Concolic Testing of Multithreaded Programs and Its Application to Testing Security Protocols

Koushik Sen and Gul Agha
Department of Computer Science
University of Illinois at Urbana-Champaign, USA.
{ksen,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. We propose a novel testing algorithm for concurrent programs in which our goal is not only to execute all reachable statements of a program, but to detect all possible data races, and deadlock states. The algorithm uses a combination of symbolic and concrete execution (called concolic execution) to explore all distinct causal structures (or partial order relations among events generated during execution) of a concurrent program. The idea of concolic testing is to use the 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. Our algorithm uses the concrete execution to compute the exact race conditions between the events of an execution at runtime. 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 at least one representative from each partial order. We describe jCUTE, a tool implementing the testing algorithm together with the results of applying jCUTE to real-world multithreaded Java applications and libraries. In one of our case studies, 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 [28]. 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* [11, 20, 31].

A number of approaches [1, 2, 9, 12] to testing concurrent programs assume that the data inputs are from a 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 [15, 32]. 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 use *concolic testing*, which combines concrete and symbolic execution by using one to guide the other [14, 24, 25]. Specifically, the idea 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. We briefly explain this further in Section 5.2.

To use concolic testing for multithreaded programs, we do the following. 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 publicly available tool, called jCUTE, for testing Java programs. Apart from detecting assertion violations and uncaught exceptions, jCUTE reports all data race conditions and deadlock states encountered during the process of testing. jCUTE also supports testing of asynchronous message-passing Java programs written using a Java Actor library. This work was reported in [24]. Note that the technique in [24], which works for asynchronous message passing processes or actors, is not directly applicable to multithreaded programs.

We provide some case studies to illustrate the utility of our approach. In our first case study, we tested the thread-

UIUC Technical Report # UIUCDCS-R-2006-2676

safe Java Collection framework provided with the Sun Microsystems' Java 1.4. Surprisingly, we discovered several data races, deadlocks, uncaught exceptions, and an infinite loop in this widely used library. All of them are potential bugs related to multithreaded 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 outline of the rest of the paper is as follows. In Section 2, we give an overview of our approach through an example. In Section 3 and Section 4, we describe the programming model and execution model that we assume for the purpose of describing our algorithms. In Section 5, we describe a simple naïve testing algorithm. Section 6 describes an efficient algorithm for testing. In Section 7, we propose a further optimization. Section 8 gives some implementation details and four case studies. In Section 9, 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 [16], 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 they are events of 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 first introduced in [26].

The algorithm first generates a random input and a schedule, which specifies the order of 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 event is *postponed* or *delayed* as much as possible. This ensures that the events involved in the race gets *flipped* or re-ordered when the program is executed with the new schedule. The new schedule is used for the next execution.

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

t1 :               t2 :
1: x = 3;          1: x = 2;
                   2: if (2 * z + 1 == x)
                   3:    ERROR;
```

Figure 1: Simple Example

A complication arises from the fact that for some symbolic constraints, our constraint solver may not be powerful enough to compute concrete values that satisfy the constraints. To address this difficulty, such symbolic constraints *are simplified by replacing some of the symbolic values with concrete values.* Because of this, our algorithm is complete only if given an oracle that can solve all constraints in a program, and the length and the number of paths is finite. Note that because the algorithm does concrete executions, it is sound, i.e. all bugs it finds are real (assuming that the potential data race conditions and deadlocks found are not unintended).

We illustrate the testing algorithm 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 the program executes the statements in the following sequence: $(t_2, 1)(t_1, 1)(t_2, 2)(t_2, 3)$, where each event or tuple (t, l) in the sequence denotes that thread t executes the statement labeled l.

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 variable representing the 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 $2 * z_0 + 1 != 2$. 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 $2 * z_0 + 1! = 2$ and computes a race between the second and the third event. 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 $2 * z_0 + 1! = 3$. 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. PROGRAMMING MODEL

In order to simplify the description of our testing approach, we define a simple concurrent imperative language, Scil (Figure 3). Scil extends the simple language presented in [25] with concurrency primitives. A Scil program consists of a sequence of labeled statements. A Scil program starts its execution with a single thread t_0. Subsequently, it can create new threads. Threads in a program communicate by acquiring and releasing locks and by accessing (reading or writing) shared variables.

A Scil program may receive data inputs from its environment. Observe that the availability of an input earlier than its use does not affect an execution. Without loss of generality, we assume that all such inputs are available from the beginning of an execution; again this assumption simplifies the description of our algorithm. To further simplify our exposition, we assume that a Scil program has no pointers and functions. However, as in [25], our algorithm can be extended to programs with pointers, complex data structures [33], and functions; this is done in the implementation.

We now informally describe the semantics of Scil. Consider a Scil program P consisting of a sequence of statements, each of which is labeled. If l is the label of a statement in some thread, then $l + 1$ is the label of the next statement in that thread, unless the statement labeled by l is a `HALT` or an `ERROR`. The execution of a thread terminates when it executes a `HALT` or an `ERROR` statement.

During the execution of a Scil program, a single thread, namely t_0, starts by executing the first statement of the program. This thread t_0 is comparable with the `main` thread in Java. The initial thread t_0 or any subsequently created thread in the program can create new threads by calling the statement $fork(l)$, where l is the label of the statement to be executed by the newly created thread of the program.

A Scil program gets input using the expression $input()$. Observe that $input()$ captures the various functions through which a program in Java may receive data from its external environment.

A program may have two kinds of variables: variables local to a thread (denoted by lv) and variables shared among threads (denoted by v). Each thread maintains a local copy of the local variables. A program supports mutual exclusion by using locks: $lock(v)$ denotes the acquisition of the lock v and $unlock(v)$ denotes the release of the lock v. 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. Note that a given statement can have at most one shared variable access (i.e. read, write, lock, or unlock).

```
scheduler()
  $T_{\text{enabled}} = \{t_0\}$; $t_{\text{current}} = \textbf{null}$;
  $pc_{t_0}$ = the label of the first statement;
  while ($T_{\text{enabled}} \neq \emptyset$)
    $t_{\text{current}}$ =choose_next_thread($T_{\text{enabled}}$);
    $s$ =statement_at($pc_{t_{\text{current}}}$);
    execute_concrete(, s);
    $T_{\text{enabled}}$ = set of enabled threads;
  if there is an active thread
    print "Found deadlock";
```

Figure 2: Default Scheduler for Scil

$$\begin{array}{rll}
P & ::= & Stmt^* \\
Stmt & ::= & l \colon S \\
S & ::= & v \leftarrow lv \mid lv \leftarrow v \mid lv \leftarrow e \mid \texttt{if } p \texttt{ goto } l' \\
 & & \mid fork(l) \mid lock(v) \mid unlock(v) \mid \texttt{HALT} \mid \texttt{ERROR} \\
e & ::= & c \mid lv\ op\ lv \mid input() \\
 & & \text{where } op \in \{+, -, /, *, \%, \ldots\}, v \text{ is a shared variable,} \\
 & & lv \text{ is a variable local to a thread, } c \text{ is a constant} \\
p & ::= & lv = lv \mid lv \neq lv \mid lv < lv \mid lv \leq lv \mid lv \geq lv \mid lv > lv
\end{array}$$

Figure 3: Syntax of Scil

The semantics of a program in the language is given using a scheduler. The scheduler runs in a loop as described in Figure 2. We use the term *schedule* to refer to the sequence of choices of threads made by the scheduler during an execution.

On executing a statement $lock(v)$, a thread waits if the lock v is already held by another thread. Otherwise, the thread acquires the lock and continues its execution. A lock v already held a thread t is released when t executes a statement of the form $unlock(v)$. Note that in Figure 2, the set T_{enabled} represents the set of enabled threads. Initially, the thread t_0 is enabled. A thread is said to be *active* if it has been created and it has not already executed a `HALT` or an `ERROR` statement. A thread is said to be *enabled* if it is active and it is not waiting to acquire a lock.

A variable pc_t represents the program counter of the thread t. pc_{t_0} is initialized to the label of the first statement of the program. Inside the loop, the scheduler *non-deterministically* chooses a thread t_{current} from the set T_{enabled}. It executes the next statement of the thread t_{current}, where the next statement is obtained by calling `statement_at`($pc_{t_{\text{current}}}$). During the execution of the statement the program counter $pc_{t_{\text{current}}}$ of the thread t_{current} is incremented by 1 except if the statement is of the form `if` p `goto` l' and the predicate p in the statement evaluates to true (in that case pc_t is set to l'). The execution of a statement of the form $fork(l)$ creates a new thread, makes it active, and the program counter pc_t, where t is the newly created thread, is initialized to l. The loop of the scheduler terminates when the set of enabled threads T_{enabled} is empty. The termination of the scheduler indicates either the normal termination of a program execution when the set of active threads is empty, or a deadlock state when the set of active threads is non-empty.

4. EXECUTION MODEL

In an execution of P, an event is a tuple of the form (t, l, m, a) representing the execution of the statement labeled l by the thread t. The shared memory location m is accessed during the execution of the statement; the access to m is a read access if $a = \mathbf{r}$, a write access if $a = \mathbf{w}$, a lock access if $a = \mathbf{l}$, and an unlock access if $a = \mathbf{u}$. If the execu-

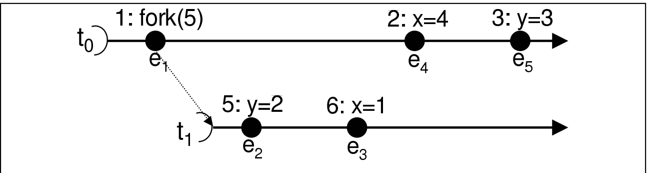

Figure 4: Time increases from left to right. $e_1 \lhd e_2$, $e_1 \lhd e_4$, $e_4 \lhd e_5$, $e_2 \lhd e_3$, $e_3 \parallel e_4$, $e_2 \parallel e_4$, $e_3 \lessdot e_4$, $e_2 \not\lessdot e_5$, $e_2 \not\lessdot e_4$, $e_3 \preccurlyeq e_4$, $e_3 \preccurlyeq e_5$, **etc.**

tion of the statement accesses no shared memory location, then both m and a are $\bot$. An execution of P can be seen as a sequence of such events. We call such a sequence an *execution path.* Let $\texttt{Ex}(P)$ be the set of all feasible execution paths exhibited by the program P on all possible inputs and all possible choices by the scheduler.

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

1. $e = e'$, or
2. $t_i = t_j$ and e happens before e', or
3. $t_i \neq t_j$, t_i created the thread t_j, e happens before t_j is created, or
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 $e \parallel e'$ iff $e \not\lhd e'$ and $e' \not\lhd e$.

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

1. $t^i \neq t^j$ and e happens before e',
2. $m_i = m_j$,
3. $e \parallel e'$,
4. there exists no e_1 and e_2 such that $e \lhd e_1$, $e_2 \lhd e'$, $e_1 \lessdot e_2$, and it is not the case that both $e = e_1$ and $e' = e_2$, and
5. one of the following holds:
   - $a_i = \mathbf{w}$ and $a_j = \mathbf{w}$,
   - $a_i = \mathbf{r}$ and $a_j = \mathbf{w}$,
   - $a_i = \mathbf{w}$ and $a_j = \mathbf{r}$,
   - $a_i = \mathbf{u}$ and $a_j = \mathbf{l}$.

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

Let the relation $\preccurlyeq$, called the *causal relation*, be the transitive closure of the relation $\lhd \cup \lessdot$. The relation $\preccurlyeq$ is a partial order. Figure 4 gives an example of the various relations defined above.

Given two execution paths τ and τ' in $\texttt{Ex}(P)$, 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 $\texttt{Ex}$ that are equivalent to τ.

We define *a representative set of executions* $\texttt{REx} \subseteq \texttt{Ex}$ as the set that contains exactly one candidate from each equivalence class $[\tau]_{\equiv_{\preccurlyeq}}$ for all $\tau \in \texttt{Ex}$. Formally, $\texttt{REx}$ is a set such that following properties hold: $\texttt{REx} \subseteq \texttt{Ex}$, $\texttt{Ex} = \bigcup_{\tau \in \texttt{REx}} [\tau]_{\equiv_{\preccurlyeq}}$, and for all $\tau, \tau' \in \texttt{REx}$, it is the case that $\tau \not\equiv_{\preccurlyeq} \tau'$.

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

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

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

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 efficient testing algorithm (see Section 6 and Section 7, we systematically permute or flip the race relation between various events by generating new schedules one by one.

The causal relation between the events in an execution can be tracked efficiently at runtime using *dynamic vector clocks* (DVC) first introduced in [26]. Dynamic vector clocks, which respect the fact that two reads can be permuted, extends the standard vector clocks [7] found in message passing systems. A dynamic vector clock $V : T \to \mathbb{N}$, where T is the set of threads that are present in the execution. We call such a map a *dynamic vector clock (DVC)* because its partiality reflects the intuition that threads are dynamically created and destroyed. To simplify the exposition and the implementation, we assume that each DVC V is a total map, where $V(t) = 0$ whenever V is not defined on thread t.

We associate a DVC with every thread t and denote it by V_t. Moreover, we associate two DVCs V_m^a and V_m^w with every shared memory m; we call the former *access DVC* and the latter *write DVC*. For DVCs V and V', we say that $V \leq V'$ if and only if $V(t) \leq V'(t)$ for all $t \in T$. We say that $V \neq V'$ if and only if $V \not\leq V'$ and $V' \not\leq V$. $\max\{V, V'\}$ is the DVC with $\max\{V, V'\}(t) = \max\{V(t), V'(t)\}$ for each $t \in T$.

At the beginning of an execution, all vector clocks associated with threads and memory locations are empty. Whenever a thread t with current DVC V_t processes event e, the following algorithm $\mathcal{A}$ is executed:

```
1. If $e$ involves the read. write, locking, or unlocking of
   a shared memory location $m$, then $V_t(t) \leftarrow V_t(t) + 1$.

2. If $e_i^k$ is a read of $m$ then
       $V_t \leftarrow \max\{V_t, V_m^w\}$
       $V_m^a \leftarrow \max\{V_m^a, V_t\}$

3. If $e_i^k$ is a write, locking, or unlocking of $m$ then
       $V_m^w \leftarrow V_m^a \leftarrow V_t \leftarrow \max\{V_m^a, V_t\}$

4. If $e$ is the execution of the statement of the form $fork(l)$
   and if $t'$ is the newly created thread then
       $V_{t'} \leftarrow V_t$
       $V_t(t) \leftarrow V_t(t) + 1$
       $V_{t'}(t') \leftarrow V_{t'}(t') + 1$
```

We can associate a DVC with every event e, denoted by V_e as follows. If e is executed by t and if VC_t is the vector clock of t just after the event e, then $VC_e = VC_t$. If e is an

```
// input: $P$ is the program to test
$run\_jCUTE(P)$
  $completed$=false; $\mathcal{I}$ = $path\_c$ = $branch\_hist$=[ ];
  while not $completed$
    ($\mathcal{I}$,$path\_c$,$branch\_hist$,$completed$)
          = $scheduler$($\mathcal{I}$,$path\_c$,$branch\_hist$);
```

Figure 5: Simple Algorithm

event of thread t, then the event in thread t that happened immediately before e is denoted by $prev(e)$.

In an execution, if we update the DVCs according to $\mathcal{A}$, then the following theorems [26] hold:

Theorem 2. *For any two events e and e', $e \preccurlyeq e'$ iff $V_e \leq V_{e'}$.*

Theorem 3. *If $V_e \neq V_{prev(e')}$ and $V_e \leq V_{e'}$, then $e \lessdot e'$, i.e. e and e' are in immediate race.*

In order to simplify the presentation of the algorithm, we first present a simple algorithm to systematically explore all execution paths in $\texttt{Ex}(P)$. However, in general, the number of execution paths in $\texttt{Ex}(P)$ is exponentially larger than that in $\texttt{REx}(P)$. In Section 6 and Section 7, we will refine the algorithm to avoid exploring almost all paths that are equivalent under the causal equivalence relation we defined above.

5. SIMPLE ALGORITHM

Our algorithms interpret programs in Scil. This is to simplify the description. However, note that the implementation of our algorithm for Java uses bytecode instrumentation and then executes the instrumented bytecode in the normal way.

Given a program P, our simple algorithm explores all execution paths in $\texttt{Ex}(P)$ (i.e., all feasible execution paths exhibited by P on all possible inputs and all possible choices by the scheduler). This is done by repeatedly executing P both symbolically and concretely on different inputs and schedules, each of which lead the program along a different execution path. At the end of each execution of P, our algorithm either computes a new schedule or a new input, which is used in the next execution of P. A new input is generated by solving constraints. To generate a new schedule, our algorithm picks a scheduler choice recorded during the execution and generates a new schedule where the particular thread chosen in the scheduler choice is *postponed*.

Our goal in describing the simple algorithm is to familiarize the readers with the various data structures that we use consistently in our algorithms. These data structures keep track of constraints, schedules, events, inputs, etc. in an execution and between executions. The data structures that the algorithm maintains between executions to pass the inputs and schedules computed in one execution of the program to the next execution are as follows: a logical input map $\mathcal{I}$ which stores the computed input values (see [25]), a sequences *path_c* which records the sequence of scheduler choices made and symbolic constraints computed during an execution, and a sequence *branch_hist* which keep track of the branches taken in an execution. The sequence *branch_hist* is only used by the concolic execution. Each element of the sequence *path_c* represents an execution point in the current execution. Such an element has the following fields:

- *constraint*: stores the constraint generated on the execution of a conditional statement at the execution point represented by the element. At the end of an execution, the conjunction of all the constraints stored in the elements of *path_c* gives the path constraint for the given execution path. (Note that in [25], each element of *path_c* was used to store only a constraint since in that work we were not concerned about concurrency).
- *event*: stores the event (as defined in Section 3) that is generated at the execution point represented by the element.
- *postponed*: contains a set of threads that cannot be executed in the next execution of P at the execution point represented by the element.
- *enabled*: contains a set of threads that are not enabled at the execution point represented by the element.
- *hasRace*: set to true if the event stored in the field *event* has an immediate race with some other future event in the execution path. The simple algorithm does not use this field; the field will be used in the efficient algorithm described in Section 6.

For each element of *path_c*, either the field *constraint* is set to a proper value or the the element's *event*, *postponed*, and *isRace* fields are set to proper values. The fields *constraint*, *enabled*, and *isRace* are only used at the end of an execution to generate a new schedule or a new input; the values of these fields are not used between executions of P. If the event e is stored in the field *event*, then we will use the term *postponed set of the event* e to refer to the set stored in the field *postponed*.

The psuedo-code for our algorithm is in Figure 5. Before starting the loop which repeatedly executes P, the algorithm initializes $\mathcal{I}$ to an empty map (see [25] for details), and both *path_c* and *branch_hist* to empty sequences. The simple scheduler, like the default scheduler in Figure 2, first initializes T_{enabled} to the set of enabled threads (i.e. $\{t_0\}$) and the program counter pc_{t_0} is initialized to the label of the first statement of the program. In addition, the simple scheduler also initializes the global counter variable i to 0. At any point in an execution, i contains the sum of the number of statements executed thus far, as well as the number of conditional statements executed. The input to the program P is also initialized using the logical input map $\mathcal{I}$ (see [25] for details). In the function `choose_next_thread`, the scheduler picks the same thread as the previous execution as long as i is less than the number of elements of *path_c*. This ensures that the current execution follows the same schedule as the previous execution as long as i is less than the length of *path_c*. The sequence *path_c* is truncated appropriately at the end of the previous execution to perform a depth first search of the execution paths in $\texttt{Ex}(P)$. Otherwise, if i is greater than or equal to the number of elements in *path_c*, the scheduler selects the smallest indexed thread that it is enabled.

The function `nextEvent` takes a thread t as an argument and returns the event that will happen if the thread t executes next. Specifically, if t accesses the shared memory location m while executing its next statement (i.e. $\texttt{statement\_at}(pc_t)$) and the access type is a, then $\texttt{nextEvent}(t) = (t, pc_t, m, a)$. The scheduler determines the next statement, say s, to be executed by t_{current} by calling the function $\texttt{statement\_at}(t_{\text{current}})$. Then t_{current} executes the statement s concolically, that is both concretely and symbolically in a cooperative way. Concolic execution (see

```
scheduler($\mathcal{I}$,$path\_c$,$branch\_hist$)
  $T_{\text{enabled}} = \{t_0\}$; $t_{\text{current}}$ = null; $i = 0$;
  $pc_{t_0}$ = the label of the first statement
  initialize input variables using $\mathcal{I}$;
  while ($T_{\text{enabled}} \neq \emptyset$)
    $t_{\text{current}}$ = choose_next_thread($T_{\text{enabled}}, i$);
    $path\_c[i].event$ = nextEvent($t_{\text{current}}$);
    $path\_c[i].enabled = T_{\text{enabled}}$;
    $i = i + 1$;
    $s$ =statement_at($pc_{t_{\text{current}}}$);
    ($i$,$path\_c$,$branch\_hist$)
          = execute_concolic($t_{\text{current}}, s, i$,$path\_c$,$branch\_hist$);
    $T_{\text{enabled}}$ = set of enabled threads;
  if there is an active thread
    print "Found deadlock";
  return compute_next_input_and_schedule($i$,$\mathcal{I}$,$path\_c$,$branch\_hist$);

choose_next_thread($T_{\text{enabled}}, i$)
  if $i < |path\_c|$
    // schedule the thread as in the previous execution
    $(t, l, m, a) = path\_c[i].event$;
    return $t$;
  else
    return smallest indexed thread from $T_{\text{enabled}}$;
```

Figure 6: Simple Testing Scheduler

```
compute_next_input_and_schedule($i$,$\mathcal{I}$,$path\_c$,$branch\_hist$)
  for ($j = i - 1$ ; $j \geq 0$ ; $j = j - 1$)
    if $path\_c[j].event$ has proper value
      // compute a new schedule
      if $|path\_c[j].enabled| > |path\_c[j].postponed| + 1$
        $(t, l, m, a) = path\_c[j].event$;
        $path\_c[j].postponed = path\_c[j].postponed \cup \{t\}$;
        $t$ = smallest indexed thread in
              $path\_c[j].enabled \setminus path\_c[j].postponed$;
        $path\_c[j].event = (t, l, m, a)$ ;
        return ($\mathcal{I}$,$path\_c[0 \ldots j]$,$branch\_hist[0 \ldots j]$,false);
    else
      // compute a new input
      if (not $branch\_hist[j].done$)
        $branch\_hist[j].branch = \neg branch\_hist[j].branch$;
        if ($\exists \mathcal{I}'$ that satisfies neg_last($path\_c[0 \ldots j]$))
          return ($\mathcal{I}'$,$path\_c[0 \ldots j]$,$branch\_hist[0 \ldots j]$,false);
    return ([ ],[ ],[ ],true);
```

Figure 7: Compute Next Schedule or Input

Section 5.2) updates both the concrete and the symbolic state of the program. It also updates the elements $path\_c[i]$ and $branch\_hist[i]$ and increments i by 1, if the statement s is a conditional statement. After the concolic execution of the statement s, the set of enabled threads T_{enabled} is updated appropriately. If the set of enabled threads is empty and there is no active thread, the loop terminates indicating the termination of the program execution. Otherwise, if the set of enabled threads is empty and there is at least one active thread, then a warning is given that there is a deadlock in the program. After the termination of the program execution, a schedule and an input for the next execution in our testing approach is computed using the procedure `compute_next_input_and_schedule` described in section 5.1.

5.1 Computing Next Schedule and Input

The function `compute_next_input_and_schedule` (see Figure 7) computes the schedule and the input that will direct the next program execution along an alternative execution path. It loops over the elements of *path_c* from the end, until a new schedule or a new input is generated. If the selected element inside the loop contains a scheduler choice and if not all scheduler choices at the execution point denoted by the element have been exercised, then a new schedule is generated. Specifically, if the thread t executed at the execution point denoted by the element $path\_c[j]$ and if t can be added to $path\_c[j].postponed$ without making $path\_c[j].postponed$ equal to the set of enabled threads at that point of execution, then t is added to the set $path\_c[j].postponed$. Moreover, the smallest indexed thread which is in the set of enabled threads at that point of execution and not in the set $path\_c[j].postponed$ is chosen and assigned to the event denoted by $path\_c[j].event$. This ensures that in the next execution at the same execution point the scheduler will pick a thread that is enabled and that is not in $path\_c[j].postponed$. Thus in subsequent executions all the threads that are enabled at the execution point will get scheduled one by one. If the selected element contains a constraint and if it has not been negated previously, then constraint solving is invoked to generate a new input (see [25]).

5.2 Concolic Execution

Concolic execution [14,25] will be important for efficiently testing concurrent programs: the availability of concrete values for all memory locations in addition to the symbolic values helps us to accurately determine the race relation at runtime. The computation of the race relation is done by maintaining multithreaded vector clocks as described in Section 4. Note that to compute the race relation in the symbolic execution, we have to determine whether two symbolic memory locations are aliased. This can be infeasible in many cases as exact alias and pointer analysis is generally not possible if the programming language supports pointer arithmetic or supports arrays. However, the symbolic execution part of the concolic execution helps us perform symbolic execution as much as possible. This symbolic execution combined with constraint solving is essential to generate data inputs for the next execution.

The details of the procedure `execute_concolic` can be found in [25]. A brief pseudo-code of the procedure is given in Figure 8. Concolic execution maintains two states at the runtime: a concrete state $\mathcal{S}$ mapping memory locations to concrete values and a symbolic state mapping memory locations to symbolic expressions over symbolic input values. During concolic execution, every statement is executed concretely using the function *evaluate_concrete* and symbolically using the function *execute_symbolic* (see [25] for details). In addition to performing symbolic execution, the function *execute_symbolic* simplifies any complex (e.g. nonlinear) symbolic expressions in the symbolic state by replacing some symbolic values in the expression by their corresponding concrete values.

Each run (except the first) is executed with the help of a record of the conditional statements (which is the list *branch_hist*) executed in the previous run. For each conditional, the algorithm records a *branch* value, which is either `true` (the *then* branch is taken) or `false` (the *else* branch is taken), as well as a *done* value, which is `false` when only one branch of the conditional has executed in prior runs (with the same history up to the branch point) and is `true` otherwise. The information associated with each conditional statement of the last execution path is stored in the array *branch_hist*, kept in a file between executions.

6. EFFICIENT ALGORITHM

We now provide an efficient algorithm which explores a much smaller superset of the execution paths in $\texttt{REx}(P)$. The algorithm accomplishes this by computing race conditions between different events in an execution. Based on these race conditions, we generate other schedules that *flip* the race conditions, to provide a depth first search of all permutations of the race conditions in the execution path. More specifically, let $e_1 e_2 \ldots e_n$ be an execution path of a

```
$execute\_concolic(t, s, i, path\_c, branch\_hist)$
  match($s$)
    case ($lv \leftarrow e$):
      $m = evaluate\_concrete(\&lv)$; $val = evaluate\_concrete(e)$;
      $execute\_symbolic(m, e)$;
      $\mathcal{S} = \mathcal{S}[m \mapsto val]$; $pc_t = pc_t + 1$;
    .
    .
    .
    // skipping other cases
    .
    .
    .
    case (if $p$ goto $l'$):
      $b = evaluate\_concrete(p)$;
      $c = evaluate\_symbolic\_predicate(p, b)$;
      if ($b$)
        $path\_c[i].hasConstraint$ = true;
        $path\_c[i].constraint = c$; $pc_t = l'$;
        $cmp\_n\_set\_branch\_hist$(true,$i$,$branch\_hist$);
      else
        $path\_c[i].constraint$=neg($c$); $pc_t = pc_t + 1$;
        $cmp\_n\_set\_branch\_hist$(false,$i$,$branch\_hist$);
      $i = i + 1$;
  if ($s$ == ERROR) print "Found Error";
  return ($i$,$path\_c$,$branch\_hist$);
```

Figure 8: Concolic Execution.

```
scheduler($\mathcal{I}$,$path\_c$,$branch\_hist$)
     $T_{\text{enabled}} = \{t_0\}$; $t_{\text{current}}$ = null; $i = 0$;
     $pc_{t_0}$ = the label of the first statement
     initialize input variables using $\mathcal{I}$;
     while ($T_{\text{enabled}} \neq \emptyset$)
       $t_{\text{current}}$ = choose_next_thread($T_{\text{enabled}}, i$);
       $path\_c[i].event$ = nextEvent($t_{\text{current}}$);
       $path\_c[i].enabled = T_{\text{enabled}}$;
M:     $path\_c$ = check_and_set_race($i$,$path\_c$);
       $i = i + 1$;
       $s$ =statement_at($pc_{t_{\text{current}}}$);
       ($i$,$path\_c$,$branch\_hist$)
             = execute_concolic($t_{\text{current}}, s, i$,$path\_c$,$branch\_hist$);
       $T_{\text{enabled}}$ = set of enabled threads;
     if there is an active thread
       print "Found deadlock";
     return compute_next_input_and_schedule($i$,$\mathcal{I}$,$path\_c$,$branch\_hist$);

choose_next_thread($T_{\text{enabled}}, i$)
     if $i < |path\_c|$
       // schedule the thread as in the previous execution
       $(t, l, m, a) = path\_c[i].event$;
       return $t$;
     else
M:     if $t_{\text{current}}$ is enabled
M:       return $t_{\text{current}}$;
M:     else
         return smallest indexed thread from $T_{\text{enabled}}$;

check_and_set_race($i$,$path\_c$)
     $\forall j \in [0, i)$ such that $path\_c[j].event \lessdot path\_c[i].event$
       if $t$ not in $path\_c[j].postponed$
         if $e$ is a read or write event
           print "Warning: data race found"
           $path\_c[j].hasRace$ = true;
         else
           // $path\_c[j].event$ is an unlock event
           let $j'$ be such that $path\_c[j'].event$ is the lock event
           matching the unlock event $path\_c[j].event$
             $path\_c[j'].hasRace$ = true;
     return $path\_c$;
```

Figure 9: Efficient Testing Scheduler

```
compute_next_input_and_schedule($i$,$\mathcal{I}$,$path\_c$,$branch\_hist$)
    for ($j = i - 1$ ; $j \geq 0$ ; $j = j - 1$)
      if $path\_c[j].event$ has proper value
        // compute a new schedule
        if $|path\_c[j].enabled| > |path\_c[j].postponed| + 1$
M:        if $path\_c.isRace$
M:          $path\_c[j].isRace$ = false;
            $(t, l, m, a) = path\_c[j].event$;
            $path\_c[j].postponed = path\_c[j].postponed \cup \{t\}$;
            $t$ = smallest indexed thread in
                $path\_c[j].enabled \setminus path\_c[j].postponed$;
            $path\_c[j].event = (t, l, m, a)$ ;
            return ($\mathcal{I}$,$path\_c[0 \ldots j]$,$branch\_hist[0 \ldots j]$,false);
      else
        // compute a new input
        if (not $branch\_hist[j].done$)
          $branch\_hist[j].branch = \neg branch\_hist[j].branch$;
          if ($\exists \mathcal{I}'$ that satisfies neg_last($path\_c[0 \ldots j]$))
            return ($\mathcal{I}'$,$path\_c[0 \ldots j]$,$branch\_hist[0 \ldots j]$,false);
      return ([ ],[ ],[ ],true);
```

Figure 10: Compute Next Schedule or Input for Efficient Testing

program and let e_i and e_j, where $i < j$, are related by the immediate race relation (i.e. $e_i \lessdot e_j$). In our efficient testing algorithm, we mark the event e_i (by setting true the field *isRace* of the element in *path_c* corresponding to the event e_i) to indicate that it has race with some future event and the thread of e_i must be postponed at that execution point in some future execution so that the race relation between e_i and e_j gets flipped. While computing the next input and schedule at the end of the execution, if we choose to backtrack at the event e_i, then we generate a schedule for the next execution where we continue the execution up to the prefix $e_1 \ldots e_{i-1}$; however, after that we postpone the execution of the thread of e_i as much as possible. This ensures that the race between e_i and e_j gets flipped or permuted (i.e. $e_j \lessdot e_i$) in the next execution and we get an execution path of the form $e_1 \ldots e_{i-1}e_{i+1} \ldots e_j e'_{j+1} \ldots e_i \ldots e'_{n'}$. For example, if $t_1\colon x = 1, t_2\colon x = 2$ is an execution path, then there is a race condition in the accesses of the shared variable x. We generate a schedule such that the next execution is $t_2\colon x = 2, t_1\colon x = 1$, i.e., the accesses to x are permuted.

In the efficient algorithm, we modify the simple scheduler described in Figure 6 by the one in Figure 9. (We label a statement with M: if the statement is modified or added to the pseudo-codes given in Section 5.) Specifically, we assume that the scheduler maintains dynamic vector clocks for each thread and two dynamic vector clocks for each shared memory location. The dynamic vector clocks are updated using the procedure described in Section 4. We omit the dynamic vector clock update procedure in the pseudo-code of the efficient algorithm to keep the description simple. The scheduler calls the function `check_and_set_race` before executing a statement. The function `check_and_set_race` determines if the current event has a race with any past event e. If such a race exists, then the *hasRace* field of the element of *path_c* corresponding to the event e is set to **true** – assuming that the race condition wasn't already flipped in a previous execution. The procedure `choose_next_thread` is modified so that a postponed thread's execution gets delayed as much as possible. The computation of the next input and schedule is done using the modified procedure `compute_next_input_and_schedule` (see Figure 10). In this procedure, a new schedule, which postpones the thread associated with an event, is generated if the event has a race with a future event. Note that in the simple scheduler, a thread is postponed at an execution point even if the corresponding event has no race with any future event.

Soundness of our algorithm is trivial. A bug reported by our algorithm is an actual bug because our algorithm provides a concrete input and schedule on which the program exhibits the bug. Moreover, our algorithm can be complete in some cases.

PROPOSITION 4. ***(Completeness)*** *During testing a program with our efficient algorithm, if the following conditions hold:*

- *The algorithm terminates.*

```
scheduler($\mathcal{I}$,$path\_c$,$branch\_hist$)
     $T_{\text{enabled}} = \{t_0\}$; $t_{\text{current}} = \textbf{null}$; $i = 0$;
     $pc_{t_0}$ = the label of the first statement
M:   $sleep = \{\}$; $postponed = \{\}$;
     initialize input variables using $\mathcal{I}$;
     while ($T_{\text{enabled}} \neq \emptyset$)
M:       $postponed = postponed \cup path\_c[i].postponed$;
M:       $sleep = \{\texttt{nextEvent}(t) \mid t \in postponed\}$;
M:       $t_{\text{current}}$ = smallest indexed thread from $T_{\text{enabled}} \setminus postponed$;
         $path\_c[i].event = \texttt{nextEvent}(t_{\text{current}})$;
         $path\_c[i].enabled = T_{\text{enabled}}$;
         $path\_c = \texttt{check\_and\_set\_race}(i, path\_c)$;
M:       $path\_c[i].postponed = postponed$;
M:       $\forall e \in sleep$ if $e \lessdot path\_c[i].event$
M:           let $(t,l,m,a) = e$ in $postponed = postponed \setminus t$;
         $i = i + 1$;
         $s = \texttt{statement\_at}(pc_{t_{\text{current}}})$;
         $(i, path\_c, branch\_hist)$
                $= \texttt{execute\_concolic}(t_{\text{current}}, s, i, path\_c, branch\_hist)$;
         $T_{\text{enabled}}$ = set of enabled threads;
     if there is an active thread
         print "Found deadlock";
     return compute_next_input_and_schedule($i$,$\mathcal{I}$,$path\_c$,$branch\_hist$);
```

Figure 11: Optimized Testing Scheduler

- *The algorithm makes no approximation during concolic execution and it is able to solve any constraint which is satisfiable.*

then our algorithm has executed all executions in `REx` *and we have hit all reachable statements of the program.*

The proof of this proposition, while fairly intuitive, is beyond the scope of this paper.

7. FURTHER OPTIMIZATION

The efficient algorithm improves the simple algorithm by providing a systematic way to flip race relations between various pairs of events. However, this may result in repeated flipping of the race relation between the same pair of events, if the pair of events are not next to each other. As an instance, for the example in the Section 6, if the next execution path is $e_1 \dots e_{i-1} e_{i+1} \dots e_j e'_{j+1} \dots e_i \dots e'_{n'}$, then our efficient algorithm may detect that there is a race between e_j and e_i. As a result our algorithm would try to flip this race once again. To avoid this, we use a technique similar to sleep sets [11]. Specifically, in the execution path $e_1 \dots e_{i-1} e_{i+1} \dots e_j e'_{j+1} \dots e_i \dots e'_{n'}$, we add the thread t, where t is the thread of the event e_i, to the set *postponed* of every event $e_{i+1}, \dots, e_j$. As a result, even if we have detected that there is a race between e_j and e_i, we would not set to true the field *isRace* of the element of *path_c* corresponding to the event e_j (see the 2nd line of the procedure `check_and_set_race`). This ensures that we do not repeatedly flip race relation between the same pair of events. The pseudo-code of the optimized scheduler is given in Figure 11.

8. CASE STUDIES

We have implemented our optimized testing algorithm for Java. The tool is called jCUTE. jCUTE supports replay of execution paths to aid the process of debugging.

We use four sets of case studies to illustrate the effectiveness of jCUTE in finding potential bugs. The tool and the code for each case study along with directions to reproduce the bugs reported in this paper can be found at `http://osl.cs.uiuc.edu/~ksen/cute/`. The experiments were run on a 2.0 GHz Pentium M processor laptop with 1 GB RAM running Windows XP.

8.1 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. Much to our surprise, 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. A simplified skeleton of the test-driver that we used is given below:

```
public class MTListTest extends Thread {
 List s1,s2;
 public MTListTest(List s1, List s2) {
   this.s1 = s1;  this.s2 = s2; }

 public void run() {
   int c = Cute.input.Integer();
   Object o1 = (Object)Cute.input.Object("java.lang.Object");
   switch(c){
    case 0: s1.add(o1); break;
    case 1: s1.addAll(s2); break;
    case 2: s1.clear(); break;
    .
    .} }
 public static void main(String[] args) {
   List s1 = Collections.synchronizedList(new LinkedList());
   List s2 = Collections.synchronizedList(new LinkedList());
   (new MTListTest(s1,s2)).start();
   (new MTListTest(s2,s1)).start();
   (new MTListTest(s1,s2)).start();
   (new MTListTest(s2,s1)).start();}
}
```

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 briefly describe an infinite loop and a data race leading to an exception that jCUTE discovered in the synchronized `LinkedList` class and the synchronized `ArrayList` class, respectively.

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. The bug can be avoided if `containsAll` holds lock on both `l1` and `l2` throughout its execution. It can also be avoided if `containsAll` uses the synchronized method `toArray` as in the `Vector` class, rather than using iterators. Moreover, the statement `nextIndex != size;` should be changed to `nextIndex < size;` in the method `hasNext`. 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);
```

We next present a simple scenario under which jCUTE found a data race leading to an uncaught exception in the class `ArrayList`. The test driver first creates two synchronized array lists by calling

```
List l1 = Collections.synchronizedList(new ArrayList());
List l2 = Collections.synchronizedList(new ArrayList());
l1.add(new Object());
l2.add(new Object());
```

The test driver then concurrently allows a new thread to invoke `l1.add(new Object())` and another new thread to invoke `l2.containsAll(l1)`. During testing, jCUTE discovered data races over the fields `size` and `modCount` of the class `ArrayList`. In a subsequent execution, jCUTE permuted the events involved in a data race and discovered an uncaught `ConcurrentModificationException` exception. However, the program never throws the `ConcurrentModificationException` exception if the methods are invoked in any order by a single thread. Note that the Java API documentation claims that there should be no such data race or uncaught `ConcurrentModificationException` exception when we use synchronized form of array list. jCUTE also provided a trace of the buggy execution. This helped us to detect the cause of the bug. It is worth mentioning that jCUTE not only detects actual races, but also flips to see if the data race can be fatal, i.e., that it can lead to uncaught exceptions.

8.2 NASA's Java Pathfinder's Case Studies

In [19], 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 [19]. We also considered a distributed sorting implementation used in [15]. 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 [15, 19], 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 [19]. 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.

8.3 Needham-Schroeder Protocol

The Needham-Schroeder public-key authentication protocol [21] aims at providing mutual authentication through message exchanges between two parties: an *initiator* and a *responder*; details of the protocol can be found elsewhere [21]. Lowe reported an attack against the original protocol and also proposed a fix, called Lowe's fix [10].

We tested a concurrent implementation of the protocol

Name	Run time in seconds	# of Paths	# of Threads	% Branch Coverage	# of Functions Tested	# of Bugs Found data races/deadlocks/infinite loops/exceptions
Vector	5519	20000	5	76.38	16	1/9/0/2
ArrayList	6811	20000	5	75	16	3/9/0/3
LinkedList	4401	11523	5	82.05	15	3/3/1/1
LinkedHashSet	7303	20000	5	67.39	20	3/9/0/2
TreeSet	7333	20000	5	54.93	26	4/9/0/2
HashSet	7449	20000	5	69.56	20	19/9/0/2

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

Name	Run time in seconds	# of Paths	# of Threads	% Branch Coverage	Lines of Code	# of Bugs Found data races/deadlocks/assertions/exceptions
BoundedBuffer	11.41	43	9	100.0	127	0/1/0/0
NestedMonitor	0.46	2	3	100.0	214	0/1/0/0
Pipeline	0.70	3	5	64.29	103	1/0/1/0
RemoteAgent	0.45	2	3	87.5	55	1/1/0/0
RWVSN	2.19	8	5	68.18	590	1/0/1/0
ReplicatedWorkers	0.34	1	5	25.93	954	0/0/1/0
DEOS	35.23	111	6	64.75	1443	0/0/1/0

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

using jCUTE. jCUTE found the attack in 406 iterations or about 95 seconds of search.

We compare these results with the ones reported previously [13, 14] for the same protocol. The explicit-state C model-checker VeriSoft [13] analyzed a concurrent implementation of the protocol with finite input domain. Verisoft was unable to find the attack within 8 hours, evolutionary testing (with manual tuning) found the attack after 50 minutes (on a somewhat slower machine). DART [14] found the attack on a sequential implementation of the protocol with a somewhat stronger intruder model[1] in 18 minutes. In comparison, jCUTE found the attack on a concurrent implementation of the protocol with a proper intruder model in only 95 seconds, which is an order of magnitude faster than the fastest previous approach. This performance difference is due to jCUTE's efficient algorithm that only explores distinct causal structures.

8.4 TMN Protocol

The Tatebayashi, Matsuzaki, and Newman (TMN) Protocol [30] is a protocol for distribution of a fresh symmetric key. In this protocol when an initiator wants to communicate with a responder, it uses a trusted server to obtain a secret symmetric session key. The details of the protocol can be found in [30].

In this protocol, an intruder can establish a parallel session through eavesdropping and obtain the secret key [17]. We tested a concurrent implementation of the protocol using jCUTE. jCUTE found the attack in 522 iterations or about 127 seconds of search.

9. OTHER RELATED WORK

Improving quality of concurrent programs is a challenging area of research. A major cause for defects in multithreaded programs is race condition. A huge body of research focuses on dynamic or static race detection [3, 5, 8, 18, 22, 23]. Race detection suffers from the problem of false warnings. Moreover, the dynamic techniques can report all possible race conditions only if there are good test inputs that can achieve high code coverage. Our algorithm not only detects races but also permutes them systematically to search if the races can lead to some bug. Moreover, jCUTE generates test inputs so that number of races caught is maximized.

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 [9] 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* [11] to prune the search space. However, the implementation in [9] considered two read accesses to the same memory location by different threads to be dependent. Thus for the 3-threaded program $t_1 : lv1 = x; t_2 : lv2 = x; t_3 :$ **if** $x > 0$ **then** `ERROR`, the algorithm described in [9] 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 [26]. As such for the above example, we will explore only one interleaving. In addition to exact race detection, we propose a technique to flip these races by combining techniques drawing from *delayed sets* [1], *persistent/sleeps sets* [9, 11]. Note that none of the previous descriptions of the above dynamic partial order reduction techniques have handled programs which have inputs.

In a similar independent work [27], 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. Thus this work can also be seen as a way of combining symbolic execution with partial order reduction based model checking techniques for the purpose of testing parallel programs. 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 [4,6,29] based on static analysis have been developed, which can detect bugs in concurrent pro-

[1]Note that a stronger intruder model makes it easier for the intruder to find an attack. This in turn makes the search space smaller resulting in faster testing time.

grams. 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.

10. CONCLUSION

We presented an efficient algorithm for testing multithreaded programs. An important aspect of our algorithm is that we treat symbolic constraint solving and race-flipping uniformly in our algorithm. In a given execution, we either do constraint solving or race-flipping. This helps us to test concurrent programs in a single go. 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 combining symbolic and concrete execution. The concrete execution 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 race flipping is an attractive technique to test concurrent programs.

Acknowledgment

We would like to thank Timo Latvala, Darko Marinov, Grigore Roşu, Abhay Vardhan, Mahesh Viswanathan, and Tao Xie for useful discussions and comments. This work is supported in part by the ONR Grant N00014-02-1-0715 and the NSF Grant NSF CNS 05-09321.

11. 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. D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proc. of the ACM SIGPLAN Conference on Programming language design and implementation*, pages 258–269, 2002.

[4] 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.

[5] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging*, May 1991.

[6] M. B. Dwyer, J. Hatcliff, Robby, and V. P. Ranganath. Exploiting object escape and locking information in partial-order reductions for concurrent object-oriented programs. *Form. Methods Syst. Des.*, 25(2–3):199–240, 2004.

[7] C. J. Fidge. Partial orders for parallel debugging. In *Proceedings of the Workshop on Parallel and Distributed Debugging (WPDD)*, pages 183–194. ACM, 1988.

[8] C. Flanagan and S. N. Freund. Detecting race conditions in large programs. In *Proc. of the Program Analysis for Software Tools and Engineering Conference*, June 2001.

[9] 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.

[10] G. Lowe. An Attack on the Needham-Schroeder Public-Key Authentication Protocol. *Inf. Processing Letters*, 1995.

[11] 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.

[12] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *24th ACM Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[13] P. Godefroid and S. Khurshid. Exploring Very Large State Spaces Using Genetic Algorithms. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2002.

[14] 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.

[15] 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.

[16] J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.

[17] G. Lowe and A. W. Roscoe. Using csp to detect errors in the TMN protocol. *Software Engg.*, 23(10):659–669, 1997.

[18] R. Netzer and B. Miller. Detecting data races in parallel program executions. In *Advances in Languages and Compilers for Parallel Computing, 1990*. MIT Press.

[19] 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.

[20] D. Peled. All from one, one for all: on model checking using representatives. In *5th Conference on Computer Aided Verification*, pages 409–423, 1993.

[21] R. Needham and M. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM*, 21(12):993–999, 1978.

[22] B. Richards and J. R. Larus. Protocol-based data-race detection. In *Proc. of the SIGMETRICS symposium on Parallel and distributed tools*, pages 40–47, 1998.

[23] 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, 1997.

[24] 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.

[25] 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.

[26] K. Sen, G. Roşu, and G. Agha. Online efficient predictive safety analysis of multithreaded programs. *International Journal on Software Technology and Tools Transfer*, 2006.

[27] 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.

[28] G. L. Steele. Making asynchronous parallelism safe for the world. In *17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 218–231, 1990.

[29] 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.

[30] M. Tatebayashi, N. Matsuzaki, and J. David B. Newman. Key distribution protocol for digital mobile communication systems. In *Proceedings on Advances in cryptology (CRYPTO '89)*, pages 324–334. Springer-Verlag, 1989.

[31] A. Valmari. Stubborn sets for reduced state space generation. In *10th Conference on Applications and Theory of Petri Nets*, pages 491–515, 1991.

[32] W. Visser, C. Pasareanu, and S. Khurshid. Test Input Generation with Java PathFinder. In *Proceedings of ACM SIGSOFT ISSTA'04*, pages 97–107, 2004.

[33] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Procs. of TACAS*, 2005.