation TS, the vertices and edges correspond to the states and transitions, respectively. Therefore, the complexity of Tarjan's algorithm here is $\mathcal{O}(|R_{IStuttering}| + |S_{IStuttering}|)$, which dominates the other steps in the procedure and therefore also determines the complexity of the procedure.

7.2. Case Studies and Results

The effectiveness of the algorithms presented in this chapter was applied to all the benchmarks in Chapter 6.

***Control Program - 1*:** A number of object code programs for stepper motor control were used as benchmarks to demonstrate the effectiveness of the methodology. Three sequences of stepper motor control that uses 4 leads are used to develop the benchmarks. Double stepping sequence an be described as $\langle 0011 \rangle$, $\langle 0110 \rangle$, $\langle 1100 \rangle$, $\langle 1001 \rangle$, $\langle 0011 \rangle$, so on. Full stepping sequence an be described as $\langle 0001 \rangle$, $\langle 0010 \rangle$, $\langle 0100 \rangle$, $\langle 1000 \rangle$, $\langle 0001 \rangle$, so on. Half stepping sequence an be described as $\langle 0001 \rangle$, $\langle 0011 \rangle$, $\langle 0010 \rangle$, $\langle 0110 \rangle$, $\langle 0100 \rangle$, $\langle 1100 \rangle$, $\langle 1000 \rangle$, $\langle 1001 \rangle$, $\langle 0001 \rangle$, so on. The benchmark name indicates the type of control used. "Full", "Double", and "Half" indicate full stepping, double stepping, and half stepping were used, respectively. "RIT" indicates that the interrupts were generated by Repetitive Interrupt Timer (RIT) to implement the timing delays for the motor control. "noRIT" indicates that instead of the RIT timer, the code was to implement timing delays. "clock" and "anti" indicate that the motor was controlled clockwise and anti-clockwise, respectively. "FuncBug" and "TimBug" indicate that the object code error was either a functional error or a timing error, respectively. The programs were developed to run on an ARM Cortex-M3 based NXP LPC1768 [58] micro-controller. Stepper motor leads were connected to four pins from PORT 2 of the LPC1768 via an electronic circuit.

***Control Program - 2*:** An infusion pump is a medical device that is used to deliver controlled dosages of medications or nutrients into the patient's circulatory system intravenously. Typical medications delivered include opioids, insulin, and chemotherapy drugs. From 2001 to 2017, the Food and Drug Administration (FDA) has issued 54 Class-1 recalls on infusion pumps due to software errors [64, 65]. Class-1 recalls are issued when the use of the device is determined to cause serious harm or death to patients. The criticality of the pump functionality is because of incorrect dosage delivery due to software errors. The Alaris Medley 8100 LVP module infusion pump [1, 13] was used for the experiments. The Alaris Medley pump uses pulse width modulation for dosage control. Pulse width modulation control code for the Alaris pump was implemented on an ARM Cortex M3 based LPC 1768 micro-controller, which was interfaced with the pump so that the code implemented can control the pump. The formal specifications were developed for

the pump control software based on the requirements in [67]. The transition system of the pump's control code had about 24.3 million transitions. The abstracted implementation model had less than 100 transitions. It was possible to apply WFS refinement verification with few seconds.

Algorithm 4 have been on all the benchmarks in Chapter 6 after the safety feature has been verified using the refinement checker algorithm 3. The programs were developed to run on an ARM Cortex-M3 based NXP LPC1768 [58] microcontroller. The verification experiments were performed on an Intel Core i7 3.1 GHz processor with 8GB memory. Repetitive Interrupt Timer (RIT) was used to generate the interrupt at regular intervals of time. Liveness verification was able to complete in less than a second for all the abstracted TSs. Without abstraction, liveness verification was not possible because of memory issues due to the size of the implementation TS.

This chapter presents the effectiveness of the strongly connected components (SCCs) checking on the real-time interrupt-driven object code programs. This algorithm first checks for self-loops since that would the case for most abstracted implementation TS. Later Tarjan's SCC is applied to the abstracted implementation TS. The time complexity of this algorithm is $\mathcal{O}(|R_{IStuttering}| + |S_{IStuttering}|)$ which is similar to the time complexity of Tarjan's SCC algorithm. The liveness verification has been demonstrated on multiple benchmarks, where the time taken is very efficient making it possible to apply this procedure multiple times in practice.

8. RELATED WORK

A large gap exists between high-level system models and the low-level actual code (*i.e.*, object-code) which is executed on the embedded device. The number of states and transitions in the high-level system models is typically less than 100, whereas, in the low-level models the number of states and transitions may be in the order of millions. Catching the design bugs early in the design cycle of the system-level models is very useful. To bridge the gap between system-level models and actual code, model-driven approaches are adopted. Here the high-level system models are represented as source code which is developed using platform-independent synthesis tools. The source code is then augmented with the device peripheral information and then compiled and assembled to generate the object code. During this process, numerous errors can creep into the object code compromising its safety. This work is targeted at bridging the gap between the real-time high-level models and real-time object code and also ensuring that the object-code is safe.

8.1. Refinement Based Verification for Real-Time Systems

Timed automata [3], is an extension of finite-state automata using clock variables. Tools like UPPAAL [42] and Kronos [11] are based on timed automata [3], and have been very successful in property-based verification of real-time system-level models and models of protocols. Another real-time system verification tool is Epsilon [31], which is aimed at verifying communication protocols. These real-time model checkers are targeted at checking properties of system-level models.

However, there is quite a large gap between the high-level models and the actual code that is executed in the device. High-level models typically have less than 100 states and transitions, while the timed transition system (TTS) corresponding to object code has millions of states and transitions. This gap is bridged using a model-driven approach [40], where platform-independent source code is generated from high-level models using synthesis tools. The platform-independent source code is then manually augmented with glue code, which is used to configure hardware interrupts and interface the software controller with low-level peripheral units and devices such as timers, communication channels, ports, etc. The glue-code-augmented source code is then compiled and assembled to generate object code. There are numerous sources of error that can compromise

the safety of the object code in this process. Our work is targeted at guaranteeing that the software controller at the object-code-level is safe.

David et al. [18] have developed an UPPAAL-based tool to check refinement between specifications of real-time systems. Boudjadar et al. [9] have developed a bisimulation relation for real-time systems with priorities and provide a method for encoding and verifying the problem using UPPAAL. The above refinement approaches for real-time systems are targeted at high-level models and do not consider stuttering and refinement maps. Since we incorporate stuttering and refinement maps, our approach is unique in this regard and applicable to the verification of low-level implementations such as object code.

Alur et al. have defined bisimulation based equivalences for TTS [2], and have proposed a method based on language inclusion for checking equivalence of TTS. However, these methods have not been applied towards real-time object code verification.

Rabinovich [59] has shown how to lift the theory of automata for discrete timed systems to continuous timed systems. The notion of stuttering is described in this context, but stuttering abstraction is not addressed. Also, no verification experiments are described in the paper.

Ray and Sumners have used a notion of refinement based on stuttering trace containment to verify concurrent programs [60]. Their focus is on functional verification and they do not consider real-time programs.

Why should the theory of Well-founded equivalence bisimulation (WEB) refinement be extend for checking equivalence of TTS? There has been a lot of previous work in developing theory and optimized techniques for WEB refinement-based verification. By extending the theory of WEB refinement to deal with TTS, the motivation is to leverage these techniques and also exploit the properties of WEB refinement in object code verification. The very nice property of WEB refinement is that, it is enough to reason about single steps of the implementation and specification. This can be exploited in object code verification, by reasoning about one instruction at a time. This property significantly reduces the verification burden. Also, the notion of timed refinement is based on the ideas of stuttering segments and stuttering transitions, which provides a natural way to abstract the implementation TTS corresponding to the object code. Without using these abstractions, the number of states and paths would explode (Section 4.5). ***For the case***

studies verified, abstractions based on stuttering reduce the size of the implementation TTS by at least 4 orders of magnitude (Section 4.5).

8.2. Timed Stuttering Abstraction

Hirshfeld and Rabinovich [35] have presented a logic to specify properties of real time high-level models that may be discrete or continuous. Decidability of this logic is the main focus in [35]. They do not consider verification methods for these properties and as such no abstractions or experiments for verification are presented. In contrast, the focus of our work is the verification of object code programs. Bouyer *et al.* [10] have proposed an expressive temporal logic for specifying properties of real-time models. The properties can then be checked using a model checker. They do not consider stuttering or stuttering abstraction. Model checking is very popular in checking correctness of higher-level models. However, model checking suffers from state explosion problem when dealing with low-level artifacts such as object code. Our approach is developed on refinement-based verification, which has been shown to be very scalable in formally checking low-level design implementations such as object code against high-level models. As such, model checking and refinement can be thought of as complimentary. Model checking is useful in verifying higher-level models and refinement can then be used to verify lower-level implementations against these higher-level models.

Namjoshi [51] have given a characterization of stuttering bisimulation, a notion of correctness that defines what it means for two transition systems to be equivalent. As the name suggests, stuttering is accounted for. The theory of equivalence we use for verification is based on a variation/extension of this characterization. Groote and Wijs [32] have presented an algorithm for checking equivalence between two transition systems that accounts for stuttering. The theory of equivalence used is based on bisimulation. Ray and Sumners have used a notion of refinement based on stuttering trace containment to verify concurrent programs [60]. Jain and Manolios [38] have developed a refinement-based testing method for functional correctness of hardware and low-level software. Delahaye *et al.* [23] have introduced some abstractions in the context of probabilistic automata, which are automata where probabilities are assigned to non-deterministic transitions. They have introduced the notion of stuttering in this context but have not discussed the idea of stuttering-based abstractions. For these works, the focus is on functional verification and they do not consider real-time programs and they do not consider stuttering-based abstraction.

Jabeen *et al.* [36] have shown how to use timed well-founded simulation (TWFS) refinement for verification of real-time FPGA. Their method does not use timed stuttering abstraction (TSA), but relies on manually generated invariants that characterize the reachable states of the FPGA design. Such an approach will not work for object code as the number of instructions will be very large and manually characterizing the reachable states of a program will not be feasible.

Etessami [27] and Dax *et al.* [19] have proposed specification languages for expressing stuttering-invariant properties, which are properties that do not distinguish behaviors of systems that differ only due to stuttering. The properties are verifiable using a model checker. Our work is complimentary to their approach, in that our goal is to exploit stuttering through abstractions to make verification more efficient and scalable.

Nejati *et al.* [52] have presented a similar idea, also called stuttering abstraction. They have proposed this idea in the context of 3-valued model checking, but do not consider real-time systems. De-Leon and Grumberg [20] and Deka *et al.* [22] have both proposed abstraction-based on stuttering equivalence in the context of model checking. Their abstraction technique is applicable to sequences of transitions that have the same atomic propositions of interest to the property being verified. Our abstraction is applicable to sequences of transitions that map to the same specification transition state (TS) state. We have also shown that our abstraction is correct in the context of TWFS refinement-based verification. Model checking works well for verification of high-level models, but suffers from state space explosion when used for low-level design artifacts such as object code. Refinement-based verification, which is a general form of equivalence verification is known to scale well for low-level design artifacts. As can be seen from the experimental results, the TTSs of the object code that we verify have millions of states and transitions. The case studies in [20] each have less than 50 states and the number of states in the case studies verified in [22] have not been mentioned.

Furia *et al.* [28] have presented a survey on how real-time systems are modeled and analyzed. They have explained what stuttering means for real-time systems. They have described some abstraction techniques that are used, but they have not talked about stuttering abstraction. Comparison of the various formalisms are presented in the paper. No verification results are presented. In contrast, our paper describes stuttering abstraction and how it can be fruitfully applied to object code verification.

TSA helps bridge this gap between object-code and higher-level models.

8.3. Refinement Checker and Liveness Detection

Jabeen et al [37] have used a theory of refinement for the verification of FPGA-based stepper motor control using proof obligations. Manually developing proof obligations for real-time interrupt driven object code is time consuming because of the size of the instructions and may introduce human errors. In contract, our paper describes abstraction based on stuttering and how its application can reduce the state space and in turn the verification time that checks for safety and liveness for real-time object code.

Failures-Divergences Refinement (FDR) [30] is a refinement checking software tool, designed to check formal models expressed in communicating sequential processes (CSP). This tool uses CSP operators with functional programming language. However, this tool does not verify object code programs.

In Chapter 4, the WEB refinement proof obligations were generated manually for the object code and discharged using an SMT solver. The goal now is to develop a procedure/tool that given the object code and specification TTS as input, will generate the TTS corresponding to the object code and check that the implementation TTS is a WEB refinement of the specification TTS. We are not aware of any previous work on procedures/tools that can check WEB refinement given the object code. Since WEB refinement is a local property, *i.e.*, , can be verified by reasoning about single transitions of the implementation/specification, we will develop a procedure that will exploit this property. Conceptually, the procedure will iterate through the implementation transitions and check the WEB refinement verification conditions. Since WEB is an equivalence relation, a prequel to checking verification conditions is to partition implementation TTS states and specification TTS states into equivalence classes. Each equivalence class will consist of one specification state and all the implementation states that map to that specification state under the refinement map.

9. CONCLUSIONS

Safety-critical applications such as airplane control systems, nuclear control systems, rockets, and many more have software codes whose failure could lead to disastrous consequences like environmental damage, monetary, and human loss. Therefore, real-time safety-critical applications need hard timing constraints that are required to be met. Some of the real-time safety-critical applications can be used in the area of embedded devices like medical, surgical robots, and microprocessors. These applications are usually programmed in high-level languages like C, Java, and Python, known as source codes. Normally, errors are checked on the source code. However, the source code is compiled/translated to object code and then executed on the device. During the compilation/translations, errors may be introduced into the object code. Therefore, it is not only necessary to ensure error-free source code, but also, it is important to check for errors in the object code. Verification of object code programs is a hard problem because object code is low level, real-time and interrupt-driven.

In this dissertation, an automated tool called timed refinement has been developed that can verify real-time interrupt-driven object code programs. Timed refinement is a notion of equivalence between two timed transition systems (TTSs) that checks for functional and timing properties. Timed refinement incorporates stuttering and refinement maps. Stuttering is a phenomenon where multiple but a finite number of transitions of low-level object code can match a single transition of the high-level model. The use of refinement maps helps in building a relationship between low-level object code against high-level models. The low-level object code is known as an implementation model and the high-level model is known as the specification model.

The automated tool has been developed in three stages. In the first stage, a procedure called timed refinement has been developed. It has been formalized and its correctness proof has been developed. The models were developed and the functional verification has been established manually. Once the functional verification has been performed, timed refinement verification has been performed using the procedure. This procedure has been verified using six case studies from the interrupt-driven motor control object code programs. The procedure has a complexity of $\mathcal{O}(|R_I|^3)$ where R_I is the number of transitions in the implementation model. To achieve better

results for the case studies that have been verified, abstractions based on stuttering has been applied ***manually***, which reduced the size of the implementation TTS by at least 4 orders of magnitude.

Based on the results obtained from the timed refinement procedure (first stage of the dissertation), the idea of ***automating*** abstraction techniques have emerged. Consequently, in the second stage of the dissertation, a novel procedure called dynamic timed stuttering abstraction (TSA) is developed. TSA can be applied to timed transition systems, which represent the real-time object code. A procedure called dynamic TSA has been demonstrated by using thirty case studies, which include stepper motor control and infusion pump control. Infusion pump control program is an ongoing commercial issue with many recent Food and Drug Administration (FDA) recalls on infusion pumps due to software errors. Verification results demonstrate the effectiveness of dynamic TSA as its application reduces the number of transitions of the implementation from millions to less than 100 for all benchmarks. As expected, most of the time is spent in performing dynamic TSA. But, once abstraction is applied, functional and timing verification times are found to be very efficient. The reason behind achieving efficient verification results in the second stage is due to the novelty in the procedure that combines dynamic TSA with symbolic simulation. In contrast to our procedure, performing symbolic simulation first followed by dynamic TSA will not work as the object code timed transition system before abstraction is so enormous that it will lead to memory issues. Other approaches that have been previously employed in literature have not targeted millions of transitions in real-time object code. Therefore, this study is a significant contribution to the literature.

From the definition of timed refinement, it can be seen that timed refinement is always preceded by functional verification. This has given a reason to develop an automated refinement checker as the final stage of the dissertation. Having an automated procedure is essential to ease the verification of large-scale programs. Two new procedures have been developed which check the following properties for timed transition systems: 1) safety and 2) liveness. Safety informally means that if the implementation makes progress, the result of that progress satisfies the specification requirements. Liveness verification checks for deadlock errors in the object code. The procedures have been implemented and the automated tool flow has been applied to several object code programs to demonstrate the effectiveness of the approach. The results indicate that the automated procedure can be applied to the verification of large-scale programs with ease, and

reduced the verification time. The reason for these fruitful results is due to the application of stuttering abstraction on the implementation model which reduces the number of transitions for all the benchmarks. The effectiveness of the refinement checker along with the strongly connected component (SCC) checking has been demonstrated from the verification results. The reduced size of the implementation model enables detecting and correcting the errors very easily.

All the algorithms, given an implementation model and a specification model, together have resulted in a comprehensive verification tool that can perform timed refinement. This verification tools can be applied several times across the various stages in the development process of real-time interrupt-driven object code for safety-critical embedded applications.

9.1. Future Works

Proposed future research directions include, but not limited to the following:

1. Explore static-timed stuttering abstraction for real-time object code verification before simulation: It has been observed in this study that dynamic timed stuttering abstraction applied during the symbolic simulation of the code reduces the memory issues. In order to reduce the abstraction time, static TSA needs to be explored in detail. The idea with static TSA is to apply TSA to the object code statically, before any simulation, i.e., to identify sequences of instructions that will only lead to stuttering transitions in the object code timed transition system and combine these instruction sequences to one complex operation. The hypothesis with static TSA is that, if abstractions can be applied to the code before simulation, this will greatly reduce the number of dynamic instructions to be simulated and therefore drastically reduce the time for computing the abstract timed transition system.

2. Partial order reduction (POR) techniques: For real-time applications, the implementation model is represented as the object code. The object code size is initially small since the loops are not unwinded. During symbolic simulation, the loops in the object code are unwinded which increase the state space of the system. For applications that have external interrupts, the tree that is used to represent the flow of the object code expands exponentially in both directions thus increasing the state space and memory space. Applying stuttering abstraction is not easy in such cases since the basic idea for abstraction is that the object should have only one incoming and one outgoing transition. Therefore, there is a need for better abstraction

techniques that can be applied on the path of the tree. This situation normally occurs for non-deterministic applications. One potential solution that can be explored as abstraction technique is partial order reduction technique, which is a technique used to reduce the size of state space in model checking. POR needs to be explored for real-time interrupt-driven object code programs.

3. Developing formal specification models from natural language requirements: One of the crucial challenges in applying refinement-based verification to commercial devices is the availability of formal specifications. For commercial devices, typically, the specification of a device is given as natural language requirements. There are many approaches toward transforming natural language requirement to formal specifications, however, none of the approaches are targeted towards refinement-based verification. Hence, there is a need for formal specification models.

REFERENCES

[1] ALARIS Medical Systems, Inc. *Directions for use. Pump Module, 8100 series.* ALARIS Medical Systems, Inc., 10221 Wateridge Circle, San Diego, CA 92121 USA, 2004. https://www.medonegroup.com/pdf/manuals/userManuals/Alaris-Medley-Pump-Module-8100-version-7.pdf.

[2] Rajeev Alur, Costas Courcoubetis, and Thomas A. Henzinger. The observational power of clocks. In Bengt Jonsson and Joachim Parrow, editors, *CONCUR '94, Concurrency Theory, 5th International Conference, Uppsala, Sweden, August 22-25, 1994, Proceedings*, volume 836 of *Lecture Notes in Computer Science*, pages 162–177. Springer, 1994.

[3] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[4] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside microsoft. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *Integrated Formal Methods, 4th International Conference, IFM 2004, Canterbury, UK, April 4-7, 2004, Proceedings*, volume 2999 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2004.

[5] Alan Baptista. Report: Software failures cost $1.1 trillion in 2016, 2017. http://servicevirtualization.com/report-software-failures-cost-1-1-trillion-2016/.

[6] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The satisfiability modulo theories library (smt-lib), 2016. http://smtlib.cs.uiowa.edu/.

[7] BBCNews. Fiat chrysler recalls 1.25m trucks over software error, May 2017. http://www.bbc.com/news/technology-39898319.

[8] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella Béguelin. Formal verification of smart contracts: Short

paper. In Toby C. Murray and Deian Stefan, editors, *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2016, Vienna, Austria, October 24, 2016*, pages 91–96. ACM, 2016.

[9] Abdeldjalil Boudjadar, Jean-Paul Bodeveix, and Mamoun Filali. Compositional refinement for real-time systems with priorities. In Ben C. Moszkowski, Mark Reynolds, and Paolo Terenziani, editors, *19th International Symposium on Temporal Representation and Reasoning, TIME 2012, Leicester, United Kingdom, September 12-14, 2012*, pages 57–64. IEEE Computer Society, 2012.

[10] Patricia Bouyer, Nicolas Markey, Joël Ouaknine, and James Worrell. On expressiveness and complexity in real-time model checking. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations*, volume 5126 of *Lecture Notes in Computer Science*, pages 124–135. Springer, 2008.

[11] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. KRONOS: A model-checking tool for real-time systems (tool-presentation for FTRTFT '98). In Anders P. Ravn and Hans Rischel, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems, 5th International Symposium, FTRTFT'98, Lyngby, Denmark, September 14-18, 1998, Proceedings*, volume 1486 of *Lecture Notes in Computer Science*, pages 298–302. Springer, 1998.

[12] Chris Bubinas. Toyota recalls 1.9 million priuses due to software error, March 2014. http://blog.klocwork.com/coding-standards/toyota-recalls-1-9-million-priuses-due-to-software-error/.

[13] CareFusion. *Alaris PC Unit, Alaris Pump Module, Technical Service Manual.* CareFusion, San Diego, California, USA, 12 2010. http://www.santoslab.org/pub/open-pca-pump/resources/PCA-product-Alaris-8000-8015-and-8100-Service-Manual.pdf.

[14] Robert N. Charette. Nissan recalls nearly 1 million cars for air bag software fix, March 2014. http://spectrum.ieee.org/riskfactor/transportation/safety/nissan-recalls-nearly-1-million-cars-for-airbag-software-fix".

[15] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[16] Telco Intercontinental Corp. Features and uses of stepper motors from telcomotion, 2014. http://www.telcointercon.com/stepper-motors-10.html.

[17] Toyota Motor Corporation. Toyota Economic Loss Settlement, 2018. http://www.toyotaelsettlement.com/.

[18] Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Timed I/O automata: a complete specification theory for real-time systems. In Karl Henrik Johansson and Wang Yi, editors, *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2010, Stockholm, Sweden, April 12-15, 2010*, pages 91–100. ACM, 2010.

[19] Christian Dax, Felix Klaedtke, and Stefan Leue. Specification languages for stutter-invariant regular properties. In Zhiming Liu and Anders P. Ravn, editors, *Automated Technology for Verification and Analysis, 7th International Symposium, ATVA 2009, Macao, China, October 14-16, 2009. Proceedings*, volume 5799 of *Lecture Notes in Computer Science*, pages 244–254. Springer, 2009.

[20] Hana De-Leon and Orna Grumberg. Modular abstractions for verifying real-time distributed systems. *Formal Methods in System Design*, 2(1):7–43, 1993.

[21] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[22] Jatindra Kumar Deka, S. Chaki, Pallab Dasgupta, and P. P. Chakrabarti. Abstractions for model checking of event timings. In *Proceedings of the 2001 International Symposium on Circuits and Systems, ISCAS 2001, Sydney, Australia, May 6-9, 2001*, pages 125–128. IEEE, 2001.

[23] Benoît Delahaye, Kim G. Larsen, and Axel Legay. Stuttering for abstract probabilistic automata. *The Journal of Logic and Algebraic Programming*, 83(1):1–19, 2014.

[24] David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Védrine. Towards an industrial use of fluctuat on safety-critical avionics software. In *International Workshop on Formal Methods for Industrial Critical Systems*, pages 53–69. Springer, 2009.

[25] Design Automation Conference Blog. History of formal verification at intel, 2018. https://dac.com/blog/post/history-formal-verification-intel.

[26] Mohana Asha Latha Dubasi, Sudarshan K. Srinivasan, and Vidura Wijayasekara. Timed refinement for verification of real-time object code programs. In Dimitra Giannakopoulou and Daniel Kroening, editors, *Verified Software: Theories, Tools and Experiments - 6th International Conference, VSTTE 2014, Vienna, Austria, July 17-18, 2014, Revised Selected Papers*, volume 8471 of *Lecture Notes in Computer Science*, pages 252–269. Springer, 2014.

[27] Kousha Etessami. Stutter-invariant languages, omega-automata, and temporal logic. In Nicolas Halbwachs and Doron A. Peled, editors, *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings*, volume 1633 of *Lecture Notes in Computer Science*, pages 236–248. Springer, 1999.

[28] Carlo A. Furia, Dino Mandrioli, Angelo Morzenti, and Matteo Rossi. Modeling time in computing: A taxonomy and a comparative survey. *ACM Comput. Surv.*, 42(2):6:1–6:59, 2010.

[29] Simson Garfinkel. Histroryś worst software bugs, November 2005. https://www.wired.com/2005/11/historys-worst-software-bugs/.

[30] Thomas Gibson-Robinson, Philip J. Armstrong, Alexandre Boulgakov, and A. W. Roscoe. FDR3 - A modern refinement checker for CSP. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International*

Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings, volume 8413 of *Lecture Notes in Computer Science*, pages 187–201. Springer, 2014.

[31] Jens Chr. Godskesen, Kim Guldstrand Larsen, and Arne Skou. Automatic verification of real-time systems using epsilon. In Son T. Vuong and Samuel T. Chanson, editors, *Protocol Specification, Testing and Verification XIV, Proceedings of the Fourteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification, Vancouver, BC, Canada, 1994*, volume 1 of *IFIP Conference Proceedings*, pages 323–330. Chapman & Hall, 1994.

[32] Jan Friso Groote and Anton Wijs. An o(m log n) algorithm for stuttering equivalence and branching bisimulation. *CoRR*, abs/1601.01478, 2016.

[33] Nick Harley. 10 of the most costly software errors in history, May 2014. https://raygun.com/blog/10-costly-software-errors-history/.

[34] Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. Timed transition systems. In J. W. de Bakker, Cornelis Huizing, Willem P. de Roever, and Grzegorz Rozenberg, editors, *Real-Time: Theory in Practice, REX Workshop, Mook, The Netherlands, June 3-7, 1991, Proceedings*, volume 600 of *Lecture Notes in Computer Science*, pages 226–251. Springer, 1991.

[35] Yoram Hirshfeld and Alexander Moshe Rabinovich. Logics for real time: Decidability and complexity. *Fundamenta Informaticae*, 62(1):1–28, 2004.

[36] Shaista Jabeen, Sudarshan K. Srinivasan, and Sana Shuja. Formal verification methodology for real-time field programmable gate array. *IET Computers & Digital Techniques*, 11(5):197–203, 2017.

[37] Shaista Jabeen, Sudarshan K. Srinivasan, Sana Shuja, and Mohana Asha Latha Dubasi. A formal verification methodology for fpga-based stepper motor control. *Embedded Systems Letters*, 7(3):85–88, 2015.

[38] Mitesh Jain and Panagiotis Manolios. An efficient runtime validation framework based on the theory of refinement. *CoRR*, abs/1703.05317, 2017.

[39] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir A. Frolov, Erik Reeber, and Armaghan Naik. Replacing testing with formal verification in intel coretm i7 processor execution engine validation. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 414–429. Springer, 2009.

[40] BaekGyu Kim, Anaheed Ayoub, Oleg Sokolsky, Insup Lee, Paul L. Jones, Yi Zhang, and Raoul Praful Jetley. Safety-assured development of the GPCA infusion pump software. In Samarjit Chakraborty, Ahmed Jerraya, Sanjoy K. Baruah, and Sebastian Fischmeister, editors, *Proceedings of the 11th International Conference on Embedded Software, EMSOFT 2011, part of the Seventh Embedded Systems Week, ESWeek 2011, Taipei, Taiwan, October 9-14, 2011*, pages 155–164. ACM, 2011.

[41] Thomas E. Kissell. *Industrial Electronics: Applications for Programmable Controllers, Instrumentation & Process Control, and ELectrical Machines & Motor Controls.* Prentice Hall, 2 edition, July 1999.

[42] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *STTT*, 1(1-2):134–152, 1997.

[43] Nancy G Leveson and Clark S Turner. An investigation of the therac-25 accidents. *IEEE Computer*, 26:18–41, July 1993.

[44] Panagiotis Manolios. Correctness of pipelined machines. In Warren A. Hunt Jr. and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings*, volume 1954 of *Lecture Notes in Computer Science*, pages 161–178. Springer, 2000.

[45] Panagiotis Manolios. *Mechanical Verification of Reactive Systems.* Phd thesis, University of Texas at Austin, August 2001.

[46] Panagiotis Manolios. A compositional theory of refinement for branching time. In Daniel Geist and Enrico Tronci, editors, *Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, October 21-24, 2003, Proceedings*, volume 2860 of *Lecture Notes in Computer Science*, pages 304–318. Springer, 2003.

[47] Panagiotis Manolios, Sudarshan K. Srinivasan, and Daron Vroon. BAT: The Bit-level Analysis Tool. 2006. Available from http://www.ccs.neu.edu/home/pete/bat/index.html.

[48] Panagiotis Manolios, Sudarshan K. Srinivasan, and Daron Vroon. BAT: The bit-level analysis tool. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 303–306. Springer, 2007.

[49] Scott Matteson. Report: Software failures cost $1.7 trillion in financial losses in 2017, 2018. https://www.techrepublic.com/article/report-software-failure-caused-1-7-trillion-in-financial-losses-in-2017.

[50] Tech Musings. 10 historical software bugs with extreme consequences, March 2009. http://royal.pingdom.com/2009/03/19/10-historical-software-bugs-with-extreme-consequences/.

[51] Kedar S. Namjoshi. A simple characterization of stuttering bisimulation. In S. Ramesh and G. Sivakumar, editors, *Foundations of Software Technology and Theoretical Computer Science, 17th Conference, Kharagpur, India, December 18-20, 1997, Proceedings*, volume 1346 of *Lecture Notes in Computer Science*, pages 284–296. Springer, 1997.

[52] Shiva Nejati, Arie Gurfinkel, and Marsha Chechik. Stuttering abstraction for model checkin. In Bernhard K. Aichernig and Bernhard Beckert, editors, *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005), 7-9 September 2005, Koblenz, Germany*, pages 311–320. IEEE Computer Society, 2005.

[53] J. O'Leary, X. Zhao, R. Gerth, and C. H. Seger. Formally verifying IEEE compliance of floating-point hardware. *Intel Technical Journal, Q1'99*, 1999.

[54] Jose Pagliery. Your car is a giant computer and it can be hacked, June 2014. http://money.cnn.com/2014/06/01/technology/security/car-hack/index.html.

[55] Miroslav Pajic, Zhihao Jiang, Insup Lee, Oleg Sokolsky, and Rahul Mangharam. Safety-critical medical device development using the upp2sf model translation tool. *ACM Transactions on Embedded Computing Systems*, 13(4s):127:1–127:26, April 2014.

[56] Jiantao Pan. Software testing, 1999. https://users.ece.cmu.edu/~koopman/des_s99/sw_testing/.

[57] Evan Perez and Shimon Prokupecz. Toyota to settle justics department probe over unintended acceleration, March 2014. http://www.cnn.com/2014/03/18/us/toyota-settlement/index.html.

[58] Keil products by Arm Limited. Keil cortex-m evaluation board comparison, 2017. http://www.keil.com/arm/boards/cortexm.asp and http://www.keil.com/dd/chip/4868.htm.

[59] Alexander Moshe Rabinovich. Automata over continuous time. *Theor. Comput. Sci.*, 300(1-3):331–363, 2003.

[60] Sandip Ray and Rob Sumners. Specification and verification of concurrent programs through refinements. *J. Autom. Reasoning*, 51(3):241–280, 2013.

[61] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. End-to-end verification of processors with isa-formal. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 42–58. Springer, 2016.

[62] Karen Sandler, Lysandra Ohrstrom, Laura Moy, and Robert McVay. Killed by code: Software transparency in implantable medical devices. *Software Freedom Law Center*, July 2010.

[63] Jaclyn Trop. Toyota seeks a settlement for sudden acceleration cases, December 2013. http://www.nytimes.com/2013/12/14/business/toyota-seeks-settlement-for-lawsuits.html.

[64] U.S. Food and Drug Administration (FDA). List of Device Recalls, 2010. http://www.fda.gov/MedicalDevices/Safety/ListofRecalls/.

[65] U.S. Food and Drug Administration (FDA). Medical Device Recalls, 2010. https://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfRES/res.cfm.

[66] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions in Embedded Computing Systems*, 7(3):36:1 – 36:53, 2008.

[67] Yi Zhang, Raoul Jetley, Paul L Jones, and Arnab Ray. Generic safety requirements for developing safe insulin pump software. *Journal of Diabetes Science and Technology*, 5(6):1403–1419, 11 2011.

[68] Michael Zhivich and Robert K. Cunningham. The real cost of software errors. *IEEE Security Privacy*, 7(2):87–90, March 2009.

APPENDIX. A: LIST OF PUBLICATIONS

- Mohana Asha Latha Dubasi, Sudarshan K. Srinivasan, & Vidura Wijayasekara. (2014). Timed Refinement for Verification of Real-Time Object Code Programs. In Verified Software: Theories, Tools and Experiments (VSTTE) - 6th International Conference, Vienna, Austria, July 17-18, Revised Selected Papers, volume 8471 of Lecture Notes in Computer Science, (pp. 252-269). Springer International Publishing

- Shaista Jabeen, Sudarshan K. Srinivasan, Sana Shuja, & Mohana Asha Latha Dubasi. (2015). A Formal Verification Methodology for FPGA-Based Stepper Motor Control. Embedded Systems Letters 7(3), (pp. 85-88)

- Mohana Asha Latha Dubasi, Sudarshan K. Srinivasan, & Sana Shuja. (2018). Dynamic Timed Stuttering Abstraction for Real-Time Object Code Verification, provisionally accepted - revised and resubmitted, IET Software

- Eman Al-Qtiemat, Sudarshan K. Srinivasan, Mohana Asha Latha Dubasi, & Sana Shuja. (2018). A Methodology for Synthesizing Formal Specification Models from Requirements for Refinement-Based Object Code Verification. In Cyber-Technologies and Cyber-Systems (CYBER) - 3rd International Conference, Athens, Greece, November 18-22, (pp. 94-101). IARIA

- Naureen Shaukat, Sana Shuja, Sudarshan K. Srinivasan, Shaista Jabeen, & Mohana Asha Latha Dubasi. (2018). Static Stuttering Abstraction for Object Code Verification. In Cyber-Technologies and Cyber-Systems (CYBER) - 3rd International Conference, Athens, Greece, November 18-22, (pp. 102-106). IARIA

- Zeyad Al-Odat, Eman Al-Qtiemat, Sudarshan K. Srinivasan, Mohana Asha Latha Dubasi, & Sana Shuja. (2018). IoT-Based Secure Embedded Scheme for Insulin Pump Data Acquisition and Monitoring. In Cyber-Technologies and Cyber-Systems (CYBER) - 3rd International Conference, Athens, Greece, November 18-22, (pp. 90-93). IARIA

- Mohana Asha Latha Dubasi, Sudarshan K. Srinivasan, & Sana Shuja. (2018). Refinement Checker for Real-Time Object Code Verification, submitted to Hindawi Journal of Electrical and Computer Engineering.

APPENDIX. B: SOURCE CODE EXAMPLE

This is an example of the source code in C language for full stepping stepper motor in clockwise direction with Repetitive Interrupt Timer (RIT). The interrupts are enabled using the timer.

```
/*****************************************************************************
Stepper motor - Full stepping with RIT interrupt in clockwise direction
*****************************************************************************/
#include "lpc17xx.h"}

 #define RITENCLR       (1UL<<1)
 #define RITEN          (1UL<<3)
 #define RITENBR      (1UL<<2)
 #define ENABLE_PCRIT     (1UL<<16)

unsigned int i = 0;
unsigned int step [] = {28,29,30,31};

void  RIT_IRQHandler (void) {
  LPC_GPIO1->FIOPIN = (1<<step[i]);

  // clear the interrupt flag
  LPC_RIT->RICTRL |= 1;

  // step i in the order 0 -> 1 -> 2 -> 3 -> 0 ->...
  if(i==3)i=0;
  else i++;
}

int main () {
  SystemInit();
/*****************************************************************
Setup the Repititive Interrupt Timer for the stepper motor
*****************************************************************/
```

```
30    LPC_GPIO1->FIODIR |= (0xF0000000);
31
32    // clear the bits 31 to 28
33    LPC_GPIO1->FIOPIN &= ~(0xF0000000);
34
35     /* enable RIT in PCON register */
36     LPC_SC->PCONP |= ENABLE_PCRIT;
37
38    // clear the CTRL register //
39    LPC_RIT->RICTRL = 0;
40
41    // set the compare value //
42    LPC_RIT->RICOMPVAL = 1500000;
43
44    // clear the counter register //
45    LPC_RIT->RICOUNTER = 0;
46
47    // set the CTRL register //
48     LPC_RIT->RICTRL = RITENCLR| RITENBR| RITEN;
49
50    // enable the RIT interrupts
51    NVIC_EnableIRQ(RIT_IRQn);
52    // loop forever
53    while(1) {
54    }
55    return 0;
56 }
```

APPENDIX. C: OBJECT CODE EXAMPLE

This is an example of the object code for full stepping stepper motor in clockwise direction with Repetitive Interrupt Timer (RIT). The interrupts are enabled using the timer. Also with this code additional information regarding the registers, special registers, memory of the microprocessor needs to be given as input. This code has not been unwinded.

```
0x00000220  480E
0x00000222  490F
0x00000224  6809
0x00000226  F8101021
0x0000022A  2001
0x0000022C  4088
0x0000022E  490D
0x00000230  6348
0x00000232  480D
0x00000234  7A00
0x00000236  F0400001
0x0000023A  490B
0x0000023C  7208
0x0000023E  4808
0x00000240  6800
0x00000242  2803
0x00000244  D103
0x00000246  2000
0x00000248  4905
0x0000024A  6008
0x0000024C  E004
```

```
0x0000024E  4804
0x00000250  6800
0x00000252  1C40
0x00000254  4902
0x00000256  6008
0x00000258  4770
0x00000270  4813
0x00000272  6A00
0x00000274  F0404070
0x00000278  4911
0x0000027A  6208
0x0000027C  4608
0x0000027E  6B40
0x00000280  F0204070
0x00000284  6348
0x00000286  480F
0x00000288  6800
0x0000028A  F4403080
0x0000028E  490D
0x00000290  39C4
0x00000292  F8C100C4
0x00000296  2000
0x00000298  490B
0x0000029A  7208
0x0000029C  480B
0x0000029E  6008
0x000002A0  2000
0x000002A2  60C8
0x000002A4  200E
```

```
0x000002A6  7208
0x000002A8  201D
0x000002AA  2101
0x000002AC  4081
0x000002AE  0942
0x000002B0  0092
0x000002B2  F10222E0
0x000002B6  F8C21100
0x000002BA  BF00
0x000002BC  BF00
0x000002BE  E7FE
```

An input of this kind would not make any sense to the automated tool. Additional information of the kind where should the simulation start has to be given.