

Model Checking Abstract Components within Concrete Software Environments

Tonglaga Bao and Mike Jones

Computer Science Department, Brigham Young University,
Provo, UT
{tonga, jones}@cs.byu.edu

Abstract. In order to model check a software component which is not a standalone program, we need a model of the software which completes the program. This is typically done by abstracting the surrounding software and the environment in which the entire system will be executed. However, abstracting the surrounding software artifact is difficult when the surrounding software is a large, complex artifact. In this paper, we take a new approach to the problem by abstracting the software component under test and leaving the surrounding software concrete. We compare three abstraction schemes, bitstate hashing and two schemes based on predicate abstraction, which can be used to abstract the components. We show how to generate the mixed abstract-concrete model automatically from a C program and verify the model using the SPIN model checker. We give verification results for three C programs each consisting of hundreds or thousands of lines of code, pointers, data structures and calls to library functions. Compared to the predicate abstraction schemes, bitstate hashing was uniformly more efficient in both error discovery and exhaustive state enumeration. The component abstraction results in faster error discovery than normal code execution when pruning during state enumeration avoids repeated execution of instructions on the same data.

1 Introduction

One way to manage the complexity of large software engineering projects is to factor the problem into cooperating components. Each component must then be written to implement that component's functionality in the context of other components. The modular verification problem is the problem of showing that each component behaves correctly in the execution environment created by the other components.

We focus on modular formal verification when no formal model of the surrounding software exists. Such a formal model would most likely be missing because it is too expensive to generate. This can happen, for example, when the implementations of some components deviate from their specifications but the nature of those deviations has not been precisely characterized. Or, there may simply be no formal model of the entire system. In these situations, the key

verification question is: does the component under test satisfy a set of properties in the context provided by the other components even though there is no formal model of the other components?

Modular verification without a formal specification of the surrounding software is important to engineers who must implement and verify components for existing software. This problem can occur when existing software is upgraded or when software development organizations decide to use formal verification after having developed a significant amount of software. In these and similar cases, a technique is needed to verify formal properties of new components in the context of existing software.

Formal approaches to modular software verification have been proposed for quite some time. However, in every case, the component is left concrete while the environment is abstracted. This poses two problems. First, the component itself may be too complex to admit formal analysis without abstraction. Second, abstraction of the software environment requires a formal model of the software environment. This means that the surrounding software must be converted to a formal model during or before abstraction. For large software environment, this is prohibitively expensive.

Fortunately, a precise model does exist for every executable software artifact. This model, though difficult to describe analytically, is simply the behavior of the software on the computational platform on which it was intended to be executed. The idea of defining semantics through execution is not new and lies at the foundation of advances in explicit state-space representation model checking [12, 9, 6].

Similarly, discovering the precise definition of software meaning through execution lies at the core of our approach. The main difficulty is creating an efficient interface between the abstract component and unabstracted surrounding software. The interface must be defined so that execution can be quickly passed to concrete software across the abstraction boundary. For example, overapproximation schemes are unsuitable because a single abstract state may represent thousands of concrete states—many of which are infeasible. Each of these concrete states would need to be passed and executed by the surrounding software.

In this paper, we present a new approach to component verification in which the component is abstracted and the environment is left concrete. We evaluate three abstraction techniques which are compatible with this approach to modular verification.

We have implemented our idea in the SPIN model checker [6]. Given a program written in C, we translate it to a model with PROMELA proctypes and C functions using an extension of the CIL compiler [10]. Abstract components are modeled by PROMELA proctypes and the surrounding software is left as C code with little modification.

Previously, Holzman and Joshi extended PROMELA to have better communication with programs written in C using the `c_code` and `c_track` mechanism. The C code enclosed inside a `c_code` block is executed directly like a normal C

program. The `c_track` block encloses variables from the C program which will be tracked. Tracked variables are included in the PROMELA state vector. `c_track` declarations can be marked as “matched” or “unmatched” to indicate that the variables will be stored into the hash table or not.

Inside a component, we translate branching instruction to PROMELA in order to expose the program control flow to the model checker. We use `c_code` and `c_track` mechanism. The environment is enclosed in `c_code` block and the component is a mixed model with `c_code` instructions interspersed with PROMELA instructions. In the component, variables are strategically tracked and matched in order to support data abstraction. When execution reaches the component boundary, the concrete state in the state exploration queue or stack can be passed directly to the software environment.

We tried three potential abstraction scheme with the component. The first one is described in [11]. It is an under-approximated abstraction scheme. Refinement is achieved by checking the preciseness of the abstraction through weakest pre-conditions. The second one is described in [8], in which refinement is obtained by checking the value range of the variables. Third one is abstraction through supertrace.

Our approach can verify software components that interact with complex surrounding software. For example, the surrounding software might contain mathematics for which first order logic theorem provers, as used in predicate abstraction, can provide no useful information. This can happen even for relatively simple operations like multiplying two variables or for more complex operations like exponentiation or trigonometry. The environment might also contain pointers and references which are too difficult to track in a formal semantic model. Our model runs without problem in these cases since we simply execute the environment instead of reasoning about it.

It would seem natural, at this point, to just execute the component instead of reasoning about it as well. After all, executing the environment sidesteps range of thorny semantic issues.

However, the surrounding software just provides the environment in which to verify the component so simply it, with no attempt at analysis or verification, is sufficient. But the component is the target of the verification effort, so tools, such as abstraction, to improve the utility verification effort are warranted.

Experimental results show this algorithm is indeed be able to deal with complex surrounding software with complex control structures and suggest that our algorithm can be used as a complementary method of testing to discover error faster by covering the state space faster. It is a good way to deal with programs that are too complex to be reasoned about by traditional model checking techniques.

Our test result shows abstracting the component using supertrace outperforms the other two abstraction schemes in terms of verification speed and error discovery speed. It also outperforms concrete execution in terms of error discovery speed when the state space includes many copies of the same large region of states. In this case, concrete state exploration still needs to execute the already

visited states since it does not have a memory about what state is already visited. Abstraction, on the other hand, can jump out of the partial state space it already visited and start to explore new state space faster.

In the next section we survey closely related work in abstraction for explicit model checking and component-based verification. Section 3 contains an explicit model checking algorithm for exploring the state space of abstract components in concrete environments. Section 4 shows how we implement this approach in SPIN. Section 5 provides experimental results. We close with conclusions and ideas for future work in Section 6.

2 Related Work

In this section, we first discuss the prior work on which this work is built, then present related work in abstraction and environment generation.

One way to view this paper is an extension of Holzmann and Joshi’s work on model-driven software verification. In [7], Holzmann and Joshi describe a method for mixing C code with PROMELA models such that data abstractions can be performed on C variables. We build on their work by creating an abstraction model for components based on data abstractions which under-approximate the state space.

Our approach to modeling the software environment is fundamentally different than that used by Holzmann and Joshi. While Holzmann and Joshi use PROMELA instructions as the test harness for C code, we use code from the surrounding software as the test harness. Our approach is appropriate when a PROMELA model of the surrounding software is too expensive to either build or execute. Our approach also admits the use of PROMELA as a test harness for parts of the system—such as for input from users.

In order to simplify the translation between branching instructions to PROMELA in components, we used the CIL [10] compiler to compile C programs into a C intermediate language, and then translate the resulting C intermediate language into PROMELA. CIL can compile all valid C programs into a few core syntactic constructs. This simplifies translation while allowing us to handle a large subset of C.

2.1 Environment Generation

When verifying only part of a program, the problem of simulating the program environment can be split into two parts: generating the test harness and simulating the surrounding software. The test harness provides the inputs to the program. Most existing work in component verification combines these two problems together to provide an abstract environment to the program under test which simulates both the test harness and the surrounding software.

Bandera [4] divides a given java program into two parts: the unit under test and the environment. The component under test is verified concretely while the

environment is abstracted to provide the unit the necessary behaviors. The abstract environment is obtained through a specification written by the user or through the source code analysis. The drawbacks of this approach are: first, abstracting the environment is a time consuming and error prone process. Moreover, it is hard to avoid the semantic gap between the concrete environment and the abstract representation of it. Second, if the component under test is complex, then model checking it concretely might cause state space explosion.

Slam [2] is a software model checker developed by Ball et al. to verify Windows device drivers. A device driver is a program which communicates with the operating system kernel on behalf of a peripheral device such as a mouse or printer. The surrounding software for a device driver is the entire kernel, so device driver verification requires a model of the kernel in order to close the execution environment. Since the Windows kernel is a large, complex program with no formal specification, manually generating a formal model of the kernel is expensive and error-prone.

Instead, Ball et al. [1] generate kernel models via merging different abstractions of the kernel procedure. Slam selects a set of device drivers that utilize a specific kernel procedure. These drivers then used as a training set by linking each driver with this specific kernel procedure and executing it in Slam. Slam automatically generates Boolean abstractions for the kernel procedure with each device driver. The Boolean abstractions for the procedure can be extracted and merged to create a library of Boolean programs which are used to verify future device drivers which utilize that kernel procedure. In this work, we take a different approach. Instead of abstracting the kernel, we abstract the device driver and leave the kernel software concrete. In our approach, a device driver would be verified by abstracting the device driver then verifying the abstract model of it in the context of the actual Windows kernel. An external environment that generates IO requests to drive verification would also be needed.

2.2 Abstraction

We abstract the component under test during verification. Abstraction is a widely studied topic in software model checking. Abstraction methods can be split based on whether they over approximate and under approximate the reachable states of a program. SLAM, which was mentioned previously, uses predicate abstraction [2] to abstract the device drivers and corresponding procedures in the kernel. SLAM uses over approximation abstraction techniques.

In this work, we need an abstraction scheme which stores abstract states in the hash table but uses concrete states in the queue of states to be expanded because this simplifies passing states between component and environment.

We have investigated three abstraction schemes which have this property: under approximation using predicates which requires a theorem prover for refinement, under approximation using predicates which do not require a theorem prover for refinement and bitstate hashing. Results for component verification using each of these abstractions are given later.

Pasareanu et al. proposed an under approximation abstraction approach which uses predicates and a theorem prover to manage refinement [11]. They explore the concrete state space, push concrete states into the stack and store their corresponding abstract states into the hash table. Abstract states are generated by evaluating a set of predicates on variable values. The state vector includes one bit per predicate and each bit is set based on the predicate's truth value. Refinement is done by examining each transition relation in terms of the existing predicates. If the existing predicates do not imply the weakest pre-condition of the next state in terms of the current transition, then the abstraction is not precise and the weakest pre-condition is added into the existing predicate set. Otherwise, abstraction is precise and no refinement is necessary. Since the state exploration is driven only by reachable concrete states in the stack, the abstraction never introduces new behaviors to the system. The abstraction misses behaviors when two concrete states satisfy the same set of predicates but lead to different program states.

Kudra and Mercer hypothesized that under approximation with predicates could be made faster by eliminating the theorem prover in refinement checking [8]. Pasareanu posed the same hypothesis, but eliminated the theorem prover by always assuming that the abstraction is imprecise. Kudra and Mercer eliminated the theorem prover by picking predicates which can be evaluated without a first order logic theorem prover and tracking extra data required to test the validity of those predicates. For each abstract state, they store the minimum and maximum value of each variables. When the minimum and maximum value of a variable is different, then they know this abstract state corresponds to at least two concrete states and needs to be refined. Eventually, this method will explore all the concrete states. However, before exploring all the concrete states, it covers more of the state space in less time in order to find errors faster. For some programs, eliminating the theorem prover results in faster error discovery because states could be generated more quickly.

Bitstate hashing stores concrete states in the queue of states to be expanded, but represents visited states using a single bit (or small set of bits) in the hash table [5]. In bitstate hashing, the location for a state in the hash table is determined by applying a hash function directly to the entire state vector. The resulting value is used as an index into the hash table and the bit at that index is set to true to indicate that the state has been visited. This abstraction misses behaviors when two concrete states hash to the same value but result in different program behaviors.

For the purposes of this work, under approximating predicate abstraction and bit state hashing are the same process with the difference that predicates are used to hash states in predicate abstraction while a hash function is used to hash states in bit state hashing. In this sense, predicate abstraction is a semantically based abstraction in which well-chosen predicates differentiate concrete states based on their meaning while bit state hashing is purely a structural abstraction in which the meanings of data values are ignored by the hashing function.

3 Algorithm

In this section first we describe the state exploration algorithm, then we discuss how abstraction is done for components.

3.1 State Exploration

Figure 1 shows our state enumeration algorithm for verifying abstract components in the context of concrete software. We have omitted property checking in order to simplify the presentation. Safety property checking can be added.

Given a program *prog*, we call the procedure **init** in line 1. Φ stores the initial set of predicates in line 2. The initial set of predicates is all of the guards in the entire program. Φ_{new} stores the set of new predicates used to refine the abstraction after each iteration and is initialized to the empty set in line 3. We check the starting instruction of the program in line 6. If the starting instruction is in the environment, then we execute instructions in the environment until control returns to the component. An instruction is in the environment if the state generated by that instruction is in the environment. The environment is specified as a range of program counter values, so an instruction is in the environment if it generates a state with a program counter value which lies outside that range. The environment execution functions returns the first state which lies in the component.

Now that we have a start state which lies in the component, we push it on the stack at line 8 and begin verification by calling the **component** function in line 9. When using predicate abstraction with refinement, we repeatedly verify the entire program until no refinement is necessary, as shown in line 10.

The **component** function is a variation of explicit state exploration in which abstract states are stored in the hash table, concrete states are stored in the stack and transitions which leave the component are serially executed without storing states. We obtain the transition out of the current state in line 19. If that transition exits the component, then we execute instructions in the environment until an instruction returns control to the component at line 20. The next state is generated by applying the current transition to the current state, line 21, or in the **environment** function at line 28. State exploration then continues by pushing the next state into the stack in line 22.

In the **environment** function, when the next instruction is in the environment, we will simply execute it at line 28. When the next instruction returns control to the component, we return the first state which lies in the component at line 31.

3.2 Abstraction

The **abstract** function takes a concrete state *s* and returns an abstract state. *abs* denotes an abstract state represented by a bit vector. The algorithm requires an abstraction which stores concrete states in the stack and stores abstract states in the hash table. If abstract states are stored in the stack, then passing control

```

1  proc init(prog)
2     $\Phi := \text{Guards}(\textit{prog})$ 
3     $\Phi_{new} := \emptyset$ 
4    do
5       $\Phi := \Phi \cup \Phi_{new}$ 
6      if start_instr  $\in$  environment then
7        start_state = environment(start_instr, start_state)
8        push(start_state)
9        component()
10     while  $\Phi_{new} \not\subseteq \Phi$ 
11  end
12
13 proc component()
14   while size(stack)  $\neq$  0
15     cur_state = top(stack)
16      $\alpha = \text{abstract}(\textit{cur\_state})$ 
17     if ( $\alpha \notin$  hash table)
18       insert  $\alpha$  into hash table
19       cur_inst = transition(cur_state)
20       if (cur_inst  $\notin$  comp) next_state = environment (cur_inst, cur_state)
21       else next_state = cur_inst(cur_state)
22       push (next_state)
23     else pop(stack)
24  end
25
26 proc environment(inst, state)
27   do
28     next_state = inst(state)
29     inst = transition (next_state)
30   while (inst  $\in$  environment)
31   return (inst(next_state))
32 end

```

Fig. 1. State enumeration algorithm that combines under-approximation with concrete execution.

between the component and environment at lines 7 and 20 of Figure 1 would be more difficult because we would need to create a concrete state which represents the abstract current state.

<pre> 1 proc abstract(<i>s</i>) 2 foreach $\phi_i \in \Phi$ do 3 if $\phi_i(s)$ then $abs_i := 1$ 4 else $abs_i := 0$ 5 if (refinement check is not valid) 6 addNewPreds(Φ_{new}) 7 return <i>abs</i> 8 end </pre>	<pre> 1 proc abstract(<i>s</i>) 2 $abs = \text{hash}(s)$ 3 return <i>abs</i> 4 end </pre>
(a), (b)	(c)

Fig. 2. Three abstraction schemes which store concrete states in the stack and abstract states in the hash table and are compatible with our approach to component verification. (a), (b) Predicate abstraction with or without a theorem prover as in [11] and [8], the difference being that the precision check at line 6 is done with or without a theorem prover. (c) Bit state hashing [5].

Figure 2 shows the abstractions which we have investigated as part of our component model in this paper. The first pair of abstractions, (a) and (b), are shown together because they differ only in the manner in which refinement is checked at line 6. In both cases, refinement is checked by determining if the abstraction of the previous state implies the abstraction of the next state with substitutions made using assignments in the instruction between the states. A detailed description can be found in [11] In Pasareanu’s case, the validity of the implication is checked using an automatic theorem prover. In Kudra’s case, the validity of the implication can be checked by determining if the variable’s value falls within a certain range.

Figure 2(c) shows bitstate hashing interpreted as an abstraction function. This is included to clarify the relationship between bitstate hashing and predicate abstraction as used in our work. Bitstate hashing is an abstraction in which a hash function is used to compute the abstract state. The abstract state is not stored directly in the hash table, but is used as an address at which to set a bit indicating that a state with that hash code has been visited. Refinement is not possible using bitstate hashing so the predicate set Φ_{new} is not updated and the while loop in line 10 of Figure 1 is simply ignored. However, bistate hashing can be made more precise by re-running the algorithm with a different hash function.

Each of the three abstractions in Figure 2 under approximate the state space. Every abstract state corresponds to at least one concrete state since the **abstract** function is only applied to already existing concrete states. States can be missed when two concrete states have the same abstract representation and only one of

them is expanded. Like other under-approximation techniques, every error found using our algorithm is a feasible error, but finding no errors does not guarantee that the component is error free.

For predicate abstraction, both with and without a theorem proving support, our under-approximation scheme can not be refined to include all behaviors of the system since we ignore system behaviors in the environment and these parts can not be included in the refinement check. More specifically, substituting the right side of an assignment for the left side of the assignment when that variable appears in the abstraction predicates can not be done safely for sequences of transitions that pass through the environment. Multiple syntactic substitutions for the transitions in the environment can mask program behavior and cause the precision check to succeed when behaviors have been ignored.

Interestingly, this loss of information in the precision check is adjustable. When the component grows to include the whole system, it is no longer required to chain together transitions in the syntactic substitution for the precision check and the refinement process works as described in [11]. A detailed discussion of the properties of refinement for predicate abstraction in the context of our component modeling method can be found in [3].

4 Implementation

We have implemented the algorithm in SPIN model checker using CIL for pre-processing of C code. For this implementation, we have assumed that a function is the basic unit of a component or the environment. In other words, each function in a C program either belongs entirely to the component or belongs entirely in the environment. We group interesting functions together to be verified as a single component, and group everything else into the environment.

Each function in the component is translated into a proctype in SPIN to be verified. The functions in environment remain unchanged as C functions. The details of how a function is translated into a SPIN proctype and how the SPIN proctypes interacts with the functions in the environment are discussed below.

SPIN supports embedded C code by providing five different primitives identified by the following keywords: `c_code`, `c_track`, `c_decl`, `c_state`, and `c_expr`. Everything enclosed inside `c_code` block is compiled directly by GCC then executed and interpreted as one atomic PROMELA state. `c_track` specifies the C variables we want to track as part of the state vector. `c_track` can be used with the `Matched` or `UnMatched` keywords. `Matched` variables are stored both in the state stack and hash table, while `UnMatched` variables are only stored in the state stack. For example

```
c_track "&i" "sizeof(int)" "UnMatched"
```

indicates C variable i will be tracked but not matched, and

```
c_track "&i" "sizeof(int)" "Matched"
```

```

1  main() {
2    int i;
3    for (i = 0; i < 10; i++)
4      if (i == 4)
5        break;
6      else i = i * 2;
7  }

```

Fig. 3. A simple C program.

```

1  c_decl{ int i; char abs[predNum];}
2  c_track "&i" "sizeof(int)" "UnMatched"
3  c_track "&abs" "sizeof(abs)" "Matched"
4  proctype main() {
5    do
6      :: c_expr{i < 10} → c_code{i++; abstraction();}
7      if
8        :: c_expr{i == 4} → break;
9        :: else -i c_code{i = i * 2; abstraction();};
10     fi;
11   od;
12 }

13 c_code {
14   void abstraction(){
15     for ( i = 0; i < predNum; i ++ )
16       if (preds[i] == true)abs[i] = 1;
17       else abs[i] = 0;
18     }
19 };

```

Fig. 4. The promela code generated from the C code shown in figure 3

indicates i is both tracked and matched. A kind of forgetful data abstraction can be obtained by tracking a variable but not matching it [7].

If a C function is contained within the component, then we translate it into an equivalent PROMELA proctype by enclosing non-branching statements in `c_code` blocks and translated branching statements into PROMELA. We do this in three steps.

First, we use CIL compiler to translate C into the C intermediate language (CIL)[10]. The CIL compiler compiles a valid C program into a C program which has a reduced number of syntactic constructs. By translating from C to a syntactic subset of C using CIL, we obtain a C program with simpler syntax, which makes translation from C to PROMELA much easier. We have implemented an extension of the CIL compiler to translate the CIL language into PROMELA.

Next, we enclose each non-branching statement of the component in a `c_code` block, so that every statement of the component is treated as a single PROMELA transition.

Finally, although each statement of the component is enclosed by a `c_code` block, control statements are translated entirely into PROMELA. This allows SPIN to mimic the branching structure of the component during verification.

As an example, consider the C code in Figure 3 which is translated into the PROMELA code shown in Figure 4 assuming the use of a predicate abstraction. Abstraction through bit state hashing is simple as it does not require additional arrays for predicates or their Boolean values. In Figure 4, `predNum` gives the number of predicates, `abs[predNum]` is a vector that contains abstract states, and `preds[predNum]` contains the given predicates. The function `abstraction()` on line 14 computes the abstraction by evaluating the predicates then storing their evaluation in the `abs[i]` vector of bits.

Predicate abstraction in the component is achieved by tracking and matching a bit vector. We declare an array of bits, called `abs[]`, which are also marked as `Matched`. After every assignment statement or function call in the component, we insert a call to a C function named `abstraction()`. `abstraction()` checks the current set of predicates and sets the corresponding bit values in `abs[]`. We then store only the values of `abs[]` and ignore all other variables.

Abstraction through bit state hashing is achieved by using SPIN's built-in implementation of hashing.

The software environment is modeled by wrapping it in a single `c_code` block. This means that segments of the environment are executed as needed by SPIN based on the behavior of the instructions in the component.

The next issue in the implementation is managing function calls within and between components and the environment. When translating C into a mixed C-and-PROMELA model, the most difficult problem is enforcing execution order in the presence of function calls. Since SPIN is designed to run concurrent code, we need to do some work to force it to avoid inappropriate interleavings in otherwise sequential programs. On the other hand, modeling concurrent components is more difficult because functions in the environment must support multiple active invocations. For purely sequential components and environments, there are four cases to consider depending on the location of the caller and the callee.

Inside a component, when a proctype calls a proctype, channels are used to enforce the order of the execution. An example is given in Figure 5. In Figure 5, `proctype main` calls `proctype proc` in line 7, and waits for `proc` to return at line 8. `proctype proc` signals the end of execution by pushing 1 into the channel at line 26. This signals the main process that it may resume execution.

When the code in a proctype calls a function in the environment, we pass an integer pointer `rtnFunFlag` with the function call. The callee indicates its return by setting `rtnFunflag` to 1 at the end of the function, at which time the caller continues execution. This is illustrated by figure 5 in lines 2 to 6 and line 16.

The most difficult case is when a function in the environment calls a proctype in the component. Since the `c_code` block is designed to be executed without interruption, if there is a call to a proctype in the middle, we must break out of the `c_code` block and run the proctype using just straight C. In the code generated by the SPIN, we find that calling a proctype is translated into an `addproc` function. In line 13 of figure 5, we add an `addproc` function to invoke the corresponding proctype `proc`. We pass the return program counter, `pc` value and function ID to `proc` so that it knows where to jump back to after execution. Then the caller function will jump out of the `c_code` block as shown in line 14. Then the `proc` begins execution and jumps back to the right place depending on the arguments.

When two functions in the environment call each other, it will be handled in unmodified C code with little additional effort. One important thing to note is that when a series of environment functions call each other, may be one of them may in turn call a function in the component, which means we need to stop execution in the environment immediately and return to the component. In this case it is important to keep a stack of function names and labels so that each environment function knows where to jump back after the component function returns back to the environment.

5 Results

The implementation of the algorithm allows us to take C code and model check parts of it. The C code can include complex data structures with pointers, and calls to library functions.

We choose three models to illustrate the result. They are matrix multiplication, sorting algorithms, and a program that simulates the operating system's dynamic storage allocator. The Matrix Multiplication and Sorting algorithm implementations are downloaded from the Internet. The dynamic storage allocator is taken from an assignment in an undergraduate operating system class.

Matrix Multiplication is a program that takes two matrices from the user and returns the product of those two matrices. This is an interesting problem for our component model because the code contains much data and many predicates, which makes it a good candidate for the predicate abstraction scheme. We supply 1000 pairs of matrices to the program. Each matrix has a user-defined number of column and rows. We insert an `assert` function to check that the dimension of the column of the first matrix equals to the row dimension of the second matrix. The result of the verification is shown in table 1.

In table 1, the first column gives the different abstraction schemes we test. PA+TP indicates predicate abstraction with theorem prover, and PA+NTP indicates predicate abstraction without theorem prover. In the first row, `cLine` is the number of lines of code in the component, `eLine` is the number of lines of code in the environment. There are also several library function calls which we do not include in the line number count. Matrix Multiplication uses library calls like "`printf`" and "`assert`". `States` is the total number of states generated from

```

1  proctype main() {
2    c_code{fun(rtnFunFlag, -1);};
3    do
4    :: c_expr{*rtnFunFlag == 1} rightarrow break;
5    :: else → skip;
6    od;
7    run proc();
8    c ? 1;
9  }
10 c_code {
11   fun(int* rtnFunFlag, int callerLabel){
12     if (callerLabel) goto label;
13     addproc(1);
14     goto end;
15     label:
16     *rtnFunFlag = 1;
17     end: ;
18  }
19 proctype proc(chan c, int callerID, int callerLabel) {
20   c_code{
21     if (callerID){
22       funarray[callerID](callerLabel);
23       goto end;
24     }
25   }
26   c ! 1;
27   c_code { end: ; };
28 }

```

Fig. 5. Functions in C translated into PROMELA

the component, mem is amount of memory used to store the state space of the component, predicates is the total number of predicates used in the verification. Time is the total time needed to complete the verification without seeded errors, and eTime stands for total time to find seeded error. Match shows the number of states that are matched inside the hash table. We group several functions that do the main computation together to compose the component and leave the rest of the software as the environment. This model consists of total of 5 million states. Bit state hashing performs best in matrix multiplication. Both of the other two algorithms fail to explore the total state space or find errors in the given time and space limit.

Table 2 also contains results for the matrix multiplication model, but this time we decrease the size of the component and increase the size of the environment by 80 lines. All three algorithms run to completion for this model. Observe that bitstate hashing generates the least number of states while TA+NTP generates the most. That is because bitstate hashing only needs one iteration of the entire program, but the other two do a refinement on their abstractions and continue exploring the whole state space until the state space covers all concrete states. PA+TP is the slowest in both error discovery and generating the whole state space. That is because it has to call the theorem prover to decide which, if any new predicates are needed for the refinement.

Table 3 shows the results for verifying a C model called sorting. It consists of several different sort algorithms. They are selection sort, insertion sort, bubble sort, and quick sort. We pick selection sort as a component. The property we check is asserting a value is less than the value after it in a list after returning back from the sorting functions. As the above models, bitstate hashing again outperforms the other two abstraction schemes.

Table 1. Matrix Multiplication, All Times in Seconds, Memory in MB, INFI indicates the result is not known because either time or space limitation is reached

matrix	cLine	eLine	states	mem	predicates	time	eTime	match
bitstate	120	270	5.3M	744	0	21	0.21	14
PA+TP	120	270	INFI	INFI	INFI	INFI	INFI	INFI
PA+NTP	120	270	INFI	INFI	INFI	INFI	INFI	INFI

Table 2. Matrix Multiplication with smaller component, All Times in Seconds, Memory in MB

matrix	cLine	eLine	states	mem	predicates	time	eTime	match
bitstate	40	350	3718	2	0	0.01	0.001	0
PA+TP	40	350	12095	54	102	170	41	198
PA+NTP	40	350	192516	111	101	5.1	0.08	100

Table 3. Sorting Model, All Times in Seconds, Memory in MB

sorting	cLine	eLine	states	mem	predicates	time	eTime	match
bitstate	44	110	8226	86		0 0.73	0.2	0
PA+TP	44	110	INFI	INFI		INFI INFI	41	INFI
PA+NTP	40	110	491830	405		449 51	3.5	8604

Table 4. Malloc Model, All Times in Seconds, Memory in MB

malloc	cLine	eLine	matched	time
bitstate	100	2000	10	1.2
concrete	0	2100	0	169

Table 5. Malloc Model, All Times in Seconds, Memory in MB

malloc	cLine	eLine	states	matched	time
bitstate	400	1700	142352	8	5.3
concrete	0	2100	0	0	169

In all the above models, bitstate hashing is by far more efficient than the other abstraction techniques. In fact, in the previous models, concrete exploration will be even more effective than bitstate hashing. However, there are several advantages to explore and store abstract states instead of simply executing them concretely. One advantage is through abstraction, the state space is covered faster because previously visited regions of the state space can be avoided through duplicate state detection using the bittable.

In table[?], we have a larger model that simulates part of an operating system which allocates, reallocates, and frees blocks of memory. This code has a bigger and more complex environment compared with the other two models. In this model, the component is 100 lines of codes and the environment is 2000 lines of codes. Both the component and environment uses library calls like “malloc”, “realloc” etc. These libraries plus the environment make it difficult to model the code formally. By concretely executing them, we don’t need a formal model of them.

We add a loop to make part of the code executes repeatedly, and we put an assert function outside of the loop. The purpose of doing that is to see if bitstate hashing can find an error faster than concrete exploration by recognizing already visited states and going to another part of the state space. The experimental result shows that it takes concrete exploration 169 seconds to discover the error, but bitstate hashing find it in only 1.2 seconds. The reason for that is bitstate hashing is able to track the states. When it sees an already explored state, it

will backtrack and explore the other part of the state space. Table [?] shows a similar result. In this test, we increase the size of the component. Bitstate algorithm again finds error faster than concrete exploration.

6 Conclusion and Future Work

In this paper, we have presented a technique for component-based verification that supports abstraction of the component under test rather than the environment in which the component is embedded. The abstraction can be applied when the source code for, but not a model of, the surrounding software is available.

The main purpose of this approach to abstraction is to save space and time by verifying only the part of the program under test rather than reasoning about the entire program. This approach assumes that errors which occur outside of the component under test are irrelevant and can be ignored. The focus is on detecting errors which are located inside the component, but which may have been caused by behaviors outside the component. Similarly, errors detected in the context of a specific software environment say nothing about errors in the context of even a slightly different software environment. The salient assumption here is that errors within a specific environment are of more interest than errors that exist in a family of environments.

Experimental result shows that we can verify a C program with the SPIN model checker automatically with little change to the original software. This software also can run in complex environments and call any library function. We abstract the component under test. The experiments suggest that bitstate hashing is the most efficient abstraction for this approach to component verification. Abstraction based on predicates did not reduce the abstract state space enough to justify the additional time to interpret states using predicates. The experiments also demonstrate that errors can be found in abstracted components more quickly than errors can be found by simply executing the component. This happens when the model checker prunes the search during state enumeration. Abstraction of components finds errors more quickly than executing the component as is when the state space includes many copies of the same large region of states. In this case, concrete state exploration still needs to execute the already visited states since it does not have a memory about what state is already visited. Abstraction, on the other hand, can jump out of the partial state space it already visited and start to explore new state space faster. Of the three abstraction methods we used, bitstate hashing found errors in the least time, mostly because it does not need a refinement.

We have not yet investigated methods for extracting components from software. Instead, we have simply assumed that the component is given by a set of *pc* values. One avenue for future work is developing methods for extracting useful components from software based on a set of verification properties. Future work also includes investigating other abstraction schemes and extending the implementation to handle concurrent software.

References

1. Thomas Ball, Vladimir Levin, and Fei Xie. Automatic creation of environment models via training. In 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pages 93–107, Barcelona, Spain, 2004.
2. Thomas Ball and Sriram K. Rajamani. The slam project: debugging system software via static analysis. In 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pages 1–3, New York, NY, 2002.
3. Tonglaga Bao. Refinement for predicate abstraction in the context of abstract component model. In (Brigham Young University), 2007.
4. James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In 22nd International Conference on Software Engineering (ICSE), pages 439–448, Limerick, Ireland, 2000.
5. G. J. Holzmann. An analysis of bitstate hashing. In Proc. 15th Int. Conf on Protocol Specification, Testing, and Verification, pages 301–314, 1995.
6. Gerard J. Holzmann. The model checker SPIN. Software Engineering, 23(5):279–295, 1997.
7. Gerard J. Holzmann and Rajeev Joshi. Model-driven software verification. In 11th International SPIN Workshop on Model Checking of Software (SPIN), pages 76–91, Barcelona, Spain, 2004.
8. Dritan Kudra and Eric G. Mercer. Finding termination and time improvement in predicate abstraction with under-approximation and abstract matching. In (MS thesis, Brigham Young University), 2007.
9. Eric Mercer and Mike Jones. Model checking machine code with the gnu debugger. In 12th International SPIN Workshop on Model Checking of Software (SPIN), pages 251–265, San Francisco, CA, 2005.
10. George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of c programs. In Computational Complexity, pages 213–228, 2002.
11. Corina. S. Pasareanu, Radek Pelanek, and Willem Visser. Concrete model checking with abstract matching and refinement. In 17th International Conference on Computer Aided Verification (CAV), pages 52–66, Edinburgh, Scotland, 2005.
12. Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In 15th IEEE International Conference on Automated Software Engineering (ASE), page 3, Washington, DC, 2000.