Towards Provably Safe
Reconfigurable Processor Code:
A Model Checking
and Proof-Carrying Code Approach

by

John Murray Cochran

B.A., University of Texas at Austin, 1990
M.A., Texas Technological University, 1996
M.S., Computer Science, University of New Mexico, 2002

Abstract

Reconfigurable processors pose unique problems for program safety because they employ two dissimilar computational approaches which are difficult to integrate into one computational model. Approaches to software and hardware safety each fail to cover the entire range of potential safety problems for reconfigurable processors, indicating a need for a new approach. One method of dealing with these difficulties is the combination of model-checking for safety verification of reconfigurable array configurations which behave more like hardware, and proof-carrying code for safety
verification of traditional aspects of programs. This approach is investigated for potential safety concerns related to software running on reconfigurable processors. Extensions to Necula’s proof-carrying methodology are proposed. The application of the proposed approach is shown to be limited due to inability of traditional model checkers such as NuSMV to handle large numbers. More sophisticated model checkers which use abstraction, modularity and provide support for arithmetic data will have to be employed.
Contents

List of Figures ix

List of Tables xii

1 Introduction 1

1.1 Problem ......................................................... 1

1.2 Approach ....................................................... 2

1.3 Results .......................................................... 3

1.4 Outline .......................................................... 4

2 Reconfigurable General Purpose Processors 7

2.1 Traditional General Purpose Processors ......................... 7

2.2 Reconfigurable Computing .................................... 9

2.3 Alternative Processor Types .................................. 12

2.4 Safety Problem with Reconfigurable Processors ............... 16
# Contents

3 Garp Reconfigurable Processor 19

3.1 Logic Array ................................................. 19
3.2 MIPS Base .................................................. 22
3.3 Potential Mischief or Bugs ................................. 24
3.4 Sample Garp Program ...................................... 26

4 Temporal Logic 33

4.1 Modal Logics ................................................ 33
4.2 CTL Temporal Logic ...................................... 34
4.3 Kripke Semantics .......................................... 36

5 Model Checking 39

5.1 Relationship Between State Machines and
Kripke Models .................................................. 39
5.2 Model Checking Problem .................................. 40
5.3 Check Function .............................................. 41

6 Binary Decision Diagrams 44

6.1 Representing Boolean Functions ............................ 44
6.2 Ordered Binary Decision Diagrams ...................... 46
6.3 Reduced OBDD Tests and Operations .................... 48

7 Symbolic Model Checking 50
Contents

7.1 Characteristic Functions of State Machines ............................ 50
7.2 OBDD Based Symbolic Model Checking ............................... 52

8 Proof-Carrying Code .......................... 54

8.1 General Idea .................................. 54
8.2 Specification of Safety Policy .................................. 56
8.3 Verification Condition Generator ............................... 63
8.4 Encoding of Proofs ................................ 69
8.5 Production of Annotations and Proofs ......................... 70

9 Combination of Proof-Carrying Code and Model Checking .......... 72

9.1 Limitations of PCC for Reconfigurable Array ...................... 72
9.2 New Safety Issues ................................ 75
9.3 Assumptions and Solutions Provided ............................ 77
9.4 Adding Model Checking ................................ 80
9.5 General Idea .................................... 81

10 Extension of SAL ................................ 83

10.1 New Registers ................................ 84
10.2 Modifications of Old Instructions ............................. 86
10.3 Extension to New Instructions ................................. 86
11 Extension of Verification Condition Generator 96

11.1 Modification for Old Instructions .......................... 96
11.2 Important Aspects of Extending the Verification Condition Generator 99
11.3 Extensions for New Instructions .......................... 103
11.4 Argument for Soundness of Modification .................. 109
11.5 Argument for Soundness of Extension ..................... 112

12 Model Checking the Garp Array 116

12.1 Garp to NuSMV Translator .............................. 116
12.2 Variables and Assignments .............................. 118
12.3 Definitions ............................................. 121
12.4 Problems and Simplifying Assumptions .................. 127
12.5 Properties to be Model Checked ........................ 129

13 Example and Results 137

13.1 Example .............................................. 137
13.2 Performance of NuSMV on Translated Input Files .......... 149

14 Conclusions and Future Work 153

14.1 Conclusions ........................................... 153
14.2 Full Implementation of the System ........................ 154
14.3 Performance ........................................... 156
Contents

14.4 Other Approaches ............................................. 157

A Garp to NuSMV translator 160

B Translated Example Garp Array Configuration 161

C SMV Example Garp Array Configuration 162

D Example Garp Array Configuration Machine Code 163

References 164
# List of Figures

3.1 Diagram of Example Garp Array Configuration .......................... 29

6.1 Binary Decision Tree for \((a \lor b) \land \neg c\) ............................ 46

6.2 OBDDs for \((a \lor b) \land \neg c\) ........................................ 48

8.1 Syntax of Program Logic ................................................. 60

8.2 Deduction System for the Program Logic ................................. 61

8.3 Helper Functions for Verification Condition Generator ................. 68

10.1 Order Definition ........................................................ 85

10.2 Gamma Order Function .................................................. 91

10.3 Delta Result Function ..................................................... 92

11.1 Full Induction Hypothesis ............................................... 110

12.1 Example Crossbar ....................................................... 126

12.2 Example Table Lookup .................................................. 126

12.3 Example Shift Invert ..................................................... 126
List of Figures

12.4 Example Select ............................................. 126
12.5 Example Carry Chain ................................. 127
12.6 Example Carry-Save Addition ...................... 127
12.7 Example Result Function ............................ 127
13.1 Safety Property for Example ....................... 139
13.2 Final Verification Condition ....................... 139
13.3 Specification of Antecedent Counter Values .... 140
13.4 Specification of Antecedent Word Alignment .... 141
13.5 Specification of Antecedent Control Path Values . 141
13.6 Specification of Antecedent Address Value ....... 141
13.7 Specification of Antecedent Address Value cont. 142
13.8 Specification of Antecedent Address Relation .... 142
13.9 Memory Fault Safety Specification ................ 143
13.10 Memory Availability Safety Specification ........ 143
13.11 Memory Synchronization Safety Specification .... 143
13.12 Memory Alignment Safety Specifications ........ 144
13.13 Greater than or Equal to Boundary Specification 145
13.14 Less than Boundary Specification .................. 146
13.15 Memory Write Limitation Specification .......... 147
13.16 Memory Write Synchronization Specification .... 147
List of Figures

13.17 Safe Operation Specification .......................... 147
13.18 Safe Operation Specification .......................... 148
13.19 Safe Operation Specification .......................... 148
## List of Tables

8.1 Operational Semantics for SAL ................. 57
8.2 Verification Condition Generator for SAL ........ 65
10.1 Extended Operational Semantics for SAL .......... 90
10.2 Extended Operational Semantics for SAL cont. ....... 91
11.1 Modified Verification Condition Generator ........... 98
11.2 Modified Verification Condition Generator cont. .... 99
11.3 Extended Verification Condition Generator .......... 104
11.4 Extended Verification Condition Generator cont. .... 105
13.1 Conditions for Example .............................. 138
13.2 Example Symbolic Evaluation ....................... 140
Chapter 1

Introduction

Reconfigurable computing is a rapidly evolving technology that has great potential for significantly improved processing efficiency for a variety of significant tasks. This improvement comes at the expense of increased risk of problems from faulty or malicious programming. Methods of mitigating this risk have not been explored deeply in the literature, although they will be necessary for the ultimate success of the technology. This thesis will explore one potential method of ensuring safety from harmful programs for reconfigurable processors.

1.1 Problem

Reconfigurable processors combine traditional processors as found in the average computer with a reconfigurable fabric as found on field programmable gate arrays. This combination is interesting for a variety of reasons. Of primary interest here, are the problems which arise from attempting to ensure that safety properties hold for programs run on such processors. Safety properties are essentially assertions that some undesirable occurrence can never happen on any potential run of a computation.
Chapter 1. Introduction

Problems with safety can arise in many contexts where untrusted code is run on the processor. The most high profile context currently is mobile code such as an applet. Here, code can be downloaded and run by the processor without any chance for the user to make a decision whether the source is trustworthy. In such a context it is absolutely vital that there are strong guarantees of safety or such a mobile code system could be exploited easily.

Although mobile code is the most obvious example of a context where safety assurance is needed, there are plenty of others. These range from running any program regardless of trust to integrating code into an application when it is not fully understood, or even just as an assurance that a compiler worked correctly.

1.2 Approach

The approach taken to the safety problem for reconfigurable processors is a combination of proof-carrying code and model checking. Proof-carrying code is used to ensure safety of sections of the program which are most similar to traditional programs, while model checking is used to ensure safety of sections of the program that are more like field programmable gate array configurations.

The reason that proof-carrying code was chosen, is that it is a highly flexible system for ensuring safety of machine level code. Because reconfigurable processors use fairly standard machine level code for part of their computations, proof-carrying code provides a good fit for these parts.

Model checking was chosen because it has had success verifying safety properties of digital hardware. The reconfigurable array components of reconfigurable processors behave very much like digital hardware, so this is a good reason to believe that they would be amenable to the model checking approach.
1.3 Results

The results for the approach were found to be somewhat mixed and disappointing. The main shortcoming is that some properties dealing with data can not be efficiently model checked using traditional model checkers such as NuSMV. Since an important goal was to completely automate the process of checking safety properties, typical manual interventions such as abstraction and modularity were deemed inappropriate. This is a significant problem because it prevents the checking of many safety properties dealing with memory accesses which are essential to the success of the approach.

Besides the proposal for combining proof-carrying code with model checking, some of the main contributions of the thesis are:

1. An extension of proof-carrying code for conventional processors to the instruction based component of reconfigurable processors is given. This includes dealing with new instructions for configuring and starting the reconfigurable array, initializing it with data, and reading data from it. A semantics for safe execution is given and the system is proved sound with respect to it.

2. Modification of proof-carrying code for a concurrent system is explored. This includes the introduction of two variants of Hilbert’s e-calculus which are used to model inaccessible and arbitrary values arising due to concurrent executions of the reconfigurable array along with the processor.

3. The application of model checking to an important subset of realistic safety properties for the reconfigurable array component of reconfigurable processors. This includes mostly control properties about synchronizing memory access, but also some data oriented properties on fairly small data types.

It is conjectured that the proposed approach will be more effective if sophisticated
model checkers which provide support arithmetic data, for example, those built using BMDs instead of BDDs or other representations of arithmetic operators, or with automated modularity and abstraction methods, are used instead of traditional model checkers such as NuSMV.

1.4 Outline

Chapter 1 - Introduction This is a brief introduction to the problem, approach, and results.

Chapter 2 - Reconfigurable General Purpose Processors This chapter contains background material on reconfigurable processors. This includes an overview of traditional processors and how reconfigurable processors differ from them, and a discussion of alternative processor types and where reconfigurable processors fit among them. The safety problem is also addressed.

Chapter 3 - Garp Reconfigurable Processor Here, the Garp processor is introduced as an example of a reconfigurable processor. After a discussion of the Garp architecture, concrete safety problems applicable to the Garp processor are addressed. Finally, an example Garp program is given.

Chapter 4 - Temporal Logic Modal logics in general and CTL temporal logic in particular are discussed. Syntax and semantics of CTL are presented. This is the beginning of background material on model checking which will extend over four chapters.

Chapter 5 - Model Checking The relationship between the state machine model of computation and the semantics of temporal logic is discussed, leading to the formulation of the model checking problem. Here, model checking is defined.
Chapter 1. Introduction

Chapter 6 - Binary Decision Diagrams An efficient data structure for model checking — OBDDs — is introduced here. Basic Boolean operations are shown to implementable with OBDDs.

Chapter 7 - Symbolic Model Checking How to do model checking with OBDDs is discussed in this chapter. This includes how to represent state machines as Boolean formulas, and how the algorithms for model checking can be adapted for use on OBDDs.

Chapter 8 - Proof-Carrying Code This chapter contains background material on proof-carrying code. Included is a discussion of a specific implementation which will be the basis for the extension to reconfigurable processors.

Chapter 9 - Combination of Proof-Carrying Code and Model Checking A discussion of why proof-carrying code is limited in the context of reconfigurable processors is followed by a cataloging of safety issues and simplifying assumptions. The general idea of how the combination will work is discussed.

Chapter 10 - Extension of SAL This chapter contains a discussion of how the semantics of proof-carrying code is modified in the new system. This includes the introduction of two variants of Hilbert’s c-calculus to model inaccessible and arbitrary values.

Chapter 11 - Extension of Verification Condition Generator The concrete embodiment of the extended semantics is discussed here. An argument for the soundness of this embodiment relative to the semantics is given.

Chapter 12 - Model Checking the Garp Array This chapter discusses a translator from the Garp configuration format to the NuSMV model checking language including some simplifying assumptions. Properties to model check are also discussed.
Chapter 1. Introduction

Chapter 13 - Example and Results The system is examined through the example given previously. This includes producing the condition to be proved for proof-carrying code and the stating the properties to be model checked. The performance of the model checker on the example specifications is presented.

Chapter 14 - Conclusions and Future Work The approach is accessed here, and avenues for future work are explored.
Chapter 2

Reconfigurable General Purpose Processors

Reconfigurable processors come in many varieties with different advantages. This section will explore the range of possible architectures and compare reconfigurable processors with other processor architectures. General problems of safety for reconfigurable processors will be discussed also.

2.1 Traditional General Purpose Processors

Most general purpose processors in use today follow the same basic architectural principles[ŠRU99]. These architectures, referred to as von Neumann architectures after the early architectural designs of John von Neumann, feature the familiar arithmetic-logic unit (ALU), registers, and memory that often seem to be coextensive with digital computers. The von Neumann architecture is so pervasive that in many contexts the term computer architecture assumes these features and only refers to the specific instruction set which they implement.
Chapter 2. Reconfigurable General Purpose Processors

The characteristics of the von Neumann architecture in which we will be most interested are the ways in which the basic units are used to compute, and in particular that von Neumann architectures have a preferred size of data on which they compute called the word size. The three main computational units are organized to use this word size, the ALU operates on word sized operands, the registers are word sized, and memory is laid out in word sized sections. These architectures store instructions in essentially the same memory system in which they store data and compute by following a fetch, execute, increment cycle in which instructions are fetched from memory, executed by the control logic, and the program counter is incremented to point to the next instruction to be fetched. While there are exceptions to this cycle, on branches the program counter is updated by the control logic instead of simply incremented and pipelining alters it a bit by fetching instructions a few cycles before fully executing them, it is generally followed.

There are problems with this traditional architecture that are becoming more noticeable as designers try to squeeze out more and more performance from processors. The fixed word size can cause a waste of resources when the natural size of the data does not fully use the ALU, register or memory resources. The most obvious example of this is boolean values. When operating on Boolean values the von Neumann architecture must waste resources somewhere unless the word size is 1 bit. For example, on a 32 bit traditional machine a Boolean operation by the ALU occupies the entire ALU for a cycle, especially with the generally followed convention that a word of all 0’s is false and anything else true, the operands must occupy an entire 32 bit register if register addressing is used and usually at least a byte of memory if direct or indirect addressing is used. If the same amount of hardware were designed to operate efficiently on Boolean values it could handle 32 binary Boolean functions at a time instead of only one, using 64 1 bit registers.

Stored instructions can also cause problems with wasted resources. Because ev-
Chapter 2. Reconfigurable General Purpose Processors

every instruction must be fetched from memory immediately prior to its execution, repetitive computations must repeatedly fetch the same series of instructions from memory. While the computation itself is repetitive, it at least should be working on large amounts of data or towards a more complex computation. The repeated fetching of instructions is simply a waste of time. It would be better if some method could be found to keep the instructions in the processor. Fortunately for the von Neumann machines most of them now have an on chip instruction cache which mitigates most of the delay from loading instructions from memory, but this might not be an acceptable solution in very low resource environments or for minimizing power consumption in very low power environments.

2.2 Reconfigurable Computing

One solution to the inefficiencies of the von Neumann architecture is reconfigurable computing[DeH96]. Reconfigurable hardware exists in a large range of complexity from small programmable logic chips which are essentially just multiplexors with control lines hooked into ROM, to large field programmable gate arrays(FPGAs) with thousands of logic cells, flexible interconnection, in-processor memory, and specialized features such as fast carry chains for addition. We will be interested in the hardware at the high complexity end of this spectrum. Large FPGAs are routinely used in many specialized tasks such as video processing, processor emulation, and cryptography, where they have proven to be quite powerful compared to traditional processors.

Like traditional processors, FPGAs usually follow a fairly standard architecture. The computation in an FPGA goes on in many identical logic cells which feature reconfigurable circuitry by which they implement arbitrary operations on their input. Typically this is implemented as look-up tables with a fairly small number of bits
as input, 2 to 8 with 4 most common, and the same number of bits as output. The logic cells usually have a small amount of memory so as to allow the implementation of sequential circuits. Cells are interconnected with a fairly substantial network of configurable wires which come in a wide variety of topologies and degrees of completeness. Completely connected architectures are rare because the number of wires needed for them grows faster than the number of cells and would occupy too much chip space in chips with even moderate numbers of logic cells. In addition to this general purpose interconnection there is often specialized interconnection such as fast carry chains to make certain common operations such as addition easier. FPGAs are configured with a configuration file which specifies the operation at each cell and the interconnection. There are a wide variety of strategies for configuration including one time configuration, slow serial configuration to minimize configuration logic and various width parallel configuration streams to speed up configuration at the expense of more chip space.

Although FPGAs are very efficient at some tasks they are not the best choice for all computational tasks. In particular, irregular computations which need many conditional instructions and use mostly common instruction level computations such as arithmetic or logic on standard word sized data types do better in the traditional architecture. This consideration has lead to the idea of using a traditional processor augmented by reconfigurable arrays. This idea has a wide range of implementation possibilities from using the reconfigurable array essentially as dynamic, flexible microcode to using it as an autonomous coprocessor. In the first instance the array implements an operation for some instruction which is called just like any other instruction. This is similar to microcoded instructions but with more flexibility of operations and dynamic configurations instead of the static initialization of standard microcode. At the other end of the spectrum the array could act essentially as a peripheral with its own pathways to memory, display, network, etc. In between these extremes there is plenty of room in which to explore alternative designs. We
Chapter 2. Reconfigurable General Purpose Processors

will concentrate on a semi-autonomous coprocessor approach with the reconfigurable array accessing memory but not any other system resources.

There are many advantages to the integration of a reconfigurable array with a traditional processor. There is a definite speed advantage for certain types of computation. This can be seen best with sub-instruction level parallelism. In instruction level parallelism, several instruction sequences of a sequential program can be executed at the same time without interfering with each other’s execution. This parallelism is exploitable by several types of processors to speed up execution. A less easily exploitable type of parallelism is sub-instruction parallelism. This parallelism is exemplified in traditional processors by a sequence of several bit masks, comparisons, operations, and recombinations which can logically be done at the same time but for which the available instructions are too coarse. While the comparison and operation instructions may be executed in parallel, they still must be preceded by masking and followed by recombining, which are not essential to the computation. Reconfigurable arrays can exploit this type of parallelism by treating each section of the word as a separate entity and executing the operations in parallel without ever having to explicitly mask off the sections or combine them after the computation. Applications which often have this type of parallelism include cryptography, network protocol processing, and digital signal processing. It is not merely a coincidence that these types of applications are often seen as examples of fast computations on FPGAs and reconfigurable processors.

Another advantage of reconfigurable processors is high computational density. FPGAs are more computationaly dense than traditional processors which means that an FPGA on the same size processor die with the same size features can execute more gate level operations per time period than a traditional processor. This is largely due to reduced overhead such as the lack of complex control overhead for fetching instructions. Of course this advantage can not be exploited for all com-
putations because of a poor fit between the data and the computational resources. A computation might need to keep track of a lot of information during execution that might never be accessed and which is simply pushed through the array, or it might require lengthy computational paths which can not be mapped to the FPGA efficiently.

The final clear advantage of the reconfigurable array is its flexibility. If speed of execution of some function is of paramount importance, then it is always possible to place a specialized coprocessor on the chip to achieve maximal speed. Thus processors are being supplied with floating-point units, multi-media units, etc. and could just as easily be supplied with DES encryptors, JPEG decoders, gene sequencers, cellular automata, and just about any special function that you could imagine. Unfortunately, this would quickly overwhelm the ability of chip makers to provide the needed area and some selection would have to be made. This is where reconfigurable arrays come to the rescue. Although not as fast as a dedicated hardware unit the array can reproduce most if not all of its function and be reconfigured for some other function when needed. This flexibility is the essence of the reconfigurable approach.

2.3 Alternative Processor Types

Several architectural varieties address some of the problems with the traditional processor architecture [ŠRU99]. These solutions can be divided into two broad categories. General purpose architectures are not targeted towards any specific class of application, although any implementation must be to some extent. These architectures include superscalar, VLIW, and vector processors. Application specific architectures are targeted toward specific applications. They include pure ASIC, DSP, and multimedia processors. These categories are not absolute because some architectures have characteristics of both and specific implementations may be hybrid
Superscalar processors are closely related to traditional processors in that they have the same types of instruction sets. This allows one to construct a superscalar processor that runs executables compiled for a traditional processor by using the same instruction set. The difference between the two is that the superscalar processor has multiple instruction issue, meaning that more than one instruction can be executed per clock cycle. There are two types of instruction sequences that can be executed concurrently. Those that can be determined to be independent of each other by the processor are instances of instruction level parallelism. Those that can be shown to depend on differing branch results are instances of speculative execution. Superscalar architectures exploit these two types of concurrency by having multiple functional units and issuing instructions to more than one of them simultaneously. This can lead to significant improvements in the speed of the processor, especially when speculative execution can mask memory access latency. Superscalar architectures have proven to be very successful and have become the standard architecture for processors aimed at use in general purpose computers. The main advantage of this approach is speed and density in executing irregular sequences of instructions. Reconfigurable processors have the advantage in speed of regular computations, computational density of regular computations, and flexibility. Superscalar processors still have to fetch instructions, can not take advantage of sub-instruction parallelism, and have preferred data sizes.

Very Long Instruction Word (VLIW) architectures exploit instruction level parallelism in a manner quite different from superscalar architectures. VLIW architectures have multiple functional units like superscalar architectures but do not rely on the processor to dynamically schedule them. Instead the compiler detects instruction level parallelism and statically schedules instructions for the multiple functional units. As each instruction to a functional unit to be simultaneously executed is ar-
ranged in a single group instruction, this leads to very long instructions and hence the name. While the VLIW architectures exploit instruction level parallelism more efficiently than superscalar architectures, they have basically the same advantages and disadvantages compared to reconfigurable processors.

Vector processor architectures exploit only one specific kind of instruction level parallelism. They have multiple functional units which all execute the same instruction with different data. This allows for further speed up of regular computations over arrays of data as compared to the previous two approaches. While these architectures can be very fast and dense on regular computations on preferred data sizes over arrays, they have no advantage on irregular computations. They cannot exploit sub-instruction parallelism because of the preferred word size of the functional units. Vector processors are less flexible than any of the architectures discussed so far, as they are aimed at regular array processing, so much so that they could be categorized as application specific if there were not so many important regular computations over arrays.

Application specific integrated circuits (ASICs) are pure application specific processors, unable to compute anything but the computation for which they were designed. Having an almost unlimited range of implementation choices and purposes, ASICs don’t conform to any specific architecture but are individually designed based on the application and outside considerations such as time to market and cost. Although they can implement some restricted form of general purpose architecture, it is more common for them to implement all functionality in hardware. This gives them a great advantage in speed and density over any other form of processor for their application when they are designed for maximal performance. Of course they sacrifice all flexibility to achieve this advantage.

Less application specific are digital signal processors (DSPs). DSPs are designed to process digital filters, codes, convolutions and other signal processing tasks which
Chapter 2. Reconfigurable General Purpose Processors

generally have streaming input and output, complex arithmetic computations, low data dependency, and high regularity. Traditional processor architecture is used with some significant differences. One is that DSPs often do not implement general ALUs but have arithmetic units implementing fixed point addition, subtraction, multiplication, and division along with minimal logical functions. Such standard features of the traditional architecture as integer operators, shifts, and bitwise logic are usually left out. Another is that DSPs have special instructions for looping to avoid repeated instruction fetches and high bandwidth memory access to hide memory latency. Finally DSPs have quite complicated and irregular instruction sets with different instruction lengths, wide varieties of addressing modes, and implicit specialized register arguments. All of these features make implementing general algorithms difficult if not impossible on a DSP so they qualify as application specific. The advantage of DSPs is the speed of signal processing applications at the expense of flexibility.

The last type of processor architecture considered is commonly seen as a coprocessor of traditional processors — the multimedia processor. These processors conform closely with the traditional architecture but differ in that they can work in parallel on small sized values and have special hardware for complex arithmetical operations on integer data such as multiplication and matrix operations. Although these processors are optimized for audio-video processing, they can be used in many other computations when the data size and operation fit well. This makes them almost general purpose and in fact, they are quite similar to vector processors with small word size. Once again these processors have greater speed and density for certain computations than traditional processors, although not a great advantage over reconfigurable processors as the instruction fetch overhead is back. They also are not as flexible as reconfigurable processors, although they are more flexible than the other application specific processors.

15
Chapter 2. Reconfigurable General Purpose Processors

Reconfigurable processors are known to have advantages over many competing architectures in speed, density and flexibility. Although there are some applications where reconfigurable processors show significant disadvantages compared to other architectures, such as when speed must be maximized for a specific application or huge arrays can be processed in a regular manner, they seem to fit well with the widest range of applications and desiderata. Therefore, reconfigurable processors should become a significant architectural choice in the near future.

2.4 Safety Problem with Reconfigurable Processors

Although there are many advantages to reconfigurable processors, there is at least one large disadvantage. With the ability to run arbitrary code with arbitrary reconfigurable array configurations, the problem of code safety becomes an issue. Users are used to being able to install and run programs from many sources, and it is unlikely that they would stop or accept limitations to this ability. This leads to safety problems with code from destroying the chip to misusing private data. While there has been much research into safety for traditional processors, there has been very little on reconfigurable processors[IHS99]. Security research by FPGA vendors and academics alike has been focused on safeguarding configurations from reverse engineering and has neglected the issue of safety for the user[Kea01]. For reconfigurable computing to become mainstream, some solution to the safety problem is needed.

The standard solutions for traditional processors do not work well with reconfigurable processors for reasons of efficiency or because they don’t work well in any context. The most visible safety solution is the use of bytecode and a virtual machine run-time environment because of the success of Java[GJS96, LY97]. In this scheme,
the code to be run is not in machine language but a slightly higher level language that can be checked for safety. This language is usually called bytecode and is designed to run on a virtual machine. Because the bytecode is checked at run-time, the safety properties checked must be efficiently decidable. There are problems with this for traditional processors such as performance penalties, decidability restrictions, and a complicated run-time environment. Reconfigurable processors only heighten these problems. The purpose of having reconfigurable elements is to increase the efficiency of computation, but running a virtual machine wastes this increase in efficiency by inserting a layer of interpretation between the code and the processor. This leaves only the virtual machine code with any chance of utilizing the reconfigurable array. If this is the case, it would be better to just design a processor which directly ran the virtual machine because it is static and doesn’t exploit reconfigurability to a significant extent.

A related approach to safety involves using high level languages with strong type safety to write programs and having the user compile them before running them [Nel91, MTH90]. Such safety properties as not adding pointers and not dereferencing integers can be easily expressed as typing rules. This solution has the problem of requiring a perfectly implemented compiler to ensure that safety is preserved even for traditional processors. For reconfigurable processors, this approach is not reasonable because compilation for the reconfigurable array is not nearly as efficient as compiling for traditional processors. Either one would have to accept much less than maximal performance or a long compile time. This assumes that the user would even have a compiler, which is doubtful, considering the high cost and secrecy surrounding hardware compilers. It is much better to have the developer compile or hand write the configuration to ensure performance and ensure safety by some other method later.

A different approach is to insert run-time safety checks into the code before it is
run [LAG93]. This has problems that are very hard to overcome even for traditional processors. It is hard to determine every place in the code that needs a check when the program is already machine code or even if it is in a weakly typed language such as C. It becomes almost impossible to do so when there are reconfigurable array configurations. Even if it could be determined where to place these checks, it might not be possible within the limited resources of the array. The configuration is not like machine code where it is always possible to delay instructions and insert a check. There may not be enough space, or the configuration could rely on very strict timing.

The least safe but most common form of safety policy is trust. This can range from cryptographically certified code from a trusted source, using only shrink wrapped code, downloading code from apparently reputable sources, or just assuming that most code comes from nice people. This entire range of policies succumbs to a single flaw. The code producer need not be malicious to be harmful. While it is unlikely that a virus would be mistakenly created, it is likely that code from trusted sources could contain bugs that produced harmful results. Unfortunately for reconfigurable processors, this could mean destruction of the chip in the extreme case. Of course there is still the problem of misplaced trust. Any trusted party might be malicious after all.
Chapter 3

Garp Reconfigurable Processor

The reconfigurable processor which I will use as an example is the Garp processor designed at The University of California at Berkeley [Hau00, HW97, Hau97]. This processor has not been physically implemented but has been thoroughly specified and documented. The availability of documentation on the machine level programming of Garp is the primary reason for its choice as most reconfigurable processors do not provide documentation with a sufficient level of detail for my purposes. It is also designed to run general purpose programs unlike many reconfigurable processor designs. The basic features of the design which I will discuss are the logic array which implements the reconfigurable part of the processor and the enhanced 32 bit MIPS processor which forms the core of the processor. This will be followed by a discussion of potential malicious or buggy code for the processor and a sample program.

3.1 Logic Array

The Garp logic array is an array of cells arranged into columns of 24 enumerated from right to left from 0 to 23. Cell 23, the leftmost cell, is a control cell in each row
and all the others are logic cells. There are an unspecified number of rows in the array but there can not be more than 32 because the configuration file format can only specify vertical wires for that many. There is a fairly complex interconnection network with vertical, horizontal and global wires providing a dense but incomplete interconnection of the cells. There is a mesh of vertical 2 bit wires for each column which is dependent in number and organization on the number of rows in the array but which includes lengths from four to the length of the column in powers of two. There are 31 horizontal 2 bit wires with a nominal length of eleven cells, some on the end of the row are shorter, which run between rows with a cell in the upper row providing input and all cells in the upper and lower rows potentially using the output. Global wires are similar to horizontal wires except that they run the entire length of the row and there are only four of them. There is also a specialized interconnect wire system for fast carry chains and shifts in each row.

Each logic cell has two 2 bit clocked registers — the Z register and the D register — which can be configured to latch on each cycle or not. The Z register can get input from the cell function output or memory and the D register can get input from the D input to the cell or memory. There are four 2 bit inputs to the logic cells — A, B, C, and D. These can come from any wire pair accessible to the cell, latched Z or D registers, or the constants 00 or 10. There is a separate output source from each logic cell for vertical, horizontal, and global wires which can be independently chosen from either the latched Z registers or the cell function output, or from either the latched D registers or the D input. There are also eight shift inputs and outputs and one carry in and carry out for each cell which can interact with the other inputs depending on the function mode.

The logic cells can implement nearly arbitrary operations with six function modes possible for each cell. Table mode takes the four 2 bit main inputs, produces four new 2 bit inputs by associating each new bit with one from the original two bits
Chapter 3. Garp Reconfigurable Processor

with a crossbar function, and outputs two bits depending on a lookup table. Each bit is decided by the same table although each crossbar function can be different. Split table mode is similar but has two separate tables for the first and second bits and only takes three inputs. The D input is not sent through a crossbar function but directly through to the D register and output. Neither the table nor split table modes use the shift or carry inputs or outputs. Select mode implements a multiplexor which uses the C input as control and selects between the A, B, and D inputs and the output to the horizontal wires from the cell above. The A, B, and C inputs can all be shifted to the left and inverted before use. Partial select mode is identical to select mode with the exception of which inputs are available. The choice here is between A or B possibly shifted and inverted, B without shifting or inverting and 00. Both select modes can have the shift in suppressed in which case all shift ins are 0.

Carry chain mode and triple add mode are more complicated modes which can be used to implement functions requiring fast carry chains or arithmetic on three inputs. The carry chain mode takes the A, B, and C inputs and puts each of them through a crossbar function, and then produces two 2 bit values — propagate value U and generate value V — using separate lookup tables accepting the three results of the crossbar functions. These two values and the carry in bit are then sent to a carry chain function which produces a 2 bit value K and carry out bit following the rules of a carry lookahead adder. The first bit of K is the carry in bit, the second bit is the result of applying the carry lookahead rule to the first bits of U, V, and K, and the carry out is the result of applying the carry lookahead rule to the second bits of U, V, and K. Finally, the result is a 2 bit value produced by taking either V, the carry out, the exclusive or of U and K, or the negation of the exclusive or of U and K. The D input simply goes to the D register and output.

Triple add mode takes the A, B and C inputs and optionally shifts and inverts
them, and sends the results to a carry-save addition function which also accepts a
shift carry in bit and outputs two bits of sum values, two bits of carry values, and a
shift carry out bit according to the usual rules of addition. The sum and carry values
are sent through two table lookups to produce U and V values as for the carry chain
mode. After this the mode operates as the carry chain mode.

Other functions of the logic array are initiated by the control cells. Loading and
storing memory can be initiated by the control cells. This is independent of the main
processor but goes through the same memory hierarchy as the main processor. There
is only one memory address bus so only one access can be initiated at a time. A
control cell can load or store the Z or D registers in its row over one of four memory
data busses. There are also three memory access queues that can be used to access
sequential addresses on demand. Control cells can send an interrupt to the main
processor for program defined reasons. Finally a completion signal can be sent either
through decrementing the array counter to zero or through a forced zeroing of the
counter by a control cell. More detailed information on any of the array features can
be attained from the Garp Architecture Manual[Hau97].

3.2 MIPS Base

The Garp processor relies on a traditional processor core implementing the MIPS-II
instruction set extended by the reconfigurable array specific instructions. The MIPS-
II instruction set is a fairly standard and well known RISC instruction set and will
not be discussed here, but it does have one important feature which the Garp design
uses. The MIPS-II instruction set leaves a significant subset of bit vectors unused as
instructions. This makes extending the instruction set fairly simple. Besides imple-
menting the MIPS-II instruction set, the Garp design uses other standard processor
features such as instruction and data caches, and internal and external busses.
Chapter 3. *Garp Reconfigurable Processor*

The interesting aspect of the traditional part of the Garp processor is the added instructions for interfacing with the reconfigurable array. There are instructions for loading and resetting array configurations, starting and stopping array processing, loading and storing array registers, loading and storing memory queue control, and saving and restoring the internal array state. The instructions for the memory queue and internal array state will not be discussed, as these features will not be considered further. The memory queues have a separate initialization from the rest of the array configuration and do not introduce any indispensable functionality and will therefore be ignored as a simplification. Saving and restoring internal array state is only needed for preemptive multitasking, and not user code, and will therefore be ignored as unnecessary.

There are five instructions which deal with loading array configurations, `gaconf`, `gaalloc`, `gaconfo`, `gareset`, and `gacinv`. `gaconf` takes a single argument which gives the address in memory of the configuration to be loaded. The first word at that address is the size of the configuration in rows and that many rows are loaded with the configuration starting at the next word. All registers are set to 0 and any other configurations are overwritten. `gaalloc` and `gaconfo` do essentially the same job as `gaconf` but in two steps and will be ignored for simplicity. `gareset` resets the array so that it must be reloaded to operate. `gacinv` flushes a configuration from the configuration cache, an operation that does not affect the logical operation of the array or processor and which will be ignored.

The `gabump` instruction increases the array clock counter by some number given as an argument. Once the clock counter is above zero the array starts executing. If the most significant bit of the counter is 0 then the counter decrements by one each cycle until it reaches zero and the array execution is halted by disabling the latching of array registers. Otherwise, if the most significant bit of the counter is 1 the execution of the array is only halted by internal array action or a `gastop`
Chapter 3. Garp Reconfigurable Processor

instruction. As the gastop instruction introduces nondeterminism to the processor execution, it will be ignored.

There are three categories of instructions with which the processor interacts with the array registers, those which interact with the middle 32 bits of the register, mtga, mfga, mtgav, and mfgav, those which interact with the right-most 32 bits of the register, mtgavy, and mfgavy, and those which interact with the left-most 14 bits of the register, mtgavz, and mfgavz. The instructions containing an f move a word from the Garp array while those with a t move a word to the array. Those containing a v take the array register row involved as an argument while the others have it as part of the instruction. They all take an argument specifying the general purpose register involved and whether the Z or D register set is involved. The full specification of instructions is available in the Garp Architecture Manual [Hau97].

3.3 Potential Mischief or Bugs

There are several types of intentional or unintentional safety problems which can result from undesirable use of the Garp processor’s reconfigurable array. The most trivial of problems are configurations with gross syntactic malformation such as impossible lengths and configuration block records that do not correspond to any possible cell configuration. These problems can be handled by local checks when parsing the configuration. The most serious and still easy to detect problems are well formed but illegal configurations being loaded into the array. These are any configurations which result in undefined behavior because of conflicts in individual cell configurations such as multiple cells driving the same wire, cells relying on carry in values which are absent, or cells relying on values which are driven through too many wires or cells before reaching them. This is the most serious problem because the behavior of the chip is undefined when an illegal configuration is run and in
many real FPGA implementations this can lead to anything from seemingly random behavior to destruction of the chip. Fortunately this type of safety problem has been investigated and can be detected by syntactic checking of configurations before they are loaded [IHS99]. Global tables to track properties which can produce illegal configurations can be kept during a configuration analysis, and this analysis can detect these problems. I will assume that programs are checked for this type of problem before they are checked for the more subtle problems with which I will deal.

Another less obvious safety problem with the Garp processor is that only one memory access may be initiated per cycle although more than one control block could be triggered to access memory depending on data in the array. Thus it is possible for a perfectly legal configuration to attempt to perform an illegal action. A similar problem is that two memory accesses initiated at different times could schedule the transfer of data during the same cycle which is also illegal. These actions could lead to unexpected behavior because there is no specification of what should happen in these circumstances. Because they cannot be checked syntactically, these control problems must be checked in some other manner.

A more data oriented than control oriented problem is the accessing and writing of data at addresses which should not be accessible due to safety policy dependent reasons. Such accesses could lead to compromise of private information, array boundary violations directly from the reconfigurable array, or array boundary violations indirectly from the reconfigurable array by alteration of indices in standard code. These are also problems which can occur in perfectly legal configurations and programs but which can be very troublesome. One of the most common low level safety problems is the buffer overflow attack which exploits array boundary violations. A method of preventing them in untrusted code is critical to a well designed safety policy.

A final example of a safety problem is an array configuration which never stops
running. Although there are some situations when this is acceptable, the processor could be responsible for issuing array stopping instructions, this would in general be unacceptable. Certainly there is no method of determining this property syntactically and in many instances not at all. However, there may still be enough reasons to restrict runaway configurations that it would be acceptable to also restrict some safe configurations which defy halting analysis.

3.4 Sample Garp Program

In order to demonstrate the working of the Garp processor a small sample program will be needed. This program will be used for subsequent examples and therefore will be very simple and less than useful in order to keep the discussion manageable. The program manipulates an array of pixels in a simple image format — αRGB — and does nothing else with it. This is not useful as a complete program but could be useful as a subroutine.

The program is given in two parts. The first is the ordinary software section which is expressed in a pseudo-assembly language.

```
.conf
CA 4 289 {...} ; configuration array goes here, 4 bytes by 289
.data
IA 4 200 {...} ; image array goes here, 4 bytes by 200
.code
g1 := 400 ; set g1 to twice size of image array
g2 := IA ; set g2 to address of image array
g3 := g2 - 14
g4 := 768
```
Chapter 3. Garp Reconfigurable Processor

g5 := 0
g6 := CA ; set g6 to address of configuration array
gconf g6 ; load configuration array
mtgavz g4 g5 ; initialize high bits of z reg of row 0
mtga g2 z0 ; move image array address to z reg of row 0
mtga g3 z5 ; move offset image address to z reg of row 5
gabump g1 ; start Garp array execution
gareset ; wait for Garp array to halt
ret ; return

A few comments are in order here. The two array values at the start of the program contain elided values. This is to indicate that actual values would go in these positions to initialize these arrays but writing them out fully would be neither clear nor helpful. The image array would be arbitrary while the configuration array will be given and explained latter.

The operation of the program is fairly simple. It first initializes the g1 register to twice the size of the image array and the g2 register to the address of the image array. The g3 register gets the address of the image array minus 14. This is because it will be used to initialize the part of the array responsible for writing back to memory and it takes seven steps to have the first value ready for writing and each cycle increases the address by two. The g4 and g5 registers are set with constants for initializing the high sections of the row 0 array registers. The g6 register is set to the address of the configuration array. Then the program loads the Garp array configuration, sets the necessary high register section, moves the address of the image array into the Z registers of the first row of the reconfigurable array, and moves the offset image array address into the Z registers of the sixth row. Next, the Garp array is started with the counter initialized to twice the size of the image array. The gareset instruction is used to simply wait for the array execution to halt. Finally, the program returns.
Chapter 3. Garp Reconfigurable Processor

The real computation of the program occurs in the Garp array. The image array consists of 182 pixels which each consist of four one byte sections. These sections correspond to the α channel which indicates the degree of transparency, the red channel, the green channel, and the blue channel. The higher the value for each of these bytes, the greater the intensity of its component in the pixel. The program function is to go through the image array and lighten by one each of the red, green, and blue components by one if the value is not already 1. The α channel is not changed. Because the Garp array takes a few cycles to do this computation, the entire 200 length image array is not processed fully, so the last eight words are assumed to be padding. This means they do not need to be manipulated but that they must be accessible to the Garp array. This leaves 182 pixels which gives a nice little 13 by 14 pixel image.

The function of the Garp array can be seen in 3.1 and is specified in the SMV input language in appendix C. The full specification will not be discussed here as it requires a detailed discussion of the SMV input language but is available for deeper study for those familiar with the language. Unfortunately, there is no standard high level language for specifying Garp configurations documented at this time or the specification would be given in it. The SMV input language was chosen because it will be the language in which the automated translation of the machine code will be given. For complete concreteness, a machine language version of the example configuration is also given in appendix D.

The first row of the configuration contains two types of logic cells, those dealing with the address of data to be loaded and those in the control path. The cells which compute the address of data to be loaded are fairly uniform. Except for the cell in the least significant position, column 4, the cells simply add zero to themselves every cycle. In the figure they are address.\_add.0\_cells and are implemented by the carry
Chapter 3. Garp Reconfigurable Processor

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Ld Addr</td>
<td>1</td>
<td>Ld Reg</td>
<td>2</td>
<td>Comp</td>
<td>3</td>
<td>Add</td>
<td>4</td>
</tr>
</tbody>
</table>

- do-nothing-cell
- address-add-0-cell
- load-register-cell
- copy-cell
- add-3-cell
- compare-to-0-cell
- cout-gen-cell
- self-add-2-cell
- delay-cell
- alternator-cell
- control cell

Figure 3.1: Diagram of Example Garp Array Configuration

chain mode as a simple addition of the constant 0 to the contents of the Z registers with carry in and out. The least significant cell is a self.add.2.cell which simply adds 2 to itself and is implemented by the carry chain mode also. These two types of cells combine to produce the next address for loading every two cycles. This is because the next address is 4 bytes from the current one. The control path cells are also quite simple. The first cell in the control path is an alternator.cell which produces a pulse every other cycle. It starts with an output of 1 onto the h wires below and changes the output from 1 to 0 and back with each cycle. It is implemented by the table mode with the Z registers as input to the A inputs. The output from this cell goes to the A input of a delay.cell which delays the data two cycles before outputting it onto the H wires below. This is accomplished by using the table mode to chose the A
input as the content of the Z register and sending the contents of the Z register to the D register through the D input. Finally, the contents of the D register are output to the H wires below. This output is sent to the A input of a copy cell, which uses table mode to send it to the Z register as for the delay cell and then outputs it onto the H wire below. This delays the data by a single cycle. The last cell in the control path of the first row is the control cell on the far left. It is a load address control cell which gets input to its A inputs from the H wire from the alternate cell and initializes a load operation when either of the inputs is 1. Although left out of the specification, this control cell initializes a 4 byte load scheduled for 4 cycles in the future. The address for the load is taken from the Z registers of cells 4 through 19.

The second row is very simple. Columns 4 through 19 contain load register cells. These cells simply retain the Z register value unless a load transfer is signalled by their control block. They are implemented as table cells with the A input from the Z register and they output the Z register to the H wire below. The control path for this row consists of a single copy cell in the 22 column which copies the output from the H wires above it to the H wires below it, and a load register control cell which gets its input from the copy cell and initiates a memory transfer when either input is 1. This will happen every other cycle starting with the fourth, when the first transfer is ready. The size of the memory transfer is 4 bytes as is the load size from above.

The third row is relatively simple. Columns 2, 7, and 12 are compare to 0 cells. These cells accept input into the A, B, C, and D inputs from the H wires above and checks if all the first input bits are 0 and if all the second input bits are 0. If all the bits are 0 then the cell sets the corresponding Z register bit to 1. If not, it is set to 0. The Z registers are then output to the H wires below. All the other columns from 2 to 20 contain copy cells. These are the same as above but come in several varieties. These varieties allow the 4 bytes from the preceding row to be spread out to allow the compare to 0 cells to fit in the row. The difference boils down to from
which h wire they receive input. The control path is a single copy_cell in column 22 which gets input from the copy_cell above it and puts output to the H wires below it. There is no functioning control cell for the row.

The fourth row is where most of the computation occurs. Before explaining the cells, it will be helpful to reexamine what they are computing and look at how they will do it. The general computation is to reduce the integer interpretation of each of the last three bytes of pixel data by 1 each if they are greater than 0. One method of doing this is to add the two’s complement representation of -1 — a byte containing all 1s — to the bytes if they do not contain all 0s, and to do nothing if they contain all 0s. This can be implemented by always adding -1 and carrying in a 1 if the byte contains all 0s and a 0 otherwise. This method is implemented by using the result of the compare_to_0_cells from the previous row to generate a carry in for each byte and always adding -1 to each byte by adding 3 to each cell. Columns 2, 7, and 12 are cout_gen_cells which take the output of the compare_to_0_cells above them and generate a carry out if they are both 1. This is implemented by a carry chain mode in which the B input gets the constant value 1 and the A input gets the values from the compare_to_0_cells and they are added. If both A inputs are 1 this generates a carry out. All the other columns from 2 to 16 are add_3_cells, which add 3 to the output value from the cell above them. The cells in columns 17 to 20 are copy_cells. Cell 22 is also a copy_cells for the control path. The control cell once again has no function.

The fifth row is where the value to be written back is placed. It has copy_cells in columns 4 to 19, although any cell function with the Z registers latched would work as well, in which the value to be written is copied from the outputs from the cells spread out in the previous row. Nothing in particular must be done with these values except to store them for a cycle while they get written to the pixel array. Cell 22 is also a copy_cells and is part of the control path. Its output goes to the control
cell which is a store_register_control_cell. This control cell transfers its Z register for storage when the input to its A input contains a 1.

The sixth and last row contains the address for writing the value in the previous row. Cells 4 to 19 are the same as in the first row, producing an address which is incremented by 2 each cycle. The control cell is a store_address_control_cell which simply initiates a write to the address in its Z registers when its A input contains a 1. This input is received from the output of cell 22 of the fifth row. Thus the initiate and transfer operations for the store are simultaneous. Because it takes seven cycles before there are any results to transfer, these address cells must be initialized with the address of the beginning of the pixel array minus 14.
Chapter 4

Temporal Logic

Temporal logics are useful for modeling and verification of finite, concurrent systems. This chapter will explore basic issues in temporal logic and the temporal logic CTL will be presented. How temporal logic is used in model checking will be discussed in the next chapter.

4.1 Modal Logics

Modal logics are extensions of propositional or predicate logic which introduce new operators to formalize some concept of possibility and necessity[HR00]. There are wide varieties of both modal logic systems and concepts to which they are applied. On the conceptual side, modal logics can model notions of necessity in logic, knowledge, belief, intention, permission, and most significant for my purposes here, time. On the formal side, there are dozens if not hundreds of proposed systems of modal logic. Which one to use is a question of the concept of possibility and necessity to be modeled, what aspects of the concept need to be expressible, and in some instances what metaphysical position one holds on the concept.
Chapter 4. Temporal Logic

Temporality can be reasoned about in a modal framework if one interprets the modal operators as referring to the possibility and necessity of future states of affairs from a given sequence of states of affairs. This interpretation of modality leads to the consideration of fairly complicated modal logics as there are interesting properties of temporality that do not often arise in other modal interpretations. For example, it is of interest in temporal modality if a proposition must eventually hold. There is rarely anything interesting in other modal interpretations which corresponds to such a question. It will be seen in the discussion of CTL that such considerations lead to a complex logic.

Temporal logic usually models time as discrete, so that temporal evolution consists of states of affairs which follow one another in some order with only one holding at any position in the order. There is usually a non-deterministic evolution of time in temporal models, which means that from any state of affairs there can be more than one state immediately following it in the order. This is referred to as a branching model of time. If there is at most one state following any state then the model is a linear model of time. One needs to distinguish this categorization of interpretations of time from a similar categorization of temporal logics which assumes an underlying branching model of time. Branching-time temporal logics allow statements about states of affairs along multiple branches of time while linear-time temporal logics only allow statements about single branches. This distinction is obviously moot for linear models of time.

4.2 CTL Temporal Logic

Computational tree logic (CTL) is a propositional branching-time temporal logic designed for reasoning about the temporal evolution of computations. The syntax of CTL is fairly simple and is easily explained by distinguishing between state formulas
Chapter 4. Temporal Logic

which are valued at a given state and path formulas which are valued along specific
paths. There is a set $A$ of atomic propositions upon which formulas are inductively
defined:

- if $f \in A$ then $f$ is a state formula
- if $f_1$ and $f_2$ are state formulas then $\neg f_1$, $f_1 \lor f_2$, and $f_1 \land f_2$ are state formulas.
- if $g$ is a path formula then $Ag$, and $Eg$ are state formulas
- if $f_1$ and $f_2$ are state formulas then $Xf_1$, $Ff_1$, $Gf_1$, and $f_1 U f_2$ are path formulas

State formulas are propositions of CTL, while path formulas are incomplete expres-
sions and are not propositions of CTL.

Informally, the propositional connectives $\neg$, $\lor$, and $\land$ are the same as in classical
propositional logic: negation, disjunction and conjunction. The path quantifiers
$A$ and $E$ quantify over path formulas universally and existentially and correspond
temporal necessity and possibility respectively. The temporal operators $X$, $F$, $G$,
and $U$ construct formulas which specify parts of temporal paths on which a formula
holds. $X$ specifies the next state, $F$ specifies some state, $G$ specifies all states, and
$U$ specifies all states until some proposition holds.

Before giving formal semantics for CTL it is useful to look at a few example
propositions and what they mean informally.

- $AF(p)$ means that along all paths from the current state $p$ holds in some state.
- $EG(p)$ means that along some path from the current state $p$ holds in all states.
- $A[pUq]$ means that along all paths from the current state $q$ holds in some state
  and $p$ holds in all prior states.
Chapter 4. Temporal Logic

There are also several important equivalences which both give some insight into the meaning of CTL propositions and allow us to pare down the number of temporal operators and path quantifiers, leaving only X, G, U, and E.

- \(AX(p) \equiv \neg EX(\neg p)\)
- \(EF(p) \equiv E[true \cup p]\)
- \(AG(p) \equiv \neg E[true \cup \neg p]\)
- \(AF(p) \equiv \neg EG(\neg p)\)
- \(A[p \cup q] \equiv \neg E[\neg q \cup (\neg p \land q)] \land \neg EG(\neg q)\)

These informal descriptions and equivalences can give only a fuzzy understanding of CTL. To gain the deep understanding needed to use CTL in formal reasoning there must be a more formal description of how propositions in CTL are related or what they mean. This requires either a system of deduction or a formal semantics. For model checking the semantics is all that is required, so it will be presented next and no deduction system for CTL will be presented.

4.3 Kripke Semantics

The standard semantics for modal logics is Kripke semantics, which is also known as possible world semantics. The basic idea is that there is a structure, the Kripke structure, which contains many states, possible worlds, in which all propositions of the logic have valuations. The propositions without modal operators are unrestricted in their valuations while the modal propositions are restricted by rules for the semantics of modal operators and an accessibility relation between worlds. The rules are that if in all worlds accessible from world \(W\), \(p\) holds then necessarily \(p\) holds
Chapter 4. Temporal Logic

in \( W \), and if in some world accessible from world \( W \), \( p \) holds then possibly \( p \) holds in \( W \). These simple rules and some restriction on the accessibility relation are often enough to obtain an interesting modal logic. Unfortunately the semantics for CTL is complicated by the paucity of acceptable restrictions on the temporally interpreted accessibility relation and by the existence of temporal operators over path formula. While the basic idea will still work it needs to be augmented with a notion of path to explain CTL.

Let \( A \) be the set of atomic propositions. \( M = \{ S, S_0, R, L \} \) is a Kripke structure over \( A \) where

- \( S \) is a finite set of states
- \( S_0 \subseteq S \) is the set of initial states
- \( R \subseteq S \times S \) is a total relation between states
- \( L : S \rightarrow 2^A \) is a function that labels all true propositions in a state for each state

Informally, \( S \) is a set of states or worlds and \( R \) is the accessibility relation between them. \( R \) is restricted so that it is total, for every \( s \in S \) there exists an \( s' \in S \) such that \( R(s, s') \), which means that time does not stop in any state. \( S_0 \) is the set of initial states where time can start.

Let \( \pi \) be a path in \( M \) if it is an infinite sequence \( s_0, s_1, s_2, ... \) where for any \( s_n \) and \( s_{n+1} \), \( R(s_n, s_{n+1}) \). Paths are simply possible evolutions of state through time. Note that it is not necessary for a path to begin in an initial state.

The semantics can now be given by induction over formulas. State and path formulas must both be considered. \( M, s \models f \) means that state formula \( f \) holds in structure \( M \) at state \( s \), and \( M, \pi \models f \) means that path formula \( f \) holds in structure
Chapter 4. Temporal Logic

\( M \) on path \( \pi \). \( \pi(s_x) \) refers to the \( x \)th state in path \( \pi \) starting with \( \pi(s_0) \), \( p \) is an atomic proposition, \( f \) and variants are state formulas, and \( g \) and variants are path formulas. The semantic entailment relation \( \models \) is defined as:

- \( M, s \models p \iff p \in L(s) \)
- \( M, s \models \neg f \iff M, s \not\models f \)
- \( M, s \models f_1 \lor f_2 \iff M, s \models f_1 \) or \( M, s \models f_2 \)
- \( M, s \models f_1 \land f_2 \iff M, s \models f_1 \) and \( M, s \models f_2 \)
- \( M, s \models \mathbf{E}g \iff \) for some path \( \pi \) from \( s \): \( M, \pi \models g \)
- \( M, s \models \mathbf{A}g \iff \) for all paths \( \pi \) from \( s \): \( M, \pi \models g \)
- \( M, \pi \models \mathbf{X}f \iff M, \pi(s_0) \models f \)
- \( M, \pi \models \mathbf{F}f \iff \) there exists a \( k \geq 0 \) such that \( M, \pi(s_k) \models f \)
- \( M, \pi \models \mathbf{G}f \iff \) for all \( i \geq 0 \), \( M, \pi(s_i) \models f \)
- \( M, \pi \models f_1 \mathbf{U}f_2 \iff \) there exists a \( k \geq 0 \) such that \( M, \pi(s_k) \models f_2 \) and for all \( 0 \leq j < k \), \( M, \pi(s_j) \models f_1 \)
Chapter 5

Model Checking

Model checking is a formal verification method that automatically examines finite models of concurrent systems to determine properties of the models. The verification properties can be expressed in any one of a variety of logics but temporal logics are the most popular and natural. I will assume that CTL is the logic used for model checking. After discussing the relationship between state machines and the semantics of temporal logic, the model checking problem and an algorithm for solving it will be presented.

5.1 Relationship Between State Machines and Kripke Models

Model checking relies on the relationship between state machines and Kripke models. Usually, the system to be checked, such as a digital circuit or communication protocol, is easily modeled by a finite state machine. However, CTL does not refer to states in a machine but to possible worlds in a Kripke model. While it may seem obvious
that the two domains are essentially the same, it is important to explore the issue further.

Giving a correspondence between states of a machine and possible worlds seems straightforward, valuations of state variables in a machine state correspond to valuations of atomic propositions at corresponding possible worlds, and transitions between states correspond to the accessibility relation. While this correspondence is generally applicable, there is a consideration which restricts it somewhat.

The totality property of paths in the Kripke semantics puts a restriction on state machines if the correspondence is to work. Because every world has a successor world, the machine corresponding to a structure must not have any states where the machine halts. This is not a severe restriction on machines as every machine which halts can be transformed into a machine that does not by the addition of a \textit{halted} state variable which is false in all states with a transition out, and true in all states with no transitions out. Then a transition can be added to all the states with \textit{halted} true back to the same state. The correspondence is now possible with the restriction that the state machine does not halt. Because this restriction is easily side-stepped, it is possible to interpret CTL propositions as referring to the actions of state machines. This makes model checking CTL properties of machines possible.

\section{Model Checking Problem}

Model checking can be easily summarized as a simple problem dealing with machines and temporal propositions\cite{CGP99}. Assume that there is some finite state machine \( M = \{S, S', T, L\} \), where \( S \) is the set of states, \( S' \subseteq S \) is the set of initial states, \( T \subseteq S \times S \) is a total transition relation, and \( L : S \rightarrow 2^A \) is an injective function from states to sets of state variables \( A \) labelling those which are true in the state. For any CTL formula \( f \), determine for which states \( S_t \subseteq S \), \( f \) is true when \( M \) is considered
Chapter 5. Model Checking

as a Kripke model.

This problem has several variations of interest which are easily solved once the model checking problem is solved. Checking if a proposition is true in all states or no states in a model is straightforward — check if $S_t = S$ or if $S_t = \emptyset$. Likewise, any subset of $S$ can be checked for the truth of the proposition for all its members. Checking if two propositions are ever true at the same state is the same as checking if their conjunction is ever true, and likewise for checking if either of two propositions is ever true is the same as checking if their disjunction is ever true. These types of variations are easily constructed and allow a great deal of interesting problems to be model checked.

5.3 Check Function

Now that there is a concrete problem one can give an algorithm to solve it. This algorithm consists of three mutually recursive functions which operate on propositions and models, and return sets of states. I will assume that the model $M$ is fixed for clarity, that $f$ and variations are arbitrary formulas in CTL, $p$ is an atomic proposition, and $s$ and variations are states in $S$. I will also assume that CTL is simplified so that all formulas are expressed as combinations of atomic propositions, negation and conjunction, and $\textbf{EX}$, $\textbf{EG}$, and $\textbf{EU}$. This restriction is allowable as every formula in CTL can be expressed in the new simplified form as is known from the elementary theory of propositional logic and the chapter on CTL. The algorithm starts with the function $\text{Check}$.

$\text{Check}(f)$

\[
\begin{align*}
&\text{if } f = \text{false } \text{return } \emptyset \\
&\text{if } f = p \text{ return } \{s \mid p \in L(s)\} \\
&\text{if } f = \neg f_1 \text{ return } \{s \mid s \not\in \text{Check}(f_1)\}
\end{align*}
\]

41
Chapter 5. Model Checking

if \( f = f_1 \land f_2 \) return \( \{ s \mid s \in \text{Check}(f_1) \text{ and } s \in \text{Check}(f_2) \} \)
if \( f = \textbf{EX}(f_1) \) return \( \{ s \mid \text{for some } s' \in \text{Check}(f_1), T(s, s') \} \)
if \( f = \textbf{EG}(f_1) \) return \( \text{CheckEG}(f_1) \)
if \( f = \textbf{E}[f_1 \cup f_2] \) return \( \text{CheckEU}(f_1, f_2) \)

The \( \text{CheckEG} \) and \( \text{CheckEU} \) functions are called from \( \text{Check} \) and also call it.

\( \text{CheckEG}(f) \)

\[
S_1 := S_2 := \text{Check}(f) \\
\text{do} \\
\text{foreach } s \in S_1 \\
\quad S_1 := S_1 - s \text{ unless } s \in \text{Check} (\textbf{EX}(f)) \\
\text{if } S_1 = S_2 \text{ return } S_1 \\
S_2 := S_1
\]

\( \text{CheckEU}(f_1, f_2) \)

\[
S_1 := S_2 := \text{Check}(f_2) \\
\text{do} \\
\text{foreach } s \in \text{Check}(f_1) \\
\quad S_1 := S_1 + s \text{ if there is an } s' \in S_1 \text{ such that } T(s, s') \\
\text{if } S_1 = S_2 \text{ return } S_1 \\
S_2 := S_1
\]

The operation of the \( \text{Check} \) function is simply to determine the result for atomic propositions and \textit{false}, and to recurse for other propositions. \( \text{CheckEG} \) uses greatest fixed-point iteration to remove states from the set in which the sub-proposition is true. If for some state there is no next state in the set of states possibly satisfying the proposition, it is removed, producing a new set. This process is repeated for the new set until a fixed-point is reached. Similarly, \( \text{CheckEU} \) uses least fixed-point iteration to add states to the set in which the second sub-proposition is true. If for
some state for which the first sub-proposition is true, there is a next state in the set of states possibly satisfying the proposition, it is added, producing a new set. This process is repeated for the new set until a fixed-point is reached.
Chapter 6

Binary Decision Diagrams

In order to make model checking feasible for large models there needs to be some method of representing Boolean functions in an efficient manner. Ordered Boolean Decision Diagrams (OBDDs) are one such representation which is used extensively in model checking. After introducing BDDs and some restrictions of them in this chapter, I will explain how they can be used for symbolic model checking in the following chapter.

6.1 Representing Boolean Functions

Boolean functions are used in many computing applications and have been extensively investigated for efficient data structures for many purposes[MT98]. The least space efficient data structure for Boolean functions is the truth table. This type of table is familiar from the semantics of propositional logic and can easily be made unique for each function by ordering the variables. The main problem is that these tables are always exponentially large in the number of variables in the function. Otherwise, they are easy to manipulate. The negation of a function is the same
Chapter 6. Binary Decision Diagrams

table with each result negated, conjunction and disjunction of two tables is simply the row-wise conjunction or disjunction of the original tables’ results. Validity, consistency, and inconsistency of the proposition characterized by the function can be easily determined by checking if the results are all true, some true, or all false respectively.

Another method of representing Boolean functions is as sentences in propositional logic. A sentence in propositional logic implicitly defines a function from its variables to truth values. Unfortunately, representing Boolean functions as propositional sentences is inefficient for most applications. Some of the reasons are that the same function can have multiple representations, parse trees for arbitrary sentences can have large depths, and the size of sentences can be very large for fairly trivial functions. Manipulation of sentences is fairly trivial but checking properties such as validity is quite difficult. Normal forms for sentences such as disjunctive normal form and conjunctive normal form help to eliminate the problem of parse tree depth so that a more space efficient structure is possible but at the cost of making most types of manipulation more difficult.

Binary decision trees are closely related to truth tables but have a binary tree structure. If you take a binary tree with depth equal to the number of parameters of a Boolean function plus one, and label each level of non-leaf nodes by one of the variables in the order they appear in the table, it is easy to map a truth table onto it. For each row in the truth table follow the tree from its root to a leaf node taking a right hand turn after a node if its corresponding variable in the row is false and a left hand turn if it is true. When you reach the leaf node label it with the result in that row of the truth table. Figure 6.1 shows the binary decision tree for \((a \lor b) \land \neg c\). This structure now encodes the same function as the truth table, and has most of the same properties of the truth table. In particular it is still of exponential size in the number of variables \(n\) — the leaf nodes by themselves number \(2^n\).
Looking closely at the binary decision tree of figure 6.1 shows that it is not optimal in its use of space. The leaf nodes have only two possible values and can thus be replaced by two nodes instead of $2^n$ nodes. Some nodes, such as the $c$ node on the right go to equivalent nodes on both their right and left paths. These nodes could be eliminated because they give no further information. Paths could just bypass them. The three leftmost $c$ nodes are redundant, they go to equivalent nodes on each side, and can be consolidated. These considerations lead to an alteration of the data structure which results in ordered binary decision diagrams.

6.2 Ordered Binary Decision Diagrams

Ordered binary decision diagrams are data structures used to represent Boolean functions which have many advantages[Bry92]. OBDDs are directed acyclic graphs at the roughest level. The non-leaf nodes in these graphs represent variables of the Boolean function, and each one has two child nodes — for true and false valuations of the variable. There are only two leaf nodes which represent true and false.
Chapter 6. Binary Decision Diagrams

desirable properties one could wish. Reduced OBDDs are a restriction of OBDDs which give unique canonical forms for Boolean functions given a fixed variable ordering. The reduction of an OBDD proceeds by three rules which should be applied from the bottom level to the root of the graph. If a leaf node has no parents remove it from the graph. If a node \( N \) has right and left children which are equivalent it is eliminated, and its parents are given \( N \)'s child as a new child. This rule eliminates nodes which are immaterial to the outcome along any path. If two nodes representing the same variable have equivalent right and left children then they can be merged into a single node which is shared by all the original parent nodes. Figure 6.2 shows two reduced OBDDs for the same Boolean function figure 6.1, with dashed lines to indicate true children and solid lines to indicate false children. They differ in the ordering of the variables as can be seen by the labels of their nodes.

Figure 6.1 also illustrates that the variable ordering can influence the size of the OBDD. The first ordering — \((a,b,c)\) — has three non-leaf nodes, while the second — \((a,c,b)\) — has four. This is common for most functions and can mean many orders of magnitude of difference for some functions. Other functions are less sensitive to variable order such as tautologies, conjunctions of atoms, and multiplication of bits interpreted as binary integers. Unfortunately when working with many OBDDs it is usually impossible to find a variable ordering which simultaneously minimizes all of them. This can lead to situations in which it is best to have several variable orderings but I will ignore this and assume that all OBDDs have the same variable ordering.
6.3 Reduced OBDD Tests and Operations

Now that reduced OBDDs have been defined, an investigation of their usefulness in common tests and operations of Boolean functions is required. Tests of validity, consistency, and equivalence of reduced OBDDs are all fast and simple. A Boolean function is valid, always returns true, if and only if it consists of nothing but the true leaf node. Similarly, it is consistent if and only if it consists of more than just the false leaf node. These two tests are obviously optimal for any representation of Boolean functions. Equivalence checking is not quite so fast but still simple — check if the two reduced OBDDs are equivalent. This works because reduced OBDDs with the same variable ordering are unique canonical forms.

Manipulations of OBDDs are generally more complex than the tests examined above. One that is as simple is negation. To negate an OBDD it is sufficient to flip the values in the leaf nodes. True becomes false and false, true. Binary operations can be done on two OBDDs by means of the Apply algorithm. This algorithm takes a binary logical operator and two OBDDs as arguments and returns an OBDD which is the result of applying the operator on the two OBDDs. The Apply algorithm
works recursively on the OBDDs from the roots to the leaves. Let $B_1$ and $B_2$ be two OBDDs, $r_1$ and $r_2$ the roots of $B_1$ and $B_2$ respectively, $l(r) = true$ if $r$ is a leaf node, $f(B)$ and $t(B)$ the false and true children of $B$ respectively, and $r_1 < r_2$ if $r_1$ represents a variable following the variable represented by $r_2$ in the variable ordering. The algorithm is:

$Apply(op, B_1, B_2)$

  case 1: $l(r_1)$ and $l(r_2)$
  if $r_1 \ op \ r_2 = 0$ return $B_1$
  if $r_1 \ op \ r_2 = 1$ return $B_2$

  case 2: $r_1$ and $r_2$ represent the same variable
  return an OBDD $B$
    with $f(B) = Apply(op, f(B_1), f(B_2))$
    and $t(B) = Apply(op, t(B_1), t(B_2))$

  case 3: $l(r_1)$ or $r_1 < r_2$
  return an OBDD $B$
    with $f(B) = Apply(op, B_1, f(B_2))$
    and $t(B) = Apply(op, B_1, t(B_2))$

  case 4: $l(r_2)$ or $r_2 < r_1$
  return an OBDD $B$
    with $f(B) = Apply(op, f(B_1), B_2)$
    and $t(B) = Apply(op, t(B_1), B_2)$

The $Apply$ algorithm can result in a non-reduced OBDD, so the reduction rules must be applied to it before it is a reduced OBDD. This statement of the algorithm is exponential in the sizes of the OBDDs but it can be significantly improved through memoization. This reduces the complexity to linear in the product of the sizes of the OBDDs.
Chapter 7

Symbolic Model Checking

While model checking with explicit state graphs is theoretically feasible, in practice the state explosion problem limits it to small problems. One method of working around this limitation is symbolic model checking based on OBDDs. This method will be presented after a discussion of how to represent state machines as Boolean functions.

7.1 Characteristic Functions of State Machines

In order to solve the model checking problem using OBDDs, it is necessary to reconcile the set based representation of machines with the Boolean functions which OBDDs represent. Characteristic functions of sets of states are one method of doing this. To construct a characteristic function of a set of states, it is first necessary to encode the states as Boolean vectors \(< x_1, x_2, x_3, ... >\). One particularly obvious method of doing this is to assign each state variable to an \( x_i \). This gives a unique representation for each state as long as each state has a unique valuation. If not, additional \( x_i \)s can be added to the vector until uniqueness can be assured. Assignment
to these new $x_i$s can be arbitrary for states that do not need to be distinguished. Although this method of assigning vectors to states is simple, any assignment of vectors with unique representations of states will work.

The characteristic function of a set of states given an assignment of length $n$ vectors to states is simply a Boolean function $C : \{0, 1\}^n \rightarrow \{0, 1\}$. The function $C$ represent a set $S$ of states if it evaluates to true for vectors representing members of the set and to false for vectors representing non-members of the set. For example a vector $<0, 1, 1, 0>$ can be characterized by the function $\neg x_1 \land x_2 \land x_3 \land x_4$, and the set of two vectors $<0, 1, 1, 0>$ and $<0, 1, 1, 1>$ can be characterized by the function $(\neg x_1 \land x_2 \land x_3 \land x_4) \lor (\neg x_1 \land x_2 \land x_3 \land x_4)$. This last function can be simplified to $\neg x_1 \land x_2 \land x_3$ which the reduced OBDD representation will do for us. This method leaves indeterminate any unused vectors. They can evaluate to true or false as desired for simplification of the function. These characteristic functions can be easily represented by OBDDs and therefore sets of states can be represented also.

While this representation allows sets of states to be specified it ignores the transition relation between states and the valuation of state variables. Individual transitions can be represented as Boolean vectors twice the length of the Boolean vectors representing states. The first half of the values represent the state before the transition and the second half of the values the state after the transition. The variables in the second half will be named the same as the first half but primed. The entire transition relation can then be represented by a characteristic function of all the individual transitions. This is simply the disjunction of all the characteristic functions of individual transitions. Valuations of state variables can be represented by OBDDs for each state variable which specify for which states the variable evaluates to true. If we restrict the possible vector assignments to those where the $x_i$s correspond to state variables, then the OBDD representation of state variables is unnecessary.
7.2 OBDD Based Symbolic Model Checking

Symbolic model checking works similarly to explicit state model checking but uses OBDDs to represent the state machine [BCM+90]. The main difference is that symbolic model checking has to work on sets of states at all times and cannot use algorithms which traverse a graph state by state. Looking at the Check algorithm, this means that the cases for temporal operators must be modified.

Symbolic model checking uses a new SCheck function which assumes an OBDD representation of the machine states, transition relation, and state variable valuation. OBDD(x) is the OBDD representing the state variable valuation of x, and neg applied to an OBDD negates its terminals.

\[
S\text{Check}(f)
\]

\[
\text{if } f = \text{false} \quad \text{return } \emptyset \\
\text{if } f = p \quad \text{return } \text{OBDD}(p) \\
\text{if } f = \neg f_i \quad \text{return } \neg (S\text{Check}(f_i)) \\
\text{if } f = f_i \land f_2 \quad \text{return } \text{Apply}(\land, \text{Check}(f_i), \text{Check}(f_2)) \\
\text{if } f = \text{EX}(f_i) \quad \text{return } S\text{CheckEX}(S\text{Check}(f_i)) \\
\text{if } f = \text{EG}(f_i) \quad \text{return } S\text{CheckEG}(S\text{Check}(f_i)) \\
\text{if } f = \text{E}[f_i \cup f_2] \quad \text{return } S\text{CheckEU}(S\text{Check}(f_i), S\text{Check}(f_2))
\]

The SCheckEX, SCheckEG, and SCheckEU functions are the main difference between explicit model checking and symbolic model checking. SCheckEX(d) can be implemented by renaming the nodes in d by their primed versions to get d', using the Apply function to get the conjunction of the d' OBDD and the transition OBDD, and computing a new OBDD that lacks the nodes for the primed variables and represents the set of states with transitions to states in d. This last computation can be carried out by a series of restrictions of the OBDD and conjunctions of the resulting OBDDs. An OBDD can be restricted to having a variable x be true by removing it from the
graph and replacing any arrows to it with arrows to the true child of $x$. A restriction to false is similar. If the restriction to true and the restriction to false of a single variable are conjoined, the resulting OBDD represents a new function in which the variable can be either true or false. If these operations are carried out on all the primed variables in an OBDD then the computation results in the new OBDD with no restriction on the valuation of the primed variables. This OBDD is the result of the $SCheckEX$ function.

The $SCheckE_G$ and $SCheckEU$ functions can be computed as fixed points of functions involving $\textbf{EX}$. $\textbf{EG}(f)$ is the greatest fixed point of $F = f \land \textbf{EX}(F)$, and $\textbf{E}[f \cup g]$ is the least fixed point of $F = g \lor (f \land \textbf{EX}(F))$. The calculation of fixed points of OBDDs is a simple iterative process. For greatest fixed point calculations start with an approximate solution of the OBDD with only the true node — for least fixed points start with false — and apply the function to the approximation and OBDDs for any other terms repeatedly to get a new approximation. When two approximations are the same stop the iteration and return the last approximation OBDD. The functions for the fixed point characterization of $\textbf{EG}$ and $\textbf{EU}$ only involve applications of the $SCheckEX$ and $Apply$ functions.

While all the machinery of symbolic model checking seems to make model checking more complicated, it is of the same asymptotic complexity as explicit state model checking. The big advantage with symbolic model checking is that the state explosion problem can be mitigated for a large number of problems. This is because a large number of states can sometimes be represented by a fairly compact OBDD. In the worst case there is no gain in using symbolic model checking but in practice a great gain in the size of checkable models is achieved. Models with about $10^{10}$ states have been successfully checked with symbolic methods while explicit state methods can only handle around $10^{10}$ states [McM92].

53
Chapter 8

Proof-Carrying Code

The problem of how to assure that code for traditional processors from untrusted sources is safe to run is addressed by proof-carrying code systems. The particular system which will be discussed is Necula’s, because it is the first and most extensively documented example. The topics discussed are the specification of safety properties, the generation of a verification condition, the encoding of proofs, and the generation of proofs.

8.1 General Idea

Proof-carrying code (PCC) is a method of assuring safety of machine code which is fairly easy to explain on a general level [Nec98, Nec97, NL98, NL97, NL96]. The first component of PCC is a safety policy for the system in which the code is to run. The safety policy is specified by the developer of the system which is running the untrusted code and can vary widely over a large space of possible policies. There is no notion of a universally acceptable safety policy for all systems in PCC, although there is a common infrastructure which is part of the final safety policy specification.
Chapter 8. Proof-Carrying Code

Each system must create a safety policy or set of such policies using the common infrastructure and advertise it to developers of programs for the system. Then they can write programs which follow the policy. As an example, a web browser which used PCC to ensure the safety of applets might include a configuration option which allowed the user to choose from several safety policies. Developers of applets for the browser would have to choose which policies to follow from a guide published by the developer of the browser.

The program developer must provide a proof of compliance with the advertised safety policy for any code. This second component of PCC, the proof of compliance, must be in a specific format to allow automated checking. This also means that the proof must refer directly to the machine code and not an abstracted version of the program. How the proof is produced does not matter for the PCC system. As long as it is a valid proof then it is acceptable. Extending the example above, the developer of the applet must make a proof available with the applet which proves that the applet satisfies the chosen safety policy.

The third component of PCC is a proof checker for the client. When the client receives code from an untrusted source it is examined along with the attached proof to assure that it follows the safety policy. This checking must be automated, fast, and highly reliable. It must be the case that if the proof checks then it is safe to run according to the advertised policy. Thus it will not matter if the code or program is tampered with or garbled. If it still checks it will be safe even if it does not work as desired. This component is provided by the browser developer in the above example.

The main advantage of PCC is that the hard work, compiling the code and producing the proof, is in the hands of the code producer, while only easy work, checking the proof and running the program, is required of the code consumer. This asymmetry allows the hard work to be done once and used many times. This is opposed to analyzing the code every time a new client uses it. This alternative is
not only wasteful of processor time for the client, but much harder than having the producer give a proof, as the client has less information about the code than the producer – no source code, no formal or informal specifications, etc.

8.2 Specification of Safety Policy

The specification of the safety policy by the code consumer consists of:

1. The Safe Assembly Language (SAL).
2. A program logic.
3. Preconditions and postconditions for code entry and system calls.

These three components interact to produce a definite notion of safety which is communicated to the code producer. Of the components, only the preconditions and postconditions need to be explicitly sent to code producers. The logic is at least partly specified by the PCC infrastructure and at most extensions to the generic base must be sent. SAL is the same for all safety policies and its contribution to the safety policy is embodied by the verification condition generator discussed in the following section.

SAL is a generic assembly language designed to be minimal and easily translated into actual assembly languages of real processors. It provides an unspecified number of general purpose registers \( r_1, \ldots, r_x \), a stack pointer register \( sp \), a return address register \( ra \), a program counter register \( pc \), and a memory state pseudo-register \( mem \). The operational semantics for SAL instructions are given in Table 8.1. The \( F_i \) column gives the instructions, the \( \Sigma' \) column the resulting state after the execution of the instruction, the control flow test splits the results of a single instruction into multiple results depending on some condition, and the safety requirement column.
Table 8.1: Operational Semantics for SAL

gives the safety requirement that must be fulfilled for an instruction to be executed. States are specified as \(<i, \rho, \mathcal{H}>\) where \(i\) is the value of the program counter, \(\rho\) is a function from the registers to values with values associated with the \texttt{mem} register being functions from base values to base values, while all other registers have base values as associated values, and \(\mathcal{H}\) is the call history represented as a sequence of register states. The table leaves out one safety requirement because it is the same for all instructions, \(i \in \text{Dom}(F)\). This simply means that the program counter does not point outside the code for the function currently on the stack. Several of the properties appearing in the safety requirement column — \textit{SafeEOP}, \textit{SafeCOP}, \textit{SafeRd}, and \textit{SafeWr} — are left undefined. These properties are defined in the program logic, preconditions, or postconditions and are left generic at this point.

The instructions are all fairly standard but need some explanation. The first two — \(r \leftarrow r'\) and \(r \leftarrow n\) — are simple register updates which increment \(i\) and update the state of the register \(r\) in \(\rho\) but have no safety requirements. The next instruction is a generic expression operator \(r \leftarrow r'\text{EOP}_{r''}\), which represents all varieties of elementary
operators such as arithmetic, logical, and shift operators. The register \( r \) is updated by the result of the actual operator on the contents of its arguments and \( i \) is incremented. There is also a safety requirement, \( \text{SafeEOP}(\rho(r'), \rho(r'')) \), which assures that the operation is safe for the contents of the argument registers. The \( \text{jump} \ n \) instruction adds its argument \( n \) to \( i \) and has no safety requirement. The next two rows deal with the generic conditional instruction \( \text{condCOP}(r) \), \( n \) which acts as \( \text{jump} \ n \) if the condition is true and simply increments the program counter otherwise. In either case \( \text{SafeCOP}(\rho(r)) \) must be true, assuring that the conditional operation is allowed on the contents of the argument register. The next instruction \( r \leftarrow M[r'] \) is a memory read which increments the program counter and updates the argument register with the value at the argument address. It requires that \( \text{SafeRd}(\rho(\text{mem}), \rho(r')) \) be true, which assures that it is permissible to read the memory location in the current memory state. The memory write instruction \( M[r'] \leftarrow r \) follows. It increments the program counter and updates the memory register with the value at the address argument replaced by the value at the register argument. It requires that \( \text{SafeWr}(\rho(\text{mem}), \rho(r'), \rho(r)) \) be true, which assures that it is permissible to write the value type to the memory location in the current memory state. The next three instructions deal with the stack and are similar to previous instructions. They have a single safety requirement which ensures that the operations stay within the stack region. The stack pointer increment instruction \( \text{~sp} \leftarrow \text{~sp} + n \) — updates the stack pointer to its incremented value, the stack read instruction \( r \leftarrow M[\text{sp} + n] \) — updates the register \( r \) with the value of the memory at the stack pointer plus \( n \), and the stack write instruction \( M[\text{sp} + n] \leftarrow r \) — updates the memory at the stack pointer plus \( n \) with the value at \( r \). The \( \text{ra} \leftarrow \text{pc} + n \) instruction sets the return address register to the program counter plus \( n \). It requires that the new return address is in the code region for the function. The \( \text{call} \ G \) instruction is a function call. If a function call has sufficient stack space for its arguments, it updates the program counter to the entrance of the function and adds the state to the call history. Oth-
erwise it does not alter the state at all, which corresponds to repeating the function call. In either case it is required that the argument is a function, and if the call succeeds, it is required that if the function is an entry point or a system call that the preconditions must be satisfied. The next instruction is the return instruction \texttt{ret}, which updates the program counter to the return address and removes a state from the call history. The safety requirements are fairly extensive. The new call history must be the same as the current history minus the state at the time of the call, and if the function is an entry or system function, then $SafeRET_{F}(\rho', \rho)$ must be true. This is equivalent to satisfying the post condition of the function, preserving callee-save registers, returning the stack pointer to the stack of the calling function, and preserving all stack contents outside of the functions stack frame. The \texttt{Annot} instruction is simply a place to put non-executed annotations. It increments $i$ and does not have any safety requirements.

An interpreter for SAL would be only able to execute safe programs because all potentially unsafe instructions are checked for safety at the time of execution. Of course, there will not be such an interpreter for PCC as the safety checks are not generally feasible. For instance, if the $SafeRd$ property relied on the state of the memory then it could require a complete scan of the memory each time it is called. Instead the safety checks will be statically proven for a piece of code which will run without run-time checks of the safety properties.

A logic for reasoning about assembly level programs provides both a medium for proving properties and a method of further specifying safety properties. The base logic for our version of PCC is an extended first order logic. Figure 8.1 gives the syntax for the base logic. Base expressions denote values which can fit into a single word in memory or a register. They consist of variables, constants, a memory read operation \texttt{sel}, a generic elementary operation \texttt{eop}, and an instruction address offset expression \texttt{offset}. Store expressions denote values of the memory at some par-
Chapter 8. Proof-Carrying Code

Base Expressions:
\[ E^b ::= x^b \mid c^b \mid \text{sel}(E^s, E^b) \mid \text{eop}(E_1^b, E_2^b) \mid \text{offset}(E_1^b, E_2^b) \]

Store Expressions:
\[ E^s ::= x^s \mid c^s \mid \text{upd}(E^s, E_1^b, E_2^b) \]

Predicates:
\[ P ::= \text{true} \mid P_1 \land P_2 \mid P_1 \supset P_2 \mid \forall x^b.P \mid \forall x^s.P \mid E_1^b = E_2^b \mid E_1^b \neq E_2^b \mid E_1^s = E_2^s \mid E_1^s \neq E_2^s \mid \text{cop}(E^b) \mid \text{notcop}(E^b) \mid \text{safeop}(E_1^b, E_2^b) \mid \text{safecop}(E^b) \mid \text{saferd}(E^s, E^b) \mid \text{safewr}(E^s, E_1^b, E_2^b) \]

Figure 8.1: Syntax of Program Logic

...ticular time and consist of variables, constants, and a memory write operation \text{upd}. Predicates denote truth values and consist of propositional expressions, quantifier expressions, equality expressions, conditional expressions, and safety expressions. Only the conditional and safety expressions need any explanation. The \text{cop} and \text{notcop} expressions denote the result of the generic conditional operators which can be taken as comparison success and failure respectively. The \text{safeop}, \text{safecop}, \text{saferd}, and \text{safewr} expressions denote the safety value of elementary operations, conditional operations, memory reads and memory writes respectively.

Because it is necessary to produce proofs with the logic, there will have to be a deduction system. This will be the natural deduction system given in figure 8.2. The rules for first order logic and equality are all standard and require no explanation. The rules for memory variables are not so standard and require a bit of explanation. The \text{mc0} rule states that any memory location updated with a value gives that value when read. The \text{mc1} rule states that updating a memory location does not change the values of any other memory location. This axiomatization seems to leave out many types of expressions. There are no rules for the conditional, safety, elementary operation, or offset expressions. This is only an apparent deficiency however as the
First order logic:

\[
\begin{align*}
\vdash & \text{true} \quad \text{truei} \\
\vdash P_1 & \quad \vdash P_2 \quad \text{and} \quad \vdash P_1 \land P_2 \quad \text{andel} \quad \vdash P_1 \land P_2 \quad \text{ander} \\
\vdash P_1 & \quad \vdash u \quad \vdash P_1 \supset P_2 \quad \text{impli} \quad \vdash P_1 \supset P_2 \quad \vdash P_1 \quad \text{imple} \\
\vdash \forall x.P & \quad \vdash [a/x]P \quad \text{all} \quad \vdash \forall x.P \quad \text{alle}
\end{align*}
\]

Equality:

\[
\begin{align*}
\vdash E=E & \quad \vdash E_1=E_2 \quad \vdash [E_1/x]P \quad \text{equiv} \quad \vdash [E_2/x]P \quad \text{congr} \\
\vdash E_1^b=E_2^b & \quad \vdash E_1^b \neq E_2^b \quad \vdash u \quad \vdash v \\
\vdash P & \quad \vdash P \quad \text{case} \quad \vdash E_1^b=E_2^b \quad \vdash P \quad \vdash E_1^b \neq E_2^b \quad \text{contr}
\end{align*}
\]

Memory variables:

\[
\begin{align*}
\vdash \text{sel}(\text{upd}(E^a,E_1^b,E_2^b),E_1^b) = E_2^b \quad \text{mc0} \\
\vdash E_1^b \neq E_2^b \\
\vdash \text{sel}(\text{upd}(E^a,E_1^b,E_2^b),E_3^b) = \text{sel}(E^a,E_3^b) \quad \text{mc1}
\end{align*}
\]

Figure 8.2: Deduction System for the Program Logic

The \text{congr} rule can be used to reason about any of the predicate expressions while the conditional, elementary operation, and offset expressions are machine dependent and require extensions to the axiomatization depending on the target architecture.
Chapter 8. Proof-Carrying Code

One further question is where do the safety predicates get introduced? The other architecture dependent expressions can have constructors specified in their extensions to the logic so it seems that there must also be extensions for the safety predicates. This is not necessarily correct. It will soon be shown that these predicates can be introduced in preconditions and postconditions. Even if there is no requirement for extensions to introduce safety predicates there is a reason to use them. Sometimes the safety of an operation can be determined without reference to a specific program. An example is that if it is safe to write to a memory location then it is safe to read that location. This type of safety condition can be expressed as an extension to the axiomatization of the logic. In this way, extensions to the logic can be used to further express the safety policy.

The final portion of the safety policy is preconditions and postconditions for the program and any procedures which are exported by the host environment to the program. The preconditions for the program and postconditions for exported procedures guarantee the truth of certain predicates at their entry or exit points and can introduce new elements into the safety property. An example is a function which has the postcondition that it is safe to read from a selection of memory addresses. Postconditions for the program and preconditions for exported procedures can also introduce new requirements for the program and thus also effect the safety policy.

The three portions of the safety policy give a complex but fairly complete method of specifying the concrete safety policy. The practical issues of stating a safety policy are beyond the scope of this thesis but it can be seen by the rather involved explanation of the safety policy that it is potentially a very intricate matter. Unfortunately this is one of the weak elements of PCC as an improperly stated safety policy can leave the environment open to attacks and bugs that were supposed to be disallowed.
8.3 Verification Condition Generator

The safety policy as described in the previous section is still too abstract for use in PCC. The concrete embodiment of the safety policy is the verification generator. The purpose of the verification condition generator is to take machine language programs with attached annotations and produce a predicate in the logic specified by the safety policy. This predicate will be provable if and only if the program is safe according to the safety policy.

The basic idea of the verification condition generator is that it symbolically evaluates machine language programs as SAL programs to create a predicate which if provable guarantees that all the SAL safety checks are satisfied. The verification condition generator must be specialized for the logic of the safety policy and uses precondition and postcondition code annotations. Thus every aspect of the safety policy influences the verification condition generation.

The verification generator uses a symbolic evaluation function which takes seven parameters and returns a formula in the program logic \( SE_{F; \sigma_0^F, i_0^F} (i, \sigma^F, s_0, L) \). The three subscripted parameters are constant over any particular function and are usually dropped. \( F \) is the function being evaluated, \( \sigma_0^F \) is the state of the registers at the start of execution of the function, and \( i_0^F \) is the address where the function starts execution. The remaining four variables are variable inside a function call. \( i \) is the offset of the current instruction from \( i_0^F \), \( \sigma^F \) is current register state, \( s_0 \) is the stack offset which is the difference between the top of the current stack frame and the current value of the stack pointer, and \( L \) is a list of pairs of addresses and register states where loop invariants are encountered. The set of registers is different for every function and is denoted \( Regs_F \). This set includes the actual general purpose registers \( r_i \), the return address register \( ra \), a pseudo-register \( mem \) which contains the memory state, and a function dependent number of local variables \( l_i \). The formula returned
has variables representing registers and memory in a different states of execution. Some of these are bound but some are not. In particular the variables in the initial state are left unbound.

Each function in a program is symbolically evaluated separately and the final verification condition is the conjunction of each function’s result. Each run of the symbolic evaluator on a new function needs to be initialized with new logical variables for each register, a new variable for the address where the function starts, 0 for the program counter, the number of arguments for the stack offset, and an empty loop map. The final result for a function is the predicate stating that the precondition of the function implies the symbolic evaluation of the function with each of the newly introduced variables universally quantified. This quantification binds the variables which are left unbound by the symbolic evaluator.

Before examining the symbolic evaluator it is necessary to discuss the needed program annotations. Two types of annotations are required. Loop invariant annotations give the symbolic evaluator information about predicates which remain true for any execution of the loop. This allows the evaluator to avoid iterating through the loop to produce an expression. This is good because such an evaluation is undecidable, it would imply a solution to the halting problem. A loop invariant is placed at the address of any backward jumps to ensure that any loops have invariants. The form of the loop invariant annotation is a triple specifying the invariant predicate $P$, the stack offset at the beginning of the loop $n$ and the set of registers modified by the loop $Mod$. The second type of annotation is the function specification. This includes the precondition $Pre_F$, postcondition $Post_F$, callee save registers $CS_F$, number of arguments $Arg_F$, and maximum number of local variables on the stack $Local_F$.

The operation of the symbolic evaluator for the verification condition generator is given in table 8.2. For each instruction $F_i$ there is a check which is a simple syntactic check and a verification condition which gives the current state of the verification
### Table 8.2: Verification Condition Generator for SAL

<table>
<thead>
<tr>
<th>FI</th>
<th>Check</th>
<th>Verification Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r \leftarrow r'$</td>
<td>$i + + \in \text{Dom}(F)$</td>
<td>$\text{SE}(i + +, \sigma[r \mapsto r'], \text{so}, \text{L})$</td>
</tr>
<tr>
<td>$r \leftarrow n$</td>
<td>$i + + \in \text{Dom}(F)$</td>
<td>$\text{SE}(i + +, \sigma[r \mapsto n], \text{so}, \text{L})$</td>
</tr>
<tr>
<td>$r \leftarrow r' \text{\texttt{EOp}}^n$</td>
<td>$i + + \in \text{Dom}(F)$</td>
<td>safeesop$(\sigma(r'), \sigma(r')) \land \text{SE}(i + +, \sigma[r \mapsto \text{EOp}(r')], \text{so}, \text{L})$</td>
</tr>
<tr>
<td>jump $n$</td>
<td>$n + i + + \in \text{Dom}(F)$</td>
<td>$\text{SE}(n + i + +, \sigma, \text{so}, \text{L})$</td>
</tr>
<tr>
<td>condOp$(r, n)$</td>
<td>$i + + \in \text{Dom}(F)$</td>
<td>safeesop$(\sigma(r)) \land \text{SE}(n + i + +, \sigma, \text{so}, \text{L})$</td>
</tr>
<tr>
<td>$r \leftarrow M[r']$</td>
<td>$i + + \in \text{Dom}(F)$</td>
<td>safeesop$(\sigma(r)) \land \text{SE}(i + +, \sigma[r \mapsto \text{sel}(\sigma(mem), \sigma(r))], \text{so}, \text{L})$</td>
</tr>
<tr>
<td>$M[r'] \leftarrow r$</td>
<td>$i + + \in \text{Dom}(F)$</td>
<td>safeesop$(\sigma(r)), \sigma(r')) \land \text{SE}(i + +, \sigma[r \mapsto \text{sel}(\sigma(mem), \sigma(r'), \sigma(r))), \text{so}, \text{L})$</td>
</tr>
<tr>
<td>sp $\leftarrow sp + n$</td>
<td>$1 \leq \text{so} - n \leq \text{LocalF}$</td>
<td>$\text{SE}(i + +, \sigma[so \mapsto n - \text{so}, \text{L})$</td>
</tr>
<tr>
<td>sp $\leftarrow sp + n$</td>
<td>$i + + \in \text{Dom}(F)$</td>
<td>$\text{SE}(i + +, \sigma[so \mapsto \sigma(i - n)], \text{so}, \text{L})$</td>
</tr>
<tr>
<td>$r \leftarrow M[sp + n]$</td>
<td>$1 \leq \text{so} - n \leq \text{LocalF}$</td>
<td>$\text{SE}(i + +, \sigma[so \mapsto \sigma(i - n)], \text{so}, \text{L})$</td>
</tr>
<tr>
<td>$M[sp + n] \leftarrow r$</td>
<td>$i + + \in \text{Dom}(F)$</td>
<td>$\text{SE}(i + +, \sigma[so \mapsto \sigma(r)], \text{so}, \text{L})$</td>
</tr>
<tr>
<td>ra $\leftarrow pc + n$</td>
<td>$n + i + + \in \text{Dom}(F)$</td>
<td>$\text{SE}(i + +, \sigma[ra \mapsto \	ext{offset}(io, n + i + +)], \text{so}, \text{L})$</td>
</tr>
<tr>
<td>call $G$</td>
<td>$G \in \Phi$</td>
<td>$\sigma_1^G(\text{Pre}<em>G) \land \forall y_1 \ldots y_k, z</em>{so + 1}, \ldots, z_{\text{LocalF}}$</td>
</tr>
<tr>
<td>$\sigma(ra) \equiv \text{offset}(io, i + +)$</td>
<td>$\sigma_2^G(\text{Post}_G)$</td>
<td>$\text{SE}(i + +, \sigma_2^G, \text{so}, \text{L})$</td>
</tr>
<tr>
<td>$i + + \in \text{Dom}(F)$</td>
<td>where:</td>
<td></td>
</tr>
<tr>
<td>$so \leq \text{Arg}_G$</td>
<td>$\sigma_1^G(\text{CopyIn}_G(\sigma', \text{Arg}<em>G, \text{so}, {r_1, \ldots, r_k} = \text{Reg}</em>{G'} - \text{CS}_G$</td>
<td></td>
</tr>
<tr>
<td>$y_1, \ldots, y_k, z_{so + 1}, \ldots, z_{\text{LocalF}}$ are new variables</td>
<td>$\sigma_2^G(\text{CopyOut}<em>G(\sigma_2^G, \sigma', \text{Arg}<em>G, \text{so}, {z</em>{so + 1}, \ldots, z</em>{\text{LocalF}}})$</td>
<td></td>
</tr>
<tr>
<td>$i + + \in \text{Dom}(F)$</td>
<td>ret</td>
<td>$\text{SE}(i + +, \sigma[ra \mapsto \text{offset}(io, n + i + +)], \text{so}, \text{L})$</td>
</tr>
<tr>
<td>$so = \text{Arg}_G$</td>
<td>$\sigma(\text{Pre}_G) \land \text{CheckEq}(\sigma, \text{so}, \text{CS}_G$</td>
<td></td>
</tr>
<tr>
<td>inv$P, n, {r_1, \ldots, r_k}$</td>
<td>$i \in \text{Dom}(L)$</td>
<td>$\sigma(P) \land \text{CheckEq}(\sigma, \text{so}, \text{L} + (i, \sigma'))$</td>
</tr>
<tr>
<td>$n = so$</td>
<td>where:</td>
<td></td>
</tr>
<tr>
<td>$i + + \in \text{Dom}(F)$</td>
<td>$y_1, \ldots, y_k$ are new variables</td>
<td></td>
</tr>
<tr>
<td>$\sigma' = \sigma[r_1 \mapsto y_1, \ldots, r_k \mapsto y_k]$</td>
<td>$\sigma(P) \land \text{CheckEq}(\sigma, \text{so}, \text{L} + (i)$</td>
<td></td>
</tr>
<tr>
<td>inv$P, n, {r_1, \ldots, r_k}$</td>
<td>$n = so$</td>
<td></td>
</tr>
<tr>
<td>$i \in \text{Dom}(L)$</td>
<td>$\sigma(P) \land \text{CheckEq}(\sigma, \text{so}, \text{L} + (i, \sigma'))$</td>
<td></td>
</tr>
</tbody>
</table>

condition at the current level of recursion. If the syntactic check fails then the program is rejected by the evaluator, otherwise it is recursively evaluated to build the full verification condition.

The first two instructions — $r \leftarrow r'$ and $r \leftarrow n$ — are for writing a register value or constant to a register. For these instructions the verification condition generator must first check that the next instruction is in the domain of the function. If this
check succeeds then the symbolic evaluator recurses to the next instruction with an updated register state.

The $r ← r'\text{EOP}r''$ instruction is similar but for writing the result of an elementary operation on two registers to another register. It has the same check as the previous two instructions but a more complex verification condition. This is the conjunction of a predicate stating that it is safe to apply the operation to the values of the registers, and the result of applying the evaluator to the next instruction with the state updated to reflect the result of the operation.

The jump $n$ and condCOP($r$), $n$ instructions and are similar to each other. The jump instruction requires that the target of the jump is in the domain of the function and that if the jump is backwards that the instruction at the target be a loop invariant. The symbolic evaluator resumes evaluation at the new instruction. The conditional instruction has the same checks plus one to ensure that the next instruction is in the domain of the function for cases when the branch is not taken. The verification condition is more complex. It is the conjunction of a predicate stating the safety of the conditional operation on the register’s value, a predicate stating the conditional success predicate applied to the value of the argument register implies the result of the symbolic evaluation function applied at the target address, and a predicate stating conditional failure predicate applied to the value of the argument register implies the result of the symbolic evaluation function applied at the next address.

The memory read and write instructions — $r ← M[r']$ and $M[r'] ← r$ — prompt a check that the next instruction is in the function’s domain. The symbolic evaluator conjoins either the safety predicate that it is safe to read the memory in the current memory state at the memory location specified by $r'$, or that it is safe to write to the memory in the current memory state at the location specified by $r'$ with the value in $r$, with the result of the symbolic evaluator run on the next instruction in the
updated state.

The next three instructions — $sp \leftarrow sp + n$, $r \leftarrow M[sp + n]$, and $M[sp + n] \leftarrow r$ — deal with stack operations. They require that the next instruction is in the function’s domain and that the stack offset minus the increment to the stack pointer stays within the stack boundaries. The symbolic evaluator goes on to the next instruction with suitably updated register state.

The $ra \leftarrow pc + n$ instruction for writing to the return address register has two checks. One for ensuring that the return address specified is in the domain of the function and the other ensuring that the next instruction is in the domain of the function. The symbolic evaluator recurses on the next instruction with the return address register updated to the sum of the function’s beginning instruction and the value written to the return address register. This reflects the relative nature of the instruction which allows the function to be relocated.

The call instruction evaluation is very complex. The checks for the instruction ensure that the function called is a function in the program or an exported function from the environment, that the return register points to the next instruction in the calling function, that the next instruction is in the domain of the calling function, and that there are a sufficient number of arguments on the stack. The basic verification condition is easy to understand. It states that the called function precondition holds, and that the postcondition of the called function implies that the predicate resulting from the symbolic evaluation of the next instruction of the calling function holds. The additional machinery is to ensure that any registers possibly changed by the called function are universally quantified so that nothing can be assumed about them, and that registers are suitably renamed for the precondition and postcondition of the called function. Figure 8.3 defines some helper functions that are used here.

For the ret instruction the symbolic evaluator checks that the stack offset is
Chapter 8. Proof-Carrying Code

\[
\text{CopyIn}^G(\sigma, \text{Arg}, \text{so})(r) = \begin{cases} 
\sigma^G(l_{\text{Arg}} + i) & \text{if } r \in \{l_i | 1 \leq i \leq \text{Arg} \} \\
0 & \text{if } r \in \{l_i | \text{Arg} < i \leq \text{Local} \} \\
\sigma^G(r) & \text{if } r \in \{x_1, \ldots, x_n, \text{ra, mem} \}
\end{cases}
\]

\[
\text{CopyOut}^G(\sigma^G, \sigma, \text{Arg}, \text{so}, \{z_{m+1}, \ldots, z_{\text{Local}}\})(r) = \begin{cases} 
\sigma^G(r) & \text{if } r \in \{l_i | 1 \leq i < \text{so} - \text{Arg} + 1 \} \\
\sigma^G(l_{\text{so} - \text{Arg}} + i) & \text{if } r \in \{l_i | \text{so} - \text{Arg} + 1 \leq i \leq \text{so} \} \\
z_i & \text{if } r \in \{l_i | \text{so} < i \leq \text{Local} \} \\
\sigma^G(r) & \text{if } r \in \{x_1, \ldots, x_n, \text{ra, mem} \}
\end{cases}
\]

\[
\text{CheckEq}(\sigma, \sigma', \text{CS}) = \bigwedge_{r \in \text{CS}} \sigma(r) = \sigma'(r)
\]

Figure 8.3: Helper Functions for Verification Condition Generator

back where it started so there is nothing of importance on the stack. It returns the
conjunction of the postcondition evaluated in the current register state and a check
that all callee save registers are the same as when the function was called. The
recursion of the symbolic evaluator stops for this instruction.

The last instruction is the loop invariant \texttt{inv P, n, \{r_1 \ldots r_k\}}. There is one case
for the first time a loop invariant is seen and another for all subsequent times. If the
invariant has not yet been seen then there is a check that the stack offset specified by
the invariant is equal to the stack offset register and a check that the next instruction
is in the domain of the function. The verification condition is the conjunction of the
evaluation of the invariant in the current state, and the predicate stating that the
invariant holding in any state where the loop modifiable variables have any value
implies the predicate returned by the symbolic evaluator run on the next instruction
in the new state with the loop invariant map updated to account for the current
invariant. The case where the invariant has already been seen requires the same
checks but the verification condition is the evaluation of the invariant in the current
state conjoined with a predicate asserting that current state and the state associated
with the invariant by the loop invariant map differ only at registers modifiable by
the loop. This is another case where the symbolic evaluation halts.
Chapter 8. Proof-Carrying Code

As the verification condition generator is the most complex piece of software that the PCC user must trust, it is important that it be sound. The full proof of its soundness is outside the scope of this thesis but is supplied with the original specification, and sketched in Chapter 11. In broad terms, the soundness proof shows by induction that in any reachable state a function has either completed or it can continue to a safe state. This implies that programs do not need to halt to be safe, which agrees with the SAL’s operational semantics which makes programs that run out of memory repeat the same state indefinitely.

8.4 Encoding of Proofs

Getting a verification condition for a program is not sufficient by itself for PCC to function. There needs to be a proof of this verification condition to transmit with the program. What form this proof will take is obvious from a broad perspective but more complex when closely examined. There is already a logic defined for the safety policy so this should be the logic for proving safety. Unfortunately, the logic as described is not well suited to transmit the proof. This is because it does not have a fully specified, concrete syntax needed for efficient checking by the code receiver.

The desiderata for a proof encoding include unambiguous representation, ease of checking, and compactness. The LF logical framework provides a solution for the first two. LF is a language for expressing logics. It is essentially a typed lambda calculus with the ability to encode a wide range of logics. One of the most important features of LF for PCC is that proofs can be equated with expressions of a specific type. This means that if the verification condition is of type \( A \) then any valid proof of it will be of type \( \text{proof} \, A \). This allows for a very simple implementation of the proof checker as a type checker. The representation of any logic in LF is unambiguous as it has structural rules which remove any potential ambiguity and a simple linear
representation for expressions. This contrasts with the previous presentation of the safety logic which had tree structures for the deduction rules.

The major problem with LF is the size of the proofs. Proofs in LF tend to include redundant subexpressions which can cause exponential increases in the size of the proof as the predicate to be proven increases in size. Necula proposes a modification of LF called implicit LF or LF_i. In LF_i placeholders are inserted for sub-terms of the proof that can be recovered by matching the proof to the type of the verification condition. This allows many redundant terms to be eliminated but at the expense of a more complex proof checker.

8.5 Production of Annotations and Proofs

In addition to the executable code for the program there are two additional components that the code producer must include in the PCC system. These are the various code annotations and the proof of safety. The executable can be created by a compiler but these two new components must be created also. There is a wide range of possibilities because the client in PCC does not care how they are created, just that they are correct.

One thing to note about the annotations and proof is that any process by which the annotations can be automatically generated can be extended to produce a proof. This is because a method for producing loop invariants implicitly proves several things about the loop. These include that the precondition for the loop, the invariant, and the loop halting condition imply the postcondition for the loop, and that the precondition for the loop, the invariant, and the loop continuing condition imply the safety property of the loop body. Once these predicates and others concerning the loop are proved, the rest of the proof is easily derived from standard Floyd-Hoare logic.
Chapter 8. Proof-Carrying Code

The least likely method of producing the annotations and proof is by hand. This method will work but would require more effort than the average or even the exceptional programmer would be willing to do. It is hard enough to produce proofs of program safety for high level abstractions, much less machine code. A better approach would be to insert annotations into the source code and derive machine level annotations and a proof from them. This would require much less effort and has the advantage of decidability. Unfortunately, it would not be very easy to figure out the correct annotations for the source code when the safety policy refers to machine code. This could lead to a repeated cycle of modification of annotations and recompilation with the hope that eventually the annotations would be correct. To be really practical, PCC needs to have a fully automated method of producing annotations and proofs. This can not be fully achieved for all possible safety policies because determining loop invariants is in general undecidable. There are special cases of safety policies such as some type based safety policies, for which the loop invariant problem is decidable. In cases such as these, it is possible to use a procedure to determine loop invariants within the compiler and then produce the proof using a theorem prover.
Chapter 9

Combination of Proof-Carrying Code and Model Checking

A new approach to the safety problem for reconfigurable processors is the combination of proof-carrying code and model checking. In this approach proof-carrying code is used for the traditional part of the program while model checking is used for the reconfigurable array configurations. After discussion the limitations of PCC and new safety problems for the reconfigurable array, I will discuss some general assumptions to make the new system tractable and give a brief overview of the new system.

9.1 Limitations of PCC for Reconfigurable Array

Before explaining the new approach, it is helpful to consider why it is required. The PCC system presented is obviously designed with traditional processors in mind. There are three significant impediments to extending PCC to a reconfigurable architecture:
Chapter 9. Combination of Proof-Carrying Code and Model Checking

1. PCC assumes that the program will consist of instructions. The computation done by the reconfigurable array is the same every cycle but chosen from a space more varied than any set of instructions.

2. PCC assumes that the program will be sequential. The computation done by the reconfigurable array is highly parallel.

3. PCC assumes that the program operates on standard sized pieces of data. The reconfigurable array operates on a wide range of fixed size pieces of data.

The problem of PCC assuming an instruction based execution can be seen in the structure of SAL and the verification condition generator. SAL is very similar to traditional processor assembly languages and is simply a set of generic processor instructions with a corresponding operational semantics. Similarly, the verification condition generator uses symbolic evaluation of instructions to produce the verification condition. The symbolic evaluation function is a recursive function which essentially chooses what to do based on the type of instruction it is currently evaluating.

Reconfigurable array configurations, in contrast, are not instruction based or easily amenable to symbolic evaluation. One of the main advantages of the reconfigurable array is that it can compute without relying on fetching instructions. Because the reconfigurable array can implement almost any operation within its size parameters, it is much more varied than any instruction set. To extend PCC to the reconfigurable array would require stretching the idea of an instruction to such an extent that it would include every possible array execution as a separate instruction. This clearly would not be feasible as the verification condition generator would then have an enormous number of cases.

The second problem is that PCC assumes a sequential execution of the program while the reconfigurable array can work in parallel. PCC’s reliance on sequential
execution can be seen in the operational semantics for SAL, the state of the program execution after an instruction includes a new instruction to execute, and the verification condition generator, the symbolic evaluator takes the next instruction to be executed as one of its parameters. While there are instances where the symbolic evaluator recurses into two calls to itself, these do not represent parallel execution but branching execution.

Reconfigurable array execution is inherently parallel however. It has parallel features such as separate functionality for each block in a row which can be used to implement sub-instruction parallelism as discussed in chapter 2, parallel execution of each row which can be used for pipelining, and potentially multiple concurrent accesses to memory which would result in undefined behavior. Clearly this parallelism must be taken into account.

None of these parallel features is easily captured by an extension to the PCC system. The main problem is the non-determinism which is introduced by parallelism. When evaluating an instruction there is no longer complete knowledge of the state of the computation after the instruction completes. Fortunately, this problem is not as severe as the previous one.

It is possible to deal with parallelism by introducing universal quantification into the symbolic evaluation function to represent the unknown state of the rest of the computation. If there is a proof that all values for a register satisfy a property, then there is no need to provide one for any particular value or even any need to determine a particular value. Unfortunately, this only works efficiently when different parallel sections of the computation do not potentially share resources. This is because no process could make any claims about the value of the shared resources and no contracts such as preconditions and postconditions could be guaranteed. There would have to be universal quantification at every access to shared resources, which would lead to verification conditions that were hard to prove because of their generality, and
there could be no postconditions for processes, which would lead to unconstrained generality.

The last problem is that PCC operates on standard sized data, 32 bits in this case. The reconfigurable array operates on a wide range of data sizes, potentially up to the number of registers in the entire array or only 1 bit. This causes a problem when the processor and array operate on the same data through memory or register reads and writes. The main problem here is what granularity to use to represent the computational state. This problem is not as severe as the preceding two and will in fact have to be addressed by the new approach.

9.2 New Safety Issues

There are four new issues for the safety of Garp programs which are not specific to the computational nature of the reconfigurable array as the preceding problems were. These are:

1. The concurrent execution of the processor and array.

2. The nondeterminism of the array execution context for an instruction.

3. The halting of array execution by the processor.

4. The lengthening of array execution by the processor.

The ordinary processor and reconfigurable array can run concurrently in the Garp architecture. This can lead to safety problems because the two sections of the processor could interfere with the other’s execution. This makes it difficult to determine safety properties, especially because Garp does not synchronize the processor and array clocks. Luckily, the potentially most difficult and disastrous case
is ruled out — reading or writing array registers during array execution. Any Garp
instruction that accesses the array registers, or loads or unloads array configurations,
blocks while the array is running. This is a fairly effective, large grain synchronization
between the two sections of the processor which prevents gross interference in the
operation of the array.

The memory access of the two sections is not as well controlled however. There
are methods of accessing memory for both the processor and array which are not
in any way synchronized. They both share the entire memory hierarchy including
caches, physical memory, and virtual memory on disk. There is no guarantee that
any memory access will occur promptly and both the processor and array might block
while waiting for a memory access to complete. This means that it is extremely hard
to reason about the safety of programs without some restriction of memory access
such as requiring software style synchronization of the two sections or limiting shared
access to memory.

The second problem is that there is no guarantee that for any instance of an
instruction, it will occur during array execution or not. This is a problem because
some instructions such as array register accesses behave differently during array
execution than they do when the array is halted. In particular, they block while
the array is executing. This makes it difficult to reason about a number of program
structures. Loops and the instructions used to code them, jumps and conditionals,
can execute in different array execution contexts at different points in a program’s
execution. Similarly, when a function is called or returns, it can be in either array
execution context. Any blocking instruction cannot be easily reasoned about in such
a situation because the state of a value in an array register becomes uncertain even
if the array does not actually execute during the instruction.

This problem is so important that the two array execution contexts will be given
the names 'fresh' and 'stale'. The fresh array execution context is when the array
may be executing. This state exists between any instruction starting the array and any instruction which blocks on array execution. Although it is possible that the array may halt before the blocking instruction is reached, there is no way of knowing this from the processor and is just another aspect of the indeterminism of the array execution. The stale state is any state where it is impossible for the array to be executing and therefore the state of the array registers and memory cannot be changed by the array.

The next problem is that the processor can force the array to stop at any point in its execution. This is a problem because it can leave the array and memory in an indeterminate state. Any reasoning about the array is likely to be incorrect as there is no way to tell when it stopped execution.

Having the processor add to the array’s execution time is also problematic for the same reason as halting it. Because this makes the execution time of the array uncertain, it becomes very difficult to reason about the final state of the array.

## 9.3 Assumptions and Solutions Provided

In order to deal with the three issues from above, there will be four assumptions about programs for Garp:

1. Resource sharing between the processor and array will be restricted.

2. Any instance of an instruction will occur in only one of the two array execution contexts.

3. Halting the array by the processor will be banned.

4. Increasing the array counter by the processor will be banned.
Resource sharing between the processor and array will be limited to times when the array is not executing. This gives the processor complete control over such sharing. This will be achieved by partitioning the memory into processor and array accessible sections for any array configuration. This limits the power of the array somewhat but it is a reasonable limitation. The types of computations for which the processor is good are quite different from those for which the array is good. It is reasonable to assume that there will be little sharing of tasks like that which often occurs in shared memory multiprocessing. The more likely programming model would be having the array do inner loop computations while the processor blocked.

One potential programming method which is banned here is a pipe where the array computes some values and puts them in an array in memory and the processor computes with these values as they become available. This would be a nice use of the resources but either would require synchronization or be very dangerous. It could be approximated by having the array stop at synchronization points and resume with a new block of accessible memory while the processor works on the previous block.

Forcing instructions to only execute in a single array execution context will be done by inserting assertions into the program at certain points. These are at any jump or conditional target and into the preconditions and postconditions of function calls. The instructions which do not deal with the array are not effected by this because it does not matter what context in which they are called. The array specific instructions will all be regarded as a set of two instructions — one for each context.

There are some problems with this approach. Some code duplication might result from having special cases before loops to ensure that the array is fresh or stale during the loop. This should not be too severe. More worrisome, some function calls which do not deal with the array registers or memory could be hampered by the restriction. Such a function would be forced to have two versions depending on the context. For the most part, the restriction is appropriate. It seems reasonable that a program
would be better structured by executing instructions in only one context.

The halting of the array by the processor must be simply banned. While there might be occasions where such an instruction is useful, it is so disruptive of reasoning about safety that it will be completely disallowed. This also applies to increasing the array counter which is also banned.

These assumptions allow solutions to the problems. Array executions can now be regarded as a type of function call. Because they no longer interact with processor accessible resources and can not be prematurely stopped, they become black boxes just like function calls. All that will be known about them to the processor will be contained in preconditions, postconditions, and the memory partition.

Nondeterministic values produced by array execution will be handled by two variants of $\epsilon$-calculus. One will model the inaccessible array registers and memory during array execution and one will model the indeterminate values in these same variables when an instruction accesses a fresh array.

For inaccessible values, the $\epsilon$-calculus will be used to construct values for which the program logic has no applicable rules. Thus any attempt to reason about these values in a proof will fail. There will be rules for reasoning about them in the metalogic so that the soundness of the verification condition generator can be proved with respect to SAL. This is why $\epsilon$-calculus is used instead of ad hoc inaccessible values.

The $\epsilon$-calculus in the second case will only be used for meta-reasoning, and not in the verification condition generator. Universal quantification will also be used in the this case in a manner similar to how it is used to reason about function calls. Because this variant of the $\epsilon$-calculus constructs indeterminate values which are accessible, these values will be universally quantified so that their actual values need not be determined by the verification condition generator. This solution depends on the
Chapter 9. Combination of Proof-Carrying Code and Model Checking

restriction of array execution contexts to be effective. Otherwise there would be excessive universal quantification in the final verification condition, resulting in a bloated proof.

9.4 Adding Model Checking

This discussion has left out how to verify safety for the reconfigurable array. PCC will simply be abandoned as inappropriate here. Not all is lost however. The idea of a safety condition for configuration execution is still tenable. It is only the particular method of producing it that is useless.

If one looks at array execution as a cousin of the function call, then there are properties of the system of PCC which can be preserved. One is that function preconditions and postconditions can be given for an array configuration just as they are for functions. Another is that array configurations can be integrated into the verification condition as if they were functions with the difference that they can be run concurrently with regular processor execution. This concurrency is minimal and can be dealt with by the use of universal quantification in the verification condition and strict separation of resource usage during concurrent execution. This leaves the basic PCC system with the ability to deal with everything but the internal verification conditions of reconfigurable array configurations. These must be produced in a new manner.

While this justifies introducing a new method of generating verification conditions for configurations, it does not explain why model checking has been chosen to verify them. It might be possible to integrate the new type of verification condition in the existing PCC system but it would require substantial modifications to the program logic and a new method of generating proofs. This is mainly because the program logic would have to be able to deal with bit level and register level con-
Chapter 9. Combination of Proof-Carrying Code and Model Checking

structs and translate between them. The problem of the lack of instructions for the reconfigurable array would probably not be solvable in any efficient manner, so this may be impossible. Regardless, it is better to try and preserve the original logic and isolate the reconfigurable array reasoning as much as possible. It is always a good idea to use the method best suited to a particular task when it is possible. Because there is this dichotomy, one becomes free to choose whatever reasoning method is best for each part. Here, model checking has been chosen for the reconfigurable array because it is fully automated, is not based on the semantics of instructions, deals naturally with concurrent constructs, and works well with bit level reasoning.

9.5 General Idea

In order to account for the reconfigurable array, two modifications to the PCC system will be used. One to account for the additional processor instructions for interaction with the reconfigurable array, and another to account for the computation occurring inside the reconfigurable array. The verification condition generator and SAL can be extended to include registers and instructions which the traditional section of the processor uses to control and communicate with the reconfigurable array. Inside the reconfigurable array, model checking will be used to verify safety.

The extension to SAL and the verification condition generator will have little impact on the general operation of PCC. It introduces no new trusted or untrusted system components and requires no radical modifications to SAL, the logic, or the verification condition generator. The new registers and instructions can be treated similarly to the old ones in most cases with only slight complications from overlapping registers and a few simplifying assumptions about memory access and looping.

The use of model checking for the reconfigurable array requires a more radical modification of the PCC architecture. A new trusted system component is intro-
Chapter 9. Combination of Proof-Carrying Code and Model Checking

duced. The code receiver now must have access to a model checker and trust its results. This is because in most cases, model checkers are not certifying. They simply explore a state space without producing any evidence for a positive conclusion for universal statements. Because these are the types of statements which are most important for safety, they will have to be rechecked by the consumer. This leaves the basic PCC architecture in place with the addition that the code consumer must model check the array configurations for a variety of properties for some of which the code producer can send hints. In addition the code producer can send hints about the global model properties such as variable orderings, or about the exact property to be checked. In either case, the type of possible hints must be chosen so that they cannot be used to fool the model checker into false conclusions, but can only help or hinder the efficiency of its operation.
Chapter 10

Extension of SAL

SAL will require three modifications:

1. New registers will be added.
2. Old instructions will be modified.
3. New instructions will be added.

These modifications will also require some assumptions about the behavior of programs as discussed in the previous chapter. These assumptions will be common safety requirements for all safety policies.

1. When the array is running, the ordinary processor component may not access memory that the array might use.
2. At any instruction instance, the array execution context is constant.
3. The processor does not stop the array.
4. The processor does not increase the array counter when the array is running.
10.1 New Registers

The new registers are of five types:

1. $\text{gard}_{n,m}$ and $\text{garz}_{n,m}$ – Reconfigurable array registers.

2. $\text{gaoD}_{n}$ and $\text{gaoZ}_{n}$ – Reconfigurable array register order registers.

3. $\text{gacr}$ – A configuration register.

4. $\text{gacnt}$ – A count register.

5. $\text{gamem}$ – A memory partition register.

The most obvious new registers needed are the array registers. Each row of the array has six registers corresponding to the two register types and three instruction types which can access them. These are accesses to the middle bits, the low bits, and the high bits. For row $n$, these registers will be $\text{gard}_{n,1}$ and $\text{garz}_{n,1}$ for the middle, $\text{gard}_{n,2}$ and $\text{garz}_{n,2}$ for the low, and $\text{gard}_{n,3}$ and $\text{garz}_{n,3}$ for the high.

This addition is not without problems. For each row of the Garp array, the three registers overlap in content for each register type and can be partially overwritten by write operations on each other. As an example, if $\text{garz}_{0,1}$ is updated followed by $\text{garz}_{0,2}$ being updated, the low bits of $\text{garz}_{0,1}$ will be altered. If the updates occur in the opposite order then the high bits of $\text{garz}_{0,2}$ will be altered. Two order registers, $\text{gaoD}_{n}$ for D registers and $\text{gaoZ}_{n}$ for Z registers, will be added per array row $n$ so that the actual contents of the row can be reconstructed from the contents of the three array registers. The content of these registers will be integers from 1 to 6, each of which will represent a different update order. Figure 10.1 shows the orders associated with each possible order register value, with the array registers represented simply as integers indicating if they are high, middle, or low. For example, if the $\text{gaoZ}_{0}$
Chapter 10. Extension of SAL

\[1 \equiv (1, 2, 3), 2 \equiv (1, 3, 2), 3 \equiv (2, 1, 3)\]
\[4 \equiv (2, 3, 1), 5 \equiv (3, 1, 2), 6 \equiv (3, 2, 1)\]

Figure 10.1: Order Definition

register contains a 5, then the order of the updates on the Z registers of the first row
would have been the high bits, followed by the middle bits, followed by the low bits.

The gacr register holds a pointer to the current configuration of the reconfigurable
array. Because the first word of any configuration is its size in rows, gacr needs only
to contain a pointer to this first word. When there is no configuration loaded the
value of the gacr register will be 0.

The gacnt register holds the value of the count when the array is started. The
array execution count is independent of the main processor cycle so it will be either
the count when the array was started or 0. The verification condition generator will
use the start value of gacnt to ensure that preconditions are met for array execution
and the 0 value to indicate that the array is halted. This works even though in
general the processor has no way of determining if the array has actually halted
because an instruction either is not influenced by the state of the array or else it
blocks waiting for the array to halt. This allows the gacnt register to be zeroed by
many instructions as if the array halted during their execution.

The gamem register marks the memory addresses accessible by the array. Because
the processor and the array can both access memory during array execution, there
must be some method of ensuring a consistent memory state. The gamem will do this
by partitioning the memory into processor and array accessible sections. Let \(\sigma\) be
some state of the computation, then the function \(\sigma(gamem)\sigma(r)\) will return true if
the array can access memory at the address in register \(r\) in state \(\sigma\), and false if the
processor can address that memory in state \(\sigma\). One constraint on the value of gamem
is that the runtime execution stack must always be in the processor’s memory region.
so that the reconfigurable array does not alter the arguments or data of functions while they are on the stack. While it is necessary to be accessible according to this memory partition for a memory access to be safe, it is not sufficient. The safety properties for memory access in general must also be satisfied by either processor or array memory accesses. This register can be set according to a code producer supplied hint and may set aside more or less memory for the array than is safe to access for succinctness.

10.2 Modifications of Old Instructions

The only modification necessary for the old instructions is for the memory access instructions \( \text{\texttt{r \leftarrow M[r']}} \) and \( \text{\texttt{M[r'] \leftarrow r}} \). This is because they are the only ones which deal with memory or registers that the reconfigurable array can possibly modify. Therefore an additional safety requirement \( \neg \rho(\text{gmem}) \rho(r') \) is needed to state that the memory location accessed is in the processor memory partition.

All other instructions presented in Table 8.1 are not modified. The stack access instructions do not have to be altered even though they access memory because the stack is always in the processor memory partition. The other instructions do not interact with array accessible memory or registers at all.

10.3 Extension to New Instructions

The new array specific instructions are a subset of the actual Garp instructions for interacting with the array. The instructions left out deal with:

1. Loading configurations in steps and at offsets.
Chapter 10. Extension of SAL

2. Flushing the array configuration cache.

3. Loading and storing the memory queue control and array state.

4. Stopping the array.

These instructions are ignored for a variety of reasons.

The memory queues are being ignored in general because they have a separate configuration file and add no inherently interesting functionality beyond what can be done with the regular memory access mechanisms. It may in fact be easier to model check memory access with the memory queues because they only allow sequential accesses, so no interesting complications to the system are being left out. Thus all instructions involving them are ignored.

Loading and storing array state is primarily for context switches by the operating system. Hopefully, the operating system will be trusted to carry out context switches safely, and these instructions can be ignored. While it is possible to use these instructions in an untrusted program, it is difficult to see any reason why they are necessary or even particularly useful.

The various methods of dealing with configurations are overly complicated and not really necessary. This includes allocating array space without loading a configuration, and loading a configuration into a previously allocated space. The only benefit from these instructions is that more than one configuration can be put into the array simultaneously at different offsets. This is not really necessary as only one of them can run at a time. This can only reduce the configuration switching time.

The instruction for stopping the array is a different matter is ignored because it violates the assumption that the processor does not stop the array. Having the processor stop the array in the middle of a computation leads to undefined states where the array configuration’s postcondition is meaningless. While it might be
reasonable to stop the array in an arbitrary state for some tasks such as context
switches or for random number generation, it is better to simply ban it here.

Tables 10.1 and 10.2 give the extension to SAL for the new instructions. There
are two major additions to the specification machinery of SAL:

1. The \( \epsilon \) function from states to states.
2. The \( \kappa \) function from states to states.

These functions take states and return states with the array accessible registers and
memory modified. They use object constructors — \( \epsilon_a \) and \( \kappa \) — to create unknown
and arbitrary terms respectively.

The \( \epsilon_a \) constructor comes in several varieties. These correspond to each register
that the array can access and all the memory positions. The register variety is
indicated by an ordered triple subscript which indicates the Z or D register, the row,
and the section. The memory variety has a single subscript indicating the memory
position. The \( \epsilon_a \) constructors follow the following rules:

1. \( \exists v. Fv \supset F(\epsilon_a v.Fv) \)
2. \( \forall v(Fv \equiv Gu) \supset \epsilon_a v.Fv = \epsilon_a v.Gv \)

These constructors are intended to model unknown terms in registers and memory
positions. It is possible to reason about them in a very general manner, but only
as far as their bound expressions and equivalence allow. The subscripts ensure that
it is not possible to either equate or differentiate unknown terms in different places.
This formalism is known as Hilbert’s \( \epsilon \)-calculus [Lei69].

The \( \epsilon \) function which uses the \( \epsilon_a \) constructor is defined as:

\[
\epsilon(\rho) = \rho[\text{gar} x_{n,m} \leftarrow \epsilon(x_{n,m}) v.\text{true}, \ldots, \text{mem} \leftarrow \rho(\text{mem})] \rho(w) \leftarrow \epsilon_w v.\text{true}] \quad (10.1)
\]
for all \( x, n, m, \) and \( w \) such that \( \neg \rho(\text{gamem})w \). This function takes a state and substitutes unknown values for all the array accessible registers and memory locations. It is intended to model the strong indeterminacy of these variables from the point of view of the processor while the array is executing.

The \( \kappa \) constructor is similar to the \( \epsilon_{\alpha} \) but it does not come in different varieties or always produce equivalent terms. This constructor is intended to produce arbitrary terms which satisfy a formula following this rule:

1. \( \exists v. Fv \supset F(\kappa v. Fv) \)

The resulting term is potentially different for each different application of the constructor and so is even more difficult to reason about than \( \epsilon_{\alpha} \) terms.

Just as the \( \kappa \) constructor is similar to the \( \epsilon_{\alpha} \) constructor, the \( \kappa \) function is similar to the \( \epsilon \) function. It is defined as:

\[
\kappa(\rho) = \rho[\text{gar} x_n m \gets \kappa v. \text{true}, \ldots, \text{mem} \gets \rho(\text{mem})[\rho(w) \leftarrow \kappa v. \text{true}]]
\]  

(10.2)

for all \( x, n, m, \) and \( w \) such that \( \neg \rho(\text{gamem})w \). This function takes a state and returns a state with all the array accessible registers and memory replaced by arbitrary values. It is intended to model nondeterministic assignment of values to these variables when the array stops executing and all the variables become accessible to the processor.

Two other auxiliary functions are used in the operational semantics of the new instructions:

1. \( \gamma(\text{ord}, \text{pos}) \)
2. \( \delta_{\text{type,row}}(\text{ord}, \text{pos}, \text{state}) \)

The \( \gamma(\text{ord}, \text{pos}) \) function is shown in Figure 10.2. It takes the value of an order register and a register position to be updated and returns a new order with the \text{pos} argument moved to the front of the order. The representation is the same as in 10.1.
Table 10.1: Extended Operational Semantics for SAL

The $\delta_{\text{type,row}}(\text{ord, pos, state})$ function retrieves the actual value of an array register. It is shown in Figure 10.3. This function takes the Z or D type, row, current order for the row and type, register position to be read, and current state; and produces the value of the register. This requires bit level manipulations. The bit array slicing operation $[x..y]$ takes the bits from $x$ to $y$ and discards the rest and the bit array concatenation operator `:' concatenates bit arrays. The $\beta$ function translates from integers to bit arrays and the $\beta'$ function reverses this translation. Thus the result of reading the array can be quite different from the value in the corresponding array register.
Chapter 10. Extension of SAL

<table>
<thead>
<tr>
<th>$F_i$</th>
<th>$\Sigma'$</th>
<th>Control Flow Test</th>
<th>Safety Req.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{mtgavr} \ s$</td>
<td>$&lt; i+ +, \rho</td>
<td>\text{gax}_n, 1 \gets \rho(r)$, \text{gax}_n \leftarrow \gamma(\rho(\text{gax}_n), 1)], H &gt;$</td>
<td>$\rho(\text{gacnt}) = 0$</td>
</tr>
<tr>
<td></td>
<td>where: $\rho(s) = (n : x)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{mtgavr} \ s$</td>
<td>$&lt; i+ +, \rho</td>
<td>\text{gax}_n, 1 \gets \rho(r)$, \text{gax}_n \leftarrow \gamma(\rho(\text{gax}_n), 1), \text{gacnt} \leftarrow 0], H &gt;$</td>
<td>$\rho(\text{gacnt}) \neq 0$</td>
</tr>
<tr>
<td></td>
<td>where: $\rho(s) = (n : x), \rho' = \kappa(\rho)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{mtgavr} \ s$</td>
<td>$&lt; i+ +, \rho</td>
<td>[\text{gax}_n, 1 \gets \rho(r)$, \text{gax}_n \leftarrow \gamma(\rho(\text{gax}_n), 1), \text{gacnt} \leftarrow 0], H &gt;$</td>
<td>$\rho(\text{gacnt}) = 0$</td>
</tr>
<tr>
<td></td>
<td>where: $\rho(s) = (n : x)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{mtgavr} \ s$</td>
<td>$&lt; i+ +, \rho</td>
<td>[\text{gax}_n, 2 \gets \rho(r)$, \text{gax}_n \leftarrow \gamma(\rho(\text{gax}_n), 2), \text{gacnt} \leftarrow 0], H &gt;$</td>
<td>$\rho(\text{gacnt}) \neq 0$</td>
</tr>
<tr>
<td></td>
<td>where: $\rho(s) = (n : x), \rho' = \kappa(\rho)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{mtgavr} \ s$</td>
<td>$&lt; i+ +, \rho</td>
<td>[\text{gax}_n, 2 \gets \rho(r)$, \text{gax}_n \leftarrow \gamma(\rho(\text{gax}_n), 2), \text{gacnt} \leftarrow 0], H &gt;$</td>
<td>$\rho(\text{gacnt}) = 0$</td>
</tr>
<tr>
<td></td>
<td>where: $\rho(s) = (n : x)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{mtgavr} \ s$</td>
<td>$&lt; i+ +, \rho</td>
<td>[\text{gax}_n, 3 \gets \rho(r)$, \text{gax}_n \leftarrow \gamma(\rho(\text{gax}_n), 3), \text{gacnt} \leftarrow 0], H &gt;$</td>
<td>$\rho(\text{gacnt}) \neq 0$</td>
</tr>
<tr>
<td></td>
<td>where: $\rho(s) = (n : x)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{mtgavr} \ s$</td>
<td>$&lt; i+ +, \rho</td>
<td>[\text{gax}_n, 3 \gets \rho(r)$, \text{gax}_n \leftarrow \gamma(\rho(\text{gax}_n), 3), \text{gacnt} \leftarrow 0], H &gt;$</td>
<td>$\rho(\text{gacnt}) = 0$</td>
</tr>
<tr>
<td></td>
<td>where: $\rho(s) = (n : x), \rho' = \kappa(\rho)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{mtgavr} \ s$</td>
<td>$&lt; i+ +, \rho</td>
<td>(r \leftarrow \delta(\rho(\text{gax}_n), 3, \rho'), \text{gacnt} \leftarrow 0], H &gt;$</td>
<td>$\rho(\text{gacnt}) \neq 0$</td>
</tr>
<tr>
<td></td>
<td>where: $\rho(s) = (n : x)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{mtgavr} \ s$</td>
<td>$&lt; i+ +, \rho</td>
<td>[\text{gax}_n, 3 \gets \rho(r)$, \text{gax}_n \leftarrow \gamma(\rho(\text{gax}_n), 3), \text{gacnt} \leftarrow 0], H &gt;$</td>
<td>$\rho(\text{gacnt}) = 0$</td>
</tr>
<tr>
<td></td>
<td>where: $\rho(s) = (n : x), \rho' = \kappa(\rho)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{gambpr}$</td>
<td>$&lt; i+ +, \rho</td>
<td>(\rho'), H &gt;$</td>
<td>$\rho(\text{gacnt}) = 0$</td>
</tr>
<tr>
<td></td>
<td>where: $\rho' = \rho</td>
<td>\text{gacnt} \leftarrow \rho(r)$, \text{gmem} \leftarrow \text{Mem}_p(\rho(\rho'))</td>
<td>$\text{Pre}_p(\rho(\rho'))$</td>
</tr>
</tbody>
</table>

Table 10.2: Extended Operational Semantics for SAL cont.

\[
\begin{align*}
\gamma(1, 1) & \rightarrow 1, \gamma(1, 2) \rightarrow 3, \gamma(1, 3) \rightarrow 5 \\
\gamma(2, 1) & \rightarrow 2, \gamma(2, 2) \rightarrow 3, \gamma(2, 3) \rightarrow 5 \\
\gamma(3, 1) & \rightarrow 1, \gamma(3, 2) \rightarrow 3, \gamma(3, 3) \rightarrow 6 \\
\gamma(4, 1) & \rightarrow 1, \gamma(4, 2) \rightarrow 4, \gamma(4, 3) \rightarrow 6 \\
\gamma(5, 1) & \rightarrow 2, \gamma(5, 2) \rightarrow 4, \gamma(5, 3) \rightarrow 5 \\
\gamma(6, 1) & \rightarrow 2, \gamma(6, 2) \rightarrow 4, \gamma(6, 3) \rightarrow 6
\end{align*}
\]

Figure 10.2: Gamma Order Function

The $\text{gacnf}r$ instruction loads a configuration into the array. This instruction stalls until the array is halted and then loads a new configuration into the array and zeros all the array registers. This is reflected in the resulting new state with $n$ and $m$ ranging over the rows of the array and the three sections of the array registers.
\[\delta_{x,n}(1, 1, \sigma) = \sigma(x_{n,1})\]
\[\delta_{x,n}(1, 2, \sigma) = \beta([23.0] \beta(\sigma(x_{n,1}))) = [7.0] \beta(\sigma(x_{n,2})))\]
\[\delta_{x,n}(1, 3, \sigma) = \beta([13.8] \beta(\sigma(x_{n,3}))) = [31.24] \beta(\sigma(x_{n,1})))\]

\[\delta_{x,n}(2, 1, \sigma) = \sigma(x_{n,1})\]
\[\delta_{x,n}(2, 2, \sigma) = \beta([23.0] \beta(\sigma(x_{n,1}))) = [7.0] \beta(\sigma(x_{n,2})))\]
\[\delta_{x,n}(2, 3, \sigma) = \beta([13.8] \beta(\sigma(x_{n,3}))) = [31.24] \beta(\sigma(x_{n,1})))\]

\[\delta_{x,n}(3, 1, \sigma) = \beta([31.24] \beta(\sigma(x_{n,1}))) = [31.8] \beta(\sigma(x_{n,2})))\]
\[\delta_{x,n}(3, 2, \sigma) = \sigma(x_{n,2})\]
\[\delta_{x,n}(3, 3, \sigma) = \beta([13.8] \beta(\sigma(x_{n,3}))) = [31.24] \beta(\sigma(x_{n,1})))\]

\[\delta_{x,n}(4, 1, \sigma) = \beta([7.0] \beta(\sigma(x_{n,3}))) = [31.8] \beta(\sigma(x_{n,2})))\]
\[\delta_{x,n}(4, 2, \sigma) = \sigma(x_{n,2})\]
\[\delta_{x,n}(4, 3, \sigma) = \sigma(x_{n,3})\]

\[\delta_{x,n}(5, 1, \sigma) = \beta([7.0] \beta(\sigma(x_{n,3}))) = [23.0] \beta(\sigma(x_{n,1})))\]
\[\delta_{x,n}(5, 2, \sigma) = \beta([23.0] \beta(\sigma(x_{n,1}))) = [7.0] \beta(\sigma(x_{n,2})))\]
\[\delta_{x,n}(5, 3, \sigma) = \sigma(x_{n,1})\]

\[\delta_{x,n}(6, 1, \sigma) = \beta([7.0] \beta(\sigma(x_{n,3}))) = [31.8] \beta(\sigma(x_{n,2})))\]
\[\delta_{x,n}(6, 2, \sigma) = \sigma(x_{n,2})\]
\[\delta_{x,n}(6, 2, \sigma) = \sigma(x_{n,3})\]

**Figure 10.3: Delta Result Function**

respectively. In the fresh context of array execution, this new state is constructed by substituting into \(\kappa(\rho)\) instead of \(\rho\). The fresh case can always be recognized by the control flow test ga\textsc{cnt} \(\neq 0\), which means that no instruction has zeroed the counter since the last array execution started. This reflects that the instruction is the first to make accessible to the processor any array accessible variables. In this case the registers are all overwritten but the memory is still accessible and is assigned arbitrary terms.

The safety requirement is that the start of the configuration be in \(\Gamma\) which is a currently under-specified set of starting addresses of configurations that are syn-
tactically valid, and stored in the correct, read only section of the executable. It is
assumed that before untrusted code is considered for execution that it is analyzed
to yield the $\Gamma$ set. It can be created by running a fairly strait-forward analysis over
the configuration section of the executable which checks at least such things as:

1. All configurations give their correct size.
2. All block configuration encodings refer to a valid configuration of a block.
3. No wires have multiple drivers.
4. No blocks require input that is absent.
5. No values travel through too many wires and blocks before getting to their
destinations.

These are all important to proving the safety of a Garp program but do not require
the power of either PCC or model checking to verify and so are relegated to the under-
specified $\Gamma$ set. Section 3.3 also briefly discussed these types of syntactic problems
and methods of checking them.

The $\text{garest}$ instruction simply removes the current configuration from the array.
It stalls if the array is executing but does not affect the state of the array registers in
any way. Its SAL specification simply zeros the configuration and counter registers.
In the fresh case, all the array accessible variables are assigned arbitrary values by
the $\kappa$ function.

The $\text{mtga}$ instruction takes three arguments — the register to get the value from
$r$, the choice of Z or D registers $x$, and the row $n$. The row and Z or D choice is
hard coded into the instruction. Optionally, it can take an argument to increase the
array counter $c$. It stalls if the array is executing and then moves the content of $r$ to
the middle of the array register specified by $x$ and $n$. 
Chapter 10. Extension of SAL

Depending on whether the instruction stalls or starts the array, several alternative updates to the state can occur. If the array is fresh then the update includes a substitution by the $\kappa$ function. If the instruction starts the array — mtgar, $x, n, c$ — there is a substitution by the $\epsilon$ function. This models the inaccessibility and unknown state of the array accessible variables when the array executes. In any case, the new state updates the array register concerned and also the corresponding order array. The $\gamma$ function in figure 10.2 is used to update the order array. If the instruction starts the array the counter is also updated to the $c$ argument.

There is a safety requirement that the $n$ argument be less than the array size which will be fixed for any realization of Garp. The instruction which starts the array execution has another safety requirement — $Pre_{\rho(gacr)}(\rho')$ — which simply states that the array configuration preconditions must be satisfied in an intermediate state after the registers are updated to new values for array execution but before they are cloaked by the $\epsilon$ function. This requirement is needed here because SAL does not have the ability to model array executions, so they are assumed to be correct as long as their preconditions hold. This is similar to system provided functions in the original SAL.

The mfga instruction is like the mtga instruction except for the direction of data movement. It reads from the array. It updates the register $r$ by using the $\delta$ function. The optional counter update and safety requirements are the same as for the mtga instruction. So is the use of the $\epsilon$ and $\kappa$ functions.

The next instructions are mtgav and mfgav, which access the middle section, mtgavv and mfgavv, which access the low section, and mtgavz and mfgavz, which access the high section. They are similar to the preceding four except they take a variable argument for the Z or D choice and the row, and they access different sections of the array registers. The $\rho(s) = (n : x)$ definition for the next state puts the least significant bit of the value in $s$ into $x$ and the rest of the bits shifted to the
right on bit into $n$.

The safety requirement is that the argument in the $s$ register when taken as a multiple bit integer $n$ concatenated with a one bit flag $x$ should be a valid specification of row and Z or D choice. This can be realized by dividing $s$ by 2 to get $n$ and taking its modulus by 2 to get $x$. All references to $n$ and $x$ in these entries and in the corresponding entries in the verification condition generator table should be regarded as abbreviations. The updated state and safety requirements are the same as the previous instructions otherwise.

The last instruction — gbump — increases the array counter which starts the array execution if it is not already executing. It simply updates the gacnt register to the value in the $r$ register and substitutes for the array accessible registers and memory with the $\epsilon$ function.

The safety requirement is that the array is not fresh, there is a configuration loaded, and its preconditions are satisfied. This prevents undefined behaviors from occurring although it is possible for the processor to increase the counter even if it is already running. Here it is simply banned to make model checking the array possible. This means that there is not the usual case split according to array execution context.
Chapter 11

Extension of Verification Condition Generator

The verification condition generator must be modified to correspond to the extension of SAL. Both modified rules for the old instructions and new rules for new array related instructions will be given. Finally, an argument for the soundness of the extension will be given.

11.1 Modification for Old Instructions

Tables 11.1 and 11.2 show the modifications to the verification condition generator for the instructions given previously. Any instruction not mentioned in the new tables has the same definition as in Table 8.2, as are any features not discussed here. The modifications to the old instructions deal with four aspects:

1. Jumps and conditional control flow.

2. Memory access restrictions.
Chapter 11. Extension of Verification Condition Generator

3. Annotations.

4. Function calls.

The first two instructions — jump and condC(A(r), n) — add a check so that if they can transfer execution forward, they always do so to a new type of code annotation — the assertion. This ensures that any forward or backward transfer of control within a function is to an invariant or assertion. This is to ensure that the extended verification condition generator can tell whether the instruction is executed with the array fresh — gacnt ≠ 0 — or stale — gacnt = 0 — for any potential entry point of a loop or destination of a jump. This is necessary because many of the new instructions have two cases in the extended verification condition generator depending on the value of gacnt. The cases with fresh array execution contexts will be seen to be much more complex than the stale cases. Thus it is important to have the two cases to minimize the complexity of the verification condition generator.

The inv instruction is modified to require that it implies that the array execution context is either always fresh or always stale at the location. Similarly, the asrt instruction is added as a new type of annotation to assert that the array execution context is always fresh or always stale at the location.

The memory access instructions r ← M[r'] and M[r'] ← r are modified so that the verification condition now includes a clause stating that it is permissible for the processor to access the memory address according to the current state of the gamed register.

The call instruction requires that the precondition of the function — PreG — must imply that the gacnt register has a constant type of value, 0 or not 0. It is necessary for the verification condition generator to know whether to start evaluating a function with the array possibly running. The program precondition must also satisfy this requirement. The table does not reflect this because any instruction

97
could be the entry point to a program, so a special case is required in the verification condition generator. The verification condition generator must also know the state of the array after the function returns, so its postcondition — Post<sub>G</sub> — must imply either that the array is always fresh or always stale.

Another very subtle change to the call instruction is not reflected in the table. The Regs<sub>G</sub> expression is extended to cover all the new array specific registers. This means that they are assigned new variables and quantified over in the verification condition for call instructions unless they are declared callee save registers. If they are declared callee save registers, then they must have the same value when the next instruction is executed after the function returns.
<table>
<thead>
<tr>
<th>$F_1$</th>
<th>Check</th>
<th>Verification Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>call $G$</td>
<td>$G \in \Phi$</td>
<td>$\sigma(ra) \equiv \text{offset}(i_0, i + _)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$i + _ \in \text{Dom}(F)$</td>
</tr>
<tr>
<td></td>
<td>so $\leq Arg$</td>
<td>$\text{Pre}_G \supset gact = 0$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{Post}_G \supset gact = 0$</td>
</tr>
<tr>
<td></td>
<td>$\sigma_1'(\text{Pre}<em>C) \land \forall y_1 \ldots y_k, z</em>{m+1} \ldots z_{\text{Local}_p}$</td>
<td>$\sigma_2'(\text{Post}_C) \sqsubseteq SE(i + _, \sigma_2', so, \mathcal{L})$</td>
</tr>
<tr>
<td></td>
<td>where:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\sigma_1' = \text{CopyIn}(\sigma_1, Arg, so)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>${r_1, \ldots, r_{k}} = \text{Reg}_{SC} - CS_G$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$y_1, \ldots, y_k, z_{m+1}, \ldots, z_{\text{Local}_p}$ are new variables</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\sigma_2' = \sigma_1'[r_1 \mapsto y_1, \ldots, r_k \mapsto y_k]$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\sigma_2' = \text{CopyOut}(\sigma_2', \sigma_1', Arg, so, {z_{m+1}, \ldots, z_{\text{Local}_p}})$</td>
<td></td>
</tr>
</tbody>
</table>

Table 11.2: Modified Verification Condition Generator cont.

### 11.2 Important Aspects of Extending the Verification Condition Generator

There are four important aspects of the definitions which must be discussed:

1. The $\epsilon$ function.
2. Fresh and stale reconfigurable array accessible variables.
3. Universal quantification instead of a $\kappa$ function.
Chapter 11. Extension of Verification Condition Generator

The $\epsilon$ function for symbolic evaluation states is nearly identical to the one for SAL states:

$$
\epsilon(\sigma) = \sigma[\text{gar}x_{n,m} \mapsto \epsilon(x_{n,m})^v.true, \ldots, \text{mem} \mapsto \sigma(\text{mem})[\sigma(w) \mapsto \epsilon(w)^v.true]]
$$

(11.1)

for all $x, n, m$, and $w$ such that $\neg \sigma(\text{gamem})w$. The only difference is that it refers to states of the symbolic evaluator and updates on them instead of SAL states.

While the verification condition generator uses $\epsilon_a$ terms, it is important to note that the logic is not extended to deal with them. This is because such terms are interpreted as being unknown and inaccessible, and therefore should not be used in any proof of the verification condition. This is ensured because the verification conditions for each instruction do not make $\epsilon_a$ terms appear outside of recursive calls to the symbolic evaluator. For example, when the array is fresh, all $\epsilon_a$ terms disappear because the registers containing them are assigned new, universally quantified variables. The rules presented are only used in reasoning about the system, not for reasoning in it.

One very important aspect of the definitions is instructions that are the first to access the array after an execution. This will be called fresh as opposed to stale, as introduced in Section 9.2. When the array access is fresh, the verification condition generator must only assume that the array registers and array accessible memory contain values consistent with the configuration postcondition and memory partition. To ensure this, the verification condition generator will have two cases covering when the $\text{gacnt}$ register is 0 and when it is not, for all the instructions except $\text{gabump}$.

The fresh case will always use the same method to ensure that no unwarranted assumptions are made about the values in the array registers or memory. Before carrying out the state modification corresponding to the routine semantics of the instruction, an intermediate state is created. This state creates a new variable to represent the values in each of the array registers and one for the memory which have
Chapter 11. Extension of Verification Condition Generator

until now been accessible to the reconfigurable array only or partly. These variables will all be universally quantified so that the proof of the verification condition will have to account for all possible values in these variables. This is in contrast to SAL which used the $\kappa$ function to fill the corresponding variables with arbitrary values. Because $\kappa$ terms have very weak deductive rules, they would prevent the formation of a proof in this context. The gacnt register will be zeroed and the gamem register will be updated to map all values to 0, because the array must halt before the instruction can execute and then the processor will have access to all the memory.

The universal quantification of the array register and memory register variables in the verification condition corresponds to the arbitrary nature of these values in SAL which are expressed by $\kappa$ terms. Because there are no rules in the program logic to reason about $\kappa$ terms, and any given would be insufficient, universal quantification is used to reason about arbitrary values by reasoning about all possible values.

The configuration postcondition and memory partition may give some information about the array register and memory values, so they are not completely arbitrary. Therefore the formula inside the quantification will be an implication. The antecedent of this implication will be called the restriction clause. It will consist of a clause stating that all memory positions which are assigned to the processor in the current state will retain their values, and a clause stating that the postcondition is satisfied in the intermediate state. The consequent of the implication will be called the condition. It will consist of a symbolic evaluation of the next instruction in a further updated state to reflect the routine semantics of the instruction. It may also contain a clause stating that the array configuration precondition is satisfied if the instruction starts an array execution.

The purpose of the restriction clause is to limit the extent of the proof for the universal quantification. Any assignment of values to variables which violates the restriction clause does not have to satisfy the condition clause. This is essentially the
same method that Necula uses for function calls which can be seen in the verification condition generator for the original PCC system in Table 8.2. There are some differences due to the concurrent execution of the array and processor. The array can begin execution at one instruction and complete at another, so many instructions have cases which account for the completion of array execution.

The definitions refer to three specifications in the contract for a configuration:

1. The precondition — $\text{Pre}_A$.
2. The postcondition — $\text{Post}_A$.
3. The memory partition — $\text{Mem}_A$.

These specifications are constant for any configuration, just as the contract for any given function call is constant. However, they can refer to variable quantities which must be evaluated in a particular state. These variables must be restricted depending on the type of a specification.

The precondition specifies any initial values for the array variables, array counter, and memory that are needed to ensure the safe execution of the configuration. The postcondition specifies the values that can be safely assumed to hold in the array registers and memory after completion of an execution of the configuration. Both the precondition and postcondition can be any logical formulas that evaluate to assertions in a state except that they must depend only on variables which are available to the reconfigurable array during an execution. This is because these specifications are used in both the verification condition generator and for the verification of the reconfigurable array. Thus they may only refer to array register variables, accessible sections of memory, and the count with which the array was started.

The memory partition specifies which memory locations are available to the array during execution. It is also a logical formula which evaluates to an assertion in a
state. The memory partition must depend only on variables which are visible to
the reconfigurable array during execution, and not on any memory location which
can change during array execution. This is because this specification must be true
throughout the entire array execution. If the memory partition can be changed by
writing to memory then it would be impossible for the verification condition generator
to assume it is constant. Because it is possible for the verification condition generator
and the array to keep track of the starting values of the counter and array registers, it
does not matter that they can be changed. This is not feasible with memory because
it would require keeping track of all array accessible memory at the start of array
execution just in case it is used by the memory partition.

11.3 Extensions for New Instructions

The extensions to the verification condition generator for SAL can be seen in Tables
11.3 and 11.4.

The first instruction — gaconf — is for loading configurations into the array. It
has a check for the state of array execution. If the array execution is stale, the state is
updated with a new configuration and zeroed array registers. The argument register
is required to point to a valid configuration and the symbolic evaluator is recursively
called on the next instruction in the new state. When the array execution is fresh,
the verification condition is the conjunction of the validity of the argument as a
configuration, and the universal quantification with respect to the array registers and
memory variables of an implication composed of a restriction clause and a condition
clause.

The restriction clause is the conjunction of a predicate stating that for a new mem-

ory state variable, none of the processor accessible memory locations are changed,
and the postcondition for the current array configuration evaluated at the state where
### Chapter 11. Extension of Verification Condition Generator

<table>
<thead>
<tr>
<th>$F_1$</th>
<th>Check</th>
<th>Verification Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>gaconf</td>
<td>$i + + \in \text{Dom}(F)$, $\sigma(\text{gact}) = 0$</td>
<td>$\sigma(r) \in \Gamma \land \text{SE}(i + +, \sigma', \text{so}, \mathcal{L})$&lt;br&gt;where: $\sigma' = \sigma[r \mapsto r, \text{gact}, i + 0 \ldots \text{gard}_{N, 3} \mapsto 0]$</td>
</tr>
</tbody>
</table>
| gaconf | $i + + \in \text{Dom}(F)$, $\sigma(\text{gact}) \neq 0$ | $\sigma(r) \in \Gamma \land \forall y_1 \ldots y_i. z. ((\lnot \text{gmem}) u \supset z(u) = \sigma(\text{mem})(u))$

\[\land \sigma'(\text{Post}_{r_{\text{gact}}}(\text{gmem})) \supset \text{SE}(i + +, \sigma'', \text{so}, \mathcal{L})\]

where: $\sigma' = \sigma[r_{\text{gact}} \mapsto r_{\text{gmem}} \mapsto y_i \ldots y_i \mapsto z, \text{gact} \mapsto 0 \ldots \text{gard}_{N, 3} \mapsto 0]$ for all $k$ and $y_i \ldots y_i \mapsto z$ new variables

$\sigma'' = \sigma[r \mapsto r, \text{gact}, i + 0 \ldots \text{gard}_{N, 3} \mapsto 0]$ |
| gareset | $i + + \in \text{Dom}(F)$, $\sigma(\text{gact}) = 0$ | $\text{SE}(i + +, \sigma[r \mapsto 0], \text{so}, \mathcal{L})$ |
| gareset | $i + + \in \text{Dom}(F)$, $\sigma(\text{gact}) \neq 0$ | $\forall y_1 \ldots y_i. z. ((\lnot \text{gmem}) u \supset z(u) = \sigma(\text{mem})(u))$

\[\land \sigma'(\text{Post}_{r_{\text{gact}}}(\text{gmem})) \supset \text{SE}(i + +, \sigma'', \text{so}, \mathcal{L})\]

where: $\sigma' = \sigma[r_{\text{gact}} \mapsto r_{\text{gmem}} \mapsto y_i \ldots y_i \mapsto z, \text{gact} \mapsto 0 \ldots \text{gard}_{N, 3} \mapsto 0]$ for all $k$ and $y_i \ldots y_i \mapsto z$ new variables

$\sigma'' = \sigma[r \mapsto r, \text{gact}, i + 0 \ldots \text{gard}_{N, 3} \mapsto 0]$ |
| gactmp | $i + + \in \text{Dom}(F)$, $\sigma(\text{gact}) = 0$ | $\sigma(\text{gact}) \neq 0 \land \sigma'(\text{Pre}_{r_{\text{gact}}}(\text{gmem})) \land \text{SE}(i + +, \epsilon(\sigma'), \text{so}, \mathcal{L})$

where: $\sigma' = \sigma[r_{\text{gact}} \mapsto r_{\text{gmem}} \mapsto \gamma(\sigma(\text{gmem}), 1), \text{so}, \mathcal{L})$ |
| mt ga r, x, n | $i + + \in \text{Dom}(F)$, $\sigma(\text{gact}) = 0$<br>$n < \text{arraysize}$ | $\forall y_1 \ldots y_i. z. ((\lnot \text{gmem}) u \supset z(u) = \sigma(\text{mem})(u))$

\[\land \sigma'(\text{Post}_{r_{\text{gact}}}(\text{gmem})) \supset \text{SE}(i + +, \sigma'', \text{so}, \mathcal{L})\]

where: $\sigma' = \sigma[r_{\text{gact}} \mapsto r_{\text{gmem}} \mapsto y_i \ldots y_i \mapsto z, \text{gact} \mapsto 0 \ldots \text{gard}_{N, 3} \mapsto 0]$ for all $k$ and $y_i \ldots y_i \mapsto z$ new variables

$\sigma'' = \sigma[r_{\text{gact}} \mapsto r_{\text{gmem}} \mapsto \gamma(\sigma(\text{gmem}), 1), \text{so}, \mathcal{L})$ |
| mt ga r, x, n | $i + + \in \text{Dom}(F)$, $\sigma(\text{gact}) \neq 0$<br>$n < \text{arraysize}$ | $\sigma'(\text{Pre}_{r_{\text{gact}}}(\text{gmem})) \land \text{SE}(i + +, \epsilon(\sigma'), \text{so}, \mathcal{L})$

where: $\sigma' = \sigma[r_{\text{gact}} \mapsto r_{\text{gmem}} \mapsto \gamma(\sigma(\text{gmem}), 1), \text{so}, \mathcal{L})$<br>$\text{gact} \mapsto c, \text{gmem} \mapsto \gamma(\text{Mem}_{\text{gact}})$ |
| mt ga r, x, n, c | $i + + \in \text{Dom}(F)$, $\sigma(\text{gact}) = 0$<br>$n < \text{arraysize}$ | $\forall y_1 \ldots y_i. z. ((\lnot \text{gmem}) u \supset z(u) = \sigma(\text{mem})(u))$

\[\land \sigma'(\text{Pre}_{r_{\text{gact}}}(\text{gmem})) \land \text{SE}(i + +, \epsilon(\sigma'), \text{so}, \mathcal{L})\]

where: $\sigma' = \sigma[r_{\text{gact}} \mapsto r_{\text{gmem}} \mapsto y_i \ldots y_i \mapsto z, \text{gact} \mapsto 0 \ldots \text{gard}_{N, 3} \mapsto 0]$ for all $k$ and $y_i \ldots y_i \mapsto z$ new variables

$\sigma'' = \sigma[r_{\text{gact}} \mapsto r_{\text{gmem}} \mapsto \gamma(\sigma(\text{gmem}), 1), \text{so}, \mathcal{L})$<br>$\text{gact} \mapsto c, \text{gmem} \mapsto \gamma(\text{Mem}_{\text{gact}})$ |
| mt ga r, x, n | $i + + \in \text{Dom}(F)$, $\sigma(\text{gact}) = 0$<br>$n < \text{arraysize}$ | $\forall y_1 \ldots y_i. z. ((\lnot \text{gmem}) u \supset z(u) = \sigma(\text{mem})(u))$

\[\land \sigma'(\text{Post}_{r_{\text{gact}}}(\text{gmem})) \supset \text{SE}(i + +, \epsilon(\sigma'), \text{so}, \mathcal{L})\]

where: $\sigma' = \sigma[r_{\text{gact}} \mapsto r_{\text{gmem}} \mapsto y_i \ldots y_i \mapsto z, \text{gact} \mapsto 0 \ldots \text{gard}_{N, 3} \mapsto 0]$ for all $k$ and $y_i \ldots y_i \mapsto z$ new variables

$\sigma'' = \sigma[r_{\text{gact}} \mapsto r_{\text{gmem}} \mapsto \gamma(\sigma(\text{gmem}), 1), \text{so}, \mathcal{L})$<br>$\text{gact} \mapsto c, \text{gmem} \mapsto \gamma(\text{Mem}_{\text{gact}})$ |

| mt ga r, x, n | $i + + \in \text{Dom}(F)$, $\sigma(\text{gact}) \neq 0$<br>$n < \text{arraysize}$ | $\forall y_1 \ldots y_i. z. ((\lnot \text{gmem}) u \supset z(u) = \sigma(\text{mem})(u))$

\[\land \sigma'(\text{Pre}_{r_{\text{gact}}}(\text{gmem})) \land \text{SE}(i + +, \epsilon(\sigma'), \text{so}, \mathcal{L})\]

where: $\sigma' = \sigma[r_{\text{gact}} \mapsto r_{\text{gmem}} \mapsto y_i \ldots y_i \mapsto z, \text{gact} \mapsto 0 \ldots \text{gard}_{N, 3} \mapsto 0]$ for all $k$ and $y_i \ldots y_i \mapsto z$ new variables

$\sigma'' = \sigma[r_{\text{gact}} \mapsto r_{\text{gmem}} \mapsto \gamma(\sigma(\text{gmem}), 1), \text{so}, \mathcal{L})$<br>$\text{gact} \mapsto c, \text{gmem} \mapsto \gamma(\text{Mem}_{\text{gact}})$ |
| mt ga r, x, n, c | $i + + \in \text{Dom}(F)$, $\sigma(\text{gact}) = 0$<br>$n < \text{arraysize}$ | $\forall y_1 \ldots y_i. z. ((\lnot \text{gmem}) u \supset z(u) = \sigma(\text{mem})(u))$

\[\land \sigma'(\text{Pre}_{r_{\text{gact}}}(\text{gmem})) \land \text{SE}(i + +, \epsilon(\sigma'), \text{so}, \mathcal{L})\]

where: $\sigma' = \sigma[r_{\text{gact}} \mapsto r_{\text{gmem}} \mapsto y_i \ldots y_i \mapsto z, \text{gact} \mapsto 0 \ldots \text{gard}_{N, 3} \mapsto 0]$ for all $k$ and $y_i \ldots y_i \mapsto z$ new variables

$\sigma'' = \sigma[r_{\text{gact}} \mapsto r_{\text{gmem}} \mapsto \gamma(\sigma(\text{gmem}), 1), \text{so}, \mathcal{L})$<br>$\text{gact} \mapsto c, \text{gmem} \mapsto \gamma(\text{Mem}_{\text{gact}})$ |

Table 11.3: Extended Verification Condition Generator

all the array registers and memory are assigned new variables, the count is zeroed, and all memory freed for processor access. The symbolic evaluation of the next instruction in a new state is the condition clause. This new state is the same as the
<table>
<thead>
<tr>
<th>( F_i )</th>
<th>Check</th>
<th>Verification Condition</th>
</tr>
</thead>
</table>
| mtgavr, s | \( i + + \in \text{Dom}(F) \)  
\( \sigma(s) \equiv (n : x) \)  
\( \sigma(\text{gacnt}) = 0 \) | \( n < \text{arraysize} \wedge \)  
\( SE(i + +, \sigma(\text{gazx}, 1) \mapsto \sigma(r), \text{gaox} \mapsto \gamma(\sigma(\text{gaox}, 1), 1)), \sigma, \mathcal{L} \) |
| mtgavr, s | \( i + + \in \text{Dom}(F) \)  
\( \sigma(s) \equiv (n : x) \)  
\( \sigma(\text{gacnt}) \neq 0 \) | \( n < \text{arraysize} \wedge \)  
\( \forall y_1 \ldots y_{i + +}, z : ((-\sigma(\text{gmem})) \cup z(u) = \sigma(\text{mem})(u)) \wedge \sigma'(\text{Post}(\text{gavz})) \supset \)  
\( SE(i + +, \sigma'', \mathcal{L}) \)  
where:  
\( \sigma' = \sigma[\text{gazx}_{0, 1} \mapsto y_1 \ldots \text{gazx}_{0, 3} \mapsto y_{i + +}, \text{mem} \mapsto z], \)  
\( \text{gacnt} \mapsto 0, (\text{gmem})k \mapsto 0 \) for all \( k \) and \( y_1 \ldots y_{i}, z \) new variables  
\( \sigma'' = \sigma'[r \mapsto \delta_{\text{gacnt}, \sigma}, \gamma(\text{gaox}, 1), \sigma, \mathcal{L}] \) |
| mfgavr, s | \( i + + \in \text{Dom}(F) \)  
\( \sigma(s) \equiv (n : x) \)  
\( \sigma(\text{gacnt}) = 0 \) | \( n < \text{arraysize} \wedge \)  
\( \forall y_1 \ldots y_{i + +}, z : ((-\sigma(\text{gmem})) \cup z(u) = \sigma(\text{mem})(u)) \wedge \sigma'(\text{Post}(\text{gavz})) \supset \)  
\( SE(i + +, \sigma'', \mathcal{L}) \)  
where:  
\( \sigma' = \sigma[\text{gazx}_{0, 1} \mapsto y_1 \ldots \text{gazx}_{0, 3} \mapsto y_{i + +}, \text{mem} \mapsto z], \)  
\( \text{gacnt} \mapsto 0, (\text{gmem})k \mapsto 0 \) for all \( k \) and \( y_1 \ldots y_{i}, z \) new variables  
\( \sigma'' = \sigma'[r \mapsto \delta_{\text{gacnt}, \sigma}, \gamma(\text{gaox}, 1), \sigma, \mathcal{L}] \) |
| mfgavry, s | \( i + + \in \text{Dom}(F) \)  
\( \sigma(s) \equiv (n : x) \)  
\( \sigma(\text{gacnt}) = 0 \) | \( n < \text{arraysize} \wedge \)  
\( \forall y_1 \ldots y_{i + +}, z : ((-\sigma(\text{gmem})) \cup z(u) = \sigma(\text{mem})(u)) \wedge \sigma'(\text{Post}(\text{gavz})) \supset \)  
\( SE(i + +, \sigma'', \mathcal{L}) \)  
where:  
\( \sigma' = \sigma[\text{gazx}_{0, 1} \mapsto y_1 \ldots \text{gazx}_{0, 3} \mapsto y_{i + +}, \text{mem} \mapsto z], \)  
\( \text{gacnt} \mapsto 0, (\text{gmem})k \mapsto 0 \) for all \( k \) and \( y_1 \ldots y_{i}, z \) new variables  
\( \sigma'' = \sigma'[r \mapsto \delta_{\text{gacnt}, \sigma}, \gamma(\text{gaox}, 1), \sigma, \mathcal{L}] \) |
| mfgavry, s | \( i + + \in \text{Dom}(F) \)  
\( \sigma(s) \equiv (n : x) \)  
\( \sigma(\text{gacnt}) = 0 \) | \( n < \text{arraysize} \wedge \)  
\( \forall y_1 \ldots y_{i + +}, z : ((-\sigma(\text{gmem})) \cup z(u) = \sigma(\text{mem})(u)) \wedge \sigma'(\text{Post}(\text{gavz})) \supset \)  
\( SE(i + +, \sigma'', \mathcal{L}) \)  
where:  
\( \sigma' = \sigma[\text{gazx}_{0, 1} \mapsto y_1 \ldots \text{gazx}_{0, 3} \mapsto y_{i + +}, \text{mem} \mapsto z], \)  
\( \text{gacnt} \mapsto 0, (\text{gmem})k \mapsto 0 \) for all \( k \) and \( y_1 \ldots y_{i}, z \) new variables  
\( \sigma'' = \sigma'[r \mapsto \delta_{\text{gacnt}, \sigma}, \gamma(\text{gaox}, 1), \sigma, \mathcal{L}] \) |
| mfgavz, s | \( i + + \in \text{Dom}(F) \)  
\( \sigma(s) \equiv (n : x) \)  
\( \sigma(\text{gacnt}) = 0 \) | \( n < \text{arraysize} \wedge \)  
\( \forall y_1 \ldots y_{i + +}, z : ((-\sigma(\text{gmem})) \cup z(u) = \sigma(\text{mem})(u)) \wedge \sigma'(\text{Post}(\text{gavz})) \supset \)  
\( SE(i + +, \sigma'', \mathcal{L}) \)  
where:  
\( \sigma' = \sigma[\text{gazx}_{0, 1} \mapsto y_1 \ldots \text{gazx}_{0, 3} \mapsto y_{i + +}, \text{mem} \mapsto z], \)  
\( \text{gacnt} \mapsto 0, (\text{gmem})k \mapsto 0 \) for all \( k \) and \( y_1 \ldots y_{i}, z \) new variables  
\( \sigma'' = \sigma'[r \mapsto \delta_{\text{gacnt}, \sigma}, \gamma(\text{gaox}, 1), \sigma, \mathcal{L}] \) |
| mfgavz, s | \( i + + \in \text{Dom}(F) \)  
\( \sigma(s) \equiv (n : x) \)  
\( \sigma(\text{gacnt}) = 0 \) | \( n < \text{arraysize} \wedge \)  
\( \forall y_1 \ldots y_{i + +}, z : ((-\sigma(\text{gmem})) \cup z(u) = \sigma(\text{mem})(u)) \wedge \sigma'(\text{Post}(\text{gavz})) \supset \)  
\( SE(i + +, \sigma'', \mathcal{L}) \)  
where:  
\( \sigma' = \sigma[\text{gazx}_{0, 1} \mapsto y_1 \ldots \text{gazx}_{0, 3} \mapsto y_{i + +}, \text{mem} \mapsto z], \)  
\( \text{gacnt} \mapsto 0, (\text{gmem})k \mapsto 0 \) for all \( k \) and \( y_1 \ldots y_{i}, z \) new variables  
\( \sigma'' = \sigma'[r \mapsto \delta_{\text{gacnt}, \sigma}, \gamma(\text{gaox}, 1), \sigma, \mathcal{L}] \) |

Table 11.4: Extended Verification Condition Generator cont.
previous one but with the new configuration and the array registers zeroed. There is no quantification over possible values of the array register because they are zeroed by the instruction before they can be accessed.

The **garest** instruction simply zeros the configuration register and recursively calls the symbolic evaluator on the next instruction in the stale case. In the fresh case, it creates new variables for all the array registers and the memory, and a new state where the array registers and memory are assigned these variables, the configuration and counter registers are zeroed, and all memory is freed for the processor. There is a universal quantification with respect to the new variables over an implication with a restriction clause antecedent and a condition clause consequent. These clauses are the same as the previous instruction except the array registers are not zeroed in the resulting state.

The **gabump** instruction requires that the array not be executing because it increases the array counter and would cause an indeterminate execution if it were used during array execution. Because it starts the array execution, it evaluates as the conjunction of an assertion that there is a loaded configuration, the configuration precondition evaluated with the new count and memory partition, and the symbolic evaluation of the next instruction. The state for the symbolic evaluation is further updated using the $\epsilon$ function to put unknown values in all the array accessible variables.

The rest of the instructions, those starting with **m**, move data to and from the array registers. These instructions are similar, but differ in four ways:

1. Constant vs. variable array register row and type arguments.

2. Moving data to the array vs. moving data from the array.

3. Starting vs. not starting array execution.
4. High vs. middle vs. low array register access.

In addition there are separate cases for when $gacnt$ is 0 and when it is not.

The array data access instructions that take a variable row and type argument occupy the entirety of Table 11.4. These instructions can be recognized by the \textit{v} in their names — $mtgav$, $mfgav$, $mtgavy$, etc. These instructions have an \textit{s} argument which specifies the row they access in the bits above the least significant and the Z or D registers in the least significant bit. They have a check which defines two variables, \textit{n} and \textit{x}, from the \textit{s} variable which represent the row and type arguments respectively. The \textit{x} argument translates in the definitions to \textit{z} if it equals 0 and \textit{d} if it equals 1. The verification condition for these instructions starts with a clause stating that they \textit{n} variable is less than the size of the array to ensure that the row argument is for a valid row. The rest of the verification condition depends on the other properties of the instruction.

The constant data access instructions have the row and type encoded in the instruction so they are slightly simpler. The \textit{n} and \textit{x} arguments shown in Table 11.3 for data access instructions are part of the instruction so the \textit{n} argument can be statically checked to see if it is valid instead of requiring a clause in the verification condition. The \textit{x} argument is interpreted here as in the variable argument case.

The most significant difference in the definitions is in the cases for stale and fresh access to the array. In the stale case the verification condition contains a symbolic evaluation of the next instruction in a new state. This evaluation is not quantified or restricted. The new state depends on other aspects of the instruction.

In the fresh case there is a quantification and the usual implication with a restriction clause and condition clause. There is also a two step updating of the state. The first update reflects the state which holds when the array stops, and the second when the instruction is completed. This is needed because the postcondition for the
array configuration must hold in the state when it halts but not after the instruction is completed. The universal quantification is with respect to the array registers and array accessible memory. The restriction is that the new memory pseudo-register respects the memory partition, and the postcondition of the completed array configuration is true in the state after the first update. This update sets all the array registers and the memory pseudo-register to new variables, sets the count to 0, and all the locations in the memory partition to 0. The fully updated state depends on other aspects of the instruction, but it is always used for the recursive call to the symbolic evaluator in the condition clause.

Two data access instructions can also start the array after the access. These are the \texttt{mtga} \(r, x, n, c\) and \texttt{mfga} \(r, x, n, c\) instructions. Where all the other data access instructions simply have a recursive call to the symbolic evaluator, these instructions have the conjunction of the precondition of the still active configuration evaluated in the state after the data access, and the symbolic evaluation. The updated state also includes an update to the array count, memory partition variables, and an application of the \(\epsilon\) function.

The direction of movement of data, to or from the array, is reflected in the new state. For data moving to the array registers, instructions containing a \texttt{t} — \texttt{mtga}, \texttt{mtgav}, \texttt{mtgavy}, \texttt{mtgavz}; the update is simply to an array register variable and its corresponding array register order variable. For data moving from the array registers, instructions containing an \texttt{f} — \texttt{mfga}, \texttt{mfgav}, \texttt{mfgavy}, \texttt{mfgavz}; the update is a little more complex. The \(\delta\) function of Figure 10.3 is used to determine the actual value to be transferred from the values of the array registers and the order they were updated.

The section of the array register which is accessed is indicated in the instruction name by a \(y\) or \(z\) or the absence of either of these. Thus \texttt{mtga}, \texttt{mfga}, \texttt{mtgav}, and \texttt{mfgav}, access the middle bits; \texttt{mtgavy} and \texttt{mfgavy} access the lower bits; and \texttt{mtgavz} and \texttt{mfgavz} access the higher bits. The verification condition reflects these differences.
with which array register variable is updated or read, and by how the order variable is updated for writes. The order update depends on the order in the current state and the section accessed for stale accesses but is simply assigned a constant depending on the section accessed for the fresh case. This is because the order register is no longer after array execution and can be somewhat arbitrary. If this could cause a problem it is up to the array execution postcondition to alleviate it.

11.4 Argument for Soundness of Modification

As it stands there is no reason to be convinced that the above modifications to the PCC system are sound, that the provability of a verification condition implies that the associated SAL program is safe. Because SAL is used to define safety properties along with the program logic, any failure of soundness would allow potential violations of the safety properties. Conversely, soundness implies that a proof of the verification condition rules out any unsafe program behavior modulo the program logic, consumer provided functions, and reconfigurable array configuration safety. Thus the soundness of the verification condition generator can only guarantee safety under the assumption that the rest of the variable components of the system are correct with regard to safety.

The argument for soundness will be that Necula’s argument for the old instructions is still valid except for very minor modifications, and that the new instructions require only a small extension to the argument. Unfortunately, Necula’s argument that his original verification condition generator is sound is quite complex and cannot be fully reproduced here. The full arguments for all the new instructions would also be very long as well as quite repetitious. Thus only a sketch of the full argument will be given.

Necula’s proof for soundness works by proving that if the verification condition
Chapter 11. Extension of Verification Condition Generator

IH1 The verification condition is valid, \( \models \tau(SE(i, \sigma, s_0, L)) \)

IH2 \( s_0 \) is correct, \( 0 \leq s_0 \leq \text{Local}_F \) and \( \rho(s_p) + s_0 = \rho_0(s_p) + \text{Arg}_F \)

IH3 \( i_0 \) is correct, \( < F, 0 > = \tau(i_0) \)

IH4 \( \sigma \) and \( \sigma_0 \) are correct, \( \models \tau(\sigma(r)) = \rho(r) \) and \( \tau(\sigma_0(r)) = \rho_0(r) \)

IH5 The stack is preserved, \( \text{Stack}(a) \) and \( a \geq \rho(s_p) + \text{Arg}_F \) implies \( \rho_0(\text{mem})(a) = \rho(\text{mem})(a) \)

IH6 The frame fits in the stack, \( \rho_0(s_p) + \text{Arg}_F - \text{Local}_F \leq a \leq \rho_0(s_p) + \text{Arg}_F - 1 \) implies \( \text{Stack}(a) \)

IH7 \( \tau \) is correct with respect to \( L \), either

IH7.1 \( L \) is empty and for all \( r \in \text{Regs}_F \)
  IH7.1.1 \( \sigma_0(r) \in \text{Dom}(\tau) \)
  IH7.1.2 \( \models \tau(\sigma_0(r)) = \rho_0(r) \)

IH7.2 \( L = L_1 + (i, \sigma'_1) \) with \( \sigma'_1 = \sigma_1[r_1 \mapsto y_1, \ldots, r_k \mapsto y_k] \) and
  IH7.2.1 \( F_i = \text{inv}P, n, r_1, \ldots, r_k \)
  IH7.2.2 \( y_i \) are new variables, \( y_j \notin (\text{Dom}(\tau_1) \cup \text{FV}(L_1 \cup \text{FV}(\sigma_1))) \)
  IH7.2.3 \( \tau = \tau_1 + [y_1 \mapsto v_1, \ldots, y_k \mapsto v_k] \)
  IH7.2.4 \( \models \tau_1(\sigma_1(P)) \)
  IH7.2.5 \( \models \tau_1(\forall y_1 \ldots y_k.\sigma'_1(P) \supset SE(i +, \sigma', n, L)) \)
  IH7.2.6 \( \tau_1 \) is correct with respect to \( L_1 \) as defined by IH7

Figure 11.1: Full Induction Hypothesis

is valid for any reachable instruction in a function, then that function either returns safely or progresses to a new safe state. This is done by induction on the length of the execution with a rather long induction hypothesis seen in Figure 11.1. In the induction hypothesis, \( \tau \) refers to a substitution of values for the variables in the symbolic execution state so that \( \tau(\sigma) = \rho \). Building this substitution during the induction argument is the cornerstone of the soundness argument.

Clause IH1 states that the \( \tau \) substitution applied to the verification condition
Chapter 11. Extension of Verification Condition Generator

for the instruction as found in the definition of the verification condition generator is a valid formula of the program logic. Clause **IH2** states that the stack offset points to the section of the stack following the stack passed arguments for the function. Clause **IH3** states that the initial value of the instruction counter is correct with respect to the current $\tau$ substitution. Clause **IH4** states that the current and initial register states for the verification condition generator are correct with respect to the current $\tau$ substitution, so that any register value in these states is equal to the value in the same register during the SAL execution after the substitution. Clause **IH5** states that the stack section for other functions and the stack passed arguments is not changed. Clause **IH6** states that there is room on the frame for the stack. Clause **IH7** has several parts but it simply states that the loop map is correct and that for every pair in it the value of $\tau$ at that point was correctly built from the value of $\tau$ in the previous pair.

The inductive argument is a case analysis on the current instruction. While the whole argument for soundness is too lengthy to reproduce here, the argument for memory writes — $M[r'] \mapsto r$ — will be given as an example.

The safety condition $SafeW r(\rho(\text{mem}), \rho(r'), \rho(r))$ follows from the validity of the verification condition $\tau(safewr(\sigma(\text{mem}), \sigma(r'), \sigma(r)))$ which follows from **IH1**, the correctness of $\sigma$ which follows from **IH4**, and the correctness of the $safewr$ definition. This means that the execution can make progress at this instruction.

The induction hypothesis must be proven with $i$ incremented, $\sigma'$ equal to $\sigma$ with \text{mem} updated to $\text{upd} (\sigma(\text{mem}), \sigma(r'), \sigma(r))$, and $\rho'$ equal to $\rho$ with \text{mem} updated to $\rho(\text{mem}[\rho(r') \mapsto \rho(r)])$. Clause **IH1** follows directly from clause **IH1** of the induction hypothesis because it is one of the conjuncts of the verification condition for memory writes. Clause **IH5** follows because the stack memory and random access memory are disjoint, and from clause **IH5** of the induction hypothesis. Clause **IH4** requires that $\models \text{upd}(\tau(\sigma(\text{mem})), \tau(\sigma(r')), \tau(\sigma(r))) = \rho(\text{mem}[\rho(r') \mapsto \rho(r)])$. This follows from
the valuation of upd. All other clauses follow trivially from the induction hypothesis.

For the argument for the soundness of the new verification condition generator, the memory access instructions have a modification to prove that they result in a safe state. These now have an additional safety requirement — $\neg \rho(gamem) \rho(r')$. This is easily proven from the verification condition $\neg \sigma(gamem) \sigma(r')$ and the $\tau$ substitution from the induction hypothesis. $\neg \tau(\sigma(gamem)) \tau(r')$ implies $\neg \rho(gamem) \rho(r')$ because the hypothesis states that $\tau(\sigma(gamem)) = \rho(gamem)$ and $\tau(r') = \rho(r')$.

The arguments for the continued correctness of $\tau$ are unaffected for ordinary values. There is one additional consideration for the affect of $\tau$ on $\epsilon_\alpha$ terms. Because the instructions which start the array execution put $\epsilon$ terms in array accessible variables, and the assurance from the verification condition generator that any instruction instance occurs with either the array definitely halted or possibly executing, it is always the case that any $\epsilon_\alpha$ terms in $\sigma$ can be mapped to themselves in $\rho$ because $\epsilon_\alpha$ terms with the same $\alpha$ and predicate are always equivalent. Thus there is no required modification or addition to $\tau$ because no substitution is necessary for $\epsilon_\alpha$ terms.

The soundness of the verification condition generator follows from the old arguments in all other cases because the instruction’s verification condition does not refer to either array accessible registers or memory.

### 11.5 Argument for Soundness of Extension

Fortunately, the induction hypothesis can be shortened considerably for the extension of the argument to the new instructions. Because the new instructions are not concerned with the stack, function calls, or loops, five out of seven clauses of the induction hypothesis can be ignored as trivial for the extended portions of SAL. The effective induction hypothesis will be:
Chapter 11. Extension of Verification Condition Generator

1. $\models \tau(SE(i, \sigma, s0, \mathcal{L}))$ — the verification condition is valid.

2. $\models \tau(\sigma(r)) = \rho(r)$ and $\tau(\sigma_0(r)) = \rho_0(r)$ — $\sigma$ and $\sigma_0$ are correct.

In the above formulas $\tau$ refers to a substitution of values for the variables in the symbolic execution state so that $\tau(\sigma) = \rho$.

The soundness of the new instructions can be similarly proven by case analysis, but this would result in a significant amount of repetition as most of the new instructions have similar verification conditions and safety requirements. An argument for one instruction — mtgar, $x, n, c$ — illustrates almost all of the mechanics of proving soundness for the new instructions. A few variations for other instructions will also be discussed.

For the case when the reconfigurable array is definitely halted on execution of mtgar, $x, n, c$, progressing to a safe state requires that the safety conditions found in Table 11.3, $n < \text{arraysize}$ and $\text{Pre}_{\rho}\text{(mtgar)}(\rho')$, are satisfied. The first is satisfied because there is a check in the verification condition generator for the instruction which ensures this. This check is possible because the instruction has this argument hard-coded into it. $\text{Pre}_{\rho}\text{(mtgar)}(\rho')$ follows from $\sigma'(\text{Pre}_{\sigma}\text{(mtgar)})$ because $\rho = \tau(\sigma)$ and $\rho' = \tau(\sigma')$ and the requirement that the formula $\text{Pre}_x$ is true in a symbolic state $\sigma$ if and only if the predicate $\text{Pre}_x$ is true in that state. Because all the alterations to the state $\rho$ to get $\rho'$ are also alterations to the state $\sigma$ resulting in $\sigma'$, it is possible to deduce $\rho' = \tau(\sigma')$.

The validity of the verification condition in the new state follows directly from the validity of the verification condition in the current state by conjunction elimination. The correctness of $\sigma'$ and $\sigma_0$ follow from the correctness of $\sigma$ and $\sigma_0$ because $\tau$ is not altered and both $\sigma'$ and $\rho'$ are constructed by a series of register updates which are related by $\tau$ so that $\rho'(r) = \tau(\sigma'(r))$. The correctness of the final state $\epsilon(\sigma')$ follows from the correctness of $\sigma'$ and the fact that the two $\epsilon$ functions put identical $\epsilon_\alpha$ terms
into the same set of registers. As was mentioned above, \( \tau \) needs no extension to deal with these values.

The case where the reconfigurable array may be executing is not as simple. The reason is that the new SAL state is not an update of \( \rho \) but \( \kappa(\rho) \). This means that nondeterministic assignments have occurred to all array accessible variables. The verification condition does not have a similar application of a \( \kappa \) function which would not give the desired result anyway. Although \( \epsilon_{\alpha} \) terms can be equated easily, this is not true for \( \kappa \) terms. Each application of \( \kappa \) to the same predicate could give a different result. This means the automatic equivalence that \( \epsilon_{\alpha} \) terms get in the substitution \( \tau \) is unavailable to \( \kappa \) terms. Instead the verification condition is universally quantified with new variables for all the array accessible registers and the memory register.

To prove progress to a safe state, the same safety requirements must be met —
\( n < \text{arraysize} \) and \( \text{Pre}_{\rho(\text{gacr})}(\rho'') \). Using the same argument as in the previous case, \( n < \text{arraysize} \) is easily satisfied. \( \text{Pre}_{\rho(\text{gacr})}(\rho'') \) takes a more complicated argument however. First \( \tau \) must be extended to the new variables to get \( \tau' \).

\[
\tau' = \tau + [y_1 \mapsto \rho'(\text{garz}_{0,1}), \ldots, y_i \mapsto \rho'(\text{gard}_{i,3}), z \mapsto \rho'(\text{mem})]
\]

Where the addition symbol refers to an update of the substitution without overlap. From the definition of universal quantification we can infer:

\[
\tau'((\neg \sigma(\text{gamem})u \supset z(u) = \sigma(u)) \land \alpha'(\text{Post}_{\sigma(\text{gacr})}) \supset \neg \sigma''(\text{Pre}_{\sigma(\text{gacr})}) \supset SE(i + +, \epsilon(\sigma'', \text{so}, \mathcal{L})))
\]

The formula \( \tau'((\neg \sigma(\text{gamem})u \supset z(u) = \sigma(u)) \) follows from the definition of \( \tau' \) and the \( \kappa \) function. This is because the \( \kappa \) function does not alter any memory locations \( u \) for which \( \neg \rho(\text{gamem})u \), and \( \tau' \) maps \( z \) to \( \rho'(\text{mem}) \), which is just the application of \( \kappa \) to \( \rho(\text{mem}) \). The formula \( \tau'(\sigma'(\text{Post}_{\sigma(\text{gacr})})) \) is assumed to hold because it is a postcondition for the last array execution and no alteration to its accessible variables.
Chapter 11. Extension of Verification Condition Generator

has yet occurred. This allows the inference to $\tau'(\sigma''(\text{Pre}_\sigma(\text{gacr})))$. By the definition of $\tau'$, we can infer the safety requirement $\text{Pre}_{\rho(\text{gacr})}(\beta'')$.

The validity of the verification condition in the new state follows from the above argument because it is the second conjunct in the consequent of the implication. The argument in the first case for the correctness of the new state is still valid, with the addition of the correctness of the new variables. Because of the construction of $\tau'$, these are obviously correct.

The other instructions can be similarly proven to be sound by the mechanics used in the argument for \texttt{mtgar}, $x, n, c$. Most of them are considerably simpler because they are either not the first to access the array variables or do not start the array execution. Some small differences should be mentioned. The safety requirement for loading configurations $\rho(r) \in \Gamma$ is reiterated in state $\sigma$ in the verification condition. This is possible because the set of legal configurations is determined by some other mechanism and is assumed to be available. \texttt{gabump} has a clause in the verification condition corresponding to the safety condition $\text{gacr} \neq 0$ but none for $\text{gacnt} = 0$. This is because the value of $\text{gacnt} = 0$ is always the same for any instruction instance and can be checked without proof, while $\text{gacr} \neq 0$ can change depending on control flow. For the variable argument array variable access instructions, there is an explicit assertion in the verification condition that $n < \text{arraysize}$ because $n$ can change depending on the argument.
Chapter 12

Model Checking the Garp Array

The second major feature of the new system is model checking the reconfigurable array for safety properties. This requires a translation from the machine level configuration format to a model checkable format, a few simplifications, and a method of determining the properties to be checked from the safety properties given by the PCC part of the system.

12.1 Garp to NuSMV Translator

The array configurations must be checked for safety before the modified PCC system can conclude that a program is safe. A translator from the machine code for the Garp array configurations to the NuSMV input language allows this to be done. NuSMV is a symbolic model checker that is freely available and quite powerful [CR98]. Of its many features, only CTL model checking will be used for array configuration in the system presented.

The translator produces a NuSMV input file which has a single main module. This is possible because there is only a single process involved in array execution.
Chapter 12. Model Checking the Garp Array

Although the array is highly parallel, it is globally clocked and there can be no non-deterministic executions for any given configuration and memory state. This makes the translation easy to produce but much longer than a well written modular specification. Fortunately, this verbosity is no problem because the modules are discarded during model checking in favor of a flat model. Therefore, starting with a flat model does not slow down the operation of the model checker.

The translated file contains five sections:

1. module — declaration of description unit and namespace
2. variables — list of variables and their types
3. definitions — list of static definitions
4. assignments — list of initial and next state valuations for variables
5. specifications — list of CTL formulas to check

The module section is very simple. It simply declares that the file contains a main module which gives the model checker a place to start. The specifications section is likewise very simple, being nothing at present but a placeholder for specifications which must be derived in another manner.

For each point in the Garp array configuration where there is a value of interest, there can be one of two types of statements added to the input file. The first is a variable statement which adds a new variable to the model. This results in a new variable in the OBDD representation of the state space by NuSMV. The other type is a definition statement. Definitions are used by NuSMV to build the OBDD representation of the state space but do not result in new OBDD variables. The reason this is possible is that definitions are limited to propositional statements about other values in the model. Thus no temporal formulas may appear in a definition and
their specification can be combined with the specification of any value that refers to
them. This limits their applicability to values that do not directly rely on any values
from the previous states or non-determinism. Luckily, the number of variables used
by the translator is small compared to the number of definitions.

12.2 Variables and Assignments

The variable and assignment sections deal with variables which might directly depend
on previous states for their values. The state of the transition system is a valuation
of the variables, while the transition is determined by the assignments. There are
eight types of variables:

1. infinite — boolean which indicates whether the counter is operational

2. counter — integer which indicates the value of the execution counter

3. mem — booleans which indicate if memory accesses are queued

4. size — integers which indicate the size of memory accesses

5. align — booleans which indicate if memory accesses are aligned

6. num — integers which indicate the number of words for memory accesses

7. bus — boolean arrays which indicate the values on memory busses

8. reg — booleans which indicate the values in array registers

There is only one infinite and one counter variable. These two variables deal with
the behavior of the array counter. The infinite variable is a boolean which indicates
whether the counter decrements or not. The array executes in an infinite loop if
Chapter 12. Model Checking the Garp Array

the counter is not operational and does not decrement. The counter variable is an integer with a range of 0 to 1023 which represents the value of the counter.

The mem, size, align, and num categories each have eight variables corresponding to the eight possible positions in the memory access queue. The mem variables are booleans which indicate whether there is a memory access queued in the position, the size variables can be the integers 0, 8, 16, or 32 and indicate the bit size of any memory access in the position, the align variables are booleans which indicate whether any memory access at the position is word aligned, and the num variables can be the integers 0, 1, 2, or 4 and indicate the number of contiguous words to access at the position.

There are up to 128 bus variables which indicate the value of any current memory read so that any two transfers will have the same values. These come in four groups of 32 to represent the four memory busses but those that are not used are suppressed, so for some configurations there may not be any bus variables at all.

The reg variables are booleans which indicate the value in a cell register. There are four for each register corresponding to the first and second Z and D registers.

The assignment section gives the temporal characteristics of the variables. These are the initial value of the variable and the formula to compute the value of the variable for the next cycle. The state transitions of the model of the array configuration are dependent on the variables and their assignments.

The simplest of the assignments is for bus variables. Because they can have any initial or next value regardless of what is happening in the rest of the array, there is no assignment statement for them. This results in the default behavior of non-deterministic assignment to the variables for each cycle. This also means that at this point, nothing can be said about values in memory, even if they have just been written or read.
The infinite and counter variables are also quite simple. Their initial state is unassigned and so is non-deterministically chosen, but their next states are fully assigned. For the infinite variable, its next value is the conjunction of its current value and a global zero defined value which indicates whether any of the control cells forced the counter to 0. For the counter value there is a case statement for the next value. If the global zero is 1, then the counter goes to 0, else if the infinite variable is 1, then counter gets its present value, else if the counter variable is 0, then it gets 0, else it gets decremented by 1.

The variables dealing with the memory access queue — mem, size, align, and num — have slightly more complicated assignments which depend on the configuration of the control blocks. The initial value for all these variables is 0. The next value depends on whether any control blocks have registered with the translator as memory initiators for the position in the queue represented by the variables. If there are control blocks using the position, then the next value is a case statement that assigns a value depending on whether any of the control blocks initiates a write during the cycle and the value statically associated with the control block for the type of variable. Because only one control block should initiate a memory access for completion at any time the case statement can ignore cases after it finds one that applies. If no case for a control block applies, the default depends on the position of the variable. Variables in the last position of the queue default to 0, and all others get the value of the variable after them in the queue by default.

The reg variables are the most complicated because they depend on their column, whether a transfer takes place, the size of any potential transfer, whether the transfer goes to the Z or D registers, and whether the Z and D registers latch. If there is no potential memory transfer for a register and it latches then its corresponding reg variable's next value is the result of the cell computation for the Z register and the D input for the D register. If it does not latch then the next value is the present
value for both types. Registers that can be written by a memory transfer can be
determined by their column position and the size of the transfer, in which case their
next value is a case statement which gives them the value of the corresponding bus
variable if there is a memory transfer and the value as calculated above if there is
not. The initial value of all reg variables is not explicitly assigned and is therefore
non-deterministically assigned.

12.3 Definitions

The most complex section of the translation is the definitions. Most of the control
path and computation is defined here. There are several important definitions that
are not specific to any cell. These deal with:

1. memory faults

2. zeroing the counter

3. interrupting the processor

The memory fault defined values include:

1. A general memory fault which is true if any of the other memory fault values
   are greater than 1.

2. A multiple memory initialization fault which is a count of how many individual
   control cells have initiated a memory access for the current cycle.

3. Eight memory fault values which are counts of the number of accesses to a
   particular position in the memory queue and accesses already using it.
Chapter 12. Model Checking the Garp Array

The zero and interrupt defined values are disjunctions of local zero and interrupt values from each of the processor control blocks. They indicate if any of the control blocks has zeroed the counter or interrupted the processor respectively.

Both control and logic blocks have a few simple definitions. These are for:

1. input
2. intermediate values
3. output

These are all defined as constants or equivalent to another single value.

Each block has four pairs of boolean input values, A, B, C, and D, which can be either:

1. constants
2. output from some cell
3. the Z registers of the block if it is a logic block
4. the D registers of the block if it is a logic block

Logic blocks have two pairs of boolean intermediate values, Z and D. The Z values can be either:

1. the Z registers of the block
2. the results of the main computation of the block

The D values can be either:
Chapter 12. Model Checking the Garp Array

1. the D registers of the block
2. the D inputs

Logic blocks also have three pairs of boolean output values, V, H, and G, which can be either:

1. the Z intermediate values of the block
2. the D intermediate values of the block

Some logic blocks also have values which allow limited, specialized communication between neighboring blocks:

1. shift ins
2. carry ins
3. shift-carry ins
4. shift outs
5. carry outs
6. shift-carry outs

The in values are just the corresponding out value from their neighbor to the right. Shift outs are simply permutations of one of the pairs of input values of the block. The carry outs and shift-carry outs are more complex, depend on the block’s computation, and are mentioned here only as sources of values for the corresponding in values.

Memory control blocks also have a constant value writer which is true if the cell can write to memory.
Chapter 12. Model Checking the Garp Array

Control blocks have a few more interesting defined values. Each pair of inputs is reduced to a new intermediate value which is either one of the inputs or their disjunction. These values are used to produce function dependent defined values. For processor control blocks there are two values for the local zero and interrupt values which are the conjunction of the A and C intermediate values, and the A and D intermediate value respectively. For memory control blocks there are three values for memory initialization, transfer, and type of access. These are the conjunctions of A and B intermediate values, A and C intermediate values, and A and D intermediate values, respectively.

Logic blocks have much more complicated defined value logic. Luckily, the various logic blocks can be decomposed into seven basic structures:

1. crossbars
2. lookup tables
3. shift inverts
4. selects
5. carry chains
6. carry-save additions
7. result functions

For example, the most complicated logic block is the triple add mode block which consists of three shift inverts, a carry save addition, two lookup tables, a carry chain, and finally a result function.

The crossbar structure simply defines for each pair of inputs, intermediate values each of which are one of the input values as in Figure 12.1.
Chapter 12. Model Checking the Garp Array

Lookup tables result in a pair of values which are each defined as the disjunctive normal form of their inputs with a trailing 0 disjunct. An example is shown in Figure 12.2.

Shift inverts define intermediate values which are the regular or negated value of their corresponding input or shift in, as seen in Figure 12.3 with a shift and invert.

The select structure defines a pair of values by means of a case statement on the C intermediate value which acts as a select line with the selections completely determined by the mode. The result for the select mode is given in Figure 12.4.

Carry chains define three values which follow from the bit level of implementation of addition with the first value being 0 or the carry in if there is one, the second the disjunction of the conjunction of the negation of the propagate input and the generate input, and the conjunction of the propagate input and the carry input, and the carry out the same as the second as in Figure 12.5.

The carry-save addition structure always defines the same two pairs of values — two exclusive disjunctions from three inputs and two disjunctions of the conjunctions each two member combination of the three inputs shifted to the left by one bit. The only possible variation being forcing the carry-shift in to 0 instead of the carry-shift out value from the cell to the right. This is shown in Figure 12.6.

The result function structure simply defines a pair of values each of which is either a defined intermediate value or the regular or negated value of the exclusive disjunction of two such values. The case where the value is the exclusive disjunction of two intermediate values is shown in Figure 12.7.
Chapter 12. Model Checking the Garp Array

ax_0_8_0 := in_0_8_a_1;
ax_0_8_1 := in_0_8_a_0;
bx_0_8_0 := in_0_8_b_1;
bx_0_8_1 := in_0_8_b_0;
cx_0_8_0 := in_0_8_c_0;
cx_0_8_1 := in_0_8_c_0;

Figure 12.1: Example Crossbar

zx_1_10_0 := (ax_1_10_0 & bx_1_10_0 & cx_1_10_0 & dx_1_10_0) |
            (ax_1_10_0 & bx_1_10_0 & cx_1_10_0 & dx_1_10_0) |
            (ax_1_10_0 & bx_1_10_0 & cx_1_10_0 & dx_1_10_0) |
            (ax_1_10_0 & bx_1_10_0 & cx_1_10_0 & dx_1_10_0) |
            (ax_1_10_0 & bx_1_10_0 & cx_1_10_0 & dx_1_10_0) |
            (ax_1_10_0 & bx_1_10_0 & cx_1_10_0 & dx_1_10_0) |
            (ax_1_10_0 & bx_1_10_0 & cx_1_10_0 & dx_1_10_0) |
            (ax_1_10_0 & bx_1_10_0 & cx_1_10_0 & dx_1_10_0) |
            (ax_1_10_0 & bx_1_10_0 & cx_1_10_0 & dx_1_10_0) |

Figure 12.2: Example Table Lookup

ax_1_10_0 := !so_1_9_a;
ax_1_10_1 := !in_1_10_a_0;

Figure 12.3: Example Shift Invert

zx_1_10_0 :=
case
  !cx_1_10_0 & !cx_1_10_1 : ax_1_10_0;
  !cx_1_10_0 & cx_1_10_1 : bx_1_10_0;
  cx_1_10_0 & !cx_1_10_1 : out_0_10_h_0;
  cx_1_10_0 & cx_1_10_1 : in_1_10_d_0;
esac;

Figure 12.4: Example Select

126
Chapter 12. Model Checking the Garp Array

\[
\begin{align*}
k_{1,10,0} & := co_{1,9}; \\
k_{1,10,1} & := (u_{1,10,0} \& v_{1,10,0}) \cup (u_{1,10,0} \& k_{1,10,0}); \\
co_{1,10} & := (u_{1,10,1} \& v_{1,10,1}) \cup (u_{1,10,1} \& k_{1,10,1}); \\
\end{align*}
\]

Figure 12.5: Example Carry Chain

\[
\begin{align*}
c_{1,10,0} & := sco_{1,9}; \\
c_{1,10,1} & := (ax_{1,10,0} \& bx_{1,10,0}) \cup \\
& \quad (ax_{1,10,0} \& cx_{1,10,0}) \cup \\
& \quad (bx_{1,10,0} \& cx_{1,10,0}); \\
sco_{1,10} & := (ax_{1,10,1} \& bx_{1,10,1}) \cup \\
& \quad (ax_{1,10,1} \& cx_{1,10,1}) \cup \\
& \quad (bx_{1,10,1} \& cx_{1,10,1}); \\
\end{align*}
\]

Figure 12.6: Example Carry-Save Addition

\[
\begin{align*}
z_{x,1,10,0} & := ! (u_{1,10,0} \leftrightarrow k_{1,10,0}); \\
z_{x,1,10,1} & := ! (u_{1,10,1} \leftrightarrow k_{1,10,1}); \\
\end{align*}
\]

Figure 12.7: Example Result Function

### 12.4 Problems and Simplifying Assumptions

There are a few problems with the implementation of the translator which mainly concern properties that cannot be inferred from the configuration. They are resolved with a few simplifying assumptions:

1. The memory queues will be treated as if they did not exist.

2. The counter will be given an arbitrary range of between 0 and 1023.

3. Necessary initial values for variables will be produced by hand as if they were generated from the specifications.
Chapter 12. Model Checking the Garp Array

The first problem is that the array memory queues are completely ignored. This is because they have separate configuration files that can be loaded independently. For reasons of simplicity in both the translator and verification condition generator, they are ignored and assumed not to exist. This does not diminish the ability of the reconfigurable array because the array memory queues are memory access constructs that can be replaced by the demand memory access constructs of the array. Because the array memory queues can only access memory sequentially, they are less powerful than the demand memory access constructs and thus should be at least as easy if not easier to model and check.

Another problem is that the counter value is given a fairly arbitrary range. This leads to problems if it is too small, some unsafe actions might manifest themselves only at larger initial counter values, or too large, the model becomes too large for efficient checking. While it might seem that this information could be obtained from the verification condition generator, this is complicated by the fact that a single configuration might be called multiple times. The verification condition generator would have to keep track of every configuration and find the maximum possible value for the counter over all its calls. This could turn out to be quite complicated if the counter arguments are calculated, or even impossible if they depend on input.

Finally, there is a problem with specifications that rely on initial values such as the initial counter value. The only way to refer to such values under the scope of a temporally quantified specification is to keep a copy of them around in a new variable or definition. Unfortunately, definitions are not up to the task as they cannot be assigned to non-deterministically, and this is essential for allowing more than a single possible initial value for different executions. This means that any initial properties which must be saved for the specification must be kept in variables which do not change but which still contribute to the size of the model’s OBDD. What is needed is a method for determining exactly which initial variables are needed to minimize
Chapter 12. Model Checking the Garp Array

the OBDD size. This information can be gotten from the safety specifications for the configuration, but not from the configuration itself. This will be simulated by producing the necessary initial values by hand and inserting them into the description file.

These problems could be solved in a full, concrete implementation of the modified PCC system by considering information other than the array configuration such as the array memory queue configurations, the context in which configurations are used, and the specifications which must be satisfied. The fact that they are currently resolved will not be considered a major shortcoming of the translator.

12.5 Properties to be Model Checked

Once a configuration is translated into NuSMV input format it can be checked for many properties. These properties fall into four classes:

1. Memory access properties that must be satisfied because of the semantics of the reconfigurable array hardware, and which are essentially the same for all configurations.

2. Memory access properties that are specific to individual configurations, and which can be inferred from the configuration’s preconditions, postconditions, memory partitions, and safety predicates, and the processor’s representation of addresses.

3. The postcondition property which is specific to individual configurations, and which can be inferred from the configuration’s preconditions, postconditions, memory partitions, and safety predicates, and the processor’s representation of types.
4. Properties that rely on added hints from the code producer because they are too complex to infer from other sources.

Each of these properties can rely on the preconditions for the array, so the full specification is an implication with the preconditions as an antecedent and a conjunction of the safety specifications as the consequent.

The first class of properties which will be checked for all configurations includes:

1. At most one memory access is initialized per clock cycle.
2. At most one memory access uses each position in the memory access queue.
3. There is a memory item ready to read when the array tries to read one.
4. There is a row initiating a memory write when a row tries to transfer to memory.

Examples given in the next chapter can be seen in Figure 13.9 for the first two properties, and Figures 13.10 and 13.11 for the next two respectively.

These properties all deal with synchronizing memory accesses so that they are defined by the semantics of Garp's reconfigurable array. If any of these properties is false, then there can be undefined behavior from the array. In this case, it would be impossible to give any formal verification of the configuration's safety. Luckily, these properties are both the most important to check and the easiest to specify. From only a list of which control blocks are memory control blocks, it is possible to deduce these specifications. The first two are even created by the translator and summed up by a single variable. These specifications simply count the number of each memory initialization or memory queue position use, and set the memory fault variable to 0 if there is more than one of either of these. The other two properties are more obvious from the example, as they are not created by the translator.

The next class of properties includes:
Chapter 12. Model Checking the Garp Array

1. All memory accesses respect the memory partition.

2. All memory accesses respect the memory access safety properties.

In the example in the next chapter, these share the specifications seen in Figures 13.13 and 13.14 because in the example the memory partition coincides with portions of the saferd and saferwr memory access safety properties which specify boundaries. In addition, Figure 13.12 is part of the memory access safety properties which deals with memory access alignment.

These properties are easy to specify because they deal with memory accesses. Because the array uses standard binary representations for addresses, the configuration cannot be as devious in its memory accesses as it can in other areas. Also, the preconditions and postconditions must be in a standard representation because they are used by the processor as well as the array. This allows a fairly standard translation of the processor version which is expressed in first order logic with standard program types such as integers and characters as primitives, and the model checking version which is expressed in CTL with bits as primitives. It is fairly commonplace to do this sort of translation for circuit design, otherwise there could be no method of computing with integers and other types with a machine which manipulates bits, so it will not be discussed in detail. An explanation of how to do such a translation is available in any introductory digital circuit design text such as [MK01]. There are a few concerns however.

If the memory partition or memory access safety predicates contain a variable initialized by some input to the array then that input would be under the scope of temporal quantifiers. If the value of such a variable ever changed in the array, for example the counter, then there would have to be new variables to preserve the initial values needed for the specification. This is why the initial value problem discussed in Section 12.4 must be solved. It is assumed here that these variables can be easily
added to the model, and some can be seen in the example in the following chapter in Figures 13.13 and 13.14 which were added by hand. They can be easily recognized because they all start with 'init_' followed by the name of the variable whose initial value they represent.

Another problem with producing these specifications is that there may be a dependency between the safe memory access properties and the safe operation properties. An example would be an abstract data type for addresses that allowed them to be constructed from a known address and an integer offset. A safe memory write property might state that it is permissible to write to any address constructed in this manner from an initial set of known addresses. This would pose no problem except that the array might represent the integer offset in a nonstandard format. Thus there would have to be some method of transmitting the information about the integer representation to the code consumer to allow the generation of specifications. This will be discussed later in this section.

The next property is that the postconditions are true when the counter is zero. This specification is trivially true for the example because the postcondition is simply 'true'. There is nothing particularly tricky about this specification as long as the postcondition is reasonable. The antecedent is similar because it is a fairly straight translation of the precondition into CTL.

The last class of properties includes properties dealing with safe operations. Examples include Figures 13.15 through 13.19.

Safe operations at the program level can be used to ensure that basic typing properties hold or can be extended to highly complex predicates for abstract data types, recursive types, or other high level features. Unfortunately, the reconfigurable array does not manipulate data according to the standard operators found in the safety properties, or necessarily represent data in a standard way. This makes the
array both useful as an alternative method of computation and less amenable to producing specifications from the proof-carrying code style safe operation properties.

It is infeasible to make the production of specifications for these safe operation predicates a job for the code consumer because of the above considerations and the combinatorial complexity of the space of possible safe operation sequences. Assistance from the code producer is needed because the final safe operation condition could be the disjunction of an unwieldy large number of cases. For instance, if an array register is loaded from an array of integers and it is safe to add 1 to an integer to get an integer, then it is safe to add 2, 3, 4, etc. to the register to get an integer. It would be helpful in this case to give the model checker a hint that the array in fact adds a specific positive integer instead of checking the entire disjunction. Because one disjunct implies the whole disjunction, this ensures the hint does not allow a violation of the security property.

The lack of an easy method for producing safe operation specifications for the array from those of the program in general suggests two possibilities.

1. Check only limited specifications for safe operations with little help from the code producer.

2. Check complete specifications for safe operations with considerable help from the code producer.

At one extreme, these predicates could be virtually ignored. One could argue that the typing restrictions on normal processor operations are out of place in a reconfigurable array because of its extreme flexibility at bit level manipulations. Maybe all that really matters is that any data accessible to the main processor through memory writes or array accesses should conform to the correct bit patterns for basic types, and this should be reflected in the postconditions. Then the postcondition
would subsume all these concerns. All the code consumer would have to know is the representation of data types.

At the other end, the argument would be that this is not sufficient to ensure safety, because the safe operation predicates must be fulfilled regardless of where the computation occurs. In this case, the specifications to be checked must be provided by the configuration as hints which specify what permissible sequence of operations is being performed by the array. Then every processor accessible data value would be checked to see that it came from a sequence of bit level operations that is equivalent to a sequence of safe operations. This is clearly the safest way to deal with the problem, but it would require more complicated properties to be checked than the previous method.

One problem with the code producer supplying hints for these properties is that they must be matched to the safety predicates supplied by the code consumer. The problem here is that the safety predicates are expressed in first order logic while the specifications to be checked must be expressed in CTL. Thus it might not be trivial to verify that the supplied property is equivalent to some combination of safety predicates. One potential solution is for the code producer to supply:

1. A formula in CTL that can be easily checked for equivalence to a series of safe operations in the program logic.

2. A CTL safety property to be checked.

3. A CTL formula which can be produced from either of the two preceding formulas by adding next operators over non-quantified formulas.

4. A CTL formula which states that memory writes do not happen before sufficient time has passed for the above formula to hold non-trivially.
Chapter 12. Model Checking the Garp Array

This solution allows for the representation and computation of the array to differ significantly from that of the processor while still respecting the safe operation properties. It would also liberate the array from following the exact temporal sequencing of a series of safe operations. Thus the array could take less time by using parallel operations or more time by having a finer grained operation and still conform to the safe operation predicates.

The most significant new requirement for this solution is the equivalency checking. While it is easy to model a sequence of instructions in temporal logic — see [BL02] — it is not as easy to check that an arbitrary array configuration is equivalent to such a series. This is due to the lack of any constraint on the representation of data and the wide range of possible computations. Equivalency checking is required here and can be implemented in a variety of ways. OBDDs can be used here as in [Kro99] with either model checking for equivalence or a specialized equivalency checking algorithm. Other possibilities include SAT based equivalency checking with Stälmarch’s method as used for equivalency of FPGA’s in [SL99]. This choice reflects a tension between efficiency and the need for a small trusted computing base, as the most efficient method is the least like the model checker, and the slowest method is simply model checking.

The decision of how to check complex properties for safe operations is not just an aesthetic one. As properties become more data related as contrasted to control related, they become less efficient to model check. This does not only happen with operation properties. Memory access properties also refer to memory addresses which can be treated as integers by programs and in safety properties. This leads to the need to model check array registers as integers involved in addition and other arithmetic operations. Thus memory safety properties can become data intensive even without reference to the data read or written. While symbolic model checking can speed up the checking of most control properties to an enormous extent, it does not do so
with most arithmetic operations because of the high dependency between different variables. This data intensity will be seen in the section on performance as a very real problem for even the simple example to be examined.
Chapter 13

Example and Results

In order to exhibit a concrete example of the processes of the new system, the example program will be taken through the steps of the new PCC system. The feasibility of model checking the array will be investigated by model checking the translated configuration file for the example, and presenting performance results.

13.1 Example

This section will walk through an example of the combination of PCC and model checking using the example presented in section 3.4. This example is a simple program to lighten an image. The image is composed of a fixed size array of pixels which have alpha, red, blue, and green components. The task is to reduce the value of the color components by one if they are not already zero. The processor does some work, including setting up the array and registers for arguments, but most of the processing is done by the reconfigurable array. This includes loading the pixels from memory, testing if the components are zero, decrementing the component values if needed, and writing the pixel values back to memory.
Table 13.1: Conditions for Example

Before getting to the functioning of the two parts of the system it is necessary to define the conditions on the program and array configuration. These include:

1. \( \text{Pre}_F \) — the function precondition.

2. \( \text{Post}_F \) — the function postcondition.

3. \( \text{Pre}_A \) — the array configuration precondition.

4. \( \text{Post}_A \) — the array configuration postcondition.

5. \( \text{Mem}_A \) — the memory partition.

Table 13.1 shows these conditions. \( \text{MaxAddr} \) refers to the highest address that can be referenced by an address pointer. This could be the highest unsigned integer that will fit in a 32 bit word or some other lesser value. Here, it will be assumed to be the highest unsigned integer, although the only thing that really matters is that the pixel array fits into memory starting at \( \text{IA} \). Because the memory partition coincides with the safe read and write conditions it will not be evident in the properties to be
checked later. If a memory access satisfies the safety condition in this instance, it will automatically satisfy the memory partition. The \( \text{byte}(&a) \) notation refers to the byte at address \( a \) in memory.

Now that the conditions are known the symbolic evaluation of the program to produce the verification condition can be examined. The instruction by instruction construction of the verification condition is shown in Table 13.2, while the final verification condition is in Figure 13.2. The starting state for the symbolic evaluator is on the first line of the table. The \( \sigma_i \) notation stands for the state at the \( i \)th step of the evaluation.

The specification to model check for the reconfigurable array is very long and will be shown here in part. The entire specification can be seen at the end of Appendix C. The specification is an implication, the antecedent of which is partly shown in Figures 13.3 through 13.8. These specifications do not occur under any temporal quantification and thus refer to the state of the array at the initial state.

The precondition that \( \text{gacnt} \) is equal to 400 produces the first two conjuncts — infinite is false and counter is 400 — seen in Figure 13.3. This means that the
### Table 13.2: Example Symbolic Evaluation

<table>
<thead>
<tr>
<th>Instruction</th>
<th>SE</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g_1 := 400$</td>
<td>$SE(0, \sigma_0, 0, 0)$</td>
</tr>
<tr>
<td>$g_2 := IA$</td>
<td>$SE(2, \sigma_1</td>
</tr>
<tr>
<td>$g_3 := g_2 - 14$</td>
<td>$safeSub(IA, 14) \land \begin{cases} SE(3, \sigma_2</td>
</tr>
<tr>
<td>$g_4 := 768$</td>
<td>$safeSub(IA, 14) \land \begin{cases} SE(4, \sigma_3</td>
</tr>
<tr>
<td>$g_5 := 0$</td>
<td>$safeSub(IA, 14) \land \begin{cases} SE(5, \sigma_5</td>
</tr>
<tr>
<td>$g_6 := CA$</td>
<td>$safeSub(IA, 14) \land \begin{cases} SE(6, \sigma_6</td>
</tr>
<tr>
<td>$gacoef ; g_6$</td>
<td>$safeSub(IA, 14) \land CA \in \Gamma \land \begin{cases} SE(7, \sigma_0[gacr \rightarrow CA, gazz_0, 1 \rightarrow 0, \ldots, gard_{2, 3} \rightarrow 0</td>
</tr>
<tr>
<td>$mtgavz ; g_4 ; g_5$</td>
<td>$safeSub(IA, 14) \land CA \in \Gamma \land 0 &lt; 6 \land \begin{cases} SE(8, \sigma_7</td>
</tr>
<tr>
<td>$mga ; g_2 ; g_5$</td>
<td>$safeSub(IA, 14) \land CA \in \Gamma \land 0 &lt; 6 \land \begin{cases} SE(9, \sigma_8</td>
</tr>
<tr>
<td>$mga ; g_3 ; g_6$</td>
<td>$safeSub(IA, 14) \land CA \in \Gamma \land 0 &lt; 6 \land \begin{cases} SE(10, \sigma_9</td>
</tr>
<tr>
<td>$gabump ; g_1$</td>
<td>$safeSub(IA, 14) \land CA \in \Gamma \land 0 &lt; 6 \land CA \neq 0 \land IA &lt; MaxAddr - 800 \land IAmod1 = 0 \land$ \begin{cases} IA = (IA - 14) + 14 \land 400 = 400 \land 768 \leq 768 &lt; 1024 \land 0 &lt; 400 \land 0 \land 0 &lt; 1024 \land SE(11, \sigma_{10}</td>
</tr>
</tbody>
</table>
| $garet \; g_2 \; g_5$ | $safeSub(IA, 14) \land CA \in \Gamma \land 0 < 6 \land CA \neq 0 \land IA < MaxAddr - 800 \land IAmod1 = 0 \land$ \begin{cases} IA = (IA - 14) + 14 \land 400 = 400 \land 768 \leq 768 < 1024 \land 0 < 400 \land 0 < 1024 \land v_{y_1}, \ldots, v_{y_2}, v_{z}(t).((t|\exists Mem_1|u \rightarrow v_{y_2}(u) = \epsilon_{11}(mem)(u) | v.true) \lor$ \begin{cases} SE(12, \sigma_{11}|gazz_{0, 1} \rightarrow y_{y_1}, \ldots, gard_{2, 3} \rightarrow y_{y_2}, gazo_1 \rightarrow 0, (gmem)k \rightarrow 0, gacr \rightarrow 0|, 0, 0) \end{cases} \end{cases}$ }

infinite = 0 & counter = 400

**Figure 13.3: Specification of Antecedent Counter Values**

counter is enabled and set to 400 in the initial state.

The next part of the antecedent is shown in Figure 13.4. It corresponds to the precondition that the address in the first row is word aligned at the initial state. This is accomplished by requiring that the two least significant bits of the Z register used for memory access are 0.

The preconditions that some of the array registers that are in the control path are in certain ranges translate to the specifications in Figure 13.5. These preconditions
Chapter 13. Example and Results

\[
\text{reg\_0\_4\_z\_0} = 0 \land \text{reg\_0\_4\_z\_1} = 0
\]

Figure 13.4: Specification of Antecedent Word Alignment

\[
\begin{align*}
\text{reg\_0\_20\_z\_0} &= 1 \land \text{reg\_0\_20\_z\_1} = 1 \land \text{reg\_0\_21\_z\_0} = 0 \land \text{reg\_0\_21\_z\_1} = 0 \\
\text{reg\_0\_21\_d\_0} &= 0 \land \text{reg\_0\_21\_d\_1} = 0 \land \text{reg\_0\_22\_z\_0} = 0 \land \text{reg\_0\_22\_z\_1} = 0 \\
\text{reg\_1\_22\_z\_0} &= 0 \land \text{reg\_1\_22\_z\_1} = 0 \land \text{reg\_2\_22\_z\_0} = 0 \land \text{reg\_2\_22\_z\_1} = 0 \\
\text{reg\_3\_22\_z\_0} &= 0 \land \text{reg\_3\_22\_z\_1} = 0 \land \text{reg\_4\_22\_z\_0} = 0 \land \text{reg\_4\_22\_z\_1} = 0
\end{align*}
\]

Figure 13.5: Specification of Antecedent Control Path Values

\[
\begin{align*}
\text{!reg\_0\_19\_z\_1} &\lor \text{!reg\_0\_19\_z\_0} \lor \text{!reg\_0\_18\_z\_1} \lor \text{!reg\_0\_18\_z\_0} \\
\text{!reg\_0\_17\_z\_1} &\lor \text{!reg\_0\_17\_z\_0} \lor \text{!reg\_0\_16\_z\_1} \lor \text{!reg\_0\_16\_z\_0} \\
\text{!reg\_0\_15\_z\_1} &\lor \text{!reg\_0\_15\_z\_0} \lor \text{!reg\_0\_14\_z\_1} \lor \text{!reg\_0\_14\_z\_0} \\
\text{!reg\_0\_13\_z\_1} &\lor \text{!reg\_0\_13\_z\_0} \lor \text{!reg\_0\_12\_z\_1} \lor \text{!reg\_0\_12\_z\_0} \\
\text{!reg\_0\_11\_z\_1} &\lor \text{!reg\_0\_11\_z\_0} \lor \text{!reg\_0\_10\_z\_1} \lor \text{!reg\_0\_10\_z\_0} \\
\text{!reg\_0\_9\_z\_1} &\lor \text{!reg\_0\_9\_z\_0} \lor (\text{!reg\_0\_8\_z\_1} \land \text{!reg\_0\_8\_z\_0}) \\
\text{(!reg\_0\_7\_z\_1} &\lor \text{!reg\_0\_7\_z\_0} \lor (\text{!reg\_0\_6\_z\_1} \land \text{!reg\_0\_6\_z\_0}) \\
\text{!reg\_0\_6\_z\_1} &\lor \text{!reg\_0\_6\_z\_0} \lor (\text{!reg\_0\_4\_z\_1} \land \text{!reg\_0\_4\_z\_0}))
\end{align*}
\]

Figure 13.6: Specification of Antecedent Address Value

are actually designed so that these specifications are their translations. It is not important that the registers in the control path have some integer data in them, but that they have a clear path to transmit a pulse down the array.

Figures 13.6 and 13.7 state that the address in the first row is less than the largest unsigned integer minus 800, and that the same address is greater than 16. These correspond to the preconditions for the middle registers of the first row. Here it is important that the registers contain an integer in this range as data. Thus the translation to Boolean language gives a more complex result than in the previous case where a complex integer expression was translated into a simple Boolean expression.

The last part of the antecedent is shown in 13.8, and states that the address in row 5 is 14 less than the address in row 0. This specification is elided as it takes several pages to fully show. The complexity of Boolean expressions corresponding to
Chapter 13. Example and Results

fairly simple integer expressions is quite evident here.

The consequent of the safety specification for the array take several figures to show in part. Figures 13.9 through 13.11 are generic properties that any configuration should satisfy because the semantics of array execution are undefined otherwise. The
rest are specific to the particular array configuration under scrutiny.

Figure 13.9 states that there are never any memory faults while the array is running. These are defined as multiple memory initializations at a time or multiple memory accesses scheduled to transfer at the same time. They are so important that they are defined into a single value by the Garp to NuSMV translation.

Figure 13.10 states that anytime a row tries to load a word that there is something ready for it to read.

Figure 13.11 states that if a row tries to store a word that there is a row initiating a write.

The specifications in Figure 13.12 state that any memory access is word aligned. This is specific to the configuration of the example and is part of the safety properties
Figure 13.12: Memory Alignment Safety Specifications

for safe memory reads and writes shown in Figure 13.1.

Figures 13.13 and 13.14 are specifications of the boundary conditions on the address in row 0 for memory accesses. They are derived with the common bit array definitions of less than, greater than and addition for an unsigned integer interpretation. Figure 13.13 states that the address is always greater than the initial address during memory access, while Figure 13.14 states that the address is never more than 800 addresses from the initial address during memory access. These specifications require the introduction of variables to retain the initial values in the registers, and they can be identified by the prefix 'init'. These variables have also been added by hand to the translated model and are the only such modification besides the specification itself.

The next part of the consequent of the specification is an attempt to specify that the safe operation sections of the safety policy are followed. As was previously discussed there are some problems with this area. One thing that must not be assumed is that every bit level operation by the array satisfies the safe operator definitions. Instead it will be assumed that only values that will be accessible outside of the array will have to be constructed by safe operations. There are three parts to this section:

1. It must be checked that only values in row 4 will ever get stored.

2. It must be checked that the values in row 4 will only be stored if three steps previously a value was loaded into row 1.
Chapter 13. Example and Results

At(mem_init_0 ->
  (init_reg_0.19_z_1 & reg_0.19_z_1) | (init_reg_0.19_z_1 <-> reg_0.19_z_1) &
  (init_reg_0.19_z_0 & reg_0.19_z_0) | (init_reg_0.19_z_0 <-> reg_0.19_z_0) &
  (init_reg_0.18_z_1 & reg_0.18_z_1) | (init_reg_0.18_z_1 <-> reg_0.18_z_1) &
  (init_reg_0.18_z_0 & reg_0.18_z_0) | (init_reg_0.18_z_0 <-> reg_0.18_z_0) &
  (init_reg_0.17_z_1 & reg_0.17_z_1) | (init_reg_0.17_z_1 <-> reg_0.17_z_1) &
  (init_reg_0.17_z_0 & reg_0.17_z_0) | (init_reg_0.17_z_0 <-> reg_0.17_z_0) &
  (init_reg_0.16_z_1 & reg_0.16_z_1) | (init_reg_0.16_z_1 <-> reg_0.16_z_1) &
  (init_reg_0.16_z_0 & reg_0.16_z_0) | (init_reg_0.16_z_0 <-> reg_0.16_z_0) &
  (init_reg_0.15_z_1 & reg_0.15_z_1) | (init_reg_0.15_z_1 <-> reg_0.15_z_1) &
  (init_reg_0.15_z_0 & reg_0.15_z_0) | (init_reg_0.15_z_0 <-> reg_0.15_z_0) &
  (init_reg_0.14_z_1 & reg_0.14_z_1) | (init_reg_0.14_z_1 <-> reg_0.14_z_1) &
  (init_reg_0.14_z_0 & reg_0.14_z_0) | (init_reg_0.14_z_0 <-> reg_0.14_z_0) &
  (init_reg_0.13_z_1 & reg_0.13_z_1) | (init_reg_0.13_z_1 <-> reg_0.13_z_1) &
  (init_reg_0.13_z_0 & reg_0.13_z_0) | (init_reg_0.13_z_0 <-> reg_0.13_z_0) &
  (init_reg_0.12_z_1 & reg_0.12_z_1) | (init_reg_0.12_z_1 <-> reg_0.12_z_1) &
  (init_reg_0.12_z_0 & reg_0.12_z_0) | (init_reg_0.12_z_0 <-> reg_0.12_z_0) &
  (init_reg_0.11_z_1 & reg_0.11_z_1) | (init_reg_0.11_z_1 <-> reg_0.11_z_1) &
  (init_reg_0.11_z_0 & reg_0.11_z_0) | (init_reg_0.11_z_0 <-> reg_0.11_z_0) &
  (init_reg_0.10_z_1 & reg_0.10_z_1) | (init_reg_0.10_z_1 <-> reg_0.10_z_1) &
  (init_reg_0.10_z_0 & reg_0.10_z_0) | (init_reg_0.10_z_0 <-> reg_0.10_z_0) &
  (init_reg_0.09_z_1 & reg_0.09_z_1) | (init_reg_0.09_z_1 <-> reg_0.09_z_1) &
  (init_reg_0.09_z_0 & reg_0.09_z_0) | (init_reg_0.09_z_0 <-> reg_0.09_z_0) &
  (init_reg_0.08_z_1 & reg_0.08_z_1) | (init_reg_0.08_z_1 <-> reg_0.08_z_1) &
  (init_reg_0.08_z_0 & reg_0.08_z_0) | (init_reg_0.08_z_0 <-> reg_0.08_z_0) &
  (init_reg_0.07_z_1 & reg_0.07_z_1) | (init_reg_0.07_z_1 <-> reg_0.07_z_1) &
  (init_reg_0.07_z_0 & reg_0.07_z_0) | (init_reg_0.07_z_0 <-> reg_0.07_z_0) &
  (init_reg_0.06_z_1 & reg_0.06_z_1) | (init_reg_0.06_z_1 <-> reg_0.06_z_1) &
  (init_reg_0.06_z_0 & reg_0.06_z_0) | (init_reg_0.06_z_0 <-> reg_0.06_z_0) &
  (init_reg_0.05_z_1 & reg_0.05_z_1) | (init_reg_0.05_z_1 <-> reg_0.05_z_1) &
  (init_reg_0.05_z_0 & reg_0.05_z_0) | (init_reg_0.05_z_0 <-> reg_0.05_z_0) &
  (init_reg_0.04_z_1 & reg_0.04_z_1) | (init_reg_0.04_z_1 <-> reg_0.04_z_1) &
  (init_reg_0.04_z_0 & reg_0.04_z_0) | (init_reg_0.04_z_0 <-> reg_0.04_z_0) &

u_counter = 0]

Figure 13.13: Greater than or Equal to Boundary Specification

3. It must be checked that row 1 is safely transformed into row 4 in three steps.

Figure 13.15 specifies the first part. All the memory control blocks are specified to not transfer their contents to memory.

Figure 13.16 specifies the second part. It states that if row 4 initiates a memory transfer in three clock cycles, then row 1 is initiating one currently. In order to prevent row 4 from initiating a transfer in the first three cycles, there must be explicit specifications such as those in the second row of the specification. This is because there are no temporal operators which specify past events which would be

145
Chapter 13. Example and Results

A mem_init_0 ->
((init_reg_0_6_z_1 & init_reg_0_6_z_0 & init_reg_0_7_z_1) |
  init_reg_0_8_z_0 | init_reg_0_8_z_1 & init_reg_0_9_z_0 &
  init_reg_0_9_z_1 & ... & init_reg_0_18_z_1 & init_reg_0_19_z_0)
<-> (init_reg_0_19_z_1) |
((init_reg_0_6_z_1 & init_reg_0_7_z_0 & init_reg_0_7_z_1) |
  init_reg_0_8_z_0 | init_reg_0_8_z_1 & init_reg_0_9_z_0 &
  init_reg_0_9_z_1 & ... & init_reg_0_18_z_0 & init_reg_0_19_z_0)
<-> (init_reg_0_19_z_0) & (reg_0_19_z_0)
((init_reg_0_6_z_1 & init_reg_0_7_z_0 & init_reg_0_7_z_1) |
  init_reg_0_8_z_0 | init_reg_0_8_z_1 & init_reg_0_9_z_0 &
  init_reg_0_9_z_1 & ... & init_reg_0_18_z_0 & init_reg_0_19_z_1)
<-> (init_reg_0_19_z_0) & (reg_0_19_z_0)

Figure 13.14: Less than Boundary Specification

146
AG(!mem_tran_0) & AG(mem_tran_1 -> !type_1) & AG(!mem_tran_5)

Figure 13.15: Memory Write Limitation Specification

AG(AX(AX(mem_tran_4)) -> mem_tran_1) &
!mem_tran_4 & AX(!mem_tran_4) & AX(AX(!mem_tran_4))

Figure 13.16: Memory Write Synchronization Specification

AG((!reg_1_4_z_0 & !reg_1_4_z_1 & !reg_1_5_z_0 & !reg_1_5_z_1 &
!reg_1_6_z_0 & !reg_1_6_z_1 & !reg_1_7_z_0 & !reg_1_7_z_1)->
AX(AX(!reg_4_4_z_0 & !reg_4_4_z_1 & !reg_4_5_z_0 & !reg_4_5_z_1 &
!reg_4_6_z_0 & !reg_4_6_z_1 & !reg_4_7_z_0 & !reg_4_7_z_1)))) &
AG((!reg_1_8_z_0 & !reg_1_8_z_1 & !reg_1_9_z_0 & !reg_1_9_z_1 &
!reg_1_10_z_0 & !reg_1_10_z_1 & !reg_1_11_z_0 & !reg_1_11_z_1)->
AX(AX(!reg_4_8_z_0 & !reg_4_8_z_1 & !reg_4_9_z_0 & !reg_4_9_z_1 &
!reg_4_10_z_0 & !reg_4_10_z_1 & !reg_4_11_z_0 & !reg_4_11_z_1)))) &
AG((!reg_1_12_z_0 & !reg_1_12_z_1 & !reg_1_13_z_0 & !reg_1_13_z_1 &
!reg_1_14_z_0 & !reg_1_14_z_1 & !reg_1_15_z_0 & !reg_1_15_z_1)->
AX(AX(!reg_4_12_z_0 & !reg_4_12_z_1 & !reg_4_13_z_0 & !reg_4_13_z_1 &
!reg_4_14_z_0 & !reg_4_14_z_1 & !reg_4_15_z_0 & !reg_4_15_z_1))))

Figure 13.17: Safe Operation Specification

the natural way to express this specification.

Figures 13.17 through 13.18 specify the safe transformation properties of the configuration. They all specify what the value ready to be written will be like in three cycles.

Figure 13.17 states that if a color byte in the pixel contains a 0 in row 1 then in three cycles the same color byte in row 4 will also be 0. This requires three similar specifications for each of the color bytes; red, green, and blue.

Figure 13.18 states that if a color byte in row 1 is not 0 then in three cycles row 4 will contain a byte 1 less than the original. This also requires three similar specifications for the three color bytes.

Figure 13.19 simply states that the α channel byte in the pixel is the same in row
Figure 13.18: Safe Operation Specification

Figure 13.19: Safe Operation Specification

1 and three cycles later in row 4.

Because the system can not produce the specifications for the safe operation properties in any reasonable manner, they must be given as hints by the code producer. This leaves a problem. How does the system know that the hints satisfy the safe
operation property? One obvious solution would be to include a proof that the hints satisfy the safety property. Unfortunately, the proof checker already in the assumed PCC system is not powerful enough to handle arbitrary proofs about temporal logic.

A method discussed in Section 12.5 could be used here. There is a sequence of safe operations which could be provided by the code producer:

1. Read a pixel value from the image array.
2. Subtract 1 from the second byte if it is greater than 0.
3. Subtract 1 from the third byte if it is greater than 0.
4. Subtract 1 from the fourth byte if it is greater than 0.
5. Write the pixel value back to the image array.

This sequence can be translated into CTL and checked to be equivalent to the specification provided, with suitable translation into bit level language and relaxation of the temporal requirements.

13.2 Performance of NuSMV on Translated Input Files

In order for the PCC and model checking approach to safety that has been outlined to work in practice it is not sufficient for it to be logically sound. There must also be an acceptable level of performance for the system. Exactly where to draw the line between acceptable and unacceptable performance is of course a fuzzy decision and is dependent on the context. In a mobile applet context the performance must be good enough that a receiver of the applet would not lose patience and abort
the applet execution. This would require performance measured in seconds. Ten to fifteen seconds would probably be the maximum delay tolerable. In the context of downloading an application for permanent installation and repeated execution, a much larger delay would be acceptable. This delay could be measured in minutes and still be tolerable, but hours or days might be too long a delay. In any case, delays that ran into weeks or months are clearly unacceptable in almost any context.

There are several factors which contribute to the performance of the system. The factors which apply in all contexts are the speed of proof checking and model checking. The process of model checking will not consider translation time because any production system would generate a model directly from the configuration and not have a separate translation step. There is no reason to believe that this would be significantly slower or faster than using the built in specification language. The speed of proof checking will not be considered because there is no implementation of this part of the system. Some indication of the performance of this aspect of the system can be gotten from Necula’s thesis[Nec98]. Another possible factor for some contexts is download time. This should be fairly fast as far as model checking is concerned because there is no inherent size blow up for communication as there is for proof communication. This leaves the performance of the model checker, NuSMV 2.0 running on an AMD Athlon at 1900 MHz with 1024 M of memory under Debian Gnu/Linux 2.2 in this case, as the main factor to be explored.

Even given the very fuzzy performance requirements, the performance for the model checking part of the system falls into two classes depending on the type of specification to be checked:

1. Acceptable for almost any purpose.

2. Unacceptable for almost any purpose.

The most obviously apparent difference between specifications in these two classes is
Chapter 13. Example and Results

their size and amount of nesting. If an expression is very large or has deep nesting of subexpressions, it almost always takes too long to check.

Specifications which do not deal with arithmetic operations besides comparisons of registers to constants can be checked in about a second and scale to larger examples very well. Examples are the antecedent specifications in Figures 13.3 through 13.7, combined with consequent specifications in Figures 13.9 through 13.12, Figures 13.15 through 13.19.

On the other side of the dividing line are specifications that involve comparing two 32 bit registers, or add values to 32 bit registers. These show up in the antecedent specification in Figure 13.8, and the consequent specifications in Figures 13.13 and 13.14. These specifications are either very long or have a large amount of nesting as can be easily seen. Experimentally, when these specifications including these expressions were checked on the same machine that checked the previous specifications in about a second, they took too long to finish. After two weeks of computation, the processes were stopped, so no actual value for their computation time is available, but it is clear that the performance of the model checker is insufficient to model check these properties for almost any purpose.

Unfortunately, this examination of performance is not very thorough. The example is very simple and covers only a small sample of possible specifications. There are undoubtedly specifications which do not fall into the two classes and which would be acceptable in some contexts but not others. The only question is whether these specifications would appear with any frequency in real applications. This question is not answered here except by pointing out that this dichotomy of fast and slow specifications is similar to the dichotomy in performance between control and data specifications that has been observed before [CGP99]. Although some data oriented specifications were checked very quickly, the safe operation and comparison to constants specifications in particular, most data oriented specifications were very slow.
Chapter 13. Example and Results

to check and all control oriented specifications were very fast to check.

One possibility that was not explored is the use of alternative representations for model checking besides OBDDs. Because arithmetic circuits are so ubiquitous, there has been much interest in finding methods to model check them efficiently. Alternatives to BDDs such as algebraic decision diagrams, binary moment diagrams, and their variants have been used successfully to model check arithmetic circuits. These and many more BDD variants are discussed in [MT98]. Completely different representations are also possible such as Presburger arithmetic which can be seen in [BGP99]. In any case, NuSMV does not allow for any of these variations in model checking representations so they were not used.

These alternatives do give reason to believe that there is some hope for checking the specifications that proved too costly with OBDD based model checking because they all involved arithmetic operations. In particular, using Presburger arithmetic representations seems to fit nicely with the reconfigurable array because they compute with essentially the same set of operations. Experimental evidence suggests that significant gains in speed can be realized as long as alternating universal and existential temporal quantifiers are not extensively used in the specifications. For safety properties, this should not be a problem.
Chapter 14

Conclusions and Future Work

Although the system presented goes a long way towards solving many safety problems for reconfigurable processor programs, there is still much work to be done. This work falls into the general categories of:

1. Implementing and exploring the full system.
2. Improving performance.
3. Exploring major modifications and alternatives to the system.

14.1 Conclusions

The system presented for combining proof-carrying code and model checking to ensure the safety of programs for reconfigurable processors is only partially successful. The problem of performance reduces the effectiveness of the system greatly.

One success on paper of the system is the extended verification condition generator for handling nondeterminism in the processor. Because the verification condition
for the processor is independent of the method of verifying the safety properties for
the reconfigurable array, the generator would still be useful even if the model check-
ing is eliminated. Other types of systems such as symmetric multiprocessors and
multi-threaded processors could also benefit from the same techniques because of
their parallel and nondeterministic execution properties.

The ability to check control properties in the reconfigurable array, such as the
lack of memory faults, is a significant improvement over any other system. Usually
these types of properties for FPGAs and similar reconfigurable fabrics are completely
ignored or simply assumed to be assured by a trusted compiler or designer. As
compilers and designers are not perfect even if trustworthy, checking these properties
is quite useful even if only for debugging a program.

Hopefully, further research into the possible variations will lead to a more com-
prehensive and effective system.

14.2 Full Implementation of the System

The most obvious area of future work is a full implementation of the system. As
it stands, only a translator from the Garp configuration language to the NuSMV
language is implemented. The full system would require a proof checker, a verification
condition generator, a concrete method of distributing proofs and hints with the
executable, and a certifying compiler at the very least. While Necula’s work includes
implementations of all these additional system components, it is necessary to extend
them to the present system to experimentally validate the approach.

The most extensive change would be to the certifying compiler because it would
have to compile for a radically different kind of processor. Necula’s Touchstone
compiler produces both executables and proofs of safety for standard processors
using a variety of decision procedures. These procedures rely on information which is generated during more mundane compilation tasks and loop invariants. The compiler also provides many optimization techniques which are shown to preserve the loop invariants.

The challenge for a Garp certifying compiler is to investigate how techniques for reconfigurable array compilation fit with techniques for certifying compilation. Does a compiler which partitions a program into standard and array sections still have the needed information to produce a proof? Do optimizations still preserve loop invariants? Are the new verification conditions amenable to the same type of proof procedures? These questions all require an implementation for satisfactory investigation. Unfortunately, there is no publicly available compiler for the Garp processor, much less a choice of possible implementations to investigate. Any research in this area would require the complete implementation of a compiler instead of an extension, and would produce uncertain answers if it did not allow the certifying compilation techniques. It is always possible that a different technique for compilation or producing proofs would work.

The other parts of the system are more similar to the original system, but could still require an implementation for some investigations. Does the verification condition get exponentially larger for larger programs? Are the proof and hints sufficiently compact for efficient network access? There is no reason to believe that the answer to these questions is different for the extended PCC system presented than for Necula’s PCC system, but there is also no reason to dismiss them. An implementation would provide answers to these questions as well as others that might not even be apparent without it.

155
14.3 Performance

One of the major problems with the system presented is the performance of the model checker. Some of the specifications in the example dealing with arithmetic data took more than two weeks to check. It might even be the case that they would take years because they were not allowed to run to completion. This is not simply an unusual effect of the specific example. Specifications using arithmetic data are needed in any program that accesses memory in a range specified by the safety condition. This includes almost all interesting programs. There must be an answer to this problem or the entire system will be virtually useless.

One possible solution to the poor performance of model checking on data registers is to introduce modular model checking [CGP99]. Modular model checking reduces the size of model checking problems using a divide-and-conquer methodology. If the offending specifications can be broken down into easier specifications that can be combined to prove the full specification, then the performance could be improved. Unfortunately, it is not easy to do this, especially for the types of specifications that are the problem. Because these arithmetic data oriented specifications have complicated interdependencies between variables, it will take a good deal of work to determine if modular model checking is of any help.

The more usual approach to dealing with data in model checking is to use data abstraction [CGP99]. In this approach, the complexity of a system is reduced by reducing the number of states which describe it. Many states may be abstracted to one which represents all of them. In the example this could work by having only a few memory location states instead of one for every 32 bit number. The specifications would then state that at every memory access the memory location state of a row is a permissible state. By reducing the complexity of the specification, a decrease in checking time can be expected. However, data abstraction is far from
an automatic process and some form of hint would have to be provided. This leaves another problem with how to verify that the hint is acceptable.

Another possible solution is to use other model checking techniques instead of or in addition to OBDD based symbolic model checking. As discussed in Section 13.2, there are many alternative representations that have been proposed to improve model checking involving arithmetic data. Which of these is best for this application is an open question, as is the question of whether there are other types of specifications that would lead to performance problems which are not handled well by these alternatives.

14.4 Other Approaches

A radical alteration to the system would be to use a higher order proof checker instead of a first order one. This gives several advantages. With a higher order proof system one can eliminate the model checking entirely. Higher order logic has been shown to be useful for hardware verification proofs, so it should be acceptable for reconfigurable array configurations which are very similar in execution to hardware[Kro99]. Thus the PCC system could handle the entire safety verification. The major problems here are that higher order logics are less automatable than first order and might pose problems for generating proofs, and that generating the verification condition by symbolic evaluation is not possible for the reconfigurable array.

There are other possibilities with the use of higher order logic. Foundational proof-carrying code uses higher order logic to greatly reduce the trusted computing base by having the safety property include the axiomatization of the logic for the proof [App01]. This eliminates the need for the verification condition generator at the expense of a larger proof, but solves the problem of generating the verification condition by symbolic evaluation. This could also allow much more room for crafting safety properties dealing with the interaction of the processor and reconfigurable
array which allowed sharing memory, halting the array by the processor, and other such actions which are simply banned by the present system.

Another approach to unifying the logics used for PCC and model checking is to use a temporal logic for both. This approach allows for a smaller trusted computing base as in foundational PCC, but does not entail the extensive proofs in foundational PCC[BL02]. In addition, having safety properties expressed in temporal logic should ease the problem of translating from first order logic to temporal logic.

Another variation would be to use model checking for control oriented specifications only, and have proofs for data oriented ones. Higher order logic checkers would provide a means for reasoning about temporal and first order specifications and providing proofs about questions involving both. Using temporal logic checkers and safety specifications would make this type of reasoning even easier.

Finally, the expressiveness of the safety conditions could be limited. Systems such as typed assembly language and typed intermediate language give safety assurances as long as the safety properties are expressible in their type systems[CCH+, MWCG99, TMC+96]. How the reduction in expressiveness affects the ability to reason about temporal properties is unclear at this point. It could possibly allow for simpler interactions between the model checker and safety policy.
Appendices
Appendix A

Garp to NuSMV translator

See

http://www.cs.unm.edu/~slugboy/thesis/Translator.tar.gz
Appendix B

Translated Example Garp Array Configuration

See

http://www.cs.unm.edu/~slugboy/thesis/example.smv
Appendix C

SMV Example Garp Array Configuration

See

http://www.cs.unm.edu/~slugboy/thesis/example.reference.smv
Appendix D

Example Garp Array
Configuration Machine Code

See

http://www.cs.unm.edu/~slugboy/thesis/example.c
References


References


References


References