MuTMuT: Efficient Exploration for Mutation Testing of Multithreaded Code

Milos Gligoric, Vilas Jagannath, and Darko Marinov

Department of Computer Science, University of Illinois at Urbana-Champaign
Urbana IL, 61801, USA
Email: {gliga, vbangal2, marinov}@illinois.edu

***Abstract*—Mutation testing is a method for measuring the quality of test suites. Given a system under test and a test suite, mutations are systematically inserted into the system, and the test suite is executed to determine which mutants it detects. A major cost of mutation testing is the time required to execute the test suite on all the mutants. This cost is even greater when the system under test is multithreaded: not only are test cases from the test suite executed on many mutants, but also each test case is executed for multiple possible thread schedules.**

We introduce a general framework that can reduce the time for mutation testing of multithreaded code. We present four techniques within the general framework and implement two of them in a tool called MuTMuT. We evaluate MuTMuT on eight multithreaded programs. The results show that MuTMuT reduces the time for mutation testing, substantially over a straightforward mutant execution and up to 77% with the advanced technique over the basic technique.

I. Introduction

Computing is undergoing a historic change as processor clock speeds stagnate while the number of computing cores increase. To extract better performance from the multiple cores, software developers need to write parallel code. The currently dominant parallel programming model is that of shared memory, where multiple threads of computation read and write shared data objects. For example, Java provides support for threads in the language and libraries, with shared data residing on the heap. Multithreaded code is notoriously hard to get right, with common concurrency bugs being atomicity violations, dataraces, and deadlocks. While new parallel programming models are emerging, developers need help right now to develop and test multithreaded code.

Mutation testing [1]–[3] is a method for evaluating and improving the quality of a test suite. A mutation testing tool proceeds in two steps. First, in *mutant generation*, the tool generates a set of *mutants* by modifying the original code under test. The tool applies *mutation operators* that perform small syntactic modifications on the code under test. For example, a modification can replace a variable with a constant (of a compatible type), say `amount` with `0`. Second, in *mutant execution*, the tool measures how many mutants a given test suite detects. The tool executes each mutant and the original code on the test cases from the test suite and compares the corresponding results. If a mutant generates a result different from the original code, the test input *kills* the mutant. Note that some of the mutants, although syntactically different from the original code, may be semantically equivalent to the original code, so there is no input that can kill them [3], [4].

There is a large body of work on *mutant generation*, some even for multithreaded code. The literature contains mutation operators for several programming languages, including Java [4]–[9]. The operators for Java include the traditional operators that modify statements and expressions, as well as object-oriented operators that modify classes, e.g., field or method declarations. Additional operators for multithreaded code [10]–[12] can also modify thread or synchronization operations, e.g., removing locks or barriers. However, to the best of our knowledge, there has been no work on efficient *mutant execution* for multithreaded code.

To understand the cost of *mutant execution* for multithreaded code, let us first consider *test execution* for multithreaded code. A multithreaded test is a piece of code that executes two or more threads [13], [14]. Since multithreaded code can have different behavior for different *thread schedules/interleavings*, a multithreaded test can produce different results for the same input. Thus, to check whether a multithreaded test can fail for some schedule, the test is typically run several times, using stress or random testing [15], [16] that repeatedly runs the test without much control over the scheduler, or using systematic *exploration* of possible schedules [17]–[21]. The most thorough exploration tries *all possible* schedules, but their number can be very high. Musuvathi and Qadeer proposed CHESS [19], a highly effective exploration heuristic that limits the number of schedules by bounding the number of preemptive context switches in thread schedules, and yet finds many bugs. Regardless of what exploration is used, executing one multithreaded test on even one version of code can be expensive. It is even more expensive for mutant execution because the test is run on several (mutated) versions of the code.

This paper makes several contributions.

Framework: We propose a framework for efficient mutant execution of multithreaded code. We call our framework and the tool that implements it MuTMuT (from "MUtation Testing of MUltiThreaded code"). The key idea of MuTMuT is to *reuse some information* collected during the exploration/execution of a test on the *original code under test* to speed up the exploration/execution of the same test on the *mutant code*. The simplest reuse is based on code coverage and is already implemented in some tools for mutation testing of sequential code [4], [8], [9]. The idea is to execute a test on the original code under test and to determine which mutations the test *reaches*; then only the mutants whose mutations were reached need to be executed, since the other mutants will be equivalent to the original code for this test. The same idea can be translated to multithreaded tests: if the original code does not reach a mutation during the entire *exploration*, then the mutant need not be explored. However, knowing only that a mutation was reached would require repeating the entire exploration for the mutant. MuTMuT can record more information—conceptually, which thread schedules reached a mutation—to prune away some more thread schedules from the exploration of a mutant.

Techniques: We discuss four specific techniques that instantiate the general MuTMuT framework using various information that can be recorded and reused from one exploration to another. The main trade-off is that collecting more information is costlier but can result in pruning away more schedules during exploration.

Implementation: We present an implementation of the MuTMuT framework and two techniques (Basic and StateSet) in Java PathFinder (JPF), an open-source model checking tool developed at NASA for verifying Java programs [18], [22]. JPF implements a backtrackable Java Virtual Machine (JVM) that provides control over non-deterministic choices including thread schedules. The backtrackable JVM allows JPF to explore multithreaded tests for a wide range of Java programs. Our implementation supports both exhaustive exploration of all thread schedules and CHESS-like bounding of context switches. By default, JPF provides guarantees about all schedules, although it does not explicitly explore them all but instead uses sophisticated partial-order reduction and state matching techniques [18], [23] to prune schedules/states that have equivalent behavior.

Evaluation: We describe an evaluation of MuTMuT on eight small multithreaded programs and their tests. The results show that MuTMuT's StateSet technique can speed up mutant execution up to 77% over the Basic technique.

II. Example

We illustrate how MuTMuT speeds up mutant execution on a simple example based on a bank simulation program from the benchmark for testing multithreaded programs [24]. Figure 1 shows the code under test and example tests. The `Account` class stores the `balance` for an account and can create an account with some initial amount, `withdraw` money from the account (as long as the resulting balance is not negative), `deposit` more money into the account, lookup the current `balance`, and `transfer` money from one account to another. Note the Java keyword `synchronized` to indicate that the methods should be executed *atomically*. If they were not, using `Account` objects in multithreaded code could have dataraces and lead to incorrect balances.

```
class Account {
    double balance;
    Account(double amount) { this.balance = amount; }
    synchronized void withdraw(double amount) {
        if (balance >= amount) {
            balance −= amount; } }
    synchronized void deposit(double amount) {
        balance += amount; }
    synchronized double balance() {
        return balance; }
    void transfer(Account other, double amount) {
        // code implementing the transfer operation...
    } }

class AccountTest {
    // example test to illustrate MuTMuT techniques
    void test1() {
        final Account a = new Account(50.00);
        Thread t1 = new Thread() { public void run() {
            a.withdraw(80.00); }};
        Thread t2 = new Thread() { public void run() {
            a.deposit(90.00); }};
        Thread t3 = new Thread() { public void run() {
            a.deposit(20.00); }};
        t1.start(); t2.start(); t3.start();
        t1.join(); t2.join(); t3.join();
        assert 80.00 == a.balance() || 160.00 == a.balance(); }
    // example test to illustrate CHESS
    void test2() {
        final Account a = new Account(0.00);
        class DepositWithdraw extends Thread { public void run() {
            a.deposit(60.00);
            a.withdraw(10.00); }};
        Thread t1 = new DepositWithdraw();
        Thread t2 = new DepositWithdraw();
        t1.start(); t2.start();
        t1.join(); t2.join();
        assert 100.00 == a.balance(); }
    // example test that does not reach any example mutant
    void test3() {
        final Account a = new Account(10.00);
        Thread t = new Thread() { public void run() {
            assert 10.00 == a.balance(); }};
        t.start(); t.join(); } }
```

Figure 1. Example code under test and multithreaded tests

A. Multithreaded tests

Figure 1 shows three example multithreaded tests for `Account`. The `test1` method validates whether concurrently executing one `withdraw` and two `deposit` operations results in a correct `balance` amount. The test has three threads that perform the operations and the main thread that makes the assertion. Note that, since `withdraw` cannot result in a negative `balance`, the final amount can differ based on the schedule of the threads. This test uses the standard Java operations to initiate thread execution (`start`) and wait for thread completion (`join`). Similarly, `test2` validates `deposit` and `withdraw` operations, and `test3` validates the constructor and `balance`.

B. Schedules and search space

Figure 2 shows the thread schedules and the state space for complete exploration of `test1`. Each node in the graph represents a *state* in this space, and we label states S_1 to S_{10}. The value inside the node is the `balance` in the account `a`

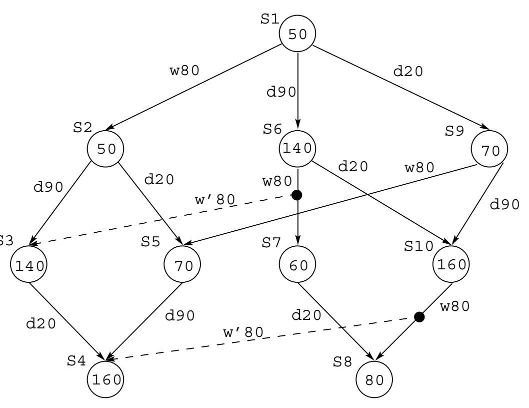


Figure 2. State space for exploring `test1`

from `test1` code. The edges represent *transitions* that the code makes from one state to another. The edge labels show the method, e.g., `w80` means `withdraw(80.00)`. Note that the methods are atomic, so there are no thread interleavings within the methods. Note additionally that dashed edges are not a part of the exploration of `test1` on the original `Account` code but on a mutant as we will discuss later.

The search space in Figure 2 is for a *stateful, depth-first* search [23]; each new state encountered in the search is compared with the previously encountered states, and if the new state matches a previously encountered state, the exploration stops. For example, executing `withdraw(80.00)` in state S_9 results in state S_5. Note that the state comparison matches entire program states, including program counters for the threads, and not only the account balance. For instance, states S_1 and S_2 differ, although they have the same account balance. While the example search we show is stateful and depth-first, we point out that our tool, MuTMuT, which is built on top of JPF, also supports stateless search and non-depth-first search strategies available in JPF (breadth-first, random, CHESS, etc.). Various search approaches can take different time and can explore different parts of the search space. As the first approximation to comparing two exploration times, we can use *the number of states being explored and the number of transitions being executed* [25]. For example, the state space in Figure 2 has 10 states and 13 transitions.

C. Mutants

Figure 3 shows three example mutants for `Account`. The first mutant replaces `+=` with `-=`, the second mutant replaces `amount` with `0`, and the third mutant modifies the `transfer` method. Note that these mutants are shown as a *mutant schemata* [26], i.e., one program that encodes all the mutants. By setting the appropriate value for the mutant id, the same program can execute either as the original code or as one of the mutants. This type of mutant generation was introduced to speed up *compilation* of mutants [26]:

```
class Account {
    double balance;
    Account(double amount) { this.balance = amount; }
    synchronized void withdraw(double amount) {
        if (balance >= amount) {
            if (IS_MUTANT(1)) balance −= 0; else balance −= amount; } }
    synchronized void deposit(double amount) {
        if (IS_MUTANT(2)) balance −= amount; else balance += amount; }
    synchronized double balance() {
        return balance; }
    void transfer(Account other, double amount) {
        if (IS_MUTANT(3)) ... // some mutation 3 in this method body
    } }
```

Figure 3. Example mutants for the code under test

rather than generating (and compiling) one new program for *each mutation*, it generates (and compiles) only one *meta-mutant* that encodes *all mutations*. We show how MuTMuT leverages these mutants to speed up exploration for multithreaded tests.

Consider the exploration of the tests from Figure 1 on the mutant schemata from Figure 3. The goal of mutant execution is to establish which mutants get killed by the tests. A very naive mutant execution would explore all the tests on all the mutants to find out which mutants get killed. Clearly, a better solution is to stop executing a mutant as soon one test kills it. An even better solution is to consider which mutations were *reached* during the exploration of the original code [4], [9] (and as we discuss later, *where* in the exploration the mutations were reached).

Figure 2 shows not only the exploration of `test1` on the original code but also the exploration of `test1` on `Mutant1`. The two dashed edges correspond to transitions that reach the mutation in `Mutant1`, namely where the `withdraw` method is called with `balance` greater than `amount`. Note that in this example the dashed edges take the execution to a state that is encountered in the original exploration, but in general exploration of mutants could lead to new states.

For our three example tests and mutants, explorations of both `test1` and `test2` reach both `Mutant1` and `Mutant2`, exploration of `test3` does not reach any mutant, and no test reaches `Mutant3`. Since `test3` reaches no mutant, it cannot kill any mutant, so it need not be explored for any mutant. Similarly, `Mutant3` cannot be killed by any of the three tests, so it need not be explored for any test. The MuTMuT *Basic* technique uses such properties to speed up mutant execution. To infer these properties, it suffices to record *whether* a test reaches a mutation, effectively keeping a set of mutations reached during the original exploration.

D. Efficient exploration

The key insight behind MuTMuT is that it is possible to record even more information from the original exploration to speed up the test exploration for the mutants. More specifically, a tool can record not only *whether* but also *where* a test reaches a mutation. Consider Figure 2 and exploration of `test1` for `Mutant1`. Rather than starting this exploration from the very initial state (S_1), it suffices to re-execute the code until S_6 and start the exploration from that state, since that is the first state in the *entire exploration* that reaches this mutation. In other words, it is not necessary to explore whether `test1` can kill `Mutant1` if the first executed operation is `withdraw`, since no such execution reaches the mutation. In this particular example, starting the exploration from S_6 avoids exploring S_2 and the three transitions that lead to and start from S_2.

Moreover, as discussed in detail in the next section, one could record even more information from the exploration of the original code—not only the first state in the *entire exploration* that reaches a mutation, but also (1) the first state on *each execution path/schedule* that reaches a mutation, or (2) what mutations can be reached for explorations starting from each state—to prune away even more exploration. In the particular example of `test1` and `Mutant1`, these two additional prunings would avoid exploring 5 states and 8 transitions, and 5 states and 9 transitions, respectively. Figure 5 shows a visualization of the state spaces for these cases. Section III-C discusses this further.

However, it is not always beneficial to record more information. The additional time required to record more information may not be offset by the savings in the exploration. For this reason, MuTMuT, in its *StateSet* technique, records only the first state on *each execution path* that reaches a mutation. As the experimental results in Section V show, this information helps MuTMuT to substantially reduce the exploration time with the StateSet technique over the Basic technique (which is already a significant improvement [4], [9] over a naive exploration of all tests on all mutants). For an `Account` class similar to our running example, but with more tests and mutants, Section V shows that MuTMuT StateSet runs 43% faster than MuTMuT Basic.

E. CHESS

MuTMuT need not be used only for exhaustive exploration of all schedules. For example, CHESS [19] is an effective heuristic that explores only schedules up to a bounded number of preemptive context switches. Consider the exploration for `test2` that has two threads with two operations each, and let us label these operations as $\mathtt{d}^{t1}$ (`deposit` from thread 1), $\mathtt{w}^{t1}$, $\mathtt{d}^{t2}$, and $\mathtt{w}^{t2}$. These operations can have six possible schedules, e.g., $\langle \mathtt{d}^{t1}, \mathtt{d}^{t2}, \mathtt{w}^{t2}, \mathtt{w}^{t1} \rangle$ is one of them. However, two of the schedules—$\langle \mathtt{d}^{t1}, \mathtt{d}^{t2}, \mathtt{w}^{t1}, \mathtt{w}^{t2} \rangle$ and $\langle \mathtt{d}^{t2}, \mathtt{d}^{t1}, \mathtt{w}^{t2}, \mathtt{w}^{t1} \rangle$—require switching twice from one thread to another. Running CHESS with bound 1 would prune these two schedules from the exploration. For the `Account` class from Section V and for the *original* program, CHESS with bound 2 explores tests 30X faster than the exhaustive exploration. Moreover, MuTMuT can speed up mutant execution even when CHESS is used; for `Account` and CHESS with bound 2, MuTMuT StateSet runs 28% faster than MuTMuT Basic.

```
int currentlyExecutingMutantId = 0; // execute original code
PassOrFail explore(Test t) {
    State $s_{init}$ = initial state for test t;
    Set<Pair<State, Transition>> ToExplore = transitions($s_{init}$);
    Set<State> Encountered = {$s_{init}$};
    Map<State, List<Transition>> Paths = {$s_{init} \mapsto$ emptyList};
    while (ToExplore $\neq$ {}) {
        <s, t> = pickOnePair(ToExplore);
        ToExplore = ToExplore $-$ {<s, t>};
        if (<s, t> satisfies some pruning condition such as the number
            of context switches made for CHESS) continue;
        restore state s;
        State s' = execute transition t on state s;
        if (s' $\notin$ Encountered) {
            List<Transition> p = append(Paths(s), t);
            if (assertion not satisfied in s') return test t failed for path p;
            ToExplore = ToExplore $\cup$ transitions(s');
            Encountered = Encountered $\cup$ {s'};
            Paths = Paths $\cup$ {s' $\mapsto$ p}; } }
    return test t passed; }
Set<Pair<State, Transition>> transitions(State s) {
    // return a set of all enabled transitions for the given state
}
```

Figure 4. Pseudo-code for traditional state-space exploration

III. Framework

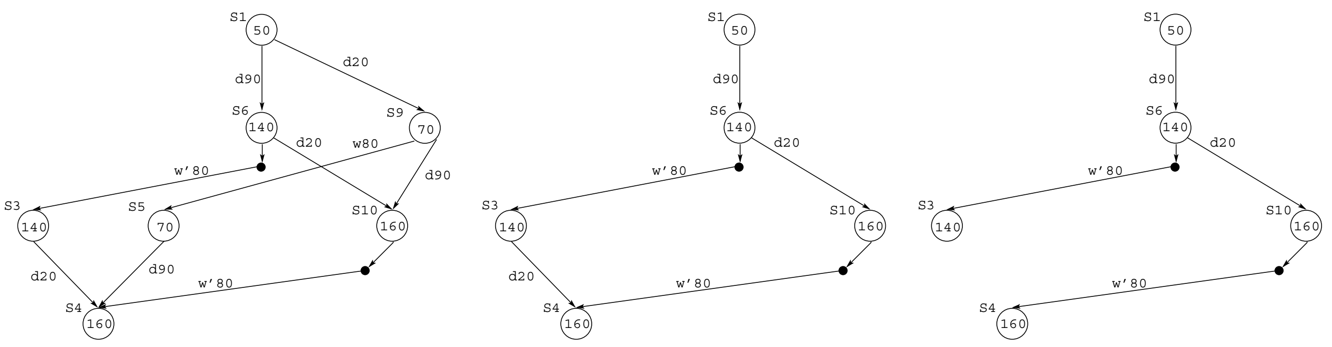
We next describe our general framework for mutant execution of multithreaded code. The approach is to first record various pieces of information during the exploration of a test on the original code under test, and then use that information to speed up the exploration of the test on the mutants. We instantiate the framework into four specific techniques based on various information that is recorded and reused. The key trade-off is that collecting more information is costlier but can result in pruning away more of the exploration.

A. Traditional exploration

We first describe how a traditional exploration searches through a state space [23] and then discuss how MuTMuT techniques modify the traditional exploration. Figure 4 shows pseudo-code for the traditional exploration. It takes as input a test `t` and reports whether `t` fails for some state/path or passes for all (explored) states. It maintains two sets: `ToExplore` is a set of states and transitions that still need to be explored, and `Encountered` is a set of states (typically their hash values [23]) that were already visited. It also maintains for each state a path (i.e., a sequence of transitions) that leads to that state. While there is more to explore, it takes a state s and a transition t, executes t on s, checks if the resulting state s' was visited, and, if not, adds s' and its transitions to `ToExplore`. If no violation is found, the test is reported as passing.

B. Common MuTMuT infrastructure

Figure 6 shows pseudo-code for the mutant execution performed by MuTMuT. Given a test suite and a set of (live) mutants, MuTMuT computes the set of mutants killed by the test suite. For each test, MuTMuT first performs a slightly modified (as explained in the next paragraph) traditional exploration of the original code. During this exploration `currentlyExecutingMutantId` is set to `0` and MuTMuT



(a) (b) (c)

Figure 5. State spaces for exploring `test1` on `Mutant1` using various MuTMuT techniques: (a) OneState, (b) StateSet, (c) AllStates

```
Set<int> findKilledMutants(TestSuite ts, Set<int> mutants) {
    Set<int> killed = {};
    foreach (Test t: ts) {
        currentlyExecutingMutantId = 0; // start exploring original
        Set<int> ReachedSet = {};
        AdditionalInformation ai = empty;
        explore(t); // during exploration, collect ReachedSet and ai
        foreach (int id: ReachedSet) {
            currentlyExecutingMutantId = id; // explore the mutant
            result = specialExplore(t, ai);
            if (result is test failed) killed = killed ∪ {id}; } }
    return killed; }
```

Figure 6. High-level overview of MuTMuT's mutant execution

```
boolean IS_MUTANT(int mutantId) {
    if (currentlyExecutingMutantId == 0) { // original code is executing
        if (mutantId ∉ killed) {
            ReachedSet = ReachedSet ∪ {mutantId};
            notifyMutantReachedForCurrentState(mutantId, s); }
        return false;
    } else // some mutant is executing
        if (currentlyExecutingMutantId == mutantId)
            return true; // this mutant is executing
        else return false; } // another mutant is executing
```

Figure 7. Pseudo-code that MuTMuT executes for mutation points

collects a set of reached (live) mutants and the additional information (depending on the specific MuTMuT technique being used, as described in Section III-C). For each of those mutants, it then performs a *special* exploration (again depending on the specific technique) of the test on each of those mutants. If the test fails, the mutant is killed.

The change that MuTMuT makes in the traditional exploration is during the execution of transitions. MuTMuT first starts the traditional exploration for the original program and during this exploration, some of the executed transitions will reach the mutation points in the meta-mutant. Figure 7 shows the code that MuTMuT executes for the mutation points to determine whether to execute the original code or the mutation. Additionally, the code notifies an execution observer that the mutation has been reached while executing a transition on the current state `s`. Various MuTMuT techniques will then record various information at these points.

technique	additional information collected
Basic	boolean: did the test exploration reach the mutant?
OneState	at most one state: the first state in the entire exploration where the mutant was reached, if it was reached
StateSet	a set of states: for each path that reached the mutant, the first state that reached the mutant
AllStates	a set of states: all states that reached the mutant

Figure 8. Information recorded for each (live) mutant and test exploration

C. Specific MuTMuT techniques

We present four MuTMuT techniques, called Basic, OneState, StateSet, and AllStates. We first describe the additional information that the techniques collect (through `notifyMutantReachedForCurrentState` calls) and then describe how the techniques use this information during their special exploration (in `specialExplore`).

Additional information: Figure 8 summarizes the additional information recorded during exploration of the original code. Figure 9 shows the pseudo-code for this collection. For Basic, all the information is already in ReachedSet. OneState records the first state (more precisely, the path needed to restore that state) that reaches the mutant *during the entire exploration*. For `test1` and `Mutant1`, this state is during the execution of `w80` from the state S_6.

StateSet records a set of states (rather, the paths to those states) that reach the mutant. It records only the first state that reaches the mutant *on a path*. It does not record multiple states that reach the same mutation on one path, which can happen when the mutation is in a loop or a recursive method. The rationale for recording only the first state on a path is as follows. When MuTMuT executes the mutant and reaches the mutation for the first time, it will execute the mutation rather than executing the original code. The mutation can potentially change the state such that the second mutation on the same path may not be reached. For `test1` and `Mutant1`, StateSet records S_6 and S_{10}.

AllStates records the largest amount of information. It collects for each state a set of mutants that can be reached *directly* in one transition leading from that state. Moreover, it collects the entire connectivity of the state space graph, i.e.,

```
// for Basic technique, all information is in ReachedSet

// for OneState technique
type of AdditionalInformation ai is Map<int, List<Transition>>;
notifyMutantReachedForCurrentState(int mutantId, State s) {
    // first time mutantId is reached during the exploration
    if (!ai.containsKey(mutantId))
        ai = ai ∪ {mutantId ↦ Paths(s)} } // record path for this state

// for StateSet technique
type of AdditionalInformation ai is Map<int, Set<List<Transition>>>;
notifyMutantReachedForCurrentState(int mutantId, State s) {
    Set<List<Transition>> previousPaths = ai(mutantId);
    List<Transition> currentPath = Paths(s);
    // first time mutantId is reached on this particular path
    if (previousPaths does not contain any prefix of currentPath) {
        Set<List<Transition>> newPaths = previousPaths ∪ {currentPath};
        ai = ai ⊎ {mutantId ↦ newPaths}; } }

// for AllStates technique
type of AdditionalInformation ai is Map<int, Set<State>>;
// this technique also needs to record the entire state space graph,
// i.e., what states are successors of what other states
notifyMutantReachedForCurrentState(int mutantId, State s) {
    Set<State> previousStates = ai(mutantId);
    // set of all states from which the mutant is reached directly in one transition
    Set<States> newStates = previousStates ∪ {s};
    ai = ai ⊎ {mutantId ↦ newStates}; }
```

Figure 9. Pseudo-code for collecting additional information

what states are successors of what other states. AllStates then post-processes these two pieces of information to compute for each state a set of mutants that can be reached *transitively* in one or more transitions leading from that state. Effectively, it computes for each state what mutants can be reached in an exploration starting from that state.

Special exploration: Various MuTMuT techniques perform different `specialExplore`. Basic performs the same exploration as the traditional exploration with the only difference being that Basic does not explore the original code but a mutant, specifically one of the mutants from the ReachedSet as shown in the loop within lines 8-11 in Figure 6.

OneState has additional information about the first state that reached a mutant. Hence, for each mutant, OneState's exploration first restores that state and its associated search environment, and then performs a search from that point. Thus, OneState explores the states reachable from the restored state and then *continues the search* to complete the rest of the exploration. Figure 5(a) shows the states and transitions that OneState explores for our example `test1` and `Mutant1`. Comparing with Figure 2, OneState does not explore state S_2 and the three transitions that lead to and from it. Note that since the pruned part of the state space does not reach the mutant, OneState is still guaranteed to correctly find whether the mutant is killed. Effectively, OneState replaces the initialization part in lines 3-6 in Figure 4: rather than starting the search from the initial state with no path, OneState starts the search from a state with a path previously recorded from the original exploration.

Note that OneState requires the original exploration and the mutant exploration to use the same search strategy. Namely, the call `pickOnePair` in line 8 in Figure 4 allows for any search strategy, e.g., picking the pair added last/first to `ToExplore` results in depth/breadth-first search. OneState also allows any strategy to be used, as long as the same strategy is used for both explorations; if different strategies were used, OneState could miss exploring a part of the state space and could thus incorrectly miss killing a mutant.

StateSet has additional information about the first state that reached a mutation for all paths that reached a mutant. Hence, for each mutant, StateSet's exploration restores these states one by one and explores the state space reachable from the restored state, much like OneState with one key difference: StateSet *does not continue the search* to complete the rest of the exploration. Figure 5(b) shows the states and transitions that StateSet explores for `test1` and `Mutant1`. Comparing with Figure 5(a), StateSet does not continue the exploration from S_1 after finishing the exploration from S_6, and so StateSet avoids exploring the state S_9 and the transitions that lead to and from that state. Like OneState, StateSet allows using various search strategies, but the same strategy has to be used for both the original and mutant explorations. Also like OneState, StateSet is guaranteed to correctly find whether the mutant is killed since the pruned part of the state space could not kill the mutant.

AllStates has the additional information about all the states that could reach the mutant not only directly in one transition but also transitively through a number of transitions. Like StateSet, AllStates finds all first states that reached a mutant on paths that reached a mutant, restores these states one by one and then explores the state space reachable from the restored state, without continuing the search to complete the rest of the exploration. Moreover, AllStates modifies the condition in line 14 in Figure 4. The traditional condition checks only if the next state s' was visited. AllStates also checks if it has information about s' (it could happen that s' is a new state encountered for the mutant such that AllStates could not have collected information about that state during the original exploration), and if it does have the information, checks whether the currently executed mutant can be reached from s'. Specifically, line 14 becomes the following:

$$\textbf{if}(s' \notin \text{Encountered} \wedge$$
$$(!\text{ai.containsKey}(s') \vee \text{currentlyExecutingMutantId} \notin \text{ai}(s')))$$

Figure 5(c) shows the exploration for AllStates. Comparing with Figure 5(b), AllStates prunes the transition `d20` from S_3, since AllStates knows that exploration from S_3 cannot reach `Mutant1`. Like OneState and StateSet, AllStates requires the same search strategy for original and mutant explorations to correctly detect the mutants that are killed.

D. Properties of MuTMuT techniques

All four MuTMuT techniques satisfy three desirable properties for (correct and) efficient mutant execution. To state the properties, we recall that MuTMuT's inputs are a test suite and a set of mutants, MuTMuT's output is a set of mutants killed by the test suite, and MuTMuT explores a state space, visiting a number of states and executing a number of transitions.

Soundness: MuTMuT does not kill any mutant that should not be killed (i.e., which passes for all execution paths of all tests in the given test suite). Each mutant that MuTMuT puts in the set of `killed` mutants indeed fails for some execution path for some test from the test suite.

Completeness: MuTMuT kills all mutants that should be killed *for the given exploration strategy* and for the given test suite. Each mutant that MuTMuT does not put in the set of `killed` mutants indeed cannot fail for the execution paths that would be explored with a naive mutant execution that tries all tests on all mutants. Note that we have to qualify this statement based on the exploration strategy, since the strategy itself can prune some exploration paths, e.g., CHESS prunes the paths where the number of context switches is over a given bound. It may be the case that the exhaustive exploration of all possible paths would kill the mutant for some test. The key is that MuTMuT gives the same result as a naive mutant execution but much faster.

Improvements: MuTMuT techniques are not heuristics that may potentially work worse than naive exploration but are *definite improvements in terms of the number of states and transitions explored*. Specifically, Basic never explores more states or transitions than a naive mutant execution, OneState never explores more than Basic, StateSet never explores more than OneState, and AllStates never explores more than StateSet. However, the fact that one technique T explores *fewer states or transitions* than another technique T' does not necessarily mean that T has *lower mutant execution time*. After all, the user of MuTMuT cares about the real time the tool takes to report killed mutants and not about the internals of the exploration. It can happen that a technique that explores less actually runs slower since it performs a much costlier collection of additional information in `notifyMutantReachedForCurrentState` or lookup of this information in `specialExplore`. We actually found this for AllStates and StateSet: while AllStates is better in theory, our initial results with a prototype implementation of AllStates showed that it had longer mutant execution times than StateSet. For this reason, we propose StateSet as the best technique for efficient exploration of multithreaded tests for mutation testing. Therefore, our MuTMuT implementation offers the StateSet technique (and also Basic for experimental comparison with StateSet).

IV. Implementation

We implemented MuTMuT in Java PathFinder (JPF), an open-source verification tool for Java [18], [22]. JPF provides an explicit-state model checker that can control thread scheduler, which is necessary for exploration of multithreaded tests. By default, JPF explores all possible schedules. We call that exploration mode *Exhaustive Exploration*, although JPF does not explicitly enumerate all schedules but prunes equivalent schedules/states using advanced partial-order reduction and state matching techniques [18], [23]. Our MuTMuT implementation also supports other exploration modes in JPF, such as bounding the number of thread context switches as in CHESS [19].

We implemented the StateSet technique because it offers the best trade-off between the cost of collecting additional information and the benefit of pruning using this information. We also implemented the Basic technique to compare it experimentally with StateSet. As discussed before, we do not provide an implementation of AllStates since it performed poorly with respect to StateSet in our initial experiments. Our implementation follows the pseudo-code algorithms presented in figures 6, 7, and 9. The key differences are related to (1) the way JPF restores the states for exploration and (2) optimizations that we performed in the implementation compared to the pseudo-code algorithm in Figure 9.

The pseudo-code algorithm in Figure 4 (in particular, line 12) and our description in Section III-C hint that the exploration restores a state by re-executing a path (a sequence of transitions) on the code being explored. While re-execution makes it easier to present the algorithms, JPF actually does not re-execute the code but stores/checkpoints a copy of the state when it is encountered, and then restores the state by re-establishing it from the copy. This allowed us to piggy back on the storing and restoring that JPF already performs such that our implementation of StateSet does not have to explicitly collect the execution path as shown in `notifyMutantReachedForCurrentState` in Figure 9.

Moreover, we leverage JPF's storing and restoring of states to remove the most expensive operation shown in `notifyMutantReachedForCurrentState` in Figure 9, namely checking whether the current path is a suffix of some previous path. Performing such a check on a large set of paths would be costly but is not necessary. Instead, the first time that a mutation is reached on some path, our implementation remembers the path and also adds the `mutantId` to the set of mutants reached on that path. This set of mutants is in the state manipulated by JPF, so it gets properly stored, restored, and backtracked.

We additionally had to extend JPF to support "misbehaving" mutants. These extensions are not particular to multithreaded tests and MuTMuT, but are required for any mutation tool [4], [8], [9]. The problem is that, even if the tests terminate normally on the original code, certain mutations can create mutants that do not terminate (e.g., replace a condition in a `while` loop with `true`), run out of the heap memory (due to loops or recursive calls that allocate objects), overflow the stack (due to recursive calls), or throw other exceptions. Some solutions to this problem involve running each mutant in a separate JVM [9] or in separate threads [27], with an appropriate timeout. However,

program	# of tests	# of mutants	# of killed	# of reached
Account	13	4	4	4
Airline	9	31	18	31
Allocation	9	36	8	29
BubbleSort	6	32	28	32
Buffer	4	15	13	15
LinkedList	16	23	14	18
NoRollback	4	66	51	66
TreeBag	9	19	17	19

Figure 10. Basic statistics about the programs used in the evaluation

program	Exhaustive			CHESS(2)		
	Basic	StateSet	speedup	Basic	StateSet	speedup
Account	1:17:10	0:44:33	43%	02:32	01:49	28%
Airline	0:19:08	0:11:29	44%	11:52	07:46	35%
Allocation	1:13:51	0:52:52	28%	31:46	24:04	24%
BubbleSort	0:59:34	0:21:47	63%	28:00	12:32	55%
Buffer	0:57:14	0:17:22	70%	03:01	01:08	62%
LinkedList	1:16:51	0:43:57	43%	32:23	18:28	43%
NoRollback	0:25:45	0:05:41	77%	04:26	01:35	72%
TreeBag	0:14:17	0:10:33	26%	01:15	00:58	22%

Figure 11. Overall mutant execution time for MuTMuT Basic and StateSet

these solutions cannot be directly applied to JPF if we want to reuse parts of one exploration for another exploration, because these explorations need to be in the same JVM.

Fortunately, JPF already provides support to check for cycles in state spaces and thus detect some infinite loops in executions. A challenge is that a "misbehaving" mutant could have a very long execution of one transition, since it could be infinite and not produce a new state so that a cycle check can be performed. We extended JPF to also check for extremely long transitions. The limit we set is that the number of bytecode instructions executed (on a path) by a mutant cannot be more than twice larger than the number of bytecode instructions executed for the original program (on any path). This is the common way to limit the length of mutant execution based on the original execution [9], [27]. We added similar checks for the number of allocated objects to ensure that JPF does not run out of memory due to a "misbehaving" mutant. Finally, we added code to catch all the exceptions that the mutants could throw and to gracefully recover from them. Note that JPF can easily detect deadlocks (no transition enabled) and recover from them (in its JVM).

V. Evaluation

The key question for MuTMuT is whether it can reduce the running time for mutant execution. We evaluate this for both Exhaustive Exploration and CHESS. Our experiments use JPF version 4.1 with the default configuration setting (for search strategy, state comparison, etc.). All experiments were performed on an Intel Pentium 4 3.4GHz desktop, running Linux 2.6.28-15 and Sun's JVM 1.6.0_10.

A. *Experimental setup*

Figure 10 presents some statistics about the programs that we use in our evaluation. All of the programs, except `Buffer` and `TreeBag`, are taken from an existing benchmark for testing multithreaded programs [24]. `Account` is similar to the example from Section II. `Airline` is a simplified system for selling flight tickets, where each flight has some number of seats, say N, and if more customers try to buy tickets at once, only the first N should be able to do that. `Allocation` is a simple memory manager that can allocate and free blocks in a multithreaded environment. `BubbleSort` implements the bubble sort algorithm using multiple threads. `Buffer` implements a bounded buffer with standard `putItem` and `getItem` operations called by any number of producers and consumers in parallel. `LinkedList` is a linked list data structure that can be used in multithreaded code. `NoRollback` simulates transactions where checking if the transaction is possible and booking an item is done in one method while purchasing an item is done from another method. `TreeBag` implements the unsorted binary tree data structure that can have nodes with repeated values; the method `find(int value, int numOfThreads)` uses multiple threads to count how many nodes have a given value.

Each program has a number of multithreaded tests as shown in the column *# of tests*. To automatically generate mutants for the programs, we use Javalanche [4], a recent tool for mutant generation and mutant execution of sequential tests. We changed Javalanche slightly to generate mutant schemata in the format expected by MuTMuT. Since Javalanche does not implement multithreaded mutation operators, we manually crated five multithreaded mutations in `Buffer` (removed `notify`, changed `notifyAll` to `notify`, and removed `synchronized`) to confirm that MuTMuT can handle such mutations. The column *# of mutants* shows the total number of mutants. The columns *# of killed* and *# of reached* show the number of mutants that the tests kill and reach, respectively. The mutation score for these tests ranges from 22% (for `Allocation`) to 100% (for `Account`). Additionally, in our experiments, CHESS killed all the mutants that Exhaustive Exploration killed, although it could have happened that CHESS, due to the pruning, does not kill all the mutants, as discussed in Section III-D.

B. *Running time*

Figure 11 shows the running time of MuTMuT with Basic and StateSet techniques for both Exhaustive Exploration and CHESS with bound 2 (this bound finds most bugs [19]). The time is in the h:mm:ss format. The speedup that StateSet achieves over Basic ranges from 28% (for `Allocation`) to 77% (for `NoRollback`) for Exhaustive and from 22% (for `TreeBag`) to 72% (for `NoRollback`) for CHESS. Note that CHESS takes much less time than Exhaustive, since CHESS prunes away many thread schedules; for Basic, the ratio of exploration times for Exhaustive and CHESS ranges from 1.5X (for `Airline`) to 30X (for `Account`). However, for both Exhaustive and CHESS, StateSet provides a consistent speedup over Basic.

program	Exhaustive						CHESS(2)					
	Memory		States		Transitions		Memory		States		Transitions	
	Basic	StateSet	Basic	StateSet	Basic	StateSet	Basic	StateSet	Basic	StateSet	Basic	StateSet
Account	113MB	89MB	296666	183610	1426011	845126	96MB	91MB	28070	19942	41133	29621
Airline	69MB	60MB	232083	143855	453131	279723	119MB	60MB	178634	116423	273631	178778
Allocation	185MB	185MB	684204	435960	1552404	959292	168MB	181MB	354061	250090	518198	378118
BubbleSort	118MB	101MB	943050	348750	2378385	849891	220MB	95MB	647056	264489	1092046	453654
Buffer	113MB	82MB	1260249	374538	3219762	888599	99MB	85MB	115973	38070	139121	46266
LinkedList	98MB	88MB	387442	220359	1124495	660951	100MB	82MB	282023	157324	464998	264453
NoRollback	114MB	98MB	636599	133660	1552296	309625	95MB	96MB	174204	53746	258225	327679
TreeBag	81MB	66MB	138467	102361	322844	235371	69MB	76MB	18833	13715	23290	16845

Figure 12. Other exploration statistics for MuTMuT Basic and StateSet

C. *Other exploration statistics*

Figure 12 shows additional exploration statistics for our experiments. For each program, we tabulate the memory that JPF/MuTMuT takes for mutant execution, the number of states explored (nodes in the state space graph, as discussed in Section II), and the number of transitions executed (edges in the state space graph). The results show that StateSet performs consistently better than Basic. The only exception is the memory required for `Allocation`, `NoRollback`, and `TreeBag` when using CHESS. We believe that this is due to the way that JPF allocates memory and sometimes ends up using more memory for a smaller exploration. StateSet achieves smaller explorations in terms of the number of states and transitions. The reduction in these numbers strongly correlates with the reduction in the overall mutant exploration time, so approximating the exploration time with the size of the exploration graph provides a good approximation [25].

VI. Related work

Mutation testing was introduced over three decades ago [1], [2], and since then a lot of work has been done on mutation testing for various languages including Ada, C, Cobol, C#, Fortran, Java, and SQL. An overview of this work can be found in the survey report by Jia and Harman [28]. The main focus of previous work has been on mutation operators, and effective and efficient *mutant generation*. In contrast, MuTMuT focuses on *mutant execution*, in particular for multithreaded tests. Note that mutant generation and execution are orthogonal; MuTMuT can use any mutant generation tool, but rather than implementing a new tool, we use Javalanche [4].

Since we implemented MuTMuT for Java (although our ideas translate into any other language with multithreading), we describe two more mutation tools for Java. MuJava [8] focuses on operators specific to Java and was among the first mutation tools for Java [27]. MuJava supports two types of operators: class-level operators and method-level operators. Jumble [9] was developed with the primary goal of efficient mutant execution in a real industrial environment. Jumble implements several heuristics that try to order tests to kill mutants faster, and it avoids running tests that would kill the same mutants. MuTMuT also monitors the exploration and executes the tests just for the mutants that are live and reached with the current test. MuTMuT currently does not implement heuristics for ordering tests, but those heuristics could further improve the performance of MuTMuT. The Jumble paper [9] briefly mentions that Jumble can work with multithreaded code, but the paper provides no details about the thread schedules explored.

Mutation has been considered for concurrent code [10], including for Java [11]. Carver [10] describes mutation in a multithreaded environment. Instead of executing all possible thread schedules, the approach is to execute multithreaded code deterministically, by specifying a set of schedules, and requiring that for each schedule, the original code and mutants give equivalent results. This is a weaker notion of mutant killing than requiring that the set of all possible results produced by a mutant is the same as the set of all possible results produced by the original code (even if the results for some schedule differ). Bradbury et al. [11] define concurrency mutation operators for Java, specifically for J2SE 5.0. These operators can modify thread or synchronization operations, for example removing locks. In our MuTMuT evaluation, we manually applied some such operators, since they were not available in Javalanche. Bradbury et al. [12] also use mutation testing to measure the effectiveness and efficiency of various testing and formal analysis tools, including JPF. Our goal is to speed up mutant execution using JPF.

Another issue in mutation testing, besides mutant generation and mutant execution, is finding equivalent mutants. While determining program equivalence is undecidable in general, recent work [4] shows how to rank the mutants by the likelihood that they are non-equivalent to the original code. This ranking helps in prioritizing inspection of live mutants to generate new test cases to kill those mutants. The work is orthogonal to MuTMuT, which focuses on efficient mutant execution. Also, the projects differ in focusing on sequential versus multithreaded code.

Fleyshgakker and Weiss present efficient mutation analysis [29] that shares the goal of our work to speed up mutant execution. There is further similarity in matching states, as in our AllStates technique. However, the key difference is that their work is for sequential tests and so does not consider the issues that arise for exploration of multithreaded tests.

Our work on incremental state-space exploration (ISSE) [30] is conceptually similar to MuTMuT as ISSE speeds up the exploration by reusing some results. However, ISSE works only for sequential code and reuses exploration during code evolution, not during mutation testing.

VII. Conclusions

Testing multithreaded code is becoming more important as such code is developed and used more frequently. Mutation testing is a well established testing approach, but one of its major cost is the time to execute a test suite on multiple mutants. This cost is exacerbated for multithreaded code where each test needs to be executed for many thread schedules. We presented an efficient exploration framework, MuTMuT, that can significantly reduce the time for mutant execution of multithreaded code. The experiments show that the MuTMuT StateSet technique speeds up MuTMuT Basic up to 77%. Motivated by these results for mutation testing, we plan to tailor other testing approaches for multithreaded code in the future.

Acknowledgements

We thank Yaniv Eytani for providing many of the multithreaded programs used in our evaluation, and David Schuler and Andreas Zeller for help with Javalanche. This material is based upon work partially supported by the National Science Foundation under Grant Nos. CCF-0916893 and CCF-0746856, and by Intel and Microsoft under the Universal Parallel Computing Research Center (UPCRC). Milos Gligoric was supported by the Saburo Muroga fellowship.

References

[1] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE TSE*, vol. 3, 1977.

[2] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, 1978.

[3] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2008.

[4] D. Schuler, V. Dallmeier, and A. Zeller, "Efficient mutation testing by checking invariant violations," in *ISSTA 2009*.

[5] S. Kim, J. Clark, and J. McDermid, "The rigorous generation of Java mutation operators using HAZOP," in *ICSSEA*, 1999.

[6] ——, "Class mutation: Mutation testing for object oriented programs," in *FMES*, 2000.

[7] Y.-S. Ma, Y.-R. Kwon, and J. Offutt, "Inter-class mutation operators for Java," in *ISSRE*, 2002.

[8] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "MuJava: A mutation system for Java," in *ICSE*, 2006.

[9] S. A. Irvine, T. Pavlinic, L. Trigg, J. G. Cleary, S. Inglis, and M. Utting, "Jumble Java Byte Code to Measure the Effectiveness of Unit Tests," in *MUTATION*, 2007.

[10] R. H. Carver, "Mutation-based testing of concurrent programs," in *ITC*, 1993.

[11] J. S. Bradbury, J. R. Cordy, and J. Dingel, "Mutation operators for concurrent Java (J2SE 5.0)," in *MUTATION*, 2006.

[12] ——, "Comparative assessment of testing and model checking using program mutation," in *MUTATION*, 2007.

[13] B. Long, D. Hoffman, and P. A. Strooper, "Tool support for testing concurrent Java components," *IEEE TSE*, 2003.

[14] W. Pugh and N. Ayewah, "Unit testing concurrent software," in *ASE*, 2007.

[15] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur, "Applications of synchronization coverage," in *PPoPP*, 2005.

[16] S. Park, S. Lu, and Y. Zhou, "CTrigger: Exposing atomicity violation bugs from their hiding places," in *ASPLOS*, 2009, pp. 25–36.

[17] P. Godefroid, "Model checking for programming languages using VeriSoft," in *POPL*, 1997.

[18] W. Visser, K. Havelund, G. Brat, and S. Park, "Model checking programs," *Springer ASE-J*, vol. 10, 2003.

[19] M. Musuvathi and S. Qadeer, "Iterative context bounding for systematic testing of multithreaded programs," in *PLDI*, 2007.

[20] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby, "Runtime model checking of multithreaded C/C++ programs," in *SPIN*, 2007.

[21] G. J. Holzmann, R. Joshi, and A. Groce, "Tackling large verification problems with the swarm tool," in *SPIN*, 2008.

[22] "JPF home page," http://babelfish.arc.nasa.gov/trac/jpf/.

[23] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 1999.

[24] Y. Eytani, K. Havelund, S. D. Stoller, and S. Ur, "Towards a framework and a benchmark for testing tools for multithreaded programs," *Wiley CC-PE*, vol. 19, 2007.

[25] M. B. Dwyer, S. Elbaum, S. Person, and R. Purandare, "Parallel randomized state-space search," in *ICSE*, 2007.

[26] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," in *ISSTA*, 1993.

[27] D. Marinov, "Automatic testing of software with structurally complex inputs," Ph.D. dissertation, MIT, 2004.

[28] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," King's College London, Tech. Rep., 2009.

[29] V. N. Fleyshgakker and S. N. Weiss, "Efficient mutation analysis: A new approach," in *ISSTA*, 1994.

[30] S. Lauterburg, A. Sobeih, M. Viswanathan, and D. Marinov, "Incremental state-space exploration for programs with dynamically allocated data," in *ICSE*, 2008.