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

ABSTRACT . . . . **ii**

LIST OF FIGURES . . . . **iv**

LIST OF TABLES . . . . **v**

ACKNOWLEDGMENTS . . . . **vi**

CHAPTERS

1. INTRODUCTION **1**

2. RELATED WORK **3**

3. BACKGROUND **4**

4. IMPLEMENTATION **7**

4.1 Thread Creation 7
4.2 Locking Primitives 7
4.3 Thread Termination 7

REFERENCES **8**

LIST OF FIGURES

3.1 SMACK Toolchain 6

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 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 static analysis tool that 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. As such, though Pthreads support has been implemented in other formal verification tools, the community will benefit from extending SMACK with a model to support symbolic execution of C/C++ programs that utilize the Pthreads library.

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 not only provides 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 translate to Boogie IVL rather than being directly integrated 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]

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

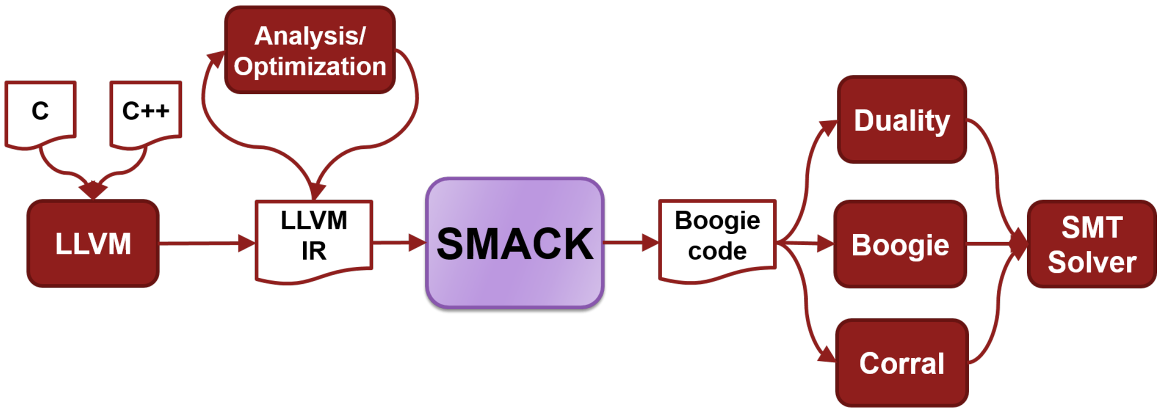

Figure 3.1. SMACK Toolchain

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.

CHAPTER 4

IMPLEMENTATION

4.1 Thread Creation

talk about thread creation

4.2 Locking Primitives

talk about locking primitives

4.3 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.