FSE'11, ACM, pp 223-233. 2011

FSE'11, ACM, pp 223-233. 2011

Improved Multithreaded Unit Testing

Vilas Jagannath, Milos Gligoric, Dongyun Jin,
Qingzhou Luo, Grigore Roşu, Darko Marinov
Department of Computer Science, University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{vbangal2, gliga, djin3, qluo2, grosu, marinov}@illinois.edu

ABSTRACT

Multithreaded code is notoriously hard to develop and test. A multithreaded test exercises the code under test with two or more threads. Each test execution follows some schedule/interleaving of the multiple threads, and different schedules can give different results. Developers often want to enforce a particular schedule for test execution, and to do so, they use time delays (`Thread.sleep` in Java). Unfortunately, this approach can produce false positives or negatives, and can result in unnecessarily long testing time.

This paper presents IMUnit, a novel approach to specifying and executing schedules for multithreaded tests. We introduce a new language that allows explicit specification of schedules as *orderings on events* encountered during test execution. We present a tool that automatically instruments the code to control test execution to follow the specified schedule, and a tool that helps developers migrate their legacy, sleep-based tests into event-based tests in IMUnit. The migration tool uses novel techniques for inferring events and schedules from the executions of sleep-based tests. We describe our experience in migrating over 200 tests. The inference techniques have high precision and recall of over 75%, and IMUnit reduces testing time compared to sleep-based tests on average 3.39x.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Verification, Reliability

Keywords

IMUnit, Unit Testing, Multithreaded Code

1. INTRODUCTION

Multicore processors are here to stay. To extract greater performance from multicore processors, developers need to write parallel code. The predominant paradigm for parallel code is that of shared memory where multiple threads of control communicate by reading and writing shared data objects. Shared-memory multithreaded code is often afflicted by concurrency bugs, which are hard to detect because multithreaded code can demonstrate different behavior based on the scheduling of threads, and the bugs may only be triggered by a small specific set of schedules.

To validate their multithreaded code, developers write multithreaded unit tests. A multithreaded test creates and executes two or more threads (and/or invokes code under test that itself creates and executes two or more threads). Each test execution follows some schedule/interleaving of the multiple threads, and different schedules can give different results. Developers often want to enforce a particular schedule for a test. For example, consider two threads, one executing a method m and the other executing a method m'. Developers may want to ensure in one test that m finishes before m' starts and in another test that m' finishes before m starts (and in more tests that m and m' interleave in certain ways). Without controlling the schedule, it is impossible to write precise assertions about the execution because the results can differ in the two scenarios, and it is impossible to guarantee which scenarios were covered during testing, even if multiple runs are performed.

To control the schedule of multithreaded tests, developers mostly use a combination of timed delays in the various test threads. In Java, the delay is performed with the `Thread.sleep` method, so we call this approach *sleep-based*. A sleep pauses a thread while other threads continue execution. Using a combination of sleeps, developers attempt to enforce the desired schedule during the execution of a multithreaded test, and then assert the intended result for the desired schedule. A sleep-based test can fail when an undesired schedule gets executed even if the code under test has no bug (false positive). Dually, a sleep-based test can pass when an unintended schedule gets executed even if the code under test has a bug (false negative). Indeed, sleeps are an unreliable and inefficient mechanism for enforcing schedules. To use sleeps, one has to estimate the real-time duration for which to delay a thread while the other threads perform their work. This is usually estimated by trial and error, starting from a small duration and increasing it until the test passes consistently on the developer's machine. The estimated duration depends on the execution environment (hardware/software configuration and the load on the machine). Therefore, when the same test is executed in a different environment, the intended schedule may not be en-

ESEC/FSE'11, September 5–9, 2011, Szeged, Hungary.

FSE'11, ACM, pp 223-233. 2011

forced, leading to false positives/negatives. Moreover, sleep can be overly inaccurate on a single machine [20]. In an attempt to mitigate the unreliability of sleep, developers often end up over-estimating the duration, which in turn leads to slow running multithreaded tests.

Researchers have previously noted the numerous problems with using sleeps to specify schedules in multithreaded tests and have developed frameworks such as ConAn [22, 23], ConcJUnit [27], MultithreadedTC [26], and ThreadControl [13] to tackle some problems in specifying and enforcing schedules in multithreaded unit tests. However, despite these frameworks, multithreaded unit testing still has many issues that could be categorized as follows:

Readability: Most current frameworks force developers to reason about the execution of threads relative to a global clock. This is unintuitive since developers usually reason about the execution of their multithreaded code in terms of event relationships (such as m finishing before m' starts). Some frameworks require users to write schedules in external scripts, which makes it even more difficult to reason about schedules. In other frameworks the schedule is implicit, as a part of the unit test code, and hence it is difficult to focus on the schedule and reason about it separately at a higher level.

Modularity: In some current frameworks, the intended schedule is intermixed with the test code and effectively hard-coded into a multithreaded unit test. This makes it difficult to specify multiple schedules for a particular unit test and/or to reuse test code among different tests.

Reliability: Some current frameworks, as well as the legacy sleep-based tests, rely on real time. As explained, this makes them very fragile, leading to false positives/negatives and/or slow testing time.

Migration Costs: Most current frameworks are very different from the traditional sleep-based tests. This makes it costly to migrate the existing sleep-based tests.

We present a new framework, called *IMUnit* (*pronounced "immunity"*), which aims to address these issues with multithreaded unit testing. Specifically, we make the following contributions:

Schedule Language: IMUnit introduces a novel language that enables natural and explicit specification of schedules for multithreaded unit tests. Semantically, the basic entity in an IMUnit schedule is an *event* that an execution can produce at various points (e.g., a thread starting/finishing the execution of a method, or a thread getting blocked). We call the IMUnit approach *event-based*. An IMUnit schedule itself is a (monitorable) property [10, 24] on the sequence of events. More precisely, each schedule is expressed as a set of desirable *event orderings*, where each event ordering specifies the order between a pair of events (note that an IMUnit schedule need not specify a total order between all events but only the necessary partial order).

While the ideas of IMUnit can be embodied in any language, we have developed an implementation for Java. Syntactically, the IMUnit constructs are represented using Java annotations. A developer can use `@Event` and `@Schedule` annotations to describe the events and intended schedules, respectively, for a multithreaded unit test.

Automated Migration: We have developed two inference techniques and a tool to ease migration of legacy, sleep-based tests to IMUnit, event-based tests. Our inference techniques can automatically infer likely relevant events and schedules from the execution traces of existing sleep-based tests. We implemented our migration tool as an Eclipse plugin which uses the results of inference to automatically refactor a given multithreaded test into an IMUnit test.

Execution and Checking: We have implemented a tool for execution of IMUnit multithreaded unit tests. The tool can work in two modes. In the active mode, it controls the thread scheduler to enforce a given IMUnit schedule during test execution. In the passive mode, it checks whether an arbitrary test execution, controlled by the regular JVM thread scheduler, follows a given IMUnit schedule. To enforce/check the schedules, our tool uses the JavaMOP monitoring framework [10, 24]. We also include a new runner for the standard JUnit testing framework to enable execution of IMUnit tests with our enforcement/checking tool.

Evaluation: To guide and refine our design of the IMUnit language, we have been inspecting over 200 sleep-based tests from several open-source projects. We manually translated 198 of those tests into IMUnit, adding events and schedules, and removing sleeps. As a result, the current version of IMUnit is highly expressive, and we were able to remove all sleeps from all but 4 tests.

We evaluated our inference techniques by automatically inferring events/schedules for the original tests that we manually translated (the subprojects on manual translation and automatic inference were performed by different authors to reduce the direct bias of manual translation on automatic inference). Computing the precision and recall of the automatically inferred events/schedules with respect to the manually translated events/schedules, we find our techniques to be highly effective, with over 75% precision and recall.

We also compared the execution time of the original tests and our translated tests. Because the main goal of IMUnit is to make tests more readable, modular, and reliable, we did not expect IMUnit to run faster. However, IMUnit did reduce the testing time, on average 3.39x, compared to the sleep-based tests, with the sleep duration that the original tests had in the code. As mentioned earlier, these duration values are often over-estimated, especially in older tests that were written for slower machines. In summary, IMUnit not only makes multithreaded unit tests more readable, modular, and reliable than the traditional sleep-based approach, but IMUnit can also make test execution faster.

This paper makes progress on our vision for improving multithreaded unit testing; our position paper [15] proposed the idea of event-based specification of schedules, but the IMUnit language and algorithms/tools for inference and execution are completely new.

2. EXAMPLE

We now illustrate IMUnit with the help of an example multithreaded unit test for the ArrayBlockingQueue class in `java.util.concurrent` (JSR-166) [17]. ArrayBlockingQueue is an array-backed implementation of a bounded blocking queue. One operation provided by ArrayBlockingQueue is `add`, which performs a non-blocking insertion of the given element at the tail of the queue. If `add` is performed on a full queue, it throws an exception. Another operation provided by ArrayBlockingQueue is `take`, which removes and returns the object at the head of the queue. If `take` is performed on an empty queue, it blocks until an element is inserted into the queue. These operations could have bugs that get triggered when the `add` and `take` operations execute on different threads. Consider testing some scenarios for these opera-

FSE'11, ACM, pp 223-233. 2011

```
1  @Test
2  public void testTakeWithAdd() {
3    ArrayBlockingQueue<Integer> q;
4    q = new ArrayBlockingQueue<Integer>(1);
5    new Thread(
6      new CheckedRunnable() {
7        public void realRun() {
8          q.add(1);
9          Thread.sleep(100);
10         q.add(2);
11       }
12     }, "addThread").start();
13   Thread.sleep(50);
14   Integer taken = q.take();
15   assertTrue(taken == 1 && q.isEmpty());
16   taken = q.take();
17   assertTrue(taken == 2 && q.isEmpty());
18   addThread.join();
19 }
```

(a) JUnit

```
1  public class TestTakeWithAdd
2      extends MultithreadedTest {
3    ArrayBlockingQueue<Integer> q;
4    @Override
5    public void initialize() {
6      q = new ArrayBlockingQueue<Integer>(1);
7    }
8    public void addThread() {
9      q.add(1);
10     waitForTick(2);
11     q.add(2);
12   }
13   public void takeThread() {
14     waitForTick(1);
15     Integer taken = q.take();
16     assertTrue(taken == 1 && q.isEmpty());
17     taken = q.take();
18     assertTick(2);
19     assertTrue(taken == 2 && q.isEmpty());
20   }
21 }
```

(b) MultithreadedTC

```
1  @Test
2  @Schedule("finishedAdd1->startingTake1,
3             [startingTake2]->startingAdd2")
4  public void testTakeWithAdd() {
5    ArrayBlockingQueue<Integer> q;
6    q = new ArrayBlockingQueue<Integer>(1);
7    new Thread(
8      new CheckedRunnable() {
9        public void realRun() {
10         q.add(1);
11         @Event("finishedAdd1")
12         @Event("startingAdd2")
13         q.add(2);
14       }
15     }, "addThread").start();
16   @Event("startingTake1")
17   Integer taken = q.take();
18   assertTrue(taken == 1 && q.isEmpty());
19   @Event("startingTake2")
20   taken = q.take();
21   assertTrue(taken == 2 && q.isEmpty());
22   addThread.join();
23 }
```

(c) IMUnit

Figure 1: Example multithreaded unit test for ArrayBlockingQueue

tions (in fact, the JSR-166 TCK provides over 100 tests for various scenarios for similar classes).

Figure 1 shows a multithreaded unit test that ArrayBlockingQueue exercises `add` and `take` in two scenarios. In particular, Figure 1(a) shows the test written as a regular JUnit test method, with sleeps used to specify the required schedule. We invite the reader to consider what scenarios are specified with that test (without looking at the other figures). It is likely to be difficult to understand which schedule is being exercised by reading the code of this unit test. While the sleeps provide hints as to which thread is waiting for another thread to perform operations, it is unclear which operations are intended to be performed by the other thread before the sleep finishes.

The test actually checks that `take` performs correctly both with and without blocking, when used with `add` from another thread. To check both scenarios, the test exercises a schedule where the first `add` finishes before the first `take` starts, and the second `take` blocks before the second `add` starts. Line 13 shows the first sleep that is intended to pause the `main` thread[1] while the `addThread` finishes the first `add`. Line 9 shows the second sleep which is intended to pause the `addThread` while the `main` thread finishes the first `take` and then proceeds to block while performing the second `take`. If the specified schedule is not enforced during the execution, there may be a false positive/negative. For example, if both `add` operations execute before a `take` is performed, the test will throw an exception and fail even if the code has no bug, and if both `take` operations finish without blocking, the test will not fail, even if the blocking `take` code had a bug.

Figure 1(b) shows the same test written using MultithreadedTC [26]. Note that it departs greatly from traditional JUnit where each test is a method. In MultithreadedTC, each test has to be written as a class, and each method in the test class contains the code executed by a thread in the test. The intended schedule is specified with respect to a global, logical clock. Since this clock measures time in *ticks*, we call the approach tick-based. When a thread executes a `waitForTick`, it is blocked until the global clock reaches the required tick. The clock advances implicitly when all threads are blocked (and at least one thread is blocked in a `waitForTick`). While a MultithreadedTC test does not rely on real time, and is thus more reliable than a sleep-based test, the intended schedule is still not immediately clear upon reading the test code. It is especially not clear when `waitForTick` operations are blocked/unblocked, because ticks are advanced implicitly when all the threads are blocked.

Figure 1(c) shows the same test written using IMUnit. The interesting events encountered during test execution are marked with `@Event` annotations[2], and the intended schedule is specified with a `@Schedule` annotation that contains a comma-separated set of *orderings* among events. An ordering is specified using the operator `->`, where the left event is intended to execute before the right event. An event specified within square brackets denotes that the thread executing that event is intended to block after that event. It should be clear from reading the schedule that the `addThread` should finish the first `add` before the `main` thread starts the first `take`, and that the `main` thread should block while performing the second `take` before the `addThread` starts the second `add`.

We now revisit, in the context of this example, the issues with multithreaded tests listed in the introduction. In terms of *readability*, we believe that making the schedules explicit, as in IMUnit, allows easier understanding and maintenance of schedules and code for both testing and debugging. In terms of *modularity*, IMUnit allows extracting the `addThread` as a helper thread (with its events) that can be reused in

[1]JVM names the thread that starts the execution `main` by default, although the name can be changed later.

[2]Note that `@Event` annotations appear on statements. The current version of Java (ver. 6) does not support annotations on statements, but the upcoming version of Java (ver. 7) will add such support. For now, `@Event` annotations can be written as comments, e.g., `/* @Event("finishedAdd1") */`, which IMUnit translates into code for test execution.

```
<Schedule>     ::= { <Ordering> [","] } <Ordering>
<Ordering>     ::= <Condition> "->" <Basic Event>
<Condition>    ::= <Basic Event> | <Block Event>
                 | <Condition> "||" <Condition>
                 | <Condition> "&&" <Condition>
                 | "(" <Condition> ")"
<Basic Event>  ::= <Event Name> ["@" <Thread Name>]
                 | "start" "@" <Thread Name>
                 | "end" "@" <Thread Name>
<Block Event>  ::= "[" <Basic Event> "]"
<Event Name>   ::= { <Id> "." } <Id>
<Thread Name>  ::= <Id>
```

Figure 2: Syntax of the IMUnit schedule language

other tests (in fact, many tests in the JSR-166 TCK [17] use such helper threads). In contrast, reusing thread methods from the MultithreadedTC test class is more involved, requiring subclassing, parametrizing tick values, and providing appropriate parameter values. Also, IMUnit allows specifying multiple schedules for the same test code (Section 4.3). In terms of *reliability*, IMUnit does not rely on real time and hence has no false positives/negatives due to unintended schedules. In terms of *migration costs*, IMUnit tests resemble legacy JUnit tests more than MultithreadedTC tests. This similarity eases the transition of legacy tests into IMUnit: in brief, add `@Event` annotations, add `@Schedule` annotation, and remove `sleep` calls. Section 4 presents our techniques and tool that automate this transition.

3. SCHEDULE LANGUAGE

We now describe the syntax and semantics of the language used in IMUnit's schedules.

3.1 Concrete Syntax

Figure 2 shows the concrete syntax of the implemented IMUnit schedule language. An IMUnit schedule is a comma-separated set of *orderings*. Each ordering defines a condition that must hold before a basic event can take place. A *basic event* is an event name possibly tagged with its issuing thread name when that is not understood from the context. An *event name* is any identifier, possibly prefixed with a qualified class name. There are two implicit event names for each thread, `start` and `end`, indicating when the thread starts and terminates. Any other event must be explicitly introduced by the user with the `@Event` annotation (see Figure 1(c)). A *condition* is a conjunctive/disjunctive combination of basic and block events, where block events are written as basic events in square brackets. A *block event* $[e']$ in the condition c of an ordering $c \to e$ states that e' must precede e and, additionally, the thread of e' is blocked when e takes place.

3.2 Schedule Logic

It is more convenient to define a richer logic than what is currently supported by our IMUnit implementation; the additional features are natural and thus may also be implemented in the future. The semantics of our logic is given in Section 3.3; here is its syntax:

$$
\begin{array}{rcl}
a & ::= & \mathit{start} \mid \mathit{end} \mid \mathit{block} \mid \mathit{unblock} \mid \text{event names} \\
t & ::= & \text{thread names} \\
e & ::= & a@t \\
\varphi & ::= & [t] \mid \varphi \to \varphi \mid \text{usual propositional connectives}
\end{array}
$$

The intuition for $[t]$ is "thread t is blocked" and for $\varphi \to \psi$ "if ψ held in the past, then φ must have held at some moment before ψ". We call these two temporal operators the *block* and the *ordering* operators, respectively. For uniformity, all events are tagged with their thread. There are four implicit events: $\mathit{start}@t$ and $\mathit{end}@t$ were discussed above, and $\mathit{block}@t$ and $\mathit{unblock}@t$ correspond to when t gets blocked and unblocked[3].

For example, the following formula in our logic

$$
\begin{array}{rl}
 & (a_1@t_1 \wedge ([t_2] \vee (\neg(\mathit{start}(t_2) \to a_1@t_1)))) \to a_2@t_2 \\
\wedge & (a_2@t_2 \wedge ([t_1] \vee (\mathit{end}(t_1) \to a_2@t_2))) \to a_2@t_2
\end{array}
$$

says that if event a_2 is generated by thread t_2 then: (1) event a_1 must have been generated before that and, when a_1 was generated, t_2 was either blocked or not started yet; and (2) when a_2 is generated by t_2, t_1 is either blocked or terminated. As explained shortly, every event except for *block* and *unblock* is restricted to appear at most once in any execution trace. Above we assumed that $a_1, a_2 \notin \{\mathit{block}, \mathit{unblock}\}$.

Before we present the precise semantics, we explain how our current IMUnit language shown in Figure 2 (whose design was driven exclusively by practical needs) is a smaller fragment of the richer logic. An IMUnit schedule is a conjunction (we use comma instead of $\wedge$) of orderings, and schedules cannot be nested. Since generating *block* and *unblock* events is expensive, IMUnit currently disallows their explicit use in schedules. Moreover, to reduce their implicit use to a fast check of whether a thread is blocked or not, IMUnit also disallows the explicit use of $[t]$ formulas. Instead, it allows *block events* of the form $[a@t]$ (note the square brackets) in conditions. Since negations are not allowed in IMUnit, and since we can show (after we discuss the semantics) that $(\varphi_1 \vee \varphi_2) \to \psi$ equals $(\varphi_1 \to \psi) \vee (\varphi_2 \to \psi)$, we can reduce any IMUnit schedule to a Boolean combination of orderings $\varphi \to e$, where φ is a conjunction of basic events or block events. All that is left to show is how block events are desugared. Consider an IMUnit schedule $(\varphi \wedge [a_1@t_1]) \to a_2@t_2$, saying that $a_1@t_1$ and φ must precede $a_2@t_2$ *and* t_1 is blocked when $a_2@t_2$ occurs. This can be expressed as $((\varphi \wedge a_1@t_1) \to a_2@t_2) \wedge ((a_2@t_2 \wedge [t_1]) \to a_2@t_2)$, relying on $a_2@t_2$ happening at most once.

3.3 Semantics

Our schedule logic is a carefully chosen fragment of *past-time linear temporal logic (PTLTL)* over special well-formed multithreaded system execution traces.

Program executions are abstracted as finite traces of events $\tau = e_1 e_2 \ldots e_n$. Unlike in conventional LTL, our traces are finite because unit tests always terminate. Traces must satisfy the obvious condition that events corresponding to thread t can only appear while the thread is alive, that is, between $\mathit{start}@t$ and $\mathit{end}@t$. Using PTLTL, this requirement states that for any trace τ and any event $a@t$ with $a \notin \{\mathit{start}, \mathit{end}\}$, the following holds:

$$\tau \vDash \neg \diamondsuit (a@t \wedge (\diamondsuit \mathit{end}@t \vee \neg \diamondsuit \mathit{start}@t))$$

where $\diamondsuit$ stands for "eventually in the past". Moreover, except for $\mathit{block}@t$ and $\mathit{unblock}@t$ events, we assume that each

[3]It is expensive to explicitly generate *block*/*unblock* events in Java precisely when they occur, because it requires polling the status of each thread; our currently implemented fragment only needs, through its restricted syntax, to check if a given thread is currently blocked or not, which is fast.

event appears at most once in a trace. With PTLTL, this says that the following must hold (⊙ is "previously"):

$$\tau \vDash \neg \diamond (a@t \wedge \odot \diamond a@t)$$

for any trace τ and any $a@t$ with $a \notin \{block, unblock\}$.

The semantics of our logic is defined as follows:

$e_1 e_2 \ldots e_n \vDash e$ iff $e = e_n$

$\tau \vDash \varphi \ \wedge/\vee \ \psi$ iff $\tau \vDash \varphi$ and/or $\tau \vDash \psi$

$e_1 e_2 \ldots e_n \vDash [t]$ iff $(\exists 1 \le i \le n)\ (e_i = block@t$ and $(\forall i < j \le n)\ e_j \ne unblock@t)$

$e_1 e_2 \ldots e_n \vDash \varphi \rightarrow \psi$ iff $(\forall 1 \le i \le n)\ e_1 e_2 \ldots e_i \nvDash \psi$ or $(\exists 1 \le i \le n)\ (e_1 e_2 \ldots e_i \vDash \psi$ and $(\exists 1 \le j \le i)\ e_1 e_2 ... e_j \vDash \varphi)$

It is not hard to see that the two new operators $[t]$ and $\varphi \rightarrow \psi$ can be expressed in terms of PTLTL as

$$\begin{array}{rcl} [t] & \equiv & \neg unblock@t \ \mathcal{S} \ block@t \\ \varphi \rightarrow \psi & \equiv & \boxdot \neg \psi \ \vee \ \diamond (\psi \wedge \diamond \varphi) \end{array}$$

where $\mathcal{S}$ stands for "since" and $\boxdot$ for "always in the past".

4. MIGRATION

We now describe the process of migrating legacy, sleep-based tests to IMUnit, event-based tests. First we present the steps that are typically performed during manual migration and then we describe the automated support that we have developed for key steps of the migration.

4.1 Manual Migration

Based on our experience of manually migrating over 200 tests, the migration process typically follows these steps:

Step 1: Optionally add explicit names for threads in the test code (by using a thread constructor with a name or by adding a call to `setName`). This step is required if events are tagged with their thread name (e.g. `finishedAdd1@addThread`) in the schedule, because by default the JVM automatically assigns a name (e.g. `Thread-5`) for each thread created without an explicit name, and the automatic name may differ between JVMs or between different runs on the same JVM.

Step 2: Introduce `@Event` annotations for the events relevant for the intended schedule. Some of these annotations will be used for block events and some for basic events.

Step 3: Introduce a `@Schedule` annotation for the intended schedule. Steps 2 and 3 are the hardest to perform as they require understanding of the intended behavior of the sleep-based test. Note that a schedule with too few orderings can lead to failing tests that are false positives. On the other hand, a schedule with too many orderings may lead to false negatives whereby a bug is missed because the schedule is over-constraining the test execution.

Step 4: Check that the orderings in the introduced schedule are actually satisfied when running the test with sleeps (Section 5 describes the passive, checking mode).

Step 5: Remove sleeps.

Step 6: Optionally merge multiple tests with different schedules (but similar test code) into one test with multiple schedules, potentially adding schedule-specific code (Section 4.3).

4.2 Automated Migration

We have developed automated tool support to enable easier migration of sleep-based tests to IMUnit. In particular, we have developed inference techniques that can compute likely relevant events and schedules for sleep-based tests by inspecting the execution logs obtained from test runs. We next describe the common infrastructure for logging the test runs. We then present the techniques for inferring events and schedules.

```
enum EntryType { SLEEP_CALL, SLEEP_RETURN, BLOCK_CALL,
      BLOCK_RETURN, OTHER_CALL, OTHER_RETURN, TH_START,
      TH_END, EVENT }
class LogEntry { EntryType type; ThreadID tid; String info; StmtID sid; }
```

Figure 3: Log Entries

4.2.1 Lightweight Logging

Our inference of events and schedules from sleep-based tests is dynamic: it first instruments the test code (using AspectJ [19]) to emit entries potentially relevant for inference, then runs the instrumented code (several times, as explained below) to collect logs of entries from the test executions, and finally analyzes the logs to perform the inference.

Figure 3 shows the generic representation for log entries, although event and schedule inference require slightly different representations. Each log entry has a type, name/ID of the thread that emits the entry, potential info/parameters for the entry, and the ID of the statement that creates the entry (which is used only for event inference). The types of log entries and their corresponding `info` are as follows:

`SLEEP_CALL`: Invocation of `Thread.sleep` method. (Only used for inferring events.)

`SLEEP_RETURN`: Return from `Thread.sleep` method.

`BLOCK_CALL`: Invocation of a thread blocking method (`LockSupport.park` or `Object.wait`).

`BLOCK_RETURN`: Return from a thread blocking method.

`OTHER_CALL`: Invocation of a method (other than those listed above) in the test class. The `info` is the method name. (Only used for inferring events .)

`OTHER_RETURN`: Return from a method executed from the test class.

`TH_START`: Invocation of `Thread.start`. The `info` is the ID of the started thread. (Only used for inferring schedules.)

`TH_END`: End of thread execution.

`EVENT`: Execution of an IMUnit event. The `info` is the name of the event. (Only available while inferring schedules.)

Note that any logging can affect timing of test execution. Because sleep-based tests are especially sensitive to timing, care must be taken to avoid false positives. We address this in three ways. First, our logging is lightweight. The instrumented code only collects log entries (and their parameters) relevant to the inference. For example, `OTHER_CALL` is not collected for schedule inference. Also, the entries are buffered in memory during test execution, and they are converted to strings and logged to file only at the end of test execution. While keeping entries in memory would not work well for very long logs, it works quite well for the relatively short logs produced by test executions. Second, our instrumentation automatically scales the duration of sleeps by a given constant N to compensate for the logging overhead. For example, for $N = 3$ it increases all sleep times 3x. Increasing all the durations almost never makes a passing test fail, but it does make the test run slower. Third, we perform multiple

runs of each test and only collect logs for passing runs. This increases the confidence that the logs indeed correspond to the intended schedules specified with sleeps.

4.2.2 Inferring Events

Figure 4 presents the algorithm for inferring IMUnit events from a sleep-based test. The input to the algorithm consists of a set of logs (as described in Section 4.2.1) and a `confidenceThreshold`. The output is a set of inferred events. Each event includes the code location where `@Event` annotation should be added and the name of the event. The intuition behind the algorithm is that `SLEEP_CALL` log entries are indicative of code locations for events. More precisely, a thread t calls `sleep` to wait for one or more events to happen on other threads (those will be "finished" events) before an event happens on t (that will be a "starting" event). Recall our example from Section 2. When the `main` thread calls `sleep`, it waits for `add` to finish before `take` starts, and thus `finishedAdd1` executes before `startingTake1`.

For each log, the algorithm first computes a set of *regions*, each of which is a sequence of log entries between `SLEEP_CALL` and the matching `SLEEP_RETURN` executed by the same thread. The log entries executed by other threads within a region are potential points for the "finished" events. Regions from different threads can be partially or completely overlapping, but regions from the same thread are disjoint (i.e., each `SLEEP_CALL` is followed directly by `SLEEP_RETURN` before any other statement is executed by the thread). Figure 5 shows two regions for a simplified log produced by our running example. In pseudo-code, each region is represented as a pair of `int`s that point to the beginning and end of the region in the list of log entries. For each region, the algorithm first calls `addFinishedEvents` to potentially add some "finished" events for threads other than the region's thread. If an event is added, the algorithm calls `addStartingEvent` to add the matching "starting" event.

The procedure `addFinishedEvents` potentially adds an inferred event for each thread that executes at least one statement in the region. For each such thread, the procedure first discovers a *relevant* statement, which is one of `SLEEP_CALL`, `BLOCK_CALL`, and `TH_END`. Only threads that have exactly one relevant statement in the region are considered. The intuition is that sleeps usually wait for exactly one event in each other thread. If a thread executes none or multiple relevant statements, it is most likely independent of the thread that started the region and therefore can be ignored. Figure 5 shows the relevant statements for each region. The procedure then finds the `OTHER_RETURN` statement immediately before the relevant statement for each thread. This statement determines the name for the new "finished" `StaticEvent`, whereas the relevant statement determines the location. Note that logging only method calls would not be enough to properly determine the previous statement since the call can come from a helper method in the test class. For our example, these before log entries are `OTHER_RETURN(add), addThread, 326` and `OTHER_RETURN(take), main, 336` (Fig. 5).

The procedure `addStartingEvent` adds an event for the thread that starts the region. The event is placed just before the first statement that follows the end of the region. The type of the statement can be any, including `OTHER_CALL`. The same statement is used for naming the event. In Figure 5, `OTHER_CALL(take), main, 336` and `OTHER_CALL(add), addThread, 330` are found following the algorithm.

```
1  // Input
2  Set⟨List⟨LogEntry⟩⟩ logs;
3  float confidenceThreshold;
4  // Output
5  class StaticEvent { StmtID sid; String name; }
6  Set⟨StaticEvent⟩ events;
7  // State
8  Bag⟨StaticEvent⟩ inferred := ∅;
9
10 class Region { int start; int end; }
11
12 void inferEvents() {
13   foreach (List⟨LogEntry⟩ log in logs) {
14     foreach (Region r in computeRegions(log)) {
15       boolean addedFinished := addFinishedEvents(r, log);
16       if (addedFinished) { addStartingEvent(r, log); }
17     }
18   }
19   filterOutLowConfidence(confidenceThreshold);
20   events := inferred.toSet();
21 }
22 Set⟨Region⟩ computeRegions(List⟨LogEntry⟩ log) {
23   return { new Region(i, j) | log(i).type = SLEEP_CALL ∧
24       j := min{ k | log(i).tid = log(k).tid ∧
25       log(k).type = SLEEP_RETURN } }
26 }
27 boolean addFinishedEvents(Region r, List⟨LogEntry⟩ log) {
28   boolean result := false;
29   foreach (ThreadID t in { log(i).tid | i ∈ r } − { log(r.start).tid }) {
30     Set⟨int⟩ relevant := { i ∈ r | log(i).tid = t ∧
31         log(i).type ∈ { SLEEP_CALL, BLOCK_CALL, TH_END } ∧
32           ¬(∃ j ∈ r | log(j).tid = t ∧
33             log(j).type ∈ { SLEEP_RETURN, BLOCK_RETURN }) }
34     if (relevant.size() ≠ 1) continue;
35     int starting := max{ j < relevant | log(j).tid = t ∧
36           log(j).type = OTHER_RETURN }
37     addEvent(relevant, "finished", starting);
38     result := true;
39   }
40   return result;
41 }
42 void addStartingEvent(Region r, List⟨LogEntry⟩ log) {
43   int finished := min{ j > r.start | log(j).tid = log(r.start).tid ∧
44       log(j).type ∈ { OTHER_CALL, TH_END } }
45   addEvent(finished, "starting", finished);
46 }
47 void addEvent(int location, String namePrefix, int suffixIdx) {
48     StmtID sid = log(location).sid;
49     events ∪= new StaticEvent(sid, namePrefix +
50         log(suffixIdx).info + sid);
51 }
```

Figure 4: Events-Inference Algorithm

```
   TH_START, main, 333
   SLEEP_CALL, main, 334
   OTHER_CALL(add), addThread, 326
   // calls/returns if add is a helper method
   OTHER_RETURN(add), addThread, 326
   SLEEP_CALL, addThread, 328 // relevant in 0
   SLEEP_RETURN, main, 334
   OTHER_CALL(take), main, 336
   OTHER_RETURN(take), main, 336
   OTHER_CALL(take), main, 339
   BLOCK_CALL, main, 155 // relevant in 1
   SLEEP_RETURN, addThread, 328
   OTHER_CALL(add), addThread, 330
   OTHER_RETURN(add), addThread, 330
   BLOCK_RETURN, main, 155
   OTHER_RETURN(take), main, 339
```

Region 0

Region 1

Figure 5: Snippet from a Log for Inferring Events

4.2.3 Inferring Schedules

FSE'11, ACM, pp 223-233. 2011

Figure 6 presents the algorithm to infer an IMUnit schedule for a sleep-based multithreaded unit test that already contains IMUnit event annotations. These annotations can be automatically produced by our event inference or manually provided by the user. The input to the algorithm is a set of logs obtained from the passing executions of the sleep-based test. Figure 7 shows a snippet from one such log for our running example sleep-based test shown in Figure 1(a). The input also contains a `confidenceThreshold` which will be described later. The output is an inferred schedule, i.e., a set of orderings that encodes the intended schedule for the test. The main part of the algorithm is the `addSleepInducedOrderings` procedure. It captures the intuition that a thread normally executes a sleep to wait for the other active threads to perform events. Recall line 13 from our example in Figure 1(a) where the `main` thread sleeps to wait for the thread `addThread` to perform an `add` operation, and line 9 where `addThread` sleeps to wait for the `main` thread to first perform one `take` operation and then block while performing the second `take` operation.

For each log, the procedure scans for `SLEEP_RETURN` entries (line 31). As shown in Figure 7, the log for our example contains two `SLEEP_RETURN` entries, one each in the `main` thread and `addThread`. For each `SLEEP_RETURN` that is found, the procedure does the following:

1) Retrieves the next `EVENT` entry for the same thread (line 33). This event will be used as the `after` event in `Ordering`s induced by the `SLEEP_RETURN`. In the example log, the two `after` events are `startingTake1` for the first `SLEEP_RETURN` and `startingAdd2` for the second `SLEEP_RETURN`.

2) Computes the other threads that were *active* between the `SLEEP_RETURN` and the `after` event (line 34). In the example, for the first `SLEEP_RETURN`, the only other active thread is `addThread` and for the second `SLEEP_RETURN`, the only other active thread is `main`.

3) Finds for each active thread the last `EVENT` entry that is before the `after` event. This event will be the `before` event in the `Ordering` induced by the `SLEEP_RETURN` with the corresponding active thread (line 38). Note that this `before` event on another thread can be even *before* the `SLEEP_RETURN`. Effectively, this event is the *current* last entry and not the last entry at the time of the sleep. In the example, the two `before` events are `finishedAdd1` and `startingTake2` for the first and second `SLEEP_RETURN`s, respectively.

4) Creates an `Ordering` for each `before` and `after` event pair and inserts it into the `inferred` bag. If a `before` event is followed immediately by a `BLOCK_CALL` (within entries for the same thread), a `BlockingOrdering` is created; otherwise, a `NonBlockingOrdering` is created (line 41). In the example, since `startingTake2` is followed by a `BLOCK_CALL`, the ordering between `startingTake2` and `startingAdd2` will be a `BlockingOrdering`, while the other ordering between `finishedAdd1` and `startingTake1` will be a `NonBlockingOrdering`.

Before the `addSleepInducedOrderings` procedure is invoked, each `log` is modified by the `preprocessLogs` procedure. This procedure looks for `SLEEP_RETURN` entries followed immediately by `TH_START` entries for the same thread. For every such instance, it swaps the `SLEEP_RETURN` and `TH_START` entries and sets the `tid` of the `SLEEP_RETURN` entry to be the ID of the thread that is *started* by the `TH_START` event. The intuition is that a `SLEEP_RETURN` followed by a `TH_START` signifies that the *started* thread, rather than the starting thread perform-

```
1  class Event { String eventName; ThreadID tid; }
2  abstract class Ordering { Event before; Event after; }
3  class NonBlockingOrdering extends Ordering {};
4  class BlockingOrdering extends Ordering {};
5  // Input
6  Set⟨List⟨LogEntry⟩⟩ logs;
7  float confidenceThreshold;
8  // Output
9  Set⟨Ordering⟩ orderings;
10 // State
11 Bag⟨Ordering⟩ inferred := ∅;
12
13 void inferSchedules() {
14   foreach (List⟨LogEntry⟩ log in logs) {
15     List⟨LogEntry⟩ preprocessed := preprocessLog(log);
16     addSleepInducedOrderings(preprocessed);
17   }
18   minimize();
19 }
20 List⟨LogEntry⟩ preprocessLog(List⟨LogEntry⟩ log) {
21   List⟨LogEntry⟩ result := log.clone();
22   foreach ({ i | log(i).type = SLEEP_RETURN }) {
23     int j := min{j > i | log(j).tid = log(i).tid };
24     if (log(j).type = TH_START) {
25       result(j) := new LogEntry(SLEEP_RETURN, _, log(j).info);
26       result(i) := log(j);
27     } }
28   return result;
29 }
30 void addSleepInducedOrderings(List⟨LogEntry⟩ log) {
31   foreach ({ i ∈ log.indexes() | log(i).type = SLEEP_RETURN }) {
32     ThreadID t := log(i).tid;
33     int j := min{ n > i | log(n).tid = t ∧ log(n).type = EVENT };
34     Set⟨ThreadID⟩ active := { t' | ( ∃ n < j |
35                   log(n).tid = t' ∧ log(n).type = EVENT ) ∧
36                   ( ∃ n > i | log(n).tid = t' ∧ log(n).type = TH_END
                       ) };
37     foreach (ThreadID t' in active − { t }) {
38       int j' := max{ n < j | log(n).tid = t' ∧ log(n).type = EVENT };
39       Event before := new Event(log(j').info, t');
40       Event after := new Event(log(j).info, t);
41       if (log(min{ n > j' | log(n).tid = t' }).type ≠ BLOCK_CALL) {
42         inferred ∪= new NonblockingOrdering(before, after);
43       } else { // before.type = BLOCK_CALL
44         inferred ∪= new BlockingOrdering(before, after);
45       } } } }
46 void minimize(List⟨LogEntry⟩ log) {
47   Set⟨Ordering⟩ graph := inferred.toSet() ∪ computeSeqOrderings(log);
48   removeCyclicOrderings(graph);
49   performTransitiveReduction(graph);
50   inferred.onlyRetainOrderingsIn(graph);
51   filterOutLowConfidence(confidenceThreshold);
52   orderings := inferred.toSet();
53 }
54 void Set⟨Ordering⟩ computeSeqOrderings(List⟨LogEntry⟩ log) {
55   return { new NonblockingOrdering(log(i), log(j)) |
56           i < j ∧ log(i).tid = log(j).tid ∧
57           log(i).type = log(j).type = EVENT ∧
58           ¬(∃ k | i < k < j ∧ log(j).tid = log(k).tid
59           ∧ log(k).type = EVENT) };
60 }
```

Figure 6: Schedule-Inference Algorithm

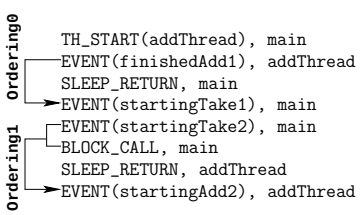

Figure 7: Snippet from a Log for Inferring Schedules

ing the `TH_START`, should wait for the other active threads to perform events, many of the sleep-based tests that we migrated included instances of this pattern. Effectively, this swap makes it appear as if the sleep was at the beginning of the `run` method for the started thread, although the sleep was actually before the `start` method.

After each log is processed by the `preprocessLogs` and `addSleepInducedOrderings` procedures, the `inferred` bag is populated with all the inferred orderings. However, the inferred orderings may contain cycles (e.g., `a->b` and `b->a`) and transitively redundant orderings (e.g., `a->b`, `b->c`, and `a->c`, where the last ordering is redundant). The `minimize` procedure removes such orderings. It first creates an ordering `graph` by combining the edges from the `inferred` orderings with the edges implied by the sequential orderings of events within each thread (the latter edges being computed by the `computeSeqOrderings` procedure). It then removes all the edges of the `graph` that participate in cycles. It finally performs a transitive reduction on the acyclic `graph` and updates the `inferred` bag by removing all orderings not included in the reduced `graph`. We use an open-source implementation [12] of the transitive reduction algorithm introduced by Aho et al. [1]. Since the transitive reduction is performed on an acyclic `graph`, we can use a simpler case of the general algorithm.

The last step of the `minimize` procedure is to remove the orderings that were inferred with low confidence. Recall that the input to our inference is a set of logs from several (passing) runs of the test being migrated. The confidence of an inferred ordering is the ratio of the count of that ordering in the `inferred` bag and the number of logs/runs. For example, an ordering may be inferred in only 60% of runs, say 3 out of 5. The `confidenceThreshold` defines the lowest acceptable confidence. All inferred orderings with confidence lower than the specified threshold are discarded.

4.2.4 Eclipse Plugin

We have developed a refactoring plugin for Eclipse to enable automated migration of existing sleep-based unit tests into event-based IMUnit tests. The plugin is implemented using the generic refactoring API provided by Eclipse. The refactoring automates the most important steps required to migrate a sleep-based test into an IMUnit test: introduction of events and schedule (using inference techniques) and checking of the introduced schedule. The refactoring can also help the user name the threads in the test.

4.3 Multiple Schedules

As mentioned in Step 6 of Section 4.1, after converting sleep-based tests to event-based IMUnit tests, developers can merge several similar tests with different schedules into one test with multiple IMUnit schedules. Recall our example sleep-based test from Figure 1(a). Its intended schedule is an `add` followed by a non-blocking `take` and a blocking `take` followed by another `add`. Suppose that the same test class contained another sleep-based test whose indented schedule is an `add` followed by a non-blocking `take` and another `add` followed by another non-blocking `take`. Although these two sleep-based tests would be almost identical (with the sleep at line 9 moved to before line 16), they cannot share the common code without using additional conditional statements to enable the appropriate sleeps during execution. In contrast, after both tests are migrated to IMUnit tests, they can be easily replaced by just one new test. This new test would have the same code as in Figure 1(a), with two added annotations: (1) `@Event("finishedAdd2")` added after the `add(2)` call, and (2) `@Schedule("finishedAdd1->startingTake1, finishedAdd2->startingTake2")` added before the test method.

5. ENFORCING & CHECKING

We now describe the IMUnit Runner, our tool for enforcing/checking schedules for IMUnit tests. It is implemented as a custom test runner for the JUnit testing framework. It executes each test for each IMUnit schedule and has two operation modes. In the *active mode*, it controls the thread scheduler to enforce an execution of the test to satisfy the given schedule. Note that this mode avoids the main problem of sleep-based tests, that of false positives and negatives due to the execution of unintended schedules. In the *passive mode*, our tool observes and checks the execution provided by the JVM against the given schedule.

Our runner is implemented using JavaMOP [10, 24], a high-performance runtime monitoring framework for Java. JavaMOP is generic in the property specification formalism and provides several such formalisms as *logic plugins*, including past-time linear temporal logic (PTLTL). Although our schedule language is a semantic fragment of PTLTL (Section 3), enforcing PTLTL specifications in their full generality on multithreaded programs is rather expensive.

Instead, we have developed a custom JavaMOP logic plugin for our current IMUnit schedule language from Figure 2. Since JavaMOP takes care of all the low-level instrumentation and monitor integration details (after a straightforward mapping of IMUnit events into JavaMOP events), we here only briefly discuss our new JavaMOP logic plugin. It takes as input an IMUnit schedule and generates as output a monitor written in pseudo-code; a *Java shell* for this language then turns the monitor into AspectJ code [19], which is further woven into the test program. In the active mode, the resulting monitor enforces the schedule by blocking the violating thread until all the conditions from the schedule are satisfied. In the passive mode, it simply prints an error when its corresponding schedule is violated.

A generated monitor for an IMUnit schedule observes the defined events. When an event e occurs, the monitor checks all the conditions that the event should satisfy according to the schedule, i.e., a Boolean combination of basic and block events (Figure 2). The status of each basic event is maintained by a Boolean variable which is true iff the event occurred in the past. The status of a block event is checked as a conjunction of this variable and its thread's blocked state. In the active mode, the thread of e will be blocked until this Boolean expression becomes true. If the condition contains any block event, periodic polling is used for checking thread states. Thus, IMUnit pauses threads only if their events are getting out of order for the schedule. Note that the user may have specified an infeasible schedule, which can cause a deadlock where all threads are paused. Our runner includes a low-overhead runtime deadlock detection that detects and reports deadlocks.

As an example, Figure 8 shows the active-mode monitor generated for the schedule in Figure 1(c). When events `finishedAdd1` and `startingTake2` occur, the monitor just sets the corresponding Boolean variables, as there is no condition for those events. For event `startingTake1`, it checks if there was an event `finishedAdd1` in the past by checking the vari-

```
1  switch (event) {
2  case finishedAdd1:
3    notifyAll();
4  case startingTake2:
5    thread_startingTake2 = currentThread();
6    occurred_startingTake2 = true; notifyAll();
7  case startingTake1:
8    while (!occurred_finishedAdd1)
9      wait();
10   occurred_startingTake1 = true; notifyAll();
11 case startingAdd2:
12   while (!(occurred_startingTake2 &&
13            isBlocked(thread_startingTake2)))
14     wait();
15   occurred_startingAdd2 = true; notifyAll(); }
```

FSE'11, ACM, pp 223-233. 2011

Figure 8: Monitor for the schedule in Figure 1(c)

able `occurred_finishedAdd1`; if not, the thread will be blocked until `finishedAdd1` occurs. For event `startingAdd2`, in addition to checking the Boolean variable for `startingTake2`, it also checks whether the thread of the event `startingTake2` is blocked; if not, the thread of the event `startingAdd2` will be blocked until both conditions are satisfied.

6. EVALUATION

To evaluate the IMUnit contributions—schedule language, automated migration, and schedule execution—we analyzed over 200 sleep-based tests from several open-source projects. Table 1 lists the projects and the number of sleep-based tests that we manually migrated to IMUnit. We first describe our experience with the IMUnit language. We then present results of our inference techniques for migration. We finally discuss the test running time.

6.1 Schedule Language

It is hard to quantitatively evaluate and compare languages, be it implementation or specification languages, including languages for specifying schedules. One metric we use is how *expressive* the language is, i.e., how many sleep-based tests can be expressed in IMUnit such that *sleeps can be removed altogether*. Note that IMUnit conceptually subsumes sleeps: sleeps and IMUnit events/schedules can coexist in the same test, and developers just need to make sleeps long enough to account for the IMUnit schedule enforcement. While every sleep-based test is trivially an IMUnit test, we are interested only in those tests where IMUnit allows removing sleeps altogether.

We were able to remove sleeps from 198 tests, in fact all sleeps from all but 4 tests. While the current version of IMUnit is highly expressive, we have to point out that we refined the IMUnit language based on the experience with migrating the sleep-based tests. When we encountered a case that could not be expressed in IMUnit, we considered how frequent the case is, and how much IMUnit would need to change to support it. For example, blocking events are very frequent, and supporting them required a minimal syntactic extension (adding events with square brackets) to the initial version of our language. However, some cases would require bigger changes but are not frequent enough to justify them. The primary example is events in a loop. IMUnit currently does not support the occurrence of an event more than once in a trace. We did find 4 tests that would require multiple event occurrences, but changing the language to support them (e.g., adding event counters or loop indices to events) would add a layer of complexity that is not justified by the small number of cases. However, as we apply IMUnit to more projects, and gain more experience, we expect that the language could grow in the future.

Subject	Tests	Events	Orderings
Collections [4]	18	51	32
JBoss-Cache [18]	27	105	47
Lucene [6]	2	3	4
Mina [7]	1	2	1
Pool [5]	2	8	3
Sysunit [11]	9	33	34
JSR-166 TCK [17]	139	577	277
Σ	198	779	398

Table 1: Subject Programs Statistics

6.2 Inference of Events and Schedules

To measure the effectiveness of our migration tool in inferring events/schedules, we calculated precision and recall of automatically inferred events/schedules with respect to the manually written events/schedules (i.e., the manual translations from sleep-based schedules). Calculating precision and recall requires comparing the automatically inferred and manually written events/schedules. For event inference, the input is a sleep-based test, and the output is a set of events. Our current comparison uses only the *source-code location* (line number) of the static events and not their name. For schedule inference, the input is a sleep-based test *with manually written (not automatically inferred) events*, and the output is a schedule. Our comparison considers all orderings from the automatically inferred and manually written schedules; two orderings match only if they have exactly the same *both before and after events* (including their name and type that can be basic or block). We performed the comparisons for all but 14 (discussed below) of our 198 tests. Table 2 shows for each project precision and recall values, averaged over the tests from that project.

Columns two and three show the results for event inference. In most cases, precision and recall are fairly high. We inspected the cases with lower precision and identified two causes for it. The first cause is due to our evaluation setup and not the algorithm itself. Namely, our current comparison requires the exact match of source-code locations. If the locations differ, the inferred event counts as a false negative, even if it was only a few lines from the manually written event, and even if those locations are equivalent with respect to the code. In the future, we plan to improve the setup by analyzing the code around the automatically inferred and manually written events to determine if their locations are equivalent. The second reason is that some tests use sleeps that are not relevant for the thread schedule (e.g., JBoss-Cache has such sleeps in the helper threads shared among tests, and Lucene has similar sleeps while interacting with the I/O library). These extra sleeps mislead our inference, which assumes that every sleep is relevant for the schedule and infers events for every sleep.

Columns four and five show the results for schedule inference. The results are even more impressive than for event inference, with precision and recall of over 75% in all cases. We identified two causes for misses. The first cause is that some threads can be independent. The algorithm always forms edges from all threads to the thread that invokes sleep method, but this should not be done for independent threads. In the future, we plan to consider an abstraction similar to regions (Figure 4) as a mechanism to detect inde-

FSE'11, ACM, pp 223-233. 2011

Subject	Inferring Events		Inferring Schedules	
	Precision	Recall	Precision	Recall
Collections	0.75	0.82	0.96	0.97
JBoss-Cache	0.83	0.86	0.87	0.96
Lucene	0.75	1.00	1.00	0.75
Mina	0.22	1.00	1.00	1.00
Pool	0.90	1.00	1.00	1.00
Sysunit	0.76	0.87	0.89	0.89
JSR-166 TCK	0.67	0.74	0.98	0.98
Overall	0.75	0.79	0.96	0.94

Table 2: Precision and Recall for Inference

Subject	Original	CR	TR	LC
Collections	33	0	0	0
JBoss-Cache	39	2	3	0
Lucene	5	0	1	1
Mina	1	0	0	0
Pool	3	0	0	0
Sysunit	39	0	5	0
JSR-166 TCK	306	0	30	1

Table 3: Numbers of Removed Orderings

pendent threads. The second cause is the same as for event inference, namely unnecessary sleeps.

A known issue in information retrieval is that some result sets may be empty, which corresponds to infinite precision and zero recall. For 14 of 198 tests, our inference techniques returned empty sets of events/schedules because these tests do not use sleeps to control schedules. Instead, these tests use `while (condition) { Thread.sleep/yield }` or `wait/notify` or `CountDownLatch` and other concurrent constructs to control schedules. We excluded these 14 tests from the evaluation of our inference techniques.

Our inference algorithms use `confidenceThreshold` to select some of the events/schedules, with the default value of 0.5 (for Table 2). We performed a set of experiments to evaluate how sensitive our inference is to the value of `confidenceThreshold`. We found that the results are quite stable. For example, for schedule inference, when changing the value from 0.5 to 0.1, only for Lucene the precision drops from 1 to 0.75. When changing the value from 0.5 to 0.9, only for JBoss-Cache the precision and recall drop from 0.87 and 0.96 to 0.86 and 0.93, respectively. For all other cases, everything else is inferred exactly the same for the values 0.1 and 0.9 as for the default value 0.5.

The other input to our inference algorithms is the set of logs obtained from passing runs of the legacy tests. By default, we collect 5 passing logs for each test (for Table 2). Different runs of the legacy test can produce different logs that can in turn result in different sets of events/schedules being inferred. Therefore, depending on the number of logs, inferred events/schedules could differ. So we evaluated how sensitive our inference is to the number of logs. We found that the logs are quite stable, and almost identical results were obtained for 1, 5, and 10 logs. For instance, going from 5 to 10 logs only the recall for JBoss-Cache drops from 0.96 to 0.94, and everything else remains the same.

Lastly, our schedule-inference algorithm runs a minimization phase after processing all the logs. Table 3 summarizes the results of this phase. It tabulates, for each project, the number of schedule orderings originally inferred before minimization (Original) and the numbers of orderings removed by cycles removal (CR), by transitive reduction (TR), and due to low confidence (LC). As it can be seen, the minimization phase does not remove many orderings. However, it is important to remove the orderings it does remove. For example, without removing the cycle for JBoss-Cache, not only would inference have a lower precision but it would also produce a schedule that is unrealizable.

Subject	Original [s]	IMUnit [s]		Speedup	
		DDD	DDE	DDD	DDE
Collections	4.96	1.06	1.67	4.68	2.97
JBoss-Cache	65.58	31.25	31.76	2.10	2.06
Lucene	11.02	3.57	6.12	3.09	1.80
Mina	0.26	0.17	0.20	1.53	1.30
Pool	1.43	1.04	1.04	1.38	1.38
Sysunit	17.67	0.35	0.45	50.49	39.27
JSR-166 TCK	15.20	9.56	9.56	1.59	1.59
GeometricMean				3.39	2.76

Table 4: Test execution time. DDD - deadlock detection disabled; DDE - deadlock detection enabled

6.3 Performance

Table 4 shows the execution times of the 198 original, sleep-based tests and the corresponding IMUnit tests (for IMUnit, with deadlock detection both disabled and enabled). We ran the experiments on an Intel i7 2.67GHz laptop with 4GB memory, using Sun JVM 1.6.0_06. Our goal for IMUnit is to improve readability, modularity, and reliability of multithreaded unit tests, and we did not expect IMUnit execution to be faster than sleep-based execution. In fact, one could even expect IMUnit to be slower because of the additional code introduced by the instrumentation and the cost of controlling schedules. It came as a surprise that IMUnit is faster than sleep-based tests, on average 3.39x. Even with deadlock detection enabled, IMUnit was on average 2.76x faster. This result is with the sleep durations that the original tests had in the code.

We also compared the running time of IMUnit with MultithreadedTC on a common subset of JSR-166 TCK tests that the MultithreadedTC authors translated from sleep-based to tick-based [25]. For these 129 tests, MultithreadedTC was 1.36x faster than IMUnit. Although MultithreadedTC is somewhat faster, it has a much higher migration cost, and in our view, produces test code that is harder to understand and modify than the IMUnit test code. Moreover, we were surprised to notice that running MultithreadedTC on these tests, translated by the MultithreadedTC authors, can result in some failures (albeit with a low probability), which means that these MultithreadedTC tests can be unreliable and lead to false positives in test runs.

7. RELATED WORK

Three areas of work are related to IMUnit: (1) unit testing of multithreaded code, (2) enforcement of schedules, and (3) automated inference of specifications. We briefly discuss each of them. (1) ConAn [22, 23] and MultithreadedTC [26] introduce unit testing frameworks that allow developers to specify schedules to be used during the execution of multithreaded unit tests. However, the schedules in both frameworks are specified relative to a global clock (real time for ConAn and logic time for MultithreadedTC), which makes it difficult to reason about the schedules. Also, neither framework supports automated migration of sleep-

based tests. ConcJUnit [27] extends JUnit to propagate exceptions raised by child threads up to the main thread and also checks whether all child threads have finished at the end of a test method. ThreadControl [13] proposes a tool to ensure that assertions are performed without interference from other threads. (2) There has been some previous work on using formally specified sequencing constraints to verify multithreaded programs [28]. The specifications are over sync events with LTL-like constraints, and the verification ensures that the implementation is faithful to the specification. In contrast, IMUnit schedule specifications are used to enforce ordering between user-specified events while the system is tested. Carver and Tai [9] use deterministic replay for concurrent programs. LEAP [14] is a more recent system using a similar record-and-replay approach to reproduce bugs. In comparison, our enforcement and checking mechanism targets ensuring the user-specified schedule rather than replaying a previously observed execution. (3) Work on automated mining of specifications for programs [2, 3, 8, 21] is related to our automated inference of events and schedules. However, most existing work focuses on mining API usage patterns/rules in a single threaded scenario, while our techniques mine the intention of sleep-based tests i.e. interesting events and event orderings across multiple threads.

8. CONCLUSIONS

Current approaches for unit testing of multithreaded code have issues with readability, modularity, reliability, and/or migration cost. We presented IMUnit, a novel approach that addresses these issues. IMUnit includes a new language that makes tests more readable and modular as it allows explicitly specifying schedules on the events during test execution. We described inference techniques and a tool that can help in migrating sleep-based tests to IMUnit. We also described a tool that can reliably execute the specified schedule to avoid false positives/negatives. The promising results with IMUnit encourage us to further explore this approach, e.g., for automatic generation of multithreaded tests (both test code and schedules) only from the code under test, or for regression testing of code with IMUnit schedules [16].

Acknowledgements

We would like to thank Feng Chen, Steven Lauterburg and Traian Şerbănuţă for initial discussion on this work. Also, we would like to thank the participants of the IWMSE 2010 workshop for useful feedback. This work is partially supported by the National Science Foundation under Grant Nos. CCF-1012759, CNS-0958199, CCF-0916893, CCF-0746856, and CNS-0720512, by Intel and Microsoft via the Universal Parallel Computing Research Center (UPCRC), by NASA contract NNL08AA23C, by an NSA grant, by a UIUC Campus Research Board Award, and by a Samsung SAIT grant.

9. REFERENCES

[1] A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1972.

[2] R. Alur, P. Cerný, M. Parthasarathy, and W. Nam. Synthesis of interface specifications for Java classes. In *POPL*, 2005.

[3] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL*, 2002.

[4] Apache Software Foundation. Apache Commons Collections. `http://commons.apache.org/collections/`.

[5] Apache Software Foundation. Apache Commons Pool. `http://commons.apache.org/pool/`.

[6] Apache Software Foundation. Apache Lucene. `http://lucene.apache.org/`.

[7] Apache Software Foundation. Apache MINA. `http://mina.apache.org/`.

[8] J. Burnim and K. Sen. DETERMIN: Inferring likely deterministic specifications of multithreaded programs. In *ICSE*, 2010.

[9] R. H. Carver and K. Tai. Replay and testing for concurrent programs. *IEEE Software*, 1991.

[10] F. Chen and G. Roşu. Mop: An efficient and generic runtime verification framework. In *OOPSLA*, 2007.

[11] Codehaus. Sysunit. `http://docs.codehaus.org/display/SYSUNIT/Home`.

[12] S. Cotton. graphlib. `http://www-verimag.imag.fr/~cotton/`.

[13] A. Dantas, F. V. Brasileiro, and W. Cirne. Improving automated testing of multi-threaded software. In *ICST*, 2008.

[14] J. Huang, P. Liu, and C. Zhang. LEAP: Lightweight deterministic multi-processor replay of concurrent Java programs. In *FSE*, 2010.

[15] V. Jagannath, M. Gligoric, D. Jin, G. Rosu, and D. Marinov. IMUnit: Improved multithreaded unit testing (position statement). In *IWMSE*, 2010.

[16] V. Jagannath, Q. Luo, and D. Marinov. Change-aware preemption prioritization. In *ISSTA*, 2011.

[17] Java Community Process. JSR 166: Concurrency utilities. `http://g.oswego.edu/dl/concurrency-interest/`.

[18] JBoss Community. JBoss Cache. `http://www.jboss.org/jbosscache`.

[19] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP*, 2001.

[20] Lassi Project. Sleep testcase. `http://tinyurl.com/4hk9zdr`.

[21] C. Lee, F. Chen, and G. Roşu. Mining parametric specifications. In *ICSE*, 2011.

[22] B. Long, D. Hoffman, and P. A. Strooper. A concurrency test tool for Java monitors. In *ASE*, 2001.

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

[24] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the MOP runtime verification framework. *Springer STTT*, 2011.

[25] W. Pugh and N. Ayewah. MultithreadedTC - A framework for testing concurrent Java applications. `http://code.google.com/p/multithreadedtc/`.

[26] W. Pugh and N. Ayewah. Unit testing concurrent software. In *ASE*, 2007.

[27] M. Ricken and R. Cartwright. ConcJUnit: Unit testing for concurrent programs. In *PPPJ*, 2009.

[28] K. Tai and R. H. Carver. Use of sequencing constraints for specifying, testing, and debugging concurrent programs. In *ICPADS*, 1994.