Systematic Testing of Multithreaded Programs

Derek Bruening, John Chapin
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
{iye,jchapin}@mit.edu

Abstract

We present a practical testing algorithm called ExitBlock that systematically and deterministically finds program errors resulting from unintended timing dependencies. ExitBlock executes a program or a portion of a program on a given input multiple times, enumerating meaningful schedules in order to cover all program behaviors. Previous work on systematic testing focuses on programs whose concurrent elements are processes that run in separate memory spaces and explicitly declare what memory they will be sharing. ExitBlock extends previous approaches to multithreaded programs in which all of memory is potentially shared. A key challenge is to minimize the number of schedules executed while still guaranteeing to cover all behaviors. Our approach relies on the fact that for a program following a mutual-exclusion locking discipline, enumerating possible orders of the synchronized regions of the program covers all possible behaviors of the program. We describe in detail the basic algorithm and extensions to take advantage of read-write dependency information and to detect deadlocks.

1 Introduction

Concurrency is increasingly used in programs of all varieties. However, concurrent programs are more difficult to test than sequential programs. Traditional testing techniques for sequential programs consist of test suites that simply run the target program on different sets of representative inputs. Such tests are likely to cover only a small fraction of the possible execution paths of a concurrent program. This is due to the nondeterministic nature of concurrent programs: merely controlling the program's input is not enough to control the program's behavior. We define a particular program execution's *behavior* to be the set of externally visible actions that the program performs during its execution.

A concurrent program consists of components that execute in parallel. These components are of two varieties: processes, which run in separate memory spaces, and threads, which all share the same memory space. Different orders of the components' actions with respect to one another may result in different program behaviors; any resulting erroneous behaviors are called *timing-dependent errors*. The order of components' actions is often nondeterministic due to asynchronous input or interrupts or simply a nondeterministic *scheduler*. The scheduler is the operating system or library component that determines at run time the actual order of execution, or *schedule*, of the processes or threads.

In this paper we describe a practical algorithm called ExitBlock for enumerating the possible behaviors of a section of a multithreaded program, for a given input. The number of possible schedules of a concurrent program's processes or threads increases exponentially with program size so quickly that testing even very small program sections by executing every schedule is intractable. Behavior-equivalence classes of schedules must be identified and as few members of each class as possible executed. ExitBlock relies on the fact that for a program following a mutual-exclusion locking discipline, enumerating possible orders of the synchronized regions of the program includes at least one schedule from each behavior-equivalence class, covering all possible behaviors of the program. The algorithm does not need a separate model or specification of the program or the source code; it does its testing dynamically using information gathered while running the actual program.

The completeness of the enumeration is defined in terms of assertion checking. If there is some execution of a program on a given input that leads to an assertion failure, ExitBlock guarantees to find that execution. Since any observable program behavior can be differentiated from all other behaviors by assertions, ExitBlock effectively guarantees to execute all behaviors of a program.

We also present a method called ExitBlock-RW that uses read-write dependency information to prune the schedules that ExitBlock considers, and a technique for augmenting both ExitBlock and ExitBlock-RW to detect deadlocks.

We have built a systematic tester that implements ExitBlock and its pruning and deadlock detection extensions for Java application programs. Our tester is implemented in Java as a tool for the Rivet virtual machine [Rivet], a Java virtual machine built by our group. The tester can successfully find errors in programs of four threads with twenty synchronized regions each in a matter of minutes. We demonstrate that the ExitBlock algorithm finds common timing-dependent errors that are often difficult to detect; for example, errors that occur only in certain orderings of modules that are individually correctly synchronized, or using an `if` instead of a `while` to test a condition variable.

The rest of the paper is organized as follows. The next section gives an example of a simple target program, and Section 3 describes related work. Section 4 describes ExitBlock. ExitBlock-RW and other efficiency improvements to ExitBlock are discussed in Section 5; extending ExitBlock

This research was supported in part by the Defense Advanced Research Projects Agency under Grant F30602-97-2-0288. Additional support was provided by an equipment grant from Intel Corporation.

```java
public class SplitSync implements Runnable {
  static class Resource { public int x; }
  static Resource resource = new Resource();

  public static void main(String[] args) {
    new SplitSync();
    new SplitSync();
  }

  public SplitSync() {
    new Thread(this).start();
  }

  /** increments resource.x */
  public void run() {
    int y;
    synchronized (resource) {
      y = resource.x;
    }
    synchronized (resource) {
      // invariant: (resource.x == y)
      resource.x = y + 1;
    }
  }
}
```

Figure 1: Sample Java program `SplitSync` illustrating an error that our algorithm can detect. Although all variable accesses in this program are correctly protected by locks, the program contains a timing-dependent error.

to detect deadlocks is explained in Section 6. Section 7 presents the results of running our tester on a number of sample programs, and Section 8 concludes the paper.

2 Example

A simple Java program exhibiting one sort of concurrency error that ExitBlock can detect is shown in Figure 1. `SplitSync` creates two threads that each wish to increment the field `x` of the variable `resource`. The Java `synchronized` statement limits its body to execute only when the executing thread can obtain the lock specified. Thus only one of the two threads can be inside either of the synchronized statements in the program at one time.

The way in which the threads execute the increment of `resource.x` is roundabout; however, it is representative of a common class of errors in large systems in which the processing of a resource is split between two modules that each correctly protect the resource with a lock. The intention is that no other module can access the resource while these two modules are processing it, but their separate synchronization allows a third module to acquire the resource before it is fully processed.

A traditional test suite might never find this timing-dependent error because the thread scheduler of its Java virtual machine may never perform a thread switch while either thread is between the two synchronized blocks. Even if the bug is found, it may be very difficult to duplicate. Our implementation of ExitBlock finds this error by systematically enumerating schedules of the program.

3 Related Work

There is a large body of work in the general area of testing concurrent programs. However, to our knowledge, no existing work can guarantee to find all assertion violations in a multithreaded program without a model or a specification.

A tool called the *Nondeterminator* [FL97] detects data races in Cilk, a multithreaded variant of C. The Nondeterminator guarantees that if a program has a data race it will find and localize it. The Nondeterminator's algorithms depend on being able to model the concurrency of a Cilk program with a directed acyclic graph. The methods used do not apply to the more general concurrency found in most multithreaded languages (such as Java), and only data races are detected.

In deterministic testing [CT91], test cases consist not just of inputs for the program to be tested but of (input, schedule) pairs, where each schedule dictates an execution path of the program. Static analysis is used to generate the test case pairs. Since static analysis is limited in the scope of what it can determine about a program, the resulting test cases are not guaranteed to cover all behaviors of the program.

Model checking tools verify properties of models of concurrent systems. They have been shown to be effective in verifying many types of properties, including checking for assertion violations. However, they depend on a manageable number of program states explicitly delineated in a model, so their technology cannot be directly applied to testing an implementation.

Verisoft [God97] extends model checking methods in order to apply them to actual programs. It uses techniques that do not require manageable numbers of states [God96], performing some static analysis to provide information needed by its dynamic methods. Verisoft's target programs are small C programs consisting of multiple processes. It uses "visible" operations (operations on shared objects) in the code to define "global states" of the program. Since processes must explicitly declare the variables that they share, the number of global states is significantly smaller than the number of states, and since processes can only communicate through shared variables, Verisoft need only consider schedules that differ in their sequences of global states. Verisoft further prunes the state space by identifying schedules that lead to identical behaviors.

Verisoft has been shown to successfully discover errors in C programs composed of several concurrent processes. However, its techniques cannot readily be transferred to multithreaded languages. Since threads share the same memory space, all variables are potentially shared. Thus there are not a small number of explicitly declared "visible" operations that can be used to dramatically reduce the search space. Verisoft would have to assume that all variables are shared, and end up searching a prohibitively large state space.

Reachability testing [HTH95] builds on a method for systematically executing all possible orderings of operations on shared variables in a program consisting of multiple processes [Hwa93]. It is similar to Verisoft but does not do any pruning; it does parallelize the executions of different schedules to speed up testing. Like Verisoft, reachability testing requires explicit declarations of which variables are shared and hence does not apply directly to multithreaded programs.

4 The ExitBlock Algorithm

This section describes the ExitBlock algorithm, which systematically enumerates the possible schedules of the synchronized regions of a multithreaded program. It can be made more efficient by pruning the schedules it considers; this is discussed in Section 5.

4.1 Testing Criteria

ExitBlock requires that a program it is testing meets three criteria: that it follows a mutual-exclusion locking discipline, that its finalization methods are "well-behaved," and that all of its threads are terminating. Collectively we refer to these requirements as the *testing criteria*.

Mutual-Exclusion Locking Discipline A mutual-exclusion locking discipline dictates that each shared variable is associated with at least one mutual-exclusion lock, and that the lock or locks are always held whenever any thread accesses that variable. Assuming that a program follows such a discipline allows ExitBlock to systematically explore a program's behaviors by considering only schedules of atomic blocks, which is the key idea of the algorithm.

By limiting ExitBlock to those programs that follow this locking discipline, some classes of valid programs are ruled out. For example, a *barrier* is a point in a program which all threads must reach before any threads are allowed to continue beyond that point. A program using barriers can validly use different sets of locks for protecting the same variable on different sides of a barrier. However, as Savage et al. [S+97] argued, even experienced programmers with advanced concurrency control mechanisms available tend to use a simple mutual-exclusion locking discipline. Java encourages the use of this discipline through its synchronization facilities. We expect that requiring such a locking discipline will not overly restrict the applicability of the algorithm.

The Eraser algorithm by Savage et al. [S+97] can be used to efficiently verify that a program follows a mutual-exclusion locking discipline. Our implementation of ExitBlock runs Eraser in parallel with itself to verify that this criterion is met. Running Eraser alone only ensures that a program follows the discipline in one particular schedule, while running Eraser in parallel with ExitBlock checks that the discipline is followed in every schedule. Even though ExitBlock does not guarantee to execute all behaviors of a program that contains a discipline violation, it does guarantee to find the violation. This is guaranteed because ExitBlock will not miss any behavior until the instant one thread communicates with another thread using shared variables that are not protected by consistent sets of locks. At this point Eraser will catch the discipline violation.

Finalization This criterion is specific to Java and similar environments. Classes in Java may have `finalize` methods, or *finalizers*, that are called by the garbage collector when instances of those classes are reclaimed. There are no guarantees on when finalizers are called. They can be called in any order or even not at all, depending on when or if the garbage collector runs. This means that ExitBlock would need to consider every possible schedule of finalizers executing at arbitrary points in the program in order to find possible assertion violations, which can be a very large number of schedules. However, most finalizers are used only to deallocate memory that was allocated in some native Java method or to clean up system resources such as open file descriptors. Since these activities do not interact with the rest of the program, the order of execution of such finalizers does not affect the rest of the program. To reduce the number of schedules tested, ExitBlock assumes that the timing of finalizers has no bearing on whether assertions will be violated or not or whether a deadlock can occur. We call this ExitBlock's *finalization criterion*. It seems to be reasonable and is met by the finalizers in the standard Java libraries.

Terminating Threads ExitBlock requires that the program region to be tested has a defined endpoint in each thread's execution. This endpoint can be termination of the thread, exit from a specified module of interest, or execution of a certain number of operations. ExitBlock makes this requirement so that its depth-first search of the program's schedules will terminate. Throughout the rest of this paper, the term "program" will be used to refer to the region being tested (up to the defined endpoint for each thread).

4.2 Algorithm Overview

A program following a mutual-exclusion locking discipline can be divided into *atomic blocks* based on its synchronized regions, which are blocks of code that are protected by locks. Shared variables cannot be accessed outside of atomic blocks. Furthermore, shared variables modified by a thread inside of an atomic block cannot be accessed by other threads until the original thread exits that block. This means that enumerating possible orders of the atomic blocks of a program covers all possible behaviors of the program.

To see why enumerating the orders of the atomic blocks is sufficient, consider the following. The order of two instructions in different threads can only affect the behavior of the program if the instructions access some common variable — one instruction must write to a variable that the other reads or writes. In a program that correctly follows a mutual-exclusion locking discipline, there must be at least one lock that is always held when either instruction is executed. Thus they cannot execute simultaneously, and their order is equivalent to the order of their synchronized regions. By enumerating only synchronized regions, we significantly reduce the number of schedules to consider — rather than considering all possible schedules at the instruction level, the algorithm need only consider schedules of the program's atomic blocks.

Since the atomic blocks of a program can change from one execution to the next due to data flow through branches, we cannot statically compute a list of atomic blocks. We instead dynamically enumerate the atomic blocks using depth-first search. Depth-first search requires that the threads of the program eventually terminate; this is the reason for that testing criterion.

To perform depth-first search on a program we first execute one complete schedule of the program. Then we back up from the end of the program to the last atomic block boundary at which we can choose to schedule a different thread. We create a new schedule, branching off from this point in the program by choosing a different thread to run. We again execute until program completion. Then we systematically repeat the process, continuing to back up to

```
Thread 1 =
1:   <arbitrary code>
2:   synchronized (a) { <arbitrary code>
3:       synchronized (b) { <arbitrary code> }
4:       <arbitrary code> }
5:   <arbitrary code>

Thread 2 =
6:   <arbitrary code>
7:   synchronized (a) { <arbitrary code> }
8:   <arbitrary code>
9:   synchronized (b) { <arbitrary code> }
10:  <arbitrary code>
```

Figure 2: Two threads, one containing nested synchronized regions. Even for nesting we delineate atomic blocks with lock exits. We divide `Thread 1` into atomic blocks 1,2,3, 4, and 5, and `Thread 2` into atomic blocks 6,7, 8,9, and 10.

atomic block boundaries where we can make different choices than before.

To implement this search we need the ability to back up the program to a previous state, and the ability to present a consistent external view of the world to the program as we execute sections of code multiple times. Our implementation uses checkpointing to back up and deterministic replay to ensure the program receives the same inputs as it is re-executed.

This method will find an assertion violation in a program if one can occur, regardless of whether the violation shows up when the program is executed on a single processor or a multiprocessor. The locking discipline enforces an ordering on shared-variable accesses — two threads executing on different processors cannot access the same variable at the same time. Theorem 1 of [Bru99] proves that the execution of a program following the discipline on a multiprocessor is equivalent to some serialization of that program on a single processor.

4.3 Atomic Blocks

For un-nested synchronized regions, it is clear what an atomic block is: the region from a lock enter to that lock's exit (a lock enter is an acquisition of a lock, which occurs at the beginning of a synchronized block or method, while a lock exit is the release of a lock that occurs either at the end of a synchronized block or method or in a `wait` operation). The first atomic block of a thread begins with its birth and ends with its first lock exit, while the last atomic block begins after its last lock exit and ends with its death. We want as few blocks as possible, so we choose to group the non-synchronized code before a synchronized region with that region in one atomic block.

Dividing nested synchronized regions into atomic blocks requires more thought. Consider the code for `Thread 1` in Figure 2. Should the code section labeled 2 be in a separate atomic block from section 3? For the purpose of finding all assertion violations, it turns out that we can combine them. This may be counter-intuitive. Intuition suggests that we need to schedule `Thread 2`'s section 9 in between sections 2 and 3, because `Thread 2` could modify variables protected by the `b` lock while `Thread 1` is in section 2. However, this has the same effect as executing section 9 before both sections 2 and 3. Intuition also suggests that we should schedule section 7, 8, and 9 in between sections 2 and 3; this cannot occur, however, because while `Thread 1` holds the `a` lock section 7 cannot execute. We use lock exits (and thread births and deaths) as the sole boundaries of our atomic blocks.

```
BEGIN:
run program until 2nd thread is started, then:
  enabled = {1st thread, 2nd thread}
  delayed_thread = null
  goto LOOP
LOOP:
if (enabled is empty)
  if (stack is empty) goto DONE
  pop (RetPt, saved_enabled, saved_thread) off stack
  change execution state back to RetPt
  enabled = saved_enabled
  delayed_thread = saved_thread
  goto LOOP
curThread = choose a member of enabled
RetPt = make new checkpoint capturing execution state
old_enabled = enabled
run curThread until one of following events occurs:
  if (lock_enter && another thread holds that lock)
    move curThread from enabled to lock's block set
    change execution state back to RetPt
    goto LOOP
  if (curThread dies or reaches execution limit)
    remove curThread from enabled and old_enabled
    goto LOOP
  if (thread T starts)
    add T to enabled set
    continue running curThread
  if (lock_exit)
    push_enabled = old_enabled minus curThread
    push on stack (RetPt, push_enabled, curThread)
    add delayed_thread to enabled set
    add all threads on lock's blocked set
      to enabled set
    goto LOOP
DONE: // testing is complete
```

Figure 3: Pseudocode for initial ExitBlock algorithm that does not handle `wait` or `notify`. ExitBlock systematically executes the tree of possible schedules by dynamically discovering the atomic blocks of a program. At each lock exit (atomic blocks are delineated by lock exits) it pushes another branch of the tree onto its stack to be executed later.

4.4 The Algorithm

The ExitBlock algorithm systematically executes the schedules of atomic blocks delineated by lock exits. It also needs to consider separately each possible thread woken by a `notify` operation. We ignore thread operations such as `wait` and `notify` for now; they will be discussed in Section 4.5. Pseudocode for the algorithm, ignoring `wait` and `notify`, is shown in Figure 3.

ExitBlock dynamically explores the tree of possible schedules using depth-first search. Each thread in the program is kept in one of three places:

1. In the "block set" of a lock, if the thread needs that lock to continue its execution and another thread holds

the lock.

2. As the "delayed thread" — the thread that is not allowed to execute at this time in order to schedule another thread.

3. In the "enabled set", the set of threads that are ready to be executed.

ExitBlock runs the program normally until there are two threads; then it places both of them in the enabled set and enters its main loop. Each iteration of the loop is the execution of an atomic block of the current thread, which is chosen from the enabled set. First a checkpoint is taken and the enabled set saved; then the current thread is chosen and executed. When it reaches a lock exit, a new branch of the tree of schedules is created and stored on a stack for later execution. This branch represents executing the rest of the threads from the point just prior to this atomic block. We need the checkpoint so we can go back in time to before the block, and we need to make sure we execute a different thread from that point. So we make the current thread the "delayed thread" of the branch. The delayed thread is re-enabled after the first atomic block of a branch so that it can be interleaved with the atomic blocks of the other threads. Note that we use the old enabled threads set in the branch; this is because newly created threads cannot interact with the atomic block in which they were created, and in fact did not exist at the time of the checkpoint.

ExitBlock keeps executing the current thread, pushing branches onto the stack, until the thread dies or reaches a preset execution limit, which could either be a certain amount of execution or exiting from the subsystem being tested. Then it chooses a different thread from the enabled set to be the current thread and executes it in the same manner. ExitBlock treats the execution from a thread's final lock exit to its death as a separate atomic block; this is unavoidable since we have no way of knowing when a thread will die beforehand. When the enabled set becomes empty ExitBlock pops a new branch off of the stack and uses the branch's state as it continues the loop.

If the current thread cannot obtain a lock, a new thread must be scheduled instead. Since we only want thread switches to occur on atomic block boundaries, we abort the current branch of the tree and undo back to the end of the previous atomic block before we schedule a different thread. Which threads own which locks must be kept track of to determine if a lock can be acquired without having the thread actually block on it. The thread is placed in the block set of the lock and a different enabled thread is tried after returning to the checkpoint. ExitBlock cannot enter a lock-cycle deadlock since it never lets threads actually block on locks, so that is not a concern (see Section 6 for a method to detect deadlocks).

ExitBlock assumes that it has the power to take and return to checkpoints, and to deterministically replay all interactions between the code of the program and the rest of the world. It must do so to make the threads of the program deterministic with respect to everything except variables shared with other threads. The pseudocode shown here assumes that deterministic replay is going on behind the scenes.

Figure 4 shows the tree of schedules executed by ExitBlock for a program consisting of the two threads in Figure 2. `Thread 1` is abbreviated as `T1` and `Thread 2` is abbreviated as `T2` in the figure. Each node of the tree lists the

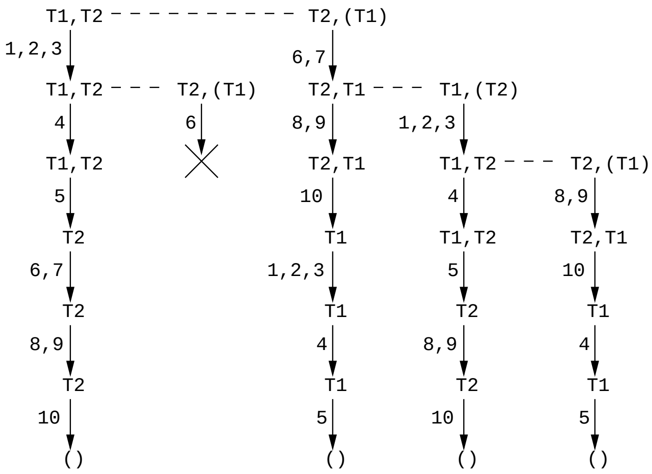

Figure 4: Tree of schedules explored by ExitBlock for the threads in Figure 2. Threads in parentheses are delayed. Arrows indicate the execution of the sections of code on their left. An `X` means that a path was aborted because a lock could not be obtained.

threads in the enabled set, and the delayed thread, if there is one, in parentheses. A solid arrow indicates execution of one atomic block by the first thread listed in the node at the tail of the arrow, with the section numbers of the code contained in that block listed to the left of the arrow. Parallel execution paths are connected by dashed lines to indicate that they are both executed from a checkpoint that was created at the left end of the dashed line. The large `X` in the figure indicates that that path was aborted because `T2` could not obtain a lock that `T1` was holding. A checkpoint is created at each node in the tree, but most are not used. Our implementation of checkpointing uses a lazy copy-on-write scheme to make this efficient.

4.5 Wait and Notify

So far we have ignored thread communication other than with shared variables protected by locks; in particular, we have ignored condition variables. In Java any object can be used as a condition variable. A thread can `wait` on an object, causing the thread to go to sleep until another thread performs a `notify` (awakening one thread) or `notifyAll` (awakening all threads) on that object. We can deal with `wait` by considering the lock exits that result when a thread `wait`s to be like other lock exits. A thread that has acquired a lock multiple times will release it multiple times when it performs a `wait` on the lock object. Since there is no way to schedule another thread in between these releases, we consider them to be one lock exit. Also, a thread that performs a `wait` should be removed from the set of enabled threads for the duration of its `wait`.

Dealing with the notification operations is more involved. For `notifyAll` we need to add every thread that is woken up to the block set for the notify lock. For `notify` we need to do the same thing for the single thread that wakes up; however, if there are multiple threads waiting on an object, which thread will be woken by `notify` is nondeterministic. The algorithm needs to explore the schedules resulting from

```
BEGIN:
run program until 2nd thread is started, then:
  wait = {}
  no_notify = {}
  ...
LOOP:
if (enabled is empty)
  ...
  pop (RetPt, saved_enabled, saved_thread, wait,
    no_notify) off stack
  ...
...
run curThread until one of following events occurs:
  ...
  if (curThread does a wait())
    move curThread from enabled set to wait set
    goto LOCKEXIT
  if (curThread does a notifyAll())
    wake up all threads waiting on notify object
    move those threads from wait set to block set
      of notify lock
  if (curThread does a notify())
    select a thread T not in no_notify to wake up
    move T from wait set to block set of notify lock
    if (threads are still waiting on notify object)
       push on stack (RetPt, enabled, delayed_thread,
         wait, no_notify+T)
    no_notify = no_notify - all threads originally
      waiting on notify object
  if (lock_exit)
    LOCKEXIT:
    ...
    push (RetPt, saved_enabled, curThread, wait, {})
    ...
DONE: // testing is complete
```

Figure 5: Changes to the pseudocode for the ExitBlock algorithm in Figure 3 in order to handle `wait` and `notify`. A set of waiting threads and a set of threads that should not be notified are added to each branch. We treat `wait` like a lock exit, and for `notify` we create new branches of the tree to consider other threads being notified.

each possible thread being woken by a `notify`, so it needs to make new branches for each. Figure 5 shows additions to the ExitBlock algorithm that will handle `wait` and `notify`. For each branch of the tree in which we wish to notify a different thread, we must prevent threads that have already been notified from being notified again. Thus we keep track of a set of threads that should not be notified (the "no_notify" set) for every branch of the execution tree. Even if there are multiple notifications in one atomic block, we can still use a single no_notify set since there can be no overlap between threads waiting on the multiple notify objects (a thread can only `wait` on one object at a time). When we execute a `notify` we wake up a thread that is not in the no_notify set. We want to carry the set to the next branch's identical `notify` operation in order to keep reducing the set of threads we wake up; however, we do not want a different `notify` that occurs later to have the threads it wakes up restricted by this `notify`. Therefore, after pushing a new `notify` branch on the stack, we remove from the no_notify set all threads originally waiting on the notify object.

Most other thread-related operations, such as `sleep`, `yield`, preemption, and thread priorities, can be ignored. As for input and output, ExitBlock assumes the user will provide any needed input and that blocking on an input will not be a problem; replaying the input deterministically for other paths is necessary and, as discussed earlier, is assumed to be happening all along.

4.6 Analysis of ExitBlock

For a program with k threads that each contain n lock exits the total number of lock exits we have is $k*n$. $\binom{kn}{n}$ is the number of ways we can place the first thread's n lock exits in between all of the other lock exits. Placing a lock exit also places the atomic block that the lock exit terminates. Now that those are placed, we have $(k-1)*n$ slots left; $\binom{(k-1)n}{n}$ is the number of ways the second thread can place its atomic blocks in between the others. The process continues until we have the number of schedules that the ExitBlock algorithm needs to consider:

$$\binom{kn}{n} * \binom{(k-1)n}{n} * \binom{(k-2)n}{n} * \cdots * \binom{n}{n} \qquad (1)$$

By Stirling's approximation,

$$\binom{kn}{n} = \theta\left(\frac{\sqrt{k}}{\sqrt{n(k-1)}}\left(\frac{1}{k}\right)^n\left(\frac{k}{k-1}\right)^{n(k-1)}\right)$$

Thus the number of schedules that must be explored grows exponentially in both the number of threads and the number of lock uses by each thread. Section 5 will modify the ExitBlock algorithm to reduce the expected growth of the number of paths from exponential to polynomial in the number of locks per thread.

4.7 Correctness of ExitBlock

When a program meets the criteria of Section 4.1, ExitBlock guarantees to find all possible assertion violations in the program. Any program condition can be detected using assertions; thus, ExitBlock guarantees to enumerate all possible behaviors of the program. We state this formally in the following theorem, which is proved correct as Theorem 3 of [Bru99].

Theorem *Consider a program P that meets the testing criteria defined in Section 4.1. Suppose that for a given input, when executed on a single processor, P can produce an assertion violation. A schedule exists that is produced by the ExitBlock algorithm in which the same assertion is violated.*

5 The ExitBlock-RW Algorithm

The ExitBlock algorithm executes all schedules of atomic blocks. However, not all schedules need to be executed in order to find all possible assertion violations. If two atomic blocks have no data dependencies between them, then the order of their execution with respect to each other has no effect on whether an assertion is violated or not. The ExitBlock-RW algorithm uses data dependency analysis to prune the tree of schedules explored by ExitBlock.

5.1 Read-Write Pruning

As an example, consider again the tree from Figure 4. The rightmost branch represents the ordering `6,7,1,2,3,8,9,10,4,5`. Suppose that the two threads are independent except for a data dependency between code sections 2 and 7. Code sections 4 and 5 only share locks with section 7 of the other thread. This means that 4 and 5 can have data dependencies only with 7 and with no other code sections of the other thread. We have already considered 4 and 5 both before and after 7 in earlier branches. So why should we execute 4 and 5 at the end of this last branch? If there were an assertion violation produced only when 4 and 5 were executed after 7, we would already have found it. Thus we can trim the end of the rightmost branch.

We take advantage of this observation as follows. We record the reads and writes performed while executing an atomic block. Instead of delaying just the current thread and re-enabling it after the first step in the new branch, we keep a set of delayed threads along with the reads and writes of the atomic block they performed just before the new branch was created. We only re-enable a delayed thread when the currently executing thread's reads and writes intersect with the delayed thread's reads and writes. If no such intersection occurs, then none of the later threads interact with the delayed thread and there is no reason to execute schedules in which the delayed thread follows them. The intersection is computed as follows: a pair consisting of a set of reads and a set of writes (r_1, w_1) intersects with a second pair (r_2, w_2) if and only if $(w_1 \cap w_2 \neq \emptyset \vee r_1 \cap w_2 \neq \emptyset \vee w_1 \cap r_2 \neq \emptyset)$.

We call the ExitBlock algorithm augmented with this read-write pruning ExitBlock-RW. Pseudocode for the changes to ExitBlock required for ExitBlock-RW is given in Figure 6. Note that the algorithm uses the reads and writes performed during the first execution of an atomic block A to check for data dependencies with other threads' atomic blocks. What if A performs different reads and writes after being delayed? Since we never execute past a block that interacts with any of A's variable accesses without re-enabling A, no blocks executed while A is delayed can affect the reads and writes it would perform upon being re-enabled.

The ExitBlock-RW algorithm's schedules for the threads in Figure 2, assuming that the only inter-thread data dependency is between sections 2 and 7, are shown in Figure 7.

```
BEGIN:
run program until 2nd thread is started, then:
  delayed = {}
  ...
LOOP:
if (enabled is empty)
  ...
  pop (RetPt, saved_enabled, saved_delayed, wait,
    no_notify) off of stack
  change execution state back to RetPt
  enabled = saved_enabled
  delayed = saved_delayed
  reads = writes = {}
  ...
...
run curThread until one of following events occurs:
  if (curThread performs a read)
    record it in the reads set
  if (curThread performs a write)
    record it in the writes set
  ...
  if (lock_exit)
    push_enabled = old_enabled minus curThread
    push_delayed = delayed plus
      (curThread, reads, writes)
    push (RetPt, push_enabled, push_delayed, wait,
      {})
    foreach (thread, r, w) in delayed
      if ((r,w) intersects (reads,writes))
        move thread from delayed set to enabled set
    move all threads in lock's blocked set
      to enabled set
    goto LOOP
DONE: // testing is complete
```

Figure 6: Changes to the ExitBlock pseudocode of Figures 3 and 5 for the ExitBlock-RW algorithm. This algorithm records the reads and writes performed during atomic blocks and only interleaves two blocks if their read and write sets intersect.

5.2 Analysis of ExitBlock-RW

If no blocks interact, no thread we delay ever becomes re-enabled. For each schedule, each thread runs a certain amount, gets delayed, and never wakes up. (Of course, there is at least one schedule for each thread that executes that thread until it dies.) Thus we can consider the problem of creating each schedule simply that of deciding where to cut off each thread; by arranging the thread sections end-to-end we have the schedule. The sections have the property that the section of thread i must precede that of thread j if i is started first in the program.

For a program with k threads that each obtain locks a total of n times, with absolutely no interactions between the atomic blocks, we have $n+1$ places to terminate each thread, and it does not matter where we terminate the last thread of each schedule; thus we have $(n+1)^{k-1}$ different schedules. This number will be lower if some threads cannot run until others finish, or other constraints are present, and higher if interactions between blocks exist (potentially as high as ExitBlock's formula if every block interacts with every other, which fortunately is very unlikely).

This best-case result, polynomial in the number of locks per thread and exponential in the number of threads, is much better than the growth of ExitBlock which is exponential in the number of locks per thread. The number of threads in a program is typically not very high, even for large programs, while the code each thread executes can grow substantially. Thus in the best case the ExitBlock-RW algorithm achieves polynomial growth.

The important issue is how close to the best case normal programs are. The number of interactions between threads in a program is usually kept to a minimum for ease of programming. Thus we expect the average case to be close to the best case.

Further pruning of the schedules considered by ExitBlock-RW is possible while preserving the guarantee that all assertion violations are detected. If the user knows that one of the program's methods is used in such a way that it causes no errors in the program, he or she can instruct the tester to not generate multiple schedules for atomic blocks

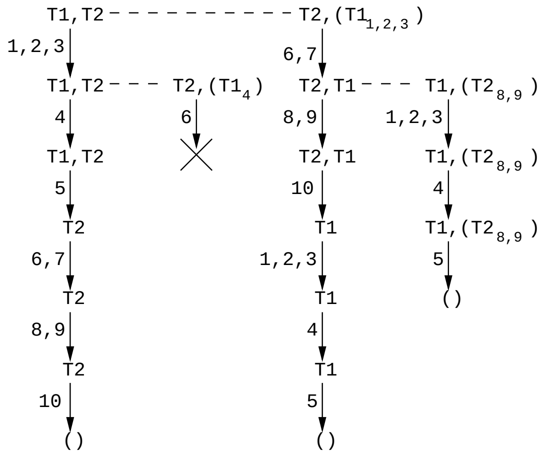

Figure 7: Tree of schedules explored by ExitBlock-RW for the threads in Figure 2 on page 4, assuming the only inter-thread data dependency is between code sections 2 and 7. Threads in parentheses are delayed, with subscripts indicating the code sections over which read/write intersections should be performed. Compared to the tree for the ExitBlock algorithm shown in Figure 4, this tree does not bother finishing the schedule 6,7,1,2,3,4,5,... and does not execute at all the final schedule of the other tree (6,7,1,2,3,8,9,10,4,5).

contained in the method. For example, if a program uses some library routines that it assumes are thread safe and error free, the user can tell the tester to trust those library routines. Then every time the tester would normally end an atomic block, it checks to see if the current method is one of the trusted routines; if so, it simply keeps executing as though there is not an atomic block boundary there. One instance of this extension is implemented in our tester as a parameter that when set assumes that all methods in any of the `java.*` packages are safe. It dramatically reduces the number of paths the algorithm must explore.

Another pruning idea is to ignore an atomic block whose lock object is not shared. If only one thread ever accesses it then there is no reason to consider possible schedules encountered while in methods called on that object. This may be computable through static analysis [WR99].

5.3 Correctness of ExitBlock-RW

This is proved correct as Theorem 4 of [Bru99].

Theorem *Consider a program P that meets the testing criteria defined in Section 4.1. Suppose that for a given input and when executed on a single processor P produces an assertion violation. A schedule exists that is produced by the ExitBlock-RW algorithm in which the same assertion is violated.*

6 Detecting Deadlocks

A deadlock is a cycle of resource dependencies that leads to a state in which all threads are blocked from execution. Two kinds of cycles are possible; if the cycle is not one of locks, then it must involve some or all threads in a `wait` state

```
Thread 1 =
1:   <arbitrary code>
2:   synchronized (a) { <arbitrary code>
3:       synchronized (b) { <arbitrary code> }
4:       <arbitrary code> }
5:   <arbitrary code>

Thread 2 =
6:   <arbitrary code>
7:   synchronized (b) { <arbitrary code>
8:       synchronized (a) { <arbitrary code> }
9:       <arbitrary code> }
10:  <arbitrary code>
```

Figure 8: Two threads with the potential to deadlock. If the code sections are executed in any order where 2 precedes 8 and 7 precedes 3 (for example, 1,2,6,7) then a deadlock is reached in which `Thread 1` holds lock `a` and wants lock `b` while `Thread 2` holds lock `b` and wants lock `a`.

and the rest blocked on locks. We will refer to deadlocks consisting solely of threads blocked on locks as *lock-cycle deadlocks*, which will be discussed in the next section, and those that contain waiting threads as *condition deadlocks*, which will be discussed in Section 6.2.

6.1 Lock-Cycle Deadlocks

The ExitBlock algorithm rarely executes schedules that result in lock-cycle deadlocks. (When it does it simply aborts the current branch in the tree of schedules.) Consider the threads in Figure 8 and the schedules that the ExitBlock algorithm produces for these threads, shown in Figure 9. In order for deadlock to occur, `Thread 1` needs to be holding lock `a` but not lock `b` and `Thread 2` needs to be holding lock `b` but not lock `a`. This will not happen since for ExitBlock the acquisitions of both locks in each thread are in the same atomic block. Deadlocks are only executed in rare cases involving multiply nested locks.

In order to always detect deadlocks that are present, we could change our definition of atomic block to also end blocks before acquiring a lock while another lock is held. This would cause ExitBlock to directly execute all schedules that result in lock-cycle deadlocks; however, this is undesirable since it would mean more blocks and thus many more schedules to search. Instead of trying to execute all deadlocks we execute our original, minimal number of schedules and detect deadlocks that *would occur* in an unexplored schedule.

The key observation is that a thread in a lock-cycle deadlock blocks when acquiring a *nested* lock, since it must already be holding a lock. Also, the lock that it blocks on cannot be the nested lock that another thread in the cycle is blocked on, since two threads blocked on the same lock cannot be in a lock cycle. The cycle must be from a nested lock of each thread to an already held lock of another thread. For example, when the threads of Figure 8 deadlock, `Thread 1` is holding its outer lock `a` and blocks on its inner lock `b` while `Thread 2` is holding its outer lock `b` and blocks on its inner lock `a`.

These observations suggest the following approach. We track not only the current locks held but also the last lock held (but no longer held) by each thread. Then, after we execute a synchronized region nested inside some other syn-

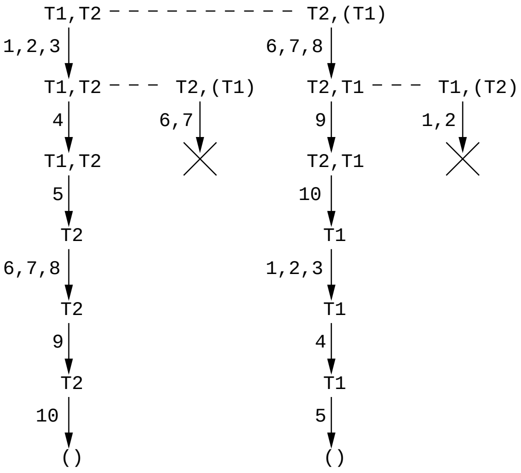

Figure 9: Tree of schedules explored by ExitBlock for the threads in Figure 8. Reverse lock chain analysis will detect two deadlocks, one at each aborted path (denoted by a large `X` in the figure). Threads in parentheses are delayed.

chronized region, the last lock held will be the lock of the inner synchronized region. When a thread cannot obtain a lock, we look at the last locks held by the other threads and see what would have happened if those threads had not yet acquired their last locks. We are looking for a cycle of matching outer and inner locks; the outer locks are currently held by the threads and the inner locks are the threads' last locks held. If the current thread cannot obtain a lock l and we can follow a cycle of owner and last lock relationships back to the current thread — if l's current owner's last lock's current owner's last lock's ... ever reaches a current owner equal to the current thread — then a lock-cycle deadlock has been detected. We call this *reverse lock chain analysis.* It is straightforward to implement, and since failures to acquire locks are relatively rare it does not cost much in performance.

In Figure 9, the two large `X`'s indicate paths that were terminated because a lock could not be obtained. These are the points where reverse lock chain analysis occurs. At the first point, thread `T1` holds lock `a` and last held lock `b`. Thread `T2` holds `b` and fails in its attempt to obtain `a`. The analysis performed at this point finds the following cycle: `a`'s current owner is `T1`, whose last lock is `b`, whose current owner is `T2`, the current thread. At the second point a similar analysis discovers the same deadlock in a different schedule.

We can improve performance by using reverse lock chain analysis with ExitBlock-RW rather than with ExitBlock. This seems to work well in practice. For all of the example programs that we tested that contained lock-cycle deadlocks, ExitBlock-RW plus analysis found the deadlocks.

However, this optimization comes at a cost. ExitBlock-RW plus reverse lock chain analysis does not always find deadlocks that are present. Figure 10 shows a counterexample. ExitBlock plus analysis finds the deadlock in this program, while ExitBlock-RW does not. The tree of execution paths for ExitBlock on the program is shown on the left in Figure 11. The potential deadlock is detected in two different places. The right of Figure 11 shows the tree of

```
Thread 1 =
1:   <arbitrary code>
2:   synchronized (a) { <arbitrary code>
3:       synchronized (b) { <arbitrary code> }
4:       <arbitrary code> }
5:   <arbitrary code>

Thread 2 =
6:   <arbitrary code>
7:   synchronized (a) { <arbitrary code> }
8:   <arbitrary code>
9:   synchronized (b) { <arbitrary code>
10:      synchronized (a) { <arbitrary code> }
11:      <arbitrary code> }
12:  <arbitrary code>
```

Figure 10: Two threads whose potential deadlock will be caught by reverse lock chain analysis in ExitBlock but not in ExitBlock-RW (see Figure 11). None of the regions of code interact.

schedules for ExitBlock-RW on the program (none of the regions of code of the two threads interact with the other thread at all). The pruning performed by ExitBlock-RW has completely removed the branches of the tree that detect the deadlock.

One idea is to have ExitBlock-RW consider a lock enter to be a write to the lock object; this would certainly prevent the pruning of the deadlock situations in this example, but we have not proved it would always do so.

6.2 Condition Deadlocks

Condition deadlocks involve a deadlocked state in which some of the live threads are waiting and the rest are blocked on locks. The tester can detect condition deadlocks by simply checking to see if there are threads waiting or blocked on locks whenever it runs out of enabled threads to run. We call the combination of this checking with ExitBlock and reverse lock chain analysis the ExitBlock-DD algorithm. The same combination but with ExitBlock-RW we call ExitBlock-RWDD.

ExitBlock-RWDD cannot use the same condition deadlock check as ExitBlock-DD because of its delayed thread set. We only delay threads to prune paths, so if we end a path with some waiting threads but also some delayed threads, we have not necessarily found a deadlock since nothing is preventing the delayed threads from executing and waking up the waiting threads.

Thus, to avoid reporting false condition deadlocks in ExitBlock-RWDD, we must not report condition deadlocks when there are delayed threads. We could attempt to execute the delayed threads to find out if there really is a condition deadlock; however, there is no way to know how long they might execute. We have not fully investigated this idea. ExitBlock-RWDD did successfully detect condition deadlocks in every sample program we tested, but it makes no guarantees. Because of this, the implementation of our tester has two modes: the default uses ExitBlock-RWDD for efficiency, while the second mode uses ExitBlock-DD in order to guarantee to find deadlocks.

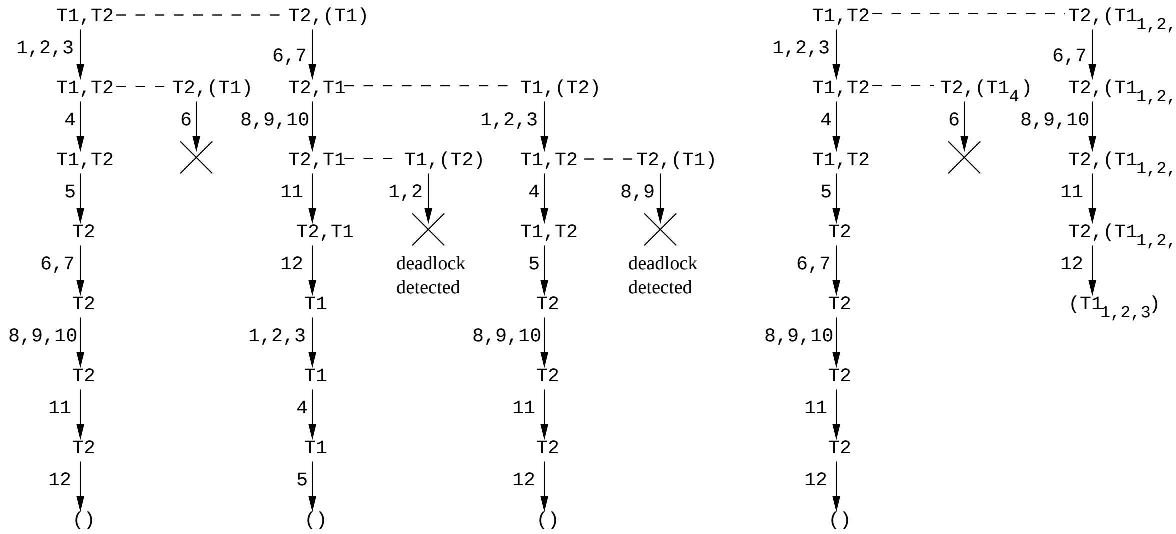

Figure 11: On the left is the tree of schedules explored by ExitBlock for the threads in Figure 10; on the right is the tree explored by ExitBlock-RW for the same threads.

6.3 Analysis of ExitBlock-DD

Both ExitBlock-DD and ExitBlock-RWDD execute no more schedules than their respective non-deadlock-detecting versions, and the deadlock checks themselves are inexpensive. However, if a guarantee is needed, using ExitBlock-DD instead of ExitBlock-RWDD is expensive. It is worth pursuing the two ideas mentioned in the previous section, considering lock enters to be writes and executing delayed threads when a potential condition deadlock is encountered. The penalty in extra paths considered for the former idea should not be too high since atomic blocks sharing the same lock often have data dependencies anyway. Both ideas will have performance penalties, but will still be more efficient than ExitBlock-DD; combined they may provide ExitBlock-RWDD with a guarantee.

6.4 Correctness of ExitBlock-DD

This is proved correct as Theorem 5 of [Bru99].

Theorem *Consider a program P that meets the testing criteria defined in Section 4.1.*
(a) Suppose that P enters a deadlock for a given input when executed on a single processor. Then a schedule is produced by the ExitBlock-DD algorithm in which a deadlock is detected (though not necessarily the same one).
(b) Furthermore, if there are no schedules of P that result in a deadlock, the ExitBlock-DD algorithm will not detect any deadlocks.

7 Experimental Results

We implemented the ExitBlock algorithm as a testing tool for the Rivet virtual machine [Rivet]. Rivet is an extensible tool platform structured as a Java virtual machine. Its primary goal is flexibility in enabling rapid tool development. Rivet contains built-in checkpointing and deterministic replay abilities that the tester makes use of.

Our tester has an assertion mechanism that client programs can use to directly notify the tester when assertions fail. This gives the tester the ability to notify the user of assertion failures and to stop testing when one occurs. If a client program cannot be modified errors must be detected by examining its output.

The tester implements the read-write pruning of the ExitBlock-RW algorithm, and also has an optional parameter that when set causes it to assume that all methods in any of the `java.*` packages are safe. This option causes a dramatic reduction in the number of schedules explored by the tester. It performs deadlock detection as ExitBlock-RWDD does, and through another optional parameter it can eliminate read-write pruning in order to execute ExitBlock-DD. This parameter was never needed in the sample programs we tested; ExitBlock-RWDD never missed a deadlock.

Our tester finds common timing-dependent errors that are often difficult to detect. In experiments it has detected:

- using an `if` instead of a `while` to test a condition variable
- using `notify` instead of `notifyAll` to wake up threads waiting on a condition variable that is used for more than one condition
- mutual-exclusion locking discipline violations that do not occur in all dynamic schedules
- errors that occur only in certain orderings of modules that are individually correctly synchronized. For example, consider the SplitSync program presented in Figure 1 in Section 2. When its invariant is made an assertion the tester reports it on the third schedule it executes.
- lock-cycle deadlocks
- condition deadlocks

The tester can successfully find errors in programs of four threads with twenty synchronized regions each in a matter of minutes when executed on a standard PC. For a complete description of experimental results, see [Bru99].

8 Conclusions

We have described practical algorithms for a systematic, behavior–complete tester for multithreaded programs. By assuming that the program to be tested follows a mutual–exclusion locking discipline, we need only enumerate all schedules of synchronized regions instead of all schedules of instructions to cover all behaviors of the program. This assumption is not overly restrictive: as Savage et al. [S+97] argue, even experienced programmers tend to follow such a discipline. They do this even when more advanced synchronization techniques are readily available.

We presented the ExitBlock algorithm that guarantees to test all program behaviors while only considering the possible schedules of synchronized regions. We showed how to further prune the schedules executed in the ExitBlock-RW algorithm. In addition, we augmented both algorithms with deadlock detection methods.

Since ExitBlock tests a program for a single input, test case generation is crucial. Generating test cases for use with ExitBlock is simplified since the effects of different inputs on scheduling can be ignored. Conventional sequential program test case generation can be used instead of the complex generation techniques [KFU97] needed to generate test cases for other multithreaded program testing methods.

Our ideas are applicable to multithreaded programs with properly nested locks. There is a large class of such programs due to the enforcement of proper nesting by the synchronization constructs available in languages such as Java.

References

[Bru99] Derek Bruening. *Systematic Testing of Multithreaded Java Programs.* Master of Engineering Thesis, Massachusetts Institute of Technology, 1999.
`http://sdg.lcs.mit.edu/rivet.html#pubs`

[CT91] Richard H. Carver and Kuo-Chung Tai. "Replay and Testing for Concurrent Programs." *IEEE Software*, 8(2):66-74, March 1991.

[FL97] Mingdong Feng and Charles E. Leiserson. "Efficient detection of determinacy races in Cilk programs." *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pp. 1-11, Newport, Rhode Island, June 1997.

[GJS96] J. Gosling, B. Joy, G. Steele. *The Java Language Specification*, Addison-Wesley, 1996.

[God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems — An Approach to the State-Explosion Problem*, volume 1032 of Lecture Notes in Computer Science. Springer-Verlag, January 1996.

[God97] Patrice Godefroid. "Model checking for programming languages using Verisoft." *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pp. 174-186, Paris, France, January 1997.

[Hwa93] Gwan-Hwan Hwang. *A Systematic Parallel Testing Method for Concurrent Programs.* Master's Thesis, Institute of Computer Science and Information Engineering, National Chiao-Tung University, Taiwan, 1993.

[HTH95] Gwan-Hwan Hwang, Kuo-Chung Tai, and Ting-Lu Hunag. "Reachability Testing: An Approach to Testing Concurrent Software" *International Journal of Software Engineering and Knowledge Engineering*, Vol. 5 No.4, December 1995.

[KFU97] T. Katayama, Z. Furukawa, K. Ushijima. "A Test-case Generation Method for Concurrent Programs Including Task-types." *Proceedings of the Asia Pacific Software Engineering Conference and International Computer Science Conference*, IEEE Comput. Soc. 1997, pp.485-94. Los Alamitos, CA, USA.

[Rivet] The Rivet Virtual Machine.
`http://sdg.lcs.mit.edu/rivet.html`

[S+97] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro and T. Anderson. "Eraser: A dynamic data race detector for multithreaded programs." *ACM Transactions on Computer Systems*, 15(4):391-411, November 1997.

[WR99] John Whaley and Martin Rinard. "Compositional Pointer and Escape Analysis for Java Programs," OOPSLA 1999.