ABSTRACT

DUVARNEY, DANIEL C. Abstraction-Based Generation of Finite State Models from C Programs. (Under the direction of Associate Professor S. Purushothaman Iyer).

Model checking is a major advancement in the quest for practical automatic verification methods for computer systems, and has been effectively used to discover flaws in real-world hardware systems. Unfortunately, applying model-checking techniques to software systems has proved to be more difficult, due to the large number of states and irregular transitions of such systems. One promising method for generating reasonably-sized models from programs is the use of *data abstraction*, in which the program data is mapped from a large set of possible values to a much smaller set of abstract values. This thesis develops a method which, given a program in the C language and an abstraction mapping, allows the automatic construction of an abstract labeled transition system (LTS), which is much smaller than the concrete LTS (the LTS which would be generated without the benefit of abstraction). The method is shown to be *sound* in the sense that if a program is well-behaved in its use of pointers, then any linear temporal logic formula which holds true for the corresponding abstract LTS will also hold true for the concrete LTS. Furthermore, if a design exists in the form of a transition system, then the abstract LTS can be checked against the design for bisimilarity. Bisimilarity ensures that the program is a faithful implementation of the design. A suite of software tools has been implemented based upon the theory. These tools interface with the Concurrency Workbench, a model checking system. A case study is presented which shows the practicality of this technique for verifying real-world C programs.

Abstraction-Based Generation of Finite State Models from C Programs

by

Daniel C. DuVarney

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial satisfaction of the
requirements for the Degree of
Doctor of Philosophy

Department of Computer Science

Raleigh

2002

Approved By:

Dr. K. C. Tai	Dr. John W. Baugh, Jr.
Dr. S. Purushothaman Iyer Chair of Advisory Committee	Dr. W. Rance Cleaveland II

Dedicated to my brother Michael.

Biography

Dan DuVarney was born on August 22, 1964, in Worcester, Massachusetts. He grew up in the town of Stone Mountain, Georgia, where he attended Redan High School. He graduated with a B.S. in Mathematics and Computer Science from Emory University in 1985. During his undergraduate years, he co-founded a company producing image processing systems. He received an M.S. degree in Information and Computer Science from the Georgia Institute of Technology in 1987, after which he developed debuggers and other programming tools for Data General Corporation until 1992. Dan began taking courses from the Computer Science department of North Carolina State University in the Summer of that year, and shortly thereafter began working towards the Ph.D. degree under the direction of Dr. Purush Iyer. His research interests include formal methods, program analysis, and programming language implementation.

His non-research interests include blues piano, Ultimate Frisbee, and long-distance running. Dan was married in 1998 to Zeynep Dayar. He currently has no children, but does enjoy the company of two Welsh Corgis, Zoe and Merlin, who kindly kept his feet warm while he typed this dissertation.

Acknowledgements

This dissertation wouldn't have been finished if weren't for the support and guidance of others. First and foremost, I would like to acknowledge the kind assistance, support, and patience of my advisor, Dr. Purush Iyer, who persistently encouraged me to finish, and always maintained a positive attitude. Also, the support and advice of Dr. Rance Cleaveland was critical. Rance even continued to offer advice, and even paid for me to come for a visit, after he left Raleigh for the colder climate of Stony Brook. I would also like to thank my other committee members, Dr. K.C. Tai, and Dr. John Baugh, who were extremely understanding, and generous with their time when called upon. Finally, and most importantly, I would like to thank my wife Zeynep for her love and support during my years as a Ph.D. student. Zeynep made my life much more comfortable than it normally would have been during the laborious process of finishing this thesis.

Raleigh, 2002

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As long as humans have been developing computational systems, there has always been the problem of "bugs" — situations where the system does not compute the expected result, or does not behave as expected. These malfunctions can be quite annoying, as any user whose work has been lost by a computer which suddenly "froze" due to an operating system crash can attest. However, the risks imposed on human civilization by defective software are much greater than a mere annoyance. In today's world, software is omnipresent, as computers are embedded in such critical devices as pacemakers and automobile brakes. Furthermore, computers are used in a host of critical safety applications, such as air- and rail-traffic control systems, nuclear power plants, and aircraft avionics. A software bug in one of these systems could potentially cause tremendous property damage and loss of human life. The high cost of failure for such systems makes it worthwhile to invest considerable resources into the detection and elimination of software bugs.

Model checking [27] is a promising technique for verification of computer systems. To date, model checking has been successfully used to verify microelectronic circuit designs. Unfortunately, due to their greater complexity, attempts to apply model checking to software systems have not been very fruitful. The goal of this thesis is to apply *abstraction* techniques to software systems (more specifically, programs written in the C programming language), in order to shrink their complexity to the point where model checking is possible, while still retaining the essential information required to verify the correctness properties of interest.

1.1 The Case for Verification

The possibility of critical software systems malfunctioning and wreaking havoc is not just a remote scenario dreamed up by paranoid software reliability experts; in fact there are a large number of well-documented malfunctions which have had large economic costs and even directly caused human deaths [57, 52]. A few such cases:

The Therac 25 radiation therapy machine. Due to a software bug in which the amount of radiation displayed on the device's user interface was not the same as the (much larger) actual dose delivered, a number of patients were exposed to severe overdoses of radiation. Three documented deaths occurred [53].

The Ariane 5 rocket. On June 4, 1996, the European Space Agency's Ariane 5 rocket exploded shortly after takeoff. Along with rocket, a satellite costing $55 million was destroyed [57]. An inquiry into the cause of the accident determined that the cause was an overflow error during a numerical computation [1].

The Year 2000 bug. The pervasive shortcut of storing only the last 2 digits of the year in computer records led to a massive effort to repair the software in numerous systems around the world before January 1, 2000. The estimated amount spent on repairs in the United States was $114 billion [65].

As these examples show, software defects are a serious problem with serious consequences for human civilization. Hence, it should come as no surprise that there has been large amount of research into techniques for developing defect-free software. The basic idea behind such techniques to develop a *specification*, which describes the intended behavior of a program, and then determine if the program *satisfies* (i.e., obeys) the specification. The problem of determining whether or not a program satisfies its specification is known as *verification.*

1.2 The Case for Model Checking

There have been three basic approaches to the verification problem. The oldest, *testing*, relies on repeated execution of all or part of a program on as many different inputs as possible. Although testing is an effective technique for finding many defects, it suffers from the drawback that it can only catch bugs which the test designers anticipate and test for. Numerous sophisticated testing techniques have been developed, such as *white box testing*,

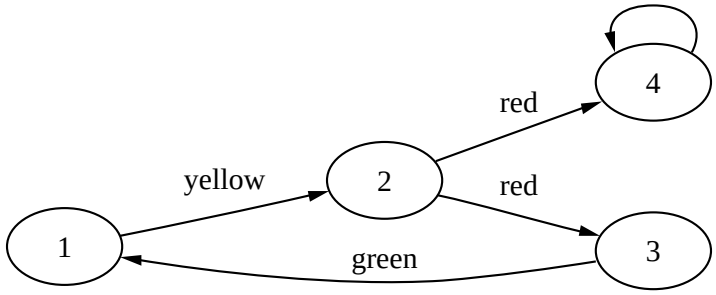

Figure 1.1: A labeled transition system representing the observed behavior of a traffic light.

which uses control- and data-flow analysis to automatically generate tests, but nevertheless, the number of tests required to show 100% compiliance with a specification is impractically large (and even infinite in most cases). As Dijkstra observed, "Testing demonstrates the presence of bugs, not their absence." [23]

The second approach is *proof-based verification*. Given a specification and a program (both expressed in a language with a formal semantics), it may be possible to construct a proof that the program satisfies the specification. This approach is much more complete than testing, since the proof effectively anticipates all the possible inputs to the program. Unfortunately, proof-based verification systems haven't been widely adopted because the tremendous complexity involved in developing the proofs requires too much expertise from the user, although research is continuing in this area.

The third approach, and the one targeted by this thesis, is *model checking*. Model checking was proposed independently in 1981 by Emerson and Clarke [28], and Quielle and Sifakis [47]. The key idea behind model checking is to represent the system as a primitive finite state system of some sort (see figure 1.1 for an example). Each state in the model corresponds to a specific point in the execution of the system — the contents of the memory, and the locus of control in each process that is executing. Transitions between states represent possible steps in the execution, and may be non-deterministic. The model as a whole represents every possible execution of the system.

Given such a simple system representation, the correctness of the system can be verified by exhaustively searching the state space to ensure that all possible execution paths satisfy the specification. This method is fully automatic, is computationally efficient, can

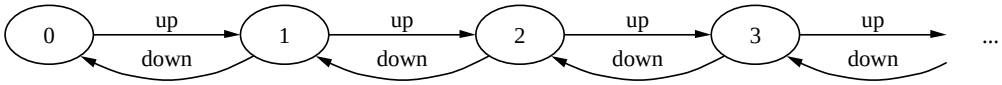

Figure 1.2: Transition system for an up/down counter.

easily deal with concurrency, and furthermore, if the system does not satisfy the specification, then the transition path which led to the specification violation provides a useful counterexample which can be used to debug the system. For example, in figure 1.1, the specification that "a red transition will eventually follow a green transition" can be shown to be true, because there only two paths leading from the first state: $1 \rightarrow 2 \rightarrow 3$ and $1 \rightarrow 2 \rightarrow 4$, and both these paths include a red transition. On the other hand, the the specification that "a green transition will eventually follow a red transition" is not satisfied, because the system includes the possibility that the system will loop forever in the "red" state. The counterexample in this case is $2 \rightarrow 4 \rightarrow 4 \rightarrow 4 \rightarrow \ldots$

Model checking has been successfully used to verify real-world computational systems, such as the floating-point unit of the Pentium microprocessor [15]. While model checking isn't a panacea, we believe it is a worthwhile weapon in the software verifier's anti-bug arsenal, particularly when used in combination with other verification techniques, and we would like to have the ability to model-check a software system.

1.3 The Case for Data Abstraction

A major advantage of model checking is the simplicity of the verification procedure. Naturally, there is a trade-off required, and in this case it is the size of the model. Since each unique combination of variable values and control-flow point results in a unique state, for a program with two 32-bit integer values and 5 nodes in the control-flow graph, the number of possible states is $5 \cdot 2^{64}$, which is many times larger than can be represented explicitly in the memory of a computer. And real-world programs are much worse, with thousands of variables and functions, including the possible use of recursion which may not even be finitely bounded. This problem is known as the *state explosion problem*.

A host of techniques have been developed for dealing with state explosion. Here is a brief summary of the major ones:

Symbolic model checking [11, 48, 55]. Symbolic model checking uses BDDs (Binary Decision Diagrams) [10] to represent the model. Symbolic model checking is very useful for certain types of hardware systems, for which it yields an exponential reduction in the state space. Unfortunately, it generally doesn't produce any space savings for software systems due mainly to the more haphazard transitions between states that software systems exhibit.

Partial-Order Reduction [59]. Partial-Order reductions are used to compress the state space of concurrent programs which have asynchronous threads running in parallel. Normally, the system model would have to represent every possible interleaving of the states of the concurrent threads. The use of a partial order instead of explicit transitions allows these interleavings to be implicitly represented, greatly reducing the number of states. Unlike symbolic model checking, partial-order reductions works well for software systems. However, it does nothing to address the problem of the huge number of states generated by the variables of a software program.

Compositional model checking [16]. Compositional model checking works by partitioning a system into modules, verifying each one separately, and then combining the results. The savings comes from exploiting the fact that it's safe to assume that the other modules are correct while proving the correctness of a particular module. Modular techniques can easily be applied to software systems, but these techniques also don't address the issue of the number of states in the program data variables.

Predicate Abstraction [37]. Predicate abstraction works by representing the values of a system's variables as a truth assignment to a set of predicates. It an effective way to deal with the data variables of a program, and in fact, research is currently under way to apply it to C programs [4].

Data Abstraction [14]. The basic idea behind abstraction is that the program data can be mapped to a much smaller set of abstract values, while still preserving the essential behaviors of the program. Of course, since abstraction involves a loss of information, it's possible that if the model is too abstract, we won't be able to determine whether the property we are interested in holds true. Hence, the art of abstraction requires a balancing of the desire to reduce memory usage with the necessity of usefulness of the resulting model. For example, consider the system depicted in figure 1.2, which represents the states of an integer counter. One property we might want to verify is that odd states always follow even

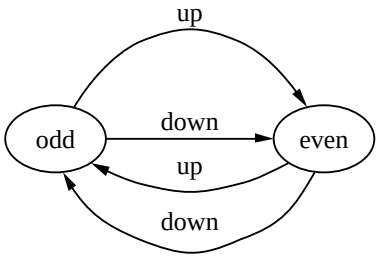

Figure 1.3: Result of applying an odd/even abstraction to the transition system of figure 1.2.

states, and vice-versa. If we apply the abstraction function

$$f(x) = \begin{cases} even & \text{if } x \; mod \; 2 = 0 \\ odd & \text{otherwise} \end{cases}$$

to the (100 state) transition system of figure 1.2, then the 2 state system of figure 1.3 will result.

Of the major techniques, only predicate and data abstraction address the issue of state explosion in the data storage used by a system. Between these two, we have chosen to take the data abstraction route. We view both abstraction techniques as viable approaches which should be investigated and compared.

1.4 The Case for Automated Abstraction

Initial applications of abstraction on real-world systems were done manually. For example, in [24], the GNU I-Protocol implementation was shown to have a bug by manually translating the C code into the input language of several different model checkers. This process is tedious and error prone. In particular, one must be careful to generate a *sound* abstract model, which means that any property shown to hold true for the abstract model must also be true for the "real" model (more commonly known as, and henceforth called the *concrete* model). In our case, the term "property" is equivalent to execution path, not any temporal logic formula which is determined to hold true by a model-checking algorithm.

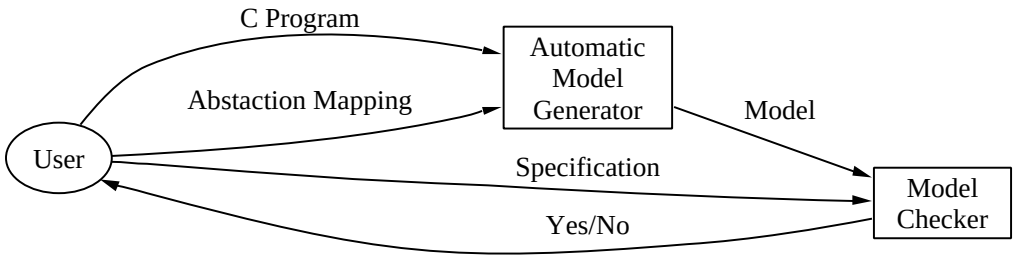

Figure 1.4: A high-level view of the model generation and checking process.

The process of abstraction is similar to that of compilation. What is basically occurring is a translation from a high-level formal language (the program) to a lower-level language (the LTS). Hence, the abstraction process should be an easy target for automation. Given both a program and a user-supplied *abstraction mapping* which specifies how to abstract each variable, an *automatic model generator* should be able to generate an LTS which can be directly fed to a model checking system. A high-level view of this process is shown in figure 1.4. We have chosen to use the *Concurrency Workbench of the New Century* [17, 61] as our model checker.

A second important application of automated abstraction is *design checking.* In application areas such as communications protocols, it is common to start with a design in the form of a *finite state machine* (FSM), and then develop a program which implements the FSM. We believe an automatic model generator could be used to effectively reverse-engineer the design of the program, which could then be compared for equality to the original design (using either trace equivalence or weak bisimulation).

1.5 The Case for C

The C programming language [49, 38] was developed by Dennis Ritchie in the early 1970s. C is extremely popular for systems implementation today, primarily because it has enough high-level features to support development of complex data structures and

control-flow patterns, and yet its type system is "low-level" in the sense that it allows direct manipulation of data at the bit- and byte-level (although we would be remiss if we didn't point out that the lack of type safety exacts a penalty in the form of memory leaks, dangling pointers, and temporal access errors). Because of the unfettered access to memory, many embedded systems, communications protocols, and operating systems are written in C.

These are the types of applications which would benefit from model checking, and have the important property that they tend to have a finite number of states, which is important because most model checking tools only work on finite state systems. For these reasons, C has been chosen as the source language.

1.6 Research Goals

The primary goals of this thesis were to develop techniques and tools for automatically generating data-abstracted finite models of C programs. Specifically, we have achieved the following:

- Developed a sound theoretical technique for applying data abstraction to C programs.
- Identified what sorts of programs the technique can be applied to.
- Designed and implemented tools for automating the abstraction process, which interface with the Concurrency Workbench of the New Century.
- Performed experiments to determine the practicality of our approach.
- Compared our system with the work of others, identifying the trade-offs involved in each approach to the abstraction problem.

1.7 Conventions

In this thesis, several context-free grammars are presented. Table 1.1 explains the notation used in our grammars. A symbol appears in one of three possible fonts, depending on its type: Non-terminal symbols, such as *Statment* appear in italics, parameterized terminal symbols, such as StringLit appear in a sans-serif font, and terminal keyword symbols

Notation	Meaning
Nonterm	When appearing in a grammar, *italic* font indicates a non-terminal symbol.
Term	When appearing in a grammar, *slanted sans-serif* font indicates a parameterized terminal symbol, such as a literal or identifier.
`Keyword`	When appearing in a grammar, `teletype` font indicates a terminal keyword or punctuation symbol.
$lhs \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_k$.	A production in which the non-terminal symbol lhs can be replaced by any member of $\{\beta_1, \beta_2, \ldots \beta_k\}$.
X^*	Any member of the Kleene-closure of X. In other words, zero or more copies of X concatenated together.
X^+	One or more copies of X concatenated together (equivalent to XX^*).
ϵ	The empty string.

Table 1.1: Grammar-related notation used in this thesis.

with no associated value, such as `while`, are in a teletype font. The rest of the notation is fairly standard.

Also, three basic fonts are used in this thesis. The first is the regular font, the font in which this sentence is typeset. The second is a `teletype` font. Teletype font is used for names of computer programs, such as `gcc`, for C expressions and statements, such as `int *x = 2`, or for any other computer inputs or outputs. The third font is *italic* font. Italic font is used to emphasize words, and in particular when a new term is introduced for the first time it generally appears in italics.

Finally, there is a fair amount of mathematical notation used to describe the semantics of our system, but its introduction is delayed until Chapter 3 (specifically, see tables 3.1 and 3.2.

1.8 Thesis Organization

This thesis is organized into seven chapters, as follows:

Chapter One: Introduction, which you are reading now, contains the motivation and goals of the research embodied in this thesis.

Chapter Two: Design describes the requirements and architecture of *C Wolf*, a system to generate abstract labeled transition systems from C programs.

Chapter Three: Theory contains the theory of how to generate an abstract labeled transition system from a C program.

Chapter Four: Implementation describes the implementation of C Wolf.

Chapter Five: Related Work compares the model generation approach of this thesis with that of other projects.

Chapter Six: Experience is a case study in which the utility of C Wolf is examined.

Chapter Seven: Conclusion summarizes this thesis and provides a road map for future research.

Chapter 2

Design

Our goal is to generate a labeled transition system from a C program and an abstraction mapping which models every possible execution path under the semantics implied by the abstraction. A straightforward way of doing this is to define an *abstract virtual machine* (AVM), which is capable of executing abstracted C programs. We can then construct an initial state q_0, apply the AVM to q_0 to get a set of successor states $succ(q_0)$, apply the AVM to each element of $succ(q_0)$, and so on, until a fixpoint is reached (i.e., the set of states stops growing). Note that due to the loss of information in abstraction, it's possible for a state to have more than one successor, since we may not be able to determine what the outcome of a conditional branch will be. This is not the case in the concrete world, where every condition can be precisely evaluated.

We have decided to name our system *C Wolf*, in honor of the lupine mascots at both North Carolina State University and the State University of New York at Stony Brook. While the AVM will be at the heart of the C Wolf system, there will be several other major components. First, we will need to transform a C source program into a more palatable format to ease its consumption by the AVM. To do this, we will define a high-level intermediate language, known as *C Wolf Intermediate* (CWI) format. The C Wolf system will have a front end consisting of both a compiler (`cwcc`) and linker (`cwld`), which will reduce a C program stored in separate C source files to a single CWI file, thereby respecting C's support for separate compilation.

Second, we will need to generate models. This will be done by a model generator

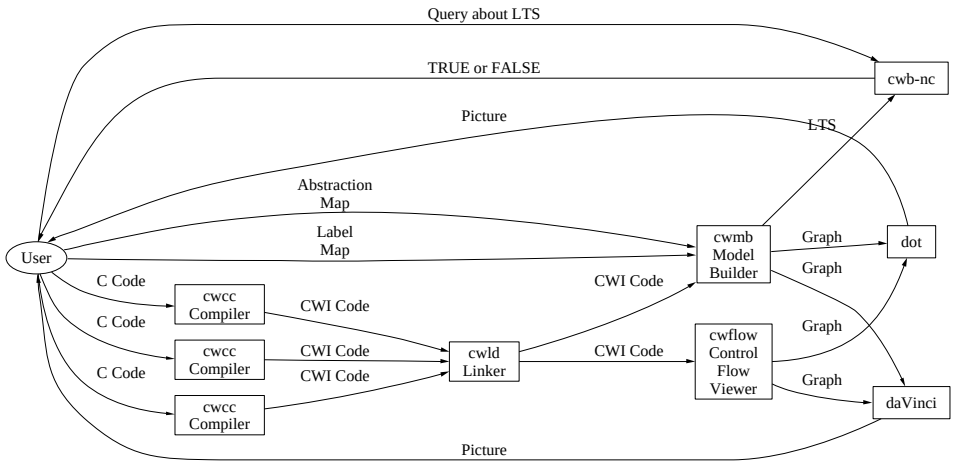

Figure 2.1: System architecture at the executable-component level.

component (`cwmb`), which will be build around the AVM and will operate on three input files. First will be a CWI file containing a compiled C program. Second will be an *abstraction map* file, which will be a text file describing what abstraction to apply to each variable in the input program. Third will be a *label map* file, which will be a text file describing how to attach labels to the transitions of the generated model. The output of the model generator will be either a graph with labeled states and edges for viewing by the user, or an LTS in a format readable by the Concurrency Workbench of the New Century (CWB-NC). To render graphs into graphical format, we will leverage two freely available graph-drawing tools, `dot` [50] and `daVinci` [33, 32].

A final, minor, component will be a CWI file control-flow graph viewer (`cwflow`), which will produce a control-flow graph in graphical format. A high-level view of the system architecture is shown in figure 2.1.

2.1 The Compiler

Our compiler, `cwcc`, accepts a C program as input and outputs a CWI file. The architecture of the compiler is shown in figure 2.2 We are using two open-source components to implement the front-end of the compiler. The preprocessing phase, in which preprocessor

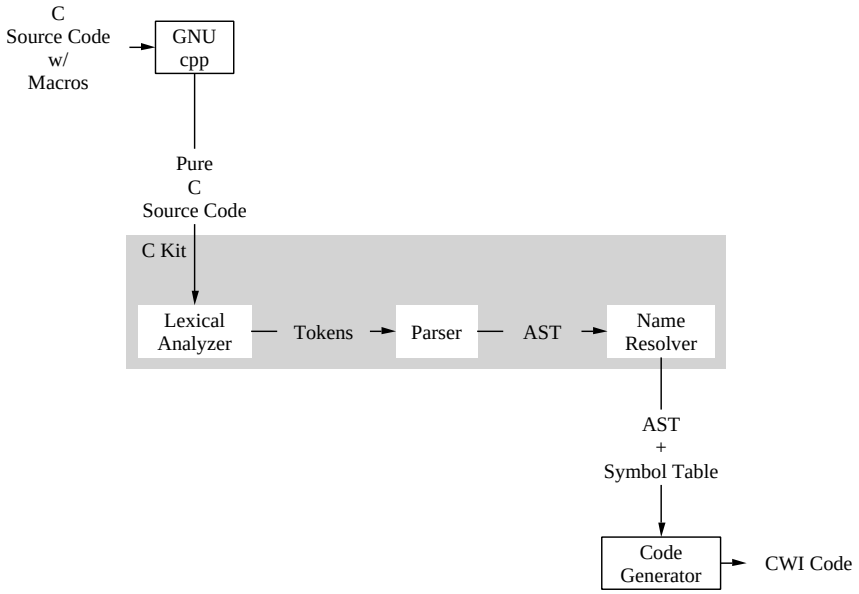

Figure 2.2: Compiler architecture.

directives such as `#define` and `#include` are handled, will be performed by `gcc` the GNU C Compiler [63]. The lexical analysis, parsing, and symbol binding phases will be done by `ckit` [51], a C front-end developed by Lucent Technologies. The code generator will iterate over abstract syntax trees (ASTs) and output CWI code.

2.1.1 Input Restrictions

While the low-level features of C make it useful for systems software development, these same features present a challenge to the task of verification. Pointers allow direct manipulation of a program's memory and forces the user to directly manage the memory.

The first major problem is *aliasing*. In C, it's possible for a pointer `p` to refer to the storage location of a variable `x`. When `*p` (the location pointed-to by `p`) is assigned to, the value of `x` also changes, since they share the same storage location. Hence, `*p` is an *alias* for `x`.

Our plan is to deal only with cases where one static or automatic variable aliases another, and the relationship can be determined during model-generation. Cases where a

pointer stored in dynamic memory (or a pointer which "passed through" dynamic memory) aliases a variable are not allowed. Inadvertent aliasing, such as caused by a *buffer overflow* (writing to an array with an invalid index, causing a nearby variable to be overwritten), may not be detected. Other tools are available which may catch such cases, and these tools can be used in combination with C Wolf.

A related problem is that of *function pointers*. These will be treated similarly; if the model-generator can determine the target of a function pointer at the point of the function call, then model generation will succeed. If not, then model generation will fail (we anticipate that this won't happen often, if ever).

A second potential problem is *recursion*. It's possible that a recursive function call will cause the program stack to grow in an unbounded fashion. If, during model generation, a bound on the recursion cannot be determined, the resulting model will be infinite-state.

We will deal with recursion by generating a warning message. If the user supplies an abstraction which forces the recursion to terminate, then the model generation will succeed. Otherwise, the model generation will fail when memory is exhausted. We will also aid the situation by optimizing all tail calls into gotos, so that tail-recursive functions will not be a problem.

A third major problem is dynamic memory. Data structures stored in dynamic memory, such as linked lists, typically have an unbounded size, and hence require a sophisticated abstraction which captures the *shape* of the structure rather than abstracting every single element of the structure. For the time being, we are abstracting away all values stored in dynamic memory. This shouldn't be too much of a problem for the types of systems we are interested in analyzing, since they typically avoid dynamic memory use, and store critical control variables in static or automatic memory.

Our model does not include concurrency, so the use of threads is not allowed. Models can be extracted from concurrent programs by slicing them by hand into sequential components and combining them using the parallel composition feature of the Concurrency Workbench, which will unfortunately require a significant amount of work by the user.

`signal` and `longjmp` are not currently supported since they interrupt the flow of control. Adding full support for signals is problematic because any signal could arrive at any time. The `setjmp` and `longjmp` system calls are actually fairly easy to support, but this work has been postponed into the future.

Finally, there are some C constructs which aren't supported due to lack of resources

to implement them. In particular the `struct` and `union` data structures aren't supported, so variables of these types are abstracted away. Support for them could easily be added in the future.

2.1.2 CWI Code

CWI code is a bytecode at a high-level of abstraction. The CWI file itself is a binary file consisting of several sections. These are:

- Source Location Information
- Type Information
- Variable Information
- Function Information
- Bytecode

The four "information" sections are essentially the symbol table plus debugging information for the program. The exact format of these sections is a detail we don't want to concern ourselves with yet; all we need to know is that the bytecode will refer to types, variables, and functions by unique ID numbers, so that no name resolution is required to interpret the bytecode.

The bytecode itself is grouped into *basic blocks*. Each basic block is a linear sequence of statements. All statements except for the last statement in a basic block are *simple assignments* in which a pure expression (i.e., an expression with no side-effects) is stored to a single variable or register. The last statement in a basic block is a *branch statement*, which refers to one or more other blocks as potential destinations. The possible branch statements are *function call, goto, if-then-else,* and *switch*. Note that there is no way for a statement to branch into the middle of a basic block; hence, when execution of a block begins, the flow of execution is guaranteed to continue in a linear fashion until the end of the block is reached. Figure 2.3 gives a grammar for the CWI bytecode.

```
Program  →  Block+
Block    →  BlockId : { Assign+ Branch }
Assign   →  Id := Expr ;
Id       →  VarId | Reg
Expr     →  BinOp ( Expr , Expr )
         |  Unop ( Expr )
         |  StringLit
         |  IntLit
         |  FloatLit
         |  VarId
         |  Reg
BinOp    →  + | * | / | - | mod
         |  > | >= | == | <= | < | <>
         |  && | || | & | | | ^ | << | >>
UnOp     →  + | - | !! | ! | & | * | ++ | --
Branch   →  goto BlockId ;
         |  if Expr then goto BlockId else goto BlockId ;
         |  call FunId ( Expr* ) ;
         |  switch ( Expr ) { Case+ } ;
         |  halt ;
Case     →  case Constant => BlockId ;
         |  default => BlockId ;
```

Figure 2.3: Grammar describing the CWI bytecode language.

2.2 The Linker

`Cwld`, our linker, accepts one or more CWI files as input and combines them into a single CWI file. Global variables and functions are unified, according to the rules of the C language [38]. The linker will also be *incremental*, meaning that not every file in the entire program needs to be linked simultaneously.

2.3 The Model Builder

`Cwmb`, the C Wolf model builder, accepts a CWI file as input, plus two source files. The first file, the *abstraction map*, specifies what abstractions to apply to each variable. The second file, the *label map*, specifies how to label transitions.

Our plan for abstraction is to focus exclusively on integer variables. The reason

Syntax	**Description**
`top`	All values are abstracted to a single value $\top$.
`minmax`	Values are abstracted to an upper and lower bound $\langle l, h \rangle$.
`mod(`k`)`	Values are abstracted to a set of remainders modulo k.
`part(`$a_1, a_2, \ldots a_k$`)`	Values are abstracted to a partition of the set of integers. The partition is $[-\infty \ldots a_1 - 1], [a_1 \ldots a_2 - 1], \ldots [a_{k-1} \ldots a_k - 1], [a_k \ldots \infty]$.
`int`	The value is either a precise integer or $\top$.
τ `array`	The value is an array of values of abstracted type τ, where $\tau \in \{$`int`, `bool`, `pointer`, `part`$(a_1, a_2, \ldots a_k)\}$.
`free`	The abstraction is chosen dynamically (assuming the abstraction of the latest assigned value).

Table 2.1: The set of available abstractions.

for this is that in embedded systems and communications protocols, the variables that embody the high-level state of the system are almost always integers. For example, a program implementing an extended finite state machine (EFSM) might use an enumeration to represent the control states of the EFSM. The abstraction map will be used to specify how each integer variable in the program should be abstracted.

The C Wolf system provides a set of predefined abstractions. Although the set of abstractions is fixed, most of them are parameterized, increasing their flexibility. The syntactic terms for the available abstractions are `top, mod, part, minmax`, and `free`. They are listed in table 2.1.

The simplest abstraction is `top`. Applying `top` to a variable `x` is equivalent to "abstracting `x` away." No information is maintained about the state of `x`. This means that when `x` appears in an expression, the expression must be evaluated based on the knowledge that the concrete value of `x` could be any possible integer value. This type of abstraction is sometimes called a *point abstraction* in the research literature. `top` is also the default abstraction for any variable without a user-specified abstraction.

The parameterized abstraction `mod` maintains a set of remainders under a divisor k. It's main use is for array indices. For example, given a variable `x` whose abstract type is `mod(4)` and whose abstract value is $\{2, 3\}$, the concrete value of `x` could be any value in the set $\{2 + 4 \cdot i \mid i \geq 0\} \cup \{3 + 4 \cdot i \mid i \geq 0\} = \{2, 3, 6, 7, 10, 11, \ldots\}$.

The parameterized abstraction `part` partitions the integers into a set of disjoint intervals which cover the set of integers. `part` is the workhorse abstraction, which we

expect will see the most use. `part` can be used to isolate particular values of interest. For example, if we are interested in following the values 3, 4, and 5, but don't care about other values, the abstraction `part(3,4,5,6)` will suffice. If the variable `x` has abstract type `part(3,4,5,6)` and abstract value $\{[4,4],[6,\infty]\}$, then its concrete type could be any value in the set $\{4,6,7,8,\ldots\}$.

The abstractions `int`, `minmax` and `free` are risky because they may result in huge number of states. Their primary intended use is internal. However, we have left them available to the user for experimental purposes, to be used at one's own risk.

The `minmax` abstraction maintains an upper and lower bound on a variable's set of possible values. For example, a `minmax` variable whose abstract value was $\langle 5,10\rangle$ would have a concrete value in the set $\{5,6,7,\ldots 10\}$. This abstraction is dangerous because it's too precise. Consider the *for* loop:

```
for(x = 0; x < 1000; ++x) {
     ⋮
}
```

If the `minmax` abstraction is applied to `x`, then the abstract value of `x` will follow the sequence $\langle 0,0\rangle, \langle 1,1\rangle, \langle 2,2\rangle \ldots \langle 1001,1001\rangle$, resulting in no reduction in the number of states from the concrete program.

On the other hand, for variables which aren't used as loop indices, `minmax` may provide useful results. `minmax` is also used internally to evaluate relational expressions.

The `int` abstraction tells the system to compute a precise value where possible, but if not possible, fall back to $\top$. It can be useful where a precise value of a particular variable is needed, but shares the same dangers as `minmax`.

The final abstraction is `free`, which is not a new abstract type, but is instead a directive indicating that the abstract type of the variable should be dynamic. Whenever a variable with type `free` is assigned-to, the type of the value resulting from the expression evaluation is left unchanged. `free` is just as dangerous as `minmax` and will result in the same sort of state-explosion if applied to a loop index variable. However, for a small program, simply running the program with the abstraction `free` applied to all variables may yield useful results. Also, the `free` abstraction is currently the only way to capture pointer values, although we eventually plan to provide specific pointer abstractions.

The `array` abstraction is not really an abstraction, but an array of elements sharing

$$
\begin{array}{lcl}
AbsMap & \rightarrow & TopDecl^* \\
TopDecl & \rightarrow & fileDecl \mid libDecl \mid typeDecl \\
fileDecl & \rightarrow & \texttt{file}\ \textit{stringLit}\ \texttt{\{}\ objDecl^*\ \texttt{\}} \\
libDecl & \rightarrow & \texttt{lib}\ \textit{stringLit}\ \texttt{;} \\
typeDecl & \rightarrow & \texttt{type}\ \textit{id}\ \texttt{=}\ objType\ \texttt{;} \\
objDecl & \rightarrow & varDecl \mid funDecl \\
varDecl & \rightarrow & \texttt{var}\ \textit{id}\ \texttt{:}\ objType\ \texttt{;} \\
funDecl & \rightarrow & \texttt{fun}\ \textit{id}\ \texttt{(}\ varDecl^*\ \texttt{)}\ \texttt{:}\ objType\ \texttt{\{}\ varDecl^*\ \texttt{\}} \\
objType & \rightarrow & \textit{id} \mid absType \\
absType & \rightarrow & \texttt{top} \mid \texttt{mod(}\ intList\ \texttt{)} \mid \texttt{minmax} \mid \texttt{part(}\ intList\ \texttt{)} \mid \texttt{free} \\
intList & \rightarrow & intList' \mid \epsilon \\
intList' & \rightarrow & \textit{intLit} \mid \textit{intLit}\ \texttt{,}\ intList'
\end{array}
$$

Figure 2.4: Grammar for abstraction map files.

the same abstraction. Support for arrays was added at the 11th hour in order to support modeling of communications protocols. The elements can be one of `int`, `part`, `bool`, or `pointer`. The `int` and `part` abstractions are the same as for scalar integer values. The `bool` abstraction maps each array element onto the set $\{true, false, maybe\}$. The final element type is `pointer`, which tracks what variable or array cell each element of the array points to. Finally, the model builder must be able to infer the size of the array, otherwise, the entire array contents are abstracted to $\top$. Users of the array abstraction should also be aware that it can lead to explosions in the size of the resulting model unless used extremely judiciously.

2.3.1 The Abstraction Map

The mapping of abstractions unto variables is specified by an *abstraction map* file. The grammar for abstraction maps is shown in figure 2.4. The top-level declarations in the abstraction map define libraries, types, and file scopes.

A *library declaration* has the form `lib "`*filename*`"`. In such a declaration, *filename* refers to an external abstraction map file which is to be processed. This allows for multiple files to be combined into a single abstraction map.

A *type declaration* has the form `type` *id* `=` *synType*, where *id* is an identifier, and *synType* is a syntactic type. The purpose of the declaration is to make *id* a shorthand name

```
LabelMap  →  TopDecl*
TopDecl   →  event => id ;
event     →  call ( fun )
          |  read ( var )
          |  write ( var )
          |  watch ( var relOp intLit )
          |  exec line

fun       →  stringLit : id | id
var       →  fun : id | line : id | id
line      →  stringLit : intLit
relOp     →  == | != | <= | < | > | >=
```

Figure 2.5: Grammar for label map files.

for *synType*. Within the scope of the declaration, *id* can be used anywhere a syntactic type is required.

A *file declaration* has the form `file "`*filename*`" { ... }`. *filename* is the name of a C source file. The curly braces enclose a *file scope*. Within the file scope are declarations defining what the (abstract) type of each variable are. This is achieved through the use of two kinds of declaration, one for variables and one for functions. All variables and functions declared within the file scope must also be defined in the C source file $filename$.

A *function declaration* creates a *function scope*. All variables declared within the function scope are assumed to be local variables of the function whose name is given in the `fun` declaration.

A *variable declaration* `var x : A;` declares `x` to have abstract type `A`. Which `x` is referred to depends on the context of the declaration. There are two cases:

- Within a `fun f` declaration, the `x` referred to is a local variable of `f`.
- Outside of any `fun` declaration, the `x` referred to is a static variable of the file whose name is specified by the enclosing `file` declaration.

2.4 The Label Map

In order to generate a labeled transition system, there must be some rules for attaching labels to the transitions. This is left to the user to specify, because the labeling rules will depend on what properties the user ultimately wishes to verify.

The labeling specification is in the form of a *label map* file. The grammar for label map files appears in figure 2.5. The label map file associates labels with *events*. An event

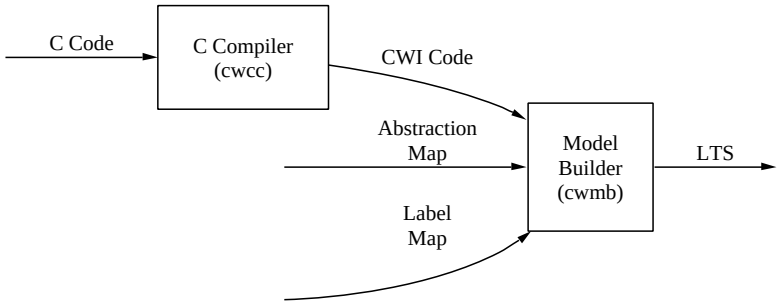

Figure 2.6: Simplified view of core system architecture.

occurs when a specific variable is read or written, a specific function is called, a specified line of code is executed, or a specified condition becomes true.

Typical uses of labeling might be to

- to associate a different label with each `case` of a `switch` statement performed on a control-state variable.
- to associate labels with certain system calls, such as semaphore operations.
- to generate a label every time a shared variable is read/written (with different labels for read and write).

2.5 Graphics

C Wolf provides for visualization at both the CWI and LTS level. This is achieved by leveraging two open-source graph drawing tools, `dot` [50] and `daVinci` [33, 32]. From CWI files, call graphs and control-flow graphs can be generated. `cwmb` can generated graphs representing LTSes or detailed Kripke Structures. The user can graphically browse any of these graphs, or get a postscript rendering of them.

2.6 Summary

The C Wolf system has two essential components. `cwcc` generates CWI bytecode files from C source files. `cwmb` generates an LTS in `cwb-nc` format from a CWI file, an abstraction map and a label map file. The abstraction map specifies what abstraction to apply to each program variable, and the label map causes labels to be associated with guards that are triggered by an event occurring in a program region while a (optional) condition is satisfied.

While most C programs can be handled, there are some significant restrictions. These include limited use of aliasing, no concurrency, and no use of `longjmp`.

A simplified system architecture reflecting this view of `cwolf` appears as figure 2.6.

Chapter 3

Theory

How does one build a labeled transition system from a C program rearranged into CWI form? The short answer to the question is *abstract interpretation* [19, 20, 21]. Abstract interpretation is a framework for executing a program, using values from an abstract domain in place of the original concrete values. In this chapter, we will establish an interpretation of C programs using the abstractions presented in chapter 2. We will further prove that the interpretation produces *sound* results, in the sense that every possible abstract result includes all possible corresponding concrete results (We'll clarify exactly what is meant by soundness later). Finally, an algorithm for generating a labeled transition system will be presented.

3.1 Preliminaries

Before we delve into the details of abstract interpretation, we need to precisely define some of the structures we will be dealing with. First, one of the main structures we will be dealing with is the *labeled transition system* (LTS), which is a state machine with labels on the transitions between states. We will extend the LTS definition to include a designated start state. Furthermore, since we want our transitions to be compatible with process algebras, we will use a notion of actions based upon Milner's Calculus of Communicating Systems [56], in which every action a has a complimentary action $\overline{a}$, and

Notation	Meaning
$q \xrightarrow{a} p$	a transition from p to q with label a.
$q \rightarrow p$	equivalent to $q \xrightarrow{\tau} p$.
$q \stackrel{a}{\Rightarrow} p$	there is a path from q to p where every transition has the label τ, except for exactly one transition which is labeled with a.
$q \Rightarrow p$	equivalent to $q \stackrel{\tau}{\Rightarrow} p$
$succ(q)$	the set $\{p \mid q \xrightarrow{a} p$ for any label $a\}$.
$succ_a(q)$	the set $\{p \mid q \xrightarrow{a} p\}$.
$pred(q)$	the set $\{p \mid p \xrightarrow{a} q$ for any label $a\}$.
$succ_a(q)$	the set $\{p \mid p \xrightarrow{a} q\}$.

Table 3.1: LTS-related notation.

in addition, there is a special silent transition, τ.

Definition 3.1.1 *A* labeled transition system *(LTS) is a tuple* $\langle Q, q_0, Tr, Act, \mathcal{L} \rangle$ *where:*

- Q *is a finite set of* ***states****.*
- $q_0 \in Q$ *is a distinguished* ***start state****.*
- $Tr \subseteq Q \times Q$ *is a set of* ***transitions****.*
- Act *is a set of* ***actions****.*
- $\mathcal{L} : Tr \rightarrow Act \cup \{\overline{a} \mid a \in Act\} \cup \{\tau\}$ *is a* ***labeling function****.*

We will use fairly standard notation to represent transitions, paths, and LTS-related concepts. This notation appears in Table 3.1. Some additional set-theoretical notation appears in table 3.2.

3.2 Abstract Interpretation

Abstract interpretation [19, 20, 21] provides a framework for evaluating a program in an abstract domain. The basic requirements of an abstract interpretation is that there

Notation	Meaning
$\alpha_{_A}$	Abstraction function for abstract domain A.
$\gamma_{_A}$	Concretization function for abstract domain A.
$+_{_A}$	Addition function for abstract domain A.
$\|S\|$	Cardinality of S.
$\mathcal{P}(S)$	Power set of S.
$\sqcup S$	Least upper bound of S
$x \sqcup y$	Least upper bound of $\{x, y\}$
$\sqcap S$	Greatest lower bound of S
$x \sqcap y$	Greatest upper bound of $\{x, y\}$
$x \stackrel{\text{def}}{=} y$	x is defined as y

Table 3.2: Set-related notation.

be two established "worlds" in which computation occurs. The first is the original, *concrete* world, whose semantics are defined by the C language definition [38]. The second world is the *abstract* world, a lattice of symbols. These worlds are connected by two functions. The abstraction function, α, maps from a set of concrete elements to an abstract element. The concretization function, γ, maps from an abstract element to a set of concrete elements. Furthermore, each domain has the same set of operators, so that we can perform exactly the same computations in either the concrete or abstract domain.

We are now ready to introduce our definition of an abstract interpretation. Our definition is slightly nonstandard, in that we have predefined the $\sqsubseteq$ relation in terms of γ. Otherwise, it is similar to Cousot's.

Definition 3.2.1 *Given a set* $\mathcal{C}$ *of* concrete values*, and a set* $O_\gamma : \mathcal{C}^k \to \mathcal{C}$ *of* concrete operators *(each operator can have a different arity), an* **abstract interpretation** *of* $\langle \mathcal{C}, O_\gamma \rangle$ *is a tuple* $\langle \mathcal{A}, \sqsubseteq, \top, \bot, \alpha, \gamma, O_\alpha \rangle$ *where*

- $\langle \mathcal{A}, \sqsubseteq, \top, \bot \rangle$ *is a lattice of* ***abstract values***
- *an* ***abstraction function*** $\alpha : \mathcal{P}(\mathcal{C}) \to \mathcal{A}$
- *a* ***concretization function*** $\gamma : \mathcal{A} \to \mathcal{P}(\mathcal{C})$

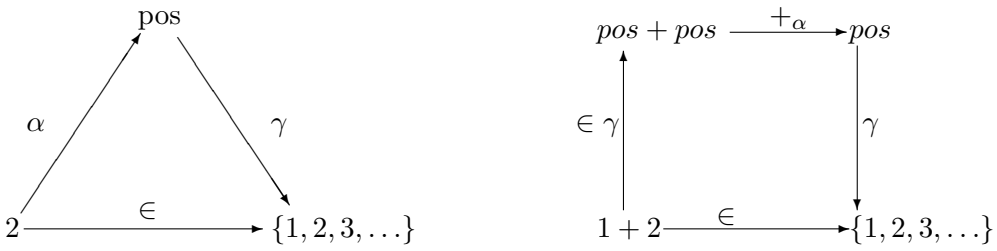

Figure 3.1: Illustration of soundness properties under the abstraction $\{pos, zero, neg\}$.

- *a set* $O_\alpha : \mathcal{A}^k \to \mathcal{A}$ *of* ***abstract operators****; each operator* $\circ_\gamma \in O_\gamma$ *has a corresponding operator* $\circ_\alpha \in O_\alpha$ *of the same arity and vice-versa.*

- *the* ***abstract partial order*** $\sqsubseteq$ *is defined as* $a_1 \sqsubseteq a_2$ *if and only if* $\gamma(a_1) \subseteq \gamma(a_2)$*.*

- $\top \in \mathcal{A}$ *is the* ***top element*** *of* $\mathcal{A}$*, and* $\gamma(\top) = \mathcal{C}$

- $\bot \in \mathcal{A}$ *is the* ***bottom element*** *of* $\mathcal{A}$*, and* $\gamma(\bot) = \varnothing$

Given an abstract interpretation, we need to establish *soundness conditions.* The basic intuition behind soundness, is if we start we a concrete value, use α to move the abstract world, then perform zero or more operations in the abstract world, and finally return to the concrete world, the resulting set of concrete values includes the result from applying the same sequence of operations to the original concrete values. This is illustrated by figure 3.1. For purposes of clarity, we have split this notion into three cases, one for the case where no operation occurs, one for unary operators, and one for binary operators.

Definition 3.2.2 *An abstraction function* $\alpha : \mathcal{P}(\mathcal{C}) \to \mathcal{A}$ *and its dual concretization function* $\gamma : \mathcal{A} \to \mathcal{P}(\mathcal{C})$ *are* ***mutually sound*** *if and only if:*

$$\gamma(\alpha(S)) \subseteq S \text{ for all } S \subseteq \mathcal{C}$$

The definition for unary functions is as follows:

Definition 3.2.3 *A unary abstract operator* $\circ_\alpha : \mathcal{A} \rightarrow \mathcal{A}$ *is* ***sound with respect to its concrete dual*** $\circ_\gamma : \mathcal{C} \rightarrow \mathcal{C}$ *if the following condition holds true:*

$$\textit{for any } a \in \mathcal{A},\ \textit{for any } x \in \gamma(\mathcal{A}), \quad \circ_\gamma(x) \in \gamma(\circ_\alpha(\mathcal{A}))$$

The generalization to binary functions is as follows:

Definition 3.2.4 *A binary abstract operator* $\circ_\alpha : \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A}$ *is* ***sound with respect to its concrete dual*** $\circ_\gamma : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$ *if the following condition holds true:*

$$\textit{for each } a \in \mathcal{A},\ \textit{for each } b \in \mathcal{A}, \quad x \in \gamma(a)\ \textit{and}\ y \in \gamma(b) \ \Rightarrow\ x \circ_\gamma y \in \gamma(a \circ_\alpha b)$$

The abstract interpretation we wish to use will involve a mix of abstractions, but all these abstractions will be linked into a common lattice, which is technically feasible because we have defined the partial order of our lattice to be $x \sqsubseteq y \ \Leftrightarrow\ \gamma(x) \subseteq \gamma(y)$. The set of abstractions is discussed in more detail in section 3.5.

3.3 The CWI Language

The source language of our abstract interpretation scheme is the CWI language, which was briefly presented in section 2.1.2, and whose grammar appears in figure 2.3. CWI has assignment statements, conditional branching, case statements, goto statements, and function calls. No side effects are allowed in the right-hand side of an expression (e.g., the traditional `++` and `--` operators are provided, but their semantics has been changed to remove the side-effect). Also forbidden are function calls embedded within expressions. Functions are called using a special statement, and the result of the call is stored in a temporary register (register number 0). While and for loops are not provided, and are instead implemented using conditional branching. These restrictions make it easy to organize the

```
int f(const int k)
{
  int sum = 0;
  for (i = 0; i <= k; ++i)
    sum += i * 5;
  return sum;
}
```

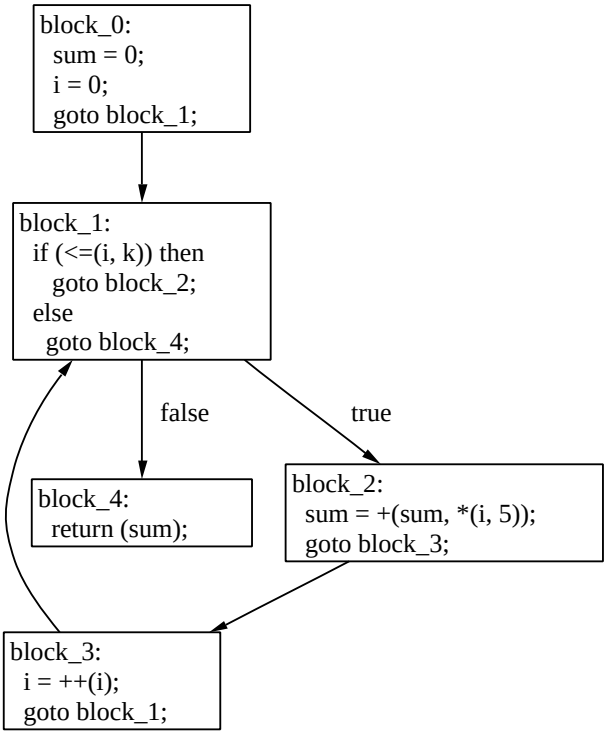

Figure 3.2: A small function and its basic block representation.

code into basic blocks. A basic block is a sequence of zero or more assignment statements followed by a branching statement of some sort (goto, conditional goto, call, or switch), with the additional requirement that only the first statement of the block can be the target of any branching statements. The advantage of this representation is that we can view the effect of each basic block atomically, effectively reducing the number of values the "program counter" can assume, resulting in a state space reduction. Figure 3.2 illustrates a small piece of C code, along with the corresponding CWI basic block representation.

3.4 Concrete Domains

The primary concrete domain we are interested in is the integers. In the C language, the set of signed integer values is fixed and bounded by two constants, *maxint*, and *minint*. On a 32-bit, 2's-complement architecture, *maxint* is $2^{31} - 1$ and *minint* is -2^{31}.

Name	Symbol	Description
Unity	U	A single abstract value.
Trilog	B_3	True, false, or maybe.
Range	R	All values are abstracted to a pair $\langle x, y \rangle$ which bounds the possible concrete values to the closed interval $[x, y]$.
Mod k	M_k	Concrete values abstracted to their remainder when divided by k.
Interval S	$\mathcal{I}_S$	$\mathbb{Z}$ is broken up into contiguous intervals specified by the set of pairs S, which are disjoint, but whose union covers every value in $\mathbb{Z}$. Each abstract value is some subset of S.

Table 3.3: Summary of abstract integer domains provided by the C Wolf system.

We will use $\mathbb{Z}$ to refer to the domain of C signed integers $\{x \mid minint \leq x \leq maxint\}$. Furthermore, some of the operators will exploit the $2's$ compliment architecture to improve their accuracy. These operators will need to be reimplemented for other architectures.

In C, there are two kinds of integer expressions, *boolean* and *numeric*. Numeric expressions, are ones whose result is any value in $\mathbb{Z}$, including bit-level operations. Boolean expressions, which include operators such as "logical and" (`||`) and "logical or" (`&&`), operate over the domain $\{0, 1\}$. We use the symbol B to represent this domain, with the understanding that it is just a subset of $\mathbb{Z}$. During expression evaluation, C converts back and forth between $\mathbb{Z}$ and B as needed, by applying the following conversion functions:

$$\begin{aligned} f_{\mathbb{Z}\to B}(x) &\stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } x \neq 0 \\ 0 & \text{if } x = 0 \end{cases} \\ f_{B\to\mathbb{Z}}(x) &\stackrel{\text{def}}{=} x \end{aligned} \tag{3.1}$$

In the following we will use the same function names $f_{\mathbb{Z}\to B}$ and $f_{B\to\mathbb{Z}}$ to denote its natural extension to sets of integers and booleans, respectively.

3.5 Abstract Domains

There are five basic abstract domains in our system. They are summarized in table 3.3. The simplest of these, *Unity* abstracts all values to a single value, and is the coarsest possible abstraction. The Unity abstraction is also known as "abstracting away" or a "point abstraction." The next simplest is *Trilog*, which is the domain of three-valued

Domain	**Elements**	**Top**	**Bottom**
Unity	$\{\top, \bot\}$	$\top$	$\bot$
Trilog	$\mathcal{P}(\{\mathit{tt}, \mathit{ff}\})$	$\{\mathit{tt}, \mathit{ff}\}$	$\varnothing$
Mod k	$\mathcal{P}(\{x \mid 1-k \leq x \leq k-1\})$	$\{x \mid 1-k \leq x \leq k-1\}$	$\varnothing$
Range	$\{\bot\} \cup \{\langle x, y\rangle \in \mathbb{Z} \times \mathbb{Z} \mid x \leq y\}$	$\langle \mathit{minint}, \mathit{maxint}\rangle$	$\bot$
Interval S	$\mathcal{P}(S)$	S	$\varnothing$

Table 3.4: Elements of each abstract integer domain in the C Wolf system.

Domain	**Abstraction function**		
Unity	$\alpha_U(S)$	$\stackrel{\text{def}}{=}$	$\begin{cases} \top & \text{if } S \neq \varnothing \\ \bot & \text{if } S = \varnothing \end{cases}$
Trilog	$\alpha_{B_3}(S)$	$\stackrel{\text{def}}{=}$	$\{b(x) \mid x \in S\}$ where $b(x) = \begin{cases} \mathit{tt} & \text{if } x \neq 0 \\ \mathit{ff} & \text{if } x = 0 \end{cases}$
Mod k	$\alpha_{Mod_k}(S)$	$\stackrel{\text{def}}{=}$	$\{x \bmod k \mid x \in S\}$
Range	$\alpha_R(S)$	$\stackrel{\text{def}}{=}$	$\begin{cases} \langle \sqcap S, \sqcup S\rangle & \text{if } S \neq \varnothing \\ \bot & \text{if } S = \varnothing \end{cases}$
Interval S	$\alpha_{\mathcal{I}_S}(T)$	$\stackrel{\text{def}}{=}$	$\{\langle l, h\rangle \in S \mid \exists x \in T \,.\, l \leq x \leq h\}$

Table 3.5: Abstraction functions for each abstract integer domain.

logic and is used in the evaluation of boolean expressions. The third domain is the *Range* domain, in which concrete values are abstracted to an upper and lower bound. The fourth domain, *Mod k*, is a parameterized family of domains in which values are abstracted to a set of possible remainders under division by the integer parameter k. Hence, the $Mod\ 2$ domain abstracts variables based on whether they are known to be odd or even, or no information on the odd/even status is available.

The final domain family is *Interval S*. Given a user-specified set S, which breaks the set of integers into subranges, an Interval S abstract value tracks which of the intervals in S the concrete values might belong to. For example, the domain

$$\mathit{Interval}\ \{\langle \mathit{minint}, -1\rangle, \langle 0, 0\rangle, \langle 1, \mathit{maxint}\rangle\}$$

abstracts variables by their sign.

Table 3.4 contains the formal definitions of the elements within each domain. As required by definition 3.2.1, each domain contains a top and bottom element. The top

Domain	**Concretization function**		
Unity	$\gamma_U(x)$	$\stackrel{\text{def}}{=}$	$\begin{cases} \mathbb{Z} & \text{if } x = \top \\ \varnothing & \text{if } x = \bot \end{cases}$
Trilog	$\gamma_{B_3}(S)$	$\stackrel{\text{def}}{=}$	$\begin{cases} \{0\} & \text{if } S = \{\mathit{ff}\} \\ \{1\} & \text{if } S = \{\mathit{tt}\} \\ \{0,1\} & \text{if } S = \{\mathit{tt}, \mathit{ff}\} \end{cases}$
Range	$\gamma_R(x,y)$	$\stackrel{\text{def}}{=}$	$\{z \mid x \leq z \leq y\}$
Mod_k	$\gamma_{Mod_k}(S)$	$\stackrel{\text{def}}{=}$	$\bigcup \begin{array}{l} \{k \cdot i + x \mid x \in S \text{ and } x \geq 0 \text{ and } i \geq 0\} \\ \{k \cdot i + x \mid x \in S \text{ and } x \leq 0 \text{ and } i \leq 0\} \end{array}$
Interval_S	$\gamma_{\mathcal{I}_S}(T)$	$\stackrel{\text{def}}{=}$	$\bigcup_{\langle l,h\rangle \in T} \{x \mid l \leq x \leq h\}$

Table 3.6: Concretization functions for each abstract integer domain.

element is the unique element whose value under application of γ is $\mathbb{Z}$, and the bottom element is the value whose γ-value is $\varnothing$. For domains which don't naturally have a top or bottom, an explicit symbol is added, such as the $\bot$ value in the Range domain. In general, we'll use the symbols $\top$ and $\bot$ for every domain, and the translation to the appropriate representation will be implicitly understood.

Table 3.5 defines the abstraction functions for each domain, and table 3.5 defines the corresponding concretization functions. These functions are perhaps best understood by a small example. Consider the value 5. If we compute $\alpha(\{5\})$ for each domain, then we get the following results:

$$\begin{array}{lcl} \alpha_U(\{5\}) & = & \top \\ \alpha_{B_3}(\{5\}) & = & \{\mathit{tt}\} \\ \alpha_R(\{5\}) & = & \langle 5,5\rangle \\ \alpha_{Mod_2}(\{5\}) & = & \{1\} \\ \alpha_{\mathcal{I}_S}(\{5\}) & = & \{\langle 1, maxint\rangle\} \text{ where } S = \{\langle minint, -1\rangle, \langle 0,0\rangle, \langle 1, maxint\rangle\} \end{array}$$

If we then use γ to back to the concrete world, what we end up is the following:

$$\begin{array}{lcl} \gamma_U(\alpha_U(\{5\}) & = & \{minint, minint+1, \ldots 0, 1, 2, \ldots maxint\} \\ \gamma_{B_3}(\alpha_{B_3}(\{5\})) & = & \{1\} \\ \gamma_R(\alpha_R(\{5\})) & = & \{5\} \\ \gamma_{M_2}(\alpha_{M_2}(\{5\}) & = & \{1,3,5,\ldots maxint\} \\ \gamma_{\mathcal{I}_S}(\alpha_{\mathcal{I}_S}(\{5\})) & = & \{1,2,3,\ldots maxint\} \\ & & \quad \text{where } S = \{\langle minint, -1\rangle, \langle 0,0\rangle, \langle 1, maxint\rangle\} \end{array}$$

A final remaining task is the proving of correctness of the α and γ functions for

each domain. A proof for each domain is now presented.

Theorem 3.5.1 *α_U and γ_U are mutually sound abstractions of $\mathbb{Z}$ according to definition 3.2.2.*

Proof Pick any $S \subseteq \mathbb{Z}$.

Either $S = \varnothing$ or $S \neq \varnothing$.

If $S = \varnothing$ then $S \subseteq \gamma_U(\alpha_U(S))$.

If $S \neq \varnothing$ then $\gamma_U(\alpha_U(S)) = \gamma_U(\top) = \mathbb{Z} \supseteq S$.

In both cases, $S \subseteq \gamma_U(\alpha_U(S))$. ■

Theorem 3.5.2 *α_{B_3} and γ_{B_3} are mutually sound abstractions of B according to definition 3.2.2.*

Proof Pick any $S \subseteq B = \{0, 1\}$.

Either (1) $S = \varnothing$, (2) $S = \{0\}$, (3) $S = \{1\}$, or (4) $S = \{0, 1\}$

Case (1) $S = \varnothing$: Since S is empty, $S \subseteq \gamma_{B_3}(\alpha_{B_3}(S))$.

Case (2) $S = \{0\}$: $\gamma_{B_3}(\alpha_{B_3}(S)) = \gamma_{B_3}(\alpha_{B_3}(\{0\})) = \gamma_{B_3}(\{\mathit{ff}\}) = \{0\} = S$.

Case (3) $S = \{1\}$: $\gamma_{B_3}(\alpha_{B_3}(S)) = \gamma_{B_3}(\alpha_{B_3}(\{1\})) = \gamma_{B_3}(\{\mathit{tt}\}) = \{1\} = S$.

Case (4) $S = \{0, 1\}$: $\gamma_{B_3}(\alpha_{B_3}(S)) = \gamma_{B_3}(\alpha_{B_3}(\{0, 1\})) = \gamma_{B_3}(\{\mathit{tt}, \mathit{ff}\}) = \{0, 1\} = S$.

In all cases, $S \subseteq \gamma_{B_3}(\alpha_{B_3}(S))$. ■

Theorem 3.5.3 *α_R and γ_R are mutually sound abstractions of $\mathbb{Z}$ according to definition 3.2.2.*

Proof Pick any $S \subseteq \mathbb{Z}$.

Either $S = \varnothing$ or $S \neq \varnothing$.

If $S = \varnothing$ then $S \subset \gamma_R(\alpha_R(S))$ for all values of $\gamma_R(\alpha_R(S))$

If $S \neq \varnothing$ then $\gamma_R(\alpha_R(S)) = \gamma_R(\langle \sqcap S, \sqcup S \rangle)) = \{x \mid \sqcap S \leq x \leq \sqcup x\}$. Pick any $y \in S$. From the definitions of $\sqcup$ and $\sqcap$, $\sqcap S \leq y \leq \sqcup S$. This satisfies the condition for membership in $\gamma_R(\alpha_R(S))$. Any $y \in S$ is also shown to be in $\gamma_R(\alpha_R(S))$, so $S \subseteq \gamma_R(\alpha_R(S))$.

In both cases, $S \subseteq \gamma_R(\alpha_R(S))$. ■

Theorem 3.5.4 *α_{M_k} and γ_{M_k} are mutually sound abstractions of $\mathbb{Z}$ according to definition 3.2.2.*

Proof Pick any $S \subseteq \mathbb{Z}$.

Either (1) $S = \varnothing$ or (2) $S \neq \varnothing$.

Case (1) $S = \varnothing$: $S \subset \gamma_{M_k}(\alpha_{M_k}(S))$ for all values of $\gamma_{M_k}(\alpha_{M_k}(S))$

Case (2) $S \neq \varnothing$: $\gamma_{M_k}(\alpha_{M_k}(S)) = \gamma_{M_k}(\{x \; mod \; k \mid x \in S\}) = \{x \; mod \; k + i \cdot k \mid x \in S, x \geq 0, \text{ and } i \geq 0\} \cup \{x \; mod \; k - i \cdot k \mid x \in S, x < 0, \text{ and } i \geq 0\}$. We will use T^+ to refer to $\{x \; mod \; k + i \cdot k \mid x \in S, x \geq 0, \text{ and } i \geq 0\}$, and T^- to refer $\{x \; mod \; k - i \cdot k \mid x \in S, x \geq 0, \text{ and } i \geq 0\}$. Note that $\gamma_{M_k}(\alpha_{M_k}(S)) = T^+ \cup T^-$.

Pick any $x \in S$, and we will show that x also $\in \gamma_{M_k}(\alpha_{M_k}(S))$. Either (2a) $x < 0$, (2b) $x \geq 0$.

Case (2a) $x < 0$: From the definition of *mod*, $x = (x \; mod \; k) - j \cdot k$ for some $j \geq 0$. This satisfies the condition for membership in T^-, so $x \in T^- \subset \gamma_{M_k}(\alpha_{M_k}(S))$.

Case (2b) $x \geq 0$: From the definition of *mod*, $x = (x \; mod \; k) + j \cdot k$ for some $j \geq 0$. This satisfies the condition for membership in T^+, so $x \in T^+ \subset \gamma_{M_k}(\alpha_{M_k}(S))$.

In both cases 2a and 2b, $S \subseteq \gamma_{M_k}(\alpha_{M_k}(S))$ because any $x \in S$ is also a member

of $\gamma_{M_k}(\alpha_{M_k}(S))$.

In both cases 1 and 2, $S \subseteq \gamma_{M_k}(\alpha_{M_k}(S))$. ■

Theorem 3.5.5 *$\alpha_{\mathcal{I}_S}$ and $\gamma_{\mathcal{I}_S}$ are mutually sound abstractions of $\mathbb{Z}$ according to definition 3.2.2.*

Proof Pick any $T \subseteq \mathbb{Z}$.

Either (1) $T = \varnothing$ or (2) $T \neq \varnothing$.

Case (1) $T = \varnothing$: $T \subset \gamma_{\mathcal{I}_S}(\alpha_{\mathcal{I}_S}(T))$ for all values of $\gamma_{\mathcal{I}_S}(\alpha_{\mathcal{I}_S}(T))$

Case (2) $S \neq \varnothing$: Pick any $z \in T$. We will show that $z \in \gamma_{\mathcal{I}_S}(\alpha_{\mathcal{I}_S}(T))$. Since S must cover every value in *Zed*, there exists $\langle l_i, h_i \rangle \in S$ such that $l_i \leq z \leq h_i$. From the definition of $\alpha_{\mathcal{I}_S}(T)$, $\langle l_i, h_i \rangle \in \alpha_{\mathcal{I}_S}(T)$. From the definition of $\gamma_{\mathcal{I}_S}$, $\{x \mid l_i \leq x \leq h_i\} \subseteq \gamma_{\mathcal{I}_S}(\alpha_{\mathcal{I}_S}(T))$. Since $l_i \leq z \leq h_i$, $z \in \{x \mid l_i \leq x \leq h_i\}$, and $\{x \mid l_i \leq x \leq h_i\} \subseteq \gamma_{\mathcal{I}_S}(\alpha_{\mathcal{I}_S}(T))$, so $z \in \gamma_{\mathcal{I}_S}(\alpha_{\mathcal{I}_S}(T))$.

For any $z \in T$, we have shown $z \in \gamma_{\mathcal{I}_S}(\alpha_{\mathcal{I}_S}(T))$, so $T \subseteq \gamma_{\mathcal{I}_S}(\alpha_{\mathcal{I}_S}(T))$.

In both cases (1) and (2), $T \subseteq \gamma_{\mathcal{I}_S}(\alpha_{\mathcal{I}_S}(T))$.

■

3.6 Conversions

During evaluation, it's possible that a conversion from one abstract domain to another will be required. Before we jump ahead of ourselves, however, we need to consider what the correctness condition of a conversion function is. Intuitively, a conversion should have the same correctness condition as an operator, i.e., information can become more imprecise, or mathematically, the set of values returned by γ could grow, but no values should be lost. Formally, this is defined as:

Definition 3.6.1 *Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be abstract domains of the concrete domains C_1 and C_2, respectively. A conversion function function $f_{\mathcal{D}_1 \to \mathcal{D}_2} : \mathcal{D}_1 \to \mathcal{D}_2$ is a* ***sound conversion*** *with respect to the converstion function $f_{C_1 \to C_2} : C_1 \to C_2$ if the following condition is satisfied:*

$$\textit{for all } x \in \mathcal{D}_1, \quad f_{C_1 \to C_2}(\gamma_{\mathcal{D}_1}(x)) \subseteq \gamma_{\mathcal{D}_2}(f_{\mathcal{D}_1 \to \mathcal{D}_2}(x))$$

Note that if $C_1 = C_2$ then $f_{C_1 \to C_2}$ and the constraint reduces to

$$\textit{for all } x \in \mathcal{D}_1, \quad \gamma_{\mathcal{D}_1}(x) \subseteq \gamma_{\mathcal{D}_2}(f_{\mathcal{D}_1 \to \mathcal{D}_2}(x))$$

A simple, generic way of converting between any two domains is simply to use the concretization function of the starting domain to go back to the concrete world, convert among the concrete worlds and then re-abstract using the abstraction function of our target domain. In other words, we could define:

$$f_{\mathcal{D}_1 \to \mathcal{D}_2}(x) = \alpha_{\mathcal{D}_2}(f_{C_1 \to C_2}(\gamma_{\mathcal{D}_1}(x)))$$

and have a conversion function for all domains. Unfortunately, this doesn't help our implementation very much, because it's not practical to compute the result of γ in most cases. Hence, in the subsections that follow, a specific conversion function is defined for each domain.

In the following note that there are only two concrete domains $\mathbb{Z}$ and B, and that B_3 is the only abstraction of B. We will make use of this information in simplifying our proofs.

3.6.1 Unity Conversions

The result of conversion function for the unity domain should be $\top$ for all values except $\bot$ (in which case it should be $\bot$). This is defined as:

$$f_{\mathcal{D} \to U}(x) \stackrel{\text{def}}{=} \begin{cases} \top_U & \text{if } x \neq \bot_{\mathcal{D}} \\ \bot_U & \text{if } x = \bot_{\mathcal{D}} \end{cases} \tag{3.2}$$

where $\mathcal{D}$ is any domain. The result of conversion *from* the unity domain is symmetric to function 3.2. Hence:

$$f_{U\to\mathcal{D}}(x) \stackrel{\text{def}}{=} \begin{cases} \top_{\mathcal{D}} & \text{if } x = \top_U \\ \bot_{\mathcal{D}} & \text{if } x = \bot_U \end{cases} \tag{3.3}$$

where $\mathcal{D}$ is any domain.

We will now show that the two conversion functions are sound.

Theorem 3.6.1 *Function $f_{U\to\mathcal{D}}$ (in Equation 3.3) is a sound conversion from $\mathcal{U}$ to D, provided $\alpha_{\mathcal{D}}$ and $\gamma_{\mathcal{D}}$ are mutually sound.*

Proof Let C be the concrete domain for $\mathcal{D}$.

There are two cases depending upon whether $x = \bot_U$ or not.

If $x = \bot_U$ then we have $f_{\mathbb{Z}\to C}(\gamma_U(x)) = \emptyset \subseteq \gamma_{\mathcal{D}}(f_{U\to\mathcal{D}}(x))$.

If $x \neq \bot_U$ then $\gamma_U(f_{\mathcal{D}\to U}(x)) = \gamma_U(\top_U) \supseteq \gamma_{\mathcal{D}}(x)$. ■

We now show that the conversion in the other direction is also sound.

Theorem 3.6.2 *Function 3.2 is a sound conversion from $\mathcal{D}$ to U (as defined by definition 3.6.1), as long as $\alpha_{\mathcal{D}}$ and $\gamma_{\mathcal{D}}$ are mutually sound according to definition 3.2.2.*

Proof Let C be the concrete domain for $\mathcal{D}$. Much like the earlier proof there are two cases, whether $x \in \mathcal{D}$ is $\bot_{\mathcal{D}}$ or not.

If $x = \bot_{\mathcal{D}}$ then $\gamma_{\mathcal{D}}(\bot_D) = \bot_C$. Further, for both cases of $C = \mathbb{Z}$ or $C = B$ we have $f_{C\to\mathbb{Z}}(\bot_C) = \emptyset \subseteq \gamma_U(f_{\mathcal{D}\to C}(x))$.

Assume $x \neq \bot_{\mathcal{D}}$. Then $f_{\mathcal{D}\to U}(x) = \top_U$ and $\gamma_U(\top_U) = \mathbb{Z} \subseteq f_{C\to\mathcal{D}}(\gamma_{\mathcal{D}}(x))$. ■

3.6.2 Trilog Conversion

The following definition of conversion mirrors the of C language, where all non-zero values are considered as true boolean value and zero is considered false.

$$f_{\mathcal{D}\to B_3}(x) \stackrel{\text{def}}{=} \{tt \mid \exists\, y \in \gamma_{\mathcal{D}}(x) \wedge y \neq 0\} \cup \{ff \mid 0 \in \gamma_{\mathcal{D}}(x)\} \tag{3.4}$$

As we will see, this function is equivalent to evaluation of the predicate $x = 0$. Function 3.4 is practically computable because for all our domains it's easy to test $\gamma(x)$ for a zero and non-zero member without actually computing the entire value of γ.

Conversion from a *Trilog* to another domain is done via γ_{B_3}, since the result has a cardinality of at most two. This one of the few cases where γ can be practically used to implement a conversion.

$$f_{B_3\to\mathcal{D}}(x) \stackrel{\text{def}}{=} \alpha_{\mathcal{D}}(f_{B\to C}(\gamma_{B_3}(x))) \tag{3.5}$$

We will now show that both of the conversions are sound.

Theorem 3.6.3 *The function $f_{B_3\to\mathcal{D}}$ defined in Equation 3.5 is sound provided $\alpha_{\mathcal{D}}$ and $\gamma_{\mathcal{D}}$ are mutually sound.*

Proof Let C be the concrete domain for abstraction $\mathcal{D}$. Then, by definition, we have

$$\gamma_{\mathcal{D}}(f_{B\to\mathcal{D}}(x)) =$$

$$\gamma_{\mathcal{D}}(\alpha_{\mathcal{D}}(f_{B\to C}(\gamma_{B_3}(x)))) \supseteq$$

by definition of abstractions

$$f_{B\to C}(\gamma_{B_3}(x))$$

as required. ■

Theorem 3.6.4 *The function $f_{\mathcal{D}\to B_3}$ defined in Equation 3.4 is sound.*

Proof Note that $B_3 \neq \mathcal{D}$, as there would be no point in the converting abstract values otherwise. Thus, the concrete world corresponding to $\mathcal{D}$ is $\mathbb{Z}$. We thus argue as follows:

$$\forall x \in \mathcal{D}.\ \gamma_{B_3}(f_{\mathcal{D}\to B_3}(x)) = \gamma_{B_3}(\{tt|\exists y \in \gamma_{\mathcal{D}}(x) \wedge y \neq 0\} \cup \{ff|0 \in \gamma_{\mathcal{D}}(x)\}) \subseteq \{0, 1\}$$

Now if $0 \in \gamma_{\mathcal{D}}(x)$ then $0 \in f_{C\to B}(\gamma_{\mathcal{D}}(x))$ and $0 \in \gamma_{B_3}(f_{\mathcal{D}\to B_3}(x)$.

If $1 \in \gamma_{\mathcal{D}}(x)$ then $1 \in f_{C\to B}(\gamma_{\mathcal{D}}((x))$ and $1 \in \gamma_{B_3}(f_{\mathcal{D}\to B_3}(x)$. Thus, we get for all $x \in \mathcal{D}$ that $\gamma_{B_3}(f_{\mathcal{D}\to B_3}(x) \subseteq f_{C\to B}(\gamma_{\mathcal{D}}((x))$, as required. ■

3.6.3 Range Conversion

The result of conversion to a range should be an upper and lower limit on the possible values. Hence, we will define two utility functions for each domain $\mathcal{D}$: A function $\lceil x \rceil_{\mathcal{D}} : \mathcal{D} \to \mathbb{Z}$, which computes an upper bound on x, and a function $\lfloor x \rfloor_{\mathcal{D}} : \mathcal{D} \to \mathbb{Z}$, which computes a lower bound. The requirement for these functions is that $\lceil x \rceil_{\mathcal{D}}$ return a value greater than or equal to $\sqcup\gamma_{\mathcal{D}}(x)$ and $\lfloor x \rfloor_{\mathcal{D}}$ return a value less than or equal to $\sqcap\gamma_{\mathcal{D}}(x)$. Ideally, these values should be as close to the actual bounds as possible. The formal definitions of these functions are:

$$
\begin{array}{lcl}
\lceil x \rceil_U & = & \mathit{maxint} \\
\lfloor x \rfloor_U & = & \mathit{minint} \\
\lceil \langle x, y \rangle \rceil_R & = & y \\
\lfloor \langle x, y \rangle \rfloor_R & = & x \\
\lceil S \rceil_{M_k} & = & \begin{cases} \mathit{maxint} & \text{if } \sqcup S \geq 0 \\ \sqcup S & \text{if } \sqcup S < 0 \end{cases} \\
\lfloor S \rfloor_{M_k} & = & \begin{cases} \sqcap S & \text{if } \sqcup S \geq 0 \\ \mathit{minint} & \text{if } \sqcup S < 0 \end{cases} \\
\lceil T \rceil_{I_S} & = & \sqcup\{y \mid \langle x, y \rangle \in T\} \\
\lfloor T \rfloor_{I_S} & = & \sqcap\{x \mid \langle x, y \rangle \in T\}
\end{array}
$$

Given the bounding functions, conversion to *Range* from any other domain is now easy:

$$
f_{\mathcal{D}\to R}(x) \stackrel{\text{def}}{=} \begin{cases} \bot_R & \text{if } x = \bot_{\mathcal{D}} \\ \langle \lfloor x \rfloor_{\mathcal{D}}, \lceil x \rceil_{\mathcal{D}} \rangle & \text{if } x \neq \bot_{\mathcal{D}} \end{cases} \qquad (3.6)
$$

where $\mathcal{D} \in \{U, M_k, I_S\}$

That the conversion function is sound can be easily established by appealing to the definition of the conversion function. Formally, we have:

Theorem 3.6.5 *For all domains $\mathcal{D}$ the function $f_{\mathcal{D}\to R}$ is a sound conversion from $\mathcal{D}$ to R.*

3.6.4 Mod k Conversion

To convert to a mod k abstraction, we can simply iterate over the concrete values produced by γ (without pre-computing the entire set), until either the values are exhausted, or $\top_{M_k}$ is reached. Hence, we have the following function:

$$f_{\mathcal{D}\to M_k}(x) \stackrel{\text{def}}{=} \alpha_{M_k}(\gamma_{\mathcal{D}}(x)) \tag{3.7}$$

A related problem is conversion between *mod* values of differing divisors. This is handled by:

$$f_{M_{k_1}\to M_{k_2}}(S) \stackrel{\text{def}}{=} \begin{cases} \top_{M_{k_2}} & \text{if } k_2 \text{ is not a divisor or multiple of } k_1 \\ \{x \; mod \; k_2 \mid x \in S\} & \text{if } k_2 \text{ is a divisor of } k_1 \\ \{x \cdot i \mid x \in S \text{ and } i \geq 1 \\ \quad \text{and } -k_2 < x \cdot i \leq k_2\} & \text{if } k_2 \text{ is a multiple of } k_1 \end{cases} \tag{3.8}$$

The justification for why these conversions are sound followed similar theorems established earlier. We have:

Theorem 3.6.6 *For all domains $\mathcal{D}$ the function $f_{\mathcal{D}\to M_k}$ is sound.*

3.6.5 Interval Conversion

Although we could be more precise, we have chosen to just generate $\top$ when values are converted from a Mod abstraction to an $Interval$ abstraction.

$$f_{M_k\to\mathcal{I}_S}(x) \stackrel{\text{def}}{=} \top_{\mathcal{I}_S} \tag{3.9}$$

Conversion of *Range* values to $Interval$ ones is done by computing which of the intervals falls within the bounds of the *Range* value.

$$f_{R\to\mathcal{I}_S}(x) \stackrel{\text{def}}{=} \begin{cases} \{\langle l', h'\rangle \in S \mid h' \geq l \text{ or } l' \leq h\} & \text{if } x = \langle l, h\rangle \\ \varnothing & \text{if } x = \bot_R \end{cases} \tag{3.10}$$

Conversion of differing $Interval$ values is done by in manner similar function 3.10, with the difference that more than one pair of bounds is involved.

ASCII	Symbol	Type	Description
`+`	$+$	$\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$	addition
`-`	$-$	$\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$	subtraction
`*`	$\cdot$	$\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$	multiplication
`/`	$/$	$\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$	division
`%`	mod	$\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$	remainder
`|`	$\overset{bit}{\vee}$	$\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$	bitwise or
`&`	$\overset{bit}{\wedge}$	$\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$	bitwise and
`^`	$\overset{bit}{\oplus}$	$\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$	bitwise exclusive or
`>>`	$\curvearrowright$	$\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$	left shift
`<<`	$\curvearrowleft$	$\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$	right shift
`==`	$=$	$\mathbb{Z} \times \mathbb{Z} \to B$	equality
`!=`	$\neq$	$\mathbb{Z} \times \mathbb{Z} \to B$	inequality
`<`	$<$	$\mathbb{Z} \times \mathbb{Z} \to B$	less than
`<=`	$\leq$	$\mathbb{Z} \times \mathbb{Z} \to B$	less than or equal
`>=`	$\geq$	$\mathbb{Z} \times \mathbb{Z} \to B$	greater than or equal
`>`	$>$	$\mathbb{Z} \times \mathbb{Z} \to B$	greater than
`-`	$-$	$\mathbb{Z} \to \mathbb{Z}$	integer negation
`+`	$+$	$\mathbb{Z} \to \mathbb{Z}$	absolute value
`++`	$+_1$	$\mathbb{Z} \to \mathbb{Z}$	increment
`--`	$-_1$	$\mathbb{Z} \to \mathbb{Z}$	decrement
`~`	$\overset{bit}{\neg}$	$\mathbb{Z} \to \mathbb{Z}$	bitwise negation
`&&`	$\wedge$	$B \times B \to B$	and
`||`	$\vee$	$B \times B \to B$	or
`!!`	$\neg$	$B \to B$	boolean negation

Table 3.7: CWI Integer operators and their types.

$$f_{\mathcal{I}_{S_1} \to \mathcal{I}_{S_2}}(x) \stackrel{\text{def}}{=} \{\langle l', h' \rangle \in S_2 \mid \exists \langle l, h \rangle \in S_1 \,.\, h' \geq l \text{ or } l' \leq h\} \tag{3.11}$$

Given that our abstractions are weak it is easy to show that all three conversion functions $f_{M_k \to \mathcal{I}_s}$, $f_{r \to \mathcal{I}_s}$ and $f_{\mathcal{I}_{s_1} \to \mathcal{I}_{s_2}}$ are sound.

3.7 Abstract Operations

Table 3.7 lists the CWI integer operators and their types. Each abstract type

needs to implement the relevant set of operations. For $Trilog$, the relevant operators are $\wedge$, $\vee$, and $\neg$. For every other domain, the relevant operators all the operators *except* $\wedge$, $\vee$, and $\neg$. These operators cover full range of C operations, which may seem daunting, until you consider that many operations will simply be implemented as $f(x, y) = \top$, such as division of Mod values.

The semantics of these operations for the concrete world should be fairly intuitive to any reader familiar with a programming language; the precise details are available in [38]. Not surprisingly, the abstract semantics are different for each domain, requiring the result of each operator to be defined distinctly. The notation

$$(v_1 \circ_{\mathcal{D}} v_2) \stackrel{\text{def}}{=} v_3$$

shall denote the fact that operator $\circ_{\mathcal{D}} : \mathcal{D} \times \mathcal{D} \rightarrow \mathcal{D}$ is defined to result in value v_3 when applied to values v_1 and v_2.

3.7.1 Default Operators

There are many operators which don't yield any useful information for a given domain. Application of such operations always results in $\top$ unless one of the arguments is $\bot$, in which case the result is $\bot$. Hence, we shall define two default operators. $\circledcirc^1_{\mathcal{D}} : \mathcal{D} \rightarrow \mathcal{D}$ for unary operations, and $\circledcirc^2_{\mathcal{D}} : \mathcal{D} \times \mathcal{D} \rightarrow \mathcal{D}$ for binary operations, where $\mathcal{D}$ can be any domain.

$$(\circledcirc^1_{\mathcal{D}}\ x) \stackrel{\text{def}}{=} \begin{cases} \top_{\mathcal{D}} & \text{if } x \neq \bot_{\mathcal{D}} \\ \bot_{\mathcal{D}} & \text{if } x = \bot_{\mathcal{D}} \end{cases} \tag{3.12}$$

$$(x \circledcirc^2_{\mathcal{D}}\ y) \stackrel{\text{def}}{=} \begin{cases} \top_{\mathcal{D}} & \text{if } x \neq \bot_{\mathcal{D}} \text{ and } y \neq \bot_{\mathcal{D}} \\ \bot_{\mathcal{D}} & \text{otherwise} \end{cases} \tag{3.13}$$

Theorem 3.7.1 *For any abstract domain $\mathcal{D}$ which is a sound abstraction of some concrete domain $\mathcal{C}$, the unary operator $\circledcirc^1_{\mathcal{D}} : \mathcal{D} \rightarrow \mathcal{D}$ is a sound abstraction of any concrete unary operator $\circ : \mathcal{C} \rightarrow \mathcal{C}$.*

Proof According to definition 3.2.3, a unary operator $\circledcirc^1_{\mathcal{D}} : \mathcal{D} \rightarrow \mathcal{D}$ is sound with respect to $\circ_{\mathcal{C}} : \mathcal{C} \rightarrow \mathcal{C}$ if the following condition is satisfied:

$$\text{For all } a \in \mathcal{D}, \text{ for all } x \in \gamma_{\mathcal{D}}(a), \quad \circ_{\mathcal{C}} x \in \gamma_{\mathcal{D}}(\circledcirc^1_{\mathcal{D}} a)$$

Pick any $a \in \mathcal{D}$.

Either (1) $a \neq \perp_{\mathcal{D}}$ or (2) $a = \perp_{\mathcal{D}}$.

Case (1): $a \neq \perp_{\mathcal{D}}$

$\circledcirc^1_{\mathcal{D}} a = \top_{\mathcal{D}}$

$\gamma_{\mathcal{D}}(\circledcirc^1_{\mathcal{D}} a) = \gamma_{\mathcal{D}}(\top_{\mathcal{D}}) = \mathcal{C}$

Pick any $x \in \gamma_{\mathcal{D}}(a)$.

$\circ_{\mathcal{C}}\ x \in \mathcal{C} = \gamma_{\mathcal{D}}(\circledcirc^1_{\mathcal{D}}\ a)$, which satisfies definition 3.2.3.

Case (2): $a = \perp$

$\gamma_{\mathcal{D}}(a) = \varnothing$

No $x \in \varnothing$ exists, so definition 3.2.3 is vacuously satisfied.

In both cases, definition 3.2.3 is satisfied.

■

Theorem 3.7.2 *For any abstract domain $\mathcal{D}$ which is a sound abstraction of some concrete domain $\mathcal{C}$, the binary operator $\circledcirc^2_{\mathcal{D}} : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ is a sound abstraction of any concrete binary operator $\circ : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$.*

Proof According to definition 3.2.4, a unary operator $\circledcirc^2_{\mathcal{D}} : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ is sound with respect to $\circ_{\mathcal{C}} : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ if the following condition is satisfied:

$$\text{For all } \langle a, b\rangle \in \mathcal{D} \times \mathcal{D}, \text{ for all } \langle x, y\rangle \in \gamma_{\mathcal{D}}(a) \times \gamma_{\mathcal{D}}(b), \quad x\ \circ_{\mathcal{C}}\ y \in \gamma_{\mathcal{D}}(a\ \circledcirc^2_{\mathcal{D}}\ b)$$

Pick any a and $b \in \mathcal{D}$

Either (1) ($a \neq \bot_{\mathcal{D}}$ and $b \neq \bot_{\mathcal{D}}$), or (2) ($a = \bot_{\mathcal{D}}$ or $b = \bot_{\mathcal{D}}$).

Case (1): $a \neq \bot_{\mathcal{D}}$ and $b \neq \bot_{\mathcal{D}}$.

$\gamma_{\mathcal{D}}(a \ \circledcirc^2_{\mathcal{D}} \ b) = \gamma_{\mathcal{D}}(\top_{\mathcal{D}}) = \mathcal{C}$

Pick any $x \in \gamma_{\mathcal{D}}(a)$, and any $y \in \gamma_{\mathcal{D}}(b)$.

$x \ \circ_{\mathcal{C}} \ y \in \mathcal{C} = \gamma_{\mathcal{D}}(\top\mathcal{D}) = \gamma_{\mathcal{D}}(a \ \circledcirc^2_{\mathcal{D}} \ b)$, which satisfies definition 3.2.4.

Case (2): $a = \bot_{\mathcal{D}}$ or $b = \bot_{\mathcal{D}}$

Without loss of generality, assume $a = \bot_{\mathcal{D}}$.

$\gamma_{\mathcal{D}} \ a = \gamma_{\mathcal{D}} \ \bot_{\mathcal{D}} = \varnothing$.

No $x \in \varnothing = \gamma_{\mathcal{D}} \ a$ exists, so definition 3.2.4 is vacuously satisfied.

In both cases, definition 3.2.4 is satisfied.

■

3.7.2 Logical Operations

The implementation of the logical operators is defined by simply applying the corresponding concrete operator to each element of the set, as follows:

$$(\neg_{B_3} v) \stackrel{\text{def}}{=} \{\neg_B x \mid x \in v\} \tag{3.14}$$

$$(v_1 \circ_{B_3} v_2) \stackrel{\text{def}}{=} \{x_1 \circ_B x_2 \mid x_1 \in v_1 \text{ and } x_2 \in v_2\} \text{ where } \circ \in \{\vee, \wedge\} \tag{3.15}$$

Theorem 3.7.3 *The binary B_3 operators are sound.*

Proof According to definition 3.2.4, an operator $\circ_{B_3}$ is sound if

$$\forall a_1, a_2 \in B_3, \{x \circ_{B_3} y \mid x \in \gamma(a_1), y \in \gamma(a_2)\} \subseteq \gamma_{B_3}(a_1 \circ_{B_3} a_2)$$

Pick any $a_1, a_2 \in B_3$.

$\{x \circ_{B_3} y \mid x \in \gamma_{B_3}(a_1), y \in \gamma_{B_3}(a_2)\}$

$= \{x \circ_{B_3} y \mid x \in a_1, y \in a_2\}$

$= a_1 \circ_{B_3} a_2$

$= \gamma_{B_3}(a_1 \circ_{B_3} a_2)$ ■

Theorem 3.7.4 *The operator $\neg_{B_3}$ is sound.*

Proof Definition 3.2.3 specifies that $\neg_{B_3}$ is sound if the following holds true:

$$\text{For all } a \in B_3, \{\neg x \mid x \in \gamma_{B_3}(a)\} \subseteq \gamma_{B_3}(\neg_{B_3} a)$$

Pick any $a \in B_3$.

$\{\neg_B x \mid x \in \gamma_{B_3}(a)\}$

$= \{\neg_B x \mid x \in a\}$

$= \neg_{B_3} a$

$= \gamma_{B_3}(\neg_{B_3} a)$

$\subseteq \gamma_{B_3}(\neg_{B_3} a)$ ■

3.7.3 Relational Operators

The relational operators are $=$, $\neq$, $<$, $\leq$, $\geq$, and $>$. For an application of the form $x \triangleleft y$, the result should be true if and only if every possible concrete value pairing in $\gamma(x) \times \gamma(y)$ satisfies $\triangleleft$. The result should be false if and only if every possible concrete value pairing in $\gamma(x) \times \gamma(y)$ doesn't satisfy $\triangleleft$. Otherwise, the result should be "maybe." Formally this is expressed as the following soundness condition:

Definition 3.7.1 *An operator $\triangleleft_{\mathcal{D}} : \mathcal{D} \times \mathcal{D} \to B_3$ is a **sound relational operator** with respect to its concrete dual $\triangleleft_{\mathbb{Z}} : \mathbb{Z} \times \mathbb{Z} \to B$ if the following condition is satisfied:*

$$\forall a, b \in \mathcal{D}, \forall x \in \gamma_{\mathcal{D}}(a), \forall y \in \gamma_{\mathcal{D}}(b), (x \triangleleft_{\mathbb{Z}} y) \in \gamma_{B_3}(a \triangleleft_{\mathcal{D}} b)$$

Equality Operators

Testing for equality is done by examining the cross-product of the γ-sets from two abstract values. Basically, the two values are definitely equal if and only if they both map (via γ) unto the same single concrete value. The are not equal if and only they map onto disjoint sets, and are "maybe equal" if the sets overlap and have cardinality greater than one.

For the *Unity* domain, the result is always "maybe," unless one of the arguments is $\perp$.

$$(x =_U y) \stackrel{\text{def}}{=} \begin{cases} \varnothing & \text{if } x = \perp_U \text{ or } y = \perp_U \\ \{tt, ff\} & \text{if } x = \top_U \text{ and } y = \top_U \end{cases} \tag{3.16}$$

For the *Range* domain, we check the bounds of the range. If the upper and lower bounds all match, then the result is true. If the two ranges are disjoint, then the answer is false. If one of the ranges is $\perp$, then the result is $\perp$. Otherwise, the answer is maybe. To simplify the definition, we use the upper and lower limit functions defined in section 3.6.3.

$$(x =_R y) \stackrel{\text{def}}{=} \begin{cases} \varnothing & \text{if } x = \perp_R \text{ or } y = \perp_R \\ \{tt\} & \text{if } \lceil x \rceil_R = \lfloor x \rfloor_R = \lceil y \rceil_R = \lfloor y \rfloor_R \\ \{ff\} & \text{if } \lceil x \rceil_R < \lfloor y \rfloor_R \text{ or } \lceil y \rceil_R < \lfloor x \rfloor_R \\ \{tt, ff\} & \text{otherwise} \end{cases} \tag{3.17}$$

For the *Modulo* domain, the result can never be true. The result is *maybe* if the values have at least one remainder in common, and *false* otherwise.

$$(x =_{M_k} y) \stackrel{\text{def}}{=} \begin{cases} \varnothing & \text{if } x = \varnothing \text{ or } y = \varnothing \\ \{ff\} & x \cap y \neq \varnothing \\ \{tt, ff\} & \text{otherwise} \end{cases} \tag{3.18}$$

Equality under the *Interval* domain is a combination of the truth test of the *Range* domain with the falsehood test of the *Modulus* domain.

$$(x =_{\mathcal{I}_S} y) \stackrel{\text{def}}{=} \begin{cases} \varnothing & \text{if } x = \emptyset \text{ or } y = \emptyset \\ \{tt\} & \text{if } \lceil x \rceil_{\mathcal{I}_S} = \lfloor x \rfloor_{\mathcal{I}_S} = \lceil y \rceil_{\mathcal{I}_S} = \lfloor y \rfloor_{\mathcal{I}_S} \\ \{ff\} & \text{if } x \cap y \neq \varnothing \\ \{tt, ff\} & \text{otherwise} \end{cases} \tag{3.19}$$

The $\neq$ relation is be computed by taking the negation of $=$. Hence, the following short function will suffice for all domains:

$$\begin{aligned}(x \neq_{\mathcal{D}} y) \stackrel{\text{def}}{=} \quad & \neg_{B_3}(x =_{\mathcal{D}} y) \\ & \text{where } \mathcal{D} \in \{U, B_3, R, M_k, \mathcal{I}_S\}\end{aligned} \tag{3.20}$$

Ordering Operators

The relational ordering operators ($<, \leq, \geq,$ and $>$) are also defined in terms of the upper- and lower-bound operators of section 3.6.3. For all the ordering relations, two values are related if and only if the upper bound of one is related to the lower bound of the other. In this case a definite yes/no answer can be reached. Otherwise, the answer to the ordering predicate is "maybe."

$$x \lhd_{\mathcal{D}} y \stackrel{\text{def}}{=} \begin{cases} \varnothing & \text{if } x = \bot_{\mathcal{D}} \text{ or } y = \bot_{\mathcal{D}} \\ \{tt\} & \text{if } \lceil x \rceil_{\mathcal{D}} \lhd_{\mathbb{Z}} \lfloor y \rfloor_{\mathcal{D}} \\ \{ff\} & \text{if } \lceil x \rceil_{\mathcal{D}} \not\lhd_{\mathbb{Z}} \lfloor y \rfloor_{\mathcal{D}} \\ \{tt, ff\} & \text{otherwise} \end{cases} \quad \begin{array}{ll} \text{where} & \lhd \in \{<, \leq, \geq, >\} \\ & \mathcal{D} \in \{U, B_3, R, M_k, \mathcal{I}_S\} \end{array} \tag{3.21}$$

Theorem 3.7.5 *The relational operators $<_{\mathcal{D}}, \leq_{\mathcal{D}}, \geq_{\mathcal{D}},$ and $>_{\mathcal{D}}$ are sound.*

Proof According to definition 3.2.4, a relational operator $\lhd_{\mathcal{D}}$ is sound if

$$\forall a, b \in \mathcal{D}, \forall x \in \gamma_{\mathcal{D}}(a), \forall y \in \gamma_{\mathcal{D}}(b), (x \lhd_{\mathbb{Z}} y) \in \gamma_{B_3}(a \lhd_{\mathcal{D}} b)$$

Pick any $\lhd \in \{<, \leq, =, \neq, \geq, >\}$.

Pick any $a_1, a_2 \in \mathcal{D}$.

Either (1) $a_1 = a_2 = \top_{\mathcal{D}}$ or else (2) ($a_1 = \bot_{\mathcal{D}}$ or $a_2 = \bot_{\mathcal{D}}$)

Case (1): $a_1 = a_2 = \top_{\mathcal{D}}$

$\gamma_{B_3}(a_1 \lhd_{\mathcal{D}} a_2) = \gamma_{B_3}(\top_{\mathcal{D}} \lhd_{\mathcal{D}} a_2) = \gamma_{B_3}(\{tt, ff\}) = \{0, 1\}$

From C semantics, we know that for every $x, y \in \mathbb{Z}$, $x \lhd_{\mathbb{Z}} y \in \{0, 1\}$.

Substituting $\gamma_{B_3}(a_1 \lhd_{\mathcal{D}} a_2)$ for $\{0,1\}$, $x \lhd_{\mathbb{Z}} y \subseteq \gamma_{\mathcal{D}}(a_1 \lhd_{\mathcal{D}} a_2)$

Case (2): ($a_1 = \perp$ or $a_2 = \perp$)

Without loss of generality assume $a_1 = \perp$

$\gamma_{\mathcal{D}}(a_1) = \varnothing$, so no $x \in \gamma_{\mathcal{D}}(a_1)$ exists and definition 3.2.4 is vacuously satisfied.

In all cases, definition 3.2.4 is satisfied.

■

3.7.4 Unity Numeric Operations

The *Unity* domain numeric operations are trivial in their complexity. Their semantics are identical to the "default" operators (3.12 and 3.13). These operators always return $\top_U$ unless one of the arguments is $\perp_U$, in which case, the $\perp_U$ is propagated forth.

$$\circ_U \stackrel{\text{def}}{=} \circledcirc^1_U \quad \text{where } \circ \in \{-, +, +_1, -_1, \overset{bit}{\neg}\} \tag{3.22}$$

$$\circ_U \stackrel{\text{def}}{=} \circledcirc^2_U \quad \text{where } \circ \in \{+, -, \cdot, /, mod, \overset{bit}{\vee}, \overset{bit}{\wedge}, \overset{bit}{\oplus}, \overset{bit}{\curvearrowright}, \overset{bit}{\curvearrowleft}\} \tag{3.23}$$

Theorem 3.7.6 *All the Unity unary numeric operators are sound.*

Proof A direct corollary of theorem 3.7.1. ■

Theorem 3.7.7 *All the Unity binary numeric operators are sound.*

Proof A direct corollary of theorem 3.7.2. ■

3.7.5 Range Domain Numeric Operations

The numeric operators over the *Range* domain need only to keep track of the extremes of the set of possible values. Hence for operators which are either non-increasing or non-decreasing, the value of an expression can be computed by computing the set of every

possible combination of values, and then finding the maximum and minimum value of the set. Thus, we could define "generic" abstract operators which work for any non-increasing or non-decreasing concrete operator. However, dealing with overflow and other problems results in a generic operator which is not easily understood. Hence we have chosen to give separate functions for each operator.

The unary negation *Range* operator is based on the assumption of a 2's complement representation. In this representation, the range of values is $-2^{n-1} \ldots 2^{n-1} - 1$. Under this representation, the negation of all values is well-defined except for *minint*, whose negation is itself. Formally, the concrete operation is as follows:

$$-_{\mathbb{Z}} x = \begin{cases} minint & \text{if } x = minint \\ |x| & \text{if } minint < x < 0 \\ -x & \text{if } x \geq 0 \end{cases}$$

We need to emulate these semantics in the abstract world, which we do so as follows.

$$-_R\, x \stackrel{\text{def}}{=} \begin{cases} \perp_R & \text{if } x = \perp_R \\ \langle -h, -l \rangle & \text{if } x = \langle l, h \rangle \text{ and } l > minint \\ \langle minint, minint \rangle & \text{if } x = \langle minint, minint \rangle \\ \langle minint, maxint \rangle & \text{if } x = \langle minint, h \rangle \text{ and } h > minint \end{cases} \tag{3.24}$$

We will now show that the abstraction operation is indeed faithful to the concrete operation.

Theorem 3.7.8 *The unary operator $-_R$ is sound.*

Proof According to definition 3.2.3 $-_R$ is sound if the following is satisfied:

$$\text{For all } a \in R, \{-x \mid x \in \gamma_R(a)\} \subseteq \gamma_R(-_R(a)$$

Pick any $a \in R$.

Either $a \neq \perp_R$ or $a = \perp_R$.

Case 1: $a \neq \perp_R$.

Let $a = \langle x, y \rangle$

$\{-w \mid w \in \gamma_{_R}(a)\}$

$= \{-w \mid w \in \gamma_{_R}(x, y)\}$

$= \{-w \mid w \in \{x \ldots y\}\}$

$= \{-w \mid x \leq w \leq y\}$

$= \{w \mid x \leq -w \leq y\}$

$= \{w \mid -y \leq w \leq -x\}$

$= \{w \mid w \in \{-y \ldots -x\}\}$

$= \{-y \ldots -x\}$

$= \gamma_{_R}(-y, -x)$

$= \gamma_{_R}(-_{_R}\langle x, y \rangle)$

$= \gamma_{_R}(-_{_R}a)$

Case 2: $a = \perp_{_R}$.

$\{-w \mid w \in \gamma_{_R}(a)\}$

$\{-w \mid w \in \gamma_{_R}(\perp_{_R})\}$

$\{-w \mid w \in \varnothing\}$

$\varnothing$

$\subseteq \gamma_{_R}(-_{_R}\langle x, y \rangle)$

In both cases, definition 3.2.3 is satisfied. ■

The absolute value operator has the same issues as unary negation. In the following we will present the abstract operations; the proof of its soundness is similar to unary negation and, hence, omitted. The problem is that there is no positive representation of

the negation of *minint*. On a 2's compliment architecture, the negation of *minint* is itself.

$$+_R\, x \stackrel{\text{def}}{=} \begin{cases} \bot_R & \text{if } x = \bot_R \\ \langle \sqcap \{+l, +h\}, \sqcup \{+l, +h\} \rangle & \text{if } x = \langle l, h \rangle \text{ and } l > minint \\ \langle minint, minint \rangle & \text{if } x = \langle minint, minint \rangle \\ \langle minint, maxint \rangle & \text{if } x = \langle minint, h \rangle \text{ and } h > minint \end{cases} \tag{3.25}$$

Using a similar strategy our abstraction for the increment operation is as follows:

$$+_{1_R}\, x \stackrel{\text{def}}{=} \begin{cases} \bot_R & \text{if } x = \bot_R \\ \langle +_1 l, +_1 h \rangle & \text{if } x = \langle l, h \rangle \text{ and } h < maxint \\ \langle minint, maxint \rangle & \text{otherwise} \end{cases} \tag{3.26}$$

The decrement operation can be abstracted in the following fashion:

$$-_{1_R}\, x \stackrel{\text{def}}{=} \begin{cases} \bot_R & \text{if } x = \bot_R \\ \langle -_1 l, -_1 h \rangle & \text{if } x = \langle l, h \rangle \text{ and } l > minint \\ \langle minint, maxint \rangle & \text{otherwise} \end{cases} \tag{3.27}$$

We assume that our target machine employs a 2's compliment mathematical architecture, and hence $\stackrel{bit}{\neg}\, x$ will be equal to $(-x) - 1$. There is no chance of underflow or overflow. Hence, we abstract as follows:

$$\stackrel{bit}{\neg}_R\ x \stackrel{\text{def}}{=} \begin{cases} \bot_R & \text{if } x = \bot_R \\ \langle -h - 1, -l - 1 \rangle & \text{if } x = \langle l, h \rangle \end{cases} \tag{3.28}$$

Binary Range operators

The abstractions for binary *Range* operators are defined in this section. First, to simplify things, all binary operators are defined to have the result $\bot$ if any of the arguments are $\bot$:

$$a \circ_R b \stackrel{\text{def}}{=} \bot_R \text{ if } a = \bot_R \text{ or } \bot_R \tag{3.29}$$

Second, we define the behavior of each operator for the case when all the arguments are non-$\bot$ values. For addition, this means we just add the lower and upper bounds, and be sure to check for overflow.

$$\langle l_1, h_1\rangle +_R \langle l_2, h_2\rangle \stackrel{\text{def}}{=} \begin{cases} \langle minint, maxint\rangle & \text{if } h_1 + h_2 > maxint \text{ or } l_1 + l_2 < minint \\ \langle l_1 + l_2, h_1 + h_2\rangle & \text{otherwise} \end{cases} \tag{3.30}$$

Once addition and unary negation are defined, then subtraction can be defined in terms of them.

$$a -_R b \stackrel{\text{def}}{=} a +_R (-_R b) \tag{3.31}$$

For multiplication, we compute every possible combination of extremes and take the limits, while checking for overflow.

$$\langle l_1, h_1\rangle \cdot_R \langle l_2, h_2\rangle \stackrel{\text{def}}{=} \begin{cases} \langle minint, maxint\rangle & \text{if } \sqcup S > maxint \text{ or } \sqcap S < minint \\ \langle \sqcap S, \sqcup S\rangle & \text{otherwise} \end{cases} \quad \text{where } S = \{l_1 \cdot l_2, l_1 \cdot h_2, h_1 \cdot l_2, h_1 \cdot h_2\} \tag{3.32}$$

We now show that this multiplication operation on ranges is sound; other binary operations defined in this section can be shown to be sound in a similar manner.

Theorem 3.7.9 *The binary operator $\cdot_R$ is sound.*

Proof According to definition 3.2.4, $\cdot_R$ is sound if

$$\text{For all } a_1, a_2 \in R, \{x + y \mid x \in \gamma(a_1), y \in \gamma(a_2)\} \subseteq \gamma_R(a_1 +_R a_2)$$

Either ($a_1 = \langle x_1, y_1\rangle$ and $a_2 = \langle x_2, y_2\rangle$) or ($a_1 = \perp$ or $a_2 = \perp$)

Case 1: $a_1 = \langle x_1, y_1\rangle$ and $a_2 = \langle x_2, y_2\rangle$.

$\{x \cdot y \mid x \in \gamma(a_1), y \in \gamma(a_2)\}$

$= \{x \cdot y \mid x \in \gamma(x_1, y_1), y \in \gamma(x_2, y_2)\}$

$= \{x \cdot y \mid x \in x_1 \leq y_1, x_2 \leq y \leq y_2\}$

$\subseteq \{z \mid x_1 \cdot x_2 \leq z \leq y_1 \cdot y_2\}[Since \cdot is an increasing function]$

$= \{z \mid z \in \gamma(x_1 \cdot x_2, y_1 \cdot y_2)\}$

$= \{z \mid z \in \gamma(\langle x_1, y_1\rangle +_R \langle x_2, y_2\rangle)\}$

$= \{z \mid z \in \gamma(a_1 +_R a_2)\}$

$= \gamma(a_1 +_R a_2)$

Case 2: $a_1 = \bot_R$ or $a_2 = \bot_R$.

Without loss of generality, assume $a_1 = \bot$.

$\{x \cdot y \mid x \in \gamma(\bot), y \in \gamma(a_2)\}$

$= \{x \cdot y \mid x \in \varnothing, y \in \gamma(a_2)\}$

$= \varnothing$

$\subseteq \gamma(a_1 +_R a_2)$

In both cases, definition 3.2.4 is satisfied.

■

For division, we simply compute every possible combination of extremes and take the limits. If the divisor includes the possibility of 0, then the result is $\top$.

$$\langle l_1, h_1\rangle /_R \langle l_2, h_2\rangle \stackrel{\text{def}}{=} \begin{cases} \langle minint, maxint\rangle & \text{if } l_2 \le 0 \le h_2 \\ \langle \sqcap S, \sqcup S\rangle & \text{otherwise} \end{cases} \quad (3.33)$$
$$\text{where } S = \{\lfloor \tfrac{l_1}{l_2} \rfloor, \lfloor \tfrac{l_1}{h_2} \rfloor, \lfloor \tfrac{h_1}{l_2} \rfloor, \lfloor \tfrac{h_1}{h_2} \rfloor\}$$

To compute the modulus, we note that the absolute value of $x \ mod \ y$ is always less than or equal the absolute values of both x and y.

$$\langle l_1, h_1\rangle mod_R \langle l_2, h_2\rangle \stackrel{\text{def}}{=} \begin{cases} \langle minint, maxint\rangle & \text{if } l_2 \le 0 \le h_2 \\ \langle -k+1, k-1\rangle & \text{otherwise} \end{cases} \quad (3.34)$$
$$\text{where } k = \sqcap\{+l_1, +h_1, +l_2, +h_2\}$$

The bitwise binary operations are neither non-increasing or non-decreasing, but actually change direction repeatedly. Not all is lost however. On a 2's compliment architecture, the following facts hold true for the *bitwise-and* operator:

- The bitwise and of two values results in a positive sign unless both are negative.
- The bitwise and of two non-negative values is bounded above by the smaller of the two values, and bounded below by zero.
- The bitwise and of a non-negative value and a negative value is bounded above by the non-negative value, and bounded below by zero.
- The bitwise and of two negative values is bounded above by the smaller of the two values, and bounded below by *minint*.

These observations lead to the following abstraction:

$$\langle l_1, h_1\rangle \overset{bit}{\wedge}_R \langle l_2, h_2\rangle \overset{\text{def}}{=} \langle l', h'\rangle$$

$$\text{where } h' = \begin{cases} \sqcap\{h_1, h_2\} & \text{if } h_1 \geq 0 \text{ and } h_2 \geq 0 \\ h_1 & \text{if } h_1 \geq 0 \text{ and } h_2 < 0 \\ h_2 & \text{if } h_1 < 0 \text{ and } h_2 \geq 0 \\ \sqcup\{h_1, h_2\} & \text{if } h_1 < 0 \text{ and } h_2 < 0 \end{cases} \tag{3.35}$$

$$\text{and } l' = \begin{cases} 0 & \text{if } l_1 \geq 0 \text{ or } l_2 \geq 0 \\ minint & \text{otherwise} \end{cases}$$

On a 2's compliment architecture, the following facts are known about the results of a *bitwise-or* operation:

- The sign of the result of a bitwise or operation is positive if and only if both values are positive.
- The bitwise or of two non-negative values is bounded below by the larger of the two values, and bounded above $2^k - 1$, where 2^k is the smallest power of 2 such that 2^k is greater than or equal to both values.
- The bitwise or of a negative value and any other value value is bounded below by the smaller value, and bounded above by -1.

Exploiting these facts results in the following function:

$$\langle l_1, h_1\rangle \overset{bit}{\vee}_R \langle l_2, h_2\rangle \overset{\text{def}}{=} \langle l', h'\rangle$$
$$\text{where } h' = \begin{cases} 2^{\lceil log_2\ (max(h_1, h_2))\rceil} - 1 & \text{if } h_1 \geq 0 \text{ and } h_2 \geq 0 \\ -1 & \text{otherwise} \end{cases}$$
$$\text{and } l' = \begin{cases} \sqcup\{l_1, l_2\} & \text{if } l_1 \geq 0 \text{ and } l_2 \geq 0 \\ \sqcap\{l_1, l_2\} & \text{otherwise} \end{cases} \tag{3.36}$$

The *exclusive-or* operator is even less predictable that the previous two operators. All we can do in this case is predict the sign, which is positive if the signs of the arguments are the same, and is negative if the signs differ (the same as for multiplication).

$$\langle l_1, h_1\rangle \overset{bit}{\vee}_R \langle l_2, h_2\rangle \overset{\text{def}}{=} \begin{cases} \langle 0, maxint\rangle & \text{if } (h_1 \leq 0 \text{ and } h_2 \leq 0) \text{ or } (l_1 \geq 0 \text{ and } l_2 \geq 0) \\ \langle minint, -1\rangle & \text{if } (h_1 < 0 \text{ and } l_2 \geq 0) \text{ or } (h_2 < 0 \text{ and } l_1 \geq 0) \\ \langle minint, maxint\rangle & \text{otherwise} \end{cases} \tag{3.37}$$

Finally, the shift operators are worth the trouble of abstracting, so we will define the following:

$$x \overset{\curvearrowright}{\sqcap}_R y \overset{\text{def}}{=} \langle minint, maxint\rangle \tag{3.38}$$

and

$$x \overset{\curvearrowleft}{\sqcap}_R y \overset{\text{def}}{=} \langle minint, maxint\rangle \tag{3.39}$$

3.7.6 Modulus k Numeric Operators

The *Modulus k* domain numeric operations maintain a set of potential remainders. For unary operators, this is done by computing the effect of the unary operation on each remainder in the argument (which is also a set of remainders).

The effect of negation on a remainder is to change the sign of remainder, because we know that if $x\ mod\ k = r$, then $-x\ mod\ k = -r$.

$$-_{M_k} x \overset{\text{def}}{=} \{-a \mid a \in x\} \tag{3.40}$$

Similarly, the effect of absolute value on a remainder is to change the sign of remainder:

$$+_{M_k} x \stackrel{\text{def}}{=} \{+a \mid a \in x\} \tag{3.41}$$

The effect of adding two a *mod* values is to add the the remainders, since we know that if $x \; mod \; k = r_x$, and $y \; mod \; k = r_y$, then $(x+y) \; mod \; k = r_x + r_y$, unless $r_x + r_y \geq k$, in which case $(x+y) \; mod \; k = r_x + r_y - k$. Both these cases can be captured by $(r_x + r_y) \; mod \; k$ (a brief way of saying this would be to observe that *mod* commutes over addition).

$$x +_{M_k} y \stackrel{\text{def}}{=} \{(r_x + r_y) \; mod \; k \mid r_x \in x, r_y \in y\} \tag{3.42}$$

The effect of incrementing and decrementing are defined in terms of addition:

$$+1_{M_k} x \stackrel{\text{def}}{=} x +_{M_k} 1 \tag{3.43}$$

$$-1_{M_k} x \stackrel{\text{def}}{=} x +_{M_k} (-1) \tag{3.44}$$

Subtraction can also be defined in terms of addition, after negating the right argument.

$$x -_{M_k} y \stackrel{\text{def}}{=} x +_{M_k} (-_{M_k} y) \tag{3.45}$$

Like addition, multiplication commutes over addition, so we can multiply the remainders. To see this consider $(i \cdot k + r_1) \cdot (j \cdot k + r_2)$. The result is $i \cdot j \cdot k + i \cdot r_2 \cdot + r_1 \cdot j \cdot k + r_1 \cdot r_2$. Every term is a k factor of k except $r_1 + r_2$. Hence, we can use the following function:

$$x \cdot_{M_k} y \stackrel{\text{def}}{=} \{a \cdot b \bmod k \mid a \in x \text{ and } b \in y\} \tag{3.46}$$

Division, on the other hand, doesn't commute over addition, and there is little information that can be inferred. We will hence punt the ball on division and define:

$$x /_{M_k} y \stackrel{\text{def}}{=} \top_{M_k} = \{i \mid 1 - k \leq i \leq k - 1\} \tag{3.47}$$

For bitwise and, little can be done other than keep track of the sign. We observe that the bitwise and of two negative values is negative, and otherwise the result is non-negative. Based on that observation, we define:

$$x \overset{bit}{\wedge} y \overset{\text{def}}{=} \begin{cases} \{0 \ldots k-1\} & \text{if } \forall r_x \in x, r_x > 0 \text{ or } \forall r_y \in y r_y > 0 \\ \{1-k \ldots 0\} & \text{if } \forall r_x \in x, r_x < 0 \text{ and } \forall r_y \in y r_y < 0 \\ \{1-k \ldots k-1\} & \text{otherwise} \end{cases} \tag{3.48}$$

The definition of bitwise or is based on the observation that the sign of the result is positive if and only if both arguments are positive.

$$x \overset{bit}{\vee} y \overset{\text{def}}{=} \begin{cases} \{0 \ldots k-1\} & \text{if } \forall r_x \in x, r_x > 0 \text{ and } \forall r_y \in y r_y > 0 \\ \{k-1 \ldots 0\} & \text{if } \forall r_x \in x, r_x < 0 \text{ or } \forall r_y \in y r_y < 0 \\ \{1-k \ldots k+1\} & \text{otherwise} \end{cases} \tag{3.49}$$

The definition of bitwise or is based on the observation that the sign of the result is positive if and only if both arguments have differing signs:

$$x \overset{bit}{\oplus} y \overset{\text{def}}{=} \begin{cases} \{0 \ldots k-1\} & \text{if } (\forall r_x \in x, r_x > 0 \text{ and } \forall r_y \in y r_y < 0) \\ & \quad \text{or } (\forall r_x \in x, r_x < 0 \text{ and } \forall r_y \in y r_y > 0) \\ \{1-k \ldots 0\} & \text{if } (\forall r_x \in x, r_x > 0 \text{ and } \forall r_y \in y r_y > 0) \\ & \quad \text{or } (\forall r_x \in x, r_x < 0 \text{ and } \forall r_y \in y r_y < 0) \\ \{1-k \ldots k+1\} & \text{otherwise} \end{cases} \tag{3.50}$$

The shift operations don't yield any useful information in the vast majority of cases, so we use the default operator:

$$x \overset{\leftarrow}{\wedge}_{M_k} y \overset{\text{def}}{=} \top_{M_k} \tag{3.51}$$

$$x \overset{\rightarrow}{\wedge}_{M_k} y \overset{\text{def}}{=} \top_{M_k} \tag{3.52}$$

Theorem 3.7.10 *The unary operator $-_{M_k}$ is sound.*

Proof According to definition 3.2.3 $-_{M_k}$ is sound if the following is satisfied:

$$\text{For all } a \in M_k, \{-x \mid x \in \gamma_{M_k}(a)\} \subseteq \gamma_{M_k}(-_{M_k}(a)$$

Pick any $a \in M_k$.

Either $a \neq \varnothing$ or $a = \varnothing$.

Case 1: $a \neq \varnothing$

Let $a = \{r_1, \ldots r_m\}$

$\{-x \mid x \in \gamma_{M_k}(a)\}$

$= \{-x \mid x \in \gamma_{M_k}(\{r_1 \ldots r_m\})\}$

$= \{-x \mid x \in \{y \mid y \bmod k \in \{r1 \ldots r_m\}\}\}$

$= \{-x \mid x \bmod k \in \{r1 \ldots r_m\}\}\}$

$= \{x \mid -x \bmod k \in \{r1 \ldots r_m\}\}\}$

$= \{x \mid x \bmod k \in \{-r1 \ldots -r_m\}\}\}$

$= \gamma_{M_k}(\{-r1 \ldots -r_m\})$

$= \gamma_{M_k}(-_{M_k}\{r1 \ldots r_m\})$

$= \gamma_{M_k}(-_{M_k}a)$

Case 2: $a = \varnothing$

$\{-x \mid x \in \gamma_{M_k}(a)\}$

$= \{-x \mid x \in \gamma_{M_k}(\varnothing)\}$

$= \{-x \mid x \in \varnothing\}$

$= \varnothing$

$\subseteq \gamma_{M_k}(-_{M_k}(-_{M_k}a)$

In both cases, definition 3.2.3 is satisfied.

■

Theorem 3.7.11 *The binary operator $+_{M_k}$ is sound.*

Proof According to definition 3.2.4, $+_{M_k}$ is sound if

$$\text{For all } a_1, a_2 \in M_k, \{x +_{\mathbb{Z}} y \mid x \in \gamma(a_1), y \in \gamma(a_2)\} \subseteq \gamma_{M_k}(a_1 +_{M_k} a_2)$$

Pick any $a_1, a_2 \in M_k$.

Either ($a_1 \neq \varnothing$ and $a_2 \neq \varnothing$) or else ($a_1 = \varnothing$ or $a_2 = \varnothing$).

Case 1: ($a_1 \neq \varnothing$ and $a_2 \neq \varnothing$)

$Let a_1 = \{r_{1,1} \dots r_{1,m}\}$ and $a_2 = \{r_{2,1} \dots r_{2,n}\}$

$\{x + y \mid x \in \gamma(a_1), y \in \gamma(a_2)\}$

$= \{x + y \mid x \in \gamma(\{r_{1,1} \dots r_{1,m}\}), y \in \gamma(\{r_{2,1} \dots r_{2,n}\})$

$= \{x+y \mid x \in \{w \mid w \bmod k \in \{r_{1,1} \dots r_{1,m}\}\}, y \in \{w \mid w \bmod k \in \{r_{2,1} \dots r_{2,n}\}\}\}$

$= \{x + y \mid x \bmod k \in \{r_{1,1} \dots r_{1,m}\}, y \bmod k \in \{r_{2,1} \dots r_{2,n}\}\}$

$= \{z \mid z \bmod k \in \{r_{1,i} + r_{2,j} \bmod k \mid i \in \{1 \dots m\}, j \in \{1 \dots n\}\}\}$

$= \{z \mid z \bmod k \in \{r_{1,1} \dots r_{1,m}\} +_{M_k} \{r_{2,1} \dots r_{2,n}\}\}$

$= \{z \mid z \bmod k \in a_1 +_{M_k} a_2\}$

$= \{z \mid z \in \gamma_{M_k}(a_1 +_{M_k} a_2)\}$

$= \gamma_{M_k}(a_1 +_{M_k} a_2)$

Case 1: ($a_1 = \varnothing$ or $a_2 = \varnothing$)

Without loss of generality, assume $a_1 = \varnothing$

$Let a_1 = \{r_{1,1} \dots r_{1,m}\}$ and $a_2 = \{r_{2,1} \dots r_{2,n}\}$

$\{x + y \mid x \in \gamma(a_1), y \in \gamma(a_2)\}$

$= \{x + y \mid x \in \gamma(\varnothing), y \in \gamma(a_2)\}$

$= \{x + y \mid x \in \varnothing, y \in \gamma(a_2)\}$

$= \varnothing$

$\subseteq \gamma_{M_k}(a_1 +_{M_k} a_2)$

In both cases, definition 3.2.4 is satisfied. ■

3.7.7 Interval Domain Numerical Operations

The *Interval* domains are parameterized by set of tuples S which partition $\mathbb{Z}$ into a disjoint set of ranges (which must cover every value in $\mathbb{Z}$).

A handy utility function is to map the range represented by the closed interval $[l, h]$ onto the smallest subset of S which covers the set of values $\{x \mid l \leq x \leq h\}$. This function is defined as:

$$\mathrm{A}_S(x, y) \stackrel{\text{def}}{=} \{\langle a, b\rangle \in S \mid a \leq x \leq b \text{ or } a \leq y \leq b\} \tag{3.53}$$

Note that this function is equivalent to $\alpha_{\mathcal{I}_S}(\{i \mid x \leq i \leq y\})$.

An extension of A is the utility function $\hat{\mathrm{A}}_{S\circ} : R \times R \to \mathcal{P}(S)$, which applies the binary operator $\circ$, and then uses A to lift the result onto $\mathcal{I}_S$:

$$\begin{aligned}\hat{\mathrm{A}}_{S\circ}(\langle x_1, y_1\rangle, \langle x_2, y_2\rangle) \stackrel{\text{def}}{=} \quad & \mathrm{A}(\sqcap S, \sqcup S) \\ & \text{where } S = \{x_1 \circ x_2, x_1 \circ y_2, y_1 \circ x_2, y_1 \circ y_2\}\end{aligned} \tag{3.54}$$

We are now ready to define the operators for the $\mathcal{I}_S$ domain. The first is negation, which takes the negation of each component of its argument, and maps them onto S using the A function.

$$-_{\mathrm{I}_S} a \stackrel{\text{def}}{=} \bigcup_{\langle x,y\rangle \in a} \mathrm{A}(-y, -x) \tag{3.55}$$

Absolute value for the *Interval* S domain is defined in very similar manner. For each tuple within the argument, the absolute value of the tuple is computed, and the result is mapped onto S.

$$\begin{aligned}+_{\mathrm{I}_S} a \stackrel{\text{def}}{=} \quad & \textstyle\bigcup_{\langle x,y\rangle \in a} \mathrm{A}(\sqcap S, \sqcup S) \\ & \text{where } S = \{|z| \mid x \leq z \leq y\}\end{aligned} \tag{3.56}$$

Incrementing a term is done by incrementing the bounds of each subrange, and lifting resulting ranges onto S. Decrementing is done in the exact same manner.

$$+_{1_{I_S}} a \stackrel{\text{def}}{=} \bigcup_{\langle x,y\rangle \in a} \mathrm{A}(+_1 x, +_1 y) \tag{3.57}$$

$$-_{1_{I_S}} a \stackrel{\text{def}}{=} \bigcup_{\langle x,y\rangle \in a} \mathrm{A}(-_1 x, -_1 y) \tag{3.58}$$

For bitwise negation, the *Range* bitwise negation operator is applied to each subrange. The resulting subranges are then mapped onto S.

$$\overset{bit}{\neg}_{\mathrm{I}_S} a \stackrel{\text{def}}{=} \bigcup_{\langle x,y\rangle \in a} \mathrm{A}(\overset{bit}{\neg}_R \langle x, y\rangle) \tag{3.59}$$

Addition of two *Interval* S terms is defined as adding each subrange in the cross product of the two terms, then using A to map the result onto S. The total result is collected by a union operator.

$$a +_{\mathcal{I}_S} b \stackrel{\text{def}}{=} \bigcup_{\langle x_1,y_1\rangle \in a, \langle x_2,y_2\rangle \in b} \mathrm{A}(x_1 + x_2, y_1 + y_2) \tag{3.60}$$

Subtraction of two *Interval* S terms is defined very similarly to addition:

$$a -_{\mathcal{I}_S} b \stackrel{\text{def}}{=} \bigcup_{\langle x_1,y_1\rangle \in a, \langle x_2,y_2\rangle \in b} \mathrm{A}(x_1 - y_2, y_1 - x_2) \tag{3.61}$$

Multiplication uses the $\hat{\mathrm{A}}$ function to find the product of each pair of subranges, with the total result is collected by a union operator.

$$a \cdot_{\mathcal{I}_S} b \stackrel{\text{def}}{=} \bigcup_{\langle x_1,y_1\rangle \in a, \langle x_2,y_2\rangle \in b} \hat{\mathrm{A}}_{\mathcal{I}_S \cdot}(\langle x_1, y_1\rangle, \langle x_2, y_2\rangle) \tag{3.62}$$

Division is defined similarly, with the addition of a check for division by zero.

$$a /_{\mathcal{I}_S} b \stackrel{\text{def}}{=} \begin{cases} \top_{\mathcal{I}_S} & \text{if } 0 \in \gamma_{\mathcal{I}_S}(b) \\ \hat{\mathrm{A}}_{\mathcal{I}_S /}(a, b) & \text{otherwise} \end{cases} \tag{3.63}$$

To implement the bitwise binary operations, we apply the corresponding range bitwise operator to each pair of ranges in the cross-product of the two Interval arguments. The result is lifted by the $\mathrm{A_S}$ function, and the collected using the union operator, all three operations can be implemented using the same template.

$$a \overset{bit}{\circ}_{\mathcal{I}_S} b \overset{\text{def}}{=} \bigcup_{\langle S_1,S_2\rangle \in a\times b} \mathrm{A}_S(S_1 \overset{bit}{\circ}_R S_2) \quad \text{where } \circ \in \{\vee, \wedge, \oplus\} \tag{3.64}$$

Theorem 3.7.12 *The unary operator $-_{\mathcal{I}_S}$ is sound.*

Proof According to definition 3.2.3, $-_{\mathcal{I}_S}$ is sound if the following is satisfied:

$$\text{For all } a \in \mathcal{I}_S, \{-x \mid x \in \gamma_{\mathcal{I}_S}(a)\} \subseteq \gamma_{\mathcal{I}_S}(-_{\mathcal{I}_S}(a)$$

Pick any $a \in \mathcal{I}_S$.

Either $a = \{[x_1, y_1], \ldots [x_n, y_n]\}$ or $a = \varnothing$.

Case 1: $a = \{[x_1, y_1], \ldots [x_n, y_n]\}$.

$\{-w \mid w \in \gamma_{\mathcal{I}_S}(a)\}$

$= \{-w \mid w \in \gamma_{\mathcal{I}_S}(\{[x_1, y_1], \ldots [x_n, y_n]\})\}$

$= \{-w \mid w \in \bigcup_{i=1\ldots n}[x_i, y_i]$

$= \{w \mid -w \in \bigcup_{i=1\ldots n}[x_i, y_i]$

$= \{w \mid w \in \bigcup_{i=1\ldots n}[-y_i, -x_i]$

$= \bigcup_{i=1\ldots n}[-y_i, -x_i]$

$= \gamma_{\mathcal{I}_S}(\{[-y_1, -x_1], \ldots [-y_n, -x_n]\})$

$= \gamma_{\mathcal{I}_S}(\{[-y_1, -x_1], \ldots [-y_n, -x_n]\})$

$= \gamma_{\mathcal{I}_S}(-_{\mathcal{I}_S}\{[x_1, y_1], \ldots [x_n, y_n]\})$

$= \gamma_{\mathcal{I}_S}(-_{\mathcal{I}_S} a)$

Case 2: $a = \varnothing$

$\{-w \mid w \in \gamma_{\mathcal{I}_S}(a)\}$

$= \{-w \mid w \in \gamma_{\mathcal{I}_S}(\varnothing)\}$

$= \{-w \mid w \in \varnothing\}$

$= \varnothing$

$\subseteq \gamma_{\mathcal{I}_S}(-_{\mathcal{I}_S} a)$

In both cases, definition 3.2.3 is satisfied. ■

Theorem 3.7.13 *The binary operator* $\cdot_{\mathcal{I}_S}$ *is sound.*

Proof According to definition 3.2.4, $\cdot_{\mathcal{I}_S}$ is sound if

$$\text{For all } a_1, a_2 \in \mathcal{I}_S, \{x + y \mid x \in \gamma(a_1), y \in \gamma(a_2)\} \subseteq \gamma_{\mathcal{I}_S}(a_1 +_{\mathcal{I}_S} a_2)$$

Either ($a_1 = \{[x_{1,1}, y_{1,1}] \ldots [x_{1,n}, y_{1,n}]\}$ and $a_2 = \{[x_{2,1}, y_{2,1}] \ldots [x_{2,m}, y_{2,m}]\}$) or ($a_1 = \varnothing$ or $a_2 = \varnothing$)

Case 1: $a_1 = \{[x_{1,1}, y_{1,1}] \ldots [x_{1,n}, y_{1,n}]\}$ and $a_2 = \{[x_{2,1}, y_{2,1}] \ldots [x_{2,m}, y_{2,m}]\}$

$\{x \cdot y \mid x \in \gamma(a_1), y \in \gamma(a_2)\}$

$= \{x \cdot y \mid x \in \gamma(\{[x_{1,1}, y_{1,1}] \ldots [x_{1,n}, y_{1,n}]\}), y \in \gamma(\{[x_{2,1}, y_{2,1}] \ldots [x_{2,m}, y_{2,m}]\})\}$

$= \{x \cdot y \mid x \in \bigcup_{i=1\ldots n}[x_{1,i}, y_{1,i}], y \in \bigcup_{i=1\ldots m}[x_{2,i}, y_{2,i}]\}$

$\subseteq \bigcup_{i=1\ldots n, j=1\ldots m}[x_{1,i} \cdot x_{2,j}, y_{1,i} \cdot y_{2,j}]$

$= \bigcup_{i=1\ldots n, j=1\ldots m} \overline{A}_{\mathcal{I}_S,\cdot}([x_{1,i}, y_{1,i}], [x_{2,j}, y_{2,j}])$

$= \bigcup_{s\in, j=1\ldots m} \overline{A}_{\mathcal{I}_S,\cdot}([x_{1,i}, y_{1,i}], [x_{2,j}x, y_{2,j}])$

$= \bigcup_{[w_1,z_1]\in a_1, [w_2,z_2]\in a_2} \overline{A}_{\mathcal{I}_S,\cdot}([w_1, z_1], [w_2, z_2])$

$= \bigcup_{[w_1,z_1]\in a_1, [w_2,z_2]\in a_2} \overline{A}_{\mathcal{I}_S,\cdot}([w_1, z_1], [w_2, z_2])$

Case 2: $a_1 = \varnothing$ or $a_2 = \varnothing$.

Without loss of generality, assume $a_1 = \varnothing$.

$\{x \cdot y \mid x \in \gamma(\varnothing), y \in \gamma(a_2)\}$

Type 1	**Type 2**	**Result Type**
Unity	Any	Unity
Range	Mod k	Range
Range	Interval	Range
Range	$\mathbb{Z}$	Range
Mod k_1	Mod k_2	Mod $\sqcap\{k_1, k_2\}$
Mod k	Interval	Mod k
Mod k	$\mathbb{Z}$	Mod k
Interval $\mathcal{I}_{S_1}$	Interval $\mathcal{I}_{S_2}$	Coarser of S_1, S_2
Interval $\mathcal{I}_S$	$\mathbb{Z}$	Interval $\mathcal{I}_S$

Table 3.8: Result of combining abstractions in a binary operator.

$= \{x \cdot y \mid x \in \varnothing, y \in \gamma(a_2)\}$

$= \varnothing$

$\subseteq \{z \mid z \in \gamma(a_1 +_{\mathcal{I}_S} a_2)\}$

In both cases, definition 3.2.4 is satisfied.

■

3.7.8 Mixed-Value Operations

When the abstraction types of two arguments to a binary expression differ, the arguments must be converted to the same abstraction type before they can be combined. table 3.7.8 gives the rules for combining types. The basic hierarchy is $Unity \sqsupset Range \sqsupset Modulo \sqsupset Interval \sqsupset \mathbb{Z}$ (although there a few exceptions, hence table 3.7.8), we will define the effect of the least upper bound operator $\sqcap$ on domain symbols to implement table 3.7.8. $\sqcap$ will be used during expression evaluation to choose the type resulting from a combination of abstractions.

$$\mathcal{D}_1 \sqcup \mathcal{D}_2 \stackrel{\text{def}}{=} \begin{cases} U & \text{if } \mathcal{D}_1 = U \text{ or } \mathcal{D}_2 = U \\ R & \text{if } (\mathcal{D}_1 = R \text{ and } \mathcal{D}_2 \neq U) \\ \text{ or } (\mathcal{D}_1 \neq U \text{ and } \mathcal{D}_2 = R) & \\ M_{\sqcap\{k_1,k_2\}} & \text{if } \mathcal{D}_1 = M_{k_1} \text{ and } \mathcal{D}_2 = M_{k_2} \\ M_k & \text{if } (\mathcal{D}_1 = M_k \text{ and } \mathcal{D}_2 \in \{\mathbb{Z}, \mathcal{I}_S\}) \\ & \text{ and } (\mathcal{D}_2 = M_k \text{ and } \mathcal{D}_1 \in \{\mathbb{Z}, \mathcal{I}_S\}) \\ \mathcal{I}_S & \text{if } (\mathcal{D}_1 = \mathcal{I}_S \text{ and } \mathcal{D}_2 \in \{\mathbb{Z}, \mathcal{I}_S\}) \\ & \text{ and } (\mathcal{D}_2 = \mathcal{I}_S \text{ and } \mathcal{D}_1 \in \{\mathbb{Z}, \mathcal{I}_S\}) \\ \mathcal{I}_{\sqcup\{S_1,S_2\}} & \text{if } \mathcal{D}_1 = \mathcal{I}_{S_1} \text{ and } \mathcal{D}_2 = \mathcal{I}_{S_2} \end{cases} \tag{3.65}$$

A special case is when a concrete integer value is mixed with an abstract value of type *Interval*. In this case, we can achieve better results by providing special mixed-type operators which improve the accuracy of the results. These functions are defined as first converting the the integer argument to a range of width 1, then using the appropriate range operator to combine the integer with each subrange within the *Interval* value. As always, the A function lifts the result, which is collected by a union over all components of the *Interval* value. Two functions are required, depending on the ordering of the values. The one presented for the case where the integer is on the left-hand side and the *Interval* value is on the right. The symmetric case is left as an exercise for the reader.

$$\begin{aligned} k \circ_{\mathbb{Z},\mathcal{I}_S} x \stackrel{\text{def}}{=} & \bigcup_{\langle l,h\rangle \in x} \mathrm{A}_S\langle k,k\rangle \circ_R \langle l,h\rangle \\ & \text{where } \circ \in \{+,-,\cdot,/,mod, \overset{bit}{\vee}, \overset{bit}{\wedge}, \overset{bit}{\oplus}, \curvearrowright, \curvearrowleft\} \end{aligned} \tag{3.66}$$

3.7.9 Bounding Operations

The last group of operations that are needed for abstract integers are the ability to compute the least upper bound and greatest lower bound of two values. Recall that we already defined $x \sqsubseteq y$ if and only if $\gamma(x) \subseteq \gamma(y)$. To find the greatest lower bound $x \sqcap y$ under this ordering, we need to find the value z such that $\gamma(z) = \gamma(x) \cap \gamma(y)$. This may be too aggressive for some domains, so we will allow $\gamma(x \sqcap y) \subseteq \gamma(x) \cap \gamma(y)$, but the closer the subset relation approaches an equality relation, the better. The upper bound $x \sqcup y$ is symmetric. We want to define $\sqcup$ so that $\gamma(x \sqcup y) \supseteq \gamma(x) \cup \gamma(y)$.

The definition of these operators for the *Unity* domain is straightforward. The upper bound of $\top$ and anything else is $\top$, and the lower bound of $\bot$ and anything else is $\bot$.

$$x \sqcap_U y \stackrel{\text{def}}{=} \begin{cases} \bot_U & \text{if } x = \bot_U \text{ or } y = \bot_U \\ \top_U & \text{otherwise} \end{cases}$$

$$x \sqcup_U y \stackrel{\text{def}}{=} \begin{cases} \bot_U & \text{if } x = \bot_U \text{ and } y = \bot_U \\ \top_U & \text{otherwise} \end{cases}$$

For domains whose values are power-sets, the bounding can be done using set intersection and set union. The includes the $B_3, M_k,$ and $\mathcal{I}_S$ domains.

$$x \sqcap_{\mathcal{D}} y \stackrel{\text{def}}{=} x \cap y \quad \text{where } \mathcal{D} \in \{B_3, M_k, \mathcal{I}_S\}$$

$$x \sqcup_{\mathcal{D}} y \stackrel{\text{def}}{=} x \cup y \quad \text{where } \mathcal{D} \in \{B_3, M_k, \mathcal{I}_S\}$$

The last remaining domain is the *Range* domain. The greatest lower bound of two ranges, is $\bot$ if the ranges don't overlap, or if either range is already $\bot$. If the ranges do overlap, then the result, is the portion that overlaps both ranges.

$$x \sqcap_R y \stackrel{\text{def}}{=} \begin{cases} \bot_R & \text{if } x = \bot_R \text{ or } y = \bot_R \\ \bot_R & \text{if } \lceil x \rceil_R < \lfloor y \rfloor_R \text{ or } \lceil y \rceil_R < \lfloor x \rfloor_R \\ \langle max(\lfloor x \rfloor_R, \lfloor y \rfloor_R), min(\lceil x \rceil_R, \lceil y \rceil_R) \rangle & \text{otherwise} \end{cases}$$

The least upper bound of two ranges is $\bot$ only if both ranges $\bot$. Otherwise, the result is the range whose lower bound is smaller of the two lower bounds, and whose upper bound is the larger of the two upper bounds.

$$x \sqcup_R y \stackrel{\text{def}}{=} \begin{cases} \bot_R & \text{if } x = \bot_R \text{ and } y = \bot_R \\ x & \text{if } x \neq \bot_R \text{ and } y = \bot_R \\ y & \text{if } x = \bot_R \text{ and } y \neq \bot_R \\ \langle min(\lfloor x \rfloor_R, \lfloor y \rfloor_R), max(\lceil x \rceil_R, \lceil y \rceil_R) \rangle & \text{otherwise} \end{cases}$$

3.7.10 Pointer Operations

In addition to integer expressions, our abstract interpretation provides limited support for pointers. The operators are listed in table 3.9. For scalar variables and functions, the only operations supported are *reference* (unary `*`) and *dereference* (unary `&`).

ASCII	Symbol	Type	Description
`&`	&	$Id \rightarrow Id\ Ref$	Create a reference to a variable or function.
`*`	$*$	$Id\ Ref \rightarrow Id\ Ref$	Get the value of a variable or function via a reference.

Table 3.9: Pointer operators in the C Wolf system.

The effect of applying a & operator to a variable or function identifier is to create a reference to the variable or function, unless the variable is an array, in which case the result is a pointer to the first element of the array. The effect of applying a $*$ operator to a reference is to fetch the value of variable, function, or array cell referred-to. In the case of an array, if the size of the array is unknown, or the pointer is beyond the bounds of the array, then the result is $\top$.

Applying numeric operators to non-array references or pointer operators to non-reference values results in $\top$.

For pointers into an array, addition and subtraction operations are supported. The result of adding a value k to a pointer into an array is to adjust the pointer by k cells, e.g., adding -4 to to a pointer to element 6 of array A would result in a pointer to element 2 of A. If the value k cannot be precisely determined, then the result is $\top$.

3.8 Expression Evaluation

So far, we have defined how to evaluate the result of an operator applied to one or two values. The next step is how to evaluate an entire expression. In the CWI language, expressions are *pure*, meaning they have no side-effects. As the CWI grammar (fig. 2.3) shows, there are basically four kinds of expression:

- A constant.
- A reference to a user-defined variable, or a system-defined temporary variable (called a *register*).
- A unary expression consisting of an operator applied to a subexpression.

- A binary expression consisting of an operator applied to two subexpressions.

3.8.1 Environments

Before we can discuss how to evaluate an entire expression, we need to deal with how variables and registers are converted to values. They are stored in a data structure known as an *environment*. An environment maps from program variable identifiers and register identifiers to values. Each program variable and register (compiler-generated temporary variables) has a unique static identifier associated with it at compile-time. The identifiers are collected in the set $Var = Var_{Prog} \cup Var_{Reg}$, where Var_{Prog} is the set of variable identifiers, and Var_{Reg} is the set of register identifiers. There is also a set Fun of function identifiers.

There are basically five kinds of values, as follows:

- $\rightarrow_{\mathcal{E}}\ x$ denotes a pointer to variable $x \in Var$ in environment $\mathcal{E}$.
- $\rightarrow_{\mathcal{E}}\ x[k]$ denotes a pointer cell k of the array contained within the variable $x \in Var$ under environment $\mathcal{E}$.
- $\langle a_0, a_1, \ldots a_n \rangle$, denotes an array of values (each a_i is another value of some sort).
- $\rightarrow\ f$ denotes a pointer to function $f \in Fun$.
- $c_{\mathcal{D}}$ denotes a constant value belonging to domain $\mathcal{D}$.

There is some additional environment-related notation. Given an environment $\mathcal{E}$, an identifier v, and a value x, $\mathcal{E}(v) = x$ is used to indicate that the value of v in $\mathcal{E}$ is x. The notation $\mathcal{E}[v \mapsto x]$ denotes an environment which is equal to equal to $\mathcal{E}$ in every respect except for the value of v, which is x. We will call the set of all values Val, and the set of all environments Env. The type of environments is $Var \rightarrow Val$. We will discuss environments in more detail in the next section. For now, all we need to know is that each expression is evaluated in the context of an environment.

The rules for evaluating a expression within the context of an environment $\mathcal{E}$ are:

- If the expression is a constant c, the result of the evaluation is $c_{\mathbb{Z}}$.
- If the expression is a variable or register $v \in Var$, the result of the evaluation is $\mathcal{E}(v)$.

e	$Eval(\mathcal{E}, e)$	**Condition**
c	$c_{\mathbb{Z}}$	$c \in Con$
v	$\mathcal{E}(v)$	$v \in Var$ and v is not an array
v	$\rightarrow_{\mathcal{E}} v[0]$	$v \in Var$ and v is an array
f	$\rightarrow f$	$f \in Fun$
$\& f$	$\rightarrow f$	$f \in Fun$
$\& v$	$\rightarrow_{\mathcal{E}} v$	$v \in Var$ and v is not an array
$\& v$	$\rightarrow_{\mathcal{E}} v[0]$	$v \in Var$ and v is an array
$e_0 \circ e_1$	$\rightarrow_{\mathcal{E}} v[i \circ k]$	$eval(\mathcal{E}, e_0) = \rightarrow_{\mathcal{E}} v[i]$ and $eval(\mathcal{E}, e_1) = k \in \mathbb{Z}$ and $\circ \in \{+, -\}$
$*v$	a_k	$eval(\mathcal{E}, e_0) \rightarrow_{\mathcal{E}} v[k]$ and $k \in \mathbb{Z}$ and $eval(\mathcal{E}, v) = \langle a_0, \ldots a_n \rangle$ and $0 \leq k \leq n-1$
$*v$	$\mathcal{E}(v)$	$eval(\mathcal{E}, e_0) \rightarrow_{\mathcal{E}} v$
$*e_0$	$\rightarrow f$	$eval(\mathcal{E}, e_0) = \rightarrow f$
$\circ e_0$	$\circ_{\mathcal{D}} v_{\mathcal{D}}$	$eval(\mathcal{E}, e_0) = v_T$
$e_0 \circ e_1$	$v_{0_{\mathcal{D}}} \circ_{\mathcal{D}} v_{1_{\mathcal{D}}}$	$eval(\mathcal{E}, e_0) = v_{0_{\mathcal{D}}}$ and $eval(\mathcal{E}, e_1) = v_{1_{\mathcal{D}}}$
$e_0 \circ e_1$	$v_{0_{\mathcal{I}_S}} \circ_{\mathcal{I}_S, \mathbb{Z}} v_{1_{\mathbb{Z}}}$	$eval(\mathcal{E}, e_0) = v_{0_{\mathcal{I}_S}}$ and $eval(\mathcal{E}, e_1) = v_{1_{\mathbb{Z}}}$
$e_0 \circ e_1$	$v_{0_{\mathbb{Z}}} \circ_{\mathbb{Z}, \mathcal{I}_S} v_{1_{\mathcal{I}_S}}$	$eval(\mathcal{E}, e_0) = v_{0_{\mathbb{Z}}}$ and $eval(\mathcal{E}, e_1) = v_{1_{\mathcal{I}_S}}$
$e_0 \circ e_1$	$f_{\mathcal{D}_0 \rightarrow \mathcal{D}_2}(v_0) \circ_{\mathcal{D}_2} f_{\mathcal{D}_1 \rightarrow \mathcal{D}_2}(v_1)$	$eval(\mathcal{E}, e_0) = v_{0_{\mathcal{D}_0}}$ and $eval(\mathcal{E}, e_1) = v_{1_{\mathcal{D}_1}}$ and $\mathcal{D}_2 = \mathcal{D}_0 \sqcup \mathcal{D}_1$
Other	$\top$	

Figure 3.3: The abstract expression evaluation function, $Eval(\mathcal{E}, e)$.

- If the expression is a function $f \in Fun$, then result of the evaluation is a pointer $\rightarrow f$.
- If the expression is a numeric unary expression of the form $\circ e_0$, then first e_0 is evaluated resulting in a value v_0 of some abstract type T. The final result is $\circ_T\ v_0$, following the rules outlined in section 3.7.
- If the expression is a unary pointer dereference operation of the form $*\ e_0$, then e_0 is evaluated, resulting in a pointer value. If the pointer is a variable pointer of the form $\rightarrow_{\mathcal{E}'} v$, then the result of the expression is $\mathcal{E}'(v)$. If the pointer is a function pointer of the form $\rightarrow f$, then the result of the expression is still $\rightarrow f$ (dereferences of function pointer are allowed, but superfluous).
- If the expression is a unary pointer creation operation of the form $\&\ x$, then the result is $\rightarrow_{\mathcal{E}} x$ if x is a (non-array) variable, and $\rightarrow f$ if f is a function. For a pointer to cell k of an array whose base pointer is in variable x, the expression $\rightarrow_{\mathcal{E}} x[k]$ is used.
- If the expression is binary $(e_1 \circ e_2)$, then first e_1 and e_0 are evaluated, resulting in values v_1 and v_2 of abstract type T_1 and T_2, respectively. If $T_1 = T_2$, then the values are combined using $v_1 \circ_{T_1} v_2$, as described in section 3.7. If $T_1 \neq T_2$ then the rules of section 3.7.8 are used to resolve the type to some type T_3, and then the values are combined by the operator $v_1 \circ_{T_3} v_2$.
- There are some expressions which we didn't discuss in section 3.7, such as dereferencing a pointer into dynamic memory, or accessing a field of a struct or union. For any such operation, the result is always $\top$.
- Note that C allows *pointer arithmetic* operations, such as adding an integer to a pointer. We allow such cases when the pointer is a pointer to an array element, in which case the two values are combined to result in a pointer to some element within the array. In other cases, such as pointer arithmetic with a function pointer, the result is always $\top$.

These rules are formalized by the function $eval(\mathcal{E}, expr)$, which evaluates the expression $expr$ under environment $\mathcal{E}$. Note that Con is the set of all integer constants. Due to its size, this function appears in figure 3.3.

Environment Operations

There are two additional operations we need to define on environments. The first is the *merge* of two environments, denoted $\mathcal{E}_1 \sqcup \mathcal{E}_2$, is defined as:

$$\mathcal{E}_1 \sqcup \mathcal{E}_2 \stackrel{\text{def}}{=} \{\langle v, \mathcal{E}_1(v) \sqcup \mathcal{E}_2(v)\rangle \mid v \in V\} \tag{3.67}$$

Where the $\sqcup$ of two values is the least upper bound determined by the $\sqsubseteq$ relation. The purpose of the merge operation is to compute the least upper bound of two environments, resulting in an environment whose abstract values capture all the concrete values in the γ of $\mathcal{E}_1$ and $\mathcal{E}_2$.

The second operation is the *join* of two environments, denoted as $\mathcal{E}_1 \bowtie \mathcal{E}_2$. The result of $\mathcal{E}_1 \bowtie \mathcal{E}_2$ is to combine the bindings of both environments into a single environment. In cases where both environments bind the same variable, $\mathcal{E}_1$ overrides $\mathcal{E}_2$. Hence, the join operation can be defined as:

$$\mathcal{E}_1 \bowtie \mathcal{E}_2 \stackrel{\text{def}}{=} \mathcal{E}_2[v_1 \mapsto \mathcal{E}_1(v_1)][v_2 \mapsto \mathcal{E}_1(v_2)] \ldots [v_n \mapsto \mathcal{E}_n(v_n)] \tag{3.68}$$

Where the variables bound by $\mathcal{E}_1$ are $\{v_1, v_2, \ldots v_n\}$. We will also use the notation

$$\mathcal{E}_1 \Cup \mathcal{E}_2$$

to represent the join of two environments whose variables are known to be disjoint.

3.9 Statement Interpretation

Unlike expressions, statements have side effects. A statement is interpreted in the context of a *location*, a *local environment*, a *activation record stack*, and a *static environment*. The *location* is a particular statement anywhere within the program. Each statement has a unique location, taken from the set *Loc* of all locations. The *local environment* is a mapping from local variables to values. The *static environment* is a mapping from global variables to values. The *activation record stack* contains the location and environment for all active functions. The initial stack contains the activation record for *main* only. New records are pushed onto the stack every time a non-tail function call occurs. Records are popped of the stack every time a function returns. The current location and environment are always on

Kind	**Form**	**Meaning**
Assignment	$e_0 = e_1$;	Expression e_1 is evaluated and stored in the variable described by (L-value) expression e_0.
Goto	`goto` loc ;	The execution jumps to location loc.
Conditional Goto	`if` e `then goto` loc_1 `else goto` loc_2	If e evaluates to a true (nonzero) result, then execution jumps to loc_1. Otherwise, execution flows to loc_2.
Switch	`switch` (e) `{` $v_0 : loc_0 \ldots v_k : loc_k$ `}`	Execution jumps to location loc_i when evaluation of e results in value v_i.
Call	`call` $e_0(e_1 \ldots e_n)$;	The function resulting from evaluation of expression e_0 is called with n arguments resulting from evaluation of expressions $e_1 \ldots e_n$.
Return	`return` e ;	The current function returns with the return value equal to the result of evaluating e.
Halt	`halt` e ;	The program halts with exit status equal to the result of evaluating e.

Table 3.10: Summary of CWI Statements.

the top of the stack. We will use the term *configuration* to refer to a particular activation record stack and static environment.

The effect of interpreting a statement is one or more new configuration. The precise effect depends on the kind of statement. The statement kinds are summarized in table 3.9. *Assignment* statements modify the either the static or local environment, and always update the location to the next statement after the current one (called the *static successor*). *Goto* statements modify the current location without affecting the environment. *Conditional goto* and *switch* statements result in multiple configurations, each containing a different environment and location. *Call* statements generate a configuration with a new activation record pushed on to the top of the stack. If the call is a *tail call*, then the new activation record replaces the old top of the stack, so that the stack size remains constant. *Return* statements pop the top activation record from the stack, resulting a configuration whose location is the static successor of the calling statement, and whose environment reflects the changes to static variables made during the function call. A *halt* statement results in a configuration with an empty stack, an environment reflecting the final value of all static variables, and the exit status (which is stored in register 0).

Notation	Meaning
$\langle \mathcal{E}, \ell \rangle$	An activation record consisting of environment $\mathcal{E}$ and program location ℓ.
$\langle \mathcal{E}, \ell \rangle :: S$	A stack where $\langle \mathcal{E}, \ell \rangle$ is the top activation record and S is the rest of the stack.
Λ	The empty stack.
$\langle \mathcal{E}_S, \langle \mathcal{E}, \ell \rangle :: S \rangle$	A configuration where $\langle \mathcal{E}, \ell \rangle :: S$ is the activation stack, and $\mathcal{E}_S$ is the static environment.

Table 3.11: Stack-related notation

.

3.9.1 Notation

Some notation is required in order to express the semantics of statement interpretation. A configuration will be represented by $\langle \mathcal{E}_S, \langle \mathcal{E}_k, \ell_k \rangle :: \langle \mathcal{E}_{k-1}, \ell_{k-1} \rangle :: \ldots :: \langle \mathcal{E}_0, \ell_0 \rangle \rangle$, where $\mathcal{E}_S$ is the static environment, and $\langle \mathcal{E}_k, \ell_k \rangle :: \langle \mathcal{E}_{k-1}, \ell_{k-1} \rangle :: \ldots :: \langle \mathcal{E}_0, \ell_0 \rangle$ is the program stack. The leftmost tuple is the top of the stack, so in this expression, $\mathcal{E}_0$ is the current local environment and ℓ_0 is the current program location.

:: is the *stack constructor*, so $F :: S$ will represent a stack constructed by pushing frame F onto stack S. The empty stack will be denoted by Λ. Note that :: is right-associative. This notation is summarized in table 3.11.

The semantic effects of statements will be presented in an operational style, using rules of the form

$$\frac{config_0 \quad stmt}{config_1 \ldots config_k} \quad (cond)$$

or, when things get too wide:

$$\frac{config_0 \quad stmt}{config_1 \ldots config_k}$$

$$\text{when} \quad (cond)$$

The interpretation of a rule is "$config_0$, whose current location refers to *stmt*, has successor configurations $config_1 \ldots config_k$ if the side-condition *cond* is satisfied.

3.9.2 Assignment Statements

The effect of an assignment is to modify either the local environment or global (static) environment. All assignment statements have two subexpressions, a *source* expression, and a *target* expression. The source expression is evaluated to get a value. This value is stored in the variable specified by the *target* expression. The target expression is a special kind of expression known as an *L-value expression*. L-value expressions are restricted expressions which evaluate to a storage location. The allowed forms of an L-value expression are:

- A variable identifier.
- An array index expression.
- A pointer dereference operation.

Our interpreter supports variable L-values completely, doesn't support array L-values at all, and provides limited support for pointer L-values. We have five rules for assignment statement interpretation. The first rule is relatively straightforward. The effect of assignment to a static variable is to bind the variable in the static environment. A minor detail that is omitted here is that the value must be cast to the abstract type of v. We are assuming that the environment binding mechanism handles this for us automatically.

$$(R1)\quad \frac{\langle \mathcal{E}_S, \langle \mathcal{E}_0, \ell_0 \rangle :: S \rangle \quad v = e_1}{\langle \mathcal{E}_S[v \mapsto x], \langle \mathcal{E}_0, succ(\ell) \rangle \rangle :: S} \quad (eval(\mathcal{E}_0 \Cup \mathcal{E}_S, e_1) = x \text{ and } v \text{ is static})$$

The second rule handles the case of an assignment to a local variable. In this case, the local environment is updated.

$$(R2)\quad \frac{\langle \mathcal{E}_S, \langle \mathcal{E}_0, \ell_0 \rangle :: S \rangle \quad v = e_1}{\langle \mathcal{E}_S, \langle \mathcal{E}_0[v \mapsto x], succ(\ell) \rangle \rangle :: S} \quad (eval(\mathcal{E}_0 \Cup \mathcal{E}_S, e_1) = x \text{ and } v \text{ is local})$$

The third rule handles the case of an assignment via pointer, when the pointer points into the static environment.

$$(R3)\quad \frac{\langle \mathcal{E}_S, \langle \mathcal{E}_0, \ell_0 \rangle :: S \rangle \quad e_0 = e_1}{\langle \mathcal{E}_S[v \mapsto x], \langle \mathcal{E}_0, succ(\ell) \rangle \rangle :: S} \quad \left(\begin{array}{c} eval(\mathcal{E}_0 \Cup \mathcal{E}_S, e_0) = \rightarrow_{\mathcal{E}_S} v \\ \text{and } eval(\mathcal{E}_0 \Cup \mathcal{E}_S, e_1) = x \end{array} \right)$$

The fourth and fifth rules handle the complex case where there is an assignment via pointer, and the the pointer points into a local environment somewhere in the program

stack. The value pointed-to could be a scalar variable or an array variable. The kind (scalar or array) is determined by the form of the pointer value.

$$(R4)\quad \frac{\langle \mathcal{E}_S, \langle \mathcal{E}, \ell\rangle :: \ldots \langle \mathcal{E}_i, \ell_i\rangle \ldots :: S\rangle \quad * e_0 = e_1}{\langle \mathcal{E}_S, \langle \mathcal{E}, succ(\ell)\rangle :: \ldots \langle \mathcal{E}_i[v \mapsto x], \ell_i\rangle \ldots :: S\rangle} \quad \left(\begin{array}{l} eval(\mathcal{E}, e_0) =\!\!\rightarrow_{\mathcal{E}_i} v \\ \text{and } eval(\mathcal{E}, e_1) = x \end{array}\right)$$

$$(R5)\quad \frac{\langle \mathcal{E}_S, \langle \mathcal{E}, \ell\rangle :: \ldots \langle \mathcal{E}_i, \ell_i\rangle \ldots :: S\rangle \quad * e_0 = e_1}{\langle \mathcal{E}_S, \langle \mathcal{E}, succ(\ell)\rangle :: \ldots \langle \mathcal{E}_i[v \mapsto \langle a_0, \ldots a_{k-1}, x, a_{k+1} \ldots a_{n-1}], \ell_i\rangle \ldots :: S\rangle}$$

$$\text{when} \quad \left(\begin{array}{l} eval(\mathcal{E}, e_0) =\!\!\rightarrow_{\mathcal{E}_i} v[k] \\ \text{and } eval(\mathcal{E}, e_1) = x \\ \text{and } \mathcal{E}(v) = \langle a_0, \ldots a_n\rangle \\ \text{and } 0 \le k \le n-1 \end{array}\right)$$

The final rule handles the case where the lvalue is too complex. In this case we simply ignore the assignment.

$$(R6)\quad \frac{\langle E_S, \langle \mathcal{E}, \ell\rangle :: S\rangle \quad * e_0 = e_1}{\langle \mathcal{E}, succ(\ell)\rangle :: S} \quad (eval(\mathcal{E}, e_0) = \top)$$

3.9.3 Goto Statements

Goto statements are a fairly easy case to handle. The effect of a goto is to modify the current location, without affecting the environment. Just a single rule is required, with no side-conditions:

$$(R7)\quad \frac{\langle \mathcal{E}_S, \langle \mathcal{E}_0, \ell_0\rangle :: S\rangle \quad \texttt{goto}\ \ell';}{\langle \mathcal{E}_S, \langle \mathcal{E}_0, \ell'\rangle :: S\rangle}$$

3.9.4 Conditional Goto Statements

The effect of a conditional is twofold. First, either one or both branches are selected depending on outcome of the evaluation of the condition. Next, for each successor, the environment is constrained based on information gleaned from the predicate. We will defer the discussion of the details of constraints for now. The notation $[\mathcal{E} \mid P]$ will indicate the environment resulting from constraining $\mathcal{E}$ based on the assumption that P holds true.

There are two rules for conditional goto statements, based on the possible values of the condition.

$$(R8)\quad \frac{\langle \mathcal{E}_S, \langle \mathcal{E}_0, \ell_0\rangle :: S\rangle \quad \texttt{if } e_0 \texttt{ then goto } \ell_1 \texttt{ else goto } \ell_2;}{\langle [\mathcal{E}_S \mid e_0], \langle [\mathcal{E}_0 \mid e_0], \ell_1\rangle :: S\rangle} \quad (\sharp\sharp \in eval_{\mathcal{E}_S \cup \mathcal{E}_0}(e_0))$$

$$(R9)\quad \frac{\langle \mathcal{E}_S, \langle \mathcal{E}_0, \ell_0\rangle :: S\rangle \quad \texttt{if } e_0 \texttt{ then goto } \ell_1 \texttt{ else goto } \ell_2;}{\langle [\mathcal{E}_S \mid \neg e_0], \langle [\mathcal{E}_0 \mid \neg e_0], \ell_2\rangle :: S\rangle} \quad (\mathit{ff} \in eval_{\mathcal{E}_S \cup E_0}(e_0))$$

Note that both (R8) and (R9) can simultaneously trigger, resulting in two successors from the same conditional goto.

3.9.5 Switch Statements

The most complex rule is for switch statements. A switch statement evaluates an integer expression, and then selects a branch. Each branch has an integer constant guard value. The branch is selected if and only if the value of the expression is equal to the guard value.

$$(R10)\quad \frac{\langle \mathcal{E}_S, \langle \mathcal{E}_0, \ell_0\rangle :: S\rangle \quad \texttt{switch } (e_0)\{\ \texttt{case } x_1\texttt{: } \ell_1; \ldots \texttt{case } x_n\texttt{: } \ell_n\texttt{;default : } \ell_d;\}}{\langle [\mathcal{E}_S \mid e_0 = x_i], \langle [\mathcal{E}_0 \mid e_0 = x_i], \ell_i\rangle :: S\rangle}$$

$$\text{when} \quad (x_i \in \gamma(eval_{\mathcal{E}_S \uplus \mathcal{E}_0}(e_0)) \text{ and } i \in \{1 \ldots n\})$$

Note that this rule can trigger up to n times for the same configuration, resulting in as many as n successors.

A separate rule handles the default branch:

$$(R11)\quad \frac{\langle \mathcal{E}_S, \langle \mathcal{E}_0, \ell_0\rangle :: S\rangle \quad \texttt{switch } (e_0)\{\ \texttt{case } x_1\texttt{: } \ell_1; \ldots \texttt{case } x_n\texttt{: } \ell_n\texttt{;default : } \ell_d\ ;\}}{\langle [\mathcal{E}_S \mid e_0 \notin \{x_1 \ldots x_n\}], \langle [\mathcal{E}_0 \mid e_0 \notin \{x_1 \ldots x_n\}], \ell_d\rangle :: S\rangle}$$

$$\text{when} \quad (\gamma(eval_{\mathcal{E}_S \uplus \mathcal{E}_0}(e_0)) - \{x_1 \ldots x_n\} \neq \varnothing)$$

It should be noted that both (R10) and (R11) can be triggered simultaneously.

3.9.6 Call Statements

The effect of a call statement is to push a new frame onto the stack. The location of the current frame is set to the static successor of the current statement, and then a new activation record is generated in which the local environment is empty except for the arguments being passed, and the location is the starting address of the called function.

$$(R12)\quad \frac{\langle E_S, \langle E_0, \ell_0\rangle :: S\rangle \quad \texttt{call}\ e_0\ (e_1, \ldots e_k);}{\langle E_S, \langle [a_1 \mapsto x_1, a_2 \mapsto x_2, \ldots a_k \mapsto x_k], \ell_f\rangle :: \langle E_0, succ(\ell_0)\rangle :: S\rangle}$$

$$\text{when}\quad \left(\begin{array}{l} eval_{E_S \uplus E_0}(e_0) =\!\!\rightarrow f \\ f \in Fun \\ eval_{E_S \uplus E_0}(e_i) = x_i \quad (i = 1 \ldots k) \\ \ell_f \text{ is the starting location of } f \\ a_1 \ldots a_k \text{ are the formal arguments of } f \end{array}\right)$$

3.9.7 Return Statements

The effect of a return statement is to pop the topmost frame from the stack. The frame below has already been configured by the call statement to have the correct return location (the static successor of the calling statement). The local environment of the caller is modified so that register 0 contains the return value. We have two rules, one for the usual case we just described, and another for the case where there is no caller and the program is terminating via a return from *main*.

$$(R13)\frac{\langle \mathcal{E}_S, \langle E_0, \ell_0\rangle :: \langle E_1, \ell_1\rangle :: S\rangle \quad \texttt{return}\ e;}{\langle \mathcal{E}_S, \langle E_1[r\#0 \mapsto x], \ell_1 :: S\rangle\rangle} \quad (eval_{\mathcal{E}_0 \uplus E_S}(e) = x)$$

$$(R14)\frac{\langle \mathcal{E}_S, \langle E_0, \ell_0\rangle :: \Lambda\rangle \quad \texttt{return}\ e;}{\langle \mathcal{E}_S[r\#0 \mapsto x], \Lambda\rangle} \quad (eval_{\mathcal{E}_0 \uplus E_S}(e) = x)$$

3.9.8 Halt Statements

The effect of a halt statement is nearly identical to a return from main. The only difference is that the program stack is wiped out no matter how large it previously was.

$$(R15)\frac{\langle \mathcal{E}_S, \langle E_0, \ell_0\rangle :: S\rangle \quad \texttt{halt}\ e;}{\langle \mathcal{E}_S[r\#0 \mapsto x], \Lambda\rangle} \quad (eval_{\mathcal{E}_0 \uplus E_S}(e) = x)$$

3.9.9 Constraint Analysis

Conditional goto and switch statements generally have multiple successors, depending on the outcome of a test expression. As we mentioned, this information can be used to increase the accuracy of the abstract values within the environment. This is done

by a constraint analyzer, which takes an environment, and a conditional expression which is assumed to be true. The constraint analyzer looks at the syntactic form of the constraint, and performs an action according to the following rules (these rules assume that $\mathcal{E}$ is the current environment):

Form	**Result**
$v = e$	$\mathcal{E}[v \mapsto eval_{\mathcal{E}}(e)]$
$v \neq e$	$\mathcal{E}[v \mapsto (\mathcal{E}(v) \sqcap \langle minint, \lceil eval_{\mathcal{E}}(e)\rceil - 1\rangle) \sqcup$ $(\mathcal{E}(v) \sqcap \langle \lfloor eval_{\mathcal{E}}(e)\rfloor + 1, maxint\rangle)]$
$v \leq e$	$\mathcal{E}[v \mapsto \mathcal{E}(v) \sqcap \langle minint, \lceil eval_{\mathcal{E}}(e)\rceil\rangle$
$v < e$	$\mathcal{E}[v \mapsto \mathcal{E}(v) \sqcap \langle minint, \lceil eval_{\mathcal{E}}(e)\rceil - 1\rangle$
$v > e$	$\mathcal{E}[v \mapsto \mathcal{E}(v) \sqcap \langle \lfloor eval_{\mathcal{E}}(e)\rfloor + 1, maxint\rangle$
$v \geq e$	$\mathcal{E}[v \mapsto \mathcal{E}(v) \sqcap \langle \lfloor eval_{\mathcal{E}}(e)\rfloor, maxint\rangle$
$e \leq v$	Same as when $v \geq e$
$e < v$	Same as when $v > e$
$e > v$	Same as when $v < e$
$e \geq v$	Same as when $v \leq e$
v	Same as when $v \neq 0$
$\neg e$	Negate e symbolically to get e', then constrain based on e'
$e_1 \wedge e_2$	Constrain $\mathcal{E}$ under e_1 to get $\mathcal{E}'$, then constrain $\mathcal{E}'$ under e_2 to get final result.
$e_1 \vee e_2$	Constrain $\mathcal{E}$ under e_1 to get $\mathcal{E}'$, then constrain $\mathcal{E}$ under e_2 to get $\mathcal{E}''$. Final result is $E' \sqcup E''$.

A small is example of this is the expression $x < y$ where $x = \langle 1, 5\rangle$ and $y = \langle 3, 8\rangle$. The constraint analyzer will produce an unchanged environment in the case where the condition holds true. In the case where the condition is false, the the environment will be $x = \langle 4, 5\rangle$ and $y = \langle 3, 4\rangle$.

3.10 State Space Generation

We are now ready to describe how to generate an entire state space. Given an initial configuration, we can use the rules of section sec:stmts to compute its successors. The successors can be checked for configurations which are "new" (meaning their successors haven't been computed yet). This process is essentially computing the transitive closure of the successor relation between configurations. If we save the configurations and the transitions between configuration, we have a Kripke structure which models every possible execution path of the program. The algorithm to do this is presented in figure 3.4. Note

$New := Q := \{q_0\}$;

$Tr := \varnothing$;

While $(New \neq \varnothing)$ {

  Pick any $q \in New$;

  $New := New - \{q\}$;

  $New := New \cup (succ(q) - Q)$;

  $Q := Q \cup succ(q)$;

  $Tr := Tr \cup q \times succ(q)$

}

$\langle Q, Tr \rangle$ is the final set of states and transitions.

Figure 3.4: Polyvariant state space generation algorithm.

that in figure 3.4, $succ(q)$ refers to a function which computes the successors of configuration q using rules (R1) through (R15).

An alternative algorithm, which uses much less space at the cost of much less accuracy is to merge all environments which correspond to the same program location. Such an approach is called *monovariant* over call stacks, as opposed to the first algorithm which is *polyvariant* over call stacks. A rough sketch of a monovariant state space generation algorithm is given in figure 3.5. This algorithm only maintains one state per call stack and program location. When a state is generated, it is checked against the other states to see if a similar state (with the same call stack and location) already exists. If a matching state exists, then the environments of the two states are merged. If the resulting environment is different than its earlier value, then the set of states is updated, and the state is put back in the New set to re-generate its successors. The results are stronger than traditional data-flow analysis, but weaker than full polyvariance.

To get traditional data-flow analysis, one need on change the condition from "same call stack and location" to just "same location." Such an algorithm will generate a single environment for every program location, merging values over all call paths.

3.11 Labeling

The final step in the model generation process is the attachment of labels to the transitions. Let $Config$ be the set of all possible configurations and Act be the set of actions (a.k.a. labels). We will define a function $\mathcal{L} : Config \times Config \rightarrow Act^*$ which takes an unlabeled transition between two configurations, and returns a (possibly empty) sequence of labels that should be attached to the transition.

Consider the transition $\langle \mathcal{E}_S, \langle \mathcal{E}_0, \ell_0 \rangle :: S \rangle \rightarrow \langle \mathcal{E}'_S, \langle \mathcal{E}'_0, \ell'_0 \rangle :: S' \rangle$, which was generated The rules for labeling this transition are:

- Let s be the sequence of labels associated with the transition. Initially, $s = \epsilon$.
- For each variable v accessed in an expression by the statement at location ℓ_0, if v appears in $read(v) \Rightarrow a$ event, the label a is appended to s. The order of these labels doesn't matter, if there is more than one.
- If the statement at location ℓ_0 is an assignment statement which writes to a variable

$New := Q := \{q_0\}$;

$Tr := \varnothing$;

While $(New \neq \varnothing)$ {

  Pick any $q \in New$;

  $New := New - \{q\}$;

  for each $q' \in succ(q)$ do;

    if ($\exists q'' \in Q$. q'' and q'

      have the same call stack and location) then {

        if $q'' \neq q'' \sqcup q'$then {

          $Q := Q \cup \{q'' \sqcup q'\} - \{q''\}$;

          $New := New \cup \{q'' \sqcup q'\} - \{q''\}$;

          $Tr := Tr \cup \{\langle q, q''\rangle\} - \{\langle q, q''\rangle\}$;

        }

    } else {

        $Q := Q \cup \{q'\}$;

        $New := New \cup \{q'\}$;

        $Tr := Tr \cup \{\langle q, q'\rangle\}$;

    }

  }

}

$\langle Q, Tr\rangle$ is the final set of states and transitions.

Figure 3.5: Monovariant state space generation algorithm.

v that appears in a $write(v) \Rightarrow a$ event, then the label a is appended to s. The single assignment structure of the intermediate language guarantees that there can be at most one such label per transition examined.

- If the statement at location ℓ_0 is a call to function f, and f appears in a $call(f) \Rightarrow a$ event, then the label a is appended to s.

- If the source-line of ℓ'_0 is line i of file f, and $f : i$ appears in an $exec(f : i) \Rightarrow a$ event, and the source-line of ℓ_0 is not the same line (line i of file f), then the label a is appended to s.

- If the statement at location ℓ_0 is an assignment statement which writes to a variable v that appears in a $watch(v\ relOp\ c) \Rightarrow a$ event, and the condition is possibly true in $E'_S \Cup E'_0$, then the label a is appended to s. If more than one watch event's condition is satisfied, then the label associated with the watch which appears first in the label map file is chosen, and the other labels are ignored. Furthermore, the system applies the constraint analyzer to $E'_S \Cup E'_0$, possibly resulting in two successors, one with a labeled transition, and one without a labeled transition. The environment without a labeled transition is recursively checked to see if it satisfies any watch conditions, and if so, it too is split. This process repeats until no more watch events are satisfied.

The astute reader may have noted that the algorithm of figure 3.4 produces an excessive number of states. In particular, there is no need to generate a separate state for each assignment within a basic block — the only case where this is necessary is when the assignment generates a label. We view this mainly as an implementation issue, which can be handled by labeling transitions on the fly and skipping over configurations within a basic block whose outgoing transition will not be labeled. The important thing is that the algorithm described is theoretically sound. Furthermore, there are a number of ways of reducing the state space, which will be described in detail in the next chapter.

Chapter 4

Implementation

The C Wolf system has been implemented as a collection of commands which can be run from the shell on the Linux operating system. The set of commands is listed in table 4.1. The three essential commands are `cwcc`, `cwld`, and `cwmb`. `cwcc` compiles C code into CWI bytecode, a high-level intermediate code organized into a control-flow graph. `cwld` links two or more CWI files into a single CWI file, resolving global cross-file references in the process. `cwmb` generates a model from a CWI file, abstraction map file, and label map file. The output of `cwmb` can be either a `cwb-nc`-readable LTS, a graphical depiction, or a human-readable textual description of the model. The other, nonessential, commands are for debugging and visualization purposes.

Following the UNIX paradigm, all the `cwolf` commands are *file transformers*, which accept one or more files are as input and generate an output file as a result. In a typical session, a user will first compile each `.c` source file with `cwcc`, then link the resulting `.cwi` files into a single program with `cwld`. Once this has been done, label map (`.lm`) and abstraction map (`.am`) source files are created using a text editor. When all three files (`.cwi`, `.am`, and `.lm`) are ready, the model builder `cwmb` is applied to them, producing an automaton in `cwb-nc` format. The suffix of the automaton file must match one of the process algebras supported by `cwb-nc`, such as `.ccs` for Milner's CCS. Finally, the `cwb-nc` model checker can be executed and the automaton file can be loaded. Once loaded, the automaton can be minimized using strong or weak bisimulation (although it should be noted that `cwmb` is capable of generating pre-minimized automata so this step isn't necessary), and then

Command	Description
`cwcc`	Compile a C source file into a CWI file.
`cwld`	Link two or more CWI files into a single CWI file.
`cwcalls`	Extract the call graph from a CWI file and view it.
`cwflow`	Extract the control flow graph from a CWI file and view it.
`cwdump`	View the detailed contents of a CWI file.
`cwmb`	Construct a transition system from a CWI file, a label map file, and an abstraction map file. The system can be output in graphical or Concurrency Workbench format.

Table 4.1: Shell-level commands provided by the C Wolf system.

compared for equivalence to a design expressed in a process algebra, (again, using strong or weak bisimulation as the notion of equivalence). Additionally, the automaton can be checked to see if it satisfies temporal logic formulas given in a branching-time temporal logic such as alternation-free μ-calculus or CTL*.

4.1 System Architecture

The system is implemented as a collection of 20 modules written in the Standard ML programming language. Each module is a collection of inter-related abstract data types. An idealized view of the modules and the dependency relationship is depicted in figure 4.1 (the full dependency relation is approximately equal to the transitive closure of the depicted relation).

At the base of the system are the *ckit*, *smlnj-lib*, and *baselib* modules. The ckit module is the C parser. The smlnj-lib module is provided as part of the Standard ML of New Jersey language implementation, and provides basic container types like hash tables and ordered trees. Both are freely available for non-commercial use from Lucent Technologies / Bell Labs. The baselib module provides some additional container types and functions.

The next layer of the system consists of interfaces to foreign programs and libraries. This layer includes the *davincilib*, *dotlib*, *cpp*, and *cki* modules. The davincilib and dotlib modules translate generic graphs into formats compatible with *daVinci* and *dot*, respectively. Both daVinci and dot are graph rendering tools. The *cpp* module invokes the GNU C Preprocessor as a subprocess. The *cki* module interfaces with *ckit*, provides additional

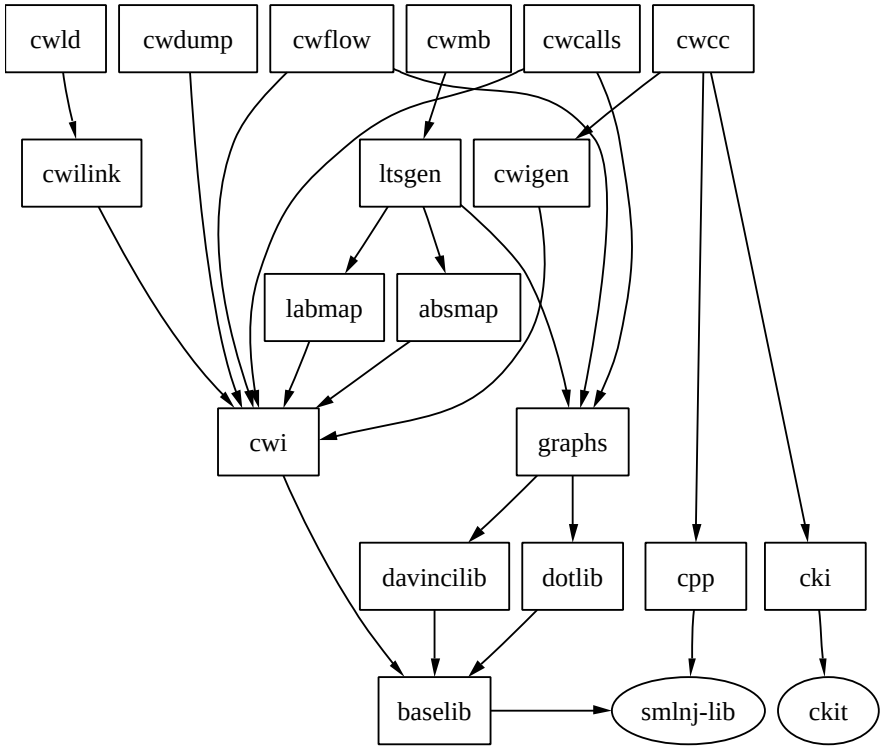

Figure 4.1: Module-level architecture of the C Wolf system.

abstractions, and hides ckit details from the rest of the system.

At the heart of C Wolf lie the core data structures embodied within the *labmap*, *ltsgen*, *graphs*, and *cwi* modules. The cwi module contains the basic data types for the C Wolf bytecode, including routines for reading and writing bytecode files. The *labmap* and *absmap* modules perform parsing and processing of label map and abstraction map files, respectively. Finally, the graphs module contains elementary graph data structures, which can be used to perform analyses such as checking for unsafe recursion, and to generate drawings.

The next to last layer contains the core algorithms, which are contained within the *ltsgen*, *cwigen*, and *cwilink* modules. *ltsgen* performs model generation, and includes the abstract interpreter, and several hashed data structures to support model generation. *cwigen* is the cwi code generator. It performs a single pass, syntax-driven translation of a ckit abstract syntax tree into cwi code. *cwilink* performs linking of two or more cwi

programs into a single program.

At the top layer are the user-interface modules, each of which implements commands a shell-level command. They are *cwcc, cwld, cwmd, cwdump, cwcalls,* and *cwflow.* Each of these modules parses the command line and invokes the appropriate functions from the layer(s) below.

4.2 Compilation

The C Wolf compiler, *cwcc*, generates CWI bytecode from a C source file. Preprocessing is handled by invoking the GNU preprocessor as a subprocess, and saving the results in a file. The *ckit* front-end is then invoked on the preprocessed file. *ckit* performs parsing, type checking, and name resolution. The type checking has been disabled because it is too strict (even though it is correct), and the type checker ends up rejecting many programs that compilers like *gcc* accept. It is the user's responsibility to make sure that a program compiles with *gcc* before attempting to compile it with *cwcc* — if a program with serious type errors is submitted to *cwcc*, the results are not guaranteed to be satisfactory.

The translation process into CWI code is done in a single pass over the abstract syntax tree generated by ckit. The CWI code, whose grammar appears in figure 2.3, is a high-level intermediate code which is very close to C source code. The basic job of the code generator can be viewed as rearranging the input code into basic blocks, and reorganizing expressions so that all side-effects occur in explicit top-level assignment statements.

The CWI file format is binary, consisting of seven parts:

- The *source map* contains descriptions of all source file locations.
- The *type table* contains descriptions of all types used in the program.
- The *variable table* contains descriptions of all program variables.
- The *function table* contains descriptions of all program functions.
- The *basic block network* contains a graph of basic blocks. Each basic block is a sequence of assignment statements followed by a single branching statement.

Each item (type, variable, function, etc.) has a unique integer identifier. This identifier is used to refer to the item. The CWI file is arranged so that there are no forward

dependencies between sections; i.e., variables refer to types, but not vice versa, functions refer to variables and types, but not vice versa, and so on. This makes it possible to read a CWI file from disk in a single pass.

4.3 Linking

The linker, *cwld*, joins two or more CWI files into a single CWI file, using the C rules for resolving global cross-file references. The linker works by merging two files at a time, in order of declaration. First the source maps are merged, and then the types, and then variables, etc. Locally visible items can be directly copied without any worry. Globally visible items are handled by two *name tables*, one for variables and one for functions. Every global declaration and definition is entered into the appropriate name table. When a definition is entered under the same name as a previously entered declaration, the declaration is discarded and replaced with the definition. On the other hand, if a prior *definition* is found, an error occurs.

4.4 Visualization

The visualization tools, *cwcalls* and *cwflow* both produce pictures of graphs. *cwflow* generates a control-flow graph from a CWI file. This is easily done, since the basic block network is essentially a control-flow graph, where each basic block is a node and the branch statements specify the edges. The hard work of laying out and rendering a drawing of the graph is done by either *dot* or *daVinci*, at the user's control. Both *dot* and *daVinci* have a textual input language; to support them, we have defined special *DotGraph* and *DaVinciGraph* abstract data types. Both of these types are annotated graphs, where edges and nodes can have properties like color, labels, shape, style, font, etc. A third, more abstract *LabeledDigraph* type hides the details of these two types. A *LabeledDigraph* is a more generic graph from which a *DotGraph*, *DaVinciGraph*, or textual description can be generated. The *LabeledDigraph* is the primary data structure used by both *cwcalls* and *cwflow*.

The job of *cwcalls* is slightly more complicated. *cwcalls* produces a call graph, by generating a node for each function in the basic block network, and then checking the blocks

local to each function for call statements (each of which results in an edge). The result is displayed graphically, in same manner as *cwflow*.

4.5 Model Generation

Of particular interest is the model generation process. As we explained in chapter 3, the model generation algorithm is basically a loop which computes the fixpoint of the *successor* relation over states. A key issue in implementing such an algorithm is the identification of previously generated states. An even more critical issue is memory usage, since each state contains an entire environment. Given that most states will have environments which are very similar to their predecessor's environment, with possibly just one or two differences, we want to avoid duplicating parts of the environment which are common to multiple states.

Both these issues are addressed through *hashing*. Hashing is a data storage technique based on a mapping from data to (arbitrary and hopefully well-distributed) integer values, called a *hashing function*. Given a collection of n items with a good hashing function, these can easily be stored in an array of n lists by applying the hashing function to each element, converting the hash value to an index i by the modulus with respect to n, and then storing the item in the list kept in the i cell of the array. Given a well-distributed hash function, the median list size will be 1, so this technique yields approximately linear results for identifying duplicate elements, making it easy to avoid duplication of items in the hash table.

C Wolf employs a two-level hashing scheme. First, environments are implemented by hashed *environment nodes*. An environment node is a binding from a variable to an abstract value, plus a pointer to another environment node (the rest of the environment). All environment nodes are stored in a hash table which is shared between all environments in the system. When an environment node is created, the system always checks the table to see if an identical node already exists, thus avoiding duplicates.

Furthermore, we impose two restrictions on environments. The first restriction is a *canonical variable ordering*. Each variable already has a unique integer identifier, and every environment is bound in decreasing order over the identifier, which roughly corresponds to the reverse of the order of declaration in the source code. The second restriction is that no

State	Local Env		Global Env	
	x	y	z	w
1	$0 \ldots 10$	$5 \ldots 10$	$1 \ldots 2$	7
2	$0 \ldots 25$	$5 \ldots 10$	$1 \ldots 2$	7
3	$\top$	$5 \ldots 10$	$3 \ldots 4$	7

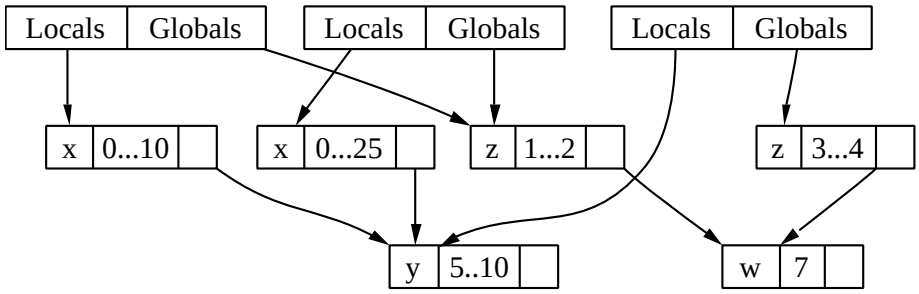

Figure 4.2: An example of common tail sharing between three states.

variable is explicitly bound to $\top$. Instead, all variables are implicitly bound to $\top$ if they are absent from the environment. These two restrictions ensure that there is only one possible representation for the same environment, so that duplicate environments cannot be created (for example, by performing the same bindings in different orders).

Additionally, environments are *purely functional* entities. When a value is bound within an environment e, e is not modified. Instead a new environment e' is computed. Hence, we can be assured than once an environment is created, it will never be modified. This makes it safe to share *common tails* between environments. By *tail*, we mean the nodes which appear at the end of the environment. The hashing scheme guarantees that any two environments which have identical tails will share the storage for those tails.

The next level of hashing is for program states. A program state is a four-word record, which consists of a program counter, a reference to the caller's state, plus two environments: a *local environment*, which contains the values of local variables in the current function, and a *global environment*, which contains the values of variables stored in static program memory. The global and local environments are separated in the hope of saving memory; at the cost of one extra word per state, states with different global environments can share common local environment tails and vice-versa. Figure 4.2 depicts the storage

Type	**Size**	**Representation**
Unity	0	None.
Tri-Log	2 bits	*True, False* or *Maybe*.
Int	1 word	An integer value.
Range	2 words	A low and high bound.
Mod_k	$2 \cdot k$ bits	A Bit-Vector indicating which remainders are possible.
$\text{Part}_{a_1,a_2,\ldots a_k}$	$k + 2$ bits	A Bit-Vector indicating which subranges are possible.
VarPtr	1 word	Id of the variable pointed-to.
FunPtr	1 word	Id of the function pointed-to.
Bool Array	$2 \cdot n$ bits	Two bit vectors.
τ Array	n words	A vector of abstract τ values.

Table 4.2: Summary of the representation of abstract values.

arrangement of three states, with common local and global environment tails.

The program values themselves are not hashed, due to their relatively small size (except for arrays). Values are dynamically typed, and carry a type tag at runtime. The full set of possible values and their representations is listed in table 4.2. All the integer C operations are implemented on the abstract types (although it is the case that many of the complex operations, such as bit-shifting, simply return $\top$). The implementation of these operations follows the algorithms presented in chapter 3.

Dynamic typing is used during expression evaluation. An integer constant evaluates to abstract type *Int* (i.e., a precise representation), and a fetched variable value has the abstract type of the variable, and operations that are abstracted away, such as dereferencing a pointer to a heap value, have abstract type *Unity*. Operators are applied using a table-driven process. A table is searched using the operator and the abstract types of the arguments for an implementation. If no implementation is found, and the types of the arguments is mixed, then the less abstract value is cast to the more abstract type and the search is repeated. If no implementation is found on the second attempt, then the result is $\top$. This approach allows selected mixed-type operations, such as adding a precise integer value to a *Range* value, to be implemented in order to improve the precision of the evaluation results.

The effect of an assignment statement is to update a single variable (recall that all CWI expressions are pure, so that no side-effects can occur during expression evaluation). The value is first cast to the abstract type of the destination variable, and then the

appropriate environment (local or global) is updated with the new binding. If the binding is $v \mapsto \top$, then the variable v is eliminated from the environment instead.

In a few of cases, the assignment statement may force a new state to be generated. First, if a variable is the target of a *read* or *write* clause in the label map file, then a transition is generated to a new state. Second, if a *watch* is triggered (i.e., the assignment changes the value of the variable in a guard condition, causing the condition to become possibly true), then two transitions are generated to two different states: a state in which the guard condition is false, and a state in which the guard condition is true. These two states have different environments, since we use the truth or falsehood of the condition to constrain the value of target variable of the *watch*. Note that if the condition is definitely true, then only one transition is generated, and if the condition is definitely false then no transition is generated. The third case is if the source line of the assignment statement is the target of an *exec* clause. This also forces a transition to a new state. Fortunately, for typical label mappings, very few assignment statement evaluations will result in transitions to new states.

The effect of a branching assignment is to generate one or more transitions to other states. The effect of a *goto* statement is to change the current program counter to the start of a new basic block, without affecting the environment. A new state is generated combining the new program counter with the current environment.

The effect of a *tail call* statement (i.e., a call whose return value is immediately returned by the caller) is the same as a goto. For *non-tail calls*, a new *caller* state is generated to capture the environment immediately before the function call, and then a new *callee* state is generated as a child of the caller state. The parent-child relationship between states corresponds directly to the stack of activation records in the concrete program.

A *return* is processed by taking the parent of the current state, then using the static successor of the the parent state as the new program counter, the local environment of the parent state with register 0 bound to the return value, and the global environment of the current state. A special case is when there is no parent state. This means that the program is terminating, and generates a transition to a halt state.

A *conditional goto* statement generates one or two successors. If the condition evaluates to a definite true/false answer, then there will be only one successor state, combining the current environment with a new program counter (the target of the goto). If the condition evaluates to *maybe*, then two successors are generated: one successor in which the

condition is assumed to be true, and one in which the condition is assumed to be false. The truth or falsehood of the condition is used to constrain the environment. The constraint analyzer looks at the condition for relational operations and uses these to reduce the abstract values of the variables involved. For example, consider a conditional goto based on the relation $x \leq y$, where $x \mapsto \{1 \ldots 10\}$ and $y \mapsto \{5 \ldots 20\}$. The condition may be true or false. In the case where the true branch is taken, then we can't constrain the value of either x or y. In the case where the false branch is taken, we know that $x > y$ and we may safely constrain x to $\{6 \ldots 10\}$ and y to $\{5 \ldots 9\}$. Hence, conditional branches result in different environments depending on which branch is taken.

The evaluation of *switch* statements is very similar to the conditional branch since a *switch* is just shorthand for a large sequence of conditional branches. Each case of the switch results in a different environment where the value of the switch expression is used to constrain the environment using an equality relation.

The successors of a state q are computed by taking the environment of q, fetching the statement at the address contained in the q's program counter, and evaluating the statement(s) at that address (keeping record of any changes to the environment) until a transition is generated. At that point have computed the effect of q, and can move on to another state. A record of q needs to be maintained in order to avoid re-computing the same states ad infinitum.

The entire state space is generated using a simple algorithm. A queue of unprocessed states is maintained. While the queue is not empty, the state at the head of the queue is selected and its successors are computed. The successors are checked to see if they are new states (via a hash-table lookup). Any new states are added to the queue. The process repeats until the queue is emptied, at which point the entire state space has been generated.

4.6 Optimization

Once the states have been generated, an optimization pass occurs. The optimizations currently supported are *peephole*, *strong bisimulation*, *observational equivalence*, and *trace equivalence*.

The peephole optimizer examines states one at a time to identify states which can

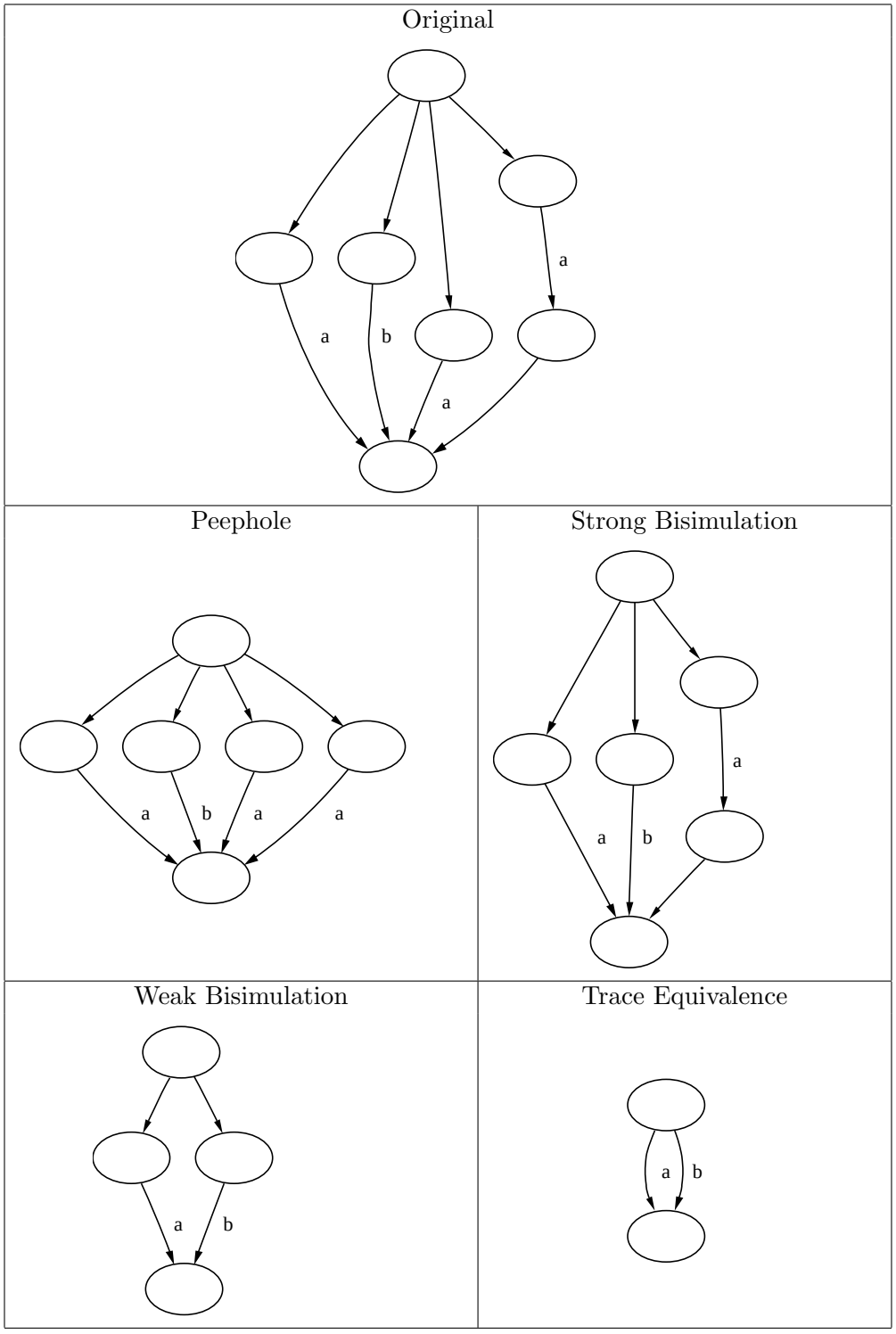

Figure 4.3: A graph optimized four different ways.

be deleted without affecting strong bisimulation. The criteria for selecting such states is simple: states which have only one outgoing τ (silent) transition, and don't self-loop, are chosen. Such states can be deleted safely by simply routing incoming transitions to the (single) successor state. The result is a system whose behavior is strongly bisimilar to the original system.

Additionally, an option is provided to run the peephole optimizer on the fly, in lock-step with the LTS generation. In this case, an additional restriction is added to the criteria for state deletion. The extra requirement is that the basic block number of the destination state be larger than the state being deleted. This prevents the system from getting caught in a loop of states connected by single τ transitions.

The second available optimization is *strong bisimulation*. Although this optimization is not strictly required, since `cwb-nc` is also capable of computing it, it is useful to reduce the size of the automaton file that will be passed to `cwb-nc`. The strong bisimulation relation, $\sim$ is relation over the set of states and is intuitively defined as "p can simulate anything q can do and vice-versa." Formally, this is defined as

$$p \sim q \Leftrightarrow p \simeq q \text{ and } q \simeq p$$

Where $\simeq$ is the simulation relation. The intuitive definition of $\simeq$ is "p simulates q if for every move q can make, p can make a similar move." The formal definition is:

$$p \simeq q \Leftrightarrow \text{ for each } a \text{ such that } q \xrightarrow{a} q' \text{ there exists } p' \text{ such that } p' \simeq q' \text{ and } p \xrightarrow{a} p'$$

There are several algorithms for computing the strong bisimulation relation [58]. Our optimizer currently uses the simple algorithm presented in [56], which starts with the complete relation $\sim_0 = Q \times Q$, and iteratively refines $\sim_i$ based on $\sim_{i-1}$, until a fixpoint is reached (when $\sim_i = \sim_{i-1}$). The refinement criteria is:

$$p \sim_i q \Leftrightarrow \text{ for each } a \in Act, \left(\begin{array}{l} p \xrightarrow{a} p' \Rightarrow \exists q'.q' \sim_{i-1} p' \text{ and } q \xrightarrow{a} q' \text{ and} \\ q \xrightarrow{a} q' \Rightarrow \exists p'.p' \sim_{i-1} q' \text{ and } p \xrightarrow{a} p' \end{array} \right)$$

$\sim$ is an equivalence relation. Once $\sim$ has been computed, for each group of states related by $\sim$, we merge all of them into a single state.

The third optimization preserves observational equivalence (also known as *weak bisimulation*). It is computed in a manner similar to strong bisimulation, but is done using weak ($\stackrel{a}{\Rightarrow}$) transitions instead of the strong ($\xrightarrow{a}$) transitions. Weak bisimulation produces

the smallest possible system whose behavior is still "the same" as the original system (in the sense of what an external observer manipulating the system by providing inputs and watch the resulting actions can tell).

Fourth, an optimizer which only preserves *trace equivalence* is provided. This optimized does the same peephole analysis with a weaker criteria. Any state whose outgoing or incoming transitions are all τ transitions is deleted, until only states which perform visible actions are left. This usually results in a very small system, although it destroys the internal decisions made within it.

The last step in model generation is to output the results. The output is always a source file; the format can be human-readable text, a *dot* graph description, a *davinci* graph description, or a *cwb-nc* automaton description.

4.7 Example

We conclude this chapter with an example of the entire model building process. The program we wish to verify is shown in figure 4.4. This program is a unix-style filter which reads from standard input and returns a successful exit status if the input string matches the regular expression `a`*`b`* (note that in UNIX, an exit status of 0 means success, and non-zero means failure).

The first step is to compile the program. The source file is named `fsm.c`, and is compiled by the command `cwcc fsm.c`. The result is the file `fsm.cwi`. `fsm.cwi` is a binary file containing CWI bytecode. The `cwflow` command can be used to visualize the contents of a CWI file. Figure 4.6 contains a `dot`-rendered control-flow graph which was generated by the command `cwflow --format=dot fsm.cwi`.

The CWI file is the text from which a model of the program is built. But before we can generate a model, we need to specify what abstractions should be applied to the program variables. This is done by the file `fsm.am`, the abstraction map for `fsm.c`, which appears in figure 4.5. `fsm.am` applies two abstractions. The first abstraction is `state : part(0,1,2)`, which specifies that the variable `state` (in function `main`) should be mapped onto the abstract domain $\mathcal{P}(\{< 0, = 0, = 1, > 1\})$. This will give us precise information as to whether or not `state` is equal to 0 or 1, the two values of particular interest.

The second abstraction is `var ch : part(-1, 0, 9, 10, 11, 97, 98, 99)`.

```
#include<stdio.h>

int main()
{
  int state = 0;
  char ch;

  while (1) {
    ch = getc(stdin);

    if (ch == '\n' || ch < 0)
      return 0;

    switch (state) {
    case 0:
      if (ch == 'b') {
        ++state;
      } else if (ch != 'a') {
        return 1;
      }
      break;

    case 1:
      if (ch != 'b') {
        return 1;
      }
      break;
    }
  }
}
```

Figure 4.4: A simple C program to recognize a^*b^*.

fsm.am

```
file "fsm.c" {
  fun main () : top {
    var state : part(0,1,2);
    var ch : part(-1, 0, 9, 10,
                11, 97, 98, 99);
  }
}
```

fsm.lm

```
exit == 0 => accept;
watch (main:ch == 97) => `a;
watch (main:ch == 98) => `b;
watch (main:ch < 0) => `eof;
watch (main:ch == 10) => `eof;
watch (main:ch >= 0) => `X;
```

Figure 4.5: Abstraction and label maps for the program of figure 4.4.

This abstraction allows us to track whether `ch` is negative (used by `getc` to indicate an end of file condition), equal to 10 (the newline character `'\n'`), equal to the character `'a'` (whose ASCII code is 97), or equal to the character `'b'` (ASCII code 98).

In addition to abstraction information, we need directions specifying how transitions should be labeled. This is done by the label map file, `fsm.lm`, which also appears in figure 4.5. `fsm.lm` generates five labels. The first four of these are generated when a value is assigned to variable `ch`. Note that the only place this can occur is after the call to `getc`, at the beginning of the while loop. Also note that the backquote (`) character is used to distinguish inputs from outputs, so that `accept` is an output label, and everything else is an input label (we interchangably use $\overline{a}$ and `` `a `` to stand for the "input a" label). The actual set of labels are:

- The label $\overline{a}$ is generated whenever an `'a'` character is stored in `ch`.
- The label $\overline{b}$ is generated whenever an `'b'` character is stored in `ch`.
- The label $\overline{eof}$ is generated whenever an `'\n'` character or a negative value is stored in `ch`.
- The label $\overline{X}$ is generated when any value not listed above is stored in `ch`.

The last labels are generated when the program exits. The label *accept* is generated if and only if the exit status is zero. This is based on the UNIX standard that an exit status of zero indicates a correct termination, while a non-zero exit status indicates an error.

Figure 4.7 shows the output of the command `cwmb --format=dot -a fsm.am -l fsm.lm fsm.cwi`. This command constructs a graphical model of the program's state space. Each state in figure 4.7 is labeled with the stack of program counters (with the current program counter value on top), and the current environment. There are a total of about 50 states. Casually examining the figure, we note that the machine does a few preliminary calculations, then "splits" into several branches, with all branches either eventually terminating or looping back to the top "split."

Figure 4.8 shows the first nine states in detail. The first state, state 0, is positioned to the beginning of block 0 and has an empty environment (all values are bound to $\top$). There are two statements executed, the assignment $state = 0$, followed by a jump to block 2.

The second state (state 1) results from the execution of block 0 on the initial environment. In state 2, the environment is empty except for *state*, which is bound to 0.

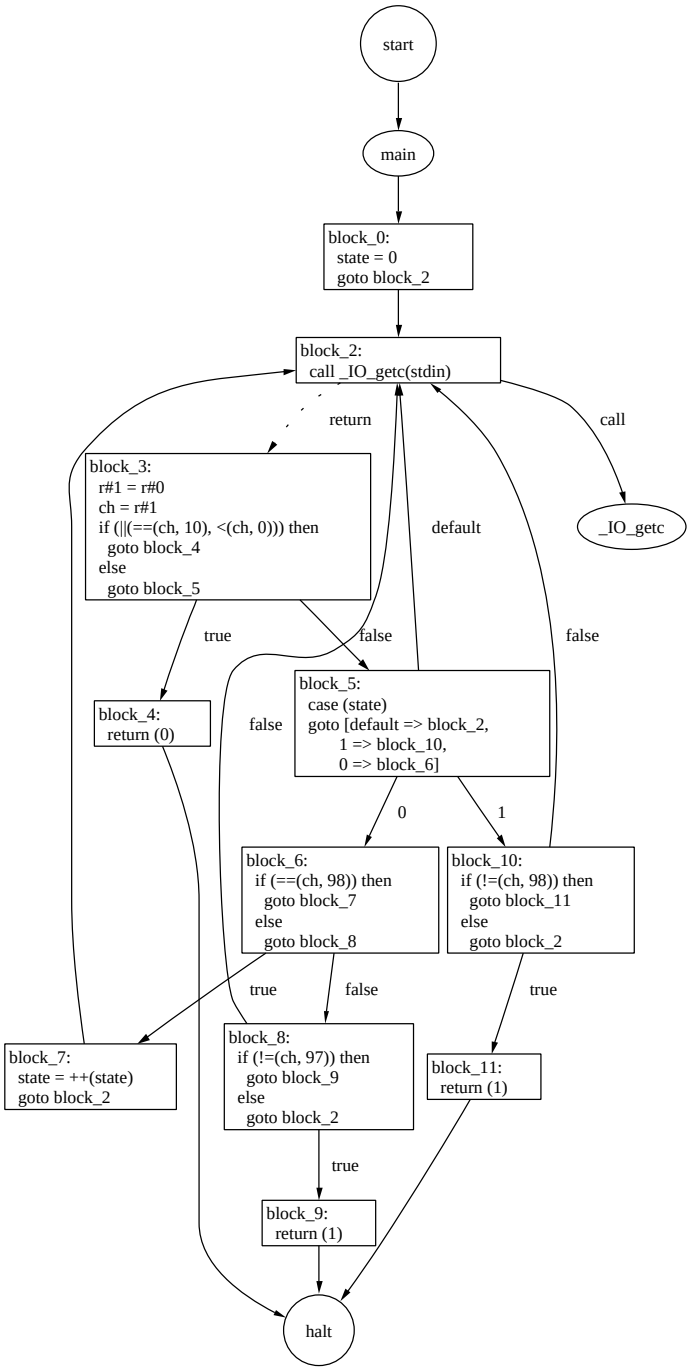

Figure 4.6: Control-flow graph for the program of figure 4.4.

The program counter is positioned to the start of block 2. The only code in block 2 is a call to the function `_IO_getc` (the "real" name of the C `getc` function).

State 2 is immediately after the call to `_IO_getc`. Since `_IO_getc` is a system function, the interpreter abstracts it by a dummy function which has no side-effects and always returns $\top$.

State 3 is after `_IO_getc` returns. The only variable not bound to top is *state*, which is still bound to 0. The code about to be executed contains 2 instructions. The first instruction is a redundant assignment of register zero to itself, which is semantically equivalent to a NOP instruction. The second instruction is the assignment of register 0 to the variable *ch*.

States 4 through 8 result from the assignment of register 0 (whose value is $\top$) to *ch*. Since *ch* is the target of *watch* clauses, the interpreter generates a different successor state for each clause that triggers. The value of *ch* is constrained differently in each state, based on the clause which was satisfied. For example, in state 4, which follows an a transition, the value of *ch* is constrained to 97, which is the ASCII code for the character a.

Figure 4.9 shows the output of the command `cwmb --format=dot -a fsm.am -l fsm.lm -d 0 fsm.cwi`. This command constructs a graphical model of the LTS corresponding to the program's behavior. Note that figure 4.7 is labeled with the stack of program counters (with the current program counter value on top), and the current environment.

Figure 4.10 shows the output of `cwmb` with the `--peep` option added. The peephole optimizer, which deletes states which have a single outgoing τ transition, was quite successful for this program, resulting in a state space reduction of 75%.

Finally, figure 4.11 shows the result of adding the `--weak-bisim` option. This option produces the smallest possible machine which is still observationally equivalent to the unoptimized machine, although at a much higher computational cost.

Visual examination of the models in figure 4.11 might convince us that the language recognized by the program is indeed a^*b^*. However, there is a more sound way to confirm this fact (which becomes particularly more important when the model has thousands or millions of states). We can use the Concurrency Workbench. To do this, we first build a model in format which `cwb-nc` can read. This is done with the command `cwmb --format=cwb -l fsm.lm -a fsm.am --peep fsm.cwi -o fsm.ccs`. The result is a text file, `fsm.ccs`, containing a concise description of an LTS (called an *automaton* in cwb-nc lingo). The contents of `fsm.ccs` appear in the left half of figure 4.12.

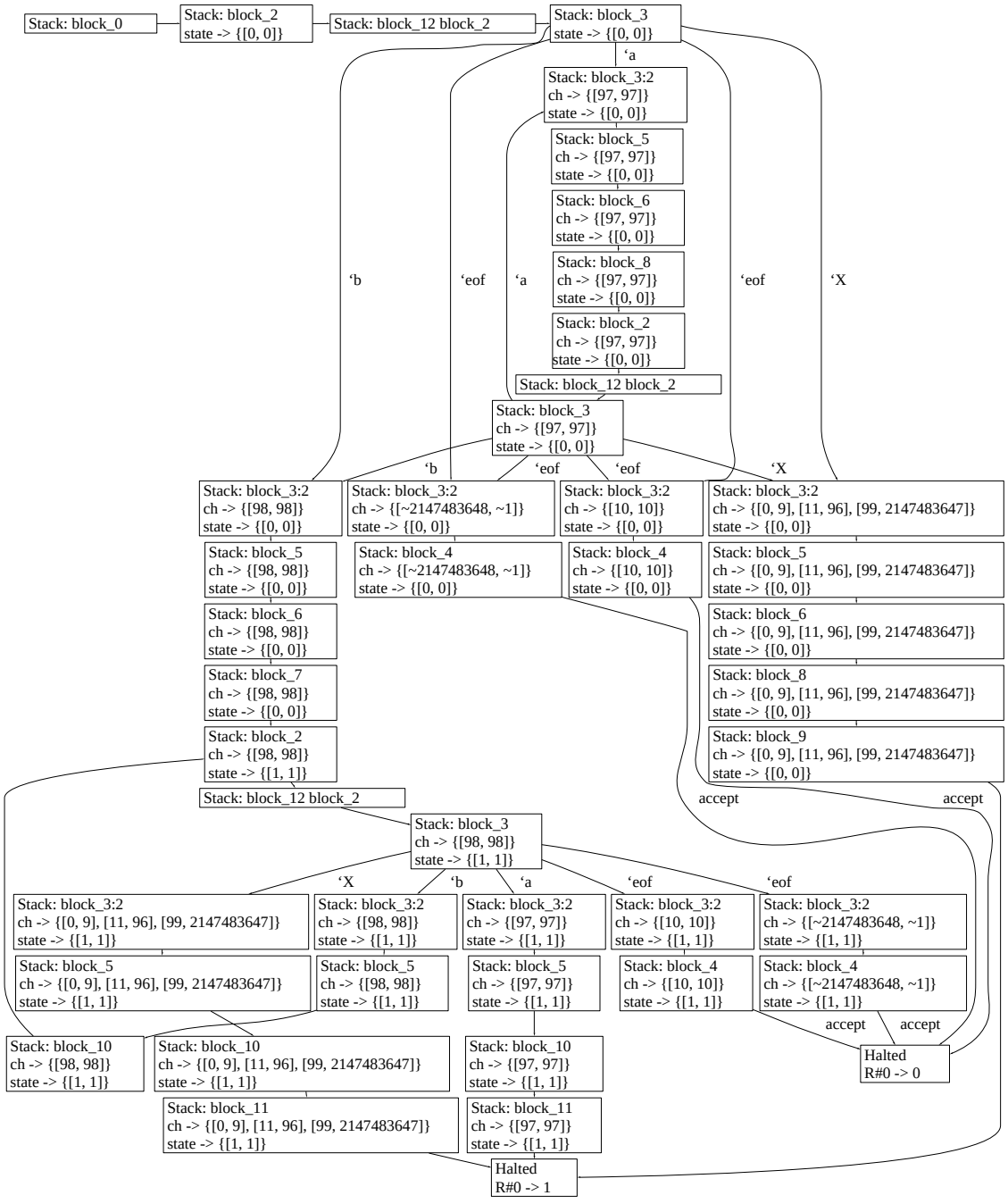

Figure 4.7: Kripke structure representation of the state space of the program of figure 4.4.

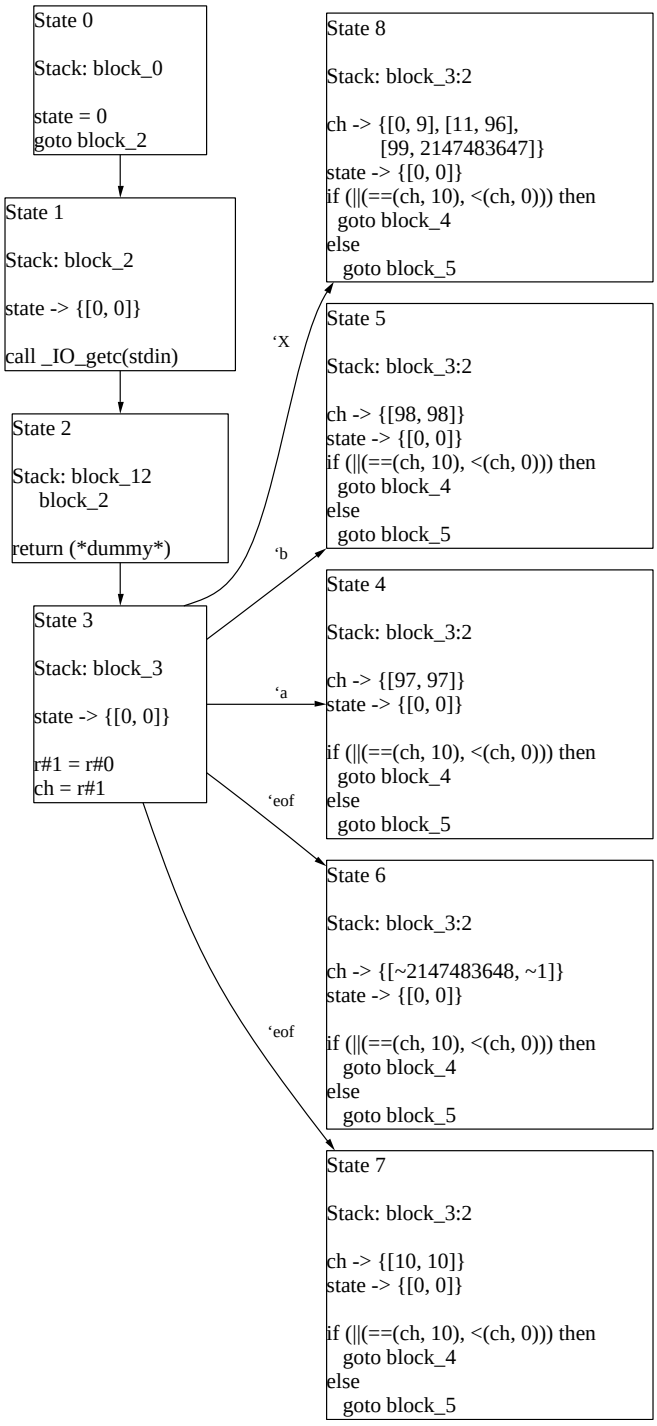

Figure 4.8: Detail of first nine states of figure 4.7.

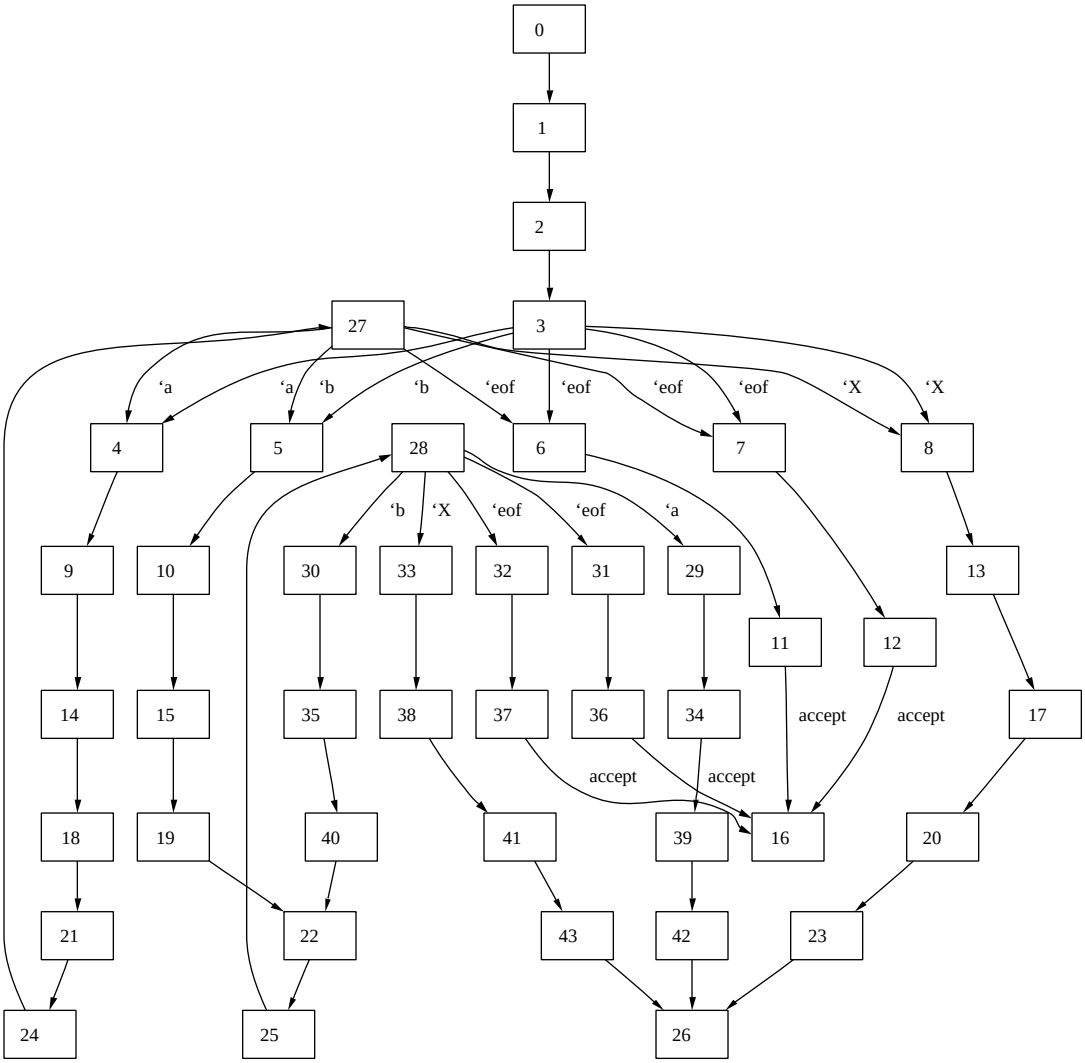

Figure 4.9: LTS for the program of figure 4.4.

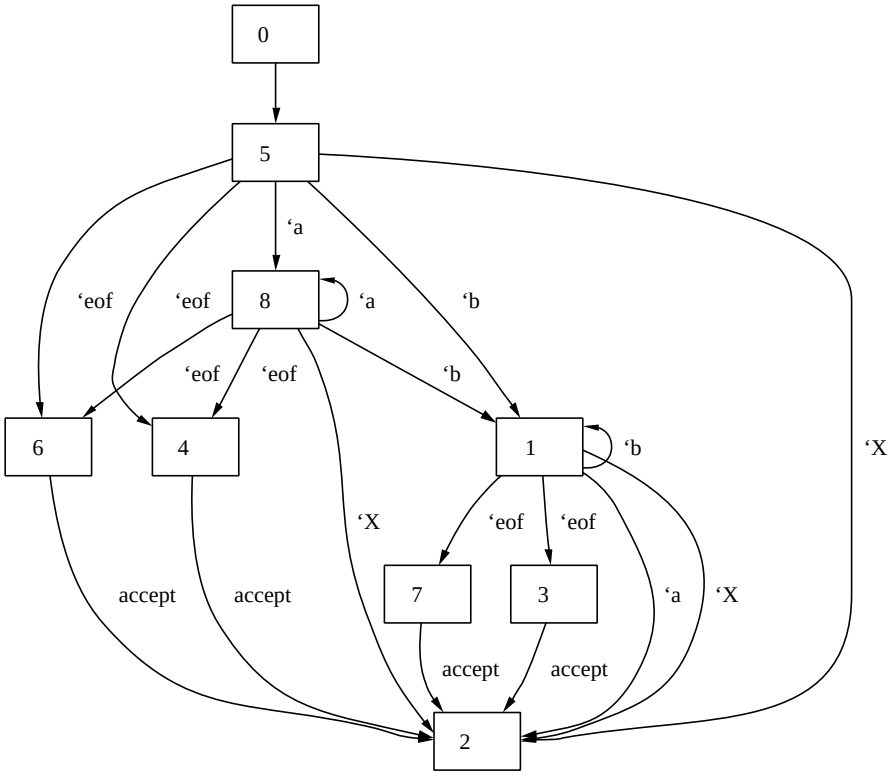

Figure 4.10: LTS of figure 4.9 after peephole optimization.

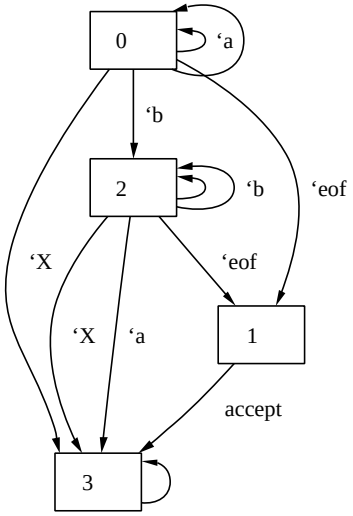

Figure 4.11: LTS of figure 4.9 after weak bisimulation optimization.

fsm.ccs

```
proc main= Aut(
start=0,
1:‘b{1}X{2}‘a{2}‘eof{3,7}
0:t{5}
3:accept{2}
4:accept{2}
5:‘b{1}‘a{8}‘X{2}‘eof{4,6}
6:accept{2}
7:accept{2}
8:‘b{1}‘a{8}‘X{2}‘eof{4,6}
)
```

fsm-spec.ccs

```
proc spec = ‘a.spec + ‘b.B
          + ‘eof.accept.nil
          + ‘X.reject

proc B = ‘b.B
       + ‘eof.accept.nil
       + ‘X.reject + ‘a.reject

proc reject = t.reject + nil
```

Figure 4.12: A `cwb-nc` automaton generated from the program of figure 4.4 and its corresponding specification.

The next to last step is to write a specification for the language a^*b^*. We do this by writing a program in the process algebra *CCS*. The file `fsm-spec.ccs` contains such a program, and appears in the second half of figure 4.12. A detailed description of CCS is beyond the scope of this thesis, but suffice it to say that we can view the process `spec` as a machine which can do four things:

- accept an a, and repeat from the beginning,
- accept a b, and then emulate the machine `B`,
- accept an eof, then emit an *accept* and terminate, or
- accept an X, and then emulate the machine `reject`

The `B` machine can also perform four actions:

- accept an a, and then emulate the machine `reject`,
- accept a b, and repeat from the beginning,
- accept an eof, then emit an *accept* and terminate, or
- accept an X, and then emulate the machine `reject`

Finally, the `reject` machine performs as many τ actions as it likes and then halts.

This machine corresponds to what the behavior of our program should be. It should accept any number of a's followed by any number of b's, followed by an eof, and anything else should be rejected. We blurred the line between inputs and outputs, but this doesn't affect the correctness condition in this case, since everything has a unique name. If we can show that extracted model from `fsm.c` is observationally equivalent to the CCS process `spec`, then we will have strong reason to believe our program is correct.

Checking for equivalence is done with the Concurrency Workbench of the New Century. The Concurrency Workbench is started with the command `cwb-nc ccs`. We then load the two files of figure 4.12 with the commands `load fsm.ccs` and `load fsm-spec.ccs`. The automaton in `fsm.ccs` will be bound to the name `main`, and the process defined by `fsm-spec.ccs` will be bound to the the name `spec`. The command `eq -S obseq main spec` is used to compare the two systems for observational equivalence. Figure 4.13 contains

```
% cwcc fsm.c
No errors, no warnings (suppressed).
Compilation succeeded. Output is in fsm.cwi
% cwmb -l fsm.lm -a fsm.am --format=cwb fsm.cwi  -o fsm.ccs
Loading fsm.cwi...
Loading abstraction map...
Loading fsm.am...done
Building abstraction table...
Loading Label Map...
Checking recursion...
Generating State Space...
52 states generated.
62 transitions generated.
29 environment nodes used.
% cwb-nc ccs

The Concurrency Workbench of the New Century
(Version 1.2 --- June, 2000)

cwb-nc> load fsm.ccs
Execution time (user,system,gc,real):(0.000,0.000,0.000,0.003)
cwb-nc> load fsm-spec.ccs
Execution time (user,system,gc,real):(0.000,0.000,0.000,0.011)
cwb-nc> eq -S obseq main spec
Building automaton...
States: 57
Transitions: 72
Done building automaton.
Transforming automaton...
Done transforming automaton.
TRUE
Execution time (user,system,gc,real):(0.010,0.000,0.010,0.018)
cwb-nc>
```

Figure 4.13: Transcript of a `cwb-nc` session in which the program of figure 4.4 is verified.

a transcript of a session in which this is done. Note that the commands entered by the user are italicized, and the unitalicized text is the output from the computer. The final result is that the process algebra system described by `fsm-spec.ccs` is observationally equivalent to the LTS extracted from `fsm.c` by C Wolf.

Chapter 5

Related Work

There are a number of projects which are addressing the problem of software model checking. A quick preview:

- *AX* is a tool for translating C programs to *Promela*, the input language of the *Spin* model checker.
- *Bandera* is a system for model-checking Java programs using data abstraction.
- *Java PathFinder* is a system for model-checking Java Virtual Machine (JVM) byte-code, aimed at finding concurrency-related errors.
- *JCAT* and *YAV* are model-checking tools for Java, similar to Java PathFinder.
- *SAL* is another Java model-checker, also aimed at finding deadlocks and other concurrency-related bugs.
- *SLAM* is a system for model-checking C programs using predicate abstraction.
- *Scott Stoller* has implemented a tool for model checking multi-threaded distributed Java programs
- *xgcc* is an extension of the *gcc* compiler, augmented with a language for generating program execution models.

We will examine each of these projects in detail and contrast them to the C Wolf approach.

5.1 Bandera

The Santos research group at Kansas State University has developed *Bandera* [26, 18, 39]. Bandera is a system which uses model-checking to verify Java programs. Like C Wolf, Bandera uses user-directed data abstraction to generate reduced models of Java programs. Models can be generated for several model checkers, including Spin, SMV and Java PathFinder. The Bandera project is currently in its second generation — an earlier version worked with Ada programs.

In Bandera, abstractions are developed in a programming language, the Bandera Abstraction Specification Language (BASL). A BASL abstraction includes a declaration of a set of tokens (the abstract domain is the power-set of the tokens), an abstraction function to map concrete values onto sets of tokens, and abstract implementations of all the basic operations. While this is more flexible than our approach, in the sense that a user can write a BASL file for almost any imaginable abstraction, it is also more cumbersome because BASL doesn't support parameterized abstractions. For example, if the user wishes to apply a modulus-8 abstraction, a specific BASL file must be written for this case, whereas our abstraction model provides for all possible modulus values. Furthermore, if the user wishes to perform a abstraction similar to C Wolf's `part` abstraction, the user must write a new BASL file for each set of breakpoints supplied to `part`. Hence, our approach is more powerful and flexible in many instances, while the BASL approach allows the user to create efficient tailored abstractions for specific applications.

An examination of the Bandera's abstraction library reveals that the `unity`, `mod`, and `part` abstractions cover all the system-provided integer abstractions. The Bandera abstraction library is organized into *families* of abstractions. The *point* family is equivalent to C Wolf's `top` abstraction. The *set* and *range* families both contain what are essentially instances of C Wolf's `part` abstraction. The *modulo* family contains instances of C Wolf's `mod` abstraction. Hence it would appear that our fixed set of parameterized abstractions is powerful enough to cover most practical cases.

Having crowed over the advantages of C Wolf's parameterized abstractions, it should be acknowledged that Bandera is much more ambitious in many other areas than C Wolf. Bandera has a much richer specification language, which combines temporal logic with fragments of the Java programming language. Bandera has a graphical user interface, which seamlessly integrates the model checker and model generator into a single verification

system. The GUI also allows one to trace program execution, perform single-step operations, and even simulates backwards execution, with the ability to examine program variables at each point. Bandera also performs program slicing based on the property being verified, eliminating irrelevant parts of the program at an early stage in the model generation process. Finally, it should be noted that Bandera generates models of concurrent programs, while `cwolf` is limited to sequential ones.

5.2 Java PathFinder

The Automated Software Engineering Group at NASA Ames Research Center has developed Java PathFinder (JPF), a model checker for Java programs [66, 40, 8]. At the core of JPF is a *model-checking Java virtual machine* (MC-JVM). The MC-JVM is designed for memory efficiency, and to support backwards execution. A DFS search engine works in concert with the MC-JVM, collecting all visited states in a hash table. The only properties that JPF can check are deadlocks and invariant conditions. Recently, support for predicate abstraction has been added to JPF. The abstraction mechanism replaces concrete program variables with boolean variables whose values reflect the status of predicates extracted from `if` and `while` statements. Furthermore, the abstraction mechanism deals effectively with some of the problems caused by object-orientation [67].

5.3 YAV

YAV and its predecessor JCAT are both Java model checking tools developed at Politecnico di Torino by Iosif, Demartini, and Sisto [45, 46, 22]. YAV accepts a Java program as input and generates as output a program in an extension of the Promela language. This extended Promela code is processed by dSpin, an enhancement of Spin with support for the modeling of dynamic memory operations, pointers-to-functions, and a function call-stack. YAV doesn't appear to support any data or predicate abstraction. Instead it relies on the state-space caching algorithm of Spin to perform incomplete model checking of the resulting Promela program, which is still useful for detecting many bugs. Overall, YAV and JCAT are very similar to JPF.

5.4 Verisoft

Godefroid at Bell labs has spearheaded the development of Verisoft [34, 36, 35]. Verisoft is a state-space exploration tool which works with C programs, as well as other programs. Verisoft never actually builds a model; instead it monitors and single steps the execution of the processes of a concurrent software system, checking the current system state for properties such as deadlock, livelock, and assertion violations. Verisoft performs a *stateless search*; it doesn't remember what states it has already visited, and in fact it can become trapped in a loop re-examining previously visited states in some circumstances. The odds of this happening are reduced by the use of *persistent sets* and *sleep sets*. The persistent set is statically determinable set of transitions which are independent of all other transitions, allowing a partial-order reduction to be exploited. The *sleep set* is a small cache of transitions from the persistent set which have already been explored. The persistent set is fairly easy to compute because Verisoft assumes a communicating process model without threads (i.e., processes can send each other messages, but there are no shared variables). The Verisoft scheduler uses these sets to control scheduling of a processes, so that execution proceeds until a transition is reached which is not persistent. Finally, it should be noted that Verisoft has no abstraction mechanism, which is fine because no set of states is collected. Overall, the Verisoft approach is radically different from ours.

5.5 Stoller's Work

Scott Stoller has developed a tool for model checking multi-threaded distributed Java programs [64]. Like Verisoft, the tool performs a stateless search, to find concurrency-related bugs, but shared variables are supported. Hence Stoller distinguishes his work by calling it both multi-threaded and distributed, vs. Verisoft, which is only "distributed."

5.6 AX

Holzmann and Smith at Bell Labs have developed several tools for translating C programs to Promela (the input language of the Spin model checker) [42, 43, 44, 60]. The tool, AX, (formerly known as FeaVer), performs a table-driven translation of C programs to

Promela. The translation process consists of three phases. In the first phase, the program is parsed, resulting in an annotated abstract syntax tree. In the second phase, each basic action and condition in the syntax tree is matched with an *interpretation* from the translation table. The interpretation can be either a fragment of Promela code, or a command to ignore (abstract away) the matching C code, among other things. In the third phase, the resulting Promela program is optimized, particularly to eliminate dead and redundant code which may result from the abstraction process.

The abstraction table employed by AX is open to modification by the user, and it is up to the user to only to employ sound abstractions. Hence, AX can be viewed as a sort of low-level abstraction engine. This differs from `cwolf`, which provides a fixed set of safe abstractions for the user to choose from.

5.7 SLAM

The SLAM project at Microsoft Research, has developed a toolset to model-check C programs with predicate abstraction [7, 5, 4, 3, 6]. The SLAM toolkit has three basic components. The first, *C2bp*, takes a set of predicates and a C program as input, and generates a *boolean program*, an abstraction of the original program whose variables reflect the truth or falsehood of the predicates in the original program. The second, *Bebop*, is a model checker for the programs generated by C2bp. The final tool, *Newton*, performs a feasibility analysis of the execution paths in the original C program, and then uses the analysis results to refine the abstracted boolean program by adding additional predicates.

A major feature of the SLAM abstraction process is that the abstraction mapping is automatically generated. The user writes a correctness specification in *Slic* (Specification Language for Interface Checking), a C-like specification language which is compiled into C code.

Once the Slic specification is ready, the program to be verified is automatically instrumented with calls to the compiled Slic code. The Slic code contains tests which abort execution when a violation of a safety property occurs, such as function `f` returning while a specific lock is not yet released.

C2bp is applied to the instrumented C program plus a set of predicates gathered from the specification, and generates a boolean program. Bebop examines the boolean

program and attempts to determine if any of the *abort* calls are reachable. If there is a definite yes or no answer, then the model checking terminates. Otherwise, the Newton program is invoked to refine the set of predicates. C2bp is then re-run using the new, more complex set of predicates, and the process starts over again. The compile-check-refine sequence iterates until a definite yes/no answer is reached, memory runs out, or a fixpoint is reached.

The SLAM process has been effectively used to rapidly find bugs in Windows NT device drivers, among other things. We view SLAM as parallel effort in a different direction; in particular the abstraction mechanism is quite different from ours (predicate abstraction instead of data abstraction).

5.8 xgcc

A group of researchers at Stanford University, led by Dawson Engler, has developed *xgcc* and *xg++*, extensions of the GNU C compiler with the capability to extract models from C and `C++` code [13, 54, 29, 30].

xgcc implements a process Engler calls *metacompilation* (MC). The philosophy of MC is to allow the user to add features to the compiler, through the use of a language which operates on the intermediate code of the compiler, using a pattern matching language that emits source code as output. The actual kind of code is up to the user.

In [13] xgcc is used together with the Murϕ model checker to find bugs in the FLASH communications protocol. The Murϕ input language is a Pascal-like imperative language, similar to that of spin and unlike the process algebra language of cwb-nc.

The language used to extend xgcc is known as *Metal*. In [13], Metal is described as having two components. First, the *Metal slicer* performs program slicing. The Metal slicer specification consists of a set of patterns over the set of C variables, expressions, and statements. These patterns are applied to the intermediate code within the compiler. Code which matches a pattern is kept, and code which doesn't match any pattern is discarded.

The second component of Metal is the *Metal Printer*. The Metal Printer generates Murϕ code from C code, again using pattern matching in a process generally known as *term-rewriting*. The patterns are expressions over the abstract syntax of the intermediate code, and the result of a pattern match is a Murϕ code fragment which effectively replaces the

matched pattern. The Murϕ code typically uses small variables to track things such as state of lock variables, and includes support for assertions which are checked in the final model to make sure that no reachable state contains an assertion violation. This version of `xgcc` was used to effectively debug the cache coherence protocols of the FLASH multiprocessor.

In a related paper, [29], `xgcc` was applied to operating systems code. For this case, the Metal language was used to implement `C++` finite state machines. The finite state machines did things like track the state of a particular lock or interrupt enable flag, and triggered an exception if an interrupt was enabled twice or a particular subroutine exited without re-enabling interrupts. These state machines were then dynamically linked into the program code, and the program is statically analyzed to find paths which cause the state machines to signal errors. This method proved extremely effective, finding hundreds of errors in code which already been extensively tested.

5.9 SAL

Park, Stern and Dill at Stanford have developed *SAL* (Symbolic Analysis Laboratory), a model-checker for Java programs. The emphasis of SAL is to support dynamic data structures, such as call stacks and dynamic object allocation. In this sense it is very much like dSpin (the dynamic extension of Spin used by YAV). The model checking process consists of four steps. In the first step, Java is compiled into JVM bytecode. In the second step, the bytecode is compiled into *Jimple*. Jimple is an three-address intermediate representation, developed as part of the *Soot* project at McGill University. The Bandera toolkit is also based on Soot, and in fact, SAL's JVM bytecode to Jimple translator was directly appropriated from Bandera.

The second step in the translation process is the compilation of Jimple into SAL. SAL is a Promela-like language of guarded commands and assignments, but unlike Promela, SAL supports dynamic data structures. The SAL code is very low-level, with the patterns typically matching program counter values, and the resulting command being a multiple assignment to program variables plus the program counter.

The third step is the compilation of SAL in `C++`. The `C++`. code contains a function to simulate each guarded command, plus classes for each type in the SAL program. This code is directly incorporated into the model checker via a `#include` directive.

Finally, the resulting `C++` program is compiled and executed. The model checking code collects program states in a hash table, and uses a partial-order reduction technique known as *atomic blocks*, developed by Bruening for *Rivet* [9], to reduce the model size by skipping past irrelevant interleavings in multi-threaded processes (the Rivet tool itself is an instrumented JVM with features somewhat like Verisoft). The code instrumented with assertions that print an error message plus execution path information if a deadlock or livelock is detected. Overall, the SAL approach is close to that of JPF — there is no abstraction mechanism, and the focus of the system is finding concurrency-related bugs.

5.10 Summary

Perhaps the best way to understand the projects presented in this chapter is to group them based on similarities. We have grouped them into five categories based on their approach.

The *data abstraction* approach, in which the variables of a program are lifted to an abstract domain before a model is generated is used by *Bandera*. We also place C Wolf in this category.

The *predicate abstraction* approach abstracts variable values based on whether or not they satisfy a set of predicates. This is the approach used by *SLAM*.

The *Metacompilation* approach is based on extending the compiler for a programming language with a user-customizable language or table for generating the input language of a model checker. In this category belong *AX* and *xgcc*.

The *specification-less* approach is used by tools which are aimed at finding bugs for which no specification is required, because they are "obvious" concurrency related-errors, such as deadlock or race conditions. The abstraction technique employed by these tools is typically to slice away variables which don't affect interaction between processes. This is the approach of *Java PathFinder*, *JCAT/YAV* and *SAL*.

The *stateless search* paradigm is based on monitoring the execution of a program at runtime, checking execution paths for bugs without building a model. In order in improve coverage and reduce the state space, the verification system may control the scheduling of processes and threads within the target system. This is the approach of *Verisoft* and Stoller's tool.

Chapter 6

Experience

We first tested C Wolf on a number of tiny (one page or so) programs, and found it to generally work very well for them, instantaneously extracting small models which were observationally equivalent to what was expected. A prime example of this is the a^*b^* finite state machine presented at the end of the last chapter.

Of course, the real acid test is how C Wolf responds to a larger, real-world system. To determine C Wolf's usefulness, we applied to the GNU i-protocol, and also to the FTP daemon code. The purpose of analyzing the GNU i-protocol was to extract a model which reveals the finite-state protocol implemented by the code, while the purpose of analyzing the FTP daemon was to extract a model which could be use for runtime intrusion detection.

6.1 The GNU I-Protocol

The GNU i-protocol is part of the Taylor Unix-to-Unix copy (UUCP) software system, available as open source software from the Free Software Foundation. Version 1.04 of the i-protocol contained a livelock error which has been used as a test case for model checkers [24, 41], and in the words of Dong, et.al., it "makes for a formidable case study for verification tools." A primary reason for our interest is that the models used in [24] were generated manually after inspecting the source code; hence it would be interesting to see how easily C Wolf could automatically extract a model similar to the ones done by hand.

The i-protocol is a sliding window protocol, in which data packets numbered from 0 to $2 \cdot N - 1$ (cycling back to 0 once $2 \cdot N$ is reached) are stored by the sender in a ring buffer of $2 \cdot N$ packets. At any given time, the *window* is positioned to some location i in the buffer, and packets in the range $i \ldots (i + N) \bmod 2 \cdot N$ are accepted (hence, the window size is N). Once packet i is acknowledged by the receiver, the beginning of the window advances to slot $i + 1$, eventually cycling back to 0 and repeating indefinitely.

The basic messages supported by the i-protocol are DATA, ACK (acknowledgement), NAK (negative acknowledgement), SPOS (seek a file position), COMMAND (execute a command at the receiver's end), and CLOSE (terminate the protocol). Only the first three are relevant to the livelock bug, so we will not mention the last three again.

The protocol starts up by establishing a communications channel between two processes and negotiating a window size, which is typically 16. However, the livelock condition is independent of window size, so we took the approach of previous studies of the i-protocol and fixed the window size to 1. Unfortunately, we soon discovered that the underlying C code doesn't work properly unless the window size is at least 2.

The i-protocol is fully bi-directional. Either party may send data messages to the other. Acknowledgement traffic is reduced by two optimizations. The first is to piggyback an acknowledgement onto every data packet (when there is traffic going both ways), and the second is to only send explicit acknowledgements for every $\lceil \frac{N}{2} \rceil$ packets received.

6.2 Source of the Livelock

The livelock in the i-protocol is caused by the receiver process assuming that acknowledgements are reliably transmitted. As soon as the receiver sends an acknowledgement for a packet, it advances its window (assuming there are no earlier unreceived packets). If the acknowledgement is not properly transmitted, then the sender will keep retransmitting the data packet (along with a NAK packet), because it thinks the packet never made it to the receiver. The receiver is programmed to ignore any message not in the active window, so it will never respond to the data packet. The ultimate result is that the receiver will sit idly while the sender keeps re-transmitting the same data packet forever. A diagram depicting this scenario appears as figure 6.2.

	Sender		**Receiver**
1	Send DATA0	→	Receive DATA0
2		Dropped ←	Send ACK0
3			Window advanced to $[1, 1]$
4	Timeout		
5	Send NAK0	→	Ignore NAK0 since not in window
6	Resend DATA0	→	Ignore DATA0 since not in window
7	Timeout		
8	Send NAK0	→	Ignore NAK0 since not in window
9	Resend DATA0	→	Ignore DATA0 since not in window
	⋮	⋮	⋮

(Repeat 7-9 until maximum number of retries reached, then abort.)

Figure 6.1: An i-protocol message sequence in which livelock occurs (window size = 1).

6.3 Initial Experiences

In the two cited studies on the i-protocol, the protocol code was translated by hand into the input languages of several model checkers. If checking these human-abstracted models is a "formidable" challenge, then automatically extracting a model which is both useful and feasibly-sized is doubly so. The main protocol code is a 1500-line C program (not including header files), with roughly 20 important global variables. Merely extracting a model from a single call to the "send packet" function with all data values abstracted away results in a system of 280,000 states and 380,000 transitions, before optimization. We quickly discovered the limits of our platform, a dual 1-GHz pentium system with 1 GB of memory. Generating a system with a million states took a few minutes; generating one with 2 million states took about 20 minutes; and generating a system with 5 million states took 12 hours. Compounding matters, the Concurrency Workbench exhausted available memory and crashed while attempting to minimize (with respect to the weak bisimulation relation) a system of one million states, and minimizing the initial test system of 280,000 states again led to a `cwb-nc` failure due to memory exhaustion (although it should be pointed he performance of `cwmb`'s weak bisimulation minimizer is even worse). These results prompted the development of the on-the-fly minimizer which helped significantly, often reducing the state space by half, or sometimes even 80% without any perceivable overhead. The peephole optimizer is also useful, because it runs in linear time, and typically results in a 75% to 90%

Type	Name	Purpose
int	`iIrequest_packsize`	Preferred packet size.
int	`iIrequest_winsize`	Preferred window size.
int	`iIforced_remote_packetsize`	Overrides remote process' packet size.
int	`iIforced_remote_winsize`	Overrides remote process' window size.
int	`cIsync_timeout`	Timeout period while waiting for the startup packet to be acknowledged.
int	`cIsync_retries`	Number of times to resend the startup packet before giving up.
int	`cItimeout`	Timeout period while waiting for a regular packet to be acknowledged.
int	`cIretries`	Number of times to retry sending a packet before giving up.
int	`cIsync_timeout`	Period to wait before timing out.
int	`cIerrors`	Number of errors to tolerate before aborting.
int	`CIerror_decay`	After this many packets are successfully received, the number of errors is decremented.

Table 6.1: Major (constant) parameters used by the I-protocol implementation.

state space reduction (these statistics include the effect of the on-the-fly minimizer, which performs a proper subset of the peephole minimizer's optimizations). The final conclusion from these experiences is that the system generated by `cwmb` should be 100,000 states or less, after peephole and on-the-fly minimization. The resulting system can then be read into `cwb-nc` and minimized under weak bisimulation in a reasonable amount of time.

6.4 Applying Abstraction

Abstracting all variables away will not suffice if one wishes to extract a useful model from the i-protocol. Instead, the global variables must be carefully dealt with in order to get the essential information required without causing a state explosion. The major I-protocol variables are listed in two tables. Table 6.1 lists the ones whose values are fixed before startup (making them effectively constants for our purposes), and table 6.2 lists the ones which change during protocol execution.

Fortunately most of these variables can be dealt with by either statically fixing their values. The real challenge is the buffer contents. Since the underlying code requires a window size of at least 2, and each packet carries two window indices, and there are 3

Type	Name	Purpose
char array	`abPrecbuf`	Raw data to transmit.
int	`iPrecstart`	Start of unsent data (index into abPrecbuf).
int	`iPrecend`	End of unsent data (index into abPrecbuf).
int	`iIremote_packsize`	Negotiated remote process' packet size.
int	`iIremote_winsize`	Negotiated remote process' window size.
int	`iIalc_packsize`	Allocated packet buffer size.
fun ptr.	`pfIsend`	Pointer to raw "send" function.
func. ptr.	`pfIreceive`	Pointer to raw "receive" function.
int	`iIsendseq`	Next sequence number to send.
int	`iIrecseq`	Last sequence number received.
int	`iIlocal_ack`	Last sequence number locally acknowledged.
int	`iIremote_ack`	Last sequence number remotely acknowledged.
int	`iIsendpos`	File position being (locally) sent from.
int	`iIrecpos`	File position being (locally) received to.
boolean	`fIclosing`	True if and only if the protocol is shutting down.
char array	`azIsendbuffers`	Array of sent packet contents (indexed by sequence number).
char array	`azIrecbuffers`	Array of received packet contents (indexed by sequence number).
boolean array	`afInaked`	Flag telling which packets were naked (indexed by sequence number).

Table 6.2: Major global variables used by the I-protocol implementation.

Name	Purpose
`fistart`	Alternative interface to `fijstart`.
`fijstart`	Establish connection and negotiate window size.
`fishutdown`	Complete unfinished transmissions and terminate connection.
`fisendcmd`	Send a command to be executed on the remote host.
`finak`	Send a negative acknowledgement (NAK) for the current packet.
`firesend`	Retransmit the oldest active unacknowledged packet.
`fiwindow_wait`	Wait until the receiver has an open slot in its window.
`zigetspace`	Allocate space to store packet data (within `azIsendbuffers`).
`fisenddata`	Send a (new) data packet.
`fiwait`	Alternative interface to `fiwait_for_packet`.
`fiwait_for_packet`	Busy wait for a packet to come in, or for a packet to go out.
`ficheck_errors`	Check the error level (`cIerrors`) and abort if it is too large. Also attempts to fix things by shrinking the packet size.
`fiprocess_data`	Process data waiting in the receive buffer.
`fiprocess_packet`	Process a packet (called by `fiprocess_data`).

Table 6.3: I-protocol functions.

fundamental packet type (ACK, NAK, DATA), the total number of basic packet kinds is $4 \cdot 4 \cdot 3 = 48$. Since the receive buffer can hold 4 packets, the total number of buffer states it can have is $48^4 \geq 5$ million. The send buffer has the same number of states for a total of roughly 25 trillion ($25 \cdot 10^1 2$) states.

We can deal with this by observing that packets outside the current window are irrelevant and can be abstracted away. Unfortunately, our tools are not currently strong enough to provide this abstraction, so a compromise is to modify the underlying C code to zero out the buffer contents once they have been processed. This reduces the number of buffer states for a single buffer to about $4 \cdot 48^2 = 9216$. The total number of states in both sender and receiver is $9216^2 \geq 8$ million, which is still too big to extract a fully general model (where any packet could be received at any time), particularly when one considers that the buffer states must be multiplied by the number of control-flow states and states of other global variables. However, we can still extract partial models which pertain to certain behaviors of the i-protocol implementation.

The i-protocol code was linked into a test harness which simulates the receiving and sending of packets. The test harness links seamlessly with the i-protocol code, because the

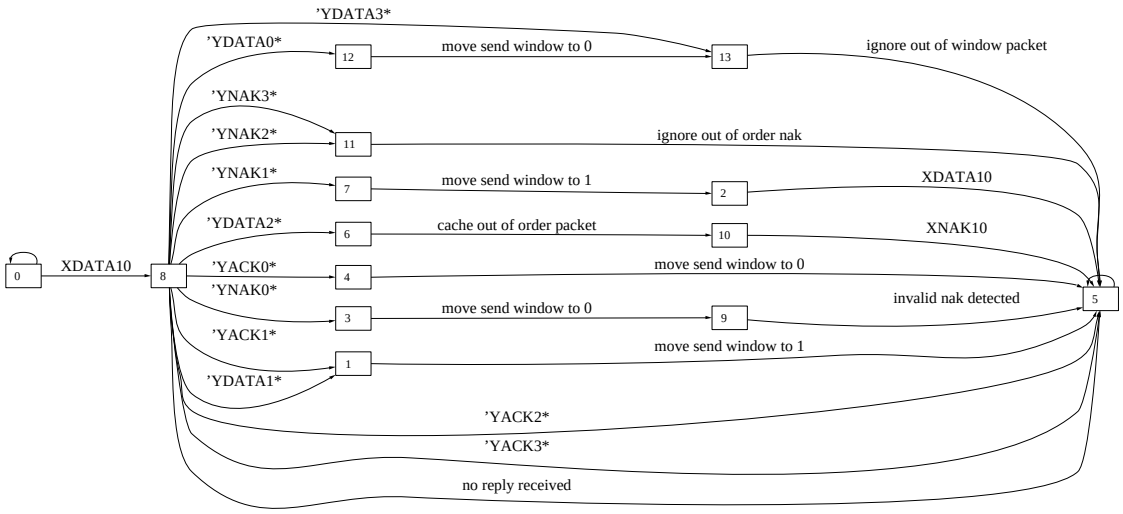

Figure 6.2: Behavior of I-protocol when sending 1 data packet, and receiving a reply.

i-protocol code relies on two function pointers, `pfIsend` and `pfIreceive`, to transmit and receive data. These functions pointers are bound at startup to point to the test harness code instead of the original network transmission code. Additionally, the protocol startup phase is skipped, and instead the window size is fixed to 2 bytes (the smallest value which doesn't "break" the i-protocol code), and the (data) packet size is set to 18 bytes, the smallest value which is a multiple of the control packet size (6 bytes). Similarly, the protocol shutdown, repositioning, and resizing operations are never invoked.

The test harness code does two jobs. It takes outgoing packets and analyzes them, encoding them into integer values in the range $0 \ldots 47$. The outgoing value is then assigned to a global variable which is the target of a watch clause in the label-map. The watch clause generates a label for each packet value, describing the packet kind, and the position of the sender's incoming and outgoing windows.

The I-protocol functions are listed in table 6.3. The test harness has a main function which explicitly calls these functions as required to generate partial models of the I-protocol's behavior. Figure 6.2 shows the case where the test harness first calls `fisendata`, to send a data packet, and then calls `fiprocess_data` to process the reply (which could be any of the 48 packets). To make life simpler, multiple transitions have been merged in the

case where the receiver or sender window number doesn't affect i-protocol behavior. These cases are indicated by a * in the label, for example, the transition `'YACK1*` really stands for 4 transitions, labeled `'YACK10` through `'YACK13`. In addition, the transitions labeled in uppercase represent packets, where packets prefixed with `'Y` are incoming packets from an external process we have named Y, and packets prefixed with `X` are outgoing packets from the process we are modeling (named X). The lower-case labels are for illustrative purposes only, and were added by inserting calls to the special `_cwolf_event` function, in order to describe the internal behavior going on within the i-protocol code. In particular, the model of figure 6.2 shows the dangerous behavior of the sender — it completely ignores negative acknowledgments outside of its window (`'YACK2*` and `'YACK3*`). Hence, if the receiver ever gets stuck waiting for a packet outside the sender's window, then livelock will result.

6.5 FTP Daemon

The program *ftpd* is a process which runs continuously on a UNIX system, processing requests to transfer files over the network. It has been shown, unfortunately, that some implementations of ftpd may be vulnerable to a *buffer overflow* attack, in which a user intentionally writes beyond the boundaries of an array stored on the stack and replaces the return address of the current activation record, allowing arbitrary code to be executed. One strategy for defending against such attacks is *process monitoring*, in which the behavior of a process is monitored at runtime, and the system calls it performs are compared against a model. When a process performs a call not captured in the model, an alert is raised. Recent research has used static analysis to extract a non-deterministic pushdown automaton from a program's source code, and used this model as the basis against which to compare runtime behavior [68].

We can easily duplicate the NDPDA of [68] by running C Wolf over the ftpd sources with no abstractions, and instructions to label all system calls. The C Wolf approach has the advantage that the result is a minimized NFA, which is much easier to deal with than an NDPDA. Figure 6.3 shows the NFA extracted by C Wolf from BSD ftpd version 0.3.3, abstracting all variables away and optimized with respect to observational equivalence. The NFA is quite tractable as a basis for runtime monitoring, with a total of 22 states. Of course, a major advantage of the C Wolf system is that values of variables can easily be

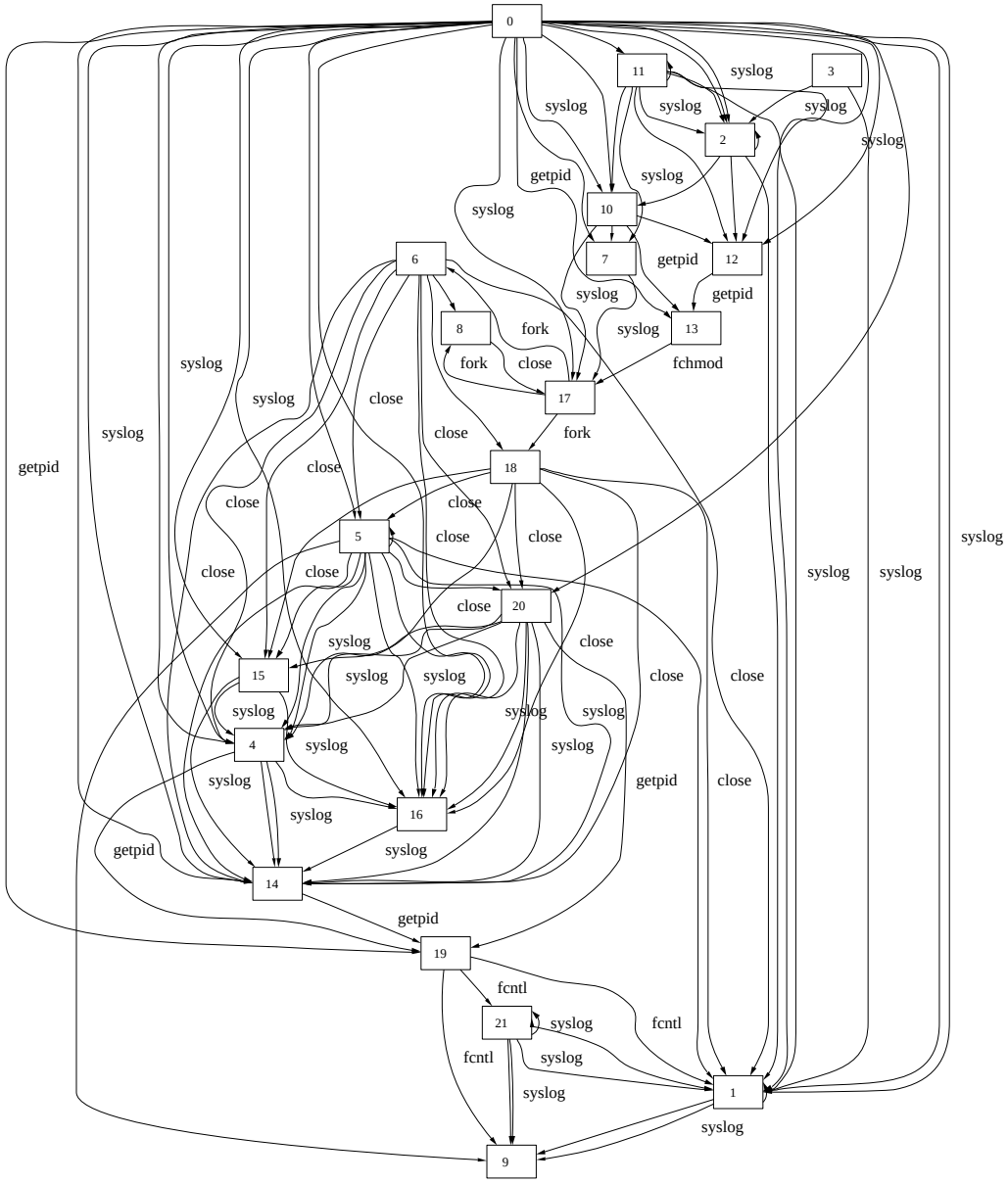

Figure 6.3: System-call structure of the FTP daemon, with all variables abstracted away.

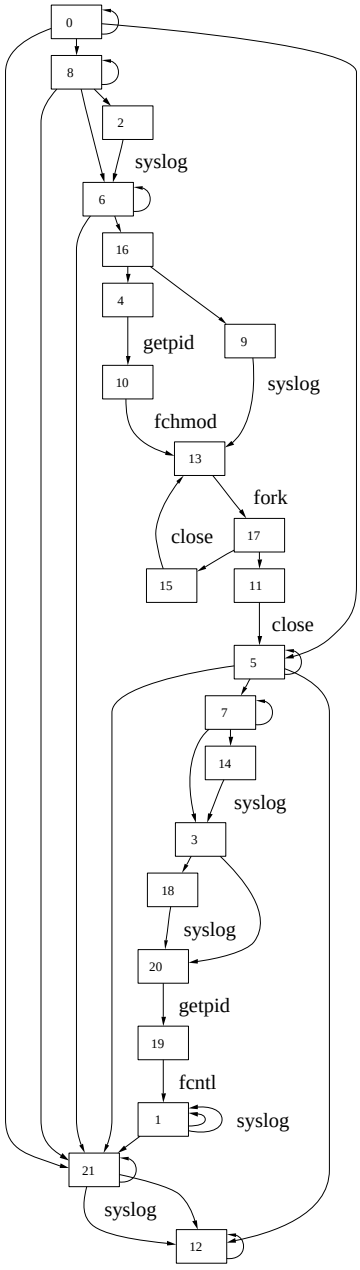

Figure 6.4: System-call structure of the FTP daemon, with all variables except `daemon_mode` abstracted away.

incorporated into the model. A prime example of this is the LTS of figure 6.4. The only difference between it and the LTS of figure 6.4 is that a single variable, `daemon_mode`, was incorporated into the model. We believe that this LTS is much more precise than can be achieved using the technique of [68], which has no mechanism for tracking data values.

Chapter 7

Conclusion

The work presented in this thesis addresses the problem of abstraction-based model extraction from C programs. By developing the C Wolf system, we have shown that it is possible to use apply user-directed abstract interpretation to a C program and extract a model of its behavior, which can then be compared to see if it is observationally equivalent to a specification, can be model-checked to see if it satisfies various temporal properties, or can be used as a reference to check for malfunctions during future executions.

The model extraction process itself is, strictly speaking, unsound because it is always possible for a program to alias an automatic or static storage location on the heap, and then overwrite that location without it being reflected in the model. We have made the C Wolf system as sound as possible, however, by carefully checking the correctness of the all the abstract operations, and hence, barring aliasing via dynamic memory, or the user willfully abstracting away a pointer which aliases some important variable, the extracted model will be sound in the sense that abstract executions will include all possible concrete executions. The behavior, however, is under-constrained, by which we mean that due to abstraction it may not be possible to determine the outcome of some conditional branches within the abstracted program, and this will result in a non-deterministic model which may include paths that aren't feasible in the concrete program. Such a model may satisfy certain temporal properties which the concrete program doesn't. It is up to the user to be aware of this possibility and guard against such "pseudo-nondeterminism" causing the verification process to be subverted.

There were a number of surprises and lessons learned during the development of the C Wolf system. One tough problem in particular was extracting models which were observationally equivalent to their specification. Initially, we believed that the outcomes of conditional branches could be used to drive the labeling process. However, if you assign a value to some variable x, and later generate labels based on the outcome of a conditional branches, then one typically ends up with a system which appears to make several internal choices and exhibits a pseudo-determinism. To get rid of that problem, we came up with the semantics for the `watch` clause, which splits the execution immediately at the point where a variable is assigned a (possibly unknown) value, and attaches different labels to each branch. When the labels are input actions, then the extracted system correctly models a system which picks one of several possible branches based on input from the environment.

Another surprise was the importance of constraint analysis. Initially, we had planned to just implement the C operators and ignore information that could be inferred from conditional branches. However, when one does this, the model that gets extracted often ends up being worthless

A pleasant, surprise was the power of the `part` extraction. As we noted in chapter 5, by parameterizing `part`, we were able to capture almost all the abstractions employed by Bandera, without having to write any operator code. We would still like to augment `part` with some additional capabilities, such as boolean and symbolic constraints, which we'll discuss further in section 7.1.

Other surprises were not so pleasant. In particular, we aren't very happy with the memory usage of C Wolf, particularly when computing the weak bisimulation relation. Given that the Concurrency Workbench also has problems computing the same relation, we suspect the real problem lies in the implementation of Standard ML that we are using, namely Standard ML of New Jersey. The choice of ML was driven by our desire to be able to tightly integrate C Wolf with the Concurrency Workbench. Since they ended up as separate components, in retrospect it would have been better to use a more efficient compiler, probably OCaml.

More importantly, applying C Wolf to real-world programs is difficult. A lot of time was spent attempting to verify the I-protocol. The major problem appears to be array contents. Unfortunately, for communications protocols, the headers of every packet must be maintained, and these headers are typically stored in ring buffers. The contents of such a buffer can't be ignored if one wishes to extract a useful model. Making the problem worse,

there are commonly numerous pointers into such buffers at various places in the program. Incorporating the packet contents into the model results in a much larger environment, and many more states. Hence, we were unable to extract a complete and general model of the I-protocol behavior, but had to resort to extracting partial models which demonstrate various behaviors of the system. We believe that for other applications, which aren't as heavily dependent on buffer contents, such as device drivers, C Wolf can be applied more fruitfully without any additional abstractions required. Also, we believe that some of the extensions we have planned for the future will greatly expand the pool of systems to which C Wolf can be applied. We discuss these ideas in the next section.

7.1 Future Work

The work done on C Wolf so far barely scratches the surface of the research issues involved in software model extraction. There are numerous potential avenues for future research, and we hope that the C Wolf system will serve as a platform for exploring them. The work to be done in the future can be group into three categories. First, are minor extensions. These are new features which extend the capabilities of C Wolf in a relatively minor way, and can be added in a relatively small amount of time. An example of this would be adding support for tracking the values of C structures. Second are performance-related enhancements. These enhancements aren't intended affect the functionality of C Wolf, just its space and time requirements (although, it is reasonable to provide the user with the options that sacrifice functionality in return for better performance). The work required for implementing such enhancements varies, generally in proportion to the benefit derived. Finally are major revisions which extend the capabilities of C Wolf in major new directions. This is where the real research is, and also the greatest risk, since a lot of work is required in return for what are essentially unpredictable results.

We present now a quick "laundry list" of topics for future work. These ideas will be discussed in more detail in the rest of section. We have tried to partition them into the three categories of extensions, performance enhancements, and major revisions, although the boundaries between these areas are somewhat blurry.

- Support for concurrency (`fork`, `wait`, etc.)

- Support for additional minor features of C, such as floating-point, `setjump/longjmp`, structs and unions.
- Support for dynamic memory.
- Support for other languages, such as C++, and Java.
- Support for machine code (the ability to generate models from programs which have already been compiled into binary form).
- Enriching the labeling language with more clauses and more powerful expressions.
- Additional abstractions, in particular more powerful array abstractions to support strings and sparse arrays.
- Adding support for *monovariance* (classical intraprocedural data flow analysis)
- Porting C Wolf to another ML compiler, such as *MLton*, in order to improve performance.
- Translating C Wolf to *ocaml* to improve performance.
- Implementing a GUI (graphical user interface) to improve the usability of C Wolf.
- Strengthening the on-the-fly minimizer to improve model generation performance
- Support for other model types, such as pushdown automata.
- Automatically inferring the label and abstraction maps based on the properties the user wants to verify.
- Support for predicate abstractions.
- Support for symbolic abstractions.
- Support for user-generated abstractions via an abstraction specification language.
- Additional case studies.
- Better tools for manipulating and visualizing large graphs.

7.1.1 Extensions

There are many extensions that could be done to C Wolf. First and foremost is support for the full C language. Remaining to be supported are multidimensional arrays, floating-point operations, and structs and unions. There are also a number of important system calls, such as `fork`, which provides concurrency, `setjump` and `longjmp`, which provide a sort of exception handling (and can be easily added to C Wolf), `signal`, which sends an interrupt to a process, and `socket`, which creates a communications channel between processes.

Concurrency support is a real challenge because of the potential of state explosion. It should be possible to use some sort of partial-order representation, with independent processes springing into life after a call to `fork`, and merging at a `wait` call. An additional complication would be interaction between processes via signals or sockets. The ideal for working with the concurrency workbench would be to generate separate agents for independent process subsections, and use the workbench's parallel composition operation to simulate their running in parallel. Unfortunately, the Concurrency Workbench has no support for partial ordering, so the final number of states would be the product of the states in each parallel process.

In addition to concurrency, support for tracking dynamically allocated values would be useful. Currently, all of dynamic memory is abstracted away. Supporting "fixed shape" objects, such as dynamically allocated arrays and fixed structures would be relatively easy to do. Dynamically shaped structures, such as linked lists, and trees would be more problematic. One Possibility would be to only track the first N nodes within a data structure (where the user picks N), and abstracting away the extra nodes.

An additional extension would be support for C++. This would require a lot of work, as a new parser would be needed, and some extensions to the CWI bytecode would probably be required to support virtual functions and the C++ type system. Currently, most of the systems programs of interest are written in C only, so such an extension may not be worth the investment. Even more challenging would be support for non-C languages, such as Java, Fortran, ML, an so on. One possibility for doing this is to hack the GNU compiler suite to generate CWI code from RTL code (gcc's intermediate code). Such a modification to gcc would be a lot of work, but would provide a very robust C interface, plus make it possible to support all the other GNU languages.

The abstraction and labeling languages could also be extended in numerous ways. Some additional abstractions include a string-prefix abstraction, where a string is abstracted by as much of its prefix as can be determined. An example of this would be the case where a string S' is generated by appending the string "/etc/" to the string S (whose contents are unknown). In this case, the prefix abstraction would be useful for determining that S' string refers to a file within the `/etc/` directory — which contains sensitive system files such as the passwords for all the users.

Also useful would be a $div(k)$ abstraction, which would map concrete values onto their quotient when divided by k. For example, in a filesystem where pages consist of 256 bytes, a $div(256)$, abstraction would map byte locations onto the page within the file they refer to. This would be useful in cases where one is only interested in which page is accessed, not which byte.

Also useful would be more powerful array abstractions. A good example of this is the i-protocol test harness, whose main job is to map integer values in the range $0 \ldots 47$ onto array contents. For network protocols, it would be very useful to be able map entire packet contents onto a small enumeration, and have the byte- and word-level array accesses be automatically supported by the abstraction mechanism. A hard problem is caused by the fact that array contents aren't generated by a single assignment, but instead are generated by a collection of assignments over time. The abstract value isn't known until the last relevant assignment occurs.

Another useful array abstraction would be support for "sparse" arrays. The user could either specify particular indices of interest, or the system could only store values which deviate from a particular default value, or the system could only store the location of particular values of interest. For example, to check for properly terminated strings, we only care about which cells within a character array contain a zero.

Finally, the labeling language could be enriched in order to allow the user be more specific about when to generate a label. Some specific ideas are a larger expression language for the `watch` clause, and the ability to conjoin multiple conditions, such a condition being true immediately after a specified line of code is executed, or a function call occurs and a specific condition holds true, and so on.

7.1.2 Performance Enhancements

One problem is the memory usage of the implementation language, Standard ML of New Jersey (SML/NJ) [2]. Compared to some of its peers, SML/NJ is quite a memory hog. There are several other ML compilers available, such as MLton [12], which transforms an ML program into C, and Poly/ML [31], which generates a heap file like SML/NJ, but claims to have better performance. The biggest problem with switching compilers is that the current implementation is dependent on the SML Compilation Manager (CM), which is only available in SML/NJ. Also, the robustness of the other compilers is uncertain (to the author, at least), as an earlier attempt to use MLton resulted in a compiler crash. At this point in time, it appears that none of the popular ML compilers are being very actively maintained, which brings the future of ML itself into doubt. The current "hot" ML-like compiler is the OCaml compiler developed at INRIA. OCaml generates native code whose performance is close code produced by `gcc`, and orders of magnitude better than the output of SML/NJ. Unfortunately, the authors of OCaml made a bunch of syntactic changes to the language, so while their language is close to ML semantically, manually translating an ML program to OCaml is a painfully laborious process. One possibility for dealing with this would be to produce an ML to OCaml translator, which might take less time than translating the 50,000 lines of C Wolf code to OCaml by hand.

Another strategy for improving the system performance is to improve the on-the-fly minimizer. While the minimizer has proved to be very useful, it still results in systems which often hundreds or thousands of times bigger than the result of post-generation optimization (using the weak-bisimulation / observational equivalence relation). One possibility is to identify subgraphs of the system which can be independently optimized before the entire system has been generated. For example, if there is a subset of states with a single entry point, and we can determine that all future states will not branch to any state other than that single entry point (e.g., the entry point is the start of a function and the other states are internal computations of that function), then the subgraph could be minimized using weak bisimulation before the rest of the system is generated.

Another possibility is some sort of incremental minimization. In an earlier, unpublished work, we developed an on-the-fly algorithm which used a *partition tree* to compute the bisimulation relation in synchronous lock step with the generation of states [25]. A partition tree maintains information about which partition each state belongs-to under the

strong-bisimulation relation. An algorithm similar to the on presented in [25] could be incorporated into the state generation process, which would at least allow the post-generation optimizations to be skipped, as the bisimulation relation would be known soon after the last state was generated. It would also be interesting to extend the algorithm to use the weak-bisimulation relation instead of strong-bisimulation.

A third direction would be to reduce the amount of state splitting within the system, by providing support for *monovariance*. Monovariance, we mean that some object (either a program location, a function, or a variable) isn't allowed to have variants. Instead, all information about that object is merged into a single abstract value. For example, if a function f were to be treated monovariantly, then all calls to f would go to the same subset of states. Within this subset the values of the variables would be abstract values reflecting all possible concrete values which could be passed-in by any caller from anywhere within the program (and is basically the same as classical data-flow analysis). The result of the monovariance is many fewer states, at the cost of having a much coarser abstraction of the program's behavior.

A technical problem which crops up with monovariant functions is keeping track of where the function returns after it is called. This could be handled by generating a pushdown automaton instead of a finite one. This would require a model-checker which supports pushdown automata (PDAs), such as XMC [62]. Generating a PDA would also have the advantage of allowing the program stack to be squashed to just the top value, significantly reducing the size of each state and also reducing the state space in cases where a function is called with the same arguments and same global-variable environment from multiple locations. Implementing an option for producing a PDA should be relatively easy, requiring only minor modifications in the representation of states and the notion of equality between states.

A final, orthogonal area which would help improve performance would be a compact binary graph representation, and tools for manipulating and viewing binary graph files. A major bottleneck in the current system is the time required to compute a weak bisimulation relation over a graph. A second problem is the fact the graph viewing tools currently employed by C Wolf (*dot* and *daVinci*) don't work well once the size of a graph exceeds a few hundred nodes. It is our belief that a graph manipulation library implemented in a more memory-efficient language (such as *OCaml* or C) would significantly improve the performance of graph minimization. Also extremely useful would be a graph viewer for

large graphs. The key to implementing this viewer would be to localize the set of nodes to just those that are being rendered. Unfortunately, dot attempts to render all nodes immediately (probably unavoidable, since its output format is Postscript), and daVinci slows unacceptably when graph size grows large. This makes graphically examining large graphs impossible, and there is no reason why that should be the case.

7.1.3 Major Revisions

There are a number of deeper issues which could be explored by radically modifying the C Wolf system. One possibility is adding support for predicate abstractions in combination with the existing abstractions. The abstraction language could be augmented by a list of predicates (boolean expressions). For each variable involved in a particular predicate, 2 bits would be added to each environment. The 2 bits would encode whether or not the predicate was currently satisfied (using the values $true, false$ or $unknown$). This would provide the ability to things like keep track or whether or not the variable x is less than the variable y, without applying a fine-grained abstraction to both variables. Within each environment, whenever x or y was assigned a new value, the system would update the truth value for the predicate based on the new values. Additionally, the constraint analyzer would also be modified to update predicate information along each conditional branch taken within the program. For example if $x \mapsto \top$ and $y \mapsto top$, and the program hits the statement `if` $(x < y)$, then in the environment following the positive (true) branch, x and y will still both be $\top$, but the predicate value for $x < y$ will be set to $true$, which could be important if, say y is the size of a dynamically-sized array and x is an index into that array. Modifying the state representation to support predicates would be fairly easy; the hard part is extending the abstraction mapping language and implementing the checks required to keep all the predicates up to date.

In addition to predicate abstraction, another area we would like to investigate in the future is the possibility of adding symbolic abstractions. Such abstractions would be represented as abstract syntax trees of expressions over variables and constants, with limits of the size of the tree. One case where this would be useful would be to represent the fact that string s is equal to the concatenation of strings d and f, which would come into play when s was passed to `fopen`, and the system was attempting to determine if the directory was was the same d or not, and the precise value of d was not known. One possibility

for integrating symbolic abstraction with the current abstraction mechanism is to have the system use concrete values when they are available, and when values are unknown (i.e., $\top$), use a symbolic value instead. A second possibility is to keep two values for each variable, an abstract data value, and a symbolic value.

Automatic abstraction is another topic for future exploration. By "automatic abstraction," we mean a process by which, given a program and a specification, the verification system mechanically chooses the appropriate abstraction. In the C Wolf system, this would correspond to a technique for automatically generating the contents of the abstraction map and/or label map files. One potential strategy is, given a formula Φ, to start with all variables abstracted to $\top$, and extract a model M. If $M \models \Phi$, or $M \models \neg\Phi$, then the process terminates successfully. Otherwise, the system chooses a variable for abstraction (a good choice would be one involved in many conditional branch expressions). The abstraction for the variable could be `free` or it could be a `part` abstraction where the breakpoints are chosen by examining the constant values the variables was compared against. A new model M' could then be extracted, and it could be checked to see if $M' \models \Phi$ or $M' \models \neg\Phi$. The process would repeat until either memory runs out, the specification is determined to be satisfied or unsatisfied, or (unlikely) no further refinements can be made to the abstraction map.

A final abstraction-related issue is the ability to for the user to come up with his or her own abstractions and add them to C Wolf. Currently, this would require modifying the system's source code and recompiling the system. A more sophisticated way would be to provide some sort of abstraction language for specifying an α function and implementations of all the appropriate C operations. The abstraction language would have to be fairly powerful, and an important issue would be the correctness of the operator implementations. One possibility would be to have the system verify any implementation by evaluating them over all abstract pairs and checking the result for correctness. A particular application where such a language would be useful would be in abstracting array values. An protocol-specific abstraction could be implemented in which the array contents was represented as a list of packet descriptors, but the array operations are implemented by the abstraction code provided by the user.

An area which would require a significant amount of work, but might not be classified as "hard" research is to add a GUI (graphical user interface) to C Wolf. A GUI would improve the usability of the tool and help tie together the various components into

a single cohesive system.

Finally, and perhaps most importantly, additional case studies need to be done. Some specific ideas for systems to examine are the CAN protocol, device drivers (in particular the Universal Serial Bus device drivers for Linux), and internet messenger clients such as GAIM.

7.2 Summary

The C Wolf system described in this thesis is an attempt to apply model checking techniques to C programs. Through a user-directed data abstraction and labeling process, it has been shown that, if the program is "well-behaved" in its use of pointers (in particular, the program's heap values can't contain pointers which are used to overwrite non-heap values), then it is practical to extract a labeled transition system which captures all possible execution paths within the original C program. Such an LTS forms a sound basis for verifying LTL properties, because any LTL proposition which holds true for the extracted LTS will also hold true for the LTS which would result from capturing all possible execution paths of the original program. Additionally, an extracted LTS can be compared for observational equivalence to designs expressed in a process algebra. While the C Wolf method is far from being a panacea for buggy software, it is useful in many cases, and we hope to extend it in the future to make it much more useful.

Bibliography

[1] Anonymous. Inquiry board traces Ariane 5 failure to overflow error. *SIAM News*, 29(8), October 1996.

[2] Andrew W. Appel and David B. MacQueen. Standard ML of new jersey. In J. Maluszyński and M. Wirsing, editors, *Proceedings of the Third International Symposium on Programming Language Implementation and Logic Programming*, number 528 in Lecture No‘tes in Computer Science, pages 1–13. Springer Verlag, 1991.

[3] Ball, Podelski, and Rajamani. Boolean and cartesian abstraction for model checking C programs. In *TACAS: International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, 2001.

[4] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram Rajamani. Automatic predicate abstraction of `c` programs. In *Proceedings of the ACM SIGPLAN 2001 Conference of Programming Language Design and Implementation (PLDI 2001)*. ACM Press, June 2001.

[5] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *The 8th International SPIN Workshop on Model Checking of Software (SPIN 2001)*, volume 2057 of *LNCS*, pages 103–122, New York-Berlin-Heidelberg, May 2001. Springer-Verlag.

[6] Thomas Ball and Sriram K. Rajamani. Bebop: A path-sensitive interprocedural dataflow engine. In *Proceedings of the 2001 ACM SIGPLAN - SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE-01)*, pages 97–103, New York, June 18–19 2001. ACM Press.

[7] Thomas Ball and Sriram K. Rajamani. The slam toolkit. In *13th Conference on Computer Aided Verification (CAV '01)*, volume 2102 of *LNCS*, New York-Berlin-Heidelberg, July 2001. Springer-Verlag.

[8] Guillaume Brat, Klaus Havelund, SeungJoon Park, and William Visser. Java pathfinder: Second generation of a java model checker, July 2000.

[9] Derek L. Bruening. Systematic testing of multithreaded java programs. Master of engineering thesis, Massachusetts Institute of Technology, 1999.

[10] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[11] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[12] Henry Cejtin, Matthew Fluet, Suresh Jagannathan, and Stephen Weeks. Mlton user guide, October 2001. Available from URL `http://www.sourcelight.com/MLton`.

[13] Andy Chou, Benjamin Chelf, Dawson Engler, and Mark Heinrich. Using meta-level compilation to check FLASH protocol code. *ACM SIGPLAN Notices*, 35(11):59–70, November 2000.

[14] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[15] Edmund M. Clarke, M. Khaira, and X. Zhao. Word level model checking-avoiding the pentium FDIV error. In *33rd Design Automation Conference (DAC'96)*, pages 645–648, New York, June 1996. Association for Computing Machinery.

[16] Edmund M. Clarke, David E. Long, and Ken L. McMillan. Compositional model checking. In *Proceedings, Fourth Annual Symposium on Logic in Computer Science*, pages 353–362, Asilomar Conference Center, Pacific Grove, California, 5–8 June 1989. IEEE Computer Society Press.

[17] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The Concurrency Workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[18] James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera: extracting finite-state models from Java source code. In *22nd International Conference on Software Engineering*, pages 439–448, Limerick, Ireland, June 2000. IEEE Computer Society.

[19] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, January 1977.

[20] Patrick Cousot. Abstract interpretation. *ACM Computing Surveys*, 28(2):324–328, June 1996.

[21] Patrick Cousot and Radhia Cousot. Refining model checking by abstract interpretation. *Automated Software Engineering: An International Journal*, 6(1):69–95, January 1999.

[22] Claudio Demartini, Radu Iosif, and Riccardo Sisto. A deadlock detection tool for concurrent java programs. *Software: Practice and Experience*, 29(7):577–603, June 1999.

[23] Edsgar W. Dijkstra. The Humble Programmer. *Communications of the ACM*, 15(10):859–866, October 1972. transcript of Turing award lecture.

[24] Yifei Dong, Xiaoqun Du, Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, Scott A. Smolka, Oleg Sokolsky, Eugene W. Stark, and David Scott Warren. Fighting livelock in the i-protocol: A comparative study of verification tools. In *Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 74–88. Springer-Verlag, 1999.

[25] Daniel C. Duvarney and S. Purushothaman Iyer. On-line minimization with partition trees. 1999.

[26] M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, Robby, C. S. Păsăreanu, W. Visser, and H. Zheng. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE-01)*, pages 177–187, Los Alamitos, California, May12–19 2001. IEEE Computer Society.

[27] Orna Grumberg Edmund M. Clark, Jr. and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA (USA), 1999.

[28] E.M. Clarke and E.A. Emerson. Synthesis of Synchronization Skeletons for Branching Time Temporal Logic. In *Logics of Programs: Workshop*, volume 131 of *Lecture Notes in Computer Science*, Yorktown Heights, New York, May 1981. Springer-Verlag.

[29] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *4th Symposium on Operating System Design and Implementation*, Berkeley, CA, October 2000. USENIX Association.

[30] Dawson R. Engler. Interface compilation: Steps toward compiling program interfaces as languages. *IEEE Transactions on Software Engineering*, 25(3):387–400, May/June 1999. Special Section: Domain-Specific Languages (DSL).

[31] Simon Finn. Using poly/ml, 1996. Available from URL `http://www.polyml.org`.

[32] M. Fröhlich and M. Werner. Demonstration of the interactive graph-visualization system davinci. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing*, volume 894 of *Lecture Notes in Computer Science*, pages 266–269. DIMACS, Springer-Verlag, October 1994. ISBN 3-540-58950-3.

[33] M. Fröhlich and M. Werner. The graph visualization system daVinci - A user interface for applications. Technical Report 5/94, Department of Computer Science; University of Bremen, September 1994.

[34] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *The 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1997)*, pages 174–186, Paris, France, 1997. ACM SIGACT and SIGPLAN, ACM Press.

[35] Patrice Godefroid, Bob Hanmer, and Lalita Jagadeesan. Model checking without a model: An analysis of the heart-beat monitor of a telephone switch using verisoft. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '98)*, pages 124–133, Clearwater Beach, FL, March 1998. ACM Press.

[36] Patrice Godefroid, Bob Hanmer, and Lalita Jagadeesan. Systematic software testing using verisoft: An analysis of the 4ess heart-beat monitor. *Bell Labs Technical Journal*, 3(2), April-June 1998.

[37] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. *Lecture Notes in Computer Science*, 1254:72–??, 1997.

[38] Samuel P. Harbison and Guy L. Steele, Jr. *C: A Reference Manual*. Prentice-Hall, Englewood Cliffs, NJ (USA), 1987.

[39] John Hatcliff and Matthew Dwyer. Using the bandera tool set to model-check properties of concurrent java software. In *CONCUR: 12th International Conference on Concurrency Theory*. LNCS, Springer-Verlag, 2001.

[40] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4), april 1998.

[41] Gerard J. Holzmann. The engineering of a model checker: the gnu i-protocol case study revisited. In *Proceedings of the 6th Spin Workshop*, volume 1680 of *LNCS*, Toulouse, France, Sept. 1999. Springer Verlag.

[42] Gerard J. Holzmann. Logic verification of ANSI-C code with SPIN. In *Proceedings of the 7th International SPIN Workshop*, volume 1885 of *LNCS*. Springer-Verlag, September 2000.

[43] G.J. Holzmann and Margaret H. Smith. Software model checking - extracting verification models from source code. In *Formal Methods for Protocol Engineering and Distributed Systems*, pages 481–497, Kluwer Academic Publ., Oct. 1999. also in: Software Testing, Verification and Reliability, Vol. 11, No. 2, June 2001, pp. 65-79.

[44] G.J. Holzmann and Margaret H. Smith. Automating software feature verification. *Bell Labs Technical Journal*, 5(2):72–87, April-June 2000. Issue on Software Complexity.

[45] Radu Iosif. *A Formal Approach to Practical Software Verification*. PhD thesis, Politecnico di Torino, November 2000.

[46] Radu Iosif. Formal verification applied to java concurrent software. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 707–709, June 2000.

[47] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming*, 1981.

[48] Ken L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1992. CMU-CS-92-131.

[49] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ (USA), 1978.

[50] Eleftherios Koutsofios. Drawing graphs with *dot*. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, USA, November 1996. This report, and the program, is included in the `graphviz` package, available for non-commercial use at URL `http://www.research.att.com/sw/tools/graphviz/`.

[51] David Ladd, Satish Chandra, Michael Siff, Nevin Heintze, Dino Oliva, and Dave MacQueen. *Ckit*: A front end for c in sml, March 2000. Available from URL `http://cm.bell-labs.com/cm/cs/what/smlnj/doc/ckit/index.html`.

[52] Nancy Leveson. *Safeware: System Safety and Computers*. Addison Wesley, 1995.

[53] Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, July 1993.

[54] David Lie, Andy Chou, Dawson Engler, and David L. Dill. A simple method for extracting models from protocol code. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 192–203, Göteborg, Sweden, June 30–July 4, 2001. IEEE Computer Society and ACM SIGARCH.

[55] Ken L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell Massachusetts, 1993.

[56] Robin Milner. *A Calculus of Communicating Systems*, volume 158 of *Lecture Notes in Computer Science*. Springer-Verlag, New York-Berlin-Heidelberg, 1983.

[57] Peter G. Neumann. *Computer-Related Risks*. ACM Press / Addison Wesley, 1995.

[58] Robert Paige and Robert Endre Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987.

[59] Doron Peled. Ten years of partial order reduction. In *Proc. 10th International Computer Aided Verification Conference*, pages 17–28, 1998.

[60] G.J. Holzmann Peter R. Gluck (NASA/JPL). Using spin model checking for flight software verification. In *Proc. 2002 Aerospace Conference*, Big Sky, MT, USA, March 2002. IEEE.

[61] R. Cleaveland and S. Sims. The NCSU concurrency workbench. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 394–397, New Brunswick, NJ, USA, July/August 1996. Springer Verlag.

[62] C. R. Ramakrishnan, I. V. Ramakrishnan, Scott A. Smolka, Yifei Dong, Xiaoqun Du, Abhik Roychoudhury, and V. N. Venkatakrishnan. XMC: A logic-programming-based verification toolset. In *Computer Aided Verification*, pages 576–580, 2000.

[63] Richard M. Stallman. Using and porting gnu cc, 1989.

[64] Scott Stoller. Model checking multi-threaded distributed java programs. In *Proceedings of the 7th International SPIN Workshop*, volume 1885 of *LNCS*, pages 224–244. Springer-Verlag, 2000.

[65] Patrick Thibodeau. Final cost of y2k repairs may top $114 billion, November 1999.

[66] William Visser, Kluas Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In P Alexander and Pierre Flener, editors, *Proceedings of ASE-2000: The 15th IEEE Conference on Automated Software Engineering*, Grenoble, France, September 2000. IEEE Computer Society Press.

[67] William Visser, SeungJoon Park, and J. Penix. Using predicate abstraction to reduce object-oriented programs for model checking, August 2000.

[68] David Wagner and Drew Dean. Intrusion detection via static analysis. In Francis M. Titsworth, editor, *Proceedings of the 2001 IEEE Symposium on Security and Privacy (S&P-01)*, pages 156–169, Los Alamitos, CA, May 14–16 2001. IEEE Computer Society.