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]

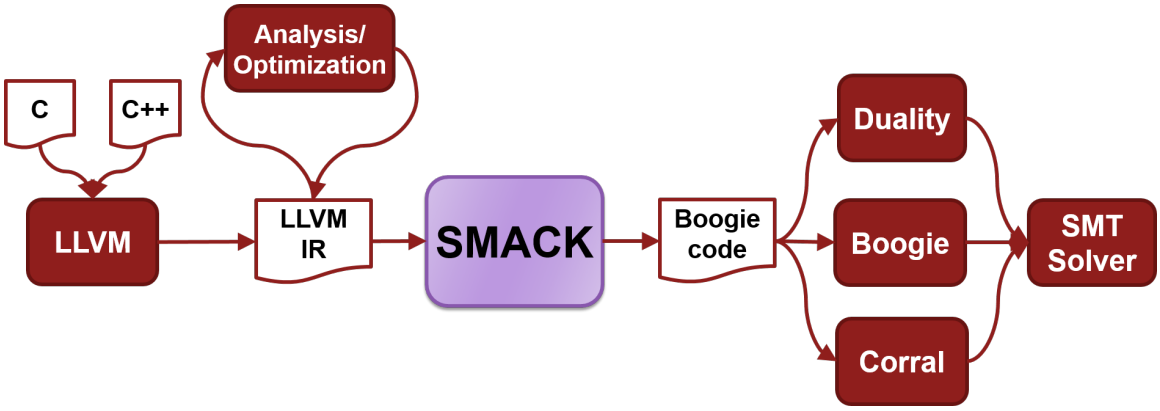
Once the behavior of the pthread API has been accurately modeled in Boogie, the front-end [someadjective] component must be extended to translate pthread API calls in the input source into the Boogie IVL model of the pthread calls.

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

[Good] In this chapter, I discuss my research efforts toward developing a model of the pthread library in SMACK. I begin by introducing details of the modeling framework used to create the pthread library model. Having introduced this, I proceed to discuss the process of modeling concurrency in the pthread library over the lifecycle of a pthread.

4.1 Modeling Framework

[Good] As described in Chapter 3, SMACK consumes LLVM IR bytecode, builds a model of the input program, and generates a Boogie IVL model of the input program. The design of this toolchain leaves us with several levels at which modeling semantics can be introduced. At the C level, external library functions that are otherwise undefined can be given a definition, rather than verifying the whole original source library. At the SMACK translation level, undefined functions can be intercepted so SMACK can perform predefined alterations to the translated Boogie program rather directly being translated from the LLVM IR representation. At the translated Boogie level, additional variables and instructions can be added to model environmental functionality that is not defined at the C level. Finally, there can be additional functionality modeled by the Boogie IVL solver itself.

To provide some context for the discussion of the modeling process, I'll briefly introduce Boogie IVL, and then discuss the mechanisms used for modeling, such as the Corral concurrency primitives and C level Boogie injection.

4.1.1 Boogie IVL

[Good] Perhaps 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. Translating source language programs into Boogie

Figure 4.1: SMACK Translation of C Program

(a) Input C Code

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

(b) Boogie Code from SMACK

```
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);
}
```

IVL programs like this provides a common language on which to perform verification for solvers like Corral, Boogie verifier, and Duality.

4.1.2 Corral Concurrency Primitives

[Good] Corral is a state of the art solver for 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.

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.

Figure 4.2: Corral Concurrency Primitives in Action

```
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);
}
```

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`.

It is this additional support for concurrency that warranted selection of Corral as the solver to target for extending SMACK to support pthreads.

4.1.3 C Level Modeling

[Ok. Could use another pass or two] Because there are several layers at which model semantics can be introduced, SMACK has introduced several special C level functions that don't get directly translated into Boogie code, allowing users to introduce program semantics at the lower levels of the modeling framework by writing code at the C level. This allows environment and library models to be completely written as C input programs, which avoids modifying SMACK's source code directly to implement such models.

An example of this is `malloc`. Rather than having SMACK itself append a memory model and `malloc` definition to each translated Boogie program, these semantics are instead

defined in a header file that gets included from each input program. Clang compiles this included header file, so when SMACK begins translation, the memory model and `malloc` are present in the AST, leaving SMACK with only the responsibility of translating the LLVM IR.

There are several C level functions that SMACK treats specially when seen in the LLVM IR AST. Upon seeing these, SMACK performs special transformations to the Boogie translation, rather than translating the function calls as is. Two of these functions were particularly important for the pthread extension to SMACK.

- `__SMACK_code(char* format, ...)` – Injects the string `format` in the Boogie code.
- `__SMACK_top_decl(char* format, ...)` – Inserts the string `format` as a global declaration

Figure 4.3: Injecting Boogie Code from C

(a) Boogie Injecting C Code

```
void main() {
  __SMACK_top_decl("var $globalVar: int;");
  int *x;
  x = malloc(sizeof(int));
  *x = 1;
  __SMACK_code("$globalVar := @", *x);
  __SMACK_code("@ := $globalVar + 1", *x);
  assert(*x == 2);
}
```

(b) Injected Boogie Translation

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

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

Input to both of these functions is similar to that of C's `printf()` function; "@" symbols in the `format` string are replaced by arguments from the list "...". Figure 4.3 demonstrates the C level usage, and Boogie level translations of both of these functions.

My implementation of pthread support for SMACK performs modeling exclusively at the C code level, using these Boogie injection functions to model environmental state where necessary.

4.2 Thread Creation

[Good] One of the critical tasks in implementing support for pthreads in SMACK was modeling the environment that the operating system provides. For each thread, the OS tracks the thread ID and thread status of each thread spawned within a process. This information is tracked by the operating system in a thread control block (TCB). At any time, a thread must be able to query for its thread ID. Similarly, a parent thread must be able to query the operating system for the running state threads it has spawned.

This section discusses the challenges of modeling the environment presented by the operating system to the pthread library, followed by a description of the model created to implement pthread creation within SMACK.

4.2.1 Modeling the Environment

[Good] Faithfully modeling the environment seen by the pthread library requires addressing a number of bookkeeping issues. These include tracking the running state of each thread, and coordinating the thread ID between the C level code and Corral itself. The critical issue can be illustrated by considering the API call `pthread_join(pthread_t* thr)`. This call blocks the calling thread until the thread specified by `thr` is in the stopped state. Consequently, properly model this function requires that the calling thread must be able to query the environment for the status of the thread referenced by `thr`, in order to allow the calling thread to unblock execution.

Within the context of static model checking, because no code is actually executing, the running state of a thread is simply a bit of program state information – a thread can be marked as waiting, running, or stopped. Thus, tracking thread states requires the ability to reference some bit of state information about the threads. A `pthread_t` object passed into `pthread_join()`, for example, must have some method of referencing this bit of thread state information. It seems natural to store each thread's thread ID in this `pthread_t` object, and use this to look up the thread's state in an array. Indeed, this is precisely how the operating system tracks such information. In addition, such an array must be

globally accessible, because threads must be able to query the state of another thread (to singal `pthread_join()`, for example). To model this, a global Boogie-level array called `$pthreadStatus` is added to the translated Boogie model of input pthread programs, as illustrated in Figure 4.4.

[Figure out better typesetting of these small guys]

Figure 4.4: Declaring Global $pthreadStatus

(a) C Declaration of $pthreadStatus

```
__SMACK_top_decl("var $pthreadStatus: [int][int]int;";)
```

(b) Boogie Translation (as Global Declaration)

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

In order to access thread state from `$pthreadStatus`, a queried thread's thread ID is used to index into the array. This requires that a thread's ID is referenced by its `pthread_t` object. This requires immediately querying Corral for the thread ID of newly spawned thread, and recording this information in its `pthread_t`. This is done from the parent with `corral_GetChildThreadID()`, and from the child with `corral_GetThreadID()`. Now, when `pthread_join()`, for example, queries Corral for the running state of a thread, it can pass in the thread ID to Corral, which will use it to index into `$pthreadStatus`.

However, there is further complication with implementing this array. In Boogie, undefined variables take on a nondeterministic value - that is, Corral assumes they can have any value. Without initializing `$pthreadStatus`, Corral must consider the case that threads start in a *stopped* state. This means that a call to `pthread_join()` immediately following a call to `pthread_create()` could query for the status of the newly spawned thread before the new thread has initially set its status to waiting, and see it is in the stopped state, causing the parent thread to unblock. To address this, `$pthreadStatus` is initialized so that all indices indicate an *uninitialized* state, as illustrated in Figure 4.5. This code gets executed as the first statement in `main()`, to ensure that any thread's running state is defined prior to its use.

With these Boogie level additions to each pthread program's Boogie translation, the operating system environment is sufficiently prepared to enable the modeling of the actual pthread library behavior.

Figure 4.5: Initializing Global $pthreadStatus

(a) C Initialization of $pthreadStatus

```
__SMACK_code("assume (forall i:int :: "
                            "$pthreadStatus[i][0] == "
                            "$pthread_uninitialized);");
```

(b) Boogie Translation

```
assume (forall i:int :: $pthreadStatus[i][0] ==
                        $pthread_uninitialized);
```

4.2.2 Modeling Thread Creation

[Good] Modeling actual pthread library function call behavior is unlocked once the OS environment has been modeled. In the pthread library, the procedure for spawning threads is `pthread_create()`. There are two aspects of `pthread_create()` that require modeling: behavior in the calling thread, and behavior in the newly spawned thread.

In the calling thread, there are two main tasks. For the first task, the new thread must be spawned, and instructed to execute the procedure indicated by the `*__start_routine` function pointer. For the second, SMACK must query Corral for the thread ID of the newly spawned child thread, and record this value in the `pthread_t` struct for future reference. This results in the model defined in the C code seen in Figure 4.6.

Figure 4.6: Model: pthread_create()

```
int pthread_create(pthread_t* __newthread,
                   pthread_attr_t* __attr,
                   void* (*__start_routine) (void *),
                   void* __arg)
{
  __SMACK_code("async call @(@, @, @);", __call_wrapper,
                                         __newthread,
                                         __start_routine,
                                         __arg);

  __SMACK_code("call @ := corral_getChildThreadID();",
               *__newthread);

  return 0;
}
```

In the newly spawned thread, again there is bookkeeping to do, as well as executing the

specified `__start_routine()`. Notice in Figure 4.6 that the function `__call_wrapper()` is the procedure being executed asynchronously, rather than `__start_routine()`. This allows for the child thread to perform bookkeeping before and after calling `__start_routine()`.

As seen in Figure 4.7 the newly spawned thread queries Corral for its thread ID, and then makes a call to `assume()`, which causes Corral to only consider paths through the program where `*__newthread` already contains the thread ID of the child thread – effectively, the new thread "waits" until its parent has set the child thread's ID in `*__newthread`. Once this has occurred, both the parent and the child have a reference to the newly spawned thread's ID, and can use this to index into `$pthreadStatus` to set and query the child thread's running state information.

Once the child and parent can both reference the correct `$pthreadStatus` element, the new thread can proceed to executing the specified `__start_routine()`. The new thread cycles through the *waiting* state to the *running* state, and then makes a synchronous call to the `__start_routine()`. After the `__start_routine()` returns, the thread's running state is finally set to *stopped*, which signals the end of the child thread.

Figure 4.7: Model: __call_wrapper()

```
void __call_wrapper(pthread_t* __newthread,
                    void* (*__start_routine)(void *),
                    void* __arg)
{
  int ctid = __SMACK_nondet();
  __SMACK_code("call @ := corral_getThreadID();", ctid);
  assume(ctid == *__newthread);

  __SMACK_code("$pthreadStatus[@][0] := $pthread_waiting;",
               *__newthread);
  __SMACK_code("$pthreadStatus[@][0] := $pthread_running;",
               *__newthread);

  __start_routine(__arg);

  __SMACK_code("$pthreadStatus[@][0] := $pthread_stopped;",
               *__newthread);
}
```

4.3 Thread Execution

[Good] With the ability to spawn threads, [wc it becomes time to] consider the functionality of the pthread library that pertains to an executing thread. Among this functionality, the most commonly utilized are the synchronization routines.

Corral extends Boogie IVL with two primitives for achieving synchronization between threads: `corral_atomic_begin()` and `corral_atomic_end()`. These calls provide the ability to create critical sections of Boogie IVL code, where only one atomic block can execute at any time. However, these atomic blocks are global, and take no parameters. There is no way to differentiate atomic blocks of unrelated code that should be allowed to run concurrently – this is much like the big kernel lock in Linux. This presents the need to model the more complex synchronization routines of pthreads using Corral's atomic primitives.

4.3.1 Modeling Mutexes

[Good] Mutexes are the most commonly used mechanisms for providing synchronization among threads. In addition, they can be used as building blocks to devise more complex synchronization mechanisms. Modeling mutexes in Corral is relatively straightforward, and more complex synchronization functionality easily builds on this foundation.

Each `pthread_mutex_t` object allocated creates an independent mutual exclusion device. To track the current owner of a mutex and prevent execution of threads that don't hold the lock, the model stores the thread ID of the current mutex owner in the `pthread_mutex_t` struct. This allows `pthread_mutex_lock()` to block until until the mutex is in the *UNLOCKED* state.

This leads to the straightforward model shown in Figure 4.8. The calling thread queries for its thread ID using `pthread_self()` (which simply queries Corral with a call to `corral_GetThreadID()`, and returns the result). The mutex is then checked to ensure it is in the initial state, and that the calling thread is not already the owner of the mutex. With these criteria ensured, the Boogie translation atomically waits for the mutex to enter the *UNLOCKED* state and updates the mutex owner to be the calling thread's ID.

`pthread_mutex_unlock()`, seen in Figure 4.9, releases a mutex, and is essentially the inverse of `pthread_mutex_lock()`; the model ensures the mutex is initialized and the calling thread is the current owner, and atomically sets the mutex as being unlocked.

Figure 4.8: Model: pthread_mutex_lock()

```
int pthread_mutex_lock(pthread_mutex_t* __mutex)
{
  int tid = (int)pthread_self();
  assert(__mutex->init == INITIALIZED);
  assert(__mutex->lock != tid);

  __SMACK_code("call corral_atomic_begin();");
  assume(__mutex->lock == UNLOCKED);
  __mutex->lock = tid;
  __SMACK_code("call corral_atomic_end();");

  return 0;
}
```

Figure 4.9: Model: pthread_mutex_unlock()

```
int pthread_mutex_unlock(pthread_mutex_t* __mutex)
{
  int tid = (int)pthread_self();
  assert(__mutex->init == INITIALIZED);
  assert(__mutex->lock == tid);

  __SMACK_code("call corral_atomic_begin();");
  __mutex->lock = UNLOCKED;
  __SMACK_code("call corral_atomic_end();");

  return 0;
}
```

4.3.2 Complex Synchronization

[Short, but sufficient?] The pthread library also includes a number of more complex synchronization mechanisms. These allow for more efficient runtime behavior, in addition to providing commonly utilized synchronization mechanisms.

`pthread_cond_wait(pthread_cond_t* cond, pthread_mutex_t* mutex)` is one such function that provides more efficient runtime behavior. It blocks the calling thread until both the `cond` predicate is true *and* `mutex` is *UNLOCKED*, at which point the thread is signaled to resume. This avoids having to continually acquire the mutex and evaluate the predicate. `pthread_barrier_wait()` is a good example of a procedure providing expected functionality. It causes calling threads to block until all threads calling `pthread_barrier_wait()` have reached this instruction. These more complex synchro-

nization routines are modeled entirely as C level code simply by using `pthread_mutex_t`'s as building blocks.

4.4 Thread Termination

[Good] As a thread's execution comes to an end, there is a bit of bookkeeping to be done to wrap up the thread's lifecycle. It is primarily this termination handling that required the upfront work of modeling the operating system environment prior to implementing support for thread creation.

There are two main considerations for handling the termination of a thread. The first consideration is the passing of a return value from a terminating thread. The second is blocking execution of a waiting thread, and retrieving the return value of the terminating thread being waited on by the blocking thread.

4.4.1 Modeling Thread Exit and Join

[Could use another pass or two] It is often necessary to pass a return value from a terminating thread to some other thread. For example, with a scatter/gather design pattern, a parent thread will send data to multiple worker threads, which return a computational result back to the parent.

The challenge with achieving this was that this return data has to exist somewhere between the time that the terminating thread exits, and the joining thread retrieves it. This functionality was implemented by extending the definition of the `$pthreadStatus` array to store a tuple, rather than a single value. With this, the model can store a return value passed into `pthread_exit(void* value)` as the second tuple element, in addition to storing thread running state as the first.

The model of `pthread_exit()`, seen in Figure 4.10, gets the calling thread's ID and ensures the thread is still in a running state, saves the return value pointer in `$pthreadStatus`, and finally sets the running to stopped.

With the return value from threads now stored in `$pthreadStatus`, a method for accessing this value is needed. This is done through `pthread_join(pthread_t* thr)`, which blocks the calling thread until the thread referenced by `thr` terminates. The value returned from `pthread_join()` is the pointer passed in to `pthread_exit()` by the terminating thread.

The model designed for `pthread_join()`, shown in Figure 4.11 implements this behavior. A call to `assume()` causes the thread to block until the target thread's running state is *stopped*. Once the terminating thread is stopped, the waiting thread fetches the

Figure 4.10: Model: pthread_exit()

```
void pthread_exit(void *retval)
{
  pthread_t tid = __SMACK_nondet();
  tid = pthread_self();

  __SMACK_code("assert $pthreadStatus[@][0] == "
               "$pthread_running;",
               tid);

  __SMACK_code("$pthreadStatus[@][1] := @;", tid, retval);

  __SMACK_code("$pthreadStatus[@][0] := $pthread_stopped;",
               tid);
}
```

return value pointer from the `$pthreadStatus` array, which is then returned by the call to `pthread_join()`.

Figure 4.11: Model: pthread_join()

```
int pthread_join(pthread_t __th, void **__thread_return)
{
  __SMACK_code("assume $pthreadStatus[@][0] == "
               "$pthread_stopped;",
               __th);

  __SMACK_code("@ := $pthreadStatus[@][1];",
               *__thread_return, __th);

  return 0;
}
```

[Editing Leftovers - may be useful during revision, or elsewhere in document] [NEW:Modeling thread creation is/was the most difficult portion of this project/modeling concurrency. OLD:There are a number of challenging aspects of modeling the thread creation process.] It is at this stage that the overall model of concurrency must be considered and accounted for. For example, concurrent calls to a driver may be driven by hardware interrupts and execute within the kernel, where the process ID is always the same and unimportant to the execution of the driver. Alternatively, multi-threaded user-mode programs often assign different tasks to threads based on their thread ID, in which case tracking thread IDs becomes relevant.

The environment within which concurrent threads of execution run must be modeled prior to thread creation to provide the framework in which threads will execute [bad sentence, idea is set up environment so threads can interact with environment as they run]

[transition]

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.