C Wolf – A Toolset for Extracting Models from C Programs*

Daniel C. DuVarney[1] and S. Purushothaman Iyer[2]

[1] Dept of Computer Science, SUNY at Stony Brook, NY, 11794-4400
dand@cs.sunysb.edu
[2] Dept of Computer Science, North Carolina State University, Raleigh, NC 27695-7534
purush@csc.ncsu.edu

Abstract. We report on the design, implementation, and use of C Wolf, a toolset which extracts finite labeled transition systems from C programs. The extraction process is guided by user input on how a program should be abstracted, and what events should be made observable to the user. The output is an abstracted model suitable for input to the Concurrency Workbench. Additionally, facilities are provided to carry out simple observational equivalence-preserving transformations which reduce the size of the generated model. Finally, we report our experiences in using the toolset to analyze the GNU i-protocol (Version 1.04) and the BSD ftp daemon (Version 0.3.3).

Keywords: Abstract Model checking, C programs, Concurrency Workbench, FDT for network protocols, Verification and validation, Software tools.

1 Introduction

The past twenty years have seen great strides in the use of model checking to validate finite state designs of programs and circuits. Due to the impressive results that have been achieved in these limited applications [9], there have been efforts to apply model checking to software systems and to infinite state designs [4,6]. The former style of work is based on abstracting programs to arrive at finite state designs, which can then be model-checked. The latter, however, depends upon considering decidable properties of certain subclasses of infinite state systems. In this paper we address the approach of abstraction.

We consider the problem of generating finite labeled transition systems from C programs, with a view towards reasoning with the abstracted version in the Concurrency Workbench [34]. Such an approach would also allow one to relate designs and implementations by comparing designs against abstractions of implementations. The utility of finite-state abstractions of programs is supported

* Supported in part by ARO under grants DAAG55-98-1-03093 and DAAD-19-01-1-0683, and by NSF under 0098037

D.A. Peled and M.Y. Vardi (Eds.): FORTE 2002, LNCS 2529, pp. 260–275, 2002.

Part A: A C program

```
#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;
    }
  }
```

Part B: Minimized LTS

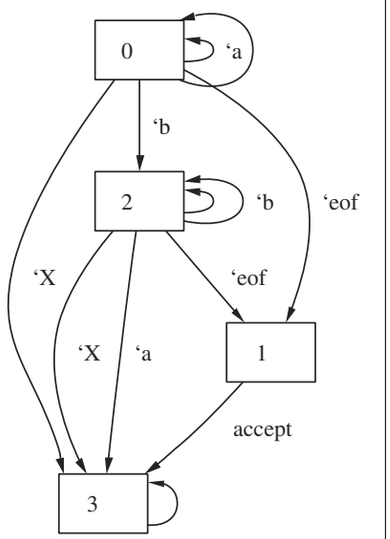

Fig. 1. A simple C program to recognize a^*b^* and its minimized LTS

by the observation that a process in a concurrent program has a finite-state synchronization skeleton with details of sequential computation filled in. In particular, early work on model checking [8] was geared towards reasoning about these finite-state synchronization skeletons.

Reverse-engineering the finite state machine that forms the synchronization skeleton of a process involves appropriately abstracting the sequential computation section and labeling the events of interest in the extracted machine. Building such models is typically done by hand, as in [14]. Lately, there have been a number of tools built to make this process easier [3,16,21,26,31,36,37]. In this paper we report on our experience in developing C Wolf (a model-extraction toolset), and our experience in using it to analyze a version of the GNU i-protocol and the Unix ftp daemon. C Wolf works with C programs and provides facilities for a user to abstract integer variables of a program, in a semantically sound way. Given user-specified information on the observable actions of a program, C Wolf will generate an abstracted labeled transition system (LTS), which can then be passed to the Concurrency Workbench of New Century [34] for further analysis. For instance, the C program given in Figure 1(A), when abstracted appropriately, yields the LTS in Figure 1(B). We will use this program as a running example through the rest of the paper.

The main contributions of our work are:

- Design and implementation of the C Wolf toolset, which allows a user to abstract C programs and to observe events of interest to the user. In particular, C Wolf offers a number of optimizations to reduce the size of the system generated with respect to trace and observational equivalence. The generated system can be minimized further, if necessary, and analyzed using the Concurrency Workbench.
- Case studies of usage of the toolset in abstracting source code, in C, of the ftp daemon and GNU i-protocol. The case studies illustrate (a) that C Wolf is practical and (b) that simple, though not necessarily complete, observational equivalence-preserving transformations have the greatest effect in reducing the size of the models generated.

Related Work: There has been a great deal of interest lately in software model-checking. Some of the tools developed to date use abstraction while others use a form of run-time monitoring. They can be further divided into several categories, as described below.

Data abstraction: Apart from our work (which falls into this class), Bandera[16, 10] is a tool that uses user-defined data abstraction to generate models from Java programs. Bandera has a generic scheme for users to express abstractions, and then apply a particular abstraction to a specific program. Furthermore, Bandera employs software-slicing techniques to remove parts of the program that are irrelevant to establishing properties of interest, which are themselves expressed in linear temporal logic. Finally, Bandera can generate models for several reasoning systems including SPIN [24] and SMV [32,17].

Predicate abstraction: The predicate abstraction approach collectively abstracts all program variables by replacing them with a set of boolean variables. The SLAM project[1,2,3] automatically generates and refines sets of predicates based upon a property (expressed as an automaton) that the user wishes to check. Given a set of predicates, each program point can be characterized by the subset of chosen predicates (or their negations) that definitely hold at that point. Those program points where such a characterization is not possible lead to non-deterministic choice points. The result is a non-deterministic schema over boolean variables, one for each predicate of interest, which can be represented as a BDD. The SLAM project has several other tools to analyze/model-check these boolean programs. To date, the SLAM group has used this tool to identify bugs in device drivers for Windows NT.

Metacompilation: In this approach, the compiler for a programming language is extended with a user-customizable language, or table, which can be used to specify the model to be generated in the input language of a model checker. Two tools that belong to this category are FeaVer/AX[26,27] and xgcc[7,18,31]. In both of these cases it is up to the user to generate the models that are believed to be an appropriate abstraction of a program.

Specification-less approach: In this approach, specific properties of concurrent programs, such as deadlock, are targeted. To show these specific properties slicing is typically used. This approach has been used in Java PathFinder [37,23,5], and JCAT/YAV[13,28].

Runtime monitoring: This approach does not use abstraction at all. The approach is based on instrumenting a program, for a specific property of interest, with additional code. The instrumented code, when run, will monitor the program execution such that if an erroneous state is reached then appropriate action, such as stopping gracefully or logging data, can be carried out. This approach has been used by Godefroid in Verisoft[20,21,22] and by Stoller [36].

Road map of the rest of the paper: In Section 2 we will provide an overview of our system, in Section 3 the language for conveying abstraction and label maps to the toolset is discussed, in Section 4 we will provide details of the techniques we use for model-generation, in Section 5 we discuss optimization, and finally in Section 6 we will present the result of our case studies. We then conclude with a discussion of our contributions and plans for future work.

2 Overview

C Wolf generates a labeled transition system from (a) a C program, (b) an abstraction mapping and (c) a list of observables. This process is performed by a small set of components, each of which carries out a specific task. The main components of the system and their functionality are as follows (see Figure 2 for a bird's eye view of the system):

cwcc — A compiler front end for C, which translates an input program to C Wolf intermediate (CWI) format. `cwcc` is based on the ckit [30] front end.
cwld — A linker to combine several files in CWI format.
cwmb — The model-generator component. It takes as input (1) a program in CWI format, (2) an *abstraction map* describing user-defined abstractions, and (3) a *label map* describing what events need to be made observable. The output is a file containing a labeled transition system, which can be loaded into the Concurrency Workbench for further processing.

2.1 Input Restrictions

While the low-level features of C make it useful for systems software development, these same features complicate task of verification. Pointers allow direct manipulation of a program's memory and the heap space, and allows for aliasing between program variables. We deal only with cases where the aliasing involves stack-allocated variables and where the relationship can be determined at model-generation time. Other forms of aliasing are not detected at this point and are all abstracted to a point domain. We follow a similar strategy with function pointers.

A second potential problem is recursion. We do not reject a recursive program a priori under the assumption that the abstraction employed could lead to a finite state model. However, we do aid the user by optimizing all tail calls.

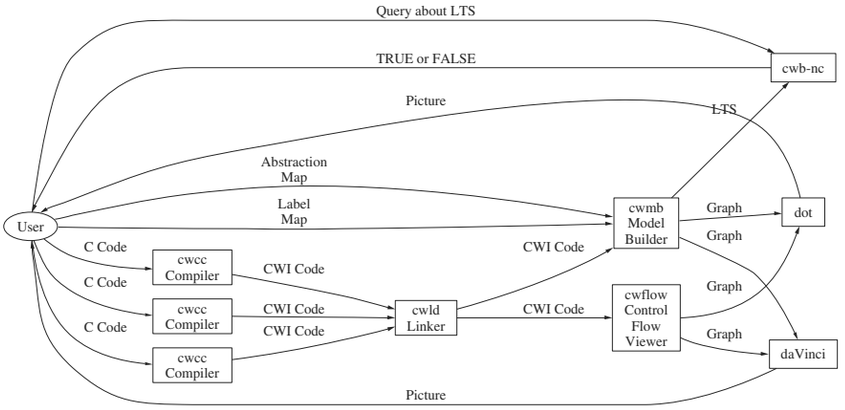

Fig. 2. System architecture at the executable-component level.

Our model does not directly include concurrency, so the use of threads is not allowed. However, models for a process of a distributed system can be computed using our toolset, followed by their parallel composition in the Concurrency Workbench. Our design decision follows the design of C, where concurrency primitives are not part of the language but are part of a library. Thus, it is the user's responsibility to model the communication mechanism directly in the Concurrency Workbench.

Finally, `signal` and `longjmp` are not currently supported since they interrupt the flow of control.

3 Abstraction and Observable Events

`cwmb`, the C Wolf model builder, accepts a CWI file as input, and two other additional 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 approach to abstractions focuses exclusively on integer variables. The reason for this is that in embedded systems and communications protocols, the variables that embody the high-level state of the system are almost always encoded as 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 1.

Table 1. The set of available abstractions.

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)
`bool`	Values are `true`, `false`, `maybe`

The simplest abstraction is `top`. Applying `top` to a variable `x` abstracts it away entirely. No information is maintained about the state of `x`.

The parameterized abstraction `mod` maintains a set of remainders under a divisor k. It's main use is for array indices. For example, consider a variable `x` whose abstract type is `mod(4)`. If it has an abstract value of $\{2, 3\}$, then we can infer that the value of `x` is in the set $\{2 + 4 \cdot i \mid i \geq 0\} \cup \{3 + 4 \cdot i \mid i \geq 0\}$.

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 typically sees 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 by using a bit-vector to track whether `x`'s value is less than 3, equal to 3, 4, or 5, or greater than 5. For example, 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 a huge number of states. Their main 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. The bounds are kept as tight as possible, which can cause state explosion in many cases, such as when a variable is used as a counter for an iterative loop.

The `int` abstraction tells the system to compute a precise value where possible, and to $\top$ otherwise. 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 used as a directive indicating that the abstract type of the variable should be dynamic. Whenever a variable of type `free` is assigned an abstract value then

fsm.am

```
file "fsm.c" {
  fun main () : top {
    var state : part(0,1,2);
    var ch : part(0,  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;
```

Fig. 3. Abstraction and label maps for the program of Figure 1(A).

no type conversion takes place. The `free` abstraction is used to incorporate important pointer variables into the extracted model.

`array` declares an array of elements sharing the same abstraction. The model builder must be able to infer the size of the array, otherwise, the entire array contents are abstracted to $\top$. Since each element is explicitly represented, `array` must be used judiciously to avoid explosions in the size of the model.

The Abstraction Map. The mapping of abstractions onto variables is specified by an *abstraction map* file. Without delving into the details of the syntax, there are three essential declarations. The first is the `file` declaration, which associates a source file name with a scope. All items declared within the scope are assumed to originate from the (given) source file. The second is the `fun` declaration, which names a function and defines a scope in which the abstractions to be applied to the function's local variables are specified. The final declaration is of the form `var` v `:` α, which declares the variable named v in the current scope to use abstraction α. Variables which aren't declared in the abstraction map receive the `top` abstraction by default.

Consider the program given in Figure 1(A). An abstraction map for this function is shown in Figure 3 as the file `fsm.am`. As can be seen from the map, the only important values for the variable `state` are 0, 1, and 2. Similarly, the values of importance for the character variable `ch` are EOF with ASCII code -1, newline with ASCII code 10, character `a` with ASCII code 97, and character `b` with code 98.

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 label map file associates labels with *events*. An event 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 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.

Table 2. Result of combining abstractions in a binary operator.

Type 1	**Type 2**	**Result Type**
`top`	Any	`top`
`minmax`	Mod k	`minmax`
`minmax`	`Interval`	`minmax`
`minmax`	`int`	`minmax`
`mod`(k_1)	`mod`(k_2)	`mod` (lcd $\{k_1, k_2\}$)
`mod`(k)	`part`	`mod`(k)
`mod`(k)	`int`	`mod`(k)
`part` $\mathcal{I}_{S_1}$	`part` $\mathcal{I}_{S_2}$	Coarser of S_1, S_2
`part` $\mathcal{I}_S$	`int`	`part` $\mathcal{I}_S$

– to generate a label every time a shared variable is read/written (with different labels for read and write).

The file `fsm.lm` specified in Figure 3 labels transitions based on the character read into the variable `ch`, which can be an `a`, `b`, `EOF` or anything else (called `X`). Note that in applying labels, the first `watch` condition that evaluates to true is used. Thus, overlapping conditions for `X` and events `a` and `b` pose no problem.

4 Model Generation

Given a program file, an abstraction map and a label map, we will now detail the steps involved in building an abstract LTS. First, `cwcc` translates the each program module to CWI code. CWI is a high-level intermediate code in which statements are organized into basic blocks, and each statement is either an assignment of a pure expression to a single variable, or a control-flow operation (a goto, conditional goto, or function call/return). Second, `cwld` is used to link the CWI modules into a single file. Third, `cwmb` constructs a model from the CWI code, directed by user-supplied abstraction and label mappings.

The model generation process can be thought of as an abstract evaluation scheme where the intermediate code is interpreted using abstract values. In particular, the evaluation will have to be carried out with respect to an environment which binds abstract values to variables. Given that model generation is based on the notion of abstract interpretation [11], we will provide details for the following crucial questions:

– How are different abstractions combined?
– Is the abstraction sound?
– How are environments represented?

The need for combining abstractions comes from the need to abstractly evaluate expressions of the from x *op* y, where x and y are abstracted using different

abstractions. Given that most of our abstractions target integer variables, we have a common concrete domain (in the terminology of abstract interpretation) that can be used to convert one abstraction to another with minimal loss of information. Table 2 depicts the various conversions that are possible. The main point to note is that since the abstractions are all based on integers it is possible to convert from one to another easily.

In [15] we have shown that all abstract operations are semantically sound with respect to the original operation in the sense of Cousot's abstract interpretation.[1] In particular, we have shown soundness with respect to 32-bit 2's complement representation for integers. Based on the soundness of arithmetic and logical operations we have also shown that symbolic evaluation of each CWI instruction is semantically sound.

Given that each instruction is abstracted soundly, it is easy to see that sequences of instructions are also abstracted soundly. In applying the symbolic evaluator to a program, we consider a basic block at a time. It is these basic blocks that form the foundations of the states of our labeled transition system. Abstract evaluation of the branches and conditional branches at the end of a basic block determine the successor states of that basic block. Finally, assignment to a variable targeted by a `watch` can also lead to a choice point.

As the states are generated, an edge is inserted between the current state and each of its successors. Edges are labeled by an observable event or by τ, the symbol for non-observable event in Milner's CCS [33] and the Concurrency Workbench.

Management of the environments, which include information about the variable bindings, has been implemented in C Wolf using functional mappings, which are then hashed. This allows two environments that are different from each other by a single binding to share the rest of the environment. Furthermore, we impose a variable ordering on the representation of the bindings; this makes the sharing easier and the comparison of two environments for equality easier. Finally, it should be noted that the generation of the model might not terminate if the abstraction has not be defined appropriately. This strategy is very similar to the generation of the labeled transition system in the Concurrency Workbench.

The last step in model generation is to output the results. The output is always a file; the format can be human-readable text, a graph (renderable by `dot` [29], `daVinci` [19], or `VCG` [35]), or a Concurrency Workbench automaton.

5 Optimizations

The result of applying the abstraction file and the label map from Figure 3 to the program of Figure 1(A) yields a labeled transition system with 52 states and 62 transitions (not shown due to space restrictions). The LTS contains many τ

[1] We have shown that for all abstract values a and b the relation $\gamma(\oplus(a,b)) \supseteq \{+(a',b') | a' \in \gamma(a),\ b' \in \gamma(b)\}$ holds, where α and γ are the abstraction and the concretization functions, and $+$ is a typical operation with $\oplus$ being its abstract version.

(silent) transitions, most of which have no impact on typical properties of interest. Several optimization passes are provided to eliminates these transitions. 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 be deleted without affecting observational equivalence. The criteria for selecting such states is simple: states which have only one outgoing τ 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 weakly bisimilar to the original system (observationally equivalent 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 must 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 set of available optimizations include *strong bisimulation* and *observational equivalence* [33]. These notions are not strictly necessary as they are available in the Concurrency Workbench. The model displayed in Figure 1(B) is the result of applying observational equivalence minimization, which reduced the number of states from 52 to 4.

Finally, `cwmb` also supports *trace equivalence*, which results in a model that concisely represents all possible execution traces. Such a model is not very useful for most verification problems as it does not even preserve deadlock behavior.

6 Case Studies

We used C Wolf to extract models from the GNU i-protocol, which is part of the Taylor UUCP package. In [14,25] the sources of version 1.04 were used as an example to test the effectiveness of various model checkers. The models used in these efforts were constructed by hand from the source files. Can the model be automatically generated from the sources? We will now discuss to what extent this is possible.

The i-Protocol is a sliding window protocol, which maintains a ring buffer of size $2 \cdot N$ to implement a window of size N. At any time, the current window of messages sent but not yet acknowledged by the receiver is marked by two indices i and $j \leq (i + N) \ mod \ 2 \cdot N$. The receiver process maintains an index k which is the index to the last message that was acknowledged. The livelock which was observed had to do with the fact that the receiver, when sent a packet outside the window $k \ldots (k + N) \ mod \ 2 \cdot N$ ignores it without sending a NAK. Consequently, when an acknowledgment sent by the receiver is lost in transit, the sender could resend a message multiple times all of which are ignored by the receiver.

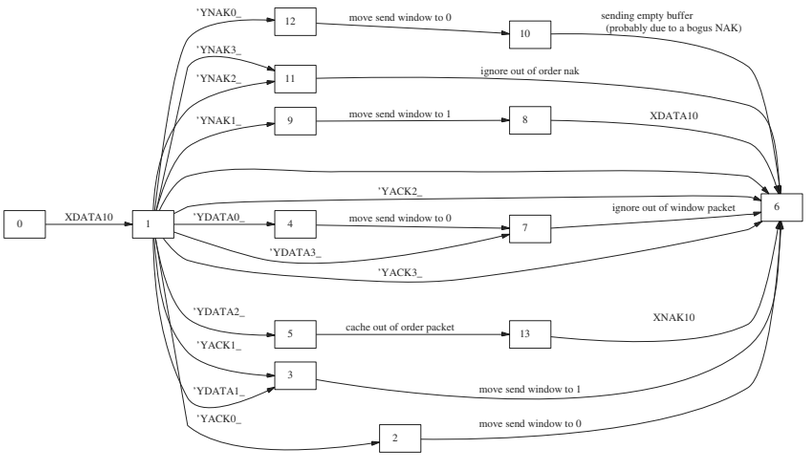

Fig. 4. Behavior of I-protocol when sending 1 data packet, and receiving a reply.

To carry out our experiments we extracted the code for the sender process from the sources, and replaced the other functionalities by a test harness. The i-protocol code was then 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 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 packets (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 were never invoked.

The test harness code does two tasks. 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 model generated is shown in Figure 4 which shows the discarding of NAKs observed by others. We ran the model extractor `cwmb` with different optimizations turned on. In table 3, we have listed the optimization performed, the CPU time, the total system time, and the number of states and transitions generated. Before the optimizations have been performed, the environments are thrown away; consequently, the number of bindings are only shown when no optimizations are done. Finally, we also show the maximum heap space that

Table 3. Resources required to model the i-protocol

Test	Opt.	CPU	Time	States	Transitions	Bindings	Memory
ip	None	3.77	4.09	7746	7746	10954	189725
ip	Peep	4.87	5.25	61	108		189729
ip	Bisim	4.88	5.36	16	63		189733
ip	ObsEq	4.86	5.16	14	63		189737
ip	OnTheFly	4.17	4.53	2883	2883		189729
ip	All	4.57	4.99	14	63		189741

SML/NJ (used to implement C Wolf) allocates across all the processes, which seems to remain a constant.

The 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 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 [38].

We applied C Wolf to the `ftpd` sources with no abstractions, and with instructions to label all system calls. This approach has the advantage that the result is a minimized NFA, which is much easier to deal with than an NPDA. Figure 5 shows the NFA extracted 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. A second advantage of using C Wolf is that the values of variables can easily be incorporated into the model, resulting in a more precise model than is possible in [38], which essentially applies a point abstraction to all variables in the monitored program.

7 Conclusions and Future Work

We have constructed the C Wolf toolset to extract finite state models from C programs in order to analyze their behavior. We have been able to apply C Wolfto several small examples and to two large programs. We have also shown that our methods are effective. In particular, we were surprised to see that the `part` abstraction was a very effective abstraction. Furthermore, we were also surprised

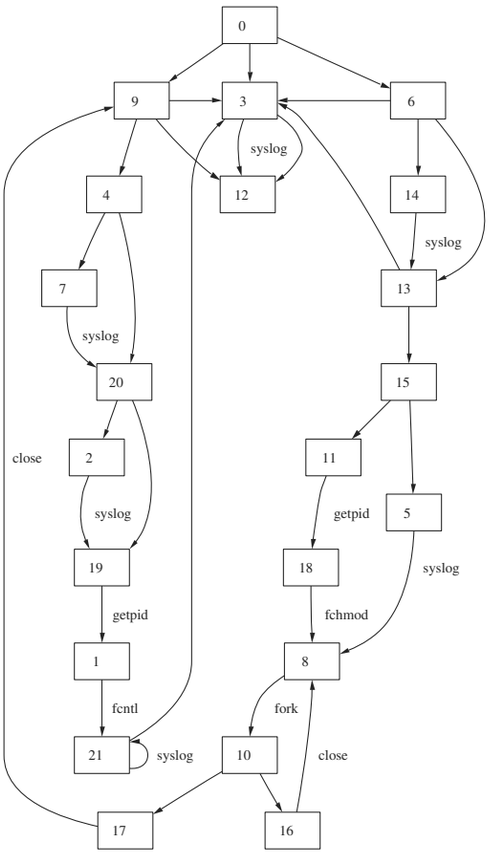

Fig. 5. System-call structure of the FTP daemon.

by the fact that simple peephole optimization, which preserved observational equivalence, did cut down the size of the models created by more than 75%.

Once a model is extracted it can be reasoned about in the Concurrency Workbench. It should, however, be noted that the abstraction produces an upper approximation of the program's behavior. Consequently, only universally path-quantified CTL* formulae which are invariant under stuttering are preserved by the extraction process – a characteristic shared by all relevant work on software model-checking.

Clearly, C Wolf can be improved upon in a number of ways, mostly in the kinds of abstraction techniques that are available to the user. New abstractions for dealing with arrays are under development, and we are experimenting with symbolic abstractions which capture relationships between the variables of a program. An additional feature being added to C Wolf is the option of generating

a pushdown automaton model for use with the XMC model checker [12], enabling arbitrarily recursive programs to be model-checked.

Acknowledgments. We are grateful to Rance Cleaveland for fruitful discussions and support. We would also like to thank the developers of Standard ML of New Jersey, ckit, dot, daVinci, and VCG.

References

1. 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.
2. 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.
3. 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.
4. Ahmed Bouajjani, Rachid Echahed, and Peter Habermehl. Verifying infinite state processes with sequential and parallel composition. In *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*, pages 95–106, San Francisco, California, January 22–25, 1995. ACM Press.
5. Guillaume Brat, Klaus Havelund, SeungJoon Park, and William Visser. Java pathfinder: Second generation of a java model checker, July 2000.
6. O. Burkart and B. Steffen. Model-checking the full-modal mu-calculus for infinite sequential processes. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *Automata, Languages and Programming (ICALP '97)*, volume 1256 of *Lecture Notes in Computer Science*, pages 419–429, Bologna, Italy, July 1997. Springer-Verlag. Full version to appear in *Theoretical Computer Science*.
7. 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.
8. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.
9. E. M. Clarke and J. M. Wing. Formal methods : state of the art and future directio ns. *ACM Computing Surveys*, 28(4):626–643, December 1996.
10. 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.
11. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

12. C.R. Ramakrishnan, I.V. Ramakrishnan, S. A. Smolka, Y. Dong, X. Du, A. Roychoudhury, and V.N. Venkatakrishnan. XMC: A logic-programming-based verification toolset. In *Computer Aided Verification (CAV 2000)*, Chicago, Illinois, June 2000.
13. 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.
14. 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.
15. Daniel C. DuVarney. *Abstraction-Based Generation of Finite State Models from C Programs*. PhD thesis, North Carolina State University, 2002.
16. 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.
17. E. M. Clarke, K. L. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic model checking. In *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 419–422. Springer Verlag, July/August 1996.
18. 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.
19. 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.
20. 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.
21. 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.
22. 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.
23. K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4), april 1998.
24. G. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

25. 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.
26. 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.
27. 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.
28. Radu Iosif. Formal verification applied to java concurrent software. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 707–709, June 2000.
29. 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/`.
30. 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`.
31. 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.
32. McMillan, K. L. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell Massachusetts, 1993.
33. R. Milner. *Communication and Concurrency*. PHI Series in Computer Science. Prentice Hall, 1989.
34. 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.
35. G. Sander. Vcg – visualization of compiler graphs. Technical Report A01-95, Universität des Saarlande, FB 14 Informatik, 1995.
36. 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.
37. 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.
38. 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.