

MACHINE CODE VERIFICATION USING THE BOGOR
FRAMEWORK

by

Joseph R. Edelman

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

August 2008

Copyright © 2008 Joseph R. Edelman

All Rights Reserved

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Joseph R. Edelman

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date

Eric Mercer, Chair

Date

Michael Jones

Date

Daniel Zappala

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Joseph R. Edelman in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

Eric Mercer
Chair, Graduate Committee

Accepted for the
Department

Parris K. Egbert
Graduate Coordinator

Accepted for the
College

Thomas W. Sederberg
Associate Dean, College of Physical and Mathematical
Sciences

ABSTRACT

MACHINE CODE VERIFICATION USING THE BOGOR FRAMEWORK

Joseph R. Edelman

Department of Computer Science

Master of Science

Verification and validation of embedded systems software is tedious and time consuming. Software model checking uses a tool-based approach automating this process. In order to more accurately model software it is necessary to provide hardware support that enables the execution of software as it should run on native hardware. Hardware support often requires the creation of model checking tools specific to the instruction set architecture. The creation of software model checking tools is non-trivial. We present a strategy for using an "off-the-shelf" model checking tool, Bogor, to provide support for multiple instruction set architectures. Our strategy supports key hardware features such as instruction execution, exceptional control flow, and interrupt servicing as extensions to Bogor. These extensions work within the tool framework using existing interfaces and require significantly less code than creating an entire model checking tool.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Eric Mercer, for his example, enduring patience, and most of all for being my friend. I have grown tremendously under his guidance. I would also like to thank Dr. Michael Jones for his encouragement and boundless optimism. Every smile, joke, and kind word made a difference. To Rahul Kumar, Neha Rungta, and Joel Self I cannot begin to express how much I treasure our friendships and the experiences we have shared during this time. Thank you for walking this path with me. Lastly, I would like to thank my parents, Rod & Kathy Edelman, for instilling in me a true love of learning.

Table of Contents

1	Introduction	1
2	Scope	5
3	Related Work	9
4	Modeling Exceptional Control Flow	13
5	Translation and Language Extension	17
6	Case Study	22
6.1	Motorola M68hc11 Language Extension	22
6.2	Timing	24
6.3	Thread Scheduler Verification	27
6.4	Multi-threaded Program with Error	29
6.4.1	Figure 6.4(a) Walkthrough	30
6.4.2	Figure 6.4(b) Walkthrough	31
6.4.3	Figure 6.5 Walkthrough	32
6.5	Software Engineering	34
7	Conclusions and Future Work	36
	Bibliography	38

Chapter 1

Introduction

Model checking software is necessary to compensate for shortcomings in less formal testing practices that leave a majority of the potential behavior or state space of software untested. Within these untested areas lie the potential for software defects that can result in inconvenient calculations or complete system failure. As software increases in complexity, the divide between tested behavior and unknown behavior increases along with the cost associated with fixing software defects. The use of formal methods complements current testing practices by providing conclusive evidence of the actual software program behavior.

Model Checking is a type of formal verification that uses a software tool to automatically verify that a system is a model of the specified property. The system is modeled as a directed graph where a node represents the current state of the system (variable values) and the edges connecting nodes are transitions that transform one state into another. The tool performs the tedious and repetitive task of exhaustively exploring the state space of the system to conclude that a property holds, or it produces a counter-example invalidating the property.

Model checking software at the machine code level exposes meaningful information to a software developer for embedded systems. At this level, the specifics of how the software runs on the target hardware are visible. The main obstacles to model checking machine code are modeling a complex processor and exceptional control flow. Exceptional control flow is a change in control flow due to run-time

conditions. These conditions may be handling an interrupt or executing a jump instruction that uses run-time information to calculate the destination. Every processor supports basic classes of instructions: arithmetic, logical, load and store, testing and branching, and input and output operations. The implementation for individual instructions is vastly different depending on the intended application, size of memory, and addressing modes. Exceptional control flow occurs when the currently executing program is pre-empted due to interrupts. When an interrupt occurs, the current state of the running program is saved off and the interrupt service routine is given priority. Intuitively, this can be visualized as a scheduled event, key press on a keyboard, reading from a sensor, or any number of possible inputs. Modeling this behavior is difficult because each Instruction Set Architecture (ISA) services interrupts differently and the representation for the model may not lend itself to dynamic changes to control flow. In order to verify this type of software accurately, the model checking tool must support the ISA and exceptional control flow.

A current strategy for supporting an ISA and exceptional control flow is to create a new tool for each ISA using existing software simulators. The simulator supports the ISA and exceptional control flow. These tools are tightly coupled with an existing simulator or Virtual Machine, making them extremely powerful tools for the supported architectures. They operate directly on compiled code by substituting the input language of the model checking tool with an interface into the simulator. Use of existing simulators lessens the overall cost, but the cost to develop and maintain a model checking tool is beyond the capabilities of most software development organizations. Creating a new tool for each architecture is not feasible.

Another approach to supporting an ISA and exceptional control flow is to work within the input language of a general purpose model checking tool that allows the user to supply the necessary ISA semantics. Certain model checking tools have input languages that are nearly as expressive as high-level development languages.

Expressing the necessary data structures and mechanisms to simulate a processor is possible. Exceptional control flow is difficult due to a lack of explicit control over the program control flow. Input languages do not necessarily use the current state of the model to determine control flow. Some languages dictate the transition system directly through the structure of the model, with little or no dependence on the changes in state that occur at a vertex. Unless there is some mechanism to dynamically determine or alter the control flow of the model an expressive language is not enough.

The architecture of general purpose model checking tools simplify the manipulating of language semantics to model exceptional control flow. General purpose model checking tools divide the model checking problem into cohesive components that can be overridden to achieve the desired behavior. Customizing general purpose model checking tools provides a more stable platform for verification and the added flexibility of using the same tool across multiple projects. The Bogor Framework, developed by Kansas State University, is one choice for verifying machine code containing exceptional control flow using user-defined ISA semantics.

The Bogor Framework can be extended to support an ISA by defining the processor simulator as a new data type in the system model using the Language Extension feature. The Language Extension feature allows the user to define new data types and services. The processor simulator models the data structures and semantics of the target architecture through user defined modules that work within the Bogor Framework rather than customizing the model checking tool. Exceptional Control Flow is handled by an additional module, the Location Changed Listener, that correctly guides the model checking tool through a statically directed model of the software. These additional modules enable the Bogor Framework to accurately and efficiently verify machine code containing exceptional control flow for a target architecture.

We show through the verification of a thread scheduler that the Bogor Framework is a viable solution to the problem of modeling ISAs containing exceptional control flow. A thread scheduler manages a collection of threads by using timer based interrupts to periodically change the currently active thread. The scheduler we verify schedules two threads in a round-robin scheme where each thread is given the same amount of time to execute. The correct execution of a thread scheduler is entirely dependent on the hardware mechanisms that service interrupts, resulting in exceptional control flow. Without precise support for both the instruction set architecture and exceptional control flow it is not possible to correctly model a thread scheduler implementation.

Chapter 2

Scope

The scope of this work is to demonstrate that the Bogor Framework can be extended to provide hardware support allowing the formal verification of software compiled for a target architecture. Extension through the Framework decouples the development and maintenance of the tool from the state space generation. Bogor Framework Extensions compartmentalize the development task and allows the user to switch between supported architectures while using the same model checking tool. Ideally this approach could be used for any processor architecture. In practice, the key limitations are the transition granularity or timing model used, program extraction from compiled code, and the size of the resulting model. Under these guidelines we support the Motorola M68hc11 microprocessor.

Implementing hardware as a finite state machine with a high-level development language (Java) means it is feasible that a simulator can be created for most architectures. The transition granularity or timing model determines the amount of detail present. Representing time as single clock ticks verses pulses with rising or falling clock edges determines the perspective that is available when following transitions or examining the state of the system. At clock ticks portions of the processor are at various stages of executing the current instruction. While at rising and falling clock edges gates and cells are being populated. The appropriate timing model to expose the necessary system state is critical. We are interested in providing a tool for software developers so we specifically target the class of microprocessors that process

interrupts following the currently executing instruction in a non-pipelined architecture. This allows us to model time more coarsely than single clock ticks. Our unit of time is the complete execution of an instruction.

Our timing model breaks down under the circumstances that additional background processing such as interrupts resolution requires multiple clock cycles and might span the execution of several instructions. A separate but related issue is instruction pipelining. Instruction pipelining increases the throughput of the processor by executing multiple instructions in stages, effectively filling the processing pipe by utilizing more of the processor per clock cycle. The M68hc11 does not support pipelining. Support for both of these features is possible but requires changing the transition granularity to be a single cycle or less.

The Timer Input Compare and Timer Output Compare features of the m68hc11 present a complexity obstacle closely related to the timing model. Timer Input Compare records the current free running counter value when an input event is triggered. Timer Output Compare is used by a program to store a value corresponding to a future event that will be triggered when the free running counter value exceeds that of the store value. In explicit state model checking any increase in the number of variables or size of the state vector impacts the state space exponentially. We are able to mitigate the state space explosion problem involving the output compare by applying an abstraction, that is described in detail in Section 6.2, and by excluding software that uses the contents of the Timer Output Compare registers for calculations. Unfortunately this same abstraction can not be applied to the Timer Input Compare and so programs utilizing the Timer Input Compare registers are excluded.

The program being verified must be extracted directly from compiled code. A control flow graph of the possible execution paths in the program must be present in the compiled code. This excludes all self-mutating or modifying code that dy-

namically create or modify the program in memory. Self-modifying code is written for the purposes of performance optimizations, obfuscation, and/or malicious intent. Depending on the supporting hardware or operating systems, the capability to dynamically modify code may not be allowed due to the potential risk of abuse. This area represents a special class of software that introduces new difficulties and questions. For now we confine our interest to typical, non self-modifying software in an effort to keep the compiled code and the Bandera Intermediate Representation (BIR) model in a user readable form and satisfy the requirement that a BIR model is well-formed prior to model checking. BIR is Bogor's input language and is used to specify a model.

The size of the program being verified should be under 10,000 assembly instructions. Observed performance of the Bogor Framework degrades significantly as the model size increases. This is based solely on observed behavior and does not reflect the actual upper limit of the tool.

As a case study we implemented several key features found in the Motorola m68hc11 microprocessor. It is an 8-bit data, 16-bit addressable system on a chip microprocessor. Internally it has two Accumulators A and B, that combined act as Accumulator D, Index Registers X and Y, a 16-bit Stack Pointer, Program Counter, Condition Code Register, with main memory dependent on configuration. Six memory addressing modes are supported: Extended, Direct, Immediate, Inherent, Indexed, and Relative. The timing systems includes a main timer and real-time interrupt support using Output compare to trigger an interrupt. The m68hc11 is a full-featured microprocessor representative of a significant portion of the embedded systems spectrum.

We feel that the Bogor Framework is best suited for providing hardware support for 8 to 16-bit addressable microprocessors with a modest amount of resources, enough input/output to create a closed system, and running small programs that do

not dynamically alter their instructions at run-time. Supporting an ISA from within the Bogor Framework provides a common verification platform, removes the need to rely on virtual machines or simulators developed externally, and can be re-used across multiple projects and architectures.

Chapter 3

Related Work

Model checking tools vary greatly according to their intended purpose. Most model checking tools focus on a particular language, level of abstraction, or other model checking strategy. The particular problem domain directly affects the architecture of the tool and determines the application of the tool. The following list represents some relevant tools to the problem of verifying compiled software containing exceptional control flow for a target architecture.

Threaded-C Bounded Model Checking is a method for transforming a threaded C program into propositional formulas that are conjuncted with properties and given to a SAT solver that determines if a satisfying assignment exists [16]. As the depth bound on recursive function calls and loops is incremented, the propositional formula increases exponentially. This increase quickly exhausts system resources. The main contribution of the work is in applying the bounded approach to context switches between threads to expose concurrency and race conditions common to concurrent software systems. The tool operates on source code and not low-level instructions found in compiled software, requiring significant modification to solve the problem of verifying machine code for a target architecture containing exceptional control flow. Support for a target ISA could be included by creating a simulator for the architecture and verifying that the simulator executes the program correctly. Verifying the simulator implementation is a much larger and separate issue.

The BLAST tool and SLAM toolkit verify safety properties of abstracted C programs using predicate abstraction [10] [1]. Predicate abstraction is an iterative process that attempts to define a set of conditional statements, or predicates, that describe system behavior with respect to some property of the system. SLAM requires a tool to perform the initial abstraction to a boolean program, another tool that model checks Boolean programs, and a third tool to perform path feasibility that aids in creating additional predicates to refine the abstracted model. BLAST performs abstraction on-the-fly as needed. These tools also operate at the source level, obscuring details specific to the target hardware.

SPIN is one of the most mature and efficient model checking tools available [11]. The main application is the verification of asynchronous process interactions. SPIN uses the Promela input language to specify models. Promela is capable of modeling processes at both high and low levels of detail. Low-level interactions are made possible by the embedding of C code directly into the Promela model. Additional extensions have been made to the language and tool to support object-oriented techniques [3]. SPIN may be used to verify machine code by creating a simulator for an ISA in Promela using large amounts of embedded C code. This approach exposes too much detail by verifying the inner working of the simulator and indirectly the program of interest. Execution of a single instruction includes several operations within the simulator. The model checking tool explores these transitions instead of grouping them together into a single transition for the corresponding machine code instruction. This requires a tool expert to perform the modifications.

StEAM is a model checking tool that verifies C++ machine code compiled for the *Internet C Virtual Machine* (ICVM) [12] [14]. The ICVM is intended to be a platform-independent virtual machine for the C++ language. Compiling to the ICVM requires modifications to the GNU C-compiler. The ICVM is modified to support multi-threading and special user commands used to assist in model exploration. The

primary focus of StEAM is the verification of concurrency properties. The authors propose that this low-level approach to creating a virtual processor can be extended to other compiled languages. The generic and platform-independent approach removes the interesting detail in verifying how a software system runs on its intended hardware.

Java PathFinder 2 (JPF) is a model checking tool that is tightly coupled with a specialized version of the Java Virtual Machine (JVM) [18]. This tool enables the verification of Java or other languages that compile to Bytecode. The JVM abstracts any hardware dependencies. JPF verifies Bytecode as executed by the JVM. The model checking tool directs the execution of the JVM. Interfacing with the JVM overcomes difficulties found in the first JPF, when Java language features could not be supported by the target translation language, Promela [9]. Hardware abstraction removes the details necessary to verify machine code on target hardware.

NIPS uses a unique bytecode language and virtual machine as a state space generation tool that can be embedded in a model checking tool [19]. This approach creates a portable data model that can be re-used in multiple host model checking tools. NIPS provides hardware support for an ATMEL ATmega processor to verify embedded systems software. The translation from ATmega16 assembly to bytecode is mostly automated by using specifications from the manufacturer. The remaining hardware components (interrupts, timers, etc.) are replaced by non-determinism or abstractions. From the published research it is unclear if the verification of timer or interrupt based software is possible. This approach presents an exceptional amount of potential. Further investigation into the specifics of the bytecode language and modeling of hardware is warranted.

Estes is a model checking tool that verifies machine code for a target architecture by using a modified version of the GNU Debugger to execute machine code [15]. The GNU Debugger (gdb) supports a variety of processor hardware models [7][8]. The tool requires a template for each processor model that indicates how to interpret the

bytestream used to communicate between the model checking tool and the debugger. Estes currently provides support solely for the Motorola M68hc11 processor. Estes is capable of verifying software as it would actually run on target hardware. This tool provides sound metrics for comparison. It serves as a reference tool for the work in this document.

The Bogor Framework is a general purpose model checking tool. The input language, Bandera Intermediate Representation (BIR), supports the Java development language as well as user-defined data types as Language Extensions [6] [17]. BIR is capable of modeling a variety of systems ranging from design specifications, to protocols, to systems written in industry standard development languages [5]. Bogor is highly-modular, permitting the replacement and customization of core model checking components for a specific problem domain.

The Bogor Framework provides the most effective path for extending the capabilities of a model checking tool to provide hardware support. Modifications to any other tool would require significant changes to the input language and tool itself. The benefit of the using the Bogor Framework is that most of the work is done within the BIR language itself and any changes to the model checking tool are isolated to single modular components. We now move on to the details of handling exceptional control flow.

Chapter 4

Modeling Exceptional Control Flow

Bogor requires that a BIR model be completely specified and well-formed prior to exploration. This means that nodes along a control flow path must be connected to be explored properly. Exceptional control flow uses run-time information that is not available when the model is generated. We present a progression of possible approaches, outline deficiencies, and conclude with a solution to the problem.

The Bogor Framework and BIR language provide several mechanisms for resolving the problem of dynamically determining control flow: functions, guarded transitions, and listener modules to augment exploration strategies. The use of functions is not explored because this information is not present at the assembly level. We explore the structure of the model, guarded transitions, and listener modules as possibilities. Figure 4.1 illustrates the evolution of our solution through experience.

Figure 4.1(a) is the correct control flow behavior for the system being modeled. The system is composed of a single process with instructions located at locations A, B, C, and I. Location I represents an interrupt handling mechanism. Execution begins at location A, follows to B. After the instruction at location B has been executed an interrupt occurs, the current state of the processor is pushed onto the stack and control flow passes to location I. The interrupt is serviced, the contents of the stack are used to restore the state of the processor and execution continues with location C. After the instruction at location C has been executed another interrupt occurs, the current state of the processor is placed on the stack, and control flow passes to location I. The

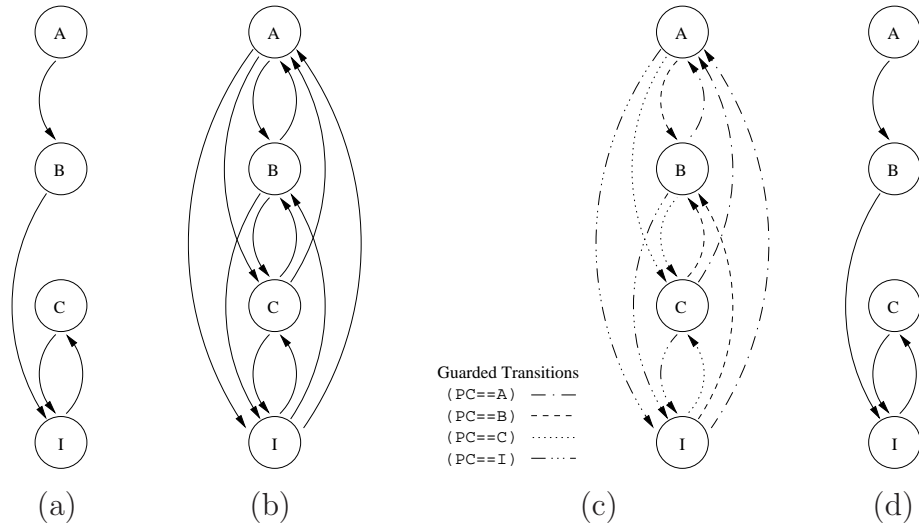


Figure 4.1: Exact control flow behavior and three possible solutions in the BIR language ranging from over approximation to a precise model. (a) The exact behavior of the actual program on the target hardware. (b) A conservative over approximation that includes false behavior. (c) A precise model that relies on guarded transitions to check the PC value for the correct transition. (d) The actual implementation using the location changed listener that is an exact model of the actual behavior.

interrupt is serviced, the processor state is restored and the instructions at location C are executed again. This is the correct behavior of the system. The difficulty in specifying correct behavior in BIR is that the interrupt changes control flow from Location B to I and C to I at run-time. This information is not available statically.

Figure 4.1(b) uses only the structure of the BIR model to determine control flow. Exceptional control flow may occur between the execution of instructions so a transition from Locations A, B, and C is made to location I. Following interrupt handling at Location I, it is possible for execution to be returned to any of A, B, or C depending on the context of the interrupt so additional transitions from I are made to these Locations. For simplicity, expected execution follows from A to B to C without any branching, so additional transitions are added to the graph. This is the assumed behavior when observing the type of instructions in the program. When a node contains multiple outgoing edges, each is treated as a point of non-determinism and is explored individually by the model checking tool. The resulting graph, is now

an over-approximation of the possible behavior of the model. Verifying this model finds relevant errors, if any, in the execution of the program. Additional errors that are not feasible are also reported.

The strategy in Figure 4.1(b) contains two main drawbacks: interrupt servicing frequency and proper return from interrupt servicing. The figure in Figure 4.1(b) demonstrates fully interleaved behavior between the program and interrupt servicing. When the model checking tool follows edges on the graph the notion of successor and predecessor information is implicit in the depth first search stack but not explicit in the model description. It is currently not possible to transition to the interrupt resolution node, pass control flow to the appropriate interrupt handler, and return control flow to the correct node where the interrupt occurred. The dynamic information stored in the model indicating how control flow should proceed is not accessible. Without this information, Location I incorrectly contains non-deterministic transitions to each node in the program. The current model does not demonstrate actual behavior of the system.

The next logical step, illustrated by Figure 4.1, to more accurately represent correct behavior is to use the Program Counter information that is stored in the state of the system. Expanding on the strategy in Figure 4.1(b) the graph is now fully connected. However, a guard is placed on every outgoing transition that requires the Program Counter in the current state of the system be equal to the location label of the next location to explore. Upon exploring a node, the model checking tool evaluates the conditional statement on each outward transition, queueing up the next set of nodes to explore. For example, after executing the instruction found at location A, the Program Counter is set to B. Each guard on the three outgoing transitions: A to B, A to C, and A to I are evaluated. Since the Program Counter has the value B, only the guard for the transition A to B is satisfied, so Location B is placed on the search stack. The use of guarded transitions and run-time information to prune

infeasible paths to determine the next location correctly models the behavior of the system. The only significant drawback to this approach is the cost associated with guard evaluation at every location. The model checking tool evaluates the guard of every outgoing transition on every visit to that location. As the size of the program and/or memory increase the feasibility of using this solution lessens.

Figure 4.1(d) uses an additional user-supplied plugin module that registers with the *Location Changed Event* in the model checking tool to perform post-processing that correctly guides the search strategy. After processing a node and calculating the next node, the *Location Changed Event* is triggered. The *Location Changed Listener* extracts the current Program Counter from the current state of the model, sets the next location to visit, and returns control to the search strategy. The structure of the model is irrelevant and complicated transitions are unnecessary. The *Location Changed Listener* maps the control flow behavior of the ISA to the BIR model. This allows Bogor to correctly verify software containing exceptional control flow while using a static model of the system in a more efficient manner than strictly using BIR to specify transitions.

Having solved the problem of accurately representing system behavior in the BIR language we can now address how to leverage the Bogor Framework to simulate the ISA on which the software system runs.

Chapter 5

Translation and Language Extension

BIR contains high and low level language constructs. High-level BIR more closely resembles source code in other languages. Low level BIR is of particular interest because it exposes the lowest level of transition granularity used by the model checking tool. Low-level BIR is made of finite state machines containing nodes and transitions. Primitive and user-defined data types, arithmetic, logical, and control operators are available. Simulating the operations of hardware on compiled code at this level is ideal.

Figure 5.1 illustrates a simple mutual exclusion algorithm and its equivalent BIR model [2]. Figure 5.1(a) has a global variable, *turn*, that determines which process is allowed to enter its critical section, labeled as CR_0 and CR_1 respectively. A process consists of an infinite loop labeled as L_0 and L_1 that contains a busy wait on the turn variable, labeled as NC_0 and NC_1 . Once the wait condition is satisfied, a process is allowed to enter its critical section and modify the turn variable that allows the other process to enter its critical section on the next context switch.

Translating Figure 5.1(a) into the BIR representation in Figure 5.1(b) begins with defining the **system** that contains global variables and FSMs for each process. The global *turn* variable used in (a) is mapped to a global variable in (b). To completely match model (a), model (b) should non-deterministically initialize *turn* to both 0 and 1 during exploration. For simplicity we initialize *turn* to 0. To model non-determinism, an additional starting location would need to be added. A **system**

<pre> P₀:: L₀: while True do NC₀: wait(turn = 0); CR₀: turn := 1; end while; P₁:: L₁: while True do NC₁: wait(turn = 1); CR₁: turn := 0; end while; </pre>	<pre> system ProcessExample { int turn := 0; active thread P0() { loc L0: when (true) do {} goto NC0; loc NC0: when (turn == 0) do {} goto CR0; loc CR0: do { turn := 1; } goto L0; } active thread P1() { loc L1: when (true) do {} goto NC1; loc NC1: when (turn == 1) do {} goto CR1; loc CR1: do { turn := 0; } goto L1; } } </pre>
(a)	(b)

Figure 5.1: A simple mutual exclusion algorithm where two processes rely upon a shared global variable *turn* to arbitrate access to a critical section. (a) The algorithm defined in a high-level language. (b) The BIR equivalent of the mutual exclusion algorithm.

may have any number threads. At least one thread is active, indicated by the **active** keyword. The first location of each active thread is the set of initial locations to explore when model checking begins. Both P_0 and P_1 are mapped to active threads.

Threads are a collection of connected locations as seen with L_0 , NC_0 , CR_0 and L_1 , NC_1 , CR_1 (b). Each line of (a) is mapped to a location in (b). A location is the smallest unit of schedule-able work used by the model checking tool. Locations within a thread have unique labels specified after the **loc** keyword. The **when** keyword may be guarded by a boolean expression that must be satisfied prior to executing the **do**

block. Locations $L0$ and $L1$ contain trivial guards but the guards for $NC0$ and $NC1$ are dependent on the value of the *turn* variable as in (a). The body of the **do** block contains the operations to execute. Operations are executed in order. After execution the tool transitions to the next location specified following the **goto** keyword.

The **goto** locations specified in (b) correspond to the control flow in (a). In Figure 5.1(a) P_0 execution begins at L_0 , transitions to NC_0 , then to CR_0 , and finally to L_0 . P_1 is similar in this regard. This mapping creates a set of very simple state machines. The BIR model in (b) is identical to the behavior of (a) with the exception of the initialization of *turn*.

Figure 5.2(a) is the assembly instructions for process P0 in Figure 5.1(a) on a PowerPC G4 processor. Verifying this implementation of Figure 5.1(a) requires a Language Extension to support the ISA of the target hardware and a location changed listener to determine control flow. Assuming these are present, we outline the process of using a Language Extension to simulate hardware.

The BIR model in Figure 5.2(b) introduces the interface for communicating with the Language Extension. The interface, identified by the **extension** keyword, defines the data type, a creation method for instantiating the data type, and a method for executing instructions. The global variable *data* contains the reference to an instance of the *SimpleProcessor* language extension. This instance stores the current state of the processor (registers, memory, program counter).

The thread implementation for $P0$ in Figure 5.2(b) is organized in a similar fashion as Figure 5.1(b) with individual instructions located within the **do** block of a location labeled with the memory address. Language Extensions cannot be declared and instantiated in a single statement so an additional location, labeled *initialize*, is added to the active thread to instantiate the language extension. This location is added to improve readability. Initialization must take place prior to use.

<pre> 2d20: 3c 4a 00 00 addis r2,r10,0 2d24: 38 42 03 08 addi r2,r2,776 2d28: 80 02 00 00 lwz r0,0(r2) 2d2c: 2f 80 00 00 cmpwi cr7,r0,0 2d30: 40 9e ff f0 bne+ cr7,2d20 2d34: 3c 4a 00 00 addis r2,r10,0 2d38: 38 42 03 08 addi r2,r2,776 2d3c: 38 00 00 01 li r0,1 2d40: 90 02 00 00 stw r0,0(r2) 2d44: 4b ff ff dc b 2d20; </pre>	<pre> system ProcessExample { extension Sim for SimpleProcessor { typedef type; expdef Sim.type create (); actiondef exe (Sim.type data, String...); } int turn := 0; Sim.type data; active thread P0() { loc initialize: do { data := Sim.create(); } goto {2d20 }; loc {2d20 }: do { Sim.exe(data, "addis", "r2", "r10", "0"); } goto {2d24 }; loc {2d24 }: do { Sim.exe(data, "addi", "r2", "r2", "776"); } goto {2d28 }; ... } ... </pre>
(a)	(b)

Figure 5.2: Assembly can be directly mapped to a BIR model that uses a language extension to support a specific processor architecture. (a) The assembly instructions for process P0 in Figure 5.1(a). (b) The BIR equivalent to (a) containing the language extension interface, global variable *turn*, and the first two instructions of P0.

Following the initialization location, the location labeled as $\{2d20\}$ contains the first machine code instruction of the mutual exclusion program to execute. The execute method accepts as parameters the current state of the processor stored in the *data* variable and the string representation of the instruction and its arguments. In practice it is easier to decode the machine code bytestream, as the human readable text contains ambiguities in determining the addressing mode of operations. The Language Extension executes the instruction and returns.

Prior to executing the instruction located at $\{2d20\}$, it is possible that an interrupt has ocured and should be serviced. To handle this efficiently, a user-defined

listener module has been attached to the *Location Changed Event*. This module interacts with the instance of the language extension stored in the *data* variable to obtain the *Program Counter* and correctly set the next location for the model checking tool to explore. The control flow of the model is dictated by the current state of the language extension regardless of the structure of the BIR model.

The language extension and location changed listener work together to correctly explore the BIR model as executed on target hardware. By working within the BIR language and tool framework this approach can be used to simulate various hardware models and reuse a common model checking tool. We now detail the application of this strategy to verify a thread scheduling algorithm targeted for the Motorola M68hc11 microprocessor.

Chapter 6

Case Study

The presented strategies using the Bogor Framework to verify software compiled for a target architecture have been used to verify a thread scheduling program targeted for the Motorola M68hc11 microprocessor. The Motorola M68hc11 is a popular microprocessor found in network cards, controllers, and other everyday applications. The M68hc11 supports multiple addressing modes, I/O operations, a rich instruction set, and has a good amount of documentation and software available. The verification of a thread scheduling algorithm demonstrates that the Bogor Framework is able to correctly simulate the necessary behavior of the M68hc11 and model dynamic control flow.

We describe the workings of the Language Extension that embodies the processor behavior, timing model, thread scheduler program, and a multi-threaded application containing an error. The topics of language translation and supporting dynamic control flow in a static model have been covered in previous sections. These items are required to enable the Language Extension to function properly.

6.1 Motorola M68hc11 Language Extension

The Language Extension for the M68hc11 is composed of three logical pieces: the data representation, a controller, and the individual instructions.

The data component implements the necessary interfaces for a non-primitive data type allowing it to be interrogated by the framework when being copied, linearized, or some simple operation performed on data types. As a processor model it contains representations for registers, program counter, free running counter, status registers, main memory, and all other necessary features to support the executing program. The model contains no processor logic and simply provides data storage and access. The Bogor Framework passes the data component to its corresponding Language Extension when performing an operation and the Language Extension returns it and the information necessary to undo the operation.

The controller provides a simple interface to the BIR model for getting and setting memory values during initialization and executing instructions. The controller possesses a collection of objects that correspond to each assembly instruction found in the hardware instruction set. When the execute method is invoked the controller is given the data model, instruction label, and instruction arguments. The instruction label is used to identify the data model and instruction arguments to the appropriate instruction. The controller performs pre and post execution operations such as checking for interrupts, stacking registers if necessary when transferring control, etc.

Each instruction implements the behavior described in the documentation for the microprocessor and provides undo information used by the framework when performing backtracking. For example, the *ldd* instruction loads a two-byte value into the double Accumulator D, which is Accumulator A and B combined. LDD supports five addressing modes: Immediate, Direct, Extended, Index X, and Index Y. Each mode uses the instruction arguments differently to load the data into the Accumulator and modifies the state of the processor differently with respect to timing. The implementation for this instruction handles each of the five addressing modes, their corresponding cycle count, and specific behavior. Each instruction implemen-

tation returns the modified data model and corresponding backtracking information for undoing the execution of the instruction.

6.2 Timing

Timing is necessary to support the Timer Output Compare (TOC) behavior. As noted in the scope section, the representation of time presents a significant obstacle in explicit state model checking. In explicit state model checking a snapshot of the variables composing the system are stored as a state. These states or a representation of them is stored and accessed to determine when a state and its subsequent paths have already been explored. Storing states ensures termination in a finite domain. Any increase in the number of variables or the range of a variable stored in the state exponentially increases the number of configurations or states that might exist. Knowledge of how information stored in the state is used in a program enables the application of abstractions that lessen the state space explosion problem. We specifically target the storage of explicit time and Output Compare registers.

The explicit inclusion of the free running counter in the state requires that a given system configuration (ie., registers, memory) must occur at a specific time. Real-time and embedded systems are typically reactionary by nature, "running" for long periods of time with their behavior dependent upon interaction with the environment rather than a specific value in the free running counter. The TOC feature compares the current clock value against the value in a TOC register and when the current instruction has finished executing and the current clock value has surpassed the compared registers value during that instruction, an interrupt is fired. Instead of explicitly storing time and the TOCx registers, we apply an abstraction that excludes the free running counter from the state and only stores the offset value of an output compare register. The offset value is the TOC value minus the free running counter value. This offset is stored in the state vector. When restored from the state

vector in the event of backtracking, the current free running counter value is added to value of the output compare register. Due to unsigned arithmetic, the TOC value "wraps" around zero and the clock overflow value. Now states that are identical with respect to everything but timing yet only differ on the absolute clock value and not the relative offset can be merged into one state. Collapsing these states reduces the number of duplicate states and paths that must be explored when a state has events scheduled at the same future offset.

The relative offset creates a set of equivalence classes based on the dynamic range of the TOC register values. Figure 6.1 contains pseudo code that illustrates three sample cases for calculating an output compare value: (a) an interval that does not account for drift, (b) an interval that accounts for drift, and (c) a monotonically increasing interval. The benefits of the abstraction are directly impacted by the number of distinct values placed in the output compare registers. At best there is a significant reduction in the number of unique states, at worst it is equivalent to storing explicit time.

The code samples in Figure 6.1 each contain a main method that sets the TOC 1 register to the next clock value that will trigger an interrupt. Figures Figure 6.1(a) and Figure 6.1(b) use the value 20,000 as the length of the desired interval between interrupts. 20,000 clock cycles correspond to 10 milliseconds at the clock frequency for the M68hc11. The *OC1_handler* is defined in the interrupt vector table to serve the Output Compare 1 register. When the Output Compare 1 interrupt is signaled, the *OC1_handler* is loaded to handle the event. The handler calculates the next clock value to signal the next Output Compare 1 interrupt service. The *set_OC1* method sets the value of the Output Compare register. The *TCNT* method returns the current value of the the clock. Figures (a), (b), (c) each calculate the interval in a different manner that affects the abstraction.

```

1:  #define INTERVAL 20000
2:  void main{
3:      set_OC1(TCNT() + INTERVAL);
4:  }
5:  void OC1_handler(){
6:      set_OC1(TCNT() + INTERVAL);
7:  }

```

(a)

```

1:  #define INTERVAL 20000
2:  unsigned short time;
3:  void main{
4:      time = TCNT() + INTERVAL;
5:      set_OC1(time);
6:  }
7:  void OC1_handler(){
8:      time += INTERVAL;
9:      set_OC1(time);
10: }

```

(b)

```

1:  unsigned short INC_INTERVAL;
2:  void main{
3:      time = 1;
4:      set_OC1(TCNT() + INC_INTERVAL);
5:  }
6:  void OC1_handler(){
7:      time += 1;
8:      set_OC1(TCNT() + INC_INTERVAL);
9:  }

```

(c)

Figure 6.1: Run-time calculation of interrupt timing determines the state space reduction provided by the timing abstraction applied to the output compare registers. (a) Demonstrates a simple timing program that schedules the interrupt based on a fixed number of cycles, disregarding the possibility of delay and results in a significant reduction. (b) Accounts for delay or drift by using a sufficiently large interval that may reduce to a single equivalence class. (c) A monotonically increasing interval creates results in worst case behavior negating the abstraction.

Figure 6.1(a) shows an interrupt handler that sets the next interrupt service time to be 20,000 cycles from the current clock value as shown on lines 3 and 6. This does not take into account delay in servicing the interrupt routine due to priorities or the time spent in the routine itself. Either of these can cause a slight shift forward in time that causes the TOC register offset value to vary slightly around the 20,000 cycle interval value. There is a significant reduction in the number of states due to a small number of equivalence classes grouped around the interval value.

Figure 6.1(b) takes into account the possibility for delay by using a variable, *time* defined on line 2, to maintain the next interval value rather than relying on the current clock value. As long as the interval is large enough to account for these delays and the configuration of the system is unchanged between interrupt servicing, the abstraction creates a single equivalence class.

Finally, Figure 6.1(c) uses a monotonically increasing variable, *INC_INTERVAL* defined on line 1, that is added to the current clock value when setting the next interrupt service time on lines 4 and 8. Since the abstraction subtracts the current free running counter value from the TOC register, this negates all savings. The TOC register now has a range that matches that of the free running counter. This particular case requires a different approach to achieve a reduction in the state space.

Storing the TOC register value as an offset most benefits the class of programs that schedule interrupts on regular intervals. We focus on this particular abstraction because it applies directly to the thread scheduling implementation that we verify.

6.3 Thread Scheduler Verification

The thread scheduler being verified is a preemptive round robin scheduler. The source code is found in Figure 6.2. The scheduler is implemented as an interrupt handler that occurs at an interval of 20,000 clock cycles. There is a global variable labeled, *QPtr*, not shown, that references the currently executing thread. *QPtr* stores a reference to the next thread datastructure, stack pointer, stack, and register values. The collection of thread datastructures form a circularly linked list. When the TOC4 interrupt occurs, the currently executing thread's register information is pushed onto the stack. The scheduler stores the current thread's stack pointer value in the *QPtr* structure, lines 3-4. *QPtr* is advanced to the next thread, and the stack pointer is restored, lines 6-7. The scheduler relies upon the interrupt handling mechanism to

```

1:  void __attribute__((interrupt)) scheduler(void) {
2:      unsigned short current_time;
3:      asm (" ldx QPtr \n"
4:           " sts 2,x");
5:      QPtr = QPtr->Next;
6:      asm (" ldx QPtr \n"
7:           " lds 2,x");
8:      _io_ports[M6811_TFLG1] |= M6811_OC4F;
9:      current_time = get_timer_counter();
10:     set_output_compare_4(current_time + 20000);
11: }

```

Figure 6.2: Source code for the Thread Scheduling implementation.

store the state of the currently executing thread, swap the stack pointer, and allow the return from interrupt mechanism to restore the next thread. Lines 8-10 ensure that the TOC4 interrupt is being actively monitored and sets the interval for the next servicing.

The Bogor Framework is able to correctly simulate hardware and exceptional control flow when verifying the thread scheduler. The Estes model checking tool is used as a reference for determining correct state count and behavior in each state [15]. Internally, Estes uses the GBD simulator for state generation and provides a target for correctness as a reference implementation. The presented work builds upon work done with Estes by extending Bogor to verify machine code for instruction set architectures at a lower development cost. We verify the scheduler is correct but find an error in our semaphore implementation for the threads.

Figure 6.3 details the verification results for an instance where an error lies in the threads themselves and not the scheduler. Each state in the verification matches directly with Estes. The difference of three states in the state count is due to extra initialization states and counting the final state. The initial state of a Bogor model is the value of the global variables without visiting the first location. Since non-primitive data types are all initialized to null an additional state must be added to initialize the

	States Count	Explore time
Bogor	24625	20.46s
Estes	24622	10.5s

Figure 6.3: The State Count and Exploration time reported by Bogor and Estes verifying the Thread Scheduling program.

Language Extension prior to exploring the model resulting in two additional states. Bogor adds the error state to the state count, whereas Estes excludes it.

The purpose of this work is to demonstrate an approach that enables software developers to apply model checking as a tool user at the hardware level without requiring a custom tool. Creating a model checking tool requires specific domain knowledge pertaining to model checking as well as the problem domain, development resources, and the time to verify the tool itself. The Estes model checking tool is more than 20,000 LOC and requires a knowledge of the inner workings of GDB to extend it to other instruction set architectures supported by GDB. Implementing the Bogor Framework requires more than 70,000 lines of code, a knowledge of language design, graph traversal algorithms, and reduction strategies. Extending the capabilities of the Bogor Framework to support ISAs by working within the framework results in a substantial savings of development effort. We qualify the savings in section 6.5

Now that we have a functional thread scheduler running on hardware we present an example demonstrating a flawed weak semaphore implementation. This example uses the detail available at the assembly level.

6.4 Multi-threaded Program with Error

The scheduler is the same used in Figure 6.2. The scheduler manages two threads that use a semaphore implementation to arbitrate access to their critical sections. Figure 6.4(a) shows the code for the semaphore datastructure, lines 1-4, and the *wait* function of the semaphore implementation, lines 5-16. The semaphore has a flattened

<pre> 1: struct sema4 { 2: short lock; 3: short S; 4: } volatile sem; 5: void wait(sema4 *sem) { 6: int lock = 0; 7: while(!lock){ 8: disableInterrupts(); 9: if (sem->lock == 0) { 10: sem->lock = 1; 11: lock = 1; 12: } 13: enableInterrupts(); 14: } 15: return; 16: }</pre>	<pre> f83d: ldx #0 load lock f840: tpa f841: sei disableInterrupts(); f842: ldd *c2 load sem->lock f844: bne f84e if (sem->lock == 0) f846: ldd #1 sem->lock = 1; f849: std *c2 f84b: lds #1 lock = 1; f84e: cli enableInterrupts(); f84f: cpx #0 f852: beq f840 while(!lock)</pre>
(a)	(b)

Figure 6.4: Differences in program behavior can emerge because source level operations translate into multiple assembly instructions. (a) Source code for a semaphore implementation’s *wait* function and shared datastructure. (b) Corresponding assembly instructions for the *wait* function.

queue so each thread sits in a busy while loop when waiting. Both threads simply call *wait*, do nothing in their critical section, and then signal. The omission of the **volatile** keyword (line 4) on the *sema4* datastructure introduces incorrect behavior specifically at lines 8 and 9 that is the focus of the following paragraphs.

6.4.1 Figure 6.4(a) Walkthrough

Figure 6.4(a) shows the shared *sema4* datastructure used in the mutual exclusion implementation containing the **volatile** keyword in the struct declaration. The **volatile** keyword instructs the compiler to not reorder the instructions as generated and to always load data directly from memory. Without the **volatile** keyword, the compiler is allowed to reorder instructions according to efficiency and reuse registers that may potentially contain stale values that differ from those actually in memory. In

a multi-threaded or memory mapped input/output environment, the currently executing process or thread is not guaranteed exclusive access to memory. At any time another process, thread, or interrupt can modify a memory value, invalidating the register value.

Lines 5-16 illustrate a simple busy wait. Execution begins with *lock* as 0, line 6. As a local variable *lock* simplifies determining when the thread has been granted access from the semaphore because it can be accessed without worrying about concurrency issues. After initializing *lock* the thread enters a busy loop, line 7, disables interrupts, line 8, and attempts to obtain access, line 9. If successful the *sem*→*lock* and *lock* values are updated, lines 10-11, interrupts are enabled, and the function exits. If unsuccessful, interrupts are enabled and another iteration of the while loop begins.

The *lock* and *unlock* methods enable and disable the servicing of maskable interrupts. When disabled, no other interrupts should be serviced.

6.4.2 Figure 6.4(b) Walkthrough

The assembly code in Figure 6.4(b) is the compiled version of (a). The corresponding C source code is added to the right of the assembly (b). The assembly code strongly resembles the source code with the noticeable difference that some source level statements require multiple assembly instructions. A single source instruction translating into multiple assembly instructions introduces the possibility of new behaviors as interrupts are considered. The following list details instructions used in Figure 6.4(b), Figure 6.5(a), and Figure 6.5(b).

LDX Load Index Register X - Loads an immediate value or a two-byte value from memory into Register X. An immediate value is preceded by the # symbol.

LDY Load Index Register Y - Loads an immediate value or a two-byte value from memory into Register Y. An immediate value is preceded by the # symbol.

TPA Transfer from CCR to Accumulator A - Transfer the Condition Code Register to Accumulator A.

SEI Set Interrupt Mask - Disables all maskable interrupts.

LDD Load Double Accumulator - Loads an immediate value or a two-byte value from memory into Accumulator D. An immediate value is preceded by the # symbol.

BNE Branch if Not Equal to Zero - Causes a branch if the Z bit of the Condition Code Register is not set. Value is set by a previous load instruction.

STD Store Double Accumulator - Stores the contents of Accumulator D to a memory address.

LDS Load Stack Pointer - Sets the Stack Pointer to a constant or memory value.

CLI Clear Interrupt Mask - Enables maskable interrupts.

CPX Compare Index Register X - Compares Register X to an immediate or memory value.

BEQ Branch if Equal - Causes a branch if the Z bit of the Condition Code Register is set.

The most important pieces of Figure 6.4(b) to understand are the placement of the *sei* instruction at address *f841* and *ldd *c2* at address *f842*. The *sei* instruction disables maskable interrupts and *ldd *c2* loads the value of *sem->lock* into Register D. When the **volatile** keyword is removed this sequencing is altered and causes deadlock. The omission of an optional, but important, keyword on a datastructure is an easy mistake that might go unnoticed.

6.4.3 Figure 6.5 Walkthrough

Figure 6.4(b) has been copied and placed in Figure 6.5(a) for a side-by-side comparison. Figure 6.5(b) address *f841 ldx *c2* shows that the compiler has reordered

Volatile sema4			Non-Volatile sema4		
f83d:	ldx #0	<i>load lock</i>	f83d:	ldy #0	<i>load lock</i>
f840:	tpa		f841:	ldx *c2	<i>load sem->lock</i>
f841:	sei	<i>disableInterrupts();</i>	f843:	tpa	
f842:	ldd *c2	<i>load sem->lock</i>	f844:	sei	<i>disableInterrupts();</i>
f844:	bne f84e	<i>if (sem->lock == 0)</i>	f845:	cpx #0	
f846:	ldd #1	<i>sem->lock = 1;</i>	f848:	bne f851	<i>if (sem->lock == 0)</i>
f849:	std *c2		f84a:	ldx #1	<i>sem->lock = 1;</i>
f84b:	lds #1	<i>lock = 1;</i>	f84d:	ldy #1	<i>lock = 1;</i>
f84e:	cli	<i>enableInterrupts();</i>	f851:	cli	<i>enableInterrupts();</i>
f84f:	cpx #0		f852:	cpy #0	
f852:	beq f840	<i>while(!lock)</i>	f856:	beq f843	<i>while(!lock)</i>
			f858:	stx *c2	<i>store sem->lock</i>

(a)

(b)

Figure 6.5: The proper use of the **volatile** keyword on the *sema4* datastructure determines whether or not the *wait* function works correctly. (a) Correct access to the *sema4* datastructure. (b) The missing **volatile** keyword allows the compiler to optimize instruction order and re-use registers, introducing incorrect behavior.

the instructions by moving the load of the *sem->lock* data before the *tpa* and *sei* instructions.

Loading the *sem->lock* data before disabling interrupts potentially results in using a stale memory value, as an interrupt can occur following the instruction at *f841 ldx *c2* but before disabling interrupts at *f844 sei*. When viewed from the source code the program should never execute the *if (sem->lock == 0)* statement before *disableInterrupts()*. This new behavior is introduced with the instruction re-ordering and is completely different from the source code. Address *f858 stx *c2* also shows that the store instruction that should occur within the critical section is placed after *f851 cli* that enables interrupts. Whether or not the ordering of loads and stores is a separate issue from the reuse of a register value, the net result is incorrect behavior manifested by deadlock.

In this example we verified the presence of deadlock given a certain environment and are able to trace the error to the absence of the **volatile** keyword.

Modules	Classes	Functions	Lines of Code	Cyclomatic Complexity			
				Min	Max	Median	Average
Translation	15	113	1228	1	14	1	2.12
Location Changed Listener	2	12	124	1	8	1	2
Processor	6	202	1429	1	17	1	1.66
Instruction Set	142	372	4409	1	38	1	3.38
Total	119	698	7109	1	38	1	2.65

Table 6.1: Software Engineering metrics reporting the number of Classes, Functions, Lines of Code, and Cyclomatic Complexity for each module. The Language Extension is divided between the Processor and Instruction Set modules

Verification at the source level would not have exposed this type of behavior. Figure 6.3 details the state count and explore time for both Estes and Bogor in finding this error. The difference in state count is due to additional initialization states and including the error state in Bogor. The difference in time is due to the difference in implementation languages.

6.5 Software Engineering

The Bogor Framework was selected to provide a platform that allowed tool-users to verify machine code without incurring the cost of developing a custom tool. Table 6.1 details the software engineering metrics for the Language Extension implementation that provides support for the M68hc11 processor using the Code Analyzer Pro tool [4]. The implementation has been grouped into: Translation, Location Changed Listener, Processor, and the Instruction Set. The Translation and Processor components required the most development effort, while the Instruction Set and Location Changed Listener are well defined by their purpose and existing documentation.

In total, there are 142 Classes, 698 Functions, and 7109 Lines of Code (LOC) in our implementation. These numbers would classify the project as being small to medium in industry, likely requiring 2-3 months for a small development team. Each module is fairly cohesive and allows simultaneous development. The interfaces between Processor and Instruction Set are as simple as possible, relying on getters and

setters. The Translation and Location Changed Listener modules operate independently from the other modules.

The Cyclomatic Complexity (CC) or McCabe Complexity is provided as an additional metric for evaluating the implementation difficulty[13]. Cyclomatic Complexity, $v()$, is defined as:

$$v(G) = e - n + p$$

G is the control flow graph of the program

e is the number of edges

n is the number of nodes

p is the number of connected components

The resulting number indicates the maximum number of linearly independent paths through a function. For testing purposes each path requires a test case. An observation from McCabe states that functions scoring more than 10 should be refactored if possible. Depending on the purpose of a function this is not always possible. The motivation for maintaining a low score is an observed inverse relationship between increasing complexity and decreased cohesion that increases development and maintenance cost of software. Table 6.1 shows that each module contains at least one function with a high degree of complexity, but both the median and average scores are manageable.

Combining the metrics of the LOC and CC it is apparent that the entire process from translation to modeling the behavior of the processor is non-trivial. However, creating a Language Extension and supporting modules requires 7000+ lines of code at a reasonable complexity level compared to writing 70,000+ lines of code at a much higher complexity level to implement the Bogor tool.

Chapter 7

Conclusions and Future Work

Verifying software compiled for a specific target architecture requires a significant investment of resources and often results in a custom tool. The Bogor Framework, an "off-the-shelf" general purpose model checking tool, combined with a Language Extension to simulate hardware and module replacement to override default tool behavior provides the benefits of a custom tool without the total development overhead. Our approach allows a single tool to support multiple architectures.

Our current implementation supports key behavior of the M68hc11 processor. Future work focuses on adding Timer Input Compare, Input Port, and Output Port features. It is likely that the Timer Input Compare register value can be modeled and abstracted similarly to the Timer Output Compare. The issue of how to model rising clock edges, falling clock edges, or both still needs to be resolved. Rising and falling clock edges occur in between our current time representation and may require significant work. The Input and Output ports allow communication with peripheral devices. These features are memory mapped making them easy to implement. The more difficult part is simulating the sending or receiving devices. Each of these features enhances the appeal of using the Bogor Framework to verify machine code implementations.

Another area of future work involves generalizing the portion of the language extension that implements the processor behavior. The execution of instructions is trivial to generalize. If the user provides the instruction set and a mapping from

opcodes to instructions then the execution engine becomes a hash table lookup that applies the instruction. Interrupt resolution might contain a reasonable number of classes that a user could select from and enhance as needed. Every tool user benefits when templates and scaffolding are provided.

Bibliography

- [1] T. Ball and S. Rajamani. The SLAM toolkit. In G. Berry, H. Comon, and A. Finkel, editors, *13th Annual Conference on Computer Aided Verification (CAV 2001)*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264, Paris, France, July 2001. Springer-Verlag.
- [2] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
- [3] Riccardo Sisto Claudio Demartini, Radu Iosif. dspin: A dynamic extension of spin. volume 1680, page 261, 1999.
- [4] Code analyzer pro. Available at http://www.geocities.com/sivaram_subr/code_analyzer/description.htm, 2007.
- [5] X. Deng, M. Dwyer, J. Hatcliff, and G. Jung. Model-checking middleware-based event-driven real-time embedded software. In *Proceedings of the 1st International Symposium on Formal Methods for Components and Objects*, November 2002.
- [6] Matthew B. Dwyer, John Hatcliff, Matthew Hoosier, and Robby. Building your own software model checker using the bogor extensible model checking framework. In *CAV*, pages 148–152, 2005.
- [7] The gnu project debugger. Available at <http://sources.redhat.com/gdb/>, 2006.
- [8] Gnu embedded libraries for 68hc11 and 68hc12. Available at <http://gel.sourceforge.net/>, 2005.
- [9] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder, 1998.
- [10] T. A. Henzinger, R. Jhala, R. Majumdar, , and G. Sutre. Software verification with Blast. In T. Ball and S.K. Rajamani, editors, *Proceedings of the 10th International Workshop on Model Checking of Software (SPIN)*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239, Portland, OR, May 2003.
- [11] G.J. Holzmann. The model checker spin. volume 23, pages 279–295, 1997.

- [12] Peter Leven, Tilman Mehler, and Stefan Edelkamp. Directed error detection in c++ with the assembly-level model checker steam.
- [13] T. J. McCabe. A complexity measure. *IEEE Trans. Softw. Eng.*, 2(4):308–320, 1976.
- [14] T. Mehler and P. Leven. Introduction to StEAM - an assembly-level software model checker. Technical report, University of Freiburg, 2003.
- [15] Eric Mercer and Michael Jones. A concurrency model for real-time verification of embedded software at the object level. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS04)*, pages 251–265, 2004.
- [16] Ishai Rabinovitz and Orna Grumberg. Bounded model checking of concurrent programs. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 82–97. Springer, 2005.
- [17] M. Robby and J. Dwyer. Bogor: an extensible and highly-modular software model checking framework, 2003.
- [18] W. Visser, K. Havelund, G. Brat, and S. Park. Java PathFinder: Second generation of a Java model checker. In G. Gopalakrishnan, editor, *Proceedings of the Workshop on Advances in Verification (WAVE'00)*, July 2000.
- [19] Michael Weber. An embeddable virtual machine for state space generation. In Dragan Bosnacki and Stefan Edelkamp, editors, *SPIN*, volume 4595 of *Lecture Notes in Computer Science*, pages 168–186. Springer, 2007.