2009-03-26

Verifying Abstract Components Within Concrete Software Environments

Tonglaga Bao
Brigham Young University - Provo

VERIFYING ABSTRACT COMPONENTS WITHIN CONCRETE SOFTWARE ENVIRONMENTS

by

Tonglaga Bao

A dissertation submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

Brigham Young University

August 2009

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a dissertation submitted by

Tonglaga Bao

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

Date	Mike Jones, Chair
Date	Eric Mercer
Date	Scott Woodfield
Date	Quinn Snell
Date	Dan Olson

BRIGHAM YOUNG UNIVERSITY

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

Date	Mike Jones Chair, Graduate Committee

Accepted for the Department

Date	Kent Seamons Graduate Coordinator

Accepted for the College

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

ABSTRACT

VERIFYING ABSTRACT COMPONENTS WITHIN CONCRETE SOFTWARE ENVIRONMENTS

Tonglaga Bao
Department of Computer Science
Doctor of Philosophy

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 problem is important for software engineers who need to deploy an existing component into a new environment. The model is typically generated by abstracting the surrounding software environment in which the component will be executed. However, abstracting the surrounding software is a difficult and error-prone task, particularly when the surrounding software is a complex software artifact which can not be easily abstracted. In this dissertation, we present a new approach to the problem by abstracting the software component under test and leaving the surrounding software concrete. We derive this abstract-concrete mixed model automatically for both sequential and concurrent C programs and verify them using the SPIN model checker. We give verification results for several components under test contained within complex software environments to demonstrate the strengths and weaknesses of our approach. We are able to find errors in components which were too complex for analysis by existing model checking techniques. We prove that this mixed abstract-concrete model can be bisimilar to the original complete software system using an abstraction refinement scheme. We then show how to generate test cases for the component under test using this abstraction refinement process.

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Mike Jones for funding my education and patiently guiding me throughout my research. I would like to thank Dr Eric Mercer and Dr Scott Woodfield for their support and help. I would also like to thank Dr Quinn Snell, Dr Dan Olson, and Dr Phillip Windley for their time and efforts put on my dissertation.

A special thanks goes to Neha Rungta for generously offering me comments and suggestions.

I also would like to thank my parents for their love, support and encouragement throughout my education.

Contents

Contents viii

List of Figures xi

List of Tables xiii

1 Introduction 1

1.1 Background 3

1.1.1 Model Checking 3

1.1.2 Abstraction 5

1.2 Overview 6

2 Model Checking Abstract Components within Concrete Software Environments 9

2.1 Introduction 10

2.2 Related Work 14

2.2.1 Environment Generation 15

2.2.2 Abstraction 17

2.3 Algorithm 19

2.3.1 State Exploration 19

2.3.2 Abstraction 20

2.4 Implementation 23

2.5 Results 29

2.6 Conclusion and Future Work 33

3 Model Checking Concurrent C Components in Concrete Environments 35

3.1 Introduction 35

3.2 Related work 39

3.2.1 Verification of concurrent components 39

3.2.2 Abstraction 41

3.3 Verification model 42

3.3.1 Component modeling 42

3.3.2 Concurrency modeling 44

3.3.3 Component abstraction 46

3.4 Implementation method 47

3.5 Experimental results 53

3.6 Conclusion and future work 56

4 Test Case Generation Using Model Checking for Software Components Deployed into New Environments 57

4.1 Introduction 59

4.2 Related Work 64

4.3 Computational Model 66

4.4 Algorithm 71

4.5 Theorems 74

4.5.1 Precision 74

4.5.2 Under-approximation 79

4.5.3 Termination and Property Preservation 80

4.6 Conclusion and Future Work 80

5 Conclusion and Future Work 82

Appendices 85

A Additional Theorems 85

B Pthread Library Function to PROMELA Translation 89

Bibliography 93

List of Figures

1.1 Model checking 4
1.2 Abstractions 5
1.3 Concretizations 6

2.1 State enumeration algorithm that combines under-approximation with concrete execution 21
2.2 Three abstraction schemes which store concrete states in the stack and abstract states in the hash table that are compatible with our approach to component verification. (a), (b) Predicate abstraction with or without a theorem prover as in [21] and [16], the difference being that the precision check at line 6 is done with or without a theorem prover. (c) Bit state hashing [11]. 22
2.3 A simple C program. 24
2.4 The PROMELA code generated from the C code shown in figure 2.3 . 25
2.5 Functions in C translated into PROMELA 28

3.1 State enumeration algorithm combining an abstract component with a concrete environment 45
3.2 An example of C to CIL 48
3.3 An example of C to CIL 50
3.4 C, CIL, and PROMELA version of creating a new thread 51
3.5 Argument passing 52

4.1 Test generation for a new environment using an abstract model of the component under test. 62
4.2 (a) Sample C program (b) Concrete system (c) Abstract system using predicates $i < 10$ and $j < 10$ (d) Mixed system 68
4.3 State enumeration algorithm that combines under-approximation with concrete execution. 73
4.4 Loss of precision in a simple C program 76
4.5 Loss of precision in a C program with a branch 77

B.1 Thread join 89
B.2 C and PROMELA mutex statements 90
B.3 C and CIL example of using conditional varialbes 91
B.4 PROMELA translation of conditional variables 92

List of Tables

2.1 Matrix multiplication, All times in seconds, memory in MB, INFI indicates the result is not known because either time or space limitation is reached, time limitation is 300s and space limitation is 1GB 30

2.2 Matrix multiplication with smaller component, all times in seconds, memory in MB 31

2.3 Sorting model, all times in seconds, memory in MB 31

2.4 Malloc model, all times in seconds, memory in MB 31

2.5 Malloc model, all times in seconds, memory in MB 31

3.1 Verifying concurrent software components in a concrete environment[a] 54

Chapter 1

Introduction

Software is increasingly built from existing components developed in multiple environments. Component reuse reduces development costs and shortens delivery time by reusing previously developed components. Verification of component-based software is, however, a difficult and challenging problem due to the complexity of both the components and the environment in which they are integrated. When the components consist of concurrent programs, verification is especially difficult due to the exponential number of interleavings which are possible between the concurrent components.

In this dissertation, we address the problem of verifying both sequential and concurrent components inside complex software environments. In this work, a component is defined as a reusable piece of code which can interact with other components or software. The environment is the software in which the component is deployed.

This verification problem is important to component based software engineers. Before adding a component to an application, engineers need to determine that the component works as expected in their application environments. This problem is also important to component testers who need to create test suites for a component under test in a specific environment. Verifying concurrent components is important as muti-core machines become more common.

One common way to address this problem is to abstract the environment and model check the component inside the abstract environment. (A brief introduction to model checking and program abstraction technique is discussed later in this chapter.) The abstraction hides the complexity of the environment and also provides different inputs to the component to invoke different behaviors of the component. One drawback of this approach is that abstracting the environment is often difficult, or even impossible, when no formal model exists for the environment. This happens, for example, when the developer or tester does not have access to the source code of the environment. Another drawback is that concretely exploring all possible behaviors of the component may be impossible when the component is complex.

Another common approach to component verification is model based testing. In this approach, both the component and its environment are modelled in an abstract model and test cases are generated based on this model. Then the abstract test cases are transformed into executable test cases and run against the concrete system. Since an abstract model is easier to analyze than the concrete system, test cases generated from the abstract model give better test suites than randomly generated test cases. Similar to the previous approach, the drawback of this approach is that modelling the system is impossible when no formal model exists for the environment.

In this dissertation, we verify components in a given environment by model checking abstract components inside the concrete software environment. We abstract the component under test and leave the rest of the software concrete. We start by running the entire program normally. When program control is in the environment, the program is executed as normal. When program control enters the component, a model checker monitors execution of the component and explores all non-deterministic behaviors caused by the abstraction or concurrency of the component. Our verification approach allows us to verify components inside any complex environment with or without access to the source code. It also allows us to verify complex components by applying abstraction to the component. Our approach is especially well suited to verify concurrent components running inside complex environments. This kind of system is difficult to verify using either testing or model checking alone. This technique can also be used with model based testing to generate test cases for complex environments. Our work is a novel integration of testing with model checking to model check only components under test while running the rest of the software concretely.

We implement the algorithm of model checking abstract components within concrete software environments in SPIN, which is an explicit state model checker that verifies protocols written in the PROMELA language. Since PROMELA protocols are eventually translated into C code before being verified, SPIN provides an easy interface between C and PROMELA. We utilize this feature in SPIN to automatically verify any C program that is either sequential or concurrent. Given a C program, we divide it into the component and environment. We translate the component into PROMELA and execute the environment as is. The experimental results demonstrate that we are able to model check programs which are otherwise impossible to model check due to the complexity of both the component and the environment.

In the rest of this chapter, model checking techniques and abstraction techniques are discussed.

1.1 Background

This section contains a discussion of core ideas in the dissertation. More complete introductions to model checking and abstraction in model checking can be found in [8]

1.1.1 Model Checking

Given a transition system and a specification, model checking is a technique to determine if the transition system satisfies, or models, the given specification. Compared with traditional validation techniques - simulation and testing - the main advantage of model checking is it can conduct an exhaustive exploration of all possible behav-

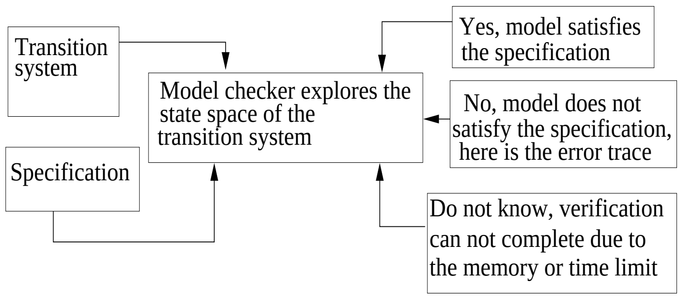

Transition system
Specification
Model checker explores the state space of the transition system
Yes, model satisfies the specification
No, model does not satisfy the specification, here is the error trace
Do not know, verification can not complete due to the memory or time limit

Figure 1.1: Model checking

iors of the systems while simulation and testing only explore some of the possible behaviors of the systems.

The model checker can automatically explore all the reachable states of a state space. This feature enables a model checker to find all bugs in a system or to prove that the system is error-free. When a bug is found, the model checker generates an error trace to pinpoint where the error is exactly located. Error traces help the tester locate and correct bugs more efficiently. Figure 1.1 summarizes the process of model checking.

Model checking has been successfully applied in hardware verification because hardware can, in many cases, be easily modeled as a finite state system. Model checking software is harder since software is generally modelled into a very large, or even infinite, state space. As Figure 1.1 illustrates, when the state space size is larger than the memory size, the model checker may terminate with no definite answer.

There is a growing interest in applying model checking to software. Abstraction is frequently used with software model checking to manage large state spaces. The next section introduces an abstraction method commonly used in software model checking.

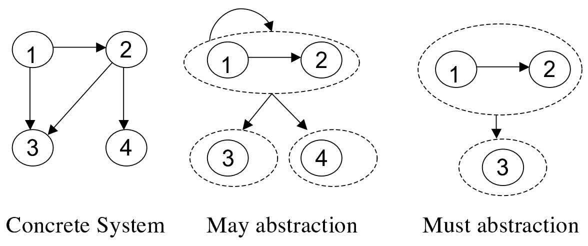


Figure 1.2: Abstractions

1.1.2 Abstraction

Imprecise abstraction techniques can often be divided into under-approximation and over-approximation. Under-approximation only considers part of the system's behaviors. This can cause false negatives – finding no errors does not guarantee that the system is free of errors. All the errors found, however, are feasible in the concrete system. Over-approximation, on the other hand, introduces new behaviors into the system and results in false positives. Not all the errors found are feasible in the concrete system, but finding no error in the abstract system proves the absence of errors in the concrete system. When a false-positive or false-negative report occurs, the abstracted system is refined with the help of a theorem prover to more accurately reflect the original concrete system.

Figure 1.2 explains under- and over-approximation. Dotted circles indicate abstract states. Suppose concrete state 1 and concrete state 2 comprise abstract state 1, concrete state 3 comprises abstract state 3 in May and Must abstraction, and concrete state 4 comprises abstract state 4 in May abstraction. The first graph gives the concrete system which consists of 4 concrete states and 4 concrete transitions. The middle graph is an abstract system obtained by using over-approximation. Over-approximation works like an existential quantifier over the concrete states. For a set of concrete states that belongs to one abstract state, if there exists a concrete state that has a transition to another concrete state, then the abstract state also has a corresponding transition to another abstract state. In this graph, for example, since

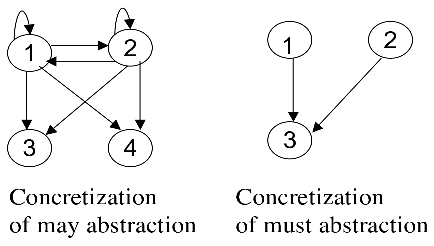

1
2
3
4
Concretization of may abstraction
1
2
3
Concretization of must abstraction

Figure 1.3: Concretizations

concrete state 1 has a transition to concrete state 3, the corresponding abstract state 1 also has a transition to the abstract state 3. The third graph gives an abstract system obtained by applying an under-approximation. Under-approximation works like a universal quantifier. All the concrete states that map to an abstract state must have a transition to another state in order to include the transition in the abstract system. For example, both concrete state 1 and concrete state 2 have a transition to concrete state 3, so the corresponding abstract state 1 also has a transition to abstract state 3. The other concrete transitions are all eliminated because they do not satisfy this requirement.

Figure 1.3 gives the concretization of both the over-approximation and under-approximation. Concretization means generating all possible concrete states and transitions for each corresponding abstract state and transition. We can see that the over-approximation represents more concrete behaviors than the original system while the under-approximation represents fewer concrete behaviors than the original system. In this research, we use various abstraction techniques in different parts of a component based system.

1.2 Overview

In this dissertation, we use model checking and an abstraction technique to verify a component in a concrete software environment. This approach is applied to three areas: model checking a sequential component in a sequential environment, model checking a concurrent component in a sequential environment, and generating tests for a component deployed into a sequential environment to determine whether or not the environment is compatible with an existing component. The component can be either sequential or concurrent in the third case. We discuss each of these cases in the chapters that follow.

Chapter 2 describes model checking sequential code in a sequential environment. The work in this chapter appears in [4]. Here, we assume both the component and the environment are sequential. We apply three kinds of under-approximation abstraction to the component and model check it in a concrete software environment. The source code for the environment need not be available. Fast error discovery is the main focus in this chapter.

Chapter 3 describes model checking concurrent code in a sequential environment. We assume that the component is concurrent and that the environment is sequential. We apply an under-approximation abstraction technique to the component and execute the environment as is. The environment source code is needed only when the environment invokes a component function, where we need to translate the function call into appropriate instruction to invoke the component. Fast state space exploration which consumes little memory and fast error discovery are the focus in this chapter. We can not guarantee full state space exploration in this method due to the abstraction employed.

Chapter 4 describes test generation for a sequential environment to create tests for determining whether or not the environment is compatible with an existing component. The work in this chapter appears in [3]. We propose to generate test cases for a component in its original environment. The generated test cases will be used in the new environment to check compatibility. In this chapter, we give theoretical results on how to get a full test coverage of the component in the original environment by reasoning about symbolically simulated environment. Here, we need the environment source code to analyze it symbolically. The generated test cases will be used in a new environment to detect errors.

Chapter 5 concludes this dissertation and discuss future work. The appendices present some additional proof and implementation details.

Chapter 2

Model Checking Abstract Components within Concrete Software Environments

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 and 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 is 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 and avoids repeated execution of instructions on the same data.

2.1 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 very 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 [23, 17, 12].

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, over-approximation 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 assume both the component and the environment are sequential. We evaluate three abstraction techniques which are compatible with this approach to modular verification.

We have implemented our idea in the SPIN model checker [12]. 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 [19]. 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 and 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 instructions to PROMELA in order to expose the program control flow to the model checker using the c_code and c_track, the environment is enclosed in a 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 schemes with the component. The first one is described in [21]. It is an under-approximated abstraction scheme. Refinement is achieved by checking the preciseness of the abstraction through weakest preconditions. The second one is described in [16], in which refinement is obtained by checking the value range of the variables. THe third one is abstraction through bitstate hashing.

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 a 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.

The main contribution of this paper is a model of components which supports data abstraction for C programs. Our model supports model checking inside unmodified, complex software environments and allows nested function calls between component and environment. Our state exploration algorithm is a simple modification of standard state enumeration algorithms. The value of this work is that it enlarges the class of C programs to which standard state enumeration algorithms can be applied. The abstraction method proposed in this work also leads to faster error discovery than normal code execution.

Experimental results show this algorithm is indeed 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 errors 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 results show 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 2.3 contains an explicit model checking algorithm for exploring the state space of abstract components in concrete environments. Section 2.4 shows how we implement this approach in SPIN. Section 2.5 provides experimental results. We close with conclusions and ideas for future work in Section 2.6.

2.2 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 [13], 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.

FeaVer [14] also verifies a C program or part of a C program. It uses a tool called Modex to extract a PROMELA model from the C source code. When verifying some specific functions of a program, Modex requires the user to provide a test harness in which to test these functions. Our approach has similar goals and is built on PROMELA commands, such as c_code and c_decl, which were used in Modex. However, our approach simplifies the process of defining a PROMELA model and test harness from a C program and our approach is designed to verify data operations of C programs rather than just the concurrent behavior of a threaded program. Our approach supports concurrency but the implementation does not. Implementation of support for concurrency is described in chapter 4.

Our approach is different. The software environment is part of our test harness, and we execute the component inside this specific environment instead of user defined environments. When the component under test calls a function, we check if the callee belongs to the component or belongs to the environment. If it belongs to the component, we translate it to a PROMELA model, otherwise we execute it as it is. Similarly, when the environment calls a function, we also check if the function belongs to the component or belongs to the environment. If it belongs to the component, we translate it to the PROMELA model, and otherwise, we execute it as it is. This is quite different from the Modex approach that simply treats the callee of the function under test as a blackbox.

In order to simplify the translation between branching instructions to PROMELA in components, we used the CIL [19] 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.

2.2.1 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 [6] 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 and 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 are 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.

2.2.2 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 [21]. 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 concretes 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 [16]. 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 variable. 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 [11]. 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 bitstate hashing are the same process with the difference being that predicates are used to hash states in predicate abstraction while a hash function is used to hash states in bitstate hashing. In this sense, predicate abstraction is a semantically based abstraction in which well-chosen predicates differentiate concrete states based on their meaning while bitstate hashing is purely a structural abstraction in which the meanings of data values are ignored by the hashing function.

2.3 Algorithm

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

2.3.1 State Exploration

Figure 3.1 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 component. 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 set of program counter values, so an instruction is in the environment if it generates a state with a program counter value which is in the set. The environment execution function 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 18. If that transition exits the component, then we execute instructions in the environment until an instruction returns control to the component at line 21. The next state is generated by applying the current transition to the current state, line 22, or in the **environment** function at line 28. State exploration then continues by pushing the next state into the stack in line 23.

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 33.

2.3.2 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 between the component and environment at lines 7 and 21 of Figure 3.1 would be more difficult because we would need to create a concrete state which represents the abstract current state.

Figure 2.2 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

```
proc init($prog$)
  $\Phi$ := Guards($prog$)
  $\Phi_{new} := \emptyset$
  do
    $\Phi := \Phi \cup \Phi_{new}$
    if $start\_instr \in$ environment then
      $start\_state$ = environment($start\_instr$, $start\_state$)
    push($start\_state$)
    component()
  while $\Phi_{new} \not\subseteq \Phi$
end

proc component()
  while size(stack) != 0
      $cur\_state$ = top(stack)
      $\alpha$ = abstract ($cur\_state$)
      if ($\alpha \notin$ hash table)
        insert $\alpha$ into hash table
        $cur\_inst$ = transition($cur\_state$)
        if ($cur\_inst \notin comp$) $next\_state$ = environment ($cur\_inst$, $cur\_state$)
        else $next\_state$ = $cur\_inst$($cur\_state$)
        push ($next\_state$)
      else pop(stack)
end

proc environment($inst$, $state$)
  do
    $next\_state$ = $inst$($state$)
    $inst$ = transition ($next\_state$)
  while ($inst \in$ environment)
  return ($inst$($next\_state$))
end
```

Figure 2.1: State enumeration algorithm that combines under-approximation with concrete execution

```
1 proc abstract($s$)
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($\Phi_{new}$)
7    return $abs$
8 end
```

(a), (b)

```
1 proc abstract($s$)
2    $abs$ = hash($s$)
3    return $abs$
4 end
```

(c)

Figure 2.2: Three abstraction schemes which store concrete states in the stack and abstract states in the hash table that are compatible with our approach to component verification. (a), (b) Predicate abstraction with or without a theorem prover as in [21] and [16], the difference being that the precision check at line 6 is done with or without a theorem prover. (c) Bit state hashing [11].

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 [21] 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 2.2(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 3.1 is simply ignored. However, bistate hashing can be made more precise by re-running the algorithm with a different hash function. Bitstate hashing can be used when a full verification is infeasible because of the memory limitations.

Each of the three abstractions in Figure 2.2 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 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, the refinement process works as described in [21]. A detailed discussion of the properties of refinement for predicate abstraction in the context of our component modeling method can be found in [3].

2.4 Implementation

We have implemented the algorithm in the SPIN model checker using CIL for preprocessing 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 environ-

```
main() {
  int i;
  for (i = 0; i < 10; i++)
    if (i == 4)
      break;
    else i = i * 2;
}
```

Figure 2.3: A simple C program.

ment. 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 the environment remain unchanged as C functions to be executed concretely. 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 a `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`”

indicates i is both tracked and matched. A kind of forgetful data abstraction can be obtained by tracking a variable but not matching it [13].

```
c_decl{ int i; char abs[predNum];}
c_track "&i" "sizeof(int)" "UnMatched"
c_track "&abs" "sizeof(abs)" "Matched"
proctype main() {
  do
  :: c_expr{i < 10} → c_code{i++; abstraction();}
     if
     :: c_expr{i == 4} → break;
     :: else → c_code{i = i * 2; abstraction();};
     fi;
  od;
}

c_code {
  void abstraction(){
     for ( i = 0; i < predNum; i + +)
        if (preds[i] == true)abs[i] = 1;
        else abs[i] = 0;
  }
};
```

Figure 2.4: The PROMELA code generated from the C code shown in figure 2.3

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 translating branching statements into PROMELA. We do this in three steps.

First, we use the CIL compiler to translate C into the C intermediate language (CIL) [19]. 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 expose the branching structure of the component during verification.

As an example, consider the C code in Figure 2.3 which is translated into the PROMELA code shown in Figure 2.4 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 2.4, `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 is 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 2.5. In Figure 2.5, `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 2.5 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 SPIN, we find that calling a proctype is translated into an `addproc` function. In line 13 of Figure 2.5, 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, it may be that 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

```
proctype main() {
  c_code{fun(rtnFunFlag, -1);};
  do
  :: c_expr{*rtnFunFlag == 1} → break;
  :: else → skip;
  od;
  run proc();
  c ? 1;
}
c_code {
  fun(int* rtnFunFlag, int callerLabel){
    if (callerLabel) goto label;
    addproc(1);
    goto end;
   label:
    *rtnFunFlag = 1;
    end: ;
}
proctype proc(chan c, int callerID, int callerLabel) {
  c_code{
    if (callerID){
      funarray[callerID](callerLabel);
      goto end;
    }
  }
  c ! 1;
  c_code { end: ; };
}
```

Figure 2.5: Functions in C translated into PROMELA

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.

2.5 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 2.1.

In Table 2.1, 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 the component, mem is the amount of memory (in Mbytes) 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 (in seconds) needed to complete the verification without seeded errors, and eTime stands for total

Table 2.1: Matrix multiplication, All times in seconds, memory in MB, INFI indicates the result is not known because either time or space limitation is reached, time limitation is 300s and space limitation is 1GB

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

time to find a 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. Bitstate 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 2.2 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 2.3 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 in the matrix models, bitstate hashing again outperforms the other two abstraction schemes.

Table 2.2: 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 2.3: 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 2.4: 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 2.5: 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 the models discussed so far, bitstate hashing is by far more efficient than the other abstraction techniques. In fact, in these same models, concrete exploration will be even faster 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 2.4, 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 code and the environment is 2000 lines of code. 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 execute 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 2.5 shows a similar result. In Table 3.5, we increase the size of the component. Bitstate hashing again finds the error faster than concrete exploration.

2.6 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 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 results show 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 that can be found by simply executing the component. This improvement is a result of when the model checker pruning 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 the 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.

Chapter 3

Model Checking Concurrent C Components in Concrete Environments

We present a novel approach to verifying concurrent components by abstracting the component under test and model checking it within the context of a concrete software environment. By abstracting the component, we can verify more complex components. By leaving the environment concrete, we are able both to avoid the environment generation process and to allow the environment to contain complex operations that are not amenable to model checking. We implement this process in the Simple PROMELA Interpreter (SPIN) to verify concurrent C components with a subset of the pthread library. We verify and find errors in concurrent components within complex environments that SPIN failed to verify directly.

3.1 Introduction

Component-based software assembles existing components to build a piece of functioning software. Interest in component-based software is increasing because it reuses already existing software, which reduces developmental costs. Verifying components in the different environments in which they will be deployed, however, is a challenging task due to the complexity of both the components under test and the environments. This problem becomes more complicated when the components have concurrent behaviors and the source code for the environment is unavailable. *Component* here refers to a reusable piece of code that can accomplish a certain task, and *environment* refers to the rest of the program, which interacts with the component under test and closes it to make it executable. The environment can contain complex data structures, pointers, and library calls.

The ability to model check components in new environments is important to software engineers who build applications from existing components for environments for which the code is too complex to admit a formal model or for which the source code is simply unavailable. They need to ensure that a new component works as expected in a given software environment. With the growing popularity of dual-core and multicore machines, verifying threaded components also becomes increasingly important for software engineers.

Previous approaches to addressing this problem have included testing and model checking. Testing is an effective and easy way to verify the system in the early stages of debugging, when there are many errors in the system. However, it is harder to find bugs and more time consuming to build tests when the system becomes cleaner and the preconditions to an error are more complex. Tracing and reproducing an error is even more difficult when the software exhibits concurrent behaviors.

Model checking complements testing by exploring all behaviors of the system and providing error traces when an error is found. The most common way to use model checking in component-based software is to abstract the environment and model check the component under test in the abstract environment. The abstract environment closes the component under test and drives it to explore all possible behaviors of the component. The drawback of this approach is that abstracting the environment is difficult or impossible when no formal model exists for the environment. This can happen, for example, when there is no documentation or source code available for the environment. Another drawback is that the component itself is often too complex to be verified concretely. When the component is a threaded program, its state space grows exponentially due to scheduling choices, which can easily cause a state space explosion problem.

In this article, we present a novel approach to verifying concurrent components inside complex software environments by abstracting the components under test and model checking them in the given concrete software environment. Given a component-based software artifact, we execute the environment in its native execution platform, and when program control enters the component, the model checker analyzes the state space of the component, while driving execution to explore all possible thread interleavings. When program control reenters the environment, the model checker is stopped, and the program is executed in the native environment. This process continues until an error is found or all possible interleavings of the component have been explored.

We model concurrency in threaded code by extending an existing software model checker to support C programs with selected pthread library calls. We translate pthread functions to corresponding functionalities of the software model checker. This allows us to use model checking infrastructure to verify all C instruction interleavings. We use the pthread library to achieve concurrency in C programs because the pthread library is well known and widely used. We support a subset of commonly used pthread functions, including thread creation, mutex, conditional variables, and thread joining. In this work, we assume that the environment is sequential, and we focus on modeling concurrent components at the C statement level, instead of at the assembly code level.

This approach has two main advantages compared with existing model checking approaches. First, it avoids the analysis of the environment by simply executing it, rather than modeling or abstracting it. Modeling and abstracting include complex behaviors that are difficult or impossible to analyze using a model checker such as complicated data structures, non-Pressburger arithmetic, library calls, and pointers. Our approach also works when the source code of the environment is not available.

The second advantage is that more complex components can be verified by applying abstraction to the component under test. State space of the component grows fast when the component is complex or when the component is composed of threaded programs. Abstraction reduces state space size by compressing the state space.

We have implemented this approach by extending the Simple PROMELA Interpreter (SPIN) model checker to verify threaded programs written in C. Given a C program, we divide the program into the component and the environment. The component is translated into a mixed C/PROMELA model, and the environment is left as is. PROMELA is the native SPIN modeling language, and we leverage support for execution of C instructions within PROMELA models. Branching C instructions are translated into equivalent PROMELA constructs, while sequential C instructions are left as is.

We evaluate the idea with several test cases that include threaded components within complex environments. The test cases include software with more than 10,000 lines of code in the environment and software that has interesting functionality in the environment such as computing options prices. Our tool finds an error in a mutex program we obtained online. These test cases are too complex for SPIN to verify directly, but our approach succeeds both in getting high coverage of the state space and in finding errors.

This article is organized as follows. In section 2, we discuss related work. In section 3, we describe the verification model. In section 4, we provide the implementation method. In section 5, we present the test results. In section 6, we offer conclusions and discuss avenues for future work.

3.2 Related work

As mentioned earlier, the goal of our approach is to model check concurrent components by abstracting the component and running the environment as is. In this section, we discuss two aspects of related work. First, we present existing methods for verifying concurrent components. Then we discuss related work in compressing the state space of the components under test.

3.2.1 Verification of concurrent components

JPF [23] is a run-time software model checker that verifies threaded Java bytecode. Similar to our approach, it also supports the integration of running the program in its native JVM environment and state tracking the program under the direction of JPF. The main difference between JPF and our approach is in environment generation: JPF requires test drivers or environment models to close the component under test, whereas we do not use test drivers or environment models explicitly. The original concrete environment is our test environment, and we close the component under test simply by executing the concrete environment itself.

Pasareanu et al. [20] extend JPF to generate test cases for the unit under analysis. They symbolically simulate the unit under test and concretely run the surrounding environment to drive the unit under test. Similar to our approach, no test drivers or environment models are necessary. Our previous work in model-based test generation [3] is similar to the work of Pasareanu et al. [20] in that both works generate test cases for components under test by exhaustively exploring the state space of the components under test with the help of a model checker in a concretely executing environment. The difference between our previous methods [3] and the methods of Pasareanu et al. [20] is that the latter generate test cases by symbolically analyzing the component under test, but our previous methods generate test cases by concretely executing the component under test. In this article, we model check the component under test, instead of generating test cases for it.

Verisoft [10] is a software model checker that verifies concurrent C components. The strength of Verisoft is that it does not need to model the application under test. It is able to verify relatively large C programs directly by stateless search. The depth of the search is generally limited to prevent stateless search from getting trapped in cycles. Verisoft provides full state space coverage within certain bounds, with the cost of redundantly exploring the same states and transitions many times. As with JPF, it needs an environment model to close the application. Our approach is also able to model check relatively large C programs directly. Unlike Verisoft, we do not need to create an environmental model, and our state space exploration is not limited within given bounds.

Pancam [24] verifies multithreaded C programs in SPIN. It first translates the given C program into Low Level Virtual Machine bytecode and then implements a virtual machine to execute the bytecode as directed by SPIN. This approach model checks the entire program, which quickly causes a state space explosion due to the large state vector size of any C program. Our approach is different in two major ways: First, we model check the given program at the C statement level, instead of at the bytecode level, which gives us less precise but more compact state space compared with Pancam; second, we divide the program into the component and the environment and completely ignore the environment code. This allows us to verify components inside complex environments of any size.

CUTE [22] is a unit testing tool for C programs. CUTE collects path constraints while it executes the program and generates new input to the unit under test so that the next run of the program with new input takes alternative paths. We model check components under test, instead of generating test cases for them. The advantage of our approach is that we can reason about more complex properties.

CHESS [18] is a software model checker developed at Microsoft for finding errors in multithreaded software. It replaces the Windows OS scheduler with its own scheduler to systematically explore all scheduling choices. The major difference between our approach and CHESS is that CHESS operates on a user-defined test harness, while our test harness is provided by the concretely executed environment.

3.2.2 Abstraction

Slam [2] is a software model checker developed at Microsoft Research to verify Windows device drivers. Since the device driver communicates with kernels, verification of the device driver needs a model of the kernel to close the execution environment. The kernel, however, is a complex program without a formal specification. Ball et al. [1] abstract the Windows device driver along with required kernel procedures using overapproximation. The abstract system is then refined by analyzing spurious counterexamples. We also use abstraction in the component under test to reduce the state space size; however, we use underapproximation, instead of overapproximation, to simplify communication between abstract component and concrete environment.

Pasareanu et al. [21] describe an underapproximating predicate abstraction scheme. This abstraction works by executing the concrete transition relation on concrete states, rather than computing an abstract transition relation that will be applied to abstract states. The abstraction function is applied to concrete states before they are inserted into the hash table. In effect, concrete states are stored in the stack of active states, and abstract states are stored in the hash table. Our abstraction is similar to this abstraction in principle; however, instead of using predicate abstraction, we use bitstate hashing, which allows us to include more complex control and data structures in the program, without exceeding the capability of the theorem prover.

3.3 Verification model

In this section, we describe the modeling approach we use to verify concurrent components in concrete software environments by discussing component modeling, concurrency modeling, and component abstraction.

3.3.1 Component modeling

Figure 3.1 shows the algorithm. It starts by calling the procedure **init**, which takes a program *prog* as input. This procedure generates the start state, pushes it into the stack in line 2, and abstracts it using bitstate hashing in lines 3 and 4. Line 3 calculates an address from the start state, and line 4 sets the bit in that address to indicate that this state has already been visited. The start state is always in the component because we always have a driver function in SPIN to start the main function in C. In line 5, we call a component procedure to start the state space exploration of the concurrent component.

The **component** and **environment** functions explore the state space of the mixed abstract concrete system. When the program instruction is inside the component, the program executes under the direction of the model checker. When the program control enters the environment, the program executes as in its native execution environment.

In line 18, the **component** function checks whether the state on top of the stack has an enabled transition. If the state does not have an enabled transition, then we have already fully explored the state, so we pop it out of the stack in line 19. If the state has an enabled transition, then we check the type of the current instruction. If the instruction is in the component and is from the pthread library, as checked in line 10, then we call the **c_to_spin** function in line 11 to simulate the pthread behaviors in SPIN. The **c_to_spin** function returns the next state by executing a simulated SPIN instruction. If the instruction is not a pthreaded instruction, but rather, resides in the component, as line 12 indicates, then we generate the next state by executing the current instruction in line 13. If the instruction does not lie in the component, then we call the **environment** function in line 14. The **environment** function returns the next state residing in the component. In line 15, we calculate an address from the state vector using a hash function. Then we check if the bit in that memory address is set in line 16. If the bit is not set, then we set the bit to show that the state has already been visited in line 17 and push the state into the stack in line 18. If the bit has already been set, then we assume that we have a duplicate state and ignore it. Hash collision occurs when two different states are hashed into the same address. Lines 15 to 17 are a simplification of the actual bitstate hashing algorithm.

As discussed in the previous paragraphs, we simulate four of the most commonly used pthread concurrent behaviors in SPIN: creating a thread, using mutex to lock and unlock shared variables, waiting and signaling on a condition, and then joining the threads. The **c_to_spin** function takes a pthread instruction and turns it into a corresponding SPIN instruction. Then it executes the SPIN instruction to generate the next state of the component and returns it back to the component function. In line 21, for example, we check the type of current instruction. If it is a thread creation instruction, then we translate it into a corresponding SPIN instruction in line 23 and execute it to generate the next state in line 24. The other cases follow a similar pattern.

The **environment** function runs the code in the native execution environment until it hits the first state residing in the component; it then returns to the component function. In lines 39 and 40, we execute the program as is. When the instruction leaves the environment, we check if it is a pthread instruction. If it is, then we call the function **c_to_spin** in line 42. Otherwise, we return to the **component** function in line 43. One special situation to consider here is when the environment invokes a thread. In this work, we assume that the environment is sequential. Therefore, when the environment invokes a thread, we assume that the thread is in the component and register it with an array of threads that SPIN creates in line 38.

3.3.2 Concurrency modeling

One of the main advantages of a model checker is its ability to reason about concurrent programs. In this section, we discuss how we model the concurrent component in our model checking environment.

In this work, we model pthread function calls using model checker concurrency primitives. This allows us to leverage the model checker's state space exploration infrastructure and concurrency abstraction. As shown in the **c_to_spin** function in Fig. 3.1, we simulate a subset of pthread that includes creating a thread, using mutex variables, using conditional variables, and joining the threads. These functions are the most commonly used functions in the pthread library, and we can generate useful real-world examples from them. Other pthread functions can be implemented but require reimplementing the pthread library and rewriting the SPIN model checker, which is beyond the scope of this work. Significant modifications are required because pausing the environment to switch thread contexts requires the ability to pause execution.

We focus on modeling concurrent components at the C statement level, instead of at the assembly code level. This approach misses some behaviors because each C statement corresponds to several assembly instructions and threaded program interleaves at the assembly instruction level. We might miss different kinds of errors such as deadlock and race conditions. The advantage of this approach is that it significantly reduces state space size by exploring fewer interleavings. This kind of state space reduction is especially important when we reason about concurrent software systems described in general-purpose programming languages because the state space size of the general program grows much more quickly than that of the regular model due to the large number of rich data types used by general programs.

```
proc init($prog$)
  push($start\_state$)
  $addr$ = hash ($start\_state$)
  $inMem[addr] = 1$
  component()
proc component()
  while (size(stack) != 0)
      $cur\_state$ = top(stack)
      if ($cur\_inst$ = next_transition($cur\_state$))
          if ($cur\_inst \in$ component && $cur\_inst$ == $pthread\_inst$)
            $next\_state$ = c_to_spin($cur\_inst$, $cur\_state$)
          else if ($cur\_inst \in$ component && $cur\_inst$ != $pthread\_inst$)
            $next\_state = cur\_inst(cur\_state)$
          else $next\_state$ = environment($cur\_inst$, $cur\_state$)
          $addr$ = hash ($next\_state$)
          if (!$inMem[addr]$)
            $inMem[addr] = 1$
            push ($next\_state$)
      else pop(stack)
proc c_to_spin($cur\_inst$, $cur\_state$)
  switch ($cur\_inst$)
    case $pthread\_create$ :
      $cur\_inst = spin\_proc$
      $next\_state = cur\_inst(cur\_state)$
    case $pthread\_mutex$ :
      $cur\_inst = spin\_mutex$
      $next\_state = cur\_inst(cur\_state)$
    case $pthread\_cond$ :
      $cur\_inst = spin\_cond$
      $next\_state = cur\_inst(cur\_state)$
    case $pthread\_join$ :
      $cur\_inst = spin\_join$
      $next\_state = cur\_inst(cur\_state)$
    case $default$ :
  return $next\_state$
proc environment($inst$, $state$)
  do {
    if ($inst$ == $pthread\_create$) register $spin$ proctype
    $next\_state = inst(state)$
    $inst$ = next_transition ($next\_state$)
  } while ($inst \in$ environment)
  if ($inst$ == $pthread\_inst$) return c_to_spin($inst$, $next\_state$)
  else return ($inst(next\_state)$)
```

Figure 3.1: State enumeration algorithm combining an abstract component with a concrete environment

3.3.3 Component abstraction

As mentioned earlier, state space explosion is the most challenging problem to address for models in general programming languages because such models have not only concurrency, like other models, but also rich data types, which contribute to most of the state vector size. We use an abstraction technique to reduce the state space size of the concurrent component. The choice of abstraction type is important here to ensure smooth communication between the abstract component and the concrete environment. In this section, we analyze the existing abstraction techniques and discuss the abstraction approach we use.

Overapproximation introduces extra behaviors to the component. Overapproximation is not a good choice for our model because overapproximation abstraction passes the extra behaviors into the concrete environment, which might cause the program behaviors to become intractable or the environment to fail. Hence we use underapproximation to derive the abstract state from the actual concrete state. The concrete states of the component under test are explored and stored on the stack, and abstract representations of these concrete states are stored in a hash table to detect termination and duplicate states. The idea of the abstraction is similar to the underapproximation abstraction proposed by Pasareanu et al. [21]. However, we use bitstate hashing, instead of predicate abstraction, because the ability to model check programs using predicate abstraction is very limited because of the limited power of the theorem prover. Also, the time required to use the theorem prover exceeds the savings gained through precision and refinement for most programs (D. Kudra and E. G. Mercer, Finding termination and time improvement in predicate abstraction with under-approximation and abstract matching, MS thesis, Brigham Young University, 2007).

Bitstate hashing [11] is the abstraction technique we use in the component under test. In Fig. 3.1, lines 15 to 17 demonstrate this process. Bitstate hashing abstracts a state into a single bit. Given a state vector, a hash function is applied to it to generate an address. If the bit in this address is set to 0, then we know that this state is a new state, and we set the bit in that address to 1 and continue exploring it. If the address has already been set to 1, we assume that the state has already been visited and stop exploring it. Hash collision occurs when two states are hashed into the same address. This can be reduced by increasing the number of bits used to represent a state or by using multiple hash functions to compute the address.

3.4 Implementation method

We implement this work in SPIN and verify component-based software written in C using the pthread library. In this section, we introduce different tools and the verification method we use to achieve this goal.

Given a C program with pthread library calls, we translate it into a PROMELA model that is usable by SPIN. The pthread library calls are translated into equivalent PROMELA instructions. The translation from C code to PROMELA is done using the C Intermediate Language Compiler (CIL) [19]. CIL can compile most valid C programs into a few core constructs with clean syntax. For example, CIL compiles all looping constructs in C into a single form, and it adds an explicit *return* statement to all function bodies. This simplifies the varieties of syntax we need to consider during the translation process. Figure 3.2 shows a simple C program and the corresponding CIL program. We use CIL to parse the given C program into a simplified but semantically equivalent form, and we modify the CIL code generator to generate a PROMELA model.

The translation between C code and PROMELA is further simplified by SPIN's support of embedded C code. SPIN provides five different primitives to allow C code to run within PROMELA. They are identified by the keywords `c_code`, `c_track`, `c_decl`, `c_state`, and `c_expr` [13]. Everything enclosed inside a `c_code` block is

```
void main() {
  int i;
  for (i = 0; i < 10; i++)
    if (i == 4) break;
    else i = i * 2;
}
```

(a) A simple C program

```
void main(void) {
  int i = 0;
  while (1){
    while_0_continue: /* CIL Label */;
    if (! (i < 10)) goto while_0_break;
    if (i == 4) goto while_0_break;
    else i * = 2;
    i++;
  }
  while_0_break: /* CIL Label */;
  return;
}
```

(b) The CIL translation

Figure 3.2: An example of C to CIL

compiled directly by the GNU C Compiler and then executed and treated as one atomic PROMELA state. Keywords `c_state` and `c_track` declare a C variable to be part of the state vector. Keyword `c_decl` is used to declare C variables inside PROMELA. Keyword `c_expr` encloses any form of C expression that does not have side effects.

These primitives facilitate a direct translation between C source code and PROMELA. We enclose each statement in the component in one single `c_code` block. This allows SPIN to generate and store states for each statement in the component. All statements, except function calls, *while* statements, and the *if* statement, can be enclosed inside a `c_code` block to generate a PROMELA statement.

Since statements can be nested in the *while* statements or *if* statement, simply enclosing them into a `c_code` block will miss exploring the behaviors of the intermediate statements. Therefore we need to turn them into loops and conditional statements of PROMELA. The *while* loop needs to be translated into *do :: sequence [:: sequence*

] od* in PROMELA, in which the sequence is composed of PROMELA statements that are similar to the C statements. In other words, PROMELA contains the statement block between *do* and *od*, instead of lying inside a *while* block. The *if* statement in C needs to be translated into *if :: sequence [:: sequence]* fi* in PROMELA. The details of this translation can be found in the work of Bao [4]. The purpose of this translation is to ensure that SPIN is aware of all possible interleavings of the C statements. The translated PROMELA model depends on SPIN's underlying state space reduction techniques to reduce its state space.

When the statement is a function call, then we have two cases to consider: If the callee function is inside the component, then we translate this function call statement into a PROMELA statement, generating a new thread. If the callee is a library function or a function inside the environment, then we simply enclose it inside a `c_code` block to execute it as is.

We use `c_decl` to embed all global variable declarations in C and use `c_track` to tell SPIN to include those variables in the state vector. All local variables in the component are declared in `c_state` to indicate that those variables should be included in the state vector and tracked by SPIN. We enclose everything in the environment in one `c_code` block to execute it as is.

Figure 3.3 shows a C program and its corresponding PROMELA program. In Fig. 3b, line 1 declares i as a local variable of proctype main and is included in the state vector. When the PROMELA variables are accessed in the `c_code` fragment, SPIN needs a special prefix to identify them. For global variables, this prefix is *now* followed by a period. For local variables, this prefix is an uppercase letter P, followed by the name of the process type, followed by a right-pointing arrow ($\rightarrow$). Line 2 of Fig. 3a, for example, corresponds with line 4 of Fig. 3b. Here i in C code is translated into $Pmain \rightarrow i$ in PROMELA. The same rule applies to all variables. The rest of

```
void main(void) {
  int i = 0;
  while (1){
    while_0_continue: ;
    if (! (i < 10))
      goto while_0_break;
    if (i == 4)
      goto while_0_break;
    else i *= 2;
    i++;
  }
  while_0_break:;
  return;
}
```

(a) A simple CIL program

```
c_state "int i " "Local main"
proctype main()
{
  c_code {Pmain → i = 0;};
  do
  :: while_0_continue: ;
    if
    :: c_expr {! (Pmain → i < 10)} →
         goto while_0_break;
    :: else → skip;
    fi;
    if
    :: c_expr {Pmain → i == 4} →
         goto while_0_break;
    :: else → c_code {Pmain → i *= 2; };
    fi;
    c_code {Pmain → i++;};
  od;
  while_0_break: ;
}
```

(b) The PROMELA translation

Figure 3.3: An example of C to CIL

the translation is mostly a straightforward statement-by-statement matching between C statements and PROMELA `c_code` fragments.

In Fig. 3.4, lines 2 and 5, *pthread_create* creates a thread, assigns it an ID, and passes an argument to the thread. SPIN has a corresponding instruction to create a thread, but the C argument cannot be passed directly through SPIN's thread creation instruction because SPIN does not support rich data types, as does C. We solve this problem by following three steps: First, we assign a unique global ID number to each SPIN thread that is generated; then we copy the C argument into a piece of global memory indexed by this thread ID using embedded C code; finally, we pass the ID as an argument to the new thread. When the new thread is executed, it receives the argument from global memory, indexed by its ID passed as the parameter.

```
1   C statement:
2     pthread_create(&thread[t], NULL, PrintHello, (void *)t);
3
4   CIL translation:
5     pthread_create((pthread_t * __restrict )(& threads[t]), (pthread_attr_t const *
    __restrict
6           )((void *)0), & PrintHello, (void * __restrict )((void *)t));
7
8   PROMELA translation:
9     atomic {
10      run PrintHello(globalID);
11      c_code {
12        *(& Pmain → threads[Pmain → t]) = now.globalID;
13        Garg[now.globalID] = (void * __restrict )((void *)Pmain → t);
14      };
15      globalID = globalID + 1;
16    };
```

Figure 3.4: C, CIL, and PROMELA version of creating a new thread

Figure 3.4 shows this translation. The function *pthread_create* creates a thread named *PrintHello*, gives it a unique identifier *thread[t]*, and passes it to the argument *t*. Lines 9 to 15 show the PROMELA translation. Line 10 invokes the thread *PrintHello* in PROMELA. Line 12 assigns the new thread a unique ID number. Line 13 stores the C argument into a global argument array *Garg*, indexed by the thread ID. All of the preceding steps are done in one atomic step to simulate *pthread_create*.

```
 1  C statement and CIL translation:
 2    void *PrintHello(void *threaarg)
 3    {
 4      struct thread_data *my_data;
 5      my_data = (struct thread_data *)threadarg;
 6    }
 7
 8  PROMELA translation:
 9    proctype PrintHello(int threadID)
10    {
11      c_code{PPrintHello → threadarg = Garg[PPrintHello → threadID];};
12      c_code {PPrintHello → my_data = (struct thread_data * )PPrintHello →
    threadarg;};
13    }
```

Figure 3.5: Argument passing

Figure 3.5 shows how the thread receives its argument. In line 9, *threadID* contains the unique ID of thread *PrintHello*. When this thread starts its execution, it will first grab its argument from global argument array *Garg* by using its ID as an index at line 11.

Function *pthread_join* blocks the caller thread until the indicated thread finishes. This is done in SPIN by channel. We declare an array of global channel indexed by the thread ID. The caller of *pthread_join* blocks in the global channel called *Gchan* that is indexed by the ID of the thread it is waiting to join. When each thread finishes, it sends a signal to the channel indexed by its thread ID. When the caller thread receives the signal, it unblocks and resumes its actions.

Functions *pthread_mutex_lock* and *pthread_mutex_unlock* lock and unlock a mutex variable, respectively, to give the thread the exclusive right to access a shared variable. We declare a corresponding mutex variable in SPIN. A thread locks the mutex variable by subtracting 1 from it and unlocks the variable by adding 1 to it.

Before locking the variable, the thread needs to ensure that the variable is unlocked. Locking and unlocking steps occur in an atomic step in PROMELA so that no other thread can access the mutex variable when one thread has access to it.

Funcion *pthread_cond_wait* blocks a thread until a condition is met. Function *pthread_cond_signal* signals the blocking thread if the condition is met. In SPIN, we create a corresponding conditional variable. The thread waits in a certain condition for the conditional variable to become 1 and signals a certain condition by assigning 1 to the conditional variable. The details of the translation can be found in Appendix B.

3.5 Experimental results

The experimental results given in this section show that our tool is able to verify and find errors of concurrent components running in complex environments in a time- and space-efficient manner.

The test cases we use to evaluate the tool include producer-consumer (PC), reader-writer (RW), dining philosopher (DP), dining philosopher with deadlock (DP_E), and conditional variable problems (CV and CV2). All the test cases are open source code downloaded from the Internet. Table 3.1 shows the verification results for all models.

The PC program is composed of two threads: the producer and the consumer. They share a common buffer with limited size. Producer and consumer use a mutex variable to ensure exclusive access to the buffer when they need to produce or consume. Producer waits if the buffer is full, and consumer waits if the buffer is empty. The model itself is composed of 207 lines of code. We extend this model to add a requirement that the consumer has to pay before he consumes a product. This is done by adding an ATM bank application to the model.

Table 3.1: Verifying concurrent software components in a concrete environment[a]

Models	cLine	eLine	States	memAbs	Mem	Time	eTime
PC	207	10000	93312	1.9	26	840	NDEF
RW	204	825	414366	2.5	94	2520	NDEF
DP	75	380	13M	269	3712	120	NDEF
DP_E	108	380	1520	3	NDEF	NDEF	4.1
CV	118	23	9M	135	3221	72	NDEF
CV2	141	0	273M	1073	180606	3300	NDEF

[a]All times are in seconds. Memory is in megabytes.

The first column gives the name of the model and the first row gives the different metrics we used to evaluate our approach. Abbreviations are as follows: cLine, number of lines of code in the component; eLine, number of lines of code in the environment; eTime, time spent finding errors; M, million states; mem, amount of memory needed when no abstraction is applied to the state space; memAbs, amount of memory used when bitstate hashing is used; NDEF, not defined; states, number of states explored; time, time spent exploring the full state space.

The ATM bank application is also an open source code downloaded from the Internet. It is a stand-alone program that simulates banking processes such as opening an account, depositing and withdrawing money, checking an account balance, and so on. This application is composed of more than 10,000 lines of code. It includes complex data structures, sockets, library calls, and pointers.

Integrating the PC problem and the ATM application directly will result in a complex program that is currently impossible to verify with the existing software model checker. In our approach, we divide the system into component and environment by including the PC threads in the component and the ATM application in the environment. Now, when the consumer calls any functions in the ATM application, they execute in the native execution environment, without interruption from the model checker.

The second verification model we used is the RW problem. Reader and writer are two threads operating on the same object. Reader waits for a condition that signals him to read, and writer waits for a signal to write. Both need to have an exclusive right to the shared object when they want to operate on the object. Here we add an environment code to enable the reader to sort the items he reads. This is done by having the reader call a sort function whenever he reads an object. The second row in Table 3.1 shows this result.

The third and fourth models are DP and DP_E, in which the environment computes options prices. The environment we provided was a stock options price calculation program that includes mathematics such as square root, exponential functions, and so on. Our verification result indicates that the third one is error-free and the fourth one has a deadlock in it. The error we discovered in the fourth model is an actual error in the original code downloaded from the Internet. This error is caused by a possible sequence of execution in which all philosophers get one chopstick and wait for the other one. The third and fourth rows of Table 3.1 show this result.

The fifth model is a conditional variable model. In this model, one thread waits on a condition, and the other threads work toward satisfying the condition. Once the condition is met, they signal the waiting thread. Different from the other examples, this model also calls the component from inside the environment.

We modify the fifth model CV to generate CV2. In this model, we verify the entire program, instead of dividing it into component and environment. The environment only includes 23 lines of code in the CV model. This environment has the simple functionality of turning an integer into a string. However, it contains some complex data structures. This model demonstrates how even very small environments can add complexity to the component. The test result is given in the last row of Table 3.1. In this model, the verification is not able to terminate. This is because the environment adds one more thread with more data sets to the system.

Table 3.1 shows one common feature for all the models; that is, although the number of states explored is quite large for all the models, they all use relatively small memory sizes. That is because we applied bitstate hashing here. The average state size for these models is 250 bytes, which is reduced to 3 bits in bitstate hashing, which is the default number of bits for bitstate hashing in SPIN. The difference between column "memAbs" and column "Mem" shows us the amount of space saved by using abstraction.

The test results indicate that our method is able to verify models that cannot be verified by SPIN directly due to their complexity. Although we are not able to guarantee full state space exploration even after the algorithm terminates due to bitstate hashing, we can predict the state space coverage quite accurately by analyzing the hash factor from the verification result [11].

3.6 Conclusion and future work

In this article, we present a concurrent component verification method that is able to verify concurrent components within complex concrete environments. We abstract the component under test and verify the abstract component using a model checker. The rest of the program is left concrete to run as is in the native execution environment. The abstraction is designed to simplify the interaction between the abstract component and the concrete environment. We implement this approach in SPIN to verify C programs using a subset of pthreaded libraries. This tool is evaluated by several test cases that are composed of concurrent components and complex environments. SPIN fails to verify them directly, while our approach is able to verify and find errors in these test cases.

In the future, we plan to work on automatically extracting components from component-based software. Currently we define a *component* as a collection of user-provided functions. We will also work on extending our tool to support the full pthread library.

Chapter 4

Test Case Generation Using Model Checking for Software Components Deployed into New Environments

Abstract

In this paper, we show how to generate test cases for a component deployed into a new software environment. This problem is important for software engineers who need to deploy a component into a new environment. Most existing model based testing approaches generate models from high level specifications. This leaves a semantic gap between the high level specification and the actual implementation. Furthermore, the high level specification often needs to be manually translated into a model, which is a time consuming and error prone process. We propose generating the model automatically by abstracting the source code of the component using an under-approximating predicate abstraction scheme and leaving the environment concrete. Test cases are generated by iteratively executing the entire system and storing the border states between the component and the environment. A model checker is used in the component to explore non-deterministic behaviors of the component due to the concurrency or data abstraction. The environment is symbolically simulated to detect refinement conditions. Assuming the run time environment is able to do symbolic execution and that the run time environment has a single unique response to a given input, we prove that our approach can generate test cases that have complete coverage

of the component when the proposed algorithm terminates. When the algorithm does not terminate, the abstract-concrete model can be refined iteratively to generate additional test cases. Test cases generated from this abstract-concrete model can be used to check whether a new environment is compatible with the existing component.

4.1 Introduction

As the complexity of software increases, so does the cost to develop, test, and maintain software. Component based software development decreases this cost by reusing software that has already been developed and tested. One difficulty of reusing a component is ensuring that a component developed in one environment works as expected in another environment.

In this paper, we address the problem of testing a new software environment for compatibility with an existing component. We define components as a reusable piece of code that can accomplish a certain task and is designed to interact with other components or programs. The environment is the rest of the software which closes the component to make it executable. The environment can include the other components or programs that interact with the component. A new environment is an environment other than the one in which the component is originally deployed. In this work, we assume the component is already defined and we do not address the problem of extracting a component from a software artifact.

This problem is important to software engineers who must deploy a component inside a new software environment. Since software is increasingly built from the existing components, verifying whether the already developed components work as expected in new environments become more important.

Existing techniques to address the problem of verifying software components include testing and formal verification. Software testing is useful in finding many errors in the early stages of software development. Since it is well understood that software can not be tested completely, finding a suitable set of test cases is critical in software testing. Model based testing is often used to generate a promising set of test cases by creating an abstract test model of the components to guide test case generation. The abstract model should capture important functionalities of the

System Under Test (SUT) and result in a useful test suite. Model based testing technique generates test cases according to the test goal, such as a coverage criterion.

The two main challenges for model based testing are how to generate a model and how to generate the test cases. The current approaches often generate the model manually. There are several drawbacks for this approach. First, It is a time consuming and error prone process to manually generate an abstract model of the SUT. In most cases, the SUT is too complex to be accurately modeled. Second, if the model is manually generated, generally there is a semantic gap between the model and the original system. Therefore, it might not be feasible to turn the abstract test cases into executable test cases. Third, once the model is generated, it is fixed. Refining the model to generate more test cases or adding new features to the original system requires an entire new modeling process. Testing an existing component within a new environment is difficult when using model based testing approaches because the new environment is often too complex to be modeled or there is simply no high level specification or source code available for the environment from which a model can be extracted.

Model checking complements software testing by locating errors which are difficult to find using software testing alone. However, applying model checking to a component is difficult due to the complexity of both the component and its environment.One technique is to abstract the environment and leave the component under test concrete. The abstract environment provides different inputs to the component so that the model checker can explore all behaviors of the component. The drawback of this approach is that abstracting the environment is difficult or impossible when there is no formal model for the environment. Furthermore, the component itself can be so complex that model checking it without abstraction is also impossible.

In this paper, we propose an original approach to generate test cases to verify the compatibility of existing components with new software environments. Our approach is a novel integration of model checking with model based testing to generate test cases for the new environment. Through the integration, we obtain the advantages of both model checking and model based testing and reduce the drawbacks of both techniques.

Given a component and the original environment, we abstract the component and leave the environment concrete. We start by executing the entire system normally. When program control enters the component, a model checker generates concrete system states from the component source code. We then apply an under-approximating predicate abstraction based on [21] to the concrete states. The abstraction saves time and space by reducing the model. When program control reaches the environment, we pause the model checker and save the program state into the test input set. We then initialize the environment using the program state and execute the system in the native execution environment. When program control returns to the component, we save the current program state into the test output set and resume the model checker. We repeat this process to obtain a set of input and output values. These values are test cases which can be used to test a new software environment for compatibility with the component.

The abstraction approach has been used to model check software consisting of hundreds of lines of code in several medium sized C programs with several thousand lines of code in less than 5 minutes. These programs include open source software and student projects from an operating systems class. The results are discussed along with the implementation of the algorithm in SPIN in [4]. In this paper, we propose an adaptation to the method to support test generation for components deployed into new environments.

Figure 4.1 illustrates this approach. The left side of the figure shows the model composed of an abstract component, which is represented by the dotted square, and a concrete environment, which is represented by the solid square. The circles and

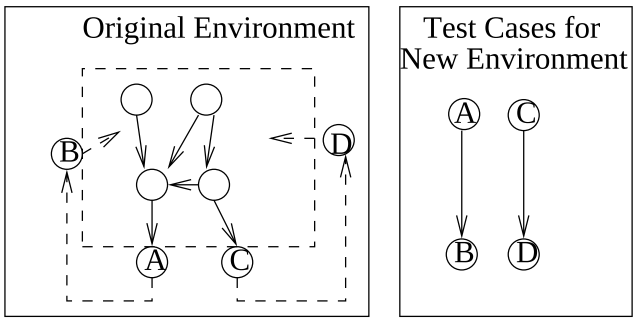


Figure 4.1: Test generation for a new environment using an abstract model of the component under test.

arrows indicate program states and transitions respectively. The state space of the abstract component is explored until execution exits the component and enters the first environment state A. Then the program executes normally until it reaches the last environment state B and reenters the component. A similar process occurs for C and D. The right figure shows the resulting list of test cases for the new environment. In this work, we assume the environment has a single unique response for a given input.

In order to generate test cases using an abstract model of a component within a concrete model of the environment, we need to address two problems. One is to simplify communication between abstract components and concrete environments. The other one is to obtain a precise model for the abstract component through refinement. The precise model is a model that is bisimilar to the concrete component so that it behaves the same as the concrete component. Communication between component and environment is simple when state exploration in both the component and environment is based on executing the original program on concrete states. If we use an abstract state in the component, however, all concrete states represented by the abstract state at the component interface must be generated, passed to the environment, executed in the environment and then recollected to form possibly new abstract states. We discuss completeness, correctness and refinement for an abstraction scheme based on [21] which address the above problems.

Generating test cases with the abstract component is possible because the new environment needs to have the same interface as the old environment in order for the component to function inside the new environment. Assuming there is a single unique response for any given input to the environment, test cases generated using an abstract model of the component are compatible with the new environment if the component is compatible with the new environment.

The goals of our test cases include getting a complete coverage of the component interface behaviors in the original environment and detecting errors in the new environment. The test coverage obtained by our approach depends on the refinement and termination of the algorithm. Assuming the run time environment is able to do symbolic execution either through instrumentation or some other mechanisms, we can obtain complete test coverage for the component interface in the original environment when our algorithm terminates. We also can obtain more behavioral coverage for the original environment by including a large part of the environment in the component. When the new environment fails the tests, we know there are errors in the new environment. If the new environment passes all the tests, then we can not claim anything about the new environment because the new environment might include more behaviors than the original environment.

Compared with traditional model based testing, the advantage of our approach is that both modeling and test generation are automatic. And since test cases are obtained through running the component within the original environment, they can be applied directly to the new environment without the need to transform abstract test cases into executable test cases. Furthermore, our modeling and test generation approach can be refined iteratively to obtain an increasingly precise model yielding more test cases.

This paper is organized as follows. In section 4.2, we present related work. In section 4.3, we give the definition of the model which we use to reason about. In section 4.4, we describe the algorithm we propose. In section 4.5, we establish the necessary conditions for a bisimulation between the abstract component model and the original software. In section 4.6, we conclude and discuss future work.

4.2 Related Work

We focus on related work in the areas of model based testing, model checking component based software, generating models through abstraction, and combining symbolic simulation with concrete exectuion.

Most of existing approaches to apply model based testing to the SUT generate models from high level specifications [7,9]. The approach has the advantage of finding errors faster in the early stages of the development process. However, there is often a semantic gap between the high level specification and the actual implementation. Furthermore, if the high level specification is not written in a standard modeling language, a manual translation from the specification to the actual model is required, which is a time consuming and error prone process. Our method extracts a model directly and automatically from the source code to obtain an accurate model of the SUT fast. Our SUT is unique in a way that it is a mixed model of abstract component and concrete environment.

Existing work in model checking of component based software includes high level specifications of components as models and verifying concrete source code for a component inside an abstract environment. In [15, 5], the components are modeled using behavioral protocols. The drawback of this approach is that a precise high-level model of a component can be difficult to obtain. We obtain the component model by executing source code.

In [6], a program is divided into two parts: the component under test and the environment. The component under test is verified concretely inside the abstract environment which provides the necessary behaviors to close the environment. The drawback of this approach is it is a time consuming and error prone task to abstract the environment and the component itself may be too complex to be verified without an abstraction. In our approach, we abstract the component, which allows us to reason about more complex components and we leave the environment concrete which allows our environment to include language features such as complex data structures, pointers and library calls.

We apply Pasareanu *et. al*'s [21] under-approximation abstraction scheme to the component under test. This abstraction also explores the concrete state space and stores concrete states into a queue or a stack. A set of predicates is used to abstract each concrete state. Each predicate is represented by a single bit in the abstract state. Precision in the abstraction is lost when two concrete states map to the same abstract state, but have different behaviors in the original system. An iterative refinement is done to include more and more behaviors of the original system by adding predicates to the abstraction. The algorithm terminates when the resulting abstract state space is bisimilar to the concrete state space. Termination of the refinement process is detected by checking the weakest precondition of each transition in terms of the given predicates with the help of a theorem prover. In sections 3 and 5 we reason about similar properties of Pasareanu's abstraction scheme in a mixed concrete/abstract computational model.

There are several works which combine concrete execution with symbolic simulation to generate test cases. The concolic testing approach [22] starts by executing the unit under test with random inputs. Path constraints are then collected along with concrete execution paths. These path constraints are used to provide new inputs to the unit under test which drive concrete execution through an alternative path. We also combine concrete execution with symbolic simulation. Our approach is different in two major ways. First, concolic execution tests a unit by providing random input values. We test a new environment to see if it is compatible with an existing component by providing the environment the input that is generated by the component. Second, concolic execution uses symbolic execution and path constraints to generate new concrete input to the unit under test. We use symbolic simulationa and path constraint to generate new predicates which refine the component and this, in turn, generates new input to the environment. Compared with concolic testing, our approach is more restricted with theorem prover capabilities but gets better test coverage.

Pasareanu et al [20] extend JPF to generate test cases for the unit under test. They symbolically simulate the unit under test and concretely run the surrounding environment to drive the unit under test. Like our approach, it generates test cases by exhaustively exploring the state space of the components under test with the help of model checker in a concretely executed environment. The difference is [20] generates test cases by symbolically analyzing the unit under test, but we generate test cases by concretely executing the component under test.

4.3 Computational Model

In this section we present a computational model for abstract components inside concrete software environments for use in model based test generation. As mentioned earlier, we apply under approximated predicate abstraction based on [21] to model the components. Under approximating abstraction with predicates allows the detection of missed program behaviors through the use of weakest preconditions as described by Pasareanu *et.al* [21]. Later, we will describe how to reason about abstraction using weakest preconditions in our component representation.

Component A component is a reusable piece of code that can accomplish a certain task and which is designed to interact with other components or programs. We assume the source code of the component is available. In this work, we simply define a component as a set of program counter (pc) values. We let PC_c denote the set of pc values that belong to the component under test. The problem of extracting a component from a software artifact is important, but left as future work.

Our model of a system consists of a concrete part and an abstract part. We model both parts separately and then combine them in a mixed model as follows.

Concrete Model Given a finite set of atomic propositions AP, the state space of a program is modeled as a transition system $M = (S, s_0, T, L)$ where:

- S is a finite set of states,
- $s_0 \in S$ is the initial state,
- $T \subseteq S \times S$ is a set of transitions,
- $L : S \rightarrow 2^{AP}$ is a labeling function, where $L(s) = \{p \in AP \mid s \models p\}$. Here $s \models p$ means atomic proposition p is true in state s.

Figure 4.2 shows an example of the concrete model with other models that will be introduced later. In (a), several simple C program statements are given, and in (b) the corresponding concrete model is shown. Each circle represents a state and each arrow represents a transition. The first number in the circle represents the value of variable i and the second number represents the value of variable j.

Abstract Model (adapted from [21]) Assume $\Phi = \{\phi_1, \ldots, \phi_n\}$ is a set of predicates. Assume $\alpha_\Phi : S \rightarrow B_n$ is an abstraction function in which $B_n = \{0, 1\}^n$ is a set of bit vectors where n is the number of predicates, $b_1 \ldots b_n \in B_n$ is a bit vector, and

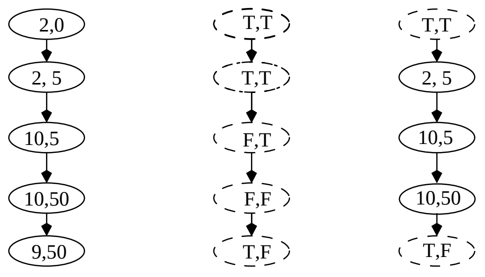
$\alpha_\Phi(s) = b_1 \ldots b_n$ with $b_i = 1$ if $s \models \phi_i$, else $b_i = 0$. Here $s \models \phi_i$ means the predicate ϕ_i is true in state s.

Given a concrete program model $M = (S, s_0, T, L)$, a set of predicates Φ, and an abstraction function α_Φ, the abstract model $A = (S_\alpha, a_0, T_\alpha, L_\alpha)$ where:

- $S_\alpha = \{a \mid s \in S \text{ and } a = \alpha_\Phi(s)\}$ is a set of abstract states,
- $a_0 = \alpha_\Phi(s_0)$ is the initial abstract state.
- $T_\alpha \subseteq S_\alpha \times S_\alpha$ is a set of abstract transitions.
- $L_\alpha : S_\alpha \rightarrow 2^{AP}$ is a labeling function for abstract states, where $L_\alpha(a) = \{p \in AP \mid a \models p\}$. and assume $AP \in \Phi$.

Figure 4.2 (c) shows the corresponding abstract model of (a) with dotted circles to represent abstract states. In this figure, we use two predicates $i < 10$ and $j < 10$ to abstract the system. The first boolean variable corresponds to $i < 10$ and the second boolean variable corresponds to $j < 10$. As is typical in predicate abstraction, we never abstract the program counter.

```
1    i = 2;
2    j = 5;
3    i = i * j;
4    j = j * i;
5    i = i - 1;
```



(a) (b) (c) (d)

Figure 4.2: (a) Sample C program (b) Concrete system (c) Abstract system using predicates $i < 10$ and $j < 10$ (d) Mixed system

Mixed Model The mixed model includes both the abstract and concrete parts of the program. Given a concrete program model $M = (S, s_0, T, L)$, a predicate set Φ, and an abstraction function α_Φ, the mixed model $X = \{S_x, x_0, T_x, L_x\}$ is

- $S_x = \{x \mid x = f(s) \text{ and } s \in S\}$ is a mixed set of abstract and concrete states, where f is a function such that

$$f(s) = \begin{cases} s & pc \notin PC_c \\ \alpha_\Phi(s) & otherwise \end{cases}$$

  If the pc value in the current state does not belong to the component, then the state remains concrete. Otherwise, the state gets abstracted.

- x_0 is a start state, so that if $pc \in PC_c$ then $x_0 = \alpha_\Phi(s_0)$, else $x_0 = s_0$,

- $T_x \subseteq S_x \times S_x$ is a set of transitions. More specifically, it consists of concrete transitions $T_x^{c \to c}$, abstract transitions $T_x^{\alpha \to \alpha}$, transitions between concrete and abstract states $T_x^{c \to \alpha}$, and transitions between abstract and concrete states $T_x^{\alpha \to c}$,

- $L_x : S_x \to 2^{AP}$ is a labeling function, where $L_x(x) = \{p \in AP \mid x \models p\}$

Figure 4.2 (d) represents the mixed model. We assume that program location 1 and program location 5 belong to the component and that the rest belong to the environment.

We write $s \xrightarrow{t} s'$ to indicate that there is a transition t between states s and s'. We write $s \to s'$ to denote a transition between s and s' when the context is obvious. If state s reaches s' through zero or more transitions, we write $s \to^* s'$ and say s' is reachable from s.

Path A path $\pi = s_0 \ldots s_n$ in a mixed model is a finite sequence of states such that $(s_i, s_{i+1}) \in T_x$ for all $i \in 0 \ldots n-1$. We use $\pi_{s_i}^{s_n}$ to denote that there is a path between s_i and s_n. Paths which start in the abstract component, enter the concrete environment and return to the abstract component will need special treatment later when defining verification properties. We call them border paths. For example, in

Figure 4.2(d), these five states compose a border path. we use $a_1 \xrightarrow{\alpha c^+ \alpha} a_2$ to indicate border path where $\xrightarrow{\alpha c^+ \alpha}$ represents the path starts with an abstract state, goes through one or more concrete states and ends at an abstract state.

For two paths $\pi_{x_0}^{x_n}$ and $\pi_{y_0}^{y_n}$, $\pi_{x_0}^{x_n} = \pi_{y_0}^{y_n}$ indicates that these two paths follow the same transitions, and $\pi_{x_0}^{x_n} \neq \pi_{y_0}^{y_n}$ indicates that one or more transitions in each path are different.

Under approximation abstracts away some behaviors of the concrete system. If there is no loss of precision in the behavioral model due to the abstraction for each transition relation, we say the abstraction is exact. Checking the exactness of the abstraction of the transition relation is the biggest challenge we face using mixed abstract and concrete models. If each transition between abstract states is exact with respect to the corresponding transition between concrete states, then a bisimulation relation can be established between the abstract and concrete models and CTL* properties will be preserved in the abstract model. When the transition resides entirely in the abstract part of the system, this check is done easily using weakest precondition [21].

However, the check is more complicated when we must check the exactness of a border transition. In order to explore all possible behaviors in such border cases, we need to reason about the accumulated effect of the intermediate transitions on the border states. We will use a method similar to symbolic simulation to capture the meaning rather than just the effect of concrete transitions on the border states.

Symbolic transitions A symbolic transition is an accumulation of the effects of a sequence of transitions on the state variables. For example, if we have a series of transitions $x = x + 1$, $x = 2x$, and $x = x^2$ in a path, then symbolic transition for these transitions is $x = (2(x + 1))^2$. Given a path $\pi_{x_0}^{x_n}$, the notation (x_0, x_n) denotes the symbolic transition from x_0 to x_n. In other words,

$$(x_0, x_n) = (x_0, x_1) \circ (x_1, x_2) \circ \cdots \circ (x_{n-1}, x_n)$$

where

$$0 \leq i \leq n-1 \text{ and } (x_i, x_{i+1}) \in T_x$$

and transition composition, $\circ$, denotes the sequential application of a transition to the result of the previous transition.

As in [21], the precision check is based on weakest preconditions. We must, however, cope with weakest preconditions defined over border transitions as well as component transitions. In either case, the basic definition is the same.

Weakest Precondition The weakest precondition, in terms of a transition and predicate ϕ, calculates a predicate that must be satisfied before a transition executes so that ϕ is satisfied after the transition. We use $wp(\phi, i)$ to express weakest precondition of transition i in terms of a predicate ϕ.

The $wp(\phi, i) = \phi[x_{new}/x_{old}]$ where x_{new} is the new value of variable x in the predicate ϕ after a transition i, and x_{old} is the old value of variable x before the transition i.

4.4 Algorithm

In this section we describe the state enumeration algorithm for generating states and test cases in abstract components in the context of a concrete environment. We have omitted property checking and have assumed that all predicates are global in order to simplify the presentation.

Figure 4.3 shows the algorithm. It starts by calling the procedure **init** which takes a program *prog* as input. In line 2, Φ is initialized with all of the guards in the program. When a component is abstracted through predicate abstraction, the predicates in Φ are the initial set of predicates used to create abstract states during the first iteration of the algorithm. Φ_{new} stores the new predicates to refine the abstraction after each iteration and is initialized to the empty set in line 3. The **environment** function returns the first state which lies in the component.

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

The **component** function explores the state space by storing abstract states in the hash table, storing concrete states on the stack, and executing transitions which leave the component without storing states. If the next transition exits the component, then we store the current state in the input set at line 20 and execute instructions in the environment until the program control returns to the component at line 21. The next state is generated by applying the current transition to the current state, line 22, or in the **environment** function at line 28. State exploration then continues by pushing the next state into the stack in line 23.

In the **environment** function, when the next instruction is in the environment, we simply execute it at line 28. Otherwise, we store the first state that lies in the component as the expected test output at line 31 and return it at line 33. Symbolic simulation of the border transition is performed in this function. The symbolic execution is used to refine the border transition. The test output and test input are paired and added to the test set at line 32.

```
proc init($prog$)
  $\Phi$ := Guards($prog$)
  $\Phi_{new}$ := $\emptyset$
  do
    $\Phi$ := $\Phi \cup \Phi_{new}$
    if $start\_instr$ $\in$ environment then
      $start\_state$ = environment($start\_instr$, $start\_state$)
    push($start\_state$)
    component()
  while $\Phi_{new} \not\subseteq \Phi$

proc component()
  while size(stack) != 0
      $cur\_state$ = top(stack)
      $\alpha$ = abstract ($cur\_state$)
      if ($\alpha \notin$ hash table)
        insert $\alpha$ into hash table
        $cur\_inst$ = transition($cur\_state$)
        if ($cur\_inst \notin comp$)
          insert $next\_state$ into input
          $next\_state$ = environment ($cur\_inst$, $cur\_state$)
        else $next\_state$ = $cur\_inst$($cur\_state$)
        push ($next\_state$)
      else pop(stack)

proc environment($inst$, $state$)
  do
    $next\_state$ = $inst$($state$)
    $inst$ = transition ($next\_state$)
  while ($inst$ $\in$ environment)
  insert $next\_state$ into output
  insert (input, output) to test set
  return ($inst$($next\_state$))
```

Figure 4.3: State enumeration algorithm that combines under-approximation with concrete execution.

4.5 Theorems

In this section, we analyze the mixed model we defined in section 2.3. First we discuss how to check the precision of the abstraction used in the component. Then we show how the under approximation abstraction scheme can be used to find test cases. After that, we discuss termination detection and property preservation in the mixed model.

4.5.1 Precision

The central problem in our mixed computational model is determining whether or not the abstraction is precise, or, in other words, creates a bisimulation between the partially abstract and the original systems. And this problem is particularly difficult for border transitions which span the boundary between the abstracted component and concrete environment.

This section focuses on reasoning about precision in predicate abstraction with a theorem prover [21]. There are two ways in which the mixed model can lose precision due to abstraction in the component. We illustrate each case with a simple example then discuss how to detect and recover lost precision. Proofs are given for each precision-recovery technique.

Figure 4.4 shows the first case. In this case, precision is lost because two concrete states are abstracted into the same abstract state, but exhibit different behaviors when they go through the concrete environment and return to the component. In (a), a simple C program is given. Here, we assume program locations 1 and 5 are in the component under test, and that program locations $2 \rightarrow 4$ are in the environment. We also assume that the initial value of i is 2, and that the initial predicate for abstraction is $i \geq 1$. Graph (b) represents the corresponding system. Solid circles represent concrete states and dotted circles represent abstract states. In each state, a small letter represents the name of the concrete state, and the number gives the value of i in that state. Capitol letters denote abstract states.

In start state a, i has the initial value of 2. Since program location 1 is in the component, we abstract the current state to abstract state A. Then program control enters the concrete environment. At this point, we execute the program through states b, c, and d until program control returns to the component at program location 5. Now in the current state e, the value of i becomes 1, which also satisfies the predicate. Therefore, state e is abstracted into abstract state C since PC values make it different than a. Next, program control returns to the beginning of the loop. Since the current state f has the same program counter value as state a and satisfies the same predicate, state exploration will stop at f. However, if we had allowed the concrete execution of f, then we would have reached state j, in which the value of i becomes 0. Since this state does not satisfy the predicate, it is abstracted into a different abstract state B, and this behavior will be missed.

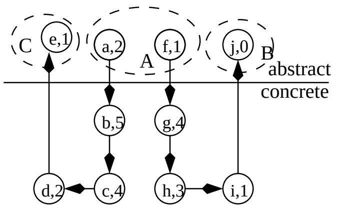
To explore both state e and j, a refinement is needed to differentiate state a and f. We perform refinement by checking the weakest precondition of the accumulated effect of the concrete transitions in the border path. In this example, we will take the accumulated behaviors of $i = i{+}3, i = i{-}1, i = i{-}2, i = i{-}1$, that is $i = i{+}3{-}1{-}2{-}1$, which will be $i = i - 1$. Then we check if the current predicate implies the weakest precondition of this accumulated transition. In other words, we check if $i \geq 1 \Rightarrow$ wp($i \geq 1$, $i = i - 1$), that is equivalent to checking if $i \geq 1 \Rightarrow i - 1 \geq 1$. Since this is false, we will add $i - 1 \geq 1$ to the predicate set to refine the system. Now states a and f are differentiated under the abstraction.

For simplicity, given a set of predicates Φ and a concrete state s, we sometimes write $\alpha_{\Phi}(s)$ to denote the conjunction of all predicates in the set Φ which evaluate to true together with the negation of remaining predicates in Φ which evaluate to false for state s.

Theorem 1 states that doing abstraction refinement on symbolic representations of transition sequences preserves the behavior of paths through boundary states.

```
1   while (i ≥ 1){
2       i = i + 3
3       i = i − 1;
4       i = i − 2;
5       i = i − 1;
6   }
```

(a)



(b)

Figure 4.4: Loss of precision in a simple C program

Theorem 1. *: Let Φ denote the set of predicates to abstract the system. Given two paths $\pi_{x_0}^{x_n}$ and $\pi_{y_0}^{y_n}$, such that $\pi_{x_0}^{x_n} = \pi_{y_0}^{y_n}$, $\alpha_\Phi(x_0) = \alpha_\Phi(y_0)$, but $\alpha_\Phi(x_n) \neq \alpha_\Phi(y_n)$, let $\Phi' = \Phi \cup \{wp(\alpha_\Phi(x_n), (x_0, x_n))\}$ then $\alpha_{\Phi'}(x_0) \neq \alpha_{\Phi'}(y_0)$*

Proof. Doing symbolic simulation with the transitions in the border path is equivalent to checking weakest precondition of the transitions in the border path in reverse order. since $\alpha_\Phi(x_n) \neq \alpha_\Phi(y_n)$, the $wp(\alpha_\Phi(x_n), (x_{n-1}, x_n))$ is a predicate that is able to differentiate state x_{n-1} and y_{n-1}, which is also equivalent to one step of symbolic simulation according to the definition of the weakest precondition. Then, this weakest precondition is symbolically substituted to produce the next level of weakest precondition following the current transition. This newly produced weakest precondition can differentiate state x_{n-2} and y_{n-2} and so on. This is repeated until we obtain a predicate which differentiates x_0 and y_0.

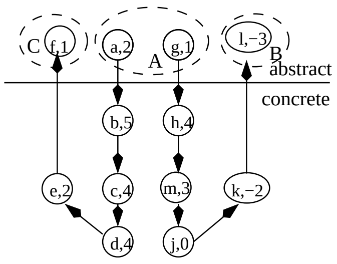
□

The second case in which precision is lost occurs when the execution path branches in the environment. If the execution path branches in the environment, then the branch guard is propagated to the predicate set.

This is shown in the graph in Figure 4.5. In (a), we assume program locations 1 and 6 are in the component and that the rest are in the environment. All other assumptions are the same as Figure 4.4.

```
while (i ≥ 1){
  i = i + 3
  i = i − 1;
  if (i == 3) i = 0;
  i = i − 2;
  i = i − 1;
}
```

(a)



(b)

Figure 4.5: Loss of precision in a C program with a branch

In graph (b) of Figure 4.5, border states a and g pass through the same transitions $i = i + 3$ and $i = i - 1$, reach states c and m, then separate into different transitions, and result in different abstract states C and B when they re-enter the component. As in the above example, g is abstracted into the same abstract state after the while loop iterates once. The state space exploration algorithm will backtrack and miss the behavior of abstract state B.

The intuition of our approach is to push the guards that distinguish c and m upwards into the path to generate predicates that distinguish a and g. In this example, we use the predicate $i == 3$. First, we create a symbolic transition for the series of transitions before the branch. In this case it will be symbolically simulating $i = i + 3$ and $i = i - 1$, which will be $i = i + 3 - 1$, which is simplified to $i = i + 2$. Then we check if $i \geq 1 \Rightarrow$ wp $(i == 3, i = i + 2)$. This is equivalent to checking if $i \geq 1 \Rightarrow i + 2 == 3$. Since this implication does not hold, we add $i == 1$ to the predicate set to refine the abstract system in the next iteration. Now, state a and g will be abstracted into different abstract state because of the newly added predicate.

Theorem 2. *: Let Φ denote the set of predicates to abstract the system. Given two path $\pi_{x_0}^{x_n}$ and $\pi_{y_0}^{y_m}$, such that*

$$\alpha_\Phi(x_0) = \alpha_\Phi(y_0), \alpha_\Phi(x_n) \neq \alpha_\Phi(y_m)$$
$$x_0 \rightarrow^* x_{i-1} \xrightarrow{t_x} x_i \rightarrow^* x_n, y_0 \rightarrow^* y_{i-1} \xrightarrow{t_y} y_j \rightarrow^* y_m,$$
$$t_x \neq t_y, \pi_{x_0}^{x_{i-1}} = \pi_{y_0}^{y_{i-1}}, \text{ and } \pi_{x_0}^{x_n} \neq \pi_{y_0}^{y_m}$$

let $\Phi' = \Phi \cup \{wp(\alpha_{\Phi''}(x_{i-1}), (x_0, x_{i-1}))\}$

in which Φ'' includes a set of guards that differentiate x_{i-1} and y_{i-1}, then $\alpha_{\Phi'}(x_0) \neq \alpha_{\Phi'}(y_0)$.

Proof. Proof is similar to theorem 1. Since Φ'' includes a set of guards that distinguish x_{i-1} and y_{i-1}, we have that $\alpha_{\Phi''}(x_{i-1}) \neq \alpha_{\Phi''}(y_{i-1})$. We propagate this predicate up towards the path until we get a predicate that differentiates x_0 and y_0. $\square$

The next theorem says that if the abstraction is exact in terms of the symbolically simulated transitions which lie outside the component, then the abstraction includes all the possible behaviors of the states at the border of the component.

Theorem 3. *Given a series of transitions $x_0 \rightarrow x_1 \rightarrow \cdots \rightarrow x_n$, if the abstraction is exact for a symbolically represented transition (x_0, x_n) using a set of predicates Φ, then the abstraction captures all effects of concrete behaviors starting at location 0 and ending at program location n in terms of the predicates Φ.*

Proof. This theorem follows from theorem 1. Suppose $\alpha_\Phi(x_0)$ is an abstract state produced in program location 0, and $\alpha_\Phi(x_n)$ is an abstract state produced in program location n. The concrete transition $x_0 \rightarrow x_1 \rightarrow \cdots \rightarrow x_n$ is replaced by symbolic transition (x_0, x_n). States x_0 and x_n can be abstracted into existing abstract states or new abstract states. There are four different possibilities. First, both $\alpha_\Phi(x_0)$ and $\alpha_\Phi(x_n)$ are existing abstract states. Second, $\alpha_\Phi(x_0)$ is an existing abstract state and $\alpha_\Phi(x_n)$ is a new abstract state. Third, $\alpha_\Phi(x_0)$ is a new abstract state and $\alpha_\Phi(x_n)$ is an existing abstract state. Fourth, both $\alpha_\Phi(x_0)$ and $\alpha_\Phi(x_n)$ are new abstract states. In the first case, since both states are existing states, no precision is lost. The second case is impossible since we assume the abstraction is exact. In the third case, the new state is generated in program location 0 and the result of execution is an existing state at program location n, which does not lose precision. In the forth case, a new abstract state is generated in program location 0, which goes to a new abstract state at program location n, which also does not lose precision. Therefore, the theorem holds for all possible cases. □

4.5.2 Under-approximation

Checking precision for border states is easier if we consider only a prefix of the sequence of transitions in the concrete environment because there are less transitions to consider in the symbolic simulation. The precision of the component abstraction will increase as the number of concrete transitions considered increases. When we consider all the transitions, and the verification algorithm terminates, we have an exact abstraction which creates a bisimulation between the program and its abstraction. The following theorem precisely states these claims.

The next theorem states we can vary the precision of the abstraction by varying the number of concrete transitions used to build the symbolic representation of the surrounding software.

Theorem 4. *Suppose $x_0 \overset{\alpha c^+ \alpha}{\to} x_n$, then the abstract component generated by doing abstraction refinement in terms of (x_0, x_i) for the predicate set Φ_i, with $0 < i < n$, is an under-approximation of the original component.*

Proof. All states and transitions visited during model checking are concrete states and generated using concrete transitions from concrete software. Since $0 < i < n$, the exactness check will include concrete behaviors up to the instruction at program location i. However, program behaviors between the instructions at locations i and n

are ignored in the precision check. This means that some execution paths that split between locations i and n may be missed. Therefore, the abstraction scheme we use results in an under-approximation and we can increase precision by including more of the environment in the component. □

If we treat each of the border transitions as one single transition by symbolically simulating it, then we will get a system that is presented in [21] and therefore we get a bisimilar system when the algorithm terminates as shown in [21]. Iteratively refining and symbolically simulating more of the SUT increases the precision of the model and supports the generation of increasingly complete sets of test cases.

4.5.3 Termination and Property Preservation

Termination is difficult to detect when execution leaves the component and never returns. This can occur when the program enters an infinite loop in the environment or when the program actually terminates in the environment. We can not, in general, detect the first case and we simply add an "exit" marker to identify the second case.

4.6 Conclusion and Future Work

We have presented an approach to generating test cases for a new environment into which an existing component will be deployed. Our approach is novel integration of model based testing with model checking and abstraction. We automatically obtain a mixed model of an abstract component within a concrete environment from the component source code and the environment in which it is currently deployed. A relatively complete set of test cases are generated from this model utilizing the ability of model checker to exhaustively explore a system. The model also can be iteratively refined by an abstraction refinement schemes to provide more complete test cases.

Avenues for future work include evaluating the idea in real world software and automatic methods for extracting components.

Chapter 5

Conclusion and Future Work

This dissertation presents a novel approach to verifying component based software by model checking abstract components inside concrete software environments. By abstracting the components under test, we allow the model checker to verify complex components. By simply executing the concrete environment, we avoid analyzing the environment, and allow complex data structures, pointers, and library calls.

The main contribution of this work is to allow the model checker to verify complex components in complex environments that are otherwise impossible to verify directly. Other contributions include generating test cases for complex systems and extending SPIN to verify concurrent C programs which use a subset of pthread libraries.

The work is done in three steps. First, we implement the algorithm in SPIN to model check sequential C programs that include pointers, libraries and complex data structures. SPIN fails to model check these test cases directly due to limited memory, but it successfully detects errors using our approach. Second, we extend the implementation to model check threaded programs, which are difficult to verify by either model checking or testing alone due to their non deterministic behavior. Our approach is able to verify and find errors on these test cases. Third, we discuss how to generate test cases for the component based software based on an abstraction refinement process.

This work opens several avenues for future work. First, an extension to the verification tool can be made to support more features. Automatic extraction of components from the system will be an interesting topic. Given a system, the tool should be able to recognize components under test and environment boundaries and invoke the model checker when it is necessary. Currently, we simply consider a collection of functions as a component and consider the rest as the environment. We also can add more features to SPIN to support all pthread libraries. Since SPIN is designed to handle non-determinism, it is appropriate to add more features to PROMELA to support all functionality of pthread libraries. This work currently supports a subset of pthread libraries that are most commonly used.

Third, a more general benchmark should be developed to evaluate the tool. Currently the evaluation is done on several concurrent programs downloaded from the Internet. It will be interesting to examine how the tool behaves with component based software on an industrial scale.

Appendices

Appendix A

Additional Theorems

This chapter presents additional proofs related to this work. The following theorem establishes the relationship between sets of errors found when using symbolic representations built from different numbers of steps through the concrete environment.

Theorem 5. *Suppose* $x_0 \overset{\alpha c^+ \alpha}{\rightarrow} x_n$*, and suppose* E_i *is the set of errors detected in the abstract component by doing refinement in terms of* (x_0, x_i)*, and* E_{i+1} *is the set of errors detected in the abstract component by doing abstraction refinement in terms of* (x_0, x_{i+1})*, where* $i < i + 1 < n$*, then,*

$$E_i \subseteq E_i \cup E_{i+1} \textit{ and } \cup_0^n E_i = E_n \textit{ but } E_i \not\subseteq E_{i+1}$$

Proof. $E_i \subseteq E_i \cup E_{i+1}$ is obvious. However, it is not the case that $E_i \subseteq E_{i+1}$. For example, if the accumulative transition through i steps is $x = x - 1$ and the next transition is $x = x + 1$, then the accumulative transition through $i + 1$ steps is $x = x$. If the complete accumulative transition is $x = x - 1$, then the model built using i steps is more precise than the model using $i + 1$ steps. Since the accumulative transition (x_0, x_n) is the only complete model of concrete program behavior, the errors found in the nth step include all the errors. □

We have dealt with transitions that enter and leave the component in the previous theorems. We now have the foundation on which to describe our model in terms of the one described in [21] and give a slightly modified version of the theorems presented in [21].

Theorem 6 states that if the algorithm terminates, we either find an error or prove that the system is error-free.

Theorem 6. $\forall x_0.\ x_0 \overset{\alpha c^+ \alpha}{\rightarrow} x_n$*, if we do abstraction refinement in terms of* (x_0, x_n) *and the algorithm terminates with errors, then the errors correspond to real errors. If the algorithm terminates without detecting any error, then the component is error free.*

Proof. The algorithm terminates only when an error is found or the abstraction is exact with respect to all transitions and symbolically represented transitions. As shown in Theorem 4, if an error is found, it will be a feasible error because we only explore actual concrete states and actual concrete transitions. And from theorem 3, we know that we can treat complete symbolically simulated transitions through the concrete software as a single transition because the abstraction does not lose any precision relative to such transitions for paths which start at the start location and end at the end location. Also, behaviors from the concrete software are ignored during verification. If we treat each of the symbolically simulated transitions as one single transition, then we will get a system that is presented in [21] and therefore all of the theorems presented there also apply to our system. □

Theorem 7 describes the cases in which the abstraction computation may not terminate.

Theorem 7. *If the algorithm does not terminate, there are two possibilities.*

1. *The concrete system includes infinite behaviors in which case we don't know anything about the component, or,*

2. *if the algorithm does not terminate because of infinite behavior in the component, then the algorithm will eventually generate a structure which is bisimilar*

to the original component although the algorithm is not able to detect such a bisimulation.

Proof. 1. The first statement holds due to fundamental limits of computability. In particular, determining if the concrete software terminates or not is the halting problem.

2. The theorem in [21] applies directly.

□

In theory, when the mixed model of abstract component and concrete environment is bisimlar to the original concrete system, CTL* properties are preserved. However, in practice, this detection of bisimilarity is hard since symbolic simulation is expensive and theorem prover might fail to check if the current predicates implies the weakest pre-condition of the accumulated behaviors of the environment. In this case, we can gain faster verification speed and faster error discovery by sacrificing some preciseness. We can completely ignore the environment and skip the refinement check on the environment transitions. Safety properties are preserved and liveness properties are preserved under certain conditions.

If the environment does not change the value of the variables we model check, then LTL_x properties are still preserved when we do not do a refinement check in the environment transitions. This is because the two paths that generated by including the environment and skipping the environment is stuttering equivalent in this case. For example, suppose there is a path $x \xrightarrow{\alpha c^+ \alpha} c$, where component state x and c are connected by a series of environment states and transitions, and we are interested in variable i which has a value 0 in state x and value 1 in state c. If the value of variable i is unchanged all along the environment, in other word, i is either 0 all along the environment or 1 all along the environment, then $x \xrightarrow{\alpha c^+ \alpha} c$ and $x \rightarrow c$ are stuttering equivalent. See [8] for a discussion of the proof that LTL_x properties are preserved under stuttering. Our empirical analysis in section 3.5 is done by skipping the environment transition in the refinement check.

Appendix B

Pthread Library Function to PROMELA Translation

This Chapter describes how we translate some of the pthread library functions to PROMELA.

Figure B.1 shows how PROMELA simulates *pthread_join*. Sending signal to the channel is omitted for simplicity.

```
C statement and CIL translation:
  pthread_join(thread[t], &status);

PROMELA translation:
  atomic {
    c_code {now.tid = Pmain → thread[Pmain → t ];};
    Gchan[tid] ? 1;
  }
```

Figure B.1: Thread join

Figure B.2 shows this approach. *if* statement in line 8 blocks until the condition holds and then executes the entire statement in one atomic step.

Figure B.3 and B.4 demonstrate how conditional variable is translated in PROMELA. In Figure B.3, the C and CIL version of using conditional variable is given. In this example, thread *watch_count* blocks until *count* reaches *COUNT_LIMIT*. Thread *inc_count* signals when *count* reaches the *COUNT_LIMIT*. Figure B.4 shows the matching PROMELA code. The conditional variable is a global variable initialized to 0 when the program starts. When *count* is less than *COUNT_LIMIT* in line 11, we unlock the mutex at line 13 so that other threads that

```
C statement and CIL translation:
  pthread_mutex_lock(&mutexsum);
  pthread_mutex_unlock(&mutexsum);

PROMELA translation:
  atomic {
    if
      ::c_expr{ *(& mutexsum) > 0} → c_code {*(& mutexsum) = *(& mutexsum) - 1;};
    fi;
  };
  c_code {*(& mutexsum) = *(& mutexsum) + 1;};
```

Figure B.2: C and PROMELA mutex statements

are blocking on this mutex can resume their executions. Then we block *watch_count* at line 14 by waiting the conditional variable to become 1. When *count* reaches *COUNT_LIMIT*, thread *inc_count* signals by turning the value of conditional variable to 1 at line 35. Then *watch_count* unblocks and turn the value of conditional variable to be 0 in line 14.

C statement:

```
void *watch_count(void *t)
{
   pthread_mutex_lock (&count_mutex);
   if (count < COUNT_LIMIT) {
      pthread_cond_wait(&count_threshold_cv, &count_mutex);
   }
   pthread_mutex_unlock(&count_mutex);
}
void *inc_count(void *t)
{
   pthread_mutex_lock (&count_mutex);
   if (count == COUNT_LIMIT) {
      pthread_cond_signal(&count_threshold_cv);
   }
   pthread_mutex_unlock(&count_mutex);
}
```

CIL translation:

```
void *watch_count(void *t)
{
   pthread_mutex_lock (&count_mutex);
   if (count < COUNT_LIMIT) {
      pthread_cond_wait((pthread_cond_t * __restrict )(& count_threshold_cv),
            (pthread_mutex_t * __restrict )(& count_mutex));
   }
   pthread_mutex_unlock(&count_mutex);
}
void *inc_count(void *t)
{
   pthread_mutex_lock (&count_mutex);
   if (count == COUNT_LIMIT) {
      pthread_cond_signal(&count_threshold_cv);
   }
   pthread_mutex_unlock(&count_mutex);
}
```

Figure B.3: C and CIL example of using conditional varialbes

```
PROMELA translation:
  proctype watch_count( int threadID)
  {
    atomic {
      if
      ::c_expr{ *(& count_mutex) > 0} →
          c_code{*(& count_mutex) = *(& count_mutex) - 1;};
      fi;
    };
    if
    ::c_expr {count < COUNT_LIMIT} →
        atomic{
          c_code{ *(& count_mutex) = *(& count_mutex) + 1;};
                c_expr {*(& count_threshold_cv) == 1} → c_code{*(& count_threshold_cv) = 0}
        };
        atomic{
          if
          ::c_expr{ *(& count_mutex) > 0 } →
              c_code{*(& count_mutex) = *(& count_mutex) - 1;};
          fi;
        };
    ::else → skip;
    fi;
    c_code{*(& count_mutex) = *(& count_mutex) + 1;};
  }
  proctype inc_count( int threadID)
  {
    atomic {
      if
      ::c_expr{ *(& count_mutex) > 0} →
          c_code{*(& count_mutex) = *(& count_mutex) - 1;};
      fi;
    };
    if
   ::c_expr {count == COUNT_LIMIT} → c_code {*(& count_threshold_cv) = 1;};
    ::else → skip;
    fi;
    c_code{*(& count_mutex) = *(& count_mutex) + 1;};
  }
```

Figure B.4: PROMELA translation of conditional variables

Bibliography

[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 and Michael D. Jones. Test case generation using model checking for software components deployed into new environments. In *5th Workshop on Advances in Model Based Testing (A-MOST)*, Denver, CO, 2009.

[4] Tonglaga Bao and Mike Jones. Model checking abstract components within concrete software environments. In *15th International SPIN Workshop on Model Checking of Software (SPIN)*, pages 42–59, Los Angeles, CA, 2008.

[5] Tomas Barros, Ludovic Henrio, and Eric Madelaine. Verification of distributed hierarchical components. In *2nd International Workshop on Formal Aspects of Component Software (FACS)*, pages 41–55, Macao, China, 2005.

[6] 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.

[7] Ian Craggs, Manolis Sardis, and Thierry Heuillard. Agedis case studies: Model-based testing in industry. In *1st European Conference on Model-Driven Software Engineering (ECMDSE)*, pages 106–117, Nuremberg, Germany, 2003.

[8] Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*, pages 146–154. The MIT Press, 2002.

[9] Galit Friedman, Alan Hartman, Kenneth Nagin, and Tomer Shiran. Projected state machine coverage for software testing. In *international symposium on Software testing and analysis (ISSTA)*, pages 134–143, New York, NY, 2002.

[10] Patrice Godefroid. Model checking for programming languages using verisoft. In *24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 174–186, New York, NY, USA, 1997.

[11] Gerard J. Holzmann. An analysis of bitstate hashing. In *15th International Conference on Protocol Specification, Testing, and Verification (PSTV)*, pages 301–314, Warsaw, Poland, 1995.

[12] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[13] 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.

[14] Gerard J. Holzmann and Theo C. Ruys. Effective bug hunting with spin and modex. In *12th International SPIN Workshop on Model Checking of Software (SPIN)*, pages 24–24, San Francisco, CA, 2005.

[15] Pavel Jezek, Jan Kofron, and Frantisek Plasil. Model checking of component behavior specification: A real life experience. In *2nd International Workshop on Formal Aspects of Component Software (FACS)*, pages 197–210, Macao, China, 2005.

[16] 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.

[17] 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.

[18] Madan Musuvathi. Systematic concurrency testing using chess. In *6th workshop on Parallel and distributed systems: Testing, Analysis, and Debugging (PADTAD)*, pages 1–1, New York, NY, 2008.

[19] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of c programs. In *11th International Conference on Compiler Construction*, pages 213–228, London, United Kingdom, 2002.

[20] Corina S. Pasareanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael R. Lowry, Suzette Person, and Mark Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 15–26, Seattle, WA, 2008.

[21] 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.

[22] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-13)*, pages 263–272, New York, NY, USA, 2005.

[23] 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.

[24] Anna Zaks and Rajeev Joshi. Verifying multi-threaded c programs with spin. In *15th International SPIN Workshop on Model Checking of Software (SPIN)*, pages 325–342, Los Angeles, CA, 2008.