MODELING CONCURRENCY: EXTENDING SMACK TO SUPPORT PTHREADS

by

Montgomery Carter

A Senior Thesis Submitted to the Faculty of
The University of Utah
In Partial Fulfillment of the Requirements for the Degree

Bachelor of Computer Science

School of Computing
The University of Utah
May 2015

Approved:

Zvonimir Rakamarić — Date
Supervisor

H. James de St. Germain — Date
Director of Undergraduate Studies
School of Computing

Ross Whitaker — Date
Director
School of Computing

ABSTRACT

SMACK is a static analysis tool for C/C++ programs. Although SMACK is an active project with support from both academia and industry, it currently lacks support for any form of concurrency. The proposed research will extend SMACK to include support for a common subset of the Pthreads API. In addition to extending SMACK, the proposed research will result in an examination of the generic constructs useful for modeling general concurrency within the context of model checking and symbolic execution.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Acknowledgement text here.

CHAPTER 1

INTRODUCTION

Verification of programs using formal methods has long been a promising area whose practical adoption has gone unrealized. Recent advances in computing performance and verification algorithms have made these well-understood formal verification techniques available for real world applications. As practical adoption of formal verification becomes a reality, it has become necessary to create tools that [wc address] the use cases where formal verification will be most advantageous.

One such use case is the verification of concurrent programs. As multi-core systems become more ubiquitous, the parallel programming paradigm is increasingly relevant. Parallel programming presents a unique challenge for developers, as an extra dimension of complexity is introduced. Indeed, concurrent programs can suffer from a host of issues that simply do not apply to the sequential programming paradigm, such as race conditions and deadlocks. As a result, verification tools suited for real world usage should support concurrency.

SMACK is a bounded software verification tool for C/C++ programs, and is the result of a joint effort between the University of Utah, IMDEA Software Institute, and Microsoft Research [3]. With a growing user base and active support from both the academic and industry communities, it is a good candidate for continued development. SMACK, however, currently lacks support for any form of concurrency. [justify selection of pthreads?] As such, though pthreads support is implemented in other formal verification tools, the community benefits from extending SMACK with a model to support abstract interpretation of C/C++ programs that utilize the pthreads library.

SMACK utilizes an intermediate verification language (IVL) called Boogie, which separates the semantic modeling of input source programs from the processes and algorithms involved in verifying modeled input programs. Naturally then, extending SMACK to support additional [wc input source libraries] requires designing models that express the behavior of the [wc source library calls] using the Boogie IVL. Designing such models for

the pthread API requires investigating and understanding the underlying constructs needed for expressing the behavior of concurrent programming paradigms in general. There is little published work discussing the design of the underlying building blocks used for modeling concurrency.

[This paragraph = improved disaster] In this paper, I demonstrate that the complex behavior of the pthread API can be accurately modeled using the very small set of concurrency modeling primitives available in Boogie. First, I generically discuss the process of modeling common concurrency constructs, using my pthread extension to SMACK as an example. Next, I [describe] the actual implementation of support for pthreads in the SMACK toolchain, allowing for the verification of concurrent programs. Finally, I highlight the empirical testing performed on the pthread extension to SMACK which demonstrates the accuracy of the implementation.

[Remaining is leftover stuff - may be useful when revising intro] This is supported by the accuracy seen in the empirical results of initial implementation testing. My proposed research will extend SMACK with a model of a common subset of the pthreads API, enabling verification of C/C++ programs utilizing the Pthreads library. Extending SMACK to provide support for programs using pthreads has not only provided the SMACK community with support for a common concurrency paradigm, but also presents an opportunity to investigate general constructs useful for modeling concurrency. These constructs, which will necessarily result from extending SMACK, will lend themselves to modeling other concurrency libraries like OpenMP and MPI. Further, the resulting constructs should be general enough for use within other verification tools.

CHAPTER 2

RELATED WORK

Related works

CHAPTER 3

BACKGROUND

Developing a full featured software verification tool from end-to-end can be a daunting task. Bounded model checking requires several steps. First, the input source code must be parsed or compiled. The resulting abstract syntax tree (AST) must then be semantically interpreted, and a model built of underlying program semantics. Finally, the modeled program must be represented as an SMT query and passed through an SMT solver. This imposes a large barrier to entry for prototyping a new model checking algorithm, or building a verification tool for a new source language using existing algorithms.

The Boogie intermediate verification language (IVL) was designed by Microsoft Research to alleviate the complexity of modeling new source languages and implementing new model checking algorithms. Boogie IVL separates the task of modeling input program semantics from the task of bounding and checking modeled programs. This allows model checking tools to take Boogie IVL files as input, rather than the input source language. Similarly, front-end tools can model the input program semantics in Boogie IVL rather than being directly integrating with the model checker. By providing a clear, distinct interface between the two tasks, Boogie IVL [allow for modular tool components rather than end-to-end verification tools].

Due to its goal of creating an abstraction between program modeling and model checking, Boogie IVL has been designed to be a very low-level modeling language. It contains support for little more than a typing system, basic arithmetic and boolean expressions & statements, control flow & procedure calls, and verification condition specification [2]. Any more complicated semantics of the source language must be modeled using the basic set of primitives available in Boogie IVL.

Though the low-level nature of Boogie IVL provides the flexibility to support a large variety of models of computation and source languages, it requires that models be created to define the semantics of operations available within the source language and computational model environments. As an example, there is no concept of a heap within Boogie. Because

of this, a memory model must be developed that accurately models the behavior of the heap. A rudimentary model of memory could use a simple, large array of integers to represent the heap, where each element represents a word of memory, and C pointers are simply indices into this array.

[I don't like the phrasing/organization of this next paragraph - I feel it highlights the Boogie verifier too much, and I don't have much to say about the Boogie verifier]

As the goal of Boogie IVL is to modularize model checking verifiers, it shouldn't be surprising that there are several back-end model checking tools that support Boogie IVL programs. The original back-end for Boogie IVL is a tool called Boogie. In addition to being a complete back-end verifier for Boogie IVL programs, it also provides an API for parsing Boogie IVL and interfacing with SMT solvers. [Paragraph intro sentence indicates paragraph will be talking about several verifiers, but only discusses Boogie]

[Where to put this paragraph?] It should be noted that there exists a back-end verifier from Microsoft Research named Boogie. The Boogie program verifier provides similar functionality to the Corral program verifier. Hereafter, "Boogie" will refer to BoogiePL unless otherwise specified.

[Bad paragraph - gets sloppy in last few sentences]Corral is another back-end Boogie IVL program verifier, and is a good example of how Boogie simplifies the implementation of new model checking algorithms. Corral leverages the Boogie verifier API to implement additional model checking algorithms, such as the Lal/Reps sequentialization algorithm [cite this - Reducing Concurrent Analysis Under...]. [... With this and other algorithms,] Corral has put considerable effort toward providing [cutting-edge] support for analyzing concurrent programs. As such, it was the ideal candidate to use as a back-end verifier for the pthreads extension of SMACK.

Though providing state of the art algorithms for checking concurrnt programs, Corral is similar to the Boogie IVL in that it provides only low-level support for modeling concurrency. The Corral program verifier provides an extension to Boogie IVL that includes a very basic set of primitives for modeling concurrent programs. This extension includes the following calls:

- `async call` *func*`(...)` - Asynchronously calls *func* with the parameter list *'...'*
- `corral_atomic_begin()` - Begins an atomic block
- `corral_atomic_end()` - Ends an atomic block
- `corral_getThreadID()` - Returns the ID of the calling thread.

- `corral_getChildThreadID()` - Returns the ID of the thread most recently spawned in the calling procedure.

The behavior prescribed in the pthread specification [cite pthreads specification] is much more complex than these low-level concurrency primitives recognized by Corral. As a result, to provide support for the more complex pthread API, it is necessary to build a model of the pthread API behavior using the primitives provided by Corral.

[Transition from modeling in Boogie to SMACK][major work needed below this point]

Once the behavior of the pthread API has been accurately modeled in Boogie, the front-end [someadjective] component must be extended to translate pthread API calls in the input source into the Boogie IVL model of the pthread calls.

The SMACK project as a whole is an end-to-end C/C++ program verifier. At its core, the SMACK executable is essentially a compiler that takes C/C++ programs and translates them into Boogie IVL [3].

The SMACK toolchain is depicted in Figure 3.1. SMACK as a whole is a front-end for a program verification toolchain that features the Corral program verifier as its core back-end. Input C/C++ programs are given to Clang to compile and link, resulting in an LLVM bytecode output. This is then passed to the SMACK executable, which translates the LLVM bytecode into a Boogie program that models the behavior of the input C/C++ program. The resulting Boogie program is then passed to Corral. Corral converts the Boogie program into an SMT query, which is given to the Z3 SMT solver for evaluation.

SMACK itself is essentially a compiler that takes C/C++ programs and translates them into Boogie IVL [3]. This converted Boogie code is then consumed by a static analysis tool that evaluates verification conditions present in the original source code.

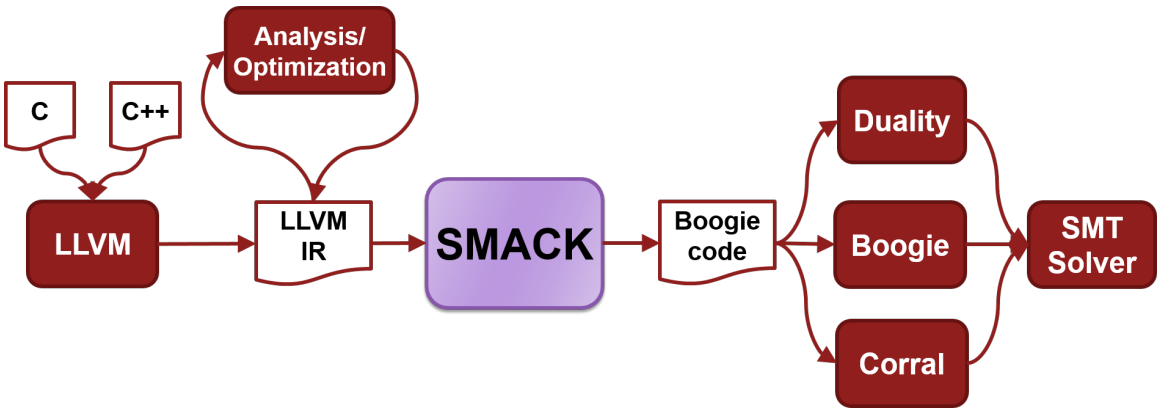

Figure 3.1: SMACK Toolchain

CHAPTER 4

MODEL DESIGN

[This paragraph is very poorly written, and only serves as a general idea of how I'll introduce the chapter]

I'm going to talk about the general concerns and techniques that I've learned for modeling concurrency, using my implementation of pthreads support in SMACK to demonstrate. To facilitate this, this chapter begins with an introduction to the modeling environment that comprises SMACK. Having introduced this, the chapter proceeds to discuss the process of modeling concurrency over the lifecycle of a thread of execution.

4.1 Modeling Environment

[Again, poorly written - just to organize thoughts] As described in Chapter 3, SMACK consumes LLVM IR bytecode, builds a model of the input program, and generates a Boogie IVL file. It turns out that the final pthread implementation is written largely using C code, which then gets converted into the Boogie IVL model of pthreads within each translated input program.

To ease the discussion of the modeling process, I'll briefly introduce Boogie IVL, the Corral concurrency primitives, as well as some helper functions defined at the C/C++ level that inject Boogie statements into the translated Boogie code.

4.1.1 Boogie IVL

The most illustrative way to introduce Boogie IVL may be to show an input C program and the relevant portions of the Boogie output that SMACK returns.

Figure 4.1b shows a reduced version of the Boogie code as translated by SMACK from the input source depicted in Figure 4.1a. Notice that dynamic memory allocation on the heap is modeled with a global array of `int`s [fix color here, or don't use lstinline]. Dereferencing pointers simply becomes indexing into the array representing the heap.

```
void main() {
  int *x, *y;
  x = malloc(sizeof(int));
  y = malloc(sizeof(int));
  *x = 1;
  *y = 2;
  assert(*x == 1);
}
```

(a) Input C Code

```
var $M: [int]int;

procedure main() {
  var $x, $y: int;
  call $x := $malloc(4);
  call $y := $malloc(4);
  $M[$x] := 1;
  $M[$y] := 2;
  assert($M[$x] = 1);
}
```

(b) Boogie Code from SMACK

Figure 4.1: C Input to Boogie Code

4.1.2 Corral Concurrency Primitives

Corral is a back-end solver that verifies Boogie IVL programs. One of Corral's recent development efforts has been to improve support for verifying concurrent programs. To this end, Corral has extended Boogie IVL, and recognizes five concurrency primitives that are not part of the base language. These are [choose whether to list here or in background]:

- `async call` *func*`(...)` - Asynchronously calls *func* with the parameter list *'...'*
- `corral_atomic_begin()` - Begins an atomic block
- `corral_atomic_end()` - Ends an atomic block
- `corral_getThreadID()` - Returns the ID of the calling thread.
- `corral_getChildThreadID()` - Returns the ID of the thread most recently spawned in the calling procedure.

[Fix this god-awful caption]

Figure 4.2 demonstrates the usage of each of these primitives within the context of a complete Boogie program. This program initializes the global variable `$x` as 0, then makes two asynchronous calls to the function `f()` and records the thread ID of each of the spawned threads of execution. In `f()`, each of the spawned threads records their thread IDs. Each thread then starts an atomic block, where it stores its thread ID to the global `$x` and then loads global `$x` into the local `$tmp`. The `assert()` in `f()` should always succeed, since `$x` is stored and loaded within an atomic block.

```
var $x: int;

procedure f()
  modifies $x;
{
  var $tid: int;
  var $tmp: int;
  call $tid := corral_getThreadID();
  call corral_atomic_begin();
  $x := $tid;
  $tmp := $x
  call corral_atomic_end();
  assert($tmp = $tid);
}

procedure main() {
  var $ch1, $ch2: int;
  $x := 0;
  async call f();
  $ch1 := corral_getChildThreadID();
  async call f();
  $ch2 := corral_getChildThreadID();
  assert($x = $ch1 ∨ $x = $ch2);
}
```

Figure 4.2: Corral Concurrency Primitive Usage Example

However, this program contains a bug. Because there is no barrier forcing the parent to wait for the child threads to execute, it is possible that `$x` won't be set to the thread ID of either children by the time `assert()` is called in `main`.

4.1.3 [wc SMACK Boogie Injection]

To enable end-users to more accurately specify verification conditions and model functionality on their own, SMACK has introduced several C functions that allow Boogie [wc models, code?] to be written at the C level. Calls to these functions are intercepted by SMACK, which injects the specified semantics in the translated Boogie code, rather than simply translating the LLVM IR into Boogie. Understanding the behavior of these functions is needed to...

These functions include:

4.2 Thread Creation

4.3 Locking Primitives

These primitives are fairly straightforward, but there are a few gotchas. The biggest one is [I am so tired and writing so poorly] that support for atomic sections is implemented as one single global lock.

Of all sections of code enclosed in an atomic begin/end block, only one can be executing at any given time. This means using the Corral atomic primitives alone, there is no way to create multiple mutual exclusion mechanisms.

4.4 [wc Complex] Synchronization

4.5 Thread Termination

CHAPTER 5

IMPLEMENTATION

5.1 Thread Creation

talk about thread creation

5.2 Locking Primitives

talk about locking primitives

5.3 Thread Termination

REFERENCES

[1] Akash Lal, Shaz Qadeer, and Shuvendu Lahiri. Corral: A solver for reachability modulo theories. In *Computer-Aided Verification (CAV)*, July 2012.

[2] K. Rustan M. Leino. This is boogie 2, 2008.

[3] Zvonimir Rakamarić and Michael Emmi. Smack: Decoupling source language details from verifier implementations. In *Computer Aided Verification*, pages 106–113. Springer, 2014.

[4] Stephen F. Siegel, Matthew B. Dwyer, Ganesh Gopalakrishnan, Ziqing Luo, Zvonimir Rakamaric, Rajeev Thakur, Manchun Zheng, and Timothy K. Zirkel. CIVL: The concurrency Intermediate Verification Language. Technical Report UD-CIS-2014/001, Department of Computer and Information Sciences, University of Delaware, 2014.

[5] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M Kirby. Distributed dynamic partial order reduction based verification of threaded software. In *Model Checking Software*, pages 58–75. Springer, 2007.