A Race-Detection and Flipping Algorithm for Automated Testing of Multi-Threaded Programs

Koushik Sen[1] and Gul Agha[2]

[1] University of California Berkeley, USA.
{ksen}@cs.berkeley.edu
[2] University of Illinois at Urbana-Champaign, USA.
{agha}@cs.uiuc.edu

Abstract. Testing concurrent programs that accept data inputs is notoriously hard because, besides the large number of possible data inputs, nondeterminism results in an exponentially large number of interleavings of concurrent events. In order to efficiently test shared-memory multi-threaded programs, we develop an algorithm based on race-detection and flipping and illustrate how it can be combined with concolic execution (a simultaneous symbolic and concrete execution method) to test multi-threaded programs with data inputs. The goal of our algorithm is to minimize redundant executions while ensuring that all reachable statements in a program are executed. To achieve this, our algorithm explores all distinct causal structures of a multi-threaded program (i.e., the partial order among events generated during an execution). Because our algorithm is based on race-detection, it enables us to report potential data races and deadlocks. We have implemented our algorithm in a tool called jCUTE. We describe the results of applying jCUTE to real-world multi-threaded Java applications and libraries. In particular, we discovered several undocumented potential concurrency-related bugs in the widely used Java collection framework distributed with the Sun Microsystems' JDK 1.4.

1 Introduction

Testing programs is generally hard because of the large number of possible inputs to a program. Testing concurrent programs is notoriously harder because of the exponentially large number of possible interleavings of concurrent events. Many of these interleavings share the same causal structure (also called the *partial order*), and thus are equivalent with respect to finding bugs in a given program. Techniques for avoiding such redundant executions are called *partial order reduction* [20, 11, 5].

A number of approaches [6, 4, 2, 1] to testing concurrent programs assume that the data inputs are from a small finite domain. These approaches rely on exhaustively executing the program for all possible inputs and perform a partial order reduction to reduce the search space. The problem with these approaches is that it is hard to scale them – the input set is often too large.

A second approach is to execute a program symbolically in a customized virtual machine which supports partial order reduction [8, 21]. This requires checking satisfiability of complex constraints (corresponding to every branch point in a program). Unfortunately, checking such satisfiability may be undecidable or computationally intractable. Moreover, in concurrent programs, partial order reduction for symbolic execution requires computing the dependency relations between memory accesses in a program. Because it involves alias analysis, such a computation is often conservative resulting in extra dependencies. For these reasons, large numbers of unreachable branches may be explored, often causing many warnings for bugs that could never occur in an actual execution.

Our approach is to extend *concolic testing*, which combines concrete and symbolic execution by using one to guide the other [15, 7, 13]. The idea behind concolic testing is to use symbolic execution to generate inputs that direct a program to alternate paths, and to use the concrete execution to guide the symbolic execution along a concrete path, and replace symbolic values (variables) by concrete values if the symbolic state is too complex to be handled by a constraint solver.

To systematically test multithreaded programs, we propose a new algorithm called the *race-detection and flipping algorithm* and combine this algorithm with concolic testing. The algorithm works as follows. For a given concrete execution, at runtime, we determine the partial order relation or the *exact* race conditions (both data race and lock race) between the various events in the execution path. Subsequently, we systematically re-order or permute the events involved in these races by generating new thread schedules as well as generate new test inputs. This way we explore one representative from each partial order. The result is an efficient testing algorithm for concurrent programs which, at the cost of missing some potential bugs, avoids the problem of false warnings.

We have implemented the algorithm in a tool, called jCUTE, for testing Java programs.[3] Apart from detecting assertion violations and uncaught exceptions, jCUTE reports all data race conditions and deadlock states encountered during the process of testing.

We provide some case studies to illustrate the utility of our approach. In our first case study, we tested the thread-safe Java Collection framework provided with the Sun Microsystems' Java 1.4. Surprisingly, we discovered several previously unknown data races, deadlocks, uncaught exceptions, and an infinite loop in this widely used library. All of them are potential bugs related to multi-threaded execution. In our second case study, we tested several small to medium sized concurrent Java programs used as case studies for evaluating NASA's Java PathFinder and KSU's Bandera tool. In all those programs, our tool discovered bugs which had previously been found by model-checking *manually abstracted* versions of the programs–of course, in our case without abstracting the program. In the last two case studies, we detected well-known security attacks in the concurrent implementation of the Needham-Schroeder and the TMN protocols.

The contributions of this paper are as follows:

- We describe a new algorithm, called *race-detection and flipping algorithm*, for efficiently exploring all non-equivalent executions of a shared-memory multi-threaded program with no data input.

[3] Available at `http://osl.cs.uiuc.edu/~ksen/cute/`.

– We apply a tool based on our method to real world case studies. The results show that our method can efficiently detect data races, deadlocks and other bugs in a multi-threaded program.

Due to space limitations, we skip the description of the concolic testing and the details about how to combine concolic testing and the race-detection and flipping algorithm. The description of concolic testing can be found in [7, 15] and the details of the combination can be found in [12].

The outline of the rest of the paper is as follows. In Section 2, we use an example to give an overview of the race-detection and flipping algorithm combined with concolic testing. In Section 3, we describe the execution model that we assume for the purpose of describing our the race-detection and flipping algorithm. In Section 4, we describe the race-detection and flipping algorithm. Section 5 describes four case studies. In Section 6, we discuss related work.

2 Overview of Our Approach

In concolic testing our goal is to generate data inputs and schedules that would exercise all feasible executions paths of a program. Our algorithm for concolic testing uses concrete values as well as symbolic values for the inputs, and executes a program both concretely and symbolically. The symbolic execution is similar to the traditional symbolic execution [9], except that jCUTE follows the path that the concrete execution takes. During the course of the execution, it collects the constraints over the symbolic values at each branch point (i.e., the *symbolic constraints*). At the end of the execution, the algorithm has computed a sequence of symbolic constraints corresponding to each branch point. We call the conjunction of these constraints a *path constraint.* Observe that all input values that satisfy a given path constraint will explore the same execution path, provided that we follow the same thread schedule.

Apart from collecting symbolic constraints, the algorithm also computes the race condition between various events in the execution of a program, where, informally, an event represents the execution of a statement in the program by a thread. We say that two events are in a *race* if the following three conditions hold: they are events belonging to different threads, they access (i.e. read, write, lock, or unlock) the same memory location without holding a common lock, and the order of the happening of the events can be permuted by changing the schedule of the threads. The race conditions are computed by analyzing the concrete execution of the program with the help of dynamic vector clocks for multithreaded programs (dynamic vector clock algorithm was introduced in [17] for predictive monitoring of multi-threaded programs.)

The algorithm first generates a random input and a schedule which specifies the order of the execution of threads. Then the algorithm does the following in a loop: it executes the code with the generated input and the schedule. At the same time the algorithm computes the race conditions between various events as well as the symbolic constraints. It backtracks and generates a new schedule or a new input and executes the program again. It continues until it has explored all possible distinct execution paths using a depth-first search strategy. The choice of new inputs and schedules is made in one of the following two ways:

1. The algorithm picks a constraint from the symbolic constraints that were collected along the execution path and negates the constraint to define a new path constraint. The algorithm then finds, if possible, some concrete values that satisfy the new path constraint. These values are used as input for the next execution.
2. The algorithm picks two events which are in a race and generates a new schedule that at the point where the first event happened, the execution of the thread involved in the first event is *postponed* or *delayed* as much as possible. This ensures that the events involved in the race get *flipped* or re-ordered when the program is executed with the new schedule. The new schedule is used for the next execution.

We illustrate how jCUTE performs concolic testing along with race-detection and flipping using the sample program P in Figure 1. The program has two threads t_1 and t_2, a shared integer variable `x`, and an integer variable `z` which receives an input from the external environment at the beginning of the program. Each statement in the program is labeled. The program reaches the `ERROR` statement in thread t_2 if the input to the program is `1` (i.e., `z` gets the value `1`) and if the program executes the statements in the following order: $(t_2, 1)(t_1, 1)(t_2, 2)(t_2, 3)$, where each event, represented by a tuple of the form (t, l), in the sequence denotes that the thread t executes the statement labeled l.

```
x is a shared variable
z = input();

Thread t1                    Thread t2

1:  x = 3;                   1:  x = 2;
                             2:  if (2*z + 1 == x)
                             3:        ERROR;
```

Fig. 1. A Simple Shared-Memory Multi-Threaded Program

jCUTE first generates a random input for `z` and executes P with a default schedule. Without loss of generality, the default schedule always picks the thread which is enabled and which has the lowest index. Thus, the first execution of P is $(t_1, 1)(t_2, 1)(t_2, 2)$. Let z_0 be the symbolic value of `z` at the beginning of the execution. jCUTE collects the constraints from the predicates of the branches executed in this path. For this execution, jCUTE generates the path constraint $\langle 2 * z_0 + 1 != 2\rangle$. jCUTE also decides that there is a race condition between the first and the second event because both the events access the same variable `x` in different threads without holding a common lock and one of the accesses is a write of `x`.

Following the depth-first search strategy, jCUTE picks the only constraint $2 * z_0 + 1 != 2$, negates it, and tries to solve the negated constraint $2 * z_0 + 1 = 2$. This has no solution. Therefore, jCUTE backtracks and generates a schedule such that the next execution becomes $(t_2, 1)(t_2, 2)(t_1, 1)$ (here the thread involved in the first event of the race in the previous execution is delayed as much as possible). This execution re-orders the events involved in the race in the previous execution.

During the above execution, jCUTE generates the path constraint $\langle 2 * z_0 + 1! = 2\rangle$ and computes that there is a race between the second and the third events. Since the negated constraint $2 * z_0 + 1 = 2$ cannot be solved, jCUTE backtracks and generates a schedule such that the next execution becomes $(t_2, 1)(t_1, 1)(t_2, 2)$. This execution re-orders the events involved in the race in the previous execution.

In the above execution, jCUTE generates the path constraint $\langle 2*z_0+1! = 3\rangle$. jCUTE solves the negated constraint $2 * z_0 + 1 = 3$ to obtain $z_0 = 1$. In the next execution, it follows the same schedule as the previous execution. However, jCUTE starts the execution with the input variable `z` set to `1` which is the value of `z` that jCUTE computed by solving the constraint. The resultant execution becomes $(t_2, 1)(t_1, 1)(t_2, 2)(t_2, 3)$ which hits the `ERROR` statement of the program.

3 Execution Model

We assume that programs under test are written in a shared-memory multi-threaded imperative programming language such as Java. Such a program consists of a finite set of *threads*, which communicate by reading or writing shared variables, or by acquiring or releasing locks. Each thread executes a sequence of deterministic statements. Without loss of generality, we assume that the execution of a statement by a thread can perform at most one shared-memory operation–this can be achieved by splitting complex statements into a sequence of simple statements. We also assume that the execution of each thread terminates.[4]

A program supports mutual exclusion by using locks.[5] A thread suspends its execution if it tries to acquire a lock which is already acquired by another thread. Normal execution of the thread resumes when the lock is released by the other thread. We assume that the acquire and release of locks take place in a nested fashion as in Java. Locks are assumed to be *re-entrant*: if a thread already holds a lock on a shared variable, then an acquire of the lock on the same variable by the same thread does not deadlock. When the execution of a thread terminates, all the locks held by the thread are released. For technical simplicity, we assume that the set of memory locations that can be locked or unlocked is disjoint from the set of memory locations that can be read or written. We assume a sequentially consistent memory model.

We fix a multi-threaded program P. The execution of each statement in P is an event. Note that a statement may involve access to a shared memory location.

[4] In practice, this can be enforced by limiting the number of execution steps.

[5] Due to space limit, we do not describe how to handle other synchronization constructs. In our implementation, we handle all synchronization primitives of Java. We express `wait` as a sequence of lock release and acquire actions. A `join` operation on a parent thread is *sequentially related* to the termination event of the child thread. Handling of message-passing primitives were discussed in [13].

We represent an event as (t, l, a), where l is the label of the statement executed by thread t and a is the type of shared memory access in the statement. If the execution of the statement accesses a shared memory location, then $a = \mathbf{r}$ if the access is a read, $a = \mathbf{w}$ if the access is a write, $a = \mathbf{l}$ if the access is a lock, and $a = \mathbf{u}$ if the access is an unlock; otherwise, $a = \bot$. If the execution of a fork statement labeled l by a thread t creates a new thread t', then we get two events: $(t, l, \bot)$ representing the fork event on the thread t and $(t', \bot, \bot)$ representing the creation of the new thread. Thus the event $(t', \bot, \bot)$ represents the first event of any newly created thread t'. We use the term *access* to represent a read, a write, a lock, or an unlock of a shared memory location. We use the term *update* to represent a write, a lock, or an unlock of a shared memory location. We call an event

- a *fork event*, if the event is of the form $(t, l, \bot)$ and l is the label of a fork statement,
- a *new thread event*, if the event is of the form $(t, \bot, \bot)$,
- a *read*, a *write*, a *lock*, an *unlock*, an *access*, or an *update event*, if the event reads, writes, locks, unlocks, accesses, or updates a memory location, respectively,
- an *internal event*, if the event is not a fork event, a new thread event, or an access event.

An execution of P can be seen as a *sequence of events*. We call such a sequence an *execution path*. Note that the execution of P on several inputs may result in the same execution path. Let $\mathtt{Ex}(P)$ be the set of all feasible execution paths exhibited by the program P on all possible inputs and all possible choices by the scheduler.

If we view each event in an execution path as a node, then $\mathtt{Ex}(P)$ can be seen as a tree. Such a tree is called the *computation tree* of a program. The goal of our testing method for concurrent programs is to systematically explore a minimum possible subset of the execution paths of $\mathtt{Ex}(P)$ such that if a statement of P is reachable by a thread for some input and some schedule, the subset must contain an execution path in which that statement is executed. To achieve this, we abstract an execution path in terms of a partial order relation called *causal relation*. Any partial order represents a set of equivalent execution paths. In our testing algorithm, the goal is to exactly explore one execution path corresponding to each partial order. However, in the actual algorithm, we are able to guarantee that at least one—though *not* at most one—execution path corresponding to each partial order is explored if a program has no data input. We next define the various binary relations that we use to define a partial order.

In an execution path $\tau \in \mathtt{Ex}(P)$, any two events $e = (t_i, l_i, a_i)$ and $e' = (t_j, l_j, a_j)$ appearing in τ are *sequentially related* (denoted by $e \lhd e'$) iff:

1. $e = e'$, or
2. $t_i = t_j$ and e appears before e' in τ, or
3. $t_i \neq t_j$, t_i created the thread t_j, and e appears before e'' in τ, where e'' is the fork event on t_i creating the thread t_j, or

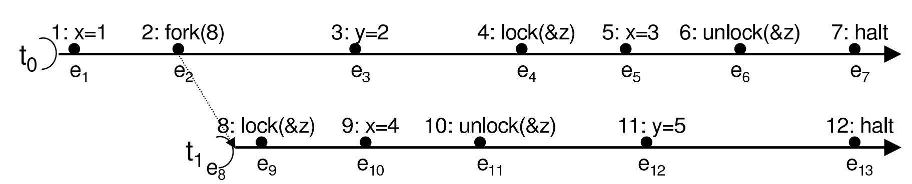

Fig. 2. Time increases from left to right. $e_3 \parallel e_{10}$, $e_9 \lessdot e_4$, $e_{10} \not\lessdot e_5$, $e_3 \lessdot e_{12}$, $e_1 \not\lessdot e_{10}$, $e_1 \preccurlyeq e_{10}$, $e_1 \preccurlyeq e_{12}$, $e_1 \lhd e_9$, $e_3 \lhd e_4$, $e_3 \Updownarrow e_{12}$, etc.

4. there exists an event e'' in τ such that $e \lhd e''$ and $e'' \lhd e'$.

Thus $\lhd$ is a partial order relation. We say $e \Updownarrow e'$ iff $e \not\lhd e'$ and $e' \not\lhd e$.

In an execution path $\tau \in \texttt{Ex}(P)$, any two events $e = (t_i, l_i, a_i)$ and $e' = (t_j, l_j, a_j)$ appearing in τ are *shared-memory access precedence related* (denoted by $e <_m e'$) iff:

1. e appears before e' in τ, and
2. e and e' both access the same memory location m, and
3. one of them is an update of m.

In the above definition, it is worth remembering that the memory locations that can be locked or unlocked are disjoint from the memory locations that can be read or written. Therefore, if $e <_m e'$ and e (or e') is a lock or unlock of m, then the e' (or e) is also a lock or unlock of m. Similarly, if $e <_m e'$ and e (or e') is a write of m, then the e' (or e) is a read or write of m.

Given the definition of the sequential relation and the shared-memory access precedence relation, we can define another relation, called *causal relation*, as follows. In an execution path $\tau \in \texttt{Ex}(P)$, any two events $e = (t_i, l_i, a_i)$ and $e' = (t_j, l_j, a_j)$ appearing in τ are *causally related* (denoted by $e \preccurlyeq e'$) iff:

1. $e \lhd e'$, or
2. $e <_m e'$ for some shared-memory location m, or
3. there exists e'' such that $e \preccurlyeq e''$ and $e'' \preccurlyeq e'$.

The causal relation is a partial-order relation. We say that $e \parallel e'$ iff $e \not\preccurlyeq e'$ and $e' \not\preccurlyeq e$. If $e \preccurlyeq e'$, then we say e *causally precedes* e'.

We next define a relation $\lessdot$, called *race relation*, that captures the race condition between two events. We say that any two events $e = (t_i, l_i, a_i)$ and $e' = (t_j, l_j, a_j)$ are *race related* (denoted by $e \lessdot e'$) iff

1. $e \Updownarrow e'$,
2. if e is a lock event and e'' is the corresponding unlock event, then $e'' <_m e'$ and there exists no e_1 such that $e_1 \neq e''$, $e_1 \neq e'$, $e'' \preccurlyeq e_1$, and $e_1 \preccurlyeq e'$, and
3. if e is a read or a write event, then $e <_m e'$ and there exists no e_1 such that $e_1 \neq e$, $e_1 \neq e'$, $e \preccurlyeq e_1$, and $e_1 \preccurlyeq e'$.

If two events in an execution path are related by $\lessdot$, then there exists an immediate *race* (data race or lock race) between the two events. Therefore, we call $\lessdot$ a *race* relation.

Figure 2 gives an example of the various relations defined above.

Given two execution paths τ and τ' in $\mathtt{Ex}(P)$, we say that τ and τ' are *causally equivalent*, denoted by $\tau \equiv_{\preccurlyeq} \tau'$, iff τ and τ' have the same set of events and they are linearizations of the same $\preccurlyeq$ relation. We use $[\tau]_{\equiv_{\preccurlyeq}}$ to denote the set of all executions in $\mathtt{Ex}$ that are equivalent to τ.

We define *a representative set of executions* $\mathtt{REx} \subseteq \mathtt{Ex}$ as a set that contains exactly one candidate from each equivalence class $[\tau]_{\equiv_{\preccurlyeq}}$ for all $\tau \in \mathtt{Ex}$. Formally, $\mathtt{REx}$ is a set such that following properties hold:

1. $\mathtt{REx} \subseteq \mathtt{Ex}$,
2. $\mathtt{Ex} = \bigcup_{\tau \in \mathtt{REx}} [\tau]_{\equiv_{\preccurlyeq}}$, and
3. for all $\tau, \tau' \in \mathtt{REx}$, it is the case that $\tau \not\equiv_{\preccurlyeq} \tau'$.

The following result shows that a systematic and automatic exploration of each element in $\mathtt{REx}$ is sufficient for testing.

Proposition 1. *If a statement is reachable in a program P for some input and schedule, then there exists a $\tau \in \mathtt{REx}$ such that the statement is executed in τ.*

The proof of this proposition is as follows. If a statement is reachable then there exists an execution τ in $\mathtt{Ex}$ such that the execution τ executes the statement. By the definition of $\equiv_{\preccurlyeq}$, any execution in $[\tau]_{\equiv_{\preccurlyeq}}$ executes the statement. Hence, the execution in $\mathtt{REx}$ that is equivalent to τ executes the statement.

The race-detection and flipping algorithm tries to explore all paths in a superset of $\mathtt{REx}(P)$ and a small subset of $\mathtt{Ex}(P)$. A key observation that guides our testing algorithm is that if two events are sequentially related then their happening order cannot be permuted by changing the schedule of the threads. However, if the two events are race related, then their happening order can be permuted by modifying the schedule. In our algorithm , we systematically permute or flip the race relation between various events by generating new schedules one by one.

4 The Race-Detection and Flipping Algorithm

We next describe the race-detection and flipping algorithm. For simplicity of exposition, we assume that a program under test has no data input. As illustrated in Section 2, a combination of race-detection and flipping algorithm and concolic testing can be used to systematically test a shared-memory multi-threaded program with data inputs. In the interest of space, this paper does not discuss the details of the combined method–the details can be found in [12].

The race-detection and flipping algorithm is given in Figure 3. Recall that $\mathtt{Ex}(P)$ is the set of all feasible execution paths that can be exhibited by the program P. Similarly, $\mathtt{REx}(P)$ is a set that contains exactly one candidate from each equivalence class of feasible execution paths of P. *test_program*(P) repeatedly executes the program P with different schedules until all paths in a $\mathtt{REx}(P)$

```
global var τ = ε; // the empty sequence

//input: P is the program to test
test_program(P)
    while testing not completed
        execute_program(P)

execute_program(P)
    execute_prefix(P, τ);
    while there is an enabled thread
        execute the next statement of the lowest indexed enabled thread in P
            to generate the event e;
        race(τ) = false;
        postponed(τ) = ∅;
        append e to τ;
        if ∃e' ∈ τ such that e' ⋖ e
            let τ = τ1 e' τ2 in race(τ1) = true;
    // end of the while loop
    if there is an active thread
        print ''Error: found deadlock'';
    generate_next_schedule();

// modifies τ
generate_next_schedule()
    if ∃e such that τ == τ1 e τ2 and backtrackable(τ1) and
        there is no e' such that τ == τ1' e' τ2' and |τ1| < |τ1'| and backtrackable(τ1')
            race(τ1) = false;
            let (t, _, _) = e in add t to postponed(τ1);
            let t = smallest indexed thread in enabled(τ1)\ postponed(τ1) in τ = τ1(t, _, _);
    else
        testing completed;

backtrackable(τ1) =
    race(τ1) ==true and |enabled(τ1)\postponed(τ1)| > 1
```

Fig. 3. The Race-Detection and Flipping Algorithm

have been explored. Given two sequences of events τ and τ', we let $\tau\tau'$ denote the concatenation of the two sequences. Similarly, given a sequence of events τ and an event e, we let τe to denote the concatenation of the sequence and the event. Let ϵ be the empty sequence. A sequence of events is called a *prefix*, if it is the prefix of a feasible execution path. The global variable τ keeps track of the execution path for each execution of P. At the end of each execution, τ is appropriately truncated so that a depth-first search of the computation tree takes place. *execute_prefix*(P, τ) executes the program from the beginning until the sequence of events generated by the execution is equal to the prefix τ. Since an execution path is solely determined by the sequence of threads that are executed in the path, from now onwards we will ignore the second and the third components of a tuple representing an event. Thus $(t, \_, \_)$ represents an event on the thread t. With every prefix τ, we associate a set, denoted by *postponed*(τ). Moreover, with every prefix τ, we associate a boolean flag, denoted by *race*(τ). *enabled*(τ) returns the set of threads that are enabled after executing the prefix τ. *enabled*$(\tau)\backslash$*postponed*(τ) represents the set of threads that are enabled but not postponed after executing τ.

In each execution of P during the testing process, P is first partly executed so that it follows the prefix τ computed in the previous execution. Then P is executed with the default schedule, where the lowest indexed enabled thread is always chosen. If $\tau = \tau' e$ before the start of an execution, then the execution path and the previous execution path has the same prefix τ'. In an execution path τ, for any prefix τ' of τ, we set $\mathit{race}(\tau')$ to **true**, if there exist e, τ_1, e', and τ_2 such that $\tau = \tau' e \tau_1 e' \tau_2$ and $e \lessdot e'$. The algorithm computes the $\lessdot$ relation at runtime using the dynamic vector clock algorithm [16, 12]. We omit the vector clock update procedures in the pseudo-code of the race-detection and flipping algorithm to keep the description simple. Setting $\mathit{race}(\tau')$ to **true** flags that in a subsequent execution, we must postpone the execution of e after the prefix τ' so that we may explore a possibly non-equivalent execution path. At the end of an execution, if τ_1 is the longest prefix of the current execution path τ such that $\mathit{race}(\tau_1)$ is set to **true** and $|\mathit{enabled}(\tau_1) \backslash \mathit{postponed}(\tau_1)| > 1$, we generate a new schedule by truncating τ to $\tau_1 e$, where e is an event of a thread t that has not been scheduled after τ_1 in any previous execution.

The following result holds for the race-detection and flipping algorithm.

Theorem 1. *If $\mathit{Ex}'(P)$ is the set of the execution paths that are explored by the race-detection and flipping algorithm, then there is a set $\mathit{REx}(P)$ such that $\mathit{REx}(P) \subseteq \mathit{Ex}'(P) \subseteq \mathit{Ex}(P)$.*

The proof of the above theorem can be found in [12].

5 Case Studies

For Java, we have implemented the combination of the race-detection and the flipping algorithm and concolic testing. The tool is called jCUTE. The details of the implementation can be found in the tool paper [14].

We use two sets of case studies to illustrate the effectiveness of jCUTE in finding potential bugs. The experiments were run on a 2.0 GHz Pentium M processor laptop with 1 GB RAM running Windows XP.

5.1 Java 1.4 Collection Library

We tested the thread-safe Collection framework implemented as part of the `java.util` package of the standard Java library provided by Sun Microsystems. A number of data structures provided by the package `java.util` are claimed as thread-safe in the Java API documentation. This implies that the library should provide the ability to safely manipulate multiple objects of these data structures simultaneously in multiple threads. No explicit locking of the objects should be required to safely manipulate the objects. More specifically, multiple invocation of methods on the objects of these data structures by multiple threads must be equivalent to a sequence of serial invocation of the same methods on the same objects by a single thread.

We chose this library as a case study primarily to evaluate the effectiveness of our jCUTE tool. As Sun Microsystems' Java is widely used, we did not expect to find potential bugs. We found several previously undocumented data races, deadlocks, uncaught exceptions, and an infinite loop in the library. Note that, although the number of potential bugs is high, these bugs are all caused by a couple of problematic design patterns used in the implementation.

Experimental Setup The `java.util` provides a set of classes implementing thread-safe Collection data structures. A few of them are ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, HashMap, TreeMap, etc. The Vector class is synchronized by implementation. For the other classes, one needs to call the static functions such as `Collections.synchronizedList`, `Collections.synchronizedSet`, etc., to get a synchronized or thread-safe object backed by a non-synchronized object of the class. To setup the testing process we wrote a multithreaded test driver for each such thread-safe class. The test driver starts by creating two empty objects of the class. The test driver also creates and starts a set of threads, where each thread executes a different method of either of the two objects concurrently. The invocation of the methods strictly follows the contract provided in the Java API documentation. We created two objects because some of the methods, such as `containsAll`, takes as an argument an object of the same type. For such methods, we call the method on one object and pass the other object as an argument. Note that more sophisticated test drivers can be written.

The arguments to the different methods are provided as input to the program. If a class is thread-safe, then there should be no error if the test-driver is executed with any possible interleaving of the threads and any input. However, jCUTE discovered data races, deadlocks, uncaught exceptions, and an infinite loop in these classes. Note that in each case jCUTE found no such error if methods are invoked in a single thread. As such the bugs detected in the Java Collection library are *concurrency related*.

The summary of the results is given in the Table 1. Here We present a simple scenario under which the infinite loop happens. The test driver first creates two synchronized linked lists by calling

```
List l1 = Collections.synchronizedList(new LinkedList());
List l2 = Collections.synchronizedList(new LinkedList());
l1.add(null);
l2.add(null);
```

The test driver then concurrently allows a new thread to invoke `l1.clear()` and another new thread to invoke `l2.containsAll(l1)`. jCUTE discovered an interleaving of the two threads that resulted in an infinite loop. However, the program never goes into infinite loop if the methods are invoked in any order by a single thread. jCUTE also provided a trace of the buggy execution. This helped us to detect the cause of the bug. The cause of the bug is as follows. The method `containsAll` holds the lock on `l2` throughout its execution. However, it acquires the lock on `l1` whenever it calls a method of `l1`. The method `clear` always holds the lock on `l1`. In the trace, we found that the first thread executes the statements

```
modCount++;
header.next = header.previous = header;
```

of the method `l1.clear()` and then there is a context switch before the execution of the statement `size=0;` by the first thread. The other thread starts executing the method `containsAll` by initializing an iterator on `l1` without holding a lock on `l1`. Since the field `size` of `l1` is not set to 0, the iterator assumes that `l1` still has one element. The iterator consumes the element and increments the field `nextIndex` to 1. Then a context switch occurs and the first thread sets `size` of `l1` to 0 and completes its execution. Then the other thread starts looping over the iterator. In each iteration `nextIndex` is incremented. The iteration continues if the method `hasNext` of the iterator returns true. Unfortunately, the method `hasNext` performs the check `nextIndex != size;` rather than checking `nextIndex < size;`. Since `size` is 0 and `nextIndex` is greater than 0, `hasNext` always returns true and hence the loop never terminates. Note that this infinite loop should not be confused with the infinite loop in the following wrongly coded sequential program commonly found in the literature.

```
List l = new LinkedList();  l.add(l); System.out.println(l);
```

Name	Run time in seconds	# of Paths	# of Threads	# of Functions Tested	# of data races/deadlocks/ infinite loops/exceptions
Vector	5519	20000	5	16	1/9/0/2
ArrayList	6811	20000	5	16	3/9/0/3
LinkedList	4401	11523	5	15	3/3/1/1
LinkedHashSet	7303	20000	5	20	3/9/0/2
TreeSet	7333	20000	5	26	4/9/0/2
HashSet	7449	20000	5	20	19/9/0/2

Table 1. Results for testing synchronized Collection classes of JDK 1.4

5.2 NASA's Java Pathfinder's Case Studies

In [10], several case studies have been carried out using Java PathFinder and Bandera. These case studies involve several small to medium-sized multithreaded Java programs; thus they provide a good suite to evaluate jCUTE. The programs include RemoteAgent, a Java version of a component of an embedded spacecraft-control application, Pipeline, a framework for implementing multithreaded staged calculations, RWVSN, Doug Lea's framework for reader writer synchronization, DEOS, a Java version of the scheduler from a real-time executive for avionics systems, BoundedBuffer, a Java implementation of multithreaded bounded buffer, NestedMonitor, a semaphore based implementation of bounded buffer, and ReplicatedWorkers, a parameterizable job scheduler. Details about these programs can be found in [10]. We also considered a distributed sorting implementation used in [8]. This implementation involves both concurrency and complex data inputs.

We used jCUTE to test these programs. Since most of these programs are designed to run in an infinite loop, we bounded our search to a finite depth. jCUTE discovered known concurrency related errors in RemoteAgent, DEOS, BoundedBuffer, NestedMonitor, and the distributed sorting implementation and seeded bugs in Pipeline, RWVSN, and ReplicatedWorkers. The summary of the results is given in the Table 2. In each case, we stopped at the first error. Note the although the running time of our experiments is many times smaller than that in [10, 8], we are also using a much faster machine.

It is worth mentioning that we tested the *un-abstracted version* of these programs rather than requiring a programmer to manually provide abstract interpretations as in [10]. This is possible with jCUTE because jCUTE tries to explore distinct paths of a program rather than exploring distinct states. Obviously, this means that we cannot prove a program correct if the program has infinite length paths. Java PathFinder and Bandera can verify a program in such cases if the state space of the abstracted program is finite.

Name	Run time in seconds	# of Paths	# of Threads	Lines of Code	# of Bugs Found data races/deadlocks/ assertions/exceptions
BoundedBuffer	11.41	43	9	127	0/1/0/0
NestedMonitor	0.46	2	3	214	0/1/0/0
Pipeline	0.70	3	5	103	1/0/1/0
RemoteAgent	0.45	2	3	55	1/1/0/0
RWVSN	2.19	8	5	590	1/0/1/0
ReplicatedWorkers	0.34	1	5	954	0/0/1/0
DEOS	35.23	111	6	1443	0/0/1/0

Table 2. Java PathFinder's Case Studies (un-abstracted)

6 Related Work

Bruening [1] first proposed a technique for *dynamic partial order reduction*, called ExitBlock-RW algorithm, to systematically test multithreaded programs. They used two sets, *delayed set* and *enabled set*, similar to the sets *postponed* and T_{enabled} in our algorithm, to enumerate meaningful schedules by re-ordering dependent atomic blocks. However, they assume that the program under test follows a consistent mutual-exclusion discipline using locks. The dynamic partial order reduction technique proposed by Carver and Lei [2] guarantees that exactly one interleaving for each partial order is explored. However, the approach involves storing schedules that have not been yet explored; this can become a memory bottleneck.

More recently, dynamic partial order reduction proposed by Flanagan and Godefroid [4] removes the memory bottleneck in [2] at the cost of possibly exploring more than one interleaving for each partial order. This technique uses dynamically constructed *persistent sets* and *sleep sets* [5] to prune the search space. The key difference between the DPOR algorithm in [4] and our race-detection and flipping algorithm is that,for every choice point, the DPOR algorithm in [4] uses a persistent set and we use a postponed set. These two sets can be different at a choice point. For example, for the 3-threaded program in Figure 4, if the first execution path is $(t_1, 1, \mathbf{w})(t_2, 2, \mathbf{w})(t_3, 3, \mathbf{w})$, then at the first choice point denoting the initial state of the program, the persistent set is $\{t_1, t_3\}$; whereas, at the same choice point, the postponed set is $\{t_1\}$. (Apart from scheduling the thread t_1, the race-detection and flipping algorithm also schedules the thread t_2 at the first choice point.) Note that the DPOR algorithm in [4] picks the elements of a persistent set by using a complex forward lookup algorithm. In contrast, we *simply* put the current scheduled thread to the postponed set at a choice point.

t_1:	t_2:	t_3:
`1: x = 1;`	`2: y = 4;`	`3: x = 2;`

Fig. 4. A Three-Threaded Program

Moreover, the implementation in [4] considers two read accesses to the same memory location by different threads to be dependent. Thus for the 3-threaded program in Figure 5, the implementation described in [4] would explore six interleavings. We remove the redundancy associated with this assumption by using a more general notion of race and its detection using dynamic vector clock algorithm. As such, for the above example, we will explore only one interleaving. Note that none of the previous descriptions of the above dynamic partial order reduction techniques have handled programs which have inputs.

t_1:	t_2:	t_3:
`1: lv1 = x;`	`2: lv2 = x;`	`3: if (x > 0)` `4:    ERROR;`

Fig. 5. Another Three-Threaded Program

In [13] concolic testing has been extended to test asynchronous message-passing Java programs written using a Java Actor library. Shared memory systems can be modeled as asynchronous message passing systems by associating a thread with every memory location. Reads and writes of a memory location can be modeled as asynchronous messages to the thread associated with the memory location. However, this particular model would treat both reads and writes similarly. Hence, the algorithm in [13] would explore many redundant executions. For example, for the 2-threaded program $t_1 : x = 1; x = 2; t_2 : y = 3; x = 4;$, the algorithm in [13] would explore six interleavings. Our algorithm assumes that two reads are not in race and thus would explore only three interleavings of the program.

In a similar independent work [18], Siegel et al. uses a combination of symbolic execution and static partial order reduction to check if a parallel numerical program is equivalent to a simpler sequential version of the program. However, their main emphasis is in symbolic execution of numerical programs with floating points, rather than programs with pointers and data-structures. Therefore, static partial order reduction proves effective in their approach.

Model checking tools [19, 3] based on static analysis have been developed, which can detect bugs in concurrent programs. These tools employ partial order reduction techniques to reduce search space. The partial order reduction depends on detection of thread-local memory locations and patterns of lock acquisition and release.

7 Conclusion

We presented an efficient algorithm for testing multithreaded programs. A pure symbolic execution based testing algorithm for concurrent programs may end up exploring redundant execution paths having the same partial order. This is because optimal partial order reduction requires accurate knowledge of dependency relation; such knowledge may not be computable due to inaccuracies of alias analysis during symbolic execution. On the other hand, a pure concrete execution based testing algorithm for concurrent programs requires the exploration of all partial orders for all possible inputs. This may not scale up if the domain of inputs is large. Our algorithm addresses the limitations of both these approaches by extending concolic testing with the race-detection and flipping algorithm. The concrete execution of concolic testing helps to resolve aliases exactly at runtime. As a result we get the exact dependency or causal relation among the events. The symbolic execution helps to generate a small set of inputs from a large domain of inputs through constraint solving. Therefore, we believe that concolic execution combined with the race-detection and flipping algorithm is an attractive technique to test concurrent programs.

References

1. D. Bruening. Systematic testing of multithreaded Java programs. Master's thesis, MIT, 1999.
2. R. H. Carver and Y. Lei. A general model for reachability testing of concurrent programs. In *6th International Conference on Formal Engineering Methods (ICFEM'04)*, volume 3308 of *LNCS*, pages 76–98, 2004.
3. J. Corbett, M. B. Dwyer, J. Hatcliff, C. S. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In *Proc. of ICSE'00: International Conference on Software Engineering*, Limerich, Ireland, June 2000. ACM Press.
4. C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proc. of the 32nd Symposium on Principles of Programming Languages (POPL'05)*, pages 110–121, 2005.
5. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*, volume 1032 of *LNCS*. Springer-Verlag, 1996.
6. P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *24th ACM Symposium on Principles of Programming Languages*, pages 174–186, 1997.
7. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, 2005.
8. S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. 9th Int. Conf. on TACAS*, pages 553–568, 2003.
9. J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.
10. C. S. Pasareanu, M. B. Dwyer, and W. Visser. Finding feasible abstract counter-examples. *International Journal on Software Tools for Technology Transfer (STTT'03)*, 5(1):34–48, 2003.

11. D. Peled. All from one, one for all: on model checking using representatives. In *5th Conference on Computer Aided Verification*, pages 409–423, 1993.
12. K. Sen. *Scalable Automated Methods for Dynamic Program Analysis*. PhD thesis, University of Illinois at Urbana-Champaign, June 2006.
13. K. Sen and G. Agha. Automated systematic testing of open distributed programs. In *International Conference on Fundamental Approaches to Software Engineering (FASE'06)*, LNCS (To appear), 2006.
14. K. Sen and G. Agha. CUTE and jCUTE : Concolic unit testing and explicit path model-checking tools. In *Computer Aided Verification (CAV'06)*, LNCS, 2006. (To Appear).
15. K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*. ACM, 2005.
16. K. Sen, G. Roşu, and G. Agha. Runtime Safety Analysis of Multithreaded Programs. In *9th European Software Engineering Conference and 11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE'03)*, pages 337–346. ACM, 2003.
17. K. Sen, G. Roşu, and G. Agha. Online efficient predictive safety analysis of multithreaded programs. *International Journal on Software Technology and Tools Transfer*, 2006.
18. S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. Technical Report UM-CS-2005-15, University of Massachusetts Department of Computer Science, 2005.
19. S. D. Stoller. Model-Checking Multi-Threaded Distributed Java Programs. In *Proc. of SPIN'00: SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 224–244. Springer, 2000.
20. A. Valmari. Stubborn sets for reduced state space generation. In *10th Conference on Applications and Theory of Petri Nets*, pages 491–515, 1991.
21. W. Visser, C. Pasareanu, and S. Khurshid. Test Input Generation with Java PathFinder. In *Proceedings of ACM SIGSOFT ISSTA'04*, pages 97–107, 2004.