Symbolic Timing Analysis of Real-Time Systems

Symbolic Timing Analysis of Real-Time Systems

By
Mark H. Pavlidis, B.Eng & Mgt.

A Thesis
Submitted to the School of Graduate Studies
in partial fulfilment of the requirements for the degree of

Master of Applied Science
Department of Computing and Software
McMaster University

Abstract

Timing analysis of a real-time control program is often required to verify that the system meets timing requirements. For example, if a real-time control program responds too slowly or too quickly, then the system may become unstable and fail. Traditional methods to determine timing bound estimates are often restrictive, labour-intensive, and error-prone. This thesis proposes an automated method of obtaining best- and worst-case timing bounds on unstructured assembly code without the need for manual annotation of loop or recursive call bounds. A prototype tool suite takes an assembly program as input and then generates the static control-flow graph. The generated static control-flow graph is then automatically translated into a timed automata model that models instruction processing times and adds variables to model the processor state. The resulting timed automata's transition relation represents the dynamic control-flow graph of the program. Fastest and slowest trace algorithms in recent prototype versions of UPPAAL, a timed automata model checker, are then used to extract tight best- and worst-case execution times of the program. The method is applied to code examples for two different low-end (i.e., no cache or pipeline) 8 and 16-bit microcontroller architectures, the PIC and IBM1800.

Acknowledgments

I am grateful for the support, guidance, and motivation of my supervisor Dr. Mark Lawford throughout the years that lead to the preparation and experience that made this work possible. Also, for presenting me with a challenging problem to solve that culminated in the tools and methods herein.

I'd like to acknowledge the financial support from McMaster University's Department of Computing and Software, the Natural Sciences and Engineering Research Council of Canada, Communication and Information Technology Ontario, and Ontario Power Generation.

Thanks to all my colleagues and friends for their support in of my research efforts, the constructive criticism, and thoughtful suggestions. Having an outlet to verbalise my ideas to better understand my thoughts helped to solidify solutions to problems, regardless if anything I said was understood. Further, I greatly appreciate the family and friends that provided me with non-research related support and balance in life when I needed it most.

Finally, I would like give a special thank you to my wife and my parents for their unwavering encouragement and support throughout my many, many years in school.

Contents

Contents vii

List of Figures xi

List of Tables xiii

1 Introduction 1
- 1.1 Motivation . . . 2
  - 1.1.1 The Reverse Engineering Project . . . 2
  - 1.1.2 Timing Analysis Difficulties . . . 3
  - 1.1.3 Timing Bound Uses . . . 4
- 1.2 Related Work . . . 4
  - 1.2.1 Timing Analysis Tool . . . 4
  - 1.2.2 Control Flow Graph Tool . . . 5
  - 1.2.3 UPPAAL . . . 5
- 1.3 Contributions . . . 8
- 1.4 Outline . . . 8

2 Preliminaries 11
- 2.1 Terminology . . . 11
  - 2.1.1 Timing Bound Properties . . . 11
- 2.2 Definitions . . . 12
  - 2.2.1 Control Flow Graph Model . . . 12
  - 2.2.2 Timed Automata . . . 14

3 Timing Analysis Overview 17
- 3.1 Dynamic Timing Analysis . . . 17
  - 3.1.1 Hardware Measurements . . . 17
  - 3.1.2 Software Measurements . . . 18
  - 3.1.3 Dynamic Timing Analysis Feasibility . . . 19
- 3.2 Static Timing Analysis . . . 19

3.2.1 Flow Analysis . . . 20
3.2.2 Low-Level Analysis . . . 21
3.2.3 Calculation . . . 22
3.3 Execution Time Calculation Methods . . . 22
3.3.1 Path-based . . . 22
3.3.2 Tree-based . . . 24
3.3.3 Implicit Path Enumeration Technique . . . 24
3.4 WCET Tools . . . 25

4 Timing Analysis by Timed Automata Model 31
4.1 Related Work . . . 31
4.1.1 Timed Automata . . . 32
4.1.2 Model Checking . . . 32
4.1.3 Model Checking Implementations . . . 33
4.1.4 Timing Analysis by Model Checking . . . 35
4.2 Method Overview . . . 36
4.2.1 Control Flow Analysis . . . 37
4.2.2 Transformation to a Timed Automata . . . 38
4.2.3 Data Flow Analysis . . . 39
4.2.4 Calculating Timing Bounds . . . 41

5 A Timing Analysis Transformation System 43
5.1 Control Flow Graph Representation . . . 43
5.2 Timed Automata Representation . . . 44
5.3 CFG to TA Transformations . . . 45
5.3.1 Sequential Instruction . . . 46
5.3.2 Sequential Updating Instruction . . . 47
5.3.3 Non-sequential Instruction . . . 48
5.3.4 Unguarded Branching Instruction . . . 48
5.3.5 Uniform Time Guarded Branching Instruction . . . 50
5.3.6 Non-Uniform Time Guarded Branching Instruction . . . 52
5.4 Summary . . . 54

6 STARTS Tool Suite 55
6.1 Tool Suite Description . . . 55
6.1.1 Operating Environment . . . 55
6.1.2 Software Dependencies . . . 56
6.1.3 Limitations . . . 57
6.2 Using the Tool . . . 58
6.2.1 Input Source Assembler Program . . . 58

6.2.2 Selecting the Code Segment 60
6.2.3 Generating the Control-Flow Graph 60
6.2.4 Generating the Timed Automata Model 61
6.2.5 Generating BCET and WCET Traces 61
6.2.6 Trace Visualisation in UPPAAL 62

7 Timing Analysis Results **67**
7.1 Timing Analysis Results 67
7.2 IBM1800 Timing Analysis Results 67
7.3 PIC Timing Analysis Results 68

8 Conclusions and Future Work **71**
8.1 Method Benefits 72
8.1.1 Tight and Safe Lower and Upper Time Bounds 72
8.1.2 Automatic Path Determination 72
8.1.3 Concrete Execution Paths 73
8.1.4 Safety and Liveness Verification 73
8.1.5 Execution Path Visualisation 74
8.1.6 Accurate Modelling of Parallel Execution 75
8.1.7 Leveraging UPPAAL 75
8.2 Future Work 76
8.2.1 Indirect Addressing 76
8.2.2 Overflow and Out-of-Range Detection 77
8.2.3 Preemption and Interrupts 78

Bibliography **79**

A Timing Analysis for the IBM 1800 **85**
A.1 IBM1800 Overview 85
A.1.1 Architecture and Instruction Set Details 85
A.2 IBM1800 Timing Analysis Transformations 86

B Timing Analysis for the PIC Microcontroller **91**
B.1 PIC Overview 91
B.1.1 Architecture and Instruction Set Features 91
B.2 PIC Timing Analysis Transformations 92

List of Figures

1.1 Reverse Engineering Tool Suite Uses Hierarchy 3
1.2 Control Flow Graph of an IBM1800 code segment 6
1.3 UPPAAL 3.6 Screenshot - Modelling Editor 7

2.1 Timing Bound Properties 12

3.1 Triganular Loop Example 23
3.2 IPET Objective Function and Constraints 25

4.1 STARTS Tool Architecture 36
4.2 Example – IBM1800 assembly code 37
4.3 Example – Annotated Control Flow Graph 38
4.4 Example – UPPAAL Timed Automaton 40

5.1 Sequential Instruction Transformation 47
5.2 Sequential Updating Instruction Transformation 48
5.3 Non-sequential Instruction Transformation 49
5.4 Unguarded Branching Instruction Transformation 49
5.5 Uniform Time Guarded Branching Instruction Transformation 51
5.6 Return Instruction Transformation 52
5.7 Potential Deadlock on Guarded Branching Instruction Transformation 53
5.8 Non-Uniform Time Guarded Branching Instruction Transformation . 54

6.1 STARTS Tool Architecture 59
6.2 Uppaal Simulator Example 1 63
6.3 Uppaal Simulator Example 2 64
6.4 Uppaal Simulator Example 3 65

List of Tables

7.1 IBM1800 BPC Results 68

A.1 Control-flow behaviour of the MDX instruction 86

Chapter 1

Introduction

Why do we need the timing bounds of an embedded real-time program?

Embedded real-time systems are used to control many tasks in the physical world that were not previously controlled with computers. These include safety-critical tasks such as the control of nuclear power generating stations, aerospace and automotive vehicles, and telecommunication systems among others. A commonality of these tasks is that they are increasingly being implemented as software programs controlling embedded real-time systems. A system failure due to a missed timing deadline may result in catastrophic loss of human and/or economic resources. Moreover, on a volume basis, nearly all processors (up to 98%) manufactured are used in embedded systems[41]. Therefore, a great deal of research effort has gone into developing methods to verify that these systems meet functional requirements to ensure correct operation.

More recently, research on determining timing bounds of embedded systems has been pursued. The timing bounds include both the *Worse Case Execution Time* (WCET) and *Best Case Execution Time* (BCET) of the system. The timing bounds are used in the validation of the timing requirements of the systems. The timing of the execution of a real-time program is critical in determining if the functional requirements are met, because control of physical systems require that decisions must made by some *hard* real-time limit. Furthermore, a large variance in timing of control decisions may make the physical system unstable. Timing bounds are also used in schedulability analysis of the system, and determining the capabilities (and cost) of

the processor required to implement the system.

This thesis presents a method of determining the timing bounds of an embedded real-time program from assembly/object code, including the execution paths that result in the BCET and WCET of the program. The method differs from the current methods used to find timing bounds (or commonly only the WCET) of real-time programs. A prototype timing analysis tool based on the method allows for the verification of timing requirements of an implementation and can be used in determining timing requirements when reverse engineering a legacy system.

1.1 Motivation

This section provides the motivation for the development of the timing analysis tool. It includes an overview of the reverse engineering project, of which the tool is a component. Further, the current difficulties in timing analysis that we desire to overcome are presented, and the utility of determining execution timing bounds described.

1.1.1 The Reverse Engineering Project

The motivation to build a timing analysis tool is part of a larger project to obtain high-level software requirements from assembly code. The project, *Reverse Engineering High-Level Requirements from Assembly Code*, involves a group of researchers from McMaster University's Software Quality Research Laboratory (SQRL) working jointly with system engineers from Ontario Power Generation (OPG) to develop methods and a *Reverse Engineering Tool Suite* (Figure 1.1). The methods and tools are intended assist in reverse engineering legacy assembly language safety-critical real-time programs to high-level requirements. The project was funded by OPG and Communication and Information Technology Ontario (CITO), from April 2003 to April 2005.

The direction of the project presented herein was constrained by the following requirements. The reverse engineering program of interest, Boiler Pressure Control (BPC) was to be based on a non-structured assembly code (sparsely commented), and a legacy processor (IBM1800 Data Acquisition and Control System) with a limited instruction set and a simple architecture without pipeline or cache. Both best- and

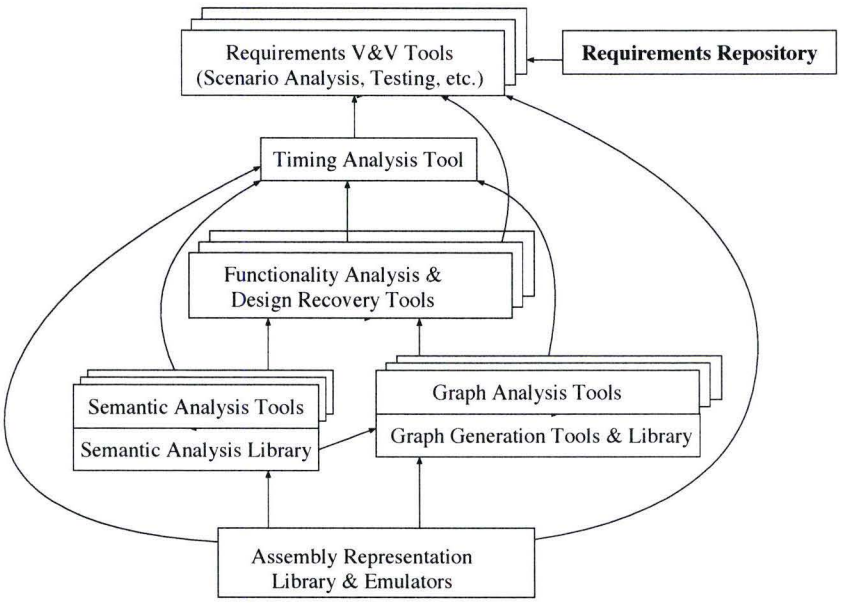

Figure 1.1: Reverse Engineering Tool Suite Uses Hierarchy

worst-case execution times are required. Finally, it was necessary to address and overcome limitations of a previous timing analysis tool by fully automating the timing analysis process.

1.1.2 Timing Analysis Difficulties

The conceptual use of WCET in scheduling algorithms for hard real-time systems has long been studied [27], but determining the actual precise execution time bounds of real-time programs is difficult, error-prone, and time consuming. More recently, research has focused on determining the WCET estimate that is a safe overestimate using static analysis of the program source or object code.

Static timing analysis methods have reduced much of the time and effort required to obtain timing bound results, but they do not entirely eliminate the human-in-the-loop required to add annotations for control flow. The required annotations include determining loop iteration bounds, branching flow and infeasible paths, and behaviour due to function calls and recursion. Some methods to automatically determine program behaviour have been developed [13], but these are restricted to special cases of

structured code and/or require high-level source code (recall the constraint of reverse engineering assembly code).

1.1.3 Timing Bound Uses

To answer the opening question *why we need to determine the timing bounds of real-time programs?*, we look at the uses of the timing bounds. First is the need to determine timing bounds that satisfy functional timing requirements and timing tolerances in reverse engineering safety-critical real-time programs to high-level requirements. Moreover, determining the timing bounds of an implementation can be used in the forward development process to validate a program implementation against functional timing requirements, and verify that the jitter is acceptable for the specified timing tolerances [43]. Finally, other uses for timing bounds include selection of sampling frequency, data rates, schedulability, hardware (i.e., processor) selection, and compiler optimisation.

1.2 Related Work

In this section, the work directly related to the development of the *Symbolic Timing Analysis of Real-Time Systems* (STARTS) tool suite is presented. It includes work previously completed for the Reverse Engineering project, and other tools used in its implementation.

1.2.1 Timing Analysis Tool

A WCET Analysis Tool (WAT) was developed by Sun [36]. The tool was developed to be the Timing Analysis Tool (TAT) component of the Reverse Engineering Tool Suite. The interactive tool consists of a path-based WCET calculation. It partially automates analysis, but it still requires intensive manual annotation to identify loops and determine their bounds, and to mark infeasible paths. Further, the traces are generated as sequential textual output.

These limitations of the WAT motivated development of a tool that automates the process to eliminate time consuming and error-prone manual annotations. It also

identified the need for a graphical visualisation of the traces as an aid in comprehension for the reverse engineering efforts. Additionally, a tool that finds both BCET and WCET is preferred to one that only computes the latter. The development of the WAT provides a methodology for obtaining possible execution traces and insight into troublesome IBM1800 instructions that complicate feasible path determination. The feasible paths are found by pruning the infeasible edges from the output of a Control Flow Graph tool.

1.2.2 Control Flow Graph Tool

Everets [14] implemented a tool for generating a static control flow graph (CFG) representing an approximation of the possible execution paths of the BPC code. The tool, Lst2Gxl, is a part of one of the lowest-level components, the Graph Analysis Tool, of the Reverse Engineering Toolset. Lst2Gxl uses the compiler generated code listing (LST) file to create a CFG with each instruction represented as a node in the graph. The graph nodes include additional annotations that contain relevant information from the assembly code that can be further used to determine the feasible dynamic execution paths. Figure 1.2 is an example of a CFG generated from a code segment of the IBM1800 assembly code. The CFG is represented in Graph eXchange Language (GXL) [19], an Extensible Markup Language (XML) sub-language designed to be a standard exchange format for graphs. The GXL-based CFG can be processed by Extensible Stylesheet Language Transformation (XSLT) [47], an XML-based language used for the transformation of XML documents, to another XML-based document. For example in Section 6.2.4, an XSLT specification is defined to transform a CFG in GXL to an XML-based timed automata model used by UPPAAL.

1.2.3 UPPAAL

UPPAAL is a graphical tool for modelling, simulation and verification of real-time systems [42], depicted in Figure 1.3. It is appropriate for systems that can be modelled as a collection of non-deterministic processes with finite control structure and real-valued clocks (i.e., timed automata), communicating through channels and/or shared data structures.

Typical application areas include real-time controllers, communication protocols,

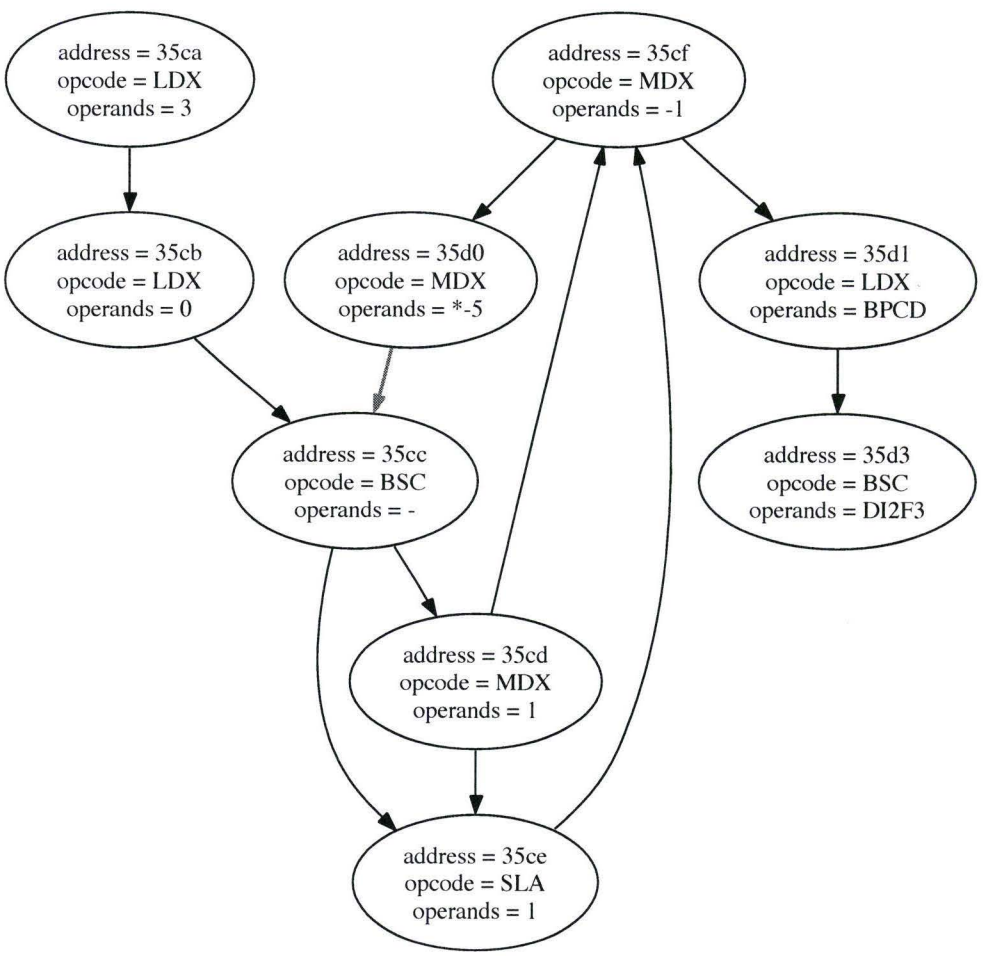

Figure 1.2: Control Flow Graph of an IBM1800 code segment

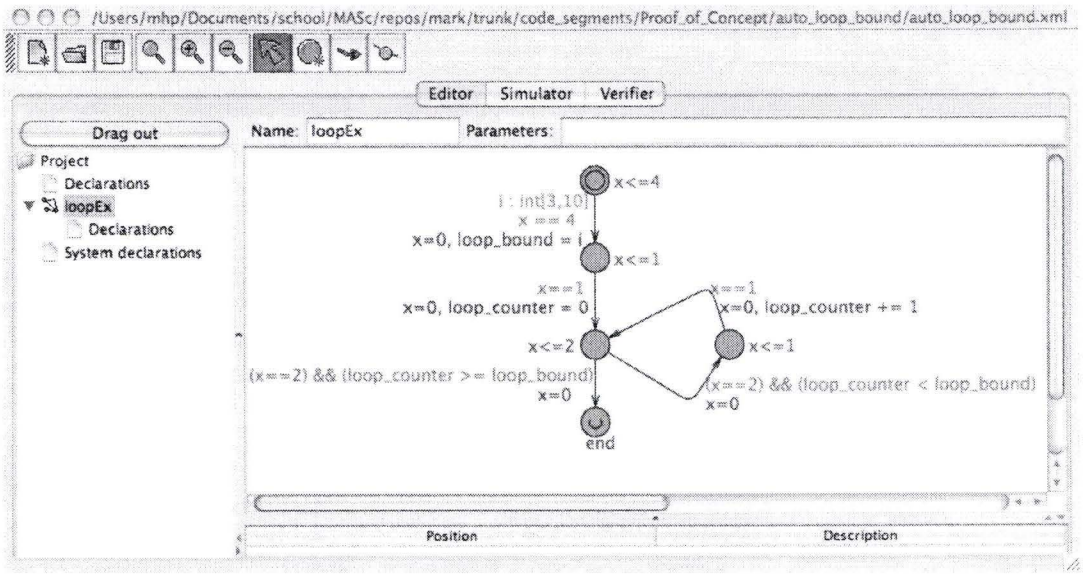

Figure 1.3: UPPAAL 3.6 Screenshot - Modelling Editor

and other systems in which timing aspects are critical. UPPAAL is a joint development between real-time system researchers at Uppsala University, in Sweden, and Aalborg University, in Denmark. It provides a model checking engine to verify safety and bounded liveness properties expressed as reachability queries [24, 3]. It was initially released in 1995, and it continues to be actively developed and supported on MS Windows, Linux, and Mac OS X platforms. Throughout the years, many notable improvements have been made to UPPAAL, including efficient data structures and algorithms, symmetry reduction, and symbolic representations that dramatically reduce computation time and memory space use in light of possibly enormous state space explosion.

The most recent stable release, version 4.0.1 (as of June 2006), includes a standalone verification engine, fastest trace generation (for BCET)[1], XML-based TA model, process priorities, progress measures, bounded integer ranges, meta variables, and user-defined functions. These features permit the modelling of microprocessor architecture and instruction execution used to perform timing analysis by the method proposed in this thesis.

[1]The slowest trace generation is currently possible in an unreleased prototype version of UPPAAL.

1.3 Contributions

In this thesis the timing analysis of real-time programs is examined and an alternative method of obtaining best- and worse-case execution times of assembly-level software is developed. The major contribution of this thesis is a new method of obtaining timing bounds that is made possible by a transformation system from a static control-flow graph to a timed automaton model of the program. The primary contribution of this method is to introduce a static timing analysis method that provides the following:

- A transformation system from static control-flow graph to timed automata model of the program and hardware architecture.
- Calculation of tight and safe timing bounds of unstructured assembly code.
- Timing bounds and respective traces are obtained automatically without the need for manual annotations.
- Timing bound and trace computation make use of pre-existing efficient optimisations of state space representation and searching provided by UPPAAL.
- A prototype implementation, the STARTS tool suite, used to develop and validated the proposed method automates the timing analysis process.
- Safety and liveness properties of the implementation can be verified, providing alternative means of validating the implementation in addition to testing.
- Traces through the timed automata model can be simulated providing a graphical visualisation of the program execution paths.

1.4 Outline

The remaining chapters of this thesis are organised as follows:

- Chapter 2 presents terminology and definitions used throughout the thesis.
- Chapter 3 provides an overview of current timing analysis methods and tools.

- Chapter 4 discusses model checking timed automata for use in timing analysis, then presents a method for generating a timed automaton model of the program.
- Chapter 5 details the transformation process of a static control-flow graph to timed automaton model of a real-time program.
- Chapter 6 describes the prototype tool suite STARTS.
- Chapter 7 presents timing analysis results for the IBM1800 and the PIC target architectures.
- Chapter 8 draws conclusions, details the benefits of the work presented and provides an overview of the possible future work to overcome the method's current limitations.

Chapter 2

Preliminaries

In this chapter we introduce the terminology and definitions used throughout the thesis.

2.1 Terminology

2.1.1 Timing Bound Properties

This section provides the terminology used to describe timing bound properties of interest. Figure 2.1 graphically demonstrates the relationship between the properties.

Worst Cast Execution Time (WCET): The slowest of all possible execution times of a program, or a program fragment. It is typically given in terms of cycles, or seconds if the CPU clock rate is known.

Best Cast Execution Time (BCET): The fastest of all possible execution times of a program, or a program fragment. It is typically given in terms of cycles, or seconds if the CPU clock rate is known.

Jitter: The largest execution time variation (i.e., the difference between WCET and BCET) of a program, or program segment.

Safe: A WCET (or BCET) estimate is *safe* if it does not underestimate (overestimate) the actual WCET (BCET).

Tight: A WCET (or BCET) estimate is *tight* when the estimate is as close to the actual WCET (BCET) as possible, but remains *safe* (i.e., the WCET (BCET) estimate is equal to actual WCET (BCET)).

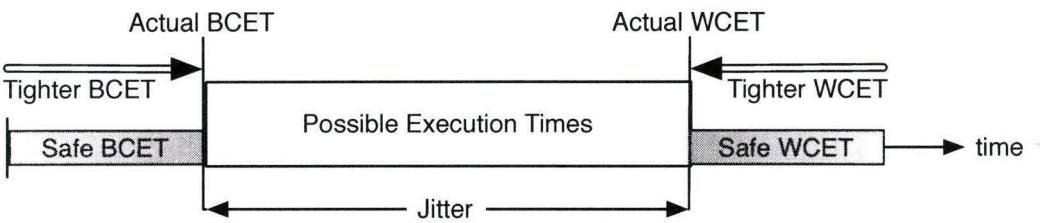

Figure 2.1: Timing Bound Properties

2.2 Definitions

2.2.1 Control Flow Graph Model

A control flow graph describes the possible execution paths through the program. Each node of the graph represents an assembly-level instruction. Each directed edge out of a node represents the next instruction that may execute. We extend the model of the CFG to include annotated nodes. The annotation of the nodes includes relevant information required to perform control and data flow analysis when converting the CFG to a timed automata.

Modelling each instruction as a separate node in the graph differs from many of the common approaches to CFG generation, that encapsulate a sequence of sequential, non-branching, non-backtracking, instructions into a *basic block*[1]. The basic block model is sufficient when determining control-flow, but in order to later model data-flow and timing effects of processors (i.e., interrupt service routines, pipelines, caches, etc.) it is necessary to model each instruction atomically.

Definition 2.1 *A **Control Flow Graph** (CFG) is a possibly cyclic directed graph given by the tuple,*

[1] A basic block is a sequence of instructions with a single entry point at the beginning and a single exit point at the end

$$G = (N, E, n_0)$$

where N is a finite set of nodes, $E \subseteq N \times N$ is the set of directed edges, n_0 is a unique start node. For regular cases, all nodes $n \in N$ are reachable from n_0.

The following definition augments a CFG to include annotated nodes. Nodes are annotated with the fields from the assembly/machine-level instruction. The annotations include the relevant fields (e.g., instruction address, opcode, operands, object code, etc.) and their respective values.

Definition 2.2 *An **Annotated Control Flow Graph** (Annotated CFG) is a possibly cyclic directed graph given by the tuple,*

$$G^A = (N, E, n_0, \lambda, I, \Sigma)$$

where N is a finite set of nodes, $E \subseteq N \times N$ is the set of directed edges, n_0 is a unique start node, $\lambda : N \rightarrow \mathcal{P}(I \times \Sigma)$ is a function that maps a node to its set of annotations, where I is the finite set of instruction fields, and Σ is the set of values for the instruction fields.

For regular cases, all nodes in N are reachable from n_0. It is possible to have a disconnected graph, where there exists a set of nodes that are not reachable from n_0. Such nodes are still included in the graph, as they may indicate a special segment of instructions or a problem with the generation of the CFG. This issue is dealt with in the transformation of the CFG to a TA, detailed in Section 5.3.5.

The Annotated CFG represents all possible execution paths of the instructions of the program. The annotations represent all relevant information for each instruction. An explicit execution path is defined by a *trace* that represents one possible sequence of instructions executed by the program.

Definition 2.3 *A **CFG Execution Trace** is any finite string of instruction nodes connected by edges from the Annotated CFG of the form $n_0 \xrightarrow{\epsilon_{(0,1)}} n_1 \rightarrow \cdots \rightarrow n_i$. Where $\epsilon_{(x,y)}$ is the directed edge from the node labelled x to the node labelled y.*

2.2.2 Timed Automata

The theory of timed automata was initially developed by Alur and Dill [5], as an extension of finite-state Büchi automata with clock variables. A timed automaton is structured as a directed graph with nodes representing locations and edges representing transitions.

Constraints on the clocks are used to restrict the behaviour of the automaton and enforce progress properties. Clock constraints on locations, called invariant conditions, force a transition when a clock value would violate the clock invariant. The transition (if one exists) is required because states where the clock invariant is violated are considered infeasible.

Transitions of the automaton have clock constraints guarding the transition edge, called triggering conditions, restricting when a transition can be taken based on the clock guard. Transitions include clock resets, where some clock variables are reset to zero when the transition is taken.

Definition 2.4 *A **Timed Automaton** is a tuple,*

$$A = (L, \mathcal{C}, l_0, E, \mathcal{I}),$$

where L is a finite set of locations, $\mathcal{C}$ is a finite set of non-negative real-valued clocks, $l_0 \in L$ is an initial location, $E \subseteq L \times \mathcal{B}(\mathcal{C}) \times 2^{\mathcal{C}} \times L$ is a set of edges labelled by guards and a set of clocks to be reset, and $\mathcal{I} : L \rightarrow \mathcal{B}(\mathcal{C})$ assigns location invariants to clocks, where $\mathcal{B}(\mathcal{C}) = \left\{ x, c, \sim \mid x \in \mathcal{C} \wedge c \in \mathbb{N} \wedge \sim \in \{<, \leq, ==, \geq, >\} : x \sim c \right\}$ is the set of guards on clocks.

Bengtsson et al. [8] provide an extension of the classical theory of timed automata to ease the task of modelling with a more expressive language. The extension adds more general data type variables (i.e., boolean, integer), in an attempt to make the modelling language closer to real-time high-level programming languages.

Definition 2.5 *An **Extended Timed Automaton** is a tuple,*

$$A_e = (L, \mathcal{C}, \mathcal{V}, \mathcal{A}, l_0, E, \mathcal{I}),$$

where L is a finite set of locations, $\mathcal{C}$ is a finite set of non-negative real-valued clocks, $\mathcal{V}$ is a set of finite data variables. $\mathcal{A}$ is the set of synchronising actions where

$\mathcal{A} = \{\alpha?|\alpha \in A\} \cup \{\alpha!|\alpha \in A\}$. $l_0 \in L$ *is an initial location,* $E \subseteq L \times \mathcal{B}(\mathcal{C}, \mathcal{V}) \times 2^{\mathcal{R}} \times L$ *is a set of edges label led by guards and a set of reset operations,* $\mathcal{R}$. *Finally,* $\mathcal{I} : L \rightarrow \mathcal{B}(\mathcal{C}, \mathcal{V})$ *assigns location invariants to clocks, where*

$$\mathcal{B}(\mathcal{C}, \mathcal{V}) = \Big\{x, i, c, \sim \mid x \in \mathcal{C} \wedge i \in \mathcal{V} \wedge c \in \mathbb{N} \wedge \sim \in \{<, \leq, ==, \geq, >\} : x \sim c \vee i \sim c\Big\}$$

is the set of guards on clocks.

Definition 2.6 *A* ***TA Trace*** *is any finite string of instruction locations connected by transitions from the Annotated TA model of the form* $l_0 \xrightarrow{\tau_{(0,1)}} l_1 \rightarrow \cdots \rightarrow l_i$. *Where* $\tau_{(x,y)}$ *is the transition from the location labelled* x *to the location labelled* y.

Chapter 3

Timing Analysis Overview

In this chapter several of the current methods to compute the timing bounds of real-time systems are presented. The benefits and limitations of each method are discussed. These methods of computing the WCET differ from the method proposed in this thesis. It should be noted that the proposed method was developed independent of the techniques of the previous work. It allowed for a fresh approach that diverges from the *status quo* to solve some of the current limitations of WCET estimation. In particular, automatically determining loop bounds, complex flow, and avoiding infeasible execution paths without manual annotation.

3.1 Dynamic Timing Analysis

In industry, WCET is commonly computed by measurements on many executions of the program code, known as dynamic timing analysis. Measurement of execution time are performed by hardware, software, or a hybrid of both types of tools.

3.1.1 Hardware Measurements

Hardware measurements use oscilloscopes and logic analysers to monitor system outputs by connecting probes to the processor and system bus pins. Oscilloscopes and logic analysers can be used to calculate the frequency of control loops (e.g., cyclic executives), and the response time from input stimulus to controlled output response.

Hardware measurement methods have minimal intrusiveness on the software being measured because probing does not affect execution time or order of execution. However, the methods can only be used on a system when the hardware setup permits the connection of the analysing probes. An example of a scenario that does not permit the use of such tools would be an embedded safety-critical system, where it is not be safe to connect probes to hardware or run test cases they may result in a system failure.

Oscilloscope measurements only provide results from externally visible signals, and cannot determine the internal state of the processor or executing program. However, the granularity of logic analysers is at the machine instruction level. For both methods, the measurements of numerous test executions are logged, and the changes of signals over time are analysed to determine the timing results. Although these methods provide the smallest timing resolutions, as with all of the dynamic WCET methods, it cannot typically guarantee safe timing bounds because in general test cases cannot be exhaustive. The latter is also true for the software measurement techniques described below.

3.1.2 Software Measurements

Software measurements involve adding instrumentation points into the source code of the program or around the program. An example of software measurement methods are function profiling tools (e.g., *gprof()* [17]) that measure the execution profile of called subroutines, providing a call graph and associated execution time. Another is using instrumentation points that drive output pins that can be measured with hardware to determine the execution time.

Unfortunately, adding instrumentation code into a real-time program changes the timing, execution path and, for complex processors with cache and pipelines, processor dynamics, of the program proper. It results in an overestimation for the timing bounds. The impact on the WCET bound is an overestimated result, that is more safe. The impact on the BCET bound is far more critical to the analysis of response jitter, since the overestimation can push the BCET bound into to the right of the safe BCET time region, as illustrated in Figure 2.1, leading to an underestimated response jitter.

Software measurements that do not add instrumentation points in the code require operating system or emulator support to obtain the start and end times of execution. The former has a high granularity (i.e., program level) and high timing resolution that is dependent on the operating platform (i.e., UNIX *time()* command). Emulation provides instruction level granularity, low timing resolution, and can provide execution traces, but development of an emulator can be costly and time consuming.

3.1.3 Dynamic Timing Analysis Feasibility

Despite its common use in industry, it is very difficult for dynamic timing analysis to ensure safe timing bound results. The methods yield timing bounds that are within the range of possible execution time (Figure 2.1). The initial state and input selection of test cases strives to push the results outward toward the actual bounds. For safety-critical real-time systems, if system response is too fast or too slow it can cause the system to enter a state that is unstable and uncontrollable, resulting in possible catastrophic failure. Thus dynamic timing analysis techniques do not provide a feasible solution to our problem with its previously stated set of constraints. The requirement to obtain safe timing bounds motivated the investigation of static timing analysis methods. More recently, dynamic measurement techniques have been used to enhance static analysis in [44] for high-end processors that cannot be effectively analysed due to computational complexity. We note that to avoid these complexities and because of market factors, we limit our analysis to the class of processor that are *low-end*, for which static timing analysis is feasible.

3.2 Static Timing Analysis

Static timing analysis involves the use of analytical methods to determine timing bounds from program code (high-, assembly-, or object-level) without executing the code. Each stage of static timing analysis provides information about a program and its execution architecture used to obtain a safe and tight (as possible) WCET. This section outlines the three stages of static WCET analysis, commonly presented in literature, and presents some timing analysis tools that provide static timing analysis.

3.2.1 Flow Analysis

The first stage of static timing analysis involves determining possible sequences of instruction execution of the program. This includes sequential blocks of instructions, branching and conditional branching instructions, and number of loop iterations. For the latter, the common approaches require manual annotation of loop iterations. This thesis presents a method that does not require manually determining loop bounds a priori. The flow analysis is represented by a CFG, where instructions are represented by nodes and the execution path is determined by the directed edges.

In cases where source code is available, the flow analysis stage may also determine called functions and recursions. Based on the constraints of the reverse engineering problem, high-level source code will not be considered in the implementation of the timing analysis tool, and it is only mentioned for completeness.

Flow analysis must determine a safe execution path approximation, that includes all feasible paths with as few infeasible paths as possible. In the method proposed in this thesis, Low-level Analysis (Section 3.2.2) of the architecture yields conditions on edges that prevent inclusion of infeasible paths in the timing bound calculation. Calculation methods that do not perform partial data flow simulation do not maintain enough of the state information to eliminate infeasible paths from the calculation of the timing bounds. This results in time bounds that are less tight. The task of flow analysis is decomposed into three steps: Extraction, Representation, and Calculation conversion.

Extraction of flow information for assembly-level code involves the identification of all possible paths for each instruction based on the operational semantics of the architecture, typically defined informally in the technical manuals of the target processor.

Once flow information is extracted, it is necessary to introduce notation to represent it. The representation of flow analysis may take the form of some type of graph [34, 37, 45] or tree [33, 10], source code annotation [22], or by defining a language to describe the possible paths through the program [31, 21]. Together with flow information, loop identification and loop iteration bounds are added to the flow representation. Commonly, loop information is annotated manually. Since manual annotation is prone to error and tedious, there have been some developments in au-

tomatically identifying loops and related iteration bounds at assembly-level code for some classes of well-structured loops [18].

The choice of representation may negatively impact the capabilities of the related Calculation method (Section 3.3) to find safe and tight timing bounds. In the final step, Calculation conversion, the flow information is mapped to a form that is a feasible input for the chosen method of computing the bounds. In addition to flow information to perform the calculation, the execution time of the underlying hardware architecture must be determined by low-level analysis.

3.2.2 Low-Level Analysis

The precise execution time of each instruction is determined at the low-level analysis stage. The target architecture instruction set defines each atomic action the processor performs, and how long each action takes. The execution time of each instruction is impacted by the complexity of the architecture. Microprocessor designers add physical complexity to increase instruction throughput with *pipelined* and out-of-order execution, and to decrease memory access delays with *cache*. We leave investigation of these types of processors to future work as they are outside of the scope of our problem definition.

Logical complexity is added the the hardware architecture with interpreted instructions, where the operator mnemonic (or opcode) alone does not determine the action of the instruction. Further, for some complex architectures (i.e., CISC-based processors like the Intel x86 line) instructions are interpreted into a set of micro-instructions encapsulated in the hardware and hidden from the instruction set. More logical complexity of the instruction set requires interpreting the instruction operator, flag bits, and operands. This is done to understand the action performed by the processor for each instruction format, and accordingly the precise execution time.

For example, the IBM1800 instruction set's precise execution time of an instruction depends on the instruction length of a single- or double-word (i.e., format bit $F = 0$ or $F = 1$, respectively), if the instruction register or index registers are used (i.e., tag bits $T = 0$ or $T = \{1, 2, 3\}$, respectively), and if indirect addressing is used. The execution time, or execution cycle count, must be determined for each case and then converted to the format used in the final stage, namely the timing

bound calculation.

3.2.3 Calculation

Once the flow and architecture timing information is determined, it is possible to compute the execution timing bounds of the program. The flow representation, architecture information, and calculation method chosen impact the tightness of the timing calculations, and the time and space complexity of the computation. Often, the method to extract and represent timing is closely related to the calculation method used. Each common type of calculation method is described in more detail in the next section. This thesis proposes a different calculation method technique to compute timing bounds in Chapter 4.

3.3 Execution Time Calculation Methods

This section provides an overview of the current set of static analysis calculation methods. It describes the capabilities and limitations of the methods to compute timing bounds given program flow and low-level architecture timing results. There are three main types of calculation methods often referenced in literature: path-based, tree-based, and implicit path enumeration technique (IPET)[1].

3.3.1 Path-based

Path-based calculation methods utilise graph based representations of program flow and timing annotation to compute execution times of paths through the graph and then search for the longest path to find the WCET.

The flow graph representation is a CFG where the timing of each instruction or basic block on the nodes, and loops are identified and annotated with iteration bounds. CFGs of this type with timing information are often called Timing Graphs (TGs). The TG is used to *explicitly* enumerate each possible execution path based on the graph, then searches for the longest. Determining the explicit path permits cal-

[1]Throughout this section, all references to timing bounds, both WCET and BCET, are referred to as WCET to maintain consistency with the literature that is primarily focused only on WCET.

culating tighter timing results due to the architecture dependent effects of execution of particular sequences of instructions.

For programs with many branches and/or loops it is obvious that the number of possible paths quickly explodes. Utilising the fact that path-based methods are manageable for limited segments of code, such as a single loop or branch path, the complexity of enumerating the paths is mitigated by decomposing the TG into a *scope graph* [12] . A scope graph is constructed from subgraphs of the TG, where each subgraph is a node representing a scope. Each scope corresponds to a loop or function contained in the TG subgraph. The WCET calculation then works hierarchically on the scopes. The longest path must verified to ensure that it is a feasible path. If it is not, the next longest path is chosen for the WCET calculation and is check for feasibility. Moreover, for unstructured code (i.e., assembly-level or optimised object code) it can be difficult to determine the scope that a node of the TG belongs to. Finally, the task of identifying loops and associated loop bounds must be performed manually or automatically, prior to the calculation phase of analysis.

Piece-wise calculation strategies, such as the scope graph, result in an overestimated (i.e., less tight) WCET if data-flow information reaches over the borders of the decomposed pieces. The problem is illustrated by considering a triangular loop, a nested loop where the number of iterations of the inner loop is dependent on the iteration count of the outer loop (see Figure 3.1). The inner loop yields a worst-case iteration bound of 10, as does the outer loop. Thus, the piece-wise calculation over the path would result in 100 executions of inner loop body, while the actual result should only be 55.

```
for(i = 0; i < 10; i++)
  for(j = i; j < 10; j++)
    body
```

Outer Loop bound: 10
Inner Loop bound: 10
Piece-wise Execution count: $10 \times 10 = 100$
Actual Execution count: $\sum_{i=0}^{10} i = 55$

Figure 3.1: Triganular Loop Example

Path-based calculation methods are useful to obtain explicit execution traces for timing bounds but are limited by the complexity of the analysed program, yielding

safe but less tight results.

3.3.2 Tree-based

Tree-based calculation generates the WCET by a bottom-up traversal of a syntax-tree of the program. The syntax-tree is a representation of the program where the nodes describe the structure of the program at the source-code level, and the leaves represent basic blocks.

The syntax-tree is derived from the flow representation (i.e., timing graph). Timing is computed by translating each node into an equation that expresses the timing of its children nodes, then summing the expressions following a set of rules for traversing the tree.

Tree-based calculation was first introduced by Park and Shaw [32], as *timing schema*. It was further extended to include hardware level timing influences, such as pipelines and cache. It is a simple and efficient method to compute WCET, but it requires source-level code or highly structured assembly-code. Some instruction sets, such as the IBM1800 or PIC microcontroller, are inherently unstructured. That is, branches and loops are not easily identifiable by mechanical or manual methods. Further, compiler optimisations of object-code results in unstructured code. Therefore, tree-based calculation is unsuitable for the constraints imposed upon the work of this thesis, but is feasible in specific cases of small and well-structured source-code.

3.3.3 Implicit Path Enumeration Technique

Due to the space and time complexity issues encountered with the previous calculation methods, Implicit Path Enumeration Technique (IPET), as the name suggests, does not find every execution path to determine the WCET. The methods initially developed in [26, 30, 34] state the problem of finding the WCET by maximising a sum that is restricted by a set of constraints modelling program flow and hardware timing (Figure 3.2).

The WCET is found either by constraint solving methods, or more popularly, by integer linear programming (ILP). The advantages of this method is that each path does not have be explicitly determined. Instead the behaviour is expressed as a set of constraints. More importantly, the calculation stage of finding the WCET

$$WCET = max(\sum_{i \in BB} x_i * t_i),$$

$$\{\text{a set of constrains on } \forall_{i \in BB} x_i\},$$
$$\textit{where } BB \textit{ is the set of all basic blocks,}$$
$$x_i \textit{ is the execution count of the basic block,}$$
$$t_i \textit{ is its execution time.}$$

Figure 3.2: IPET Objective Function and Constraints

is performed by ILP methods and tools (e.g., `lp_solve()` [11]) that are extensively researched and efficient. This serves to remove the onus on the WCET tools by way of reuse. An analogous strategy is used for the proposed calculation method described in the next chapter.

IPET offloads the calculation stage of analysis to other tools, but the earlier stages of determining loops and loop bounds are still required. Depending on the analysed code, this process is often done manually and it is tedious and prone to error. An additional limitation of IPET is that the WCET path is *not* explicitly defined. Instead, only a number representing the amount of time or number cycles the worst-case path would take is reported. Determining the explicit WCET path requires additional processing. The actual WCET path is valuable information when analysing the behaviour of the program and requires an additional processing stage to search for the path. Despite this limitation, IPET is the favoured calculation method of the popular academic and commercial WCET tools.

3.4 WCET Tools

This section provides an overview of some of the popular timing analysis tools. These tools are designed to find WCET and they are branded as such, but some are also capable of determining BCET.

The primary motivation for commercial development of tools is their effective and efficient use in industry and current practical issues that the theory of static timing analysis aids in overcoming. Ermedahl has developed the following set of requirements

that a WCET tool should support [12]:

- The tool should produce safe and tight estimates of the WCET and provide deeper insight into the timing behaviour of the analysed program and target hardware.
- The tool should be reasonably retargetable, supporting several type of processors with different hardware configurations. It is valuable to provide insight in how different hardware features will affect the execution time.
- The tool should be able to handle optimised and unstructured code. Also, code for which some of the source-code is not available (e.g., library functions and hand-written assembler).
- The tool and analysis should be reasonably automatic, easy to use and should not require any complex user interaction.
- The user should be able to interact with the tool and provide additional information for tightening the WCET estimate, (e.g., constraints on variable values and information on infeasible paths).
- The user should be able to specify which part of the code to measure, ranging from individual statements, loops and functions to the whole program.
- The user should be able to view extracted results on both a source code and object code level. The information should provide insight in code parts which are executed, and how often.

A survey of available tools, both academic and commercial, yields no tool that fully supports all of the requirements listed above. The reasons noted for the absence of such a tool are the complexity in determining precise timing behaviour of modern processors (with pipelines, caches, branch prediction, etc.), and the fact that embedded system engineers are unfamiliar with static analysis methods, thus limiting the market for theses tools. Moreover, there is a belief that the market for timing analysis tools is small because the number of high-end processors used in embedded systems is proportionally very small. While this is true for the high-end processors

that these tools target, since these tools are primarily focused on solving the *hard* problem of processors with high complexity. There is generally a lack of focus on *low-end* processors, despite the tremendous market size in terms of volume. The processor market for 4-, 8-, 16-bit, DSPs and microcontrollers is over 90%, with the simple 8-bit processors making up 55% of the market. Further, of the less than 10% market share of 32-bit processors, 98% of those are used in embedded systems [41]. Thus, despite the huge potential market for timing analysis tools for these low-end processors, the research and industrial communities have largely ignored this market in favour of attempting to solve more complex problems for processors with a small market share.

Herein lies the motivation for developing a tool that provides precise timing analysis for high volume, low-end processors. Having the capability of proving that a given task can safely execute on a less powerful processor yields significant production cost savings. This is instead of using a much more powerful processor to ensure timing behaviour is safe but at a higher unit cost.

Another reason for the lack of tools that fully support the list of requirements is that most tools support WCET analysis at either source-code level or object-code level. The difficulty these methods introduce is in the mapping of high-level flow information from source-code down to low-level object-code in order to compute the timing bounds. The compiler is the logical component that links the source-code flow with the object level timing information. Unfortunately, many compilers are closed-source provided by the processor manufacturer, and open-source compilers (i.e., `gcc`[16]) generate much less efficient object code.

Obtaining the flow of data information (e.g., loop bounds) automatically from source code is significantly less complex than from object code, that requires maintaining flow of data between registers and memory. For the Reverse Engineering project, we are in effect only considering the latter case because the program was implemented in assembly. Further, at the source level the variable names are more meaningful to the analyst than memory and register addresses. Regardless of whether the methods to obtain loop bounds operate on source- or object-code are manual or automatic, the problem of converting the information into a form for the calculation state remains.

Academic WCET tools have developed out of the motivation for a proof-of-concept

implementation of a new theoretical approach, or to provide a set of functions that allow for theory to be examined and extended. Development of the academic type of tools tends to stall once it reaches a desired level of stability, or the researchers move on to other areas of interest.

One such example is the research prototype *Cinderella* [25], developed at Princeton University, that has not been actively developed for ten years. Cinderella determines both BCET and WCET for the Intel i910 and Motorola M68000 processors. It provides a graphical development environment from source-code to object-code, and the mapping between them.

In Cinderella, timing bound calculation is based on the IPET developed by Li and Malik [26]. The tool determines the linear constraints for cache and pipeline behaviour of the processors, but requires manual annotation of flow behaviour to tighten the timing bounds. For example, the triangular loop problem (Figure 3.1) would require the analyst to determine the linear constraint `x3 = 55 x1`, where `x1` represents the outer for-loop's basic block, and `x3` represents the inner-loop body's basic block. Another limitation of the prototype is that it does not provide the best- and worst-case execution paths because it depends on IPET for timing bound calculation.

The tool provides the capability to retarget the timing bound estimation of different hardware platforms through a well-defined C++ interface. Retargeting requires the implementation of backends for the object file, instruction set, and machine model. Cinderella provides a good model of modularisation to separate the computation and interface implementation from the backend modules that perform the control flow and low-level analysis.

Other academic tools are prototype research implementations strictly focused on source-code level dependent analysis. These are not feasible to extend for this work due to the dependence on high-level source-code and manual annotation requirements. They include Calc_wcet_167 [40] that is an implementation of Kirner and Puschner [21] for the Siemens C167CR processor from the Vienna University of Technology and Heptane [2] static WCET analyser for several processors from ACES Group at IRISA.

Some research projects have evolved into commercial products. One such tool is Bound-T [39] from Tidorum Ltd. in Finland. Bound-T supports several architec-

tures and analyses machine-level code to compute WCET and its execution path. It automatically determines loop bounds for counter-type loops and allows for user assertions on the program behaviour. Another commercial tool is aiT [1], an implementation of the Abstract Interpretation and ILP method developed by Theiling et al[38]. aiT is targeted for high-end processors with cache and pipelines and it computes the WCET and graphically displays the worse-case execution path, but aiT requires manual annotations of loop and recursion bounds and does not find the BCET.

Chapter 4

Timing Analysis by Timed Automata Model

This chapter presents a method of finding precise (i.e., safe and tight) timing bounds, both BCET and WCET, of a program implementation on a specific hardware platform. First, the related work of modelling real-time programs with timed automata and associated results are presented. Second, is an overview of our method of finding timing bounds automatically from assembly/object-code (or related representation) *without* requiring manual annotation for loop bounds and infeasible paths. In the following chapter, we present the transformations from CFG representation of the program to the TA model used in UPPAAL to find the timing bounds using the method proposed in this chapter.

4.1 Related Work

The related work to our proposed method of finding timing bounds begins with an overview of timed automata and model checking. Then an overview of an example of model checking an assembly-level implementation, where the interrupt behaviour (and implicitly timing behaviour) is of interest. Finally, we summarise previous work that presents an argument against the feasibility of using model checking to find timing bounds followed by a rebuttal counter-argument of the claim.

4.1.1 Timed Automata

The original theory of timed automata was developed by Alur and Dill [5, 6], for modelling and verifying real-time systems. A continuous-time timed automaton model is a finite-state Büchi automaton with a finite set of non-negative real-valued clocks. The automaton is an abstract model of a real-time system, providing a state transition system of the modelled system. It is represented as a directed graph, with nodes representing locations and edges representing transitions.

Constraints on the clocks are used to restrict the behaviour of the automaton and enforce progress properties. Clock constraints on locations, called *invariant conditions*, are invariants forcing a transition when the state would violate the clock invariant.

Transitions of the automaton have clock constraints guarding the transition edge, called *triggering conditions*, restricting when a transition can be enabled based on the clock guard. Transitions include clock resets, where some clock variables are reset to zero.

Describing a complete system in one timed automaton is large and cumbersome, thus TA have been extended with communication signals on transitions. The real-time system can then be described as a set of timed automata, with each process of the system decomposed into its own automata. Later, the theory of timed automata was later extended by Bengtsson et al. [8], to include data variables in the state space, that can be used in transition guards and updates.

4.1.2 Model Checking

Given a real-time system modelled as a timed automaton, system properties can be expressed as temporal logic formulas and model checked. The temporal logic commonly used is Timed Computation Tree Logic (TCTL) [4], and is used to express safety and liveness properties of the system. Model checkers search the automata state space exhaustively until a counterexample is found to refute a claimed system property of the model. Due to the potential of state space explosion, model checker implementations use data structures, approximation methods, and symbolic states to efficiently represent and search the state space. Further, some model checkers (e.g., UPPAAL) restrict clock guards to maintain convex zones and check only a subset of

TCTL formulae. Consequently, the model checking results are generated faster, in less space, making checking of larger models feasible.

Model checking has proven to be very successful in the verification of system requirements and design models against system specification properties. In particular, model checking has been used in practice for verifying real-time embedded safety-critical systems design, potentially identifying design errors early in the development cycle. The behaviour of the system design is modelled according to the state transition system notation used by the model checker (e.g., timed automata) with respect to high-level abstract actions that the system performs. The model is created manually by, or in collaboration with, a domain expert of the system, and care must be taken to accurately represent the system.

The model representation of the system abstracts away the details of the underlying software and hardware systems execution, and only models significant timing events [15] (e.g., deadlines, periods, feasible WCET). When the design model successfully verifies against the specification properties, the system is implemented according to the design in the selected programming language and compiled for the target hardware. Foreshadowing our method proposed in the next section, an interesting observation can be made; although model checking is a widely accepted method of verifying the system design, attempts to verify system implementations are typically done by executing test cases on the implementation and/or manual inspection.

4.1.3 Model Checking Implementations

Published examples of the use of model checking representations of implementations are sparse. One such example, by Fidge and Cook [15], is a case study that investigated the behaviour of interrupt-driven software.

The case study models the assembly code of an aircraft altitude computation and display program, to be referred to as Altitude Display, that reads an altimeter and computes an estimate for the aircraft's altitude. The program asynchronously requests the current value from the altimeter and the value is returned to the program by way of an interrupt. While waiting for the interrupt, the program computes an estimate of the altitude. If the interrupt from the altimeter does not arrive in time, the estimate is displayed instead of the actual value returned from the altimeter. The

assembly code is modelled in the Symbolic Analysis Laboratory (SAL, one of the SRI FormalWare tools)[35], with each state representing a basic block of assembly code.

The motivation for the case study is an interest in the behaviour of the interrupt-driven program, that depends on the relative arrivals of the interrupt over time and its impact on the accuracy of the altitude displayed. Instead of modelling all possible interrupt points, only those points in time when *significant events* occur (i.e., the cases when the altimeter responds on time or not) are modelled. Many equivalent states are thereby eliminated when modelling interrupt behaviour of the program's execution.

Model checkers, such as SAL, exhaustively search the state space until a counterexample is found to refute a claimed property of the model, or the system is found to satisfy the property. In the case of SAL's Bounded Model Checker and the model of the Altitude Display program, the property claim is a temporal logic formula asserting reachability of the last instruction. Modelling the Altitude Display task in SAL involves understanding the behaviour of the code and manually modelling its effects on the register and memory values, program flow (i.e., assignment of the program counter), and the instruction cycle execution time of each action. The first is directly translated from the assembly code, while the latter two are implicitly described in the code and require knowledge of the hardware environment for precise execution time and program counter assignment. The execution time is modelled by a variable, *Now*, of type Time, that is used to guard actions and is updated on the transitions with a value representing the execution time of the instructions model by the associated action. Moreover, the interrupt behaviour of the hardware is used to model the different possible *significant events*, modelling equivalent actions. Thus, a limitation of the method is that it requires complete understanding of the given assembly code and hardware environment to correctly manually model the task.

A further limitation of the method presented by Fidge [15] that prevents it from being used to obtain timing bounds from model checking is that SAL searches for the shortest counterexample to the property given as a temporal logic formula. In this case, it would be a reachability property to the guarded action representing the last basic block of assembly code. The counterexample found is the shortest trace, in terms of number of transitions, through the model representing the program code. Thus, SAL's Bounded Model Checker is not capable of finding the fastest and

slowest traces, with respect to the variable modelling time, because its search does not explicitly take time (or time variables) into consideration when searching for the counterexample. Moreover, there is no way to express the temporal logic property to find fastest/slowest timed traces. Therefore, in its current form, the SAL model checker is not a feasible tool to use to find timing bounds of program implementations by model checking.

4.1.4 Timing Analysis by Model Checking

The lack of publications in the area of performing timing analysis with model checking possibly indicates that model checking alternatives do not offer advantages to the present static timing analysis methods. Wilhelm [46] presents several methods for using model checking and argues that none offer acceptable performance. The solutions focused on target hardware with cache, which this thesis does not address, but the method applies this work. The cited problems with the model checking methods were that the state space is too large or require too many model checking iterations to find a precise upper bound.

Metzner [28] countered the argument that model checking is adequate for finding timing bounds and, furthermore, can improve the results. The method models the program and its interaction with the hardware as an automaton. The automaton includes a set of variables, some of which are used to track time consumed. It begins with the source C program with annotation to bound loop iterations. The annotations are preserved in the translation into assembly code. The assembly code is used to generate an automaton. The automaton representation is a C program that is used as input to the OFFIS verification environment. The automaton is model checked for a reachability query to a termination point for some cycles bound N. Based on the result, the bound is then increased or decreased as appropriate until a tight value is obtained for the upper bound of execution cycles. Typically the approach uses a binary search method to choose the next attempt for N. The experimental results revealed that the multiple model checking iterations do not lead to an infeasible method.

Metzner's method and results indicate that timing analysis by model checking is feasible. It finds a precise bound and provides a concrete execution path for the

WCET of the program. However, the method does not satisfy the constraints set out in Section 1.1.1. In particular, the method requires manual annotation of loop bounds and high-level source code.

4.2 Method Overview

This section provides an overview of the method proposed to find timing bounds by model checking a timed automaton. The process differs from the traditional static timing analysis methods, but we identify where each traditional stage (i.e., Flow Analysis, Low-Level Analysis, Calculation) relates to our proposed method. As previously noted, we desire an automated *push-button* method that automatically finds the bounds of a real-time program implementation, without user intervention to identify function calls, loops and loop bounds, input values, etc. An implementation based on this method is detailed in Chapter 6, the STARTS tool suite, and Figure 4.1 illustrates the process used by the tool.

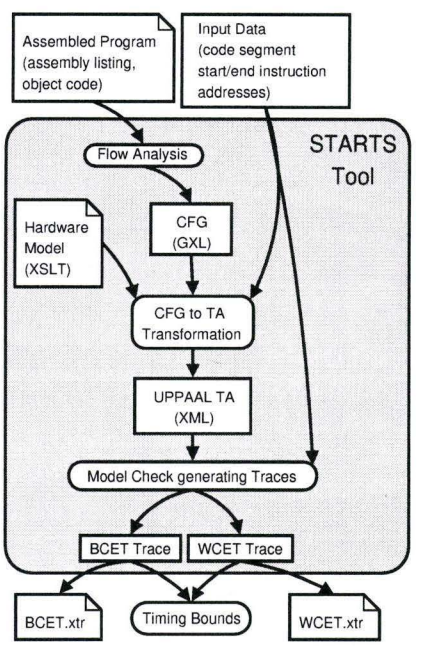

Figure 4.1: STARTS Tool Architecture

To achieve a completely automated timing bound calculation tool, the user inputs

to the tool must be a representation of the program that will run on the target hardware (i.e., an assembly listing or object code), and the start and end instruction addresses of the analysed portion of the program. The given instruction addresses may be the first and last instructions of the program or some segment within the code. Each processing stage is illustrated with an example from a code segment (Figure 4.2) of a control program written in IBM1800 assembly. The example code segment performs a majority vote of 3 input bits from XR2 (Index Register 2), and loops three times storing the count of high bits in XR3.

```
ADDR REL OBJECT   ST.NO.   LABEL OPCD FT OPERANDS  COMMENT
35C9 0   0000      0703    DI2F3 DC     0
35CA 0   6203      0704          LDX    2 3
35CB 0   6300      0705          LDX    3 0         ZERO DI COUNT
35CC 0   4810      0706          BSC      -
35CD 0   7301      0707          MDX    3 1
35CE 0   1001      0708          SLA      1
35CF 0   72FF      0709          MDX    2 -1
35D0 0   70FB      0710          MDX      *-5
35D1 00 66002099   0711          LDX   L2 BPCD      RESTORE PAGE
```

Figure 4.2: Example – IBM1800 assembly code

4.2.1 Control Flow Analysis

The timing tool uses the given inputs to process and compute the timing bounds automatically. The first processing stage is the Flow Analysis stage that generates a static control flow graph (CFG) representation of the program. The CFG nodes represent instructions and are annotated with relevant instruction information (i.e., opcode, operands, etc.) for later processing. The edges represent the possible subsequent instructions that can be executed (see Figure 4.3).

The CFG edges are generated by computing the possible changes to the program counter for each instruction. For program segments that are loops or subroutines, the CFG does not duplicate the instruction nodes (i.e., they are only represented once in a static CFG). Thus, subroutines have multiple in-bound edges to the first instruction node of the called subroutine. Similarly, the last node of the subroutine typically has out-bound edges that return to each of its callers. In the static CFG, it is not possible to determine which return edge should be taken. It would require generating a set of dynamic CFGs, for each call to a subroutine call. In our process,

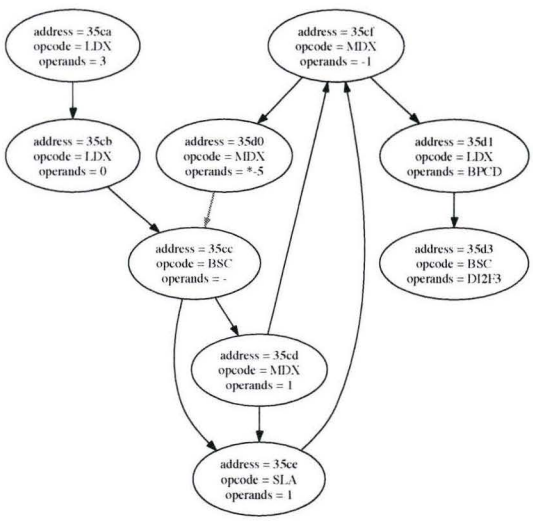

Figure 4.3: Example – Annotated Control Flow Graph

the information from the node annotations is used in a later stage to model the data flow that determines the correct return edge.

4.2.2 Transformation to a Timed Automata

In the second processing stage, the CFG is transformed into an extended Timed Automata (TA). The extension on the classical TA model is defined for use with the real-time model checker UPPAAL. It provides a rich expression language that includes a network of TA, clock and variable tests and updates, bounded integer variables, synchronisation channels, and user defined functions.

The transformation combines the CFG with a Hardware Model of the target hardware. The Hardware Model contains the timing information of each instruction and other timing related behaviour of the target microprocessor. Creation of the Hardware Model represents the Low-Level Analysis static analysis stage, and it provides

information independent of the given program.

The transformation to a TA uses the flow information from the CFG, and combines it with the timing information from the Hardware Model. With one exception, the locations and edges in the TA are, in a one-to-one mapping with the CFG nodes and edges, respectively. As we will see in section 5.3.6, there is a special type of instruction that requires two additional locations and edges to separate the clock and variable guards. Without separating the guards, the model checker finds a deadlock state that does not actually occur in the program. The additional locations and edges are required to correctly model the program's timing and behaviour. The annotations from the CFG nodes, and timing information from the Hardware Model, are used to define clock invariants on locations, and the edge clock and variable guards, variable updates, and synchronisation channels. The location invariants model instruction execution time delay, the edge guards restrict the possible transitions with respect to clock time and variable assignment, and the edge updates assign values to variables modelling the data flow. These are used to define the state space of the control and data flow of the program when model checking the TA (see Figure 4.4). UPPAAL requires manual placement of all labels. All invariants, updates, and synchronisation labels are placed at the upper left corner of the automata figure to reduce clutter and maintain readability.

4.2.3 Data Flow Analysis

The task of identifying loops, the iteration bounds of loops, and depths of recursion calls are frequently performed manually. The process is time consuming and error-prone. For methods that automatically determine loop and recursion bounds, the bound information must still be mapped into a form suitable for the traditional calculation methods. Utilising the extended TA model, the behaviour of data flow, and its effects on the control flow of the program, are modelled. The states they define are then automatically maintained by the model checker without the need for manual intervention.

The TA model's variables can be used to represent the values of the actual memory locations (e.g., general registers, special registers, stack, etc.). The edge update is used to change the value of the variable corresponding to the actual assignment that

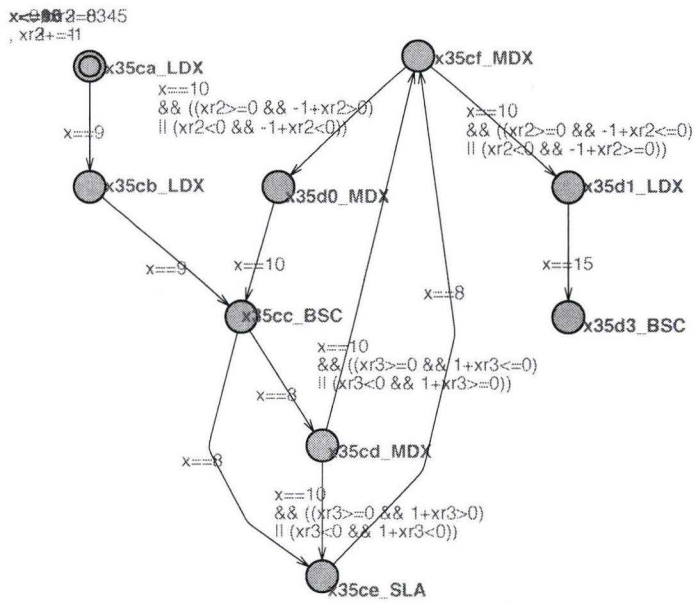

Figure 4.4: Example – UPPAAL Timed Automaton

would occur when the instruction is executed. While the model checker examines the state space of the model, the edge guards and updates restrict the possible values of the variables. Thus for the variables representing loop counters, the state space of the model is restricted to the actual number of iterations the program could possibly execute. The loop bounds are automatically determined without requiring explicit identification of the loop, the loop counter, or the loop bound.

Variables can be maintained to represent input values, computed values, values that are used in conditional branches, and the return instruction address of a subroutine call. The effect of guarding and updating the variables on the state space of the model is that it restricts the possible execution traces to the actual execution traces of the program running on the target hardware. Moreover, the static CFG represents a superset of all possible execution paths through the program. Thus, all infeasible paths in the dynamic CFG are excluded from the state space of the program model using variable guards. The execution time of the feasible execution traces can be calculated from the clock delay that is enforced on each location by its clock invariant and the clock guards of its edge(s).

4.2.4 Calculating Timing Bounds

Creating a TA model from the CFG, followed by annotating the model with timing information from the Hardware Model and data flow information from the CFG annotation, results in a precise model of the program execution. Further, the model includes clock invariants, guards, and updates that provide the precise clock delay of the actual program execution.

The model checker must feature the capability to find the fastest and slowest traces based on the accumulated clock delay of the TA model (e.g., as is done in a prototype version of UPPAAL). The real-time model checker can be used to calculate the timing of the BCET and WCET of the program. The reachability checker for the slowest trace must include the capability to detect infinite loops and ensure termination, as UPPAAL provides [7].

The model checker attempts to verify a reachability property from the initial location (i.e., the user given start instruction address) to the end location (i.e., the given end instruction address). Instructing the model checker to generate fastest and slowest witnessing traces when verifying the reachability property produces the BCET and WCET traces, respectively, for the modelled program. The value of the accumulated clock delay at the end of the trace represents the number of clock cycles the trace execution would require on the target microprocessor. If the clock rate of the target processor is known, the program's timing bounds could also be expressed in units of time.

Chapter 5

A Timing Analysis Transformation System

The details of the transformation process from machine-code representation to CFG, and CFG to TA, previously described are presented in this chapter. The primary focus is on the latter transformation of the CFG representation of the program instructions to a TA model that describes the execution of the instructions. The instruction set operations are decomposed into six types of transformations, categorised by the instruction's effects on control- and data-flow. The TA model obtained from the transformations is used to determine the BCET and WCET of the program.

5.1 Control Flow Graph Representation

The transformation from assembly- or machine-code to a static CFG representation is the first step of the process to generate a TA that can be used for timing analysis. The primary contribution of this thesis is the timing analysis of real-time programs by model checking Timed Automata. Thus, the details of the CFG generation is outside its scope. The reader is referred to [37, 23, 21, 45] for in-depth details of how the graph representing execution paths of the program is constructed. Here, we limit the description of the CFG to what is required to proceed to the next processing step, transformation to a TA.

With respect to the transformation process, it is assumed that all the paths

through the CFG represent at least all the feasible execution paths of the program. The transformation to a TA requires the CFG nodes to be annotated with specific relevant instruction information (i.e., instruction address, opcode, operands, etc.). A complete list of the required information depends on the instruction set and the hardware architecture. Thus, the precise node annotations will vary between target architectures. In general, the annotations must include all the fields of the assembly code instruction and the instruction address. The information in the annotated CFG is combined with the Hardware Model of the target architecture to transform the model into a TA describing the behaviour of the program's execution.

5.2 Timed Automata Representation

The second step of the transformation process is from the CFG representation of the program to a TA model. The model is transformed into a format that can be used with TA model checker that will generate the BCET and WCET.

The instructions of the program, represented as the nodes of the CFG, are mapped directly to locations in the TA. The execution time of each instruction is represented by a clock invariant on the location of the form $x \leq n$, where x is a clock maintaining the instruction execution time, and $n \in \mathbb{N}$. The units of n may be a clock cycle, an instruction cycle, or the smallest fraction of instruction execution time, and is defined in the Hardware Model. The invariant creates a delay transition in the TA that represents the passage of time (clock cycles) for the execution of the instruction, by allowing the TA to remain at the location. The invariant forces one of the feasible action transitions to occur when the delay at the location equals the execution time. The action transitions include a clock guard, of the form $x == n$, that prevents the transition from occurring early.

The action transitions indicate the possible subsequent executable instructions. Each transition includes the clock guard described above, that enforces the instruction's execution time, and a reset of the clock (i.e., $v[x := 0]$) for the next location. Additional data variable guards are added to restrict the possible traces to feasible execution traces. For example, if memory location x34B1 stores a loop bound, and x34B0 stores a loop counter, then the transition that represents exiting a loop will have a variable guard of the form, x34B0 $\geq$ x34B1. Similarly, a transition from some

location that goes to the first instruction of the loop will have a variable guard of the the form x34B0 < x34B1. The variables used in the guards are maintained via a transition update, much like the clocks. The values assigned to the variables represent the same values that the processor would assign during execution. In effect, the model checker emulates the data flow of the program via updates of the data variables in the model.

5.3 CFG to TA Transformations

To obtain a TA representation of the program that can be used with a model checker to extract BCET and WCET, the Hardware Model information is combined with the CFG to transform the graph into a TA that preserves the feasible execution timing and behaviour.

The instruction set architecture for a particular target hardware is decomposed into six classes of instructions. Each class corresponds to the effect the instruction has on the data variables, enabled transitions, and execution time delay. The features of the instruction set determines the completeness of the data-flow required for guarding loop and recursion bounds. That is, those instructions that update data variables involve data-flow, instructions that have guarded transitions that reference data variables, and other instructions that only represent time elapsing with no update or guard on data variables.

When all the data-flow is modelled, every instruction with multiple transitions (represented as out-bound edges in the CFG) has guarded transitions[1], and the execution path is deterministic with respect to the selection of input variable values.

If only a subset of the data-flow is required to determine loop or recursive call bounds, then conditional instructions with multiple transitions that are not related to loops can be modelled as non-deterministic (i.e., unguarded) transitions. With respect to timing analysis, including unguarded conditional instructions will result in BCET and WCET that may be less tight. Conversely, the visualisation of the program provided by the non-deterministic transitions in the TA model is useful for reverse engineering efforts. The unguarded conditional instructions allows an analyst

[1]The term *guarded transitions* is used herein for all transitions that have data variable guards, since every transition has a clock guard representing the execution time.

to choose a particular transition without having to trace back to the input value required to enable the guard on that transition.

The instruction set is decomposed into classes of transformations based on the operations the instruction requires in the TA. The six classes of transformations of instructions are: Sequential, Sequential Updating, Non-sequential Updating Jump, Unguarded Branching, Uniform Execution Time Guarded Branching, and Non-uniform Execution Time Guarded Branching. The transformation of each class of instruction, from CFG to TA representations, is defined below. Implementation of the transformation for a particular target hardware requires each instruction of the architecture to be classified, and the appropriate guards and updates to be determined. Examples of two implementations, for the IBM1800 and 8-bit PIC microcontroller, can be found in Appendices A and B, respectively.

5.3.1 Sequential Instruction

Sequential instructions are the nodes in the CFG with a single out-bound edge to the next sequential instruction that do not modify the data state space (i.e., no assignment to data variables is required for data-flow). Some examples of instructions of this class are: `nop`, and arithmetic or logical instructions that are not required for data-flow.

The next sequential instruction is identified by two nodes, m and n, in the Annotated CFG. Recall that $\lambda(m)$ is a function that returns the set of annotation field and value pairs. If $(\texttt{address}, z) \in \lambda(m)$, where z is the memory address of the instruction, then n is the sequential instruction after m and $(\texttt{address}, z+1) \in \lambda(n)$.

The transformation from CFG to TA is a one-to-one mapping of the node to a location, and edge to a transition. The location is annotated with a label (possibly the instruction address, opcode, or some other desired combination of annotation values), an invariant of the form $x \leq c$ (where c the execution time obtained from the Hardware Model), and a clock guard of the the form $x == c$. The transition update has one entry, the reset of the clock variable x for the form $x = 0$. The transition is illustrated in Figure 5.1. In this example, the delay asserted with the invariant and guard is nine clock cycles or time units.

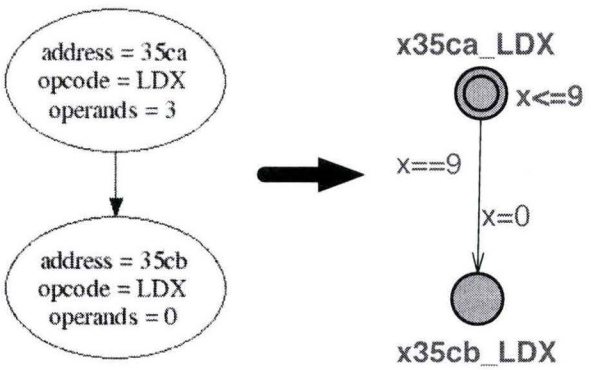

Figure 5.1: Sequential Instruction Transformation

5.3.2 Sequential Updating Instruction

Sequential updating instruction transitions are an extension of Sequential instruction transition, that include an additional transition update. The update is an assignment to the data variable representing the memory location that is modified by the instruction. In effect, the transition update assignment to the data variable emulates the operation performed by the hardware execution of the instruction.

If the instruction operation changes a memory location that is used to determine control-flow (i.e., loop counter, call return address) then the transformation of the sequential instruction must be of this class. The multiple updates on a transition in the TA model are represented by a comma spaced list of assignment statements. When all data-flow requires to modelling, then all sequential instructions (except for `nop`) should be Sequential updating instructions. The transition is illustrated in Figure 5.2.

The source of the value in the update assignment to a data variable is either: (1) a literal obtained from the instruction operand(s), (2) a memory location (that is modelled as a data variable) obtained from the instruction operand(s), or by a selection over a range of input values that represents obtaining an external input value (e.g., a digital input from a sensor).

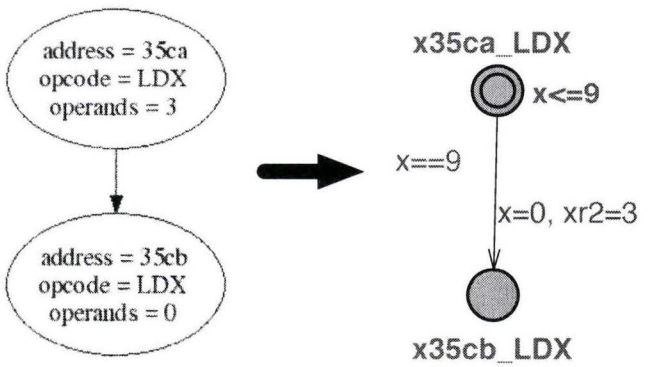

Figure 5.2: Sequential Updating Instruction Transformation

5.3.3 Non-sequential Instruction

Non-sequential instructions are nodes in the CFG with out-degree equal to one, similar to the previously described classes of instructions. The difference is that the destination node of the edge represents a non-sequential (i.e., if $(\texttt{address}, z) \in \lambda(m)$, then $(\texttt{address}, z+1) \notin \lambda(n)$). The types of instructions that are classified as Non-sequential are branch always (e.g, `jump`), subroutine call and return, or branches to an interrupt service routine (ISR) and the ISR return instruction.

These instructions usually update data variables, such as setting bits in the Program Status Register (PSR), store the return instruction address of a call, or push/pop values from the stack which is modelled in the TA as an array of integer variables. The transformation is illustrated in Figure 5.3.

5.3.4 Unguarded Branching Instruction

Branching instructions are nodes in the CFG with multiple out-bound edges. The multiple-edged instructions represent two types of instructions: true branch instructions where the next instruction is chosen based on a comparison of some condition, or the instruction that is a return of a subroutine. In the case of the former, there are typically two or three out-bound edges representing the path taken if the condition is true/false, or $<, =, >$. For the latter, data variable guards must be used (described below in Uniform Time Guarded Branching Instruction).

In instances where it is not required to maintain full data-flow and the instruction

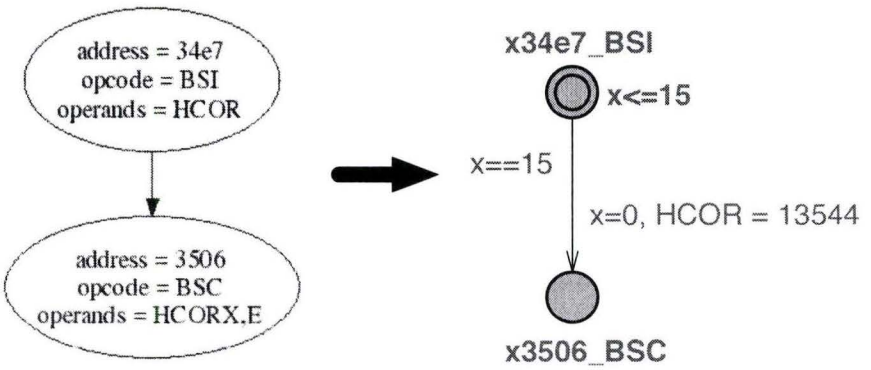

Figure 5.3: Non-sequential Instruction Transformation

is not used to determine loop and call bounds, the transformation for these types of instructions does not need to include data variable guards on the transitions. The Unguarded branching instruction transformation maps the CFG node to a location in the TA model, and maps the edges to transitions from the location to the corresponding target locations. The transformation is similar to the Sequential and Non-sequential transforms. The clock guards on the transitions are the number of clocks it take to execute each of the branches, and the invariant on the location is the largest clock value of all the clock guards. There are no data variable guards. The update resets the instruction clock, and assigns data variables as necessary, as determined by the Hardware Model. The transformation is illustrated in Figure 5.4.

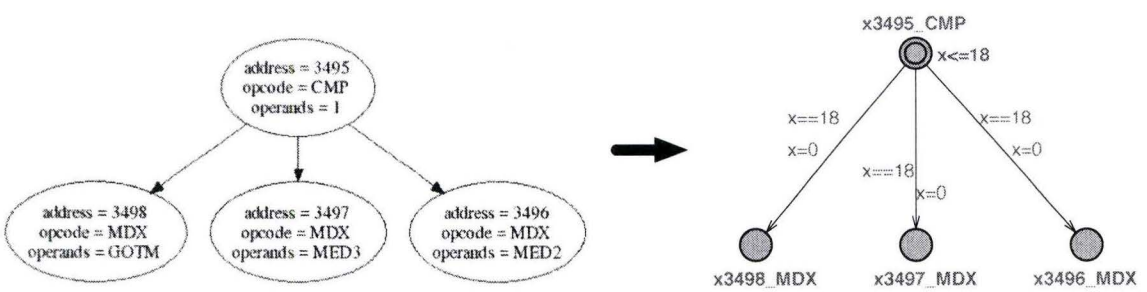

Figure 5.4: Unguarded Branching Instruction Transformation

The timing bounds (BCET/WCET) may be less tight by not adding data variable guards to the transitions of the branching instructions. The trace through the

location is deterministic when the program is executing on real data on the target hardware, but the model does not have guards on that data so the trace in the TA model is non-deterministic (i.e., all transitions are enabled). When the model checker finds the fastest and slowest trace though the TA model, the values that would cause the control-flow path to take such traces may not be the same values expected during execution. When the requirement of data-flow is weakened in this way, the feasible traces of the TA are less restricted and my expose unexpected behaviour due to incorrect input data. With respect to timing analysis, Unguarded branching instruction transformations should be avoided, but they may be useful when the TA model is also used to visualise program execution and other reverse engineering tasks.

5.3.5 Uniform Time Guarded Branching Instruction

For branching instructions with data variable guards there are two classes of instructions. This class, the Uniform time guarded branching instruction, is an extension of Unguarded branching instruction that also has multiple out-bound edges but eliminates the non-determinism introduced by the previous class by adding data variable guards to the transitions. The data variable guard on an out-bound transition represent the condition that needs to be satisfied for the execution of the program to take that path. In the TA model, only one of the transitions is enabled after the time delay, instead of all of them which is the case for the Unguarded Branching Instruction.

Instructions in this branching class have a unique property, the clock guards on all the transitions are the same. That is, the execution time of the instruction does not vary with the branch taken. Guarded transitions with different clock guards (i.e., different branch execution times) introduce a problem in the TA model and are handled by the next class of instructions, Non-uniform time guarded branching instructions.

The transformation of instructions of the Uniform Time Guarded Branching class from the CFG to a TA model maps the instruction node to a location, and each edge is mapped to a transition. The transition is similar to all previous transformations. The value c of invariant on the location (of the form $x \leq c$) is the same value of the clock guards on each transition (of the form $x == c$). The additional transition

guards impose the conditions required to model the equivalent behaviour of the branch instruction execution. The data variable guards for branch instructions must be disjoint and complete, to avoid potential deadlock or multiple enabled transition states that do not correctly describe the behaviour of the program's execution. The update includes the clock variable reset and, if any, the appropriate assignments to data variables. The transition is illustrated in Figure 5.5.

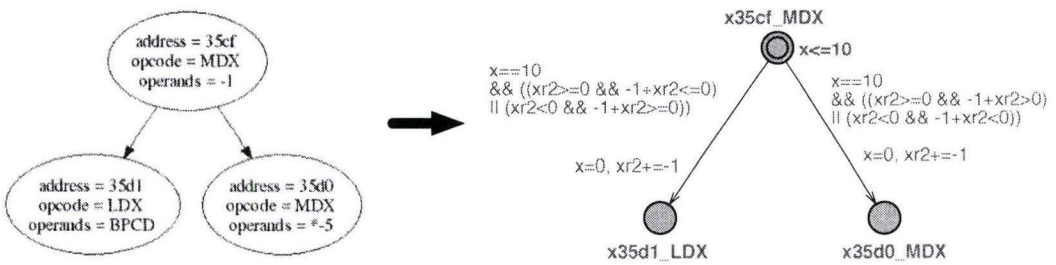

Figure 5.5: Uniform Time Guarded Branching Instruction Transformation

According to the classification criteria given for the Uniform time guarded branching instruction, it includes the instructions that return from a subroutine call. The CFG is a static CFG, so there is only one correct return edge for a subroutine return instruction for a dynamic trace, but that edge cannot be determined from the CFG alone. This problem is over come in the TA model using a data variable to store the return address of the calling instruction.

In the case of the return instruction, the number of out-bound edges depends on the number of times the subroutine is called. In the TA model, each transition from the location represents a return instruction that must be guarded with a condition that checks the instruction address of the transition target with the data variable that stores the current return instruction address for the subroutine. Structuring the TA model in this way results in a trace that can only take the correct return transition, because the data variable guard on the return instruction address guarantees it is the only enabled transition. Further, the time execution time for a return is constant among all the transitions. Hence, the transformation of a return instruction is classified as a Uniform time guarded branching instruction. A return instruction example is illustrated in Figure 5.6.

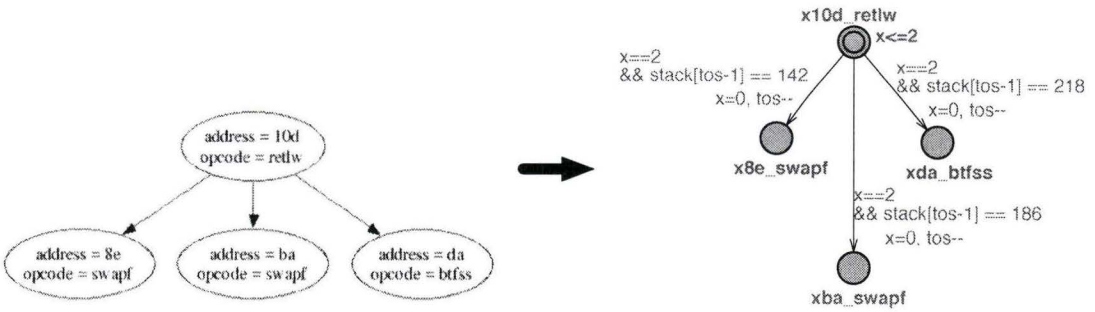

Figure 5.6: Return Instruction Transformation

5.3.6 Non-Uniform Time Guarded Branching Instruction

Non-uniform time guarded branching instructions are similar to instructions of the previous class, but differ in one important aspect. The execution time of the multiple transitions take different clock times. If the clock guards are transformed using the method applied for Uniform Time Guarded Branching instruction of the previous subsection, deadlock states are introduced into the TA model when no deadlock occurs in the program.

The potential deadlock in the model stems from the fact that the invariant on the location must be the largest clock value of all the transition clock guard values. The invariant allows for a delay transition to model the execution time of the instruction. As a result, the invariant value must be the maximum clock delay. The data variable guards on the transitions of a branching instruction are disjoint and complete. Based on the value of the data variable in the guard, only one of the multiple transitions is enabled, and that occurs only when the instruction clock satisfies the clock guard. An example of the TA model is illustrated in Figure 5.7.

The deadlock state is introduced when the transition has a satisfied data variable guard and a clock guard with a value less than the maximum embodied by invariant value. The transition is only enabled when the clock guard is satisfied. That is, at the value of the clock variable that represents the execution time of taking that branch. In this case, TA model does not force taking the enabled transition, because it can remain at the location while the clock invariant is satisfied. In this case, there are no enabled transitions when the instruction clock reaches its upper bound enforced by

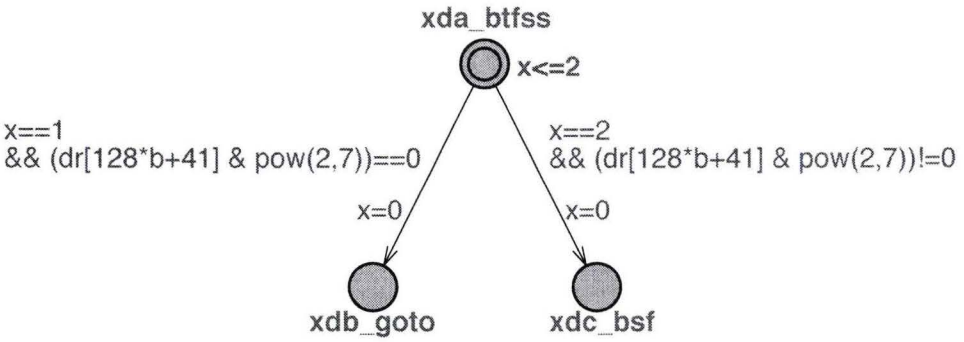

Figure 5.7: Potential Deadlock on Guarded Branching Instruction Transformation

the invariant. Therefore, the model checker identifies a deadlock state that does not accurately reflect the behaviour of the program.

To overcome the introduction of a deadlock state that does not occur in the program code, the mapping of location and edges does not follow the typical one-to-one as with all other transformations. The instructions of this transform class have a one-to-$(n+1)$ node to location mapping, where n is the number of out-bound edges.

Similar to the previous transforms, one of the locations represents the instruction. This location is an *urgent* location (i.e., time is not allowed to pass in an urgent location), so it does not model the execution time delay of the instruction. It is the target of all in-bound transitions to the instruction and has n data variable guarded transitions representing the paths to the subsequent instructions. The targets of these transitions are not the next instruction locations, rather they are auxiliary locations create by the transformation. The auxiliary locations model the execution time of the instruction, thus the transformation puts the clock invariant on the auxiliary location for each branch.

The edges of the CFG that represent the possible branches of the instruction are mapped one-to-two into transitions in the TA model. The first transition is from the instruction location to the respective auxiliary location. There is no update assignment or clock guard on the transition. The only guard is the data variable guard representing the condition that needs to be satisfied to take the branch. The second transition is from the auxiliary location to the instruction location that represents the target of the edge from the CFG. This transition is annotated with all the the appro-

priate guards and updates (similar to the previous classes of transformations). The transition guard includes the clock guard that models execution time, and the same data variable guard from the first mapped transition. It also includes a reset of the instruction clock, and any necessary data variable assignments. This transformation is illustrated in Figure 5.8.

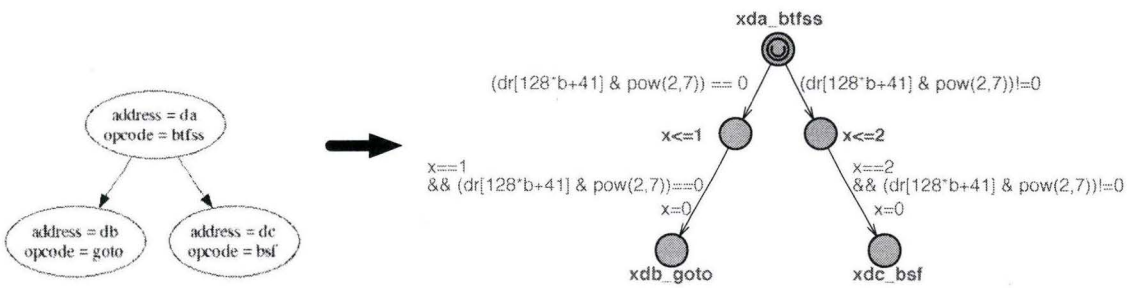

Figure 5.8: Non-Uniform Time Guarded Branching Instruction Transformation

5.4 Summary

The TA model generated by the transformations on the CFG describes the timing and behaviour of the source assembly program. The execution time of the instruction is modelled by the delay transition that is caused by the combination of the clock invariant on the location and the clock guard on transitions. The behaviour of the the program is modelled by the data variable guards and data variable update on the transitions. The updates emulate the program's assignments to memory locations, and the guards restrict the traces through the model to only the feasible execution traces.

With a formalisation of the transformations, the behaviour of the program described by the TA model can be proven correct (i.e., the model includes only the feasible traces) using structural induction over the instructions of the program. Informally, it is shown that the behaviour described by the TA model is correct because the result of the transformation of each instruction exhibits the same operations that occur in the processor executing the machine code.

Chapter 6

STARTS Tool Suite

This chapter describes the prototype tool suite, STARTS, developed to investigate and validate the proposed method of performing timing analysis of real-time programs by model checking a timed automata model. The details of the tool's software requirements and limitations are presented, followed by the usage of the the tool and a description of the intermediate processing stages.

6.1 Tool Suite Description

This section outlines the operating environment of the STARTS tool suite, including its software dependencies and limitations of its use.

6.1.1 Operating Environment

The STARTS tool suite prototype is a command line based application. The tool is currently implemented for UNIX-based operating systems, but it can be extended to run on MS Windows. The command line parameters of the tool reference the input program file and other user-defined options. Presently, the tool supports assembly programs for the IBM1800 and 8-bit PIC target hardware as input. The output files of the STARTS tool is a timed automata model for UPPAAL, and the BCET and WCET traces. UPPAAL binaries are available for MS Windows, Linux, Sun Solaris, and Apple Mac OS X (10.4, as a Universal Binary). Due to the dependance on

UPPAAL to create the traces, the operating environment of the STARTS tool suite is limited to these platforms.

6.1.2 Software Dependencies

The third-party software packages required to use the STARTS tool suite are:

Ruby: An object-oriented programming language used to call the various tools to perform the intermediate processing steps.

Lst2Gxl: A tool to create the CFG from an IBM1800 assembly program listing.

gxltodot: A tool that creates an attributed graph file from the GXL representation of the CFG.

Graphviz's dot: A filter for drawing directed graphs that reads attributed graph files and writes drawings, and is used to create CFG figures and to obtain co-ordinates for the TA model.

xsltproc: An XSL Transformation tool that converts the CFG into a TA model.

Uppaal's verifyta: A command line based version of the model checker used to generate the BCET and WCET, and respective traces.

The software tools provided by the STARTS tools suite are:

PICdasm2gxl: A tool to create the CFG from a disassembled PIC program.

dot2cooridinates: A tool to obtain the layout co-ordinates from dot.

mergeGXLloc: A tool to combine the output of dot2coordinates with the GXL representation of the CFG.

6.1.3 Limitations

Indirect Addressing

The addressing modes supported by STARTS are limited to immediate and direct addressing. For the former, the literal value is obtained directly from the operands of an instruction. For direct addressing, the memory location of the operand value is determined from the instruction operands. For both, the values for the transition guards and updates of the transformation are obtained from the instruction operands only.

The current implementations for the IBM1800 and PIC do not include support for indirect addressing in the source program. Indirect addressing is a scheme in which the address specifies a memory location that contains the memory location address of the operand value. The transformations map memory locations to integer data variables in the TA model. Thus, indirect addressing data-flow cannot be easily modelled in UPPAAL because it requires a method to link data variable values with data variable names. This method requires further investigation.

IBM1800 Data-Flow and Control-Flow

The IBM1800 Instruction Set Architecture was the first attempt at performing timing analysis with the proposed method of using timed automata and a TA model checker. The initial challenge to eliminate the need for manual annotation to loops and subroutine calls required adding data-flow guards and updates on memory location data variables.

The IBM1800 architecture provides three special Index Registers (XR) that can be used to store loop counters. In combination with the *Modify Index and Skip* (MDX) instruction, that adds the operand value to the indicated XR and skips the following instruction if the modified XR reaches zero or changes sign, they realise the implementation of loops in the IBM1800.

A subroutine call is implemented with the *Branch and Store Instruction Register* (BSI) instruction. The operand value is a memory location. The BSI instruction stores the current value of the Instruction Register (i.e., program counter), then modifies the Instruction Register so control-flow branches to the next following memory location. The return from the subroutine is implemented with the *Branch or Skip on*

Condition (BSC), that performs an unconditional branch to the instruction that was stored by the BSI instruction.

To simplify experimentation to determine the feasibility of the proposed timing analysis method, only the data-flow required to model loops and subroutine calls for the IBM1800 is modelled. Other data-flow of inputs, arithmetic and logic instructions, or conditional branching instructions, are not included in the model. Therefore, the control-flow of the model contains some non-deterministic branches that do not accurately represent the deterministic execution of the program. This may result in timing bounds that are not tight.

The second architecture implemented was for the PIC microcontroller. The PIC does not have special registers that are used for loop counters similar to the IBM1800. This required all the memory locations to be modelled, and its data-flow to be maintained in the model. With all the data-flow modelled, the problem of non-deterministic branches can be eliminated by adding the appropriate data variable guards to the transitions. With the knowledge gained from developing the Hardware Model and transformations for the PIC, the IBM1800 implementation can be revised to model the complete and accurate data- and control-flow.

6.2 Using the Tool

This section provides an overview of the use of the STARTS tool suite. Figure 6.1 (a reproduction of Figure 4.1) details the inputs, outputs, and intermediate processing stages of the tool suite.

6.2.1 Input Source Assembler Program

The format of the source program required for performing the timing analysis of IBM1800 programs is the assembly listing. The assembly listing is the output of the assembler that contains assembler-language instructions, machine-language instructions, memory addresses, and possibly other information.

The format for PIC programs is the disassembled machine-code of the program. The disassembled representation is used because the output of the disassembler is easier to parse than the assembly listing. Further, the instructions operands are all

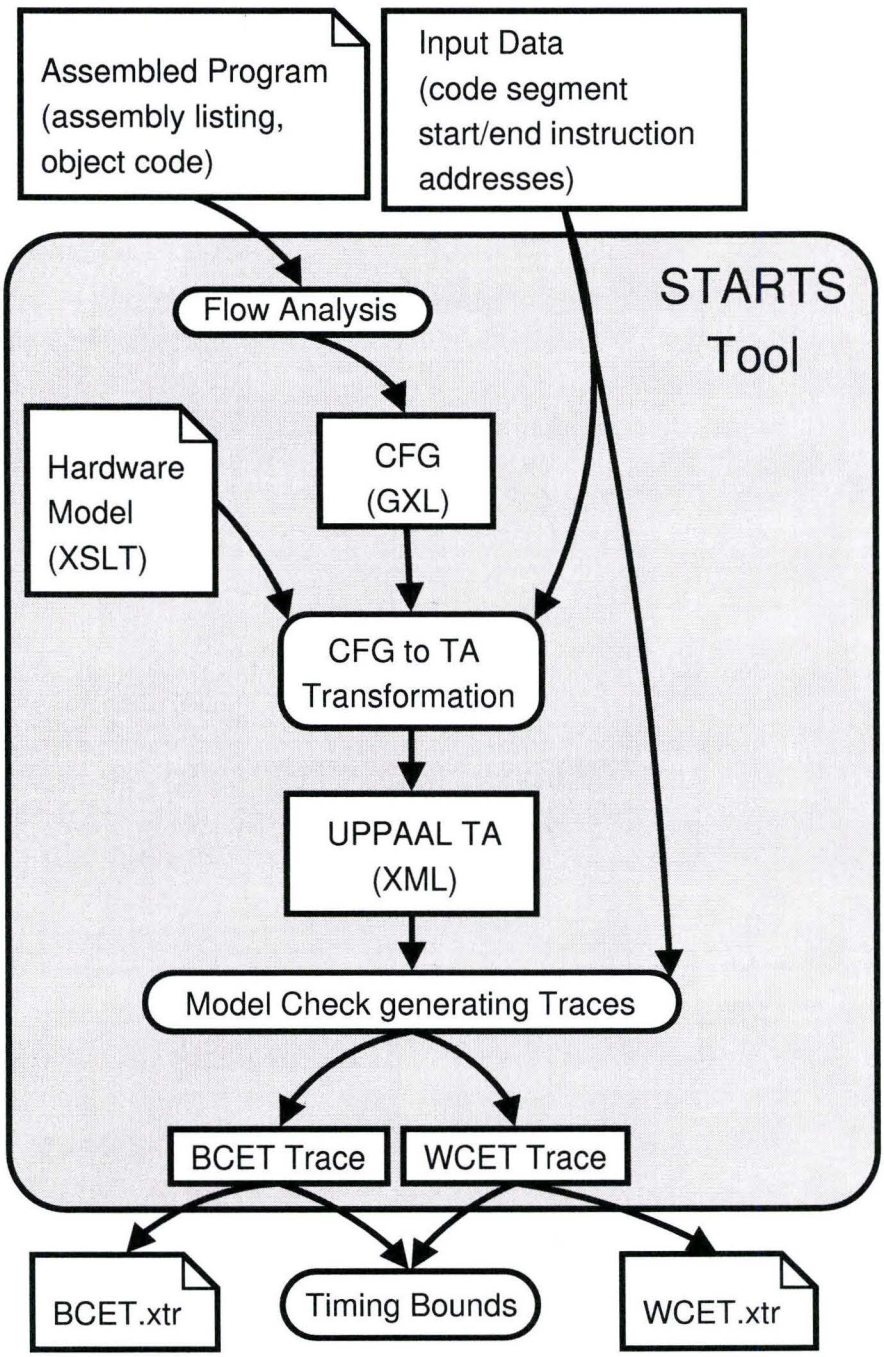

Figure 6.1: STARTS Tool Architecture

hexadecimal values, instead of some assembler-language variable names that require further computation to determine the memory address or value.

6.2.2 Selecting the Code Segment

Along with the required source program, the user may specify start and end instruction address of the code segment to be analysed. These instruction addresses are used in the TA model and by the TA model checker. Unless otherwise specified, the STARTS tool will assume the first instruction is the start instruction, which becomes the initial location in the TA model. The end instruction address is used to define the reachability property for the model checker, and used to generate the BCET and WCET traces.

6.2.3 Generating the Control-Flow Graph

The STARTS tool realises this stage of the process for the IBM 1800 architecture by using the Lst2Gxl tool (see Section 1.2.2). There are some issues with regards to the paths in the CFG generated. In the BPC example code some instructions branch outside BPC program code, thus the graph lacks an edge to the external code and the edge that returns to the next sequential instruction. Also, the graph generation lacks the capability to identify indirect addressing and branching. Thus, imposing the same limitation on the STARTS tool.

The STARTS tool implementation for the PIC microcontroller includes built-in CFG generation from the output of disassembled machine-code. The STARTS tool development was a proof-of-concept. This approach was used because it was the simplest to parse and required the least amount of non-critical development time. For the most part, the PIC architecture control-flow is straightforward to generate and is realised in the current implementation. The complex flow (e.g., indirect addressing, interrupt branch points) has been left to future work. Thus, the PIC implementation does not handle indirect addressing and interrupts must be analysed separately from the main program.

The generated CFG is represented in GXL, an XML-based format developed as a graph exchange language. The GXL output is used to generate a graph figure of the

static CFG of the program using `dot`, and it is used as the input file to generate the timed automata model of the program execution.

6.2.4 Generating the Timed Automata Model

The timed automata model of the behaviour and timing of the program execution is generated by an XSL transformation. The behaviour, timing, and hardware effects (i.e., the Hardware Model) of each instruction format are encapsulated in template rules of an XSLT stylesheet. The rules define the transformation from the XML-based input file to an XML-based TA output file, based on the transformations described in Section 5.3. An XSLT processor is used to apply the XSLT stylesheet of the Hardware Model to the CFG, represented as a GXL file, to generate the XML output of a UPPAAL TA model. The model includes an automaton of the transformed CFG, and any other automata required to model the hardware behaviour and timing (e.g., A/D conversion, clock timer interrupts).

The layout of the locations and transitions in the TA model must be manually placed in the graphical version of UPPAAL, typically performed in the Editor window of the application. STARTS automatically obtains the co-ordinates from `dot`, resulting in a TA layout that is similar to the CFG figure.

6.2.5 Generating BCET and WCET Traces

Using the TA model of the program with guards and updates to enforce the actual behaviour and timing of the program's execution, the best- and worst-case execution times are generated using the UPPAAL verification engine, `verifyta`. In addition to the TA model file, a query file is required as input.

The query file contains the reachability queries that the verifier will check and use to generate the traces. The first property is a liveness property that the model of the program will always eventually (e.g., `A<> bpc.x35d3_BSC`) reach the last instruction of the code segment (e.g., `x35d3_BSC` is given as the last instruction). It verifies the program will not deadlock or loop infinitely, and that it will eventually reach the last instruction of the code segment. If a deadlock or infinite loop is found by the model checker then a trace is generated to the problem location.

The second property verified checks that the model of the program possibly reaches the last instruction location (e.g., `E<> bpc.x35d3_BSC`). If the first property is satisfied, this property will also be satisfied. The second property is used in combination with UPPAAL's Diagnostic Trace options Fastest and Slowest, to find the BCET and WCET traces, respectively. The resulting BCET and WCET are determined from the final value of a clock variable that accumulates the time delay from the start to the end instruction locations of each trace.

6.2.6 Trace Visualisation in UPPAAL

The BCET and WCET trace can be loaded in the Simulator component of the graphical version of UPPAAL. The Simulator is a validation tool that enables examination of the possible dynamic executions of a model and is used to visualise execution traces generated by the verifier. It supports stepping through the model transition-by-transition, or replaying the entire trace.

The Simulator display includes four panels: the Simulation Control (i.e., a listing of the current enabled transitions, current simulation trace, and trace control buttons) on the left, the Variables Panel (i.e., clock and data variable values) in the middle, the Process Panel (i.e., an instance of the model graph indicating the current location and transition in red) on the upper right, and a Message Sequence Chart on the lower right. The following set of figures illustrates a simulation of a WCET trace from the initial state (Figure 6.2), to an intermediate state with a choice of enabled transitions (Figure 6.3), and the final state (Figure 6.4). The time delay of the WCET trace of the example IBM1800 code segment presented in the figures is 177 clocks, or 44.25 μs.

The simulation of the program model effectively represents emulation of the program execution. In forward development, the simulation can be verified against the program code to verify the behaviour of the model is consistent with the source code. In reverse development, the simulation visualisation augments comprehension of the program behaviour and aids in identifying input dependent behaviour. The simulation capabilities are not limited to the generated traces, but it can be used to trace any feasible execution path that is in the state space of the model.

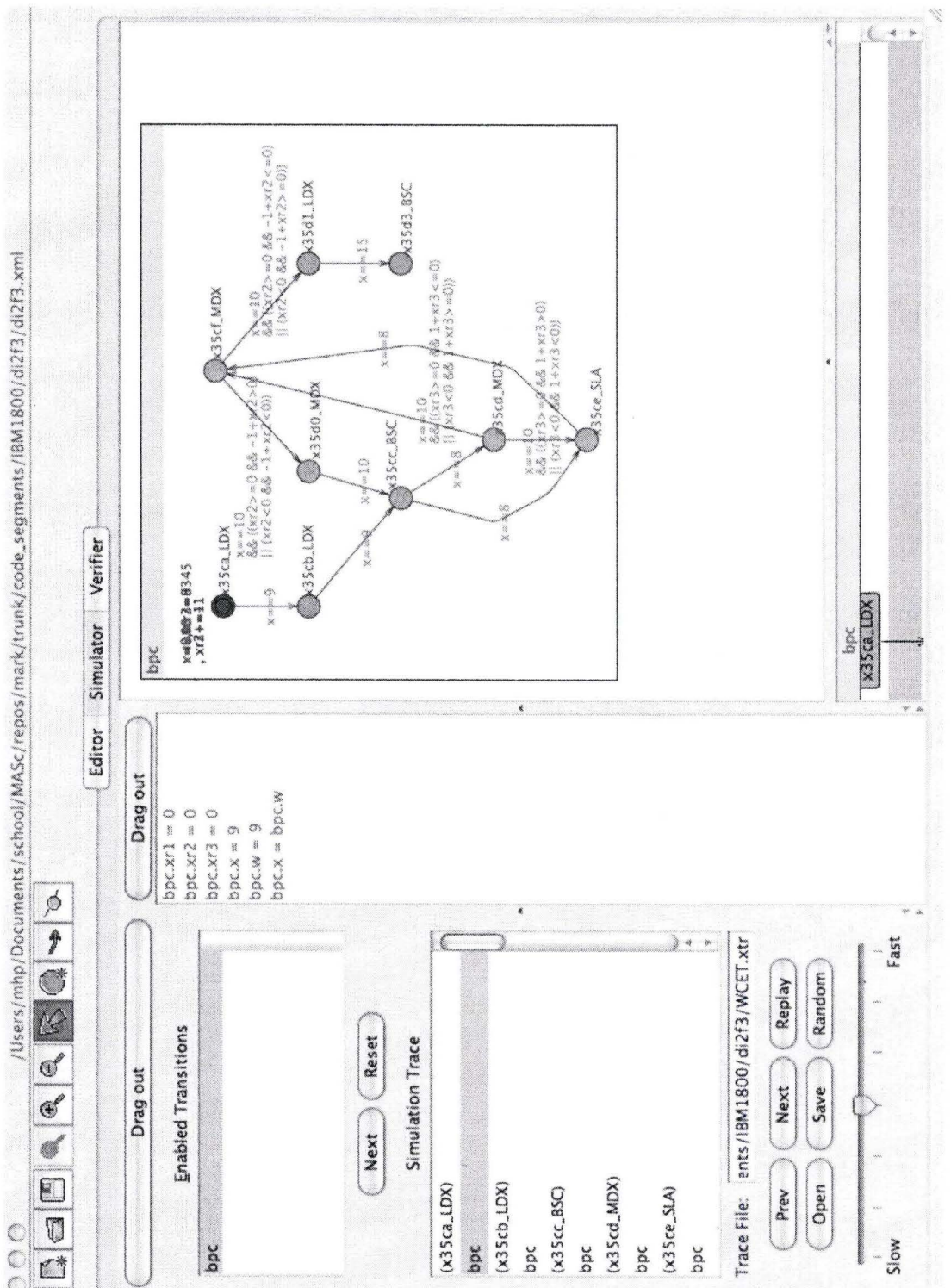

Figure 6.2: Uppaal Simulator Example 1

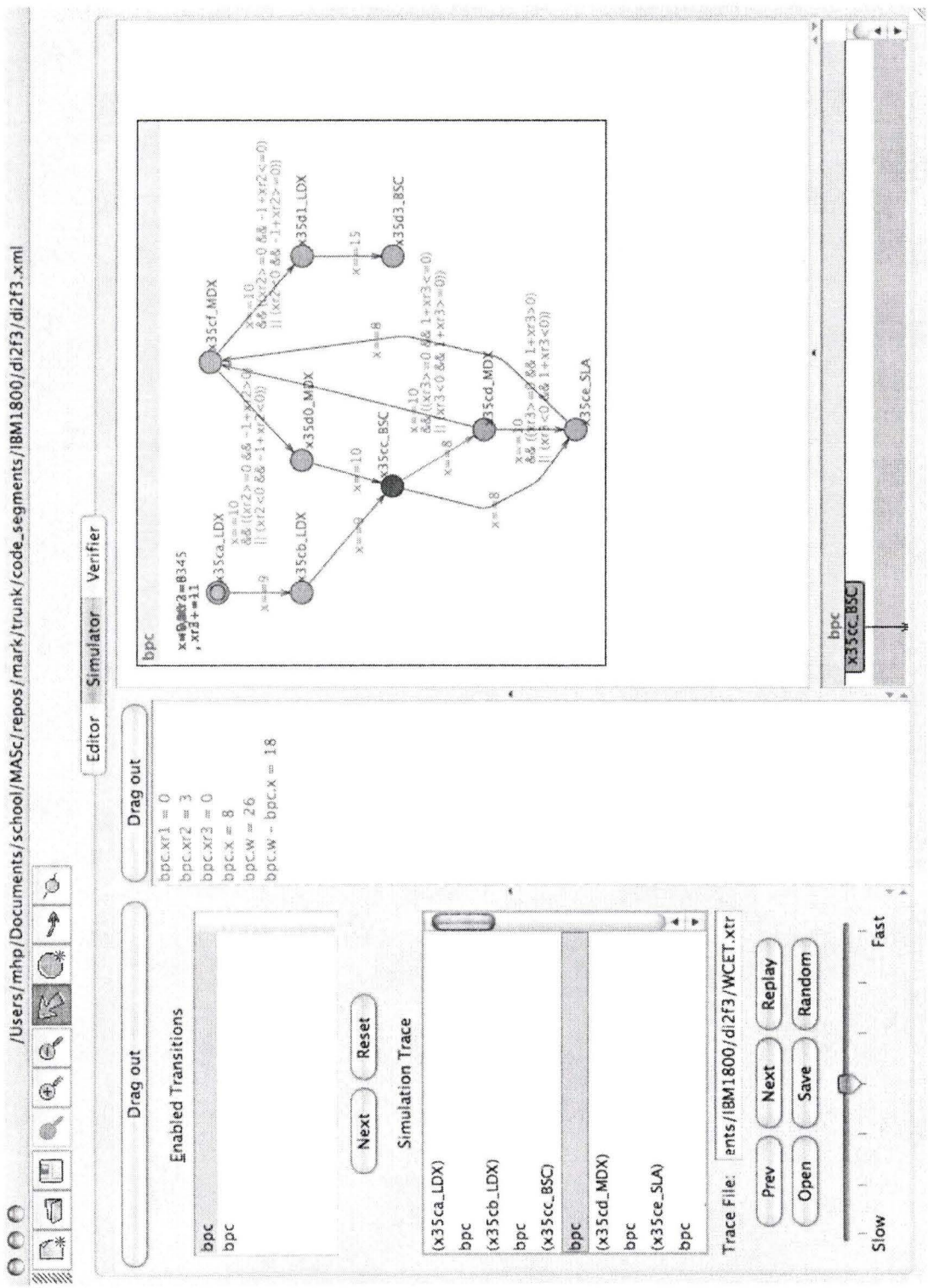

Figure 6.3: Uppaal Simulator Example 2

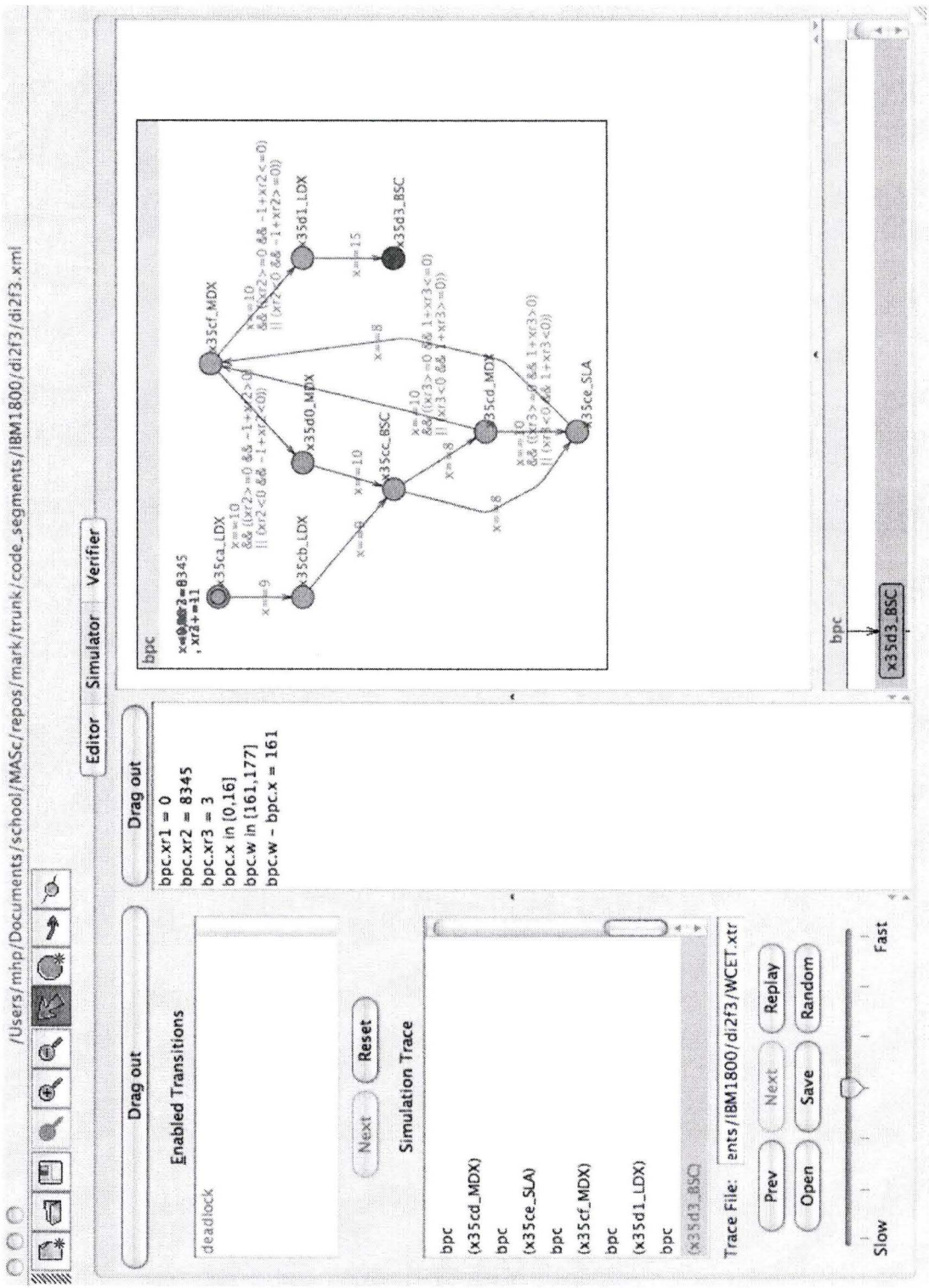

Figure 6.4: Uppaal Simulator Example 3

Chapter 7

Timing Analysis Results

This chapter presents results analysing programs for the IBM1800 and PIC target architectures using the STARTS tool suite.

7.1 Timing Analysis Results

The timing analysis results of using the method introduced in this thesis are illustrated with examples from the IBM800 Boiler Pressure Control code and PIC-microcontroller code of a PID controller used to stabilise an inverted pendulum. A second PIC example is based on the inverted pendulum code to control the levitation of a permanent magnet beneath a solenoid.

7.2 IBM1800 Timing Analysis Results

The code segments of the IBM1800 assembly program analysed are from the Boiler Pressure Control (BPC) program that was the focus of the reverse engineering project (Section 1.1.1). The complete program could not be analysed due to absent or superfluous edges in the CFG (discussed in Section 6.2.3), instead functions within the program were analysed.

Functions in the BPC code were identified manually as segments of instructions from the assembly code that are connected nodes in the CFG. A function is identified by the first and last instruction address. A subgraph of the BPC program's entire

CFG is generated that includes only the connected instruction nodes of the function. The BPC functions analysed include Boiler Pressure Median (`MEDIAN`), Corrected Hilborn Average (`HLBN`), Setback Majority Vote (`DI2F3`), and the Turbine Feedback Calculation (`TRBFB`) that also calls `DI2F3` . The `TRBFBns` is the same function but the path that shortcuts the feedback calculation is made unfeasible. The last example demonstrates the different traces through the `DI2F3` subroutine that affects the branch taken after the subroutine has completed for the BCET and WCET traces of the feedback calculation.

The code examples were small enough to allow manual verification of the results generated by the STARTS tool suite is correct and presented in Table 7.1.

Table 7.1: IBM1800 BPC Results

Function	BCET (clocks)	BCET (μs)	WCET (clocks)	WCET (μs)
`MEDIAN`	45	10.25	143	35.75
`HLBN`	1437	359.25	4461	1115.25
`DI2F3`	147	36.75	177	44.25
`TRBFB`	452	113	771	192.75
`TRBFBns`	646	161.5	771	192.75

Sun [36] analysed the same `TRBFB` segment of code with the WCET tool he implemented. The worse-case execution path generated by the tool is exactly the same at the WCET trace generated by UPPAAL. Sun's WCET reported by the tool was 189.5 ms (units should be μs). The difference from the STARTS output of 192.75 μs is attributed to small differences in the execution time assigned to some instructions. For example, the first `LD` instruction of the code segment is 3.75 μs in Sun's work, but 17 clocks (or 4.25 μs) in the STARTS hardware model. The execution time of each instruction was obtained from the IBM1800 Functional Characteristics manual [20]. The source of the discrepancy in instruction execution times could not be determined.

7.3 PIC Timing Analysis Results

The program used to experiment with the PIC implementation of the STARTS tool suite was a Software PID Control of an Inverted Pendulum Using the PIC16F684

[9] provided by Microchip. The assembly code implementation includes loops, subroutine calls, A/D input conversion, and an ISR. Experimentation with the code provided insights in to handling data-flow, subroutine calls, and additional hardware functionality.

The assembly code of the inverted pendulum controller was modified by a student for a graduate course project. The project was to design and implement a controller that stabilises the magnetic levitation of a permanent magnet beneath a solenoid by controlling the current flow and direction through the solenoid. The program is a busy-wait loop that modifies the output of the program's feedback control loop after the ISR updates the input sensor value. The ISR is triggered by an internal clock timer initially set to update the input value, and accordingly execute the feedback control loop to modify the output value, every 256Hz. The implementation of the magnetic levitation controller realised a system that would not remain stable for longer than short periods of time.

Analysing the implementation with the STARTS tool, the WCET of the feedback control loop was calculated to be 867 cycles, and the WCET of the ISR was 51 cycles. Thus the combined worse-case execution time of the ISR and the feedback control loop was 918 cycles, or 114.75μs. The combined execution time is the minimum period required for the interrupt timer. By identifying the WCET of the feedback control loop and the ISR, the maximum frequency of the interrupt timer was computed to be 8714Hz. Therefore the setting for the clock timer to interrupt was changed from 256Hz to 8kHz, and increasing the response rate resulted in a stable system.

Chapter 8

Conclusions and Future Work

In this thesis the timing analysis of real-time programs is examined and an alternative method of obtaining best- and worse-case execution times of assembly-level software is developed. The major contribution of this thesis is a new method of obtaining timing bounds that is made possible by a transformation system from a static control-flow graph to a timed automaton model of the program. The model describes the state space of the program's dynamic behaviour and timing. The state space is searched with the timed automata model checker UPPAAL to generate fastest and slowest traces. The STARTS tool suite is a proof-of-concept implementation used to validate the method developed in this thesis.

In summary, the major benefits of the timing analysis method proposed in this thesis are:

- A transformation system from static control-flow graph to timed automata model of the program and hardware architecture.
- Calculation of tight and safe timing bounds of unstructured assembly code.
- Timing bounds and respective traces are obtained automatically without the need for manual annotations.
- Timing bound and trace computation make use of pre-existing efficient optimisations of state space representation and searching provided by UPPAAL.

- A prototype implementation, the STARTS tool suite, used to develop and validated the proposed method automates the timing analysis process.
- Safety and liveness properties of the implementation can be verified, providing alternative means of validating the implementation in addition to testing.
- Timed Automata model representation traces can be simulated providing a graphical visualisation of the program execution paths.

In the remainder of this chapter, the first section details the major benefits of this contribution in comparison to other timing analysis methods. It is followed by an outline of future work that would contribute to overcoming present hardware architecture features. Solving the future work problems will provide the STARTS tool the capabilities to become a robust timing analysis tool suite.

8.1 Method Benefits

8.1.1 Tight and Safe Lower and Upper Time Bounds

The TA model defines the entire state space of the real-time program's execution. The symbolic traces generated by UPPAAL describe the precise execution paths that result in the best- and worse-case execution times of the program. Thus the best- and worst-case timing bounds obtained from model checking a TA representation of the program are both safe (i.e, not over/underestimated, respectively) and tight (i.e., not under/overestimated, respectively).

8.1.2 Automatic Path Determination

The benefit of the method in comparison to other static timing analysis methods that generate explicit execution paths is that the manual annotation required to identify loop bounds, infeasible paths, and return instruction addresses is not required. Further, the source program is not required to be high-level code or highly structured assembly code.

Timing analysis of the program from the CFG is difficult due to the problem of the subroutine call return paths from the static CFG. The issue is overcome with

the addition of data variable guards on transitions that only enable the transition corresponding to the correct return path.

The task of manually asserting the loop and recursive call bounds is time consuming and error prone. The TA model includes data variables that update the loop counters and data variable guards on the transitions that represent the conditional branch out of the loop. Also, the return instruction address is maintained for each recursive call and the appropriate return instruction guard ensures that the correct transition is enabled. As a result, this combination of guards and updates automatically bounds loops and recursive calls within the state space of the model without the need of manual intervention.

8.1.3 Concrete Execution Paths

The BCET and WCET traces describe concrete execution paths through the program. The traces also provide input values that generate the paths. The traces may not be unique, rather they are one of many possible traces with the same total execution time. The traces can be analysed to determine the input values, and the values can be used as test cases for the program to verify the results using dynamic timing analysis methods.

8.1.4 Safety and Liveness Verification

For safety-critical hard real-time systems, much care is taken to ensure that safety properties are maintained and that stringent timing requirements are met. Verification of the properties and requirements of an implementation can be difficult and time consuming. The TA model of the implementation can be used for more than obtaining timing bounds. Various safety and liveness properties can be verified by the TA model checker UPPAAL.

Model checking that the program will *always eventually* reach the last instruction of the code segment (e.g., the branch to the beginning of the control loop) verifies the liveness property (i.e., `A<> p`, where `p` is a location) that the program will not deadlock or livelock, and that it will always reach the last instruction of the code segment. If a state exists with no enabled transitions or an infinite loop is detected, UPPAAL will generate a trace to the problem location. The location indicates the

instruction in the program where the deadlock or livelock occurs. When generating the WCET trace, this property is checked prior to generating the trace to ensure termination of the worse-case trace.

Reachability properties (i.e, `E<> p`) to instructions of interest can be used to verify that instructions of the implementation are on a feasible execution path. That is, the instruction is reachable from the beginning of the program (i.e., helping to identify dead code). With traces enabled, if the property is satisfied then a trace will be generated to the instruction that can be used for further analysis of the program's functional and timing behaviour.

Safety properties of the program can be verified with the above temporal properties to ensure an instruction is reached, or that it is reached within a specific time limit. Another safety property that can be verified makes use of the *leads to* (i.e., `p --> q`) property, meaning whenever `p` holds eventually `q` will hold as well. `p --> q` is equivalent to `A[] (p imply A<> q)`. This property can be used to verify that an instruction will always be executed at some point after some other instruction. An example is a safety property that states: if a shutdown signal is detected the system must shutdown. Whether the implementation satisfies the safety property can be verified by checking the property `p --> q`, where `p` is the instruction where the shutdown signal is detected and `q` is the instruction that shutdowns the system.

Other properties can be verified with respect to execution states of the program, values or range of values on inputs or memory locations. Thus, the TA model of program can be used for numerous tasks other than finding timing bounds. It is useful in forward development to verify system requirements, and for the identification of system requirements in reverse development.

8.1.5 Execution Path Visualisation

In addition to verifying specific properties of a real-time program, the model can be used to effectively emulate the program's execution. Using the Simulator component of UPPAAL, the execution path of the program is visualised graphically. Input values and locations with multiple enabled transitions can be selected manually and the resulting traces examined.

The visualisation can be used in reverse engineering efforts to facilitate under-

standing of the program's behaviour. In particular, when assembly code is unstructured and contains many branches and loops, the simulation of the execution clearly demonstrates the control-flow of the program.

For example, the BPC function `HLBN`, obtains 14 input values, corrects them, then calculates the average. Tracing through the source code is difficult because there are seven iterations of a loop, and each loop calls the correction subroutine twice as it steps through subsequent input values. Simulating the BCET or WCET traces for the code segment, quickly and clearly identifies the 14 calls to the subroutine and the seven loop iterations.

8.1.6 Accurate Modelling of Parallel Execution

Using networks of communicating timed automata, the behaviour and timing of the execution of parallel processes can be modelled. An example of this is the analog-to-digital (A/D) converter automaton in the PIC model. The program loops until the A/D conversion of an input is complete and then proceeds with execution. This expands the state space to include all possible input values that can be obtained from the A/D conversion.

In future work, other automata could be added that model timers, interrupt service routines, preemptive execution, or the external environment. This would potentially result in a more complete model that includes support for analysis of these types of complex architecture features. This topic is discussed further in Section 8.2.3.

8.1.7 Leveraging UPPAAL

UPPAAL is a tool developed for modelling, validation and verification of real-time system design. In addition to design verification, using it with the method developed in this thesis allows verification of program implementation. As a result, both ends of the development cycle can use the same underlying tool. This continuity of the development tool throughout the process aids in adoption of formal methods tools in real-time program development.

Another advantage to using UPPAAL in the context of timing analysis is akin to the use of linear programming solvers with the IPET static analysis method. In the case of the latter, tools such as `lp_solve` are robust, mature, and highly optimised.

Such tools are used to off-load the calculation stage of timing analysis from the timing tool. Similarly, UPPAAL is a mature tool with numerous optimisations to symbolically represent the infinite state space of the timed automata. It quickly and efficiently searches the symbolic states space, removing the burden of developing similar algorithms for the STARTS tool.

8.2 Future Work

The future work described in this section outlines the outstanding issues that are required to be solved to provide the STARTS tool with robust capabilities to automatically generate and analyse a timed automata model for various hardware architectures. The following issues have been identified for future work:

- Indirect memory address references and indirect branching included in the model.
- Complex hardware features similar to the A/D converter, such as special instruction set features, interrupt clock timers, watchdog timers, preemption, caches, and pipelines support.
- Identification and handling of overflow and out-of-range operations.
- Limiting the state space explosion when generating traces of models with large input value ranges.
- Extending support for other processors and identifying architectures that cannot be modelled.
- Formalisation of the transformations and proof of execution path equivalence.

8.2.1 Indirect Addressing

The scheme chosen to model memory address locations as integer variable names makes indirect memory addressing infeasible. The scheme is imposed by the available data variables supported by UPPAAL. A method needs to be devised in order to obtain the value of the memory location variable that is represented by the integer value of

some other memory location variable. Such a method could possibly use a function[1] to set or return the appropriate value.

The current control-flow graph generation tools do not maintain information required to include branching to instructions via indirect addressing. The indirect branching must be included in the CFG for the indirect branching transitions to be included in the model. Further, the model must support indirect addressing if the transitions of an indirect branch are guarded with the target instruction address (e.g., indirect reference to a return instruction address).

8.2.2 Overflow and Out-of-Range Detection

The TA model defines the state space of feasible execution paths and values for the program. The model can be used to verify that the implementation does not perform an operation that will cause an overflow or out-of-range assignment, or to identify input values that result in an overflow. Overflow in a hard real-time system can have catastrophic results, thus overflow detection is important to the verification of an implementation.

For example, an input value used in an arithmetic operation that will overflow the size of the memory location that stores the result. An 8-bit memory location can be represented in the model by a bounded integer within the range [0, 255] or [-127,128]. The model checker can detect any out-of-range assignments that are in the state space of the model, thereby identifying the instruction and a value that creates an overflow.

The documented behaviour of UPPAAL when an out-of-range integer variable assignment occurs is to stop the simulation or verification and report the error to the user. A simple example was created that models a pathological loop where the loop counter and loop bound are both incremented. The example revealed UPPAAL did not stop and identify the error as documented, but proceeds without performing the assignment to the integer variable. A bug report was submitted[2] and the problem was corrected in the recently released version 4.0.1. A full investigation of overflow and out-of-range detection can now be performed.

[1] UPPAAL supports function declarations.

[2] `http://bugsy.grid.aau.dk/cgi-bin/bugzilla/show_bug.cgi?id=48`

8.2.3 Preemption and Interrupts

The current model does not include support for preemption and interrupt service routines (ISR). ISR code segments are disconnected from the main program in the control-flow graph. The separate code segments currently must be analysed separately from the main program and manually.

One method proposed, but not investigated, is to include ISR in the timed automata model and add transitions to and from ever location where interrupts are enabled in the segment of locations representing the ISR. This unnecessarily increases the state space of the model and clutters the graph layout of the model. A possible alternative could be to create a separate automaton for the ISR instructions with an additional initial location with a single transition to the first instruction of the ISR. The transition can be guarded with conditions, such as, interrupts enabled and interrupt triggered. An additional automaton could model the environment, clock timer, or any other interrupt trigger.

Bibliography

[1] AbsInt, "aiT: Worse-Case Execution Time Analyzers." `http://www.absint.com/ait/`, 2006.

[2] ACES Group, "Heptane static wcet analyzer." `http://www.irisa.fr/aces/work/heptane-demo/heptane.html`, 2003.

[3] L. Aceto, A. Bergueno, and K. G. Larsen, "Model checking via reachability testing for timed automata," in *In Proceedings of the 4th International Workshop on Tools and Algorithms for the Construction and Analysis of Systems. Gulbenkian Foundation, Lisbon , Portugal, 31 March - 2 April, 1998.* (B. Steffen, ed.), Lecture Notes in Computer Science 1384, pp. 263–280, 1998.

[4] R. Alur, C. Courcoubetics, and D. Dill, "Model Checking for Real-Time Systems," in *Fifth Annual IEEE Symposium on Logic in Computer Science*, (Washington, D.C.), pp. 414–425, IEEE Computer Society Press, June 1990.

[5] R. Alur and D. Dill, "Automata for modeling real-time systems, in lecture notes in computer science 443," in *Proc. of the 17th International Colloquium on Automata, Languages and Programming*, Springer-Verlag, 1990.

[6] R. Alur and D. Dill, "Automata for modelling real-time systems," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–236, Apr. 1994.

[7] G. Behrmann, K. G. Larsen, and J. I. Rasmussen, "Beyond liveness: Efficient parameter synthesis for time bounded liveness," in *Formal Modeling and Analysis of Timed Systems, Third International Conference, FORMATS 2005, Uppsala, Sweden, September 26-28, 2005, Proceedings* (P. Pettersson and W. Yi, eds.), vol. 3829 of *Lecture Notes in Computer Science*, pp. 81–94, Springer, 2005.

[8] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi, "UPPAAL — a Tool Suite for Automatic Verification of Real–Time Systems," in *Proc. of Workshop on Verification and Control of Hybrid Systems III*, no. 1066 in Lecture Notes in Computer Science, pp. 232–243, Springer–Verlag, Oct. 1995.

[9] J. Charais and R. Lourens, "An964 - software pid control of an inverted pendulum using the pic16f684." `http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1824&appnote=en021807`, 2006.

[10] A. Colin and G. Bernat, "Scope-tree: A program representation for symbolic worst-case execution time analysis," in *ECRTS*, p. 50, IEEE Computer Society, 2002.

[11] K. Eikland and P. Notebaert, "lp_solve." `http://lpsolve.sourceforge.net/5.5/`, 2006.

[12] A. Ermedahl, *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Sweden, May 14 2003.

[13] A. Ermedahl and J. Gustafsson, "Deriving annotations for tight calculation of execution time," in *European Conference on Parallel Processing*, pp. 1298–1307, 1997.

[14] K. Everets, "Assembly language representation and graph generation in a pure functional programming language," Master's thesis, Dept. of Computing and Software, McMaster University, December 2004.

[15] C. Fidge and P. Cook, "Model checking interrupt-dependent software," *12th Asia-Pacific Software Engineering Conference (APSEC'05)*, Jan. 01 2005.

[16] Free Software Foundation, "Gcc, the gnu compiler collection." `http://www.gnu.org/software/gcc/`, 2006.

[17] Free Software Foundation, "Gnu binutils." `http://www.gnu.org/software/binutils/`, 2006.

[18] C. Healy, M. Sjodin, V. Rustagi, and D. B. Whalley, "Bounding loop iterations for timing analysis," in *IEEE Real Time Technology and Applications Symposium*, pp. 12–21, 1998.

[19] R. Holt, A. Winter, and A. Schrr, "Gxl: Towards a standard exchange format," 2000.

[20] IBM Systems Reference Library, *IBM 1800 Functional Characteristics*, eighth ed., July 1969.

[21] R. Kirner and P. Puschner, "Timing analysis of optimised code," 2003.

[22] R. Kirner, R. Lang, G. Freiberger, and P. P. Puschner, "Fully automatic worst-case execution time analysis for matlab/simulink models," in *ECRTS*, pp. 31–40, IEEE Computer Society, 2002.

[23] R. Kirner and P. Puschner, "Supporting control-flow-dependent execution times on WCET calculation," Dec. 07 2000.

[24] K. G. Larsen, P. Pettersson, and W. Yi, "Model-Checking for Real-Time Systems," in *Proc. of Fundamentals of Computation Theory*, no. 965 in Lecture Notes in Computer Science, pp. 62–88, Aug. 1995.

[25] Y.-T. S. Li, "Cinderella 3.0 home page." `http://www.princeton.edu/~yauli/cinderella-3.0/`, 1996.

[26] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," in *Workshop on Languages, Compilers, & Tools for Real-Time Systems*, pp. 88–98, 1995.

[27] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[28] A. Metzner, "Why model checking can improve WCET analysis," in *CAV, Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings* (R. Alur and D. Peled, eds.), vol. 3114 of *Lecture Notes in Computer Science*, pp. 334–347, Springer, 2004.

[29] Microchip Technologies Inc., *PIC12F629/675 Data Sheet - 8-Pin FLASH-Based 8-Bit CMOS Microcontrollers*, 2003.

[30] G. Ottosson and M. Sjödin, "Worst-case execution time analysis for modern hardware architectures," in *ACM SIGPLAN 1997 Workshop on Languages, Compilers, and Tools for Real-Time Systems (LCT-RTS'97)*, 1997.

[31] C. Y. Park, "Predicting program execution times by analyzing static and dynamic program paths," *Real-Time Systems*, vol. 5, no. 1, pp. 31–62, 1993.

[32] C. Y. Park and A. C. Shaw, "Experiments with a program timing tool based on source-level timing schema," *Computer*, vol. 24, no. 5, pp. 48–57, 1991.

[33] P. Puschner and C. Koza, "Calculating the maximum execution time of real-time programs," *Real-Time Systems*, vol. 1, no. 2, pp. 159–176, 1989.

[34] P. P. Puschner and A. V. Schedl, "Computing maximum task execution times - A graph-based approach," *Real-Time Systems*, vol. 13, no. 1, pp. 67–91, 1997.

[35] SRI International, "Symbolic analysis laboratory." `http://sal.csl.sri.com/`, 2006.

[36] J. Sun, "Documentation and tools to support worst case execution time analysis," Master's thesis, Dept. of Computing and Software, McMaster University, April 2005.

[37] H. Theiling, "Extracting safe and precise control flow from binaries," in *RTCSA*, pp. 23–30, IEEE Computer Society, 2000.

[38] H. Theiling, C. Ferdinand, and R. Wilhelm, "Fast and precise WCET prediction by separated cache and path analyses," *Real-Time Systems*, vol. 18, no. 2/3, pp. 157–179, 2000.

[39] Tidorum Ltd., "Bound-t execution time analyzer." `http://www.tidorum.fi/bound-t/`, 2006.

[40] TU-Vienna, "calc_wcet_167." `http://www.vmars.tuwien.ac.at/~raimund/calc_wcet/`, 2004.

[41] J. Turley, "The two percent solution." `http://www.embedded.com/showArticle.jhtml?articleID=9900861`, December 2002.

[42] UPPAAL, "About uppaal." `http://uppaal.com/`, 2006.

[43] A. Wassyng, M. Lawford, and X. Hu, "Timing tolerances in safety-critical software," in *FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Newcastle, UK, July 18-22, 2005, Proceedings* (J. Fitzgerald, I. J. Hayes, and A. Tarlecki, eds.), vol. 3582 of *Lecture Notes in Computer Science*, pp. 157–172, Springer, 2005.

[44] I. Wenzel, R. Kirner, B. Rieder, and P. Puschner, "Measurement-based worst-case execution time analysis," *SEUS*, vol. 00, pp. 7–10, 2005.

[45] I. Wenzel, B. Rieder, R. Kirner, and P. P. Puschner, "Automatic timing model generation by CFG partitioning and model checking," in *DATE*, pp. 606–611, IEEE Computer Society, 2005.

[46] R. Wilhelm, "Why AI + ILP is good for WCET, but MC is not, nor ILP alone," in *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings* (B. Steffen and G. Levi, eds.), vol. 2937 of *Lecture Notes in Computer Science*, pp. 309–322, Springer, 2004.

[47] World Wide Web Consortium, "The extensible stylesheet language family (xsl)." `http://www.w3.org/Style/XSL/`, 2006.

Appendix A

Timing Analysis for the IBM 1800

A.1 IBM1800 Overview

The IBM 1800 Data Acquisition and Control System is a legacy control computer developed in the 1960s and 1970s. The IBM1800 was designed for real-time process control and high-speed data acquisition applications. It has been used as an industrial control computer for automation of production plants to power generating stations.

A.1.1 Architecture and Instruction Set Details

The IBM1800 instruction set architecture is composed of 32 single and/or double 16-bit word instructions. In contrast to most current architectures that follow the separation principal, the IBM1800 stores instructions and data together in a common address space on a core storage device. The instruction set includes complex arithmetic, logic, and control-flow operations. The processor contains several registers for working data, in particular three Index Registers (XR1, XR2, XR3).

The Index Registers are used as counters, typically for computing memory offsets, loop iteration counts, or for branching conditions. The architectures includes three specific instructions for manipulation of the Index Registers. Literal or memory location values can be loaded and stored via the *LDX* and *STX* instructions respectively. The third instruction performs an addition operation on the value of an Index Register with the value of the operand, *disp*, then skips the next instruction if the result is equal to zero or changes sign, illustrated in Table A.1. Thus, only a subset of the

data-flow is required since loops can be modelled considering only the values in the Index Registers and the operations acting upon them.

Table A.1: Control-flow behaviour of the MDX instruction

Initial Value	No Skip	Skip
$XR_i > 0$	$XR_i + D > 0$	$XR_i + D \leq 0$
$XR_i \leq 0$	$XR_i + D < 0$	$XR_i + D \geq 0$

Subroutine calls are initiated with the BSI instruction that stores the return address (i.e., the program counter) to the memory address given by the operand, and begins executing the instruction following that address. Control is returned to the calling routine with a BSC or BSI instruction referencing the same memory address as its target memory address. Thus, the dynamic control-flow can be modelled using the target address as its guard. Further, due to the BSI method of calling subroutines recursion is not supported by the IBM1800.

A.2 IBM1800 Timing Analysis Transformations

The following table lists the all the instruction set formats. It includes the location invariant for the instruction and transition guard and updates for the transition that satisfies the condition field. No condition indicates a location with a single transition. [Indirect] indicates an indirect addressing instruction format and is not implemented in STARTS. The instruction clock variable reset (`x = 0`) is omitted from all updates for space and readability considerations. Further, $\Delta(x)$ and $\Sigma(x)$ in the Guards are replace by the following conditions:

$\Delta(x) \equiv (XRx > 0 \wedge XRx + disp > 0) \vee (XRx \leq 0 \wedge XRx + disp < 0)$

$\Sigma(x) \equiv (XRx > 0 \wedge XRx + disp \leq 0) \vee (XRx \leq 0 \wedge XRx + disp \geq 0)$

Opcode	Format	Tag	Target Condition	Invariant	Guard	Update
A	0	0-3		$x \leq 17$	$x = 17$	
A	1	0		$x \leq 24$	$x = 24$	
A	1	1-3		$x \leq 25$	$x = 25$	
AD	0	0-3		$x \leq 27$	$x = 27$	

AD	1	0		$x \leq 33$	$x = 33$	
AD	1	1-3		$x \leq 35$	$x = 35$	
AND	0	0-3		$x \leq 17$	$x = 17$	
AND	1	0		$x \leq 24$	$x = 24$	
AND	1	1-3		$x \leq 25$	$x = 25$	
BSC	0	0-3		$x \leq 8$	$x = 8$	
BSC	1	0-3	= Source+1	$x \leq 16$	$x = 8$	
BSC	1	0	$\neq$ Source+1	$x \leq 16$	$x = 16$	
BSC	1	1-3	= Source+1	$x \leq 17$	$x = 8$	
BSC	1	1-3	[Indirect]	$x \leq 17$	$x = 17$	
BSI (short)	0	0-3	= Source+1	$x \leq 15$	$x = 8$	
BSI (long)	0	0-3	= Source+2	$x \leq 15$	$x = 8$	
BSI (short)	0	0-3	$\neq$ Source+1	$x \leq 15$	$x = 15$	['operands'] $\leftarrow$ Source+1
BSI (long)	0	0-3	$\neq$ Source+2	$x \leq 15$	$x = 15$	['operands'] $\leftarrow$ Source+2
BSI (short)	1	0-3	= Source+1	$x \leq 24$	$x = 8$	
BSI (long)	1	0-3	= Source+2	$x \leq 24$	$x = 8$	
BSI (short)	1	0	$\neq$ Source+1	$x \leq 24$	$x = 24$	['operands'] $\leftarrow$ Source+1
BSI (long)	1	0	$\neq$ Source+2	$x \leq 24$	$x = 24$	['operands'] $\leftarrow$ Source+2
BSI (short)	1	1-3	$\neq$ Source+1	$x \leq 25$	$x = 25$	['operands'] $\leftarrow$ Source+1
BSI (long)	1	1-3	$\neq$ Source+2	$x \leq 25$	$x = 25$	['operands'] $\leftarrow$ Source+2
CMP	0	0-3	all	$x \leq 18$	$x = 18$	
CMP	1	1-3	all	$x \leq 25$	$x = 25$	
CMP	1	1-3	all	$x \leq 26$	$x = 26$	
D	0	0-3		$x \leq 171$	$x = 171$	
D	1	1-3		$x \leq 176$	$x = 176$	
D	1	1-3		$x \leq 178$	$x = 178$	
DCM	0	0-3	all	$x \leq 27$	$x = 27$	
DCM	1	1-3	all	$x \leq 33$	$x = 33$	
DCM	1	1-3	all	$x \leq 35$	$x = 35$	
EOR	0	0-3		$x \leq 17$	$x = 17$	
EOR	1	0		$x \leq 24$	$x = 24$	
EOR	1	1-3		$x \leq 25$	$x = 25$	
LD	0	0-3		$x \leq 17$	$x = 17$	
LD	1	1-3		$x \leq 24$	$x = 24$	
LD	1	1-3		$x \leq 25$	$x = 25$	
LDD	0	0-3		$x \leq 25$	$x = 25$	
LDD	1	1-3		$x \leq 32$	$x = 32$	
LDD	1	1-3		$x \leq 33$	$x = 33$	
LDS				$x \leq 8$	$x = 8$	
LDX	0	0		$x \leq 9$	$x = 9$	
LDX	0	1		$x \leq 9$	$x = 9$	XR1 $\leftarrow$ 'operands'
LDX	0	2		$x \leq 9$	$x = 9$	XR2 $\leftarrow$ 'operands'
LDX	0	3		$x \leq 9$	$x = 9$	XR3 $\leftarrow$ 'operands'
LDX	1	0		$x \leq 15$	$x = 15$	
LDX	1	1		$x \leq 15$	$x = 15$	XR1 $\leftarrow$ sub('binary',5)
LDX	1	2		$x \leq 15$	$x = 15$	XR2 $\leftarrow$ sub('binary',5)
LDX	1	3		$x \leq 15$	$x = 15$	XR3 $\leftarrow$ sub('binary',5)

M	0	0-3		$x \leq 61$	$x = 61$	
M	1	1-3		$x \leq 68$	$x = 68$	
M	1	1-3		$x \leq 69$	$x = 69$	
MDX	0	0		$x \leq 10$	$x = 10$	
MDX	0	1	No Skip	$x \leq 10$	$x = 10 \wedge \Delta(1)$	XR1 $\leftarrow$ 'operands'
MDX	0	1	Skip	$x \leq 10$	$x = 10 \wedge \Sigma(1)$	XR1 $\leftarrow$ 'operands'
MDX	0	2	No Skip	$x \leq 10$	$x = 10 \wedge \Delta(2)$	XR2 $\leftarrow$ 'operands'
MDX	0	2	Skip	$x \leq 10$	$x = 10 \wedge \Sigma(2)$	XR2 $\leftarrow$ 'operands'
MDX	0	3	No Skip	$x \leq 10$	$x = 10 \wedge \Delta(3)$	XR3 $\leftarrow$ 'operands'
MDX	0	3	Skip	$x \leq 10$	$x = 10 \wedge \Sigma(3)$	XR3 $\leftarrow$ 'operands'
MDX	1 (L)	0		$x \leq 41$	$x = 41$	
MDX	1 (L)	1	No Skip	$x \leq 41$	$x = 41 \wedge \Delta(1)$	XR1 $\leftarrow$ ['operands']
MDX	1 (L)	1	Skip	$x \leq 41$	$x = 41 \wedge \Sigma(1)$	XR1 $\leftarrow$ ['operands']
MDX	1 (L)	2	No Skip	$x \leq 41$	$x = 41 \wedge \Delta(2)$	XR2 $\leftarrow$ ['operands']
MDX	1 (L)	2	Skip	$x \leq 41$	$x = 41 \wedge \Sigma(2)$	XR2 $\leftarrow$ ['operands']
MDX	1 (L)	3	No Skip	$x \leq 41$	$x = 41 \wedge \Delta(3)$	XR3 $\leftarrow$ ['operands']
MDX	1 (L)	3	Skip	$x \leq 41$	$x = 41 \wedge \Sigma(3)$	XR3 $\leftarrow$ ['operands']
MDX	1 (I)	0		$x \leq 19$	$x = 19$	
MDX	1 (I)	1	No Skip	$x \leq 19$	$x = 19 \wedge \Delta(1)$	[Indirect]
MDX	1 (I)	1	Skip	$x \leq 19$	$x = 19 \wedge \Sigma(1)$	[Indirect]
MDX	1 (I)	2	No Skip	$x \leq 19$	$x = 19 \wedge \Delta(2)$	[Indirect]
MDX	1 (I)	2	Skip	$x \leq 19$	$x = 19 \wedge \Sigma(2)$	[Indirect]
MDX	1 (I)	3	No Skip	$x \leq 19$	$x = 19 \wedge \Delta(3)$	[Indirect]
MDX	1 (I)	3	Skip	$x \leq 19$	$x = 19 \wedge \Sigma(3)$	[Indirect]
NOP				$x \leq 8$	$x = 8$	
OR	0	0-3		$x \leq 17$	$x = 17$	
OR	1	0		$x \leq 24$	$x = 24$	
OR	1	1-3		$x \leq 25$	$x = 25$	
S	0	0-3		$x \leq 17$	$x = 17$	
S	1	0		$x \leq 24$	$x = 24$	
S	1	1-3		$x \leq 25$	$x = 25$	
SD	0	0-3		$x \leq 27$	$x = 27$	
SD	1	0		$x \leq 33$	$x = 33$	
SD	1	1-3		$x \leq 35$	$x = 35$	
SLA		0	'operands'-4 ≤ 0	$x \leq 8$	$x = 8$	
SLA		0	'operands'-4 < 0	$x \leq$'operands'+4	$x =$ 'operands'+4	
SLA		1-3		$x \leq 67$	$x \geq 8 \wedge x \leq 67$	
SLC		0	'operands'-4 ≤ 0	$x \leq 8$	$x = 8$	
SLC		0	'operands'-4 < 0	$x \leq$'operands'+4	$x =$ 'operands'+4	
SLC		1-3		$x \leq 69$	$x \geq 10 \wedge x \leq 69$	
SLCA		0	'operands'-4 ≤ 0	$x \leq 8$	$x = 8$	
SLCA		0	'operands'-4 < 0	$x \leq$'operands'+4	$x =$ 'operands'+4	
SLCA		1-3		$x \leq 69$	$x \geq 10 \wedge x \leq 69$	
SLT		0	'operands'-4 ≤ 0	$x \leq 8$	$x = 8$	
SLT		0	'operands'-4 < 0	$x \leq$'operands'+4	$x =$ 'operands'+4	
SLT		1-3		$x \leq 67$	$x \geq 8 \wedge x \leq 67$	
SRA		0	'operands'-4 ≤ 0	$x \leq 8$	$x = 8$	

SRA		0	'operands'-4 < 0	$x \leq$'operands'+4	$x =$ 'operands'+4	
SRA		1-3		$x \leq 67$	$x \geq 8 \wedge x \leq 67$	
SRT		0	'operands'-4 $\leq$ 0	$x \leq 8$	$x = 8$	
SRT		0	'operands'-4 < 0	$x \leq$'operands'+4	$x =$ 'operands'+4	
SRT		1-3		$x \leq 67$	$x \geq 8 \wedge x \leq 67$	
STD	0	0-3		$x \leq 25$	$x = 25$	
STD	1	1-3		$x \leq 32$	$x = 32$	
STD	1	1-3		$x \leq 33$	$x = 33$	
STO	0	0-3		$x \leq 17$	$x = 17$	
STO	1	1-3		$x \leq 24$	$x = 24$	
STO	1	1-3		$x \leq 25$	$x = 25$	
STS	0	0-3		$x \leq 15$	$x = 15$	
STS	1	0		$x \leq 24$	$x = 24$	
STS	1	1-3		$x \leq 25$	$x = 25$	
STX	0	0-3		$x \leq 15$	$x = 15$	
STX	1	0-3		$x \leq 24$	$x = 24$	
RTE		0	'operands'-4 $\leq$ 0	$x \leq 8$	$x = 8$	
RTE		0	'operands'-4 < 0	$x \leq$'operands'+4	$x =$ 'operands'+4	
RTE		1-3		$x \leq 67$	$x \geq 8 \wedge x \leq 67$	
WAIT				$x \leq 8$	$x = 8$	
XIO	0	0-3		$x \leq 33$	$x \geq 25 \wedge x \leq 33$	
XIO	1	0		$x \leq 40$	$x \geq 32 \wedge x \leq 40$	
XIO	1	1-3		$x \leq 41$	$x \geq 33 \wedge x \leq 41$	

Appendix B

Timing Analysis for the PIC Microcontroller

B.1 PIC Overview

The series of PIC microcontrollers are a RISC family of programmable processors produced by Microchip Technology Inc. The PIC12F629/75 model [29] of the microcontroller is the target architecture implementation of the PIC instruction set architecture. It is a CMOS Flash-based 8-bit microcontroller architecture in an 8-pin package and features 4 channels for the 10-bit Analog-to-Digital (A/D) converter, 1 channel comparator and 128 bytes of EEPROM data memory. This device is used for automotive, industrial, appliances and consumer entry-level product applications that require field re-programmability. The PIC microcontroller is used for a wide range of embedded systems from simple controllers, such as the Apple iPod remote, to hard real-time motor controllers and communication system components. Other models of the PIC will have a similar hardware model to the one developed for the STARTS tool suite.

B.1.1 Architecture and Instruction Set Features

The PIC instruction set is small, 35 single word instructions, and highly orthogonal. Each instruction executes in one instruction cycle (4 oscillator cycles), except branching instructions that may take two cycles. The instructions with transitions

of different execution delays are in the Non-Uniform Guarded Branching class of instructions.

The arithmetic operations, addition and subtraction, on literal values performs a two's complement operation. Literal operands representing negative numbers are encoded in the range 128 to 255 and must be converted to its correct negative value prior to performing the operation in the model.

The data memory map includes several special registers that are read and/or written to. They contain or set configuration information about the processor state and its auxiliary features. Some of these features include setting the processors clock rate, A/D convertor, interrupt timers, or reading the processor status register, interrupt and peripheral control registers. The other registers are general purpose and are used by the executing program for local variables and constants. There are no special registers for loop bounds and counters, thus the entire data-flow must be modelled to automatically determine loop iteration bounds.

The A/D conversion is initiated by setting bit 1 of the ADCON register to 1. The processor performs the analog data acquisition, converts it to a digital representation and clears the set bit, generating an interrupt if enabled. The model of the A/D conversion is done in a separate automaton that is a higher priority than the program model automaton.

The processor stack is eight levels deep and stores the return instruction address of the next instruction from the calling routine (i.e. the value of the program counter). It operates a circular buffer should the call depth grow larger than eight, thus losing the first values pushed on the stack. The instruction set cannot directly read or write to the stack, and there are no status bits set to indicate stack overflow or underflow. Therefore, the model of the stack does not permit a call depth greater than 8 and the model checker will discover those programs that violate the maximum call depth.

B.2 PIC Timing Analysis Transformations

The following table lists the all the instruction set formats. It includes the location invariant for the instruction and transition guard and updates for the transition that satisfies the condition field. No condition indicates a location with a single transition. The instruction clock variable reset (`x = 0`) is omitted from all updates for space

and readability considerations. Memory locations and literal values are indicated by italicised variable name (e.g., W, f, d, k), and the contents of memory locations are indicated by parentheses. Stared (*) opcodes indicate Non-Uniform Guarded Branching instructions that have different execution times depending on the transition.

Opcode	Operands	Condition	Invariant	Guard	Update
ADDLW	k		$x \leq 1$	$x = 1$	$(W) \leftarrow (W) + twos(k)$
ADDWF	f, d	$d = 0$	$x \leq 1$	$x = 1$	$(W) \leftarrow (W) + (f)$
ADDWF	f, d	$d = 1$	$x \leq 1$	$x = 1$	$(f) \leftarrow (W) + (f)$
ANDLW	k		$x \leq 1$	$x = 1$	$(W) \leftarrow (W)\&k$
ANDWF	f, d	$d = 0$	$x \leq 1$	$x = 1$	$(W) \leftarrow (W)\&(f)$
ANDWF	f, d	$d = 1$	$x \leq 1$	$x = 1$	$(f) \leftarrow (W)\&(f)$
BCF	f, b		$x \leq 1$	$x = 1$	$(f) \leftarrow (f)\&(255 - 2^b)$
BSF	f, b		$x \leq 1$	$x = 1$	$(f) \leftarrow (f)\|(255 - 2^b)$
BTFSC*	f, b	No skip	$x \leq 1$	$x = 1 \wedge (f)\&2^b \neq 0$	
BTFSC*	f, b	Skip	$x \leq 2$	$x = 2 \wedge (f)\&2^b = 0$	
BTFSS*	f, b	No skip	$x \leq 1$	$x = 1 \wedge (f)\&2^b = 0$	
BTFSS*	f, b	Skip	$x \leq 2$	$x = 2 \wedge (f)\&2^b \neq 0$	
CALL	k		$x \leq 2$	$x = 2$	stack[tos] ← 'address'+1, tos ← tos+1
CLRF	f		$x \leq 1$	$x = 1$	$(f) \leftarrow 0$
CLRW			$x \leq 1$	$x = 1$	$(W) \leftarrow 0$
CLRWDT			$x \leq 1$	$x = 1$	
DECF	f, d	$d = 0$	$x \leq 1$	$x = 1$	$(W) \leftarrow (f) - 1$
DECF	f, d	$d = 1$	$x \leq 1$	$x = 1$	$(f) \leftarrow (f) - 1$
DECFSZ*	f, d	$d = 0 \wedge$ No Skip	$x \leq 1$	$x = 1$	$(W) \leftarrow (f) - 1$
DECFSZ*	f, d	$d = 0 \wedge$ Skip	$x \leq 2$	$x = 2$	$(W) \leftarrow (f) - 1$
DECFSZ*	f, d	$d = 1 \wedge$ No Skip	$x \leq 1$	$x = 1$	$(f) \leftarrow (f) - 1$
DECFSZ*	f, d	$d = 1 \wedge$ Skip	$x \leq 1$	$x = 1$	$(f) \leftarrow (f) - 1$
GOTO	k		$x \leq 1$	$x = 1$	
INCF	f, d	$d = 0$	$x \leq 1$	$x = 1$	$(W) \leftarrow (f) + 1$
INCF	f, d	$d = 1$	$x \leq 1$	$x = 1$	$(f) \leftarrow (f) + 1$
INCFSZ*	f, d	$d = 0 \wedge$ No Skip	$x \leq 1$	$x = 1$	$(W) \leftarrow (f) + 1$
INCFSZ*	f, d	$d = 0 \wedge$ Skip	$x \leq 2$	$x = 2$	$(W) \leftarrow (f) + 1$
INCFSZ*	f, d	$d = 1 \wedge$ No Skip	$x \leq 1$	$x = 1$	$(f) \leftarrow (f) + 1$
INCFSZ*	f, d	$d = 1 \wedge$ Skip	$x \leq 1$	$x = 1$	$(f) \leftarrow (f) + 1$
IORLW	k		$x \leq 1$	$x = 1$	$(W) \leftarrow (W)\|k$
IORWF	f, d	$d = 0$	$x \leq 1$	$x = 1$	$(W) \leftarrow (W)\|(f)$
IORWF	f, d	$d = 1$	$x \leq 1$	$x = 1$	$(f) \leftarrow (W)\|(f)$
MOVF	f, d	$d = 0$	$x \leq 1$	$x = 1$	$(W) \leftarrow (f)$
MOVF	f, d	$d = 1$	$x \leq 1$	$x = 1$	$(f) \leftarrow (f)$
MOVLW	k		$x \leq 1$	$x = 1$	$(W) \leftarrow twos(k)$
MOVWF	f		$x \leq 1$	$x = 1$	$(f) \leftarrow (W)$
NOP			$x \leq 1$	$x = 1$	
RETFIE			$x \leq 2$	$x = 2 \wedge$ stack[tos-1] = TgtAddr	tos ← tos-1
RETLW	k		$x \leq 2$	$x = 2 \wedge$ stack[tos-1] = TgtAddr	$(W) \leftarrow k$, tos ← tos-1
RETURN			$x \leq 2$	$x = 2 \wedge$ stack[tos-1] = TgtAddr	tos ← tos-1

RLF	f, d	$d = 0$	$x \leq 1$	$x = 1$	$(W) \leftarrow (f) << 1$
RLF	f, d	$d = 1$	$x \leq 1$	$x = 1$	$(f) \leftarrow (f) << 1$
RRF	f, d	$d = 0$	$x \leq 1$	$x = 1$	$(W) \leftarrow (f) >> 1$
RRF	f, d	$d = 1$	$x \leq 1$	$x = 1$	$(f) \leftarrow (f) >> 1$
SLEEP			$x \leq 1$	$x = 1$	
SUBLW	k		$x \leq 1$	$x = 1$	$(W) \leftarrow twos(k) - (W)$
SUBWF	f, d	$d = 0$	$x \leq 1$	$x = 1$	$(W) \leftarrow (f) - (W)$
SUBWF	f, d	$d = 1$	$x \leq 1$	$x = 1$	$(f) \leftarrow (f) - (W)$
XORLW	k		$x \leq 1$	$x = 1$	$(W) \leftarrow (W)\hat{}k$
XORWF	f, d	$d = 0$	$x \leq 1$	$x = 1$	$(W) \leftarrow (W)\hat{}(f)$
XORWF	f, d	$d = 1$	$x \leq 1$	$x = 1$	$(f) \leftarrow (W)\hat{}(f)$