Taming the Concurrency: Controlling Concurrent Behavior while Testing Multithreaded Software

Evgeny Vainer Amiram Yehudai

The Blavatnik School of Computer Science, Tel Aviv University, Tel Aviv, Israel

{zvainer, amiramy}@post.tau.ac.il

Abstract

Developing multithreaded software is an extremely challenging task, even for experienced programmers. The challenge does not end after the code is written. There are other tasks associated with a development process that become exceptionally hard in a multithreaded environment. A good example of this is creating unit tests for concurrent data structures. In addition to the desired test logic, such a test contains plenty of synchronization code that makes it hard to understand and maintain.

In our work we propose a novel approach for specifying and executing schedules for multithreaded tests. It allows explicit specification of desired thread scheduling for some unit test and enforces it during the test execution, giving the developer an ability to construct deterministic and repeatable unit tests. This goal is achieved by combining a few basic tools available in every modern runtime/IDE and does not require dedicated runtime environment, new specification language or code under test modifications.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Debugging aids; D.3.3 [*Language Constructs and Features*]: Concurrent programming structures

General Terms Algorithms, Languages

Keywords concurrent code, unit test, multithreaded, thread scheduling, bug reproduction

1. Introduction

In recent years multicore hardware has became a commodity in end user products. In order to support such a change and to guarantee better performance and hardware utilization, more and more application developers had to switch to using multiple threads in their code. Developing such a code introduces new challenges that the developer has to cope with, like multiple threads synchronization or data races, making concurrent applications much more difficult and complicated to create, even for experienced developers [1, 2]. Fortunately, during the years, a lot of tools supporting development process has been created - starting with new synchronization primitives and concurrent data structures and till frameworks that fully isolate all the multithreaded work from the developer.

Another challenge the developer has to face while creating concurrent application is its testing and validation. While testing “traditional” single threaded application, the tester is usually able to reproduce the bug by providing the application some constant set of input parameters. This capability allows him, for example, to create a test (or unit test) that demonstrates some buggy behavior and later on use it to validate that the bug was fixed. Unfortunately, such a useful property of the bugs disappear when switching to multithreaded code. In fact, the result of some multithreaded code strongly depends on the context switches that happened during the run, while the developer has almost no ability to control or even predict them [3, 4]. This kind of “non determinism” during the tests run makes concurrent code very hard to check - some test may always pass on developer’s machine or team’s test server but always fail in end user’s environment.

To overcome this problem, the unit tests developers try to force context switches in the critical code regions or to delay some code block execution until the other code block execution ends. These goals are usually achieved by adding additional operations (like Sleep or Wait/Notify) to the test logic, thus making a test more complicated. This approach creates additional problems. The sleep intervals are usually chosen by trial and error method, and there is no guarantee that the next run will pass even if there is no bug. Using Wait/Notify pair instead of Sleep method usually requires modifications in the code under test , since the test scheduling almost always depends on its state (i.e. test code should wait until code under test will enter some state). But even this is often not enough, since in many cases the test failure depends on context switch that should happen in

[Copyright notice will appear here once ’preprint’ option is removed.]

some third party component. In such a case, the developers has no convenient way to reproduce the bug.

This problem is well known, and many papers and tools have tried to simplify concurrent code testing [5, 6]. These papers try to apply very powerful techniques like static and runtime analysis or context switch enumeration in order to decide whether or not some concurrent code is buggy. Although these techniques are very powerful, the problem the authors address is very complex. As a result, none of these works can propose a complete solution. There are many interesting and promising results (we mention some of them in the Related Works section), but more work is required. The authors of these techniques have to overcome such challenging problems like scale, precision rates (both for false positives and false negatives) and extend their methods to the whole set of synchronization primitives existing in modern languages.

In this work, we propose another approach to the given problem. Instead of solving a very general question whether a given code is correct, we want to give the developers an ability to control the thread scheduling during the test run. In other words, if the success or failure of the test depends on the context switches that occur during the test run, then include the desired schedule as part of the test set up. To demonstrate and evaluate our ideas we implemented a framework called *Interleaving* using the Java programming language. Our framework allows the developers:

- to introduce context switches in any arbitrary place in the code, including code under test and third party libraries
- to delay some code block execution until some other code reaches the desired state
- to reproduce buggy behavior in a deterministic way
- to separate all scheduling logic from the test's functional logic

and much more. These capabilities are achieved by combining together a few simple tools most of the developers are familiar with, so that there is no need for code under test modifications, special runtime or a new language to define the schedule. In addition, our work is based on ideas and tools that exist in every modern platform and IDE and it has no strict dependences on JRE, so a similar framework could be easily implemented for other development platforms.

The rest of the paper is organized as follows:

- chapter two gives more detailed description of our idea, including some implementation details
- chapter three provides some examples of usage of *Interleaving* framework in order to reproduce bugs in concurrent code
- chapter four reviews some other works in this area
- chapter five concludes and provides some ideas for future research

2. Solution

We now describe the core idea and the implementation details of the *Interleaving* framework.

2.1 Idea

To achieve such a challenging goals we would like to define a new concept we call Gate. For now, it is an abstract concept and its implementation in Java environment will be discussed later in this paper.

DEFINITION 1. *Gate*
$\mathcal{G} = <\mathcal{L}, \mathcal{C}>$
where:
$\mathcal{L}$ *- some location in code which the execution flow could reach during the test run*
$\mathcal{C}$ *- some boolean condition that evaluates to true or false*

The intuition behind this definition is as following - like any gate in the real world that has a location it is placed in and could be opened or closed, our *Interleaving* gate is placed somewhere in the code ($\mathcal{L}$) and could be opened ($\mathcal{C}$ evaluates to true) or closed ($\mathcal{C}$ evaluates to false).

Please note that the latter definition does not limit the position of gate in any way. The gate could be placed anywhere - in the code of the test, in the code under test or even in some third party library. Furthermore, the gate does not have to be bound to a specific line of code. Its position could be defined in some other way like "the first time method X is invoked" or "the fifth iteration of loop P".

The same remark holds for condition $\mathcal{C}$ - it could check anything the one wants. For example, some condition could evaluate to true only if the time of the day is between 8:00AM to 5:00PM while another one will be true only if it rains outside. Of course, such a strange conditions will have no value for real tests and its more likely that the test developers will be interested in conditions like "thread X passed line Y of the code" or "object O is in state S".

While executing the test, the execution flow of some thread $\mathcal{T}$ could reach the location $\mathcal{L}$. At this point the execution of $\mathcal{T}$ is suspended and condition $\mathcal{C}$ is evaluated. The following behavior of $\mathcal{T}$ depends on $\mathcal{C}$'s value:

- $\mathcal{C}$ evaluates to true - thread $\mathcal{T}$ is resumed and continues its execution in a regular way
- $\mathcal{C}$ evaluates to false - thread $\mathcal{T}$ remains suspended and will be resumed only after the value of $\mathcal{C}$ changes to true

For now, we are not interested in the mechanism used to notify the runtime about the changes in condition's state. Let us just assume that such a mechanism exists and that thread $\mathcal{T}$ will be resumed as soon as $\mathcal{C}$'s value will change to true.

Now assume that the unit test developer has an easy and convenient way to define the gates (both location and condition), to combine them into sets and to bind these sets to a specified test. Such a powerful tool will allow the developer to enforce any thread scheduling he wants. All the

one needs to do is to identify the code blocks that should be executed in a particular order and define the gate before the latter (second) block that will open only after execution of the first block is completed.

To demonstrate this idea let us assume the example in the Java programming language shown in figure 1.

```
1  public class SharedMemoryAccessExample {
2    int multiplier = −1;
3    int result = 0;
4
5    class Worker1 extends Thread {
6      public void run() {
7        multiplier = 1;
8      }
9    }
10
11   class Worker2 extends Thread {
12     public void run() {
13       result = multiplier ∗ 10;
14     }
15   }
16
17   public int Calculate() throws Exception {
18     Thread t1 = new Worker1();
19     Thread t2 = new Worker2();
20
21     t1.start();
22     t2.start();
23     t1.join();
24     t2.join();
25     return result;
26   }
27 }
```

Figure 1. Shared Memory Access

In this very simple example each call to the Calculate method will cause the runtime to create two threads, execute them and return the value stored in "result" variable. One could easily note that the value returned by Calculate method depends on the order in which the worker threads were executed. Lets assume that the result of 10 is the expected one, while the result of -10 (which will be returned if line 13 executed before line 07) is a bug.

Even such a simple example of multithreaded class could be very difficult to test. Following the encapsulation principle of OOP all the members of this class are internal, so the unit test code that is external to the class has no access to them. As a result, the only thing the unit test developer could do is to call the Calculate method and to check its return value. It is obvious that the outcome of such a test will depend on thread schedule that took place during the test run. Such a unit test has no value at all since its outcome is not deterministic and the fact that the test passed does not guarantee that the code is bug free. One could try to increase the confidence of the test by calling the Calculate method multiple times during the test and validating all the values returned. Such a test will not be much better than the previous version since it still can result in false negative.

Now assume that the unit test developer is able to define gates as described before. In such a case, the one could define the gate $\mathcal{G} \ =< line\ 07\ ,\ thread\ Worker2\ has\ finished\ its\ execution >$ and bind it to the test. According to the semantics of the gates defined earlier, doing so will cause Worker1 thread to pause its execution just before line 07 of the code and to remain suspended until Worker2 is done. As a result, a call to the Calculate method will return -10, thus failing the test. This thread ordering will be constantly enforced every time the test will be executed, allowing the the developer to reproduce the buggy behavior in a deterministic way.

2.2 Implementation

In order to demonstrate and evaluate our ideas we implemented the above concept using the Java programming language and JRE environment. The resulting framework, we called *Interleaving*, provides an ability to place the gates in arbitrary places in code and to evaluate the conditions when the gate is reached, forcing the behavior defined earlier. The framework could be used together with Eclipce IDE, providing the developers familiar and convenient environment to define and manage their gates. Of course, the concept of a gate defined earlier is very general, so we had to make some relaxations while implementing it.

2.2.1 Condition definition

First of all, in our implementation, we decided to utilize Java programming language for gate conditions definitions. There are several advantages for such a choice:

- Java is a very powerful programming language. It's syntax and semantics has been developed over years by a large and experienced community. Any special language we could create for condition definitions would be less expressive than Java, so we decided not to limit our user by introducing some syntactic limitations.
- JRE contains a lot of frameworks and code libraries that allow the developers to perform very complicated tasks and simplify the development process. All of them could be used while defining gate conditions. Such a reuse simplifies conditions' definitions and allows the developers to create more complicated gates without need to reimplement already existing functionality.
- Since our framework is intended to be used in Java environment, we can assume that all its users are familiar with Java syntax and semantics. Using familiar language to define gate conditions significantly simplifies migration to our framework.

- Using Java for conditions definitions allows us to use JRE in order to evaluate its state, thus saving us the effort to develop our own evaluation engine.
- The fact that conditions are defined using the same programming language that was used while developing the application makes the conditions much more powerful. For example, the code in gate condition can interact with objects defined in application, check their states or even call their methods. All of this is possible because of the same language used to define conditions and application and because of the same runtime used to execute them.

Using Java for conditions definitions limits the power of gates, with respect to definition given in section 2.1. Nevertheless, the code under test is created using the same programming language and executed using the same runtime engine as *Interleaving*'s gates' conditions. This observation refines the fact that the gates are at least as powerful as the application itself, making this implementation decision affordable.

2.2.2 Notification mechanism

Another implementation decision we made deals with the gate notification mechanism. As section 2.1 states, if some thread T is suspended on gate $\mathcal{G} =< \mathcal{L}, \mathcal{C} >$, it is resumed immediately when $\mathcal{C}$'s value becomes true. This definition assumes some mechanism that observes the value of the condition all the time and is able to resume $\mathcal{T}$ whenever its state changes. Although it is possible to implement such a mechanism, the implementation may be pretty complex and somewhat tricky. Since the purpose of our framework is to demonstrate the ideas and not to provide market ready solution, we decided to simplify this behavior. In the *Interleaving* framework,the implementation of the notification mechanism is part of the condition's logic and is the responsibility of the test developer. In other words, when thread $\mathcal{T}$ reaches gate $\mathcal{G} =< \mathcal{L}, \mathcal{C} >$ its state $\mathcal{S}$ is saved somewhere aside and the condition's logic is evaluated. This evaluation should return only after the gate is considered to be opened. After the condition's evaluation ends, the thread's state $\mathcal{S}$ is restored and $\mathcal{T}$ continues its execution in the regular way. This behavior fits the gate's behavior from section 2.1, since thread $\mathcal{T}$ can not continue it's execution until $\mathcal{C}$ is satisfied. Since conditions' logic is defined using Java programming language, it is not a problem to create such a complex conditions.

This relaxation allows test developers to define different and complex conditions whose behavior depends on test requirements. From the observations we made while evaluating our framework, most of the test scheduling could be created using very simple "manual" gates, i.e. the gates whose state has to be changed explicitly. The condition of such a gate contains one expression only - calling for Wait method on some object, while appropriate Notify call has to be made explicitly somewhere else in the code. Please pay attention that such a call could be placed anywhere in the code (even in third party libraries) using fictitious gate whose condition contains Notify call only. Of course, as we mentioned earlier, more complex conditions could be introduced in order to create more complex schedules. Some examples of such conditions will be discussed later on, in the evaluation section of this paper.

2.2.3 Location definition

Now we describe the technique we used to define the location $\mathcal{L}$ for some gate. While developing *Interleaving* framework we searched for a way to represent the location that will satisfy the following requirements:

- The test developer should have fine grained control over gates positions, i.e. one should be able to bind the gate to some line in source code, to some instruction in the binary file or, if possible, to some event that happens during the application execution (like first exception thrown or entering some method)
- The developer should be able to define gates locations using some familiar and convenient technique, so we would like to avoid creating special location definition language or syntax.
- The framework should be able to intercept the execution flow of any thread that reaches the location defined by some gate $\mathcal{G}$ in order to evaluate the condition and suspend thread's execution if needed

Fortunately, we are not the the first who looked for such capabilities. The entity that satisfies these requirements was invented long ago and already exists in all modern development languages and platforms - it is a breakpoint. Indeed, the breakpoint mechanism of JRE allows the developer to put the breakpoint in almost arbitrary place in the code, including third party libraries. It also supports more complex conditions like hits counter, method entry/exit or class load events. Every modern IDE (like Eclipse, for example) provides the developer some convenient, usually graphic, interface for breakpoint definition, fully abstracting from the real syntax used to define breakpoint location/condition. On the other hand, Java Debugging Interface (JDI) libraries supported by the last versions of JVM provide very powerful programmatic interface which allows us to define and remove breakpoints, receive notifications when some breakpoint is hit and execute some custom action when this happens. All of this makes a breakpoint mechanism an ideal solution for defining gates' locations.

2.2.4 Flow control

We now present a short description of the technique the *Interleaving* framework uses in order to intercept and control the flow of test execution.

Each *Interleaving* test is a simple JUnit test while we use JUnit rules to enrich its functionality. At runtime, JUnit will discover that the test has additional rule and will pass the

control to this rule. This is how *Interleaving* comes into the game. The rule code will investigate current test and locate the gates relevant for the test (the way we associate gates to tests is described later in section 2.2.6). Next, a few things will happen.

- First, *Interleaving* will compile the Java code defined in gates' conditions fields, creating separate static method for each one of the gates.
- Next, *Interleaving* uses JDI to set the breakpoints in all of the code locations defined by the gates, and starts a special thread that will handle those breakpoints hits.

After this work is done, the rule returns the flow to JUnit and it continues test execution in regular way.

While running the test, some of the breakpoints might be hit. When this happens, the thread $\mathcal{T}$ that hit the breakpoint is suspended by JVM (all other application threads continue to run) and a notification is sent to the special *Interleaving* thread mentioned earlier. The notification contains all the necessary information required by *Interleaving* in order to identify the gate that was reached and to locate a method containing the gate's condition's code. Next, this method is placed on top of $\mathcal{T}$'s stack and $\mathcal{T}$ is resumed. This technique causes $\mathcal{T}$ to leave the state it was in when it hit the breakpoint, and forces it to execute new code - the code of the condition the developer supplied. Moreover, when the condition's code will return, the stack frame of condition's method will be destroyed and the thread will return to the same state it was in when it was suspended. Since the thread is not suspended anymore it continues the execution of the original test logic as if nothing happened. The only side effect one could notice is a delay caused by the condition's evaluation. This delay, combined with the condition's behavior defined earlier (section 2.2.2), gives us all we need to enforce the desired scheduling.

It is important to notice that all the operations described in the current section are achieved using standard APIs and extension points provided by JUnit, JVM and JDI library. At the cost of some additional code written, we manged to implement these capabilities without modifications made to any of those libraries. As a result, the *Interleaving* framework does not requires special versions of JVM or JRE in order to run the tests. The tests can be executed using the same environment that is used in the production stage.

2.2.5 Putting everything together

Now, we would like to describe how all the things we mentioned earlier are combined together in the *Interleaving* framework. For the demonstration purpose, we assume some developer is required to create a test that reproduces a concurrent bug that exists in code on figure 1. After investigating the bug, the developer concludes that the bug happens only if line 13 of code is executed before line 07, so while creating the test he needs to enforce this schedule.

To do so, he will have to use one of the gates defined in *Interleaving* framework named "SimpleGate". This gate defines a simple API composed of two methods - Wait and Open. Each SimpleGate instance maintains some internal condition that initially evaluates to false (i.e. the gate is considered to be closed) and it remains so until the Open method is called. Calling this method changes the internal condition's value in such a way that from this point it always evaluates to true (i.e. the gate is considered to be open) and there is no way to switch the gate back to the closed state. The Wait method of the gate implements the notification logic we described earlier in section 2.2.2. Whenever this method is called it returns only after the gate's instance it was called on is in opened state. Using this gate the developer can ensure that the code block following the gate's Wait call will be executed only after the code block preceding the gate's Open call is done.

Now, in the test, the developer has to create an instance of SimpleGate and give it some meaningful name, "Worker2Done" for example. Next, he has to locate it somewhere in the code. Following the example, he wants to suspend the execution of the code on line 07 so this is the line where the gate should be located. In order to mark this line as a gate location the developer puts a breakpoint on it. Now, he has to specify the condition associated with the breakpoint. For this purpose we decided to utilize the conditional breakpoint window of Eclipse IDE. So, the developer marks the earlier created breakpoint as conditional one and in the condition window writes the code that calls for Wait method of "Worker2Done" gate. Next, he has to choose the point where the gate is to be opened. Obviously, this point is at line 14 (alternatively, it might be the point where some thread finishes the execution of Worker2.run method). So, the developer puts another breakpoint on line 14 (or method exit breakpoint on Worker2.run method), marks it as conditional and writes the condition that calls for "Worker2Done" gate's Open method. The combination of these two breakpoints creates a deterministic schedule which always enforces the code at line 07 to run after the code at line 13.

Now, all is left is to write the test that calls for Calculate method and to associate the gates created earlier to this specific test. This association could be done using Working Sets. Working set is a convenient way the Eclipse IDE provides for the purpose of grouping some related entities of any kind. All the developer has to do to associate the gates with the test is to create breakpoints working, give it a name of the test and add the breakpoints created earlier to this set. Now, the test can be run using standard JUnit test runner.

While executing the test the breakpoint set on line 07 will be hit by thread $\mathcal{T}_1$. At this point, *Interleaving* will use the technique we described in section 2.2.4 to cause $\mathcal{T}_1$ to execute the breakpoint's condition. This condition contains the call to Wait method of "Worker2Done" gate. As we recall, the Wait method of the gate will return only after the

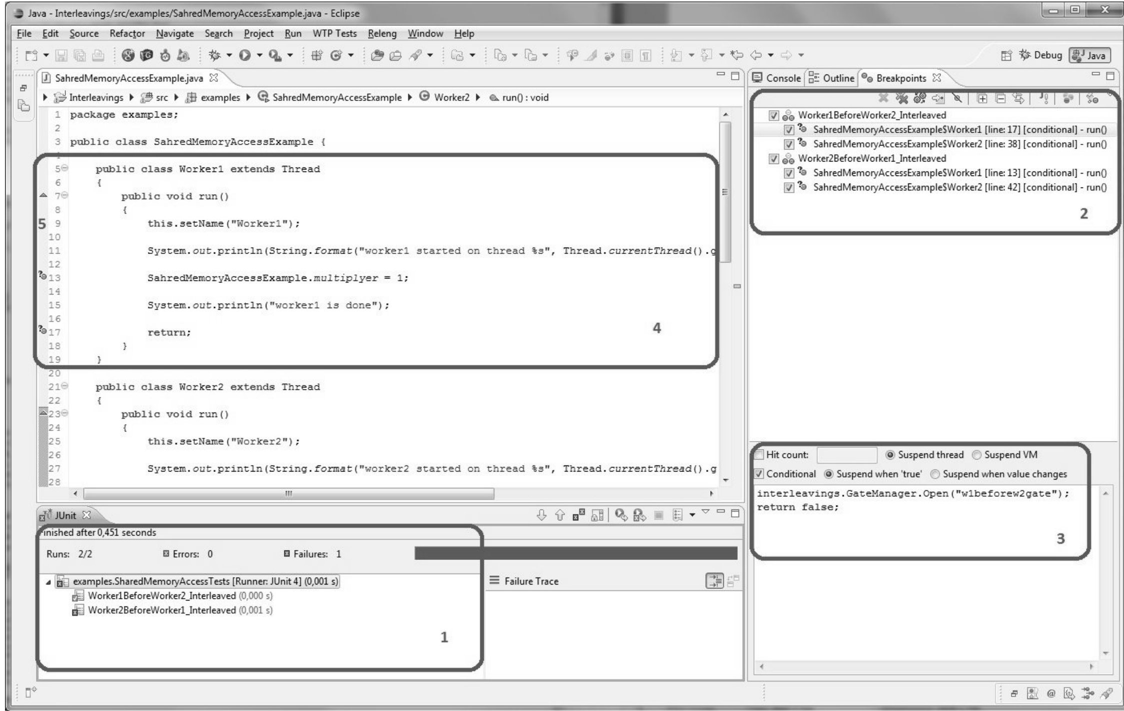

Figure 2. UI for working with Interleaving framework

Open method of the same gate was called. Let us assume that the Open method was not called yet. Therefore, $\mathcal{T}_1$ will remain inside the code of Wait method, while all the other application threads will execute the test logic in the regular way. At some point of time, some other thread $\mathcal{T}_2$ will hit the breakpoint located at line 14, this will cause $\mathcal{T}_2$ to stop its current flow execution and to execute the code defined by the condition of this breakpoint and, as a part of it, to execute the call for Open method of "Worker2Done" gate. This call will return immediately allowing $\mathcal{T}_2$ to return to the test logic. In addition, this call will cause the Wait method of "Worker2Done" gate to return, releasing $\mathcal{T}_1$ and allowing it to return to the test logic execution.

As a conclusion of the flow described, one can notice that adding gates to the test introduced some new ordering constraints on events that occur during the test run. These constraints are as follows (we use the notation of $\mathcal{E}_1 \to \mathcal{E}_2$ to denote that event $\mathcal{E}_1$ occurs before event $\mathcal{E}_2$):

- The code in line 13 is executed ($\mathcal{A}$) before the breakpoint on line 14 is hit ($\mathcal{B}$) ($\mathcal{A} \to \mathcal{B}$)
- "Worker2Done" Open method is called ($\mathcal{C}$) after the breakpoint on line 14 is hit ($\mathcal{B} \to \mathcal{C}$)
- "Worker2Done" Wait method returns ($\mathcal{D}$) after its Open method is called ($\mathcal{C} \to \mathcal{D}$)
- condition evaluation in $\mathcal{T}_1$ ends ($\mathcal{E}$) after "Worker2Done" Wait method returns ($\mathcal{D} \to \mathcal{E}$)
- thread $\mathcal{T}_1$ returns to test logic execution ($\mathcal{F}$) after it completed condition evaluation ($\mathcal{E} \to \mathcal{F}$)
- the breakpoint in line 07 is hit before the code on the same line is executed, as a result $\mathcal{T}_1$ will execute the code in line 07 ($\mathcal{G}$) only after it returns back to the test logic evaluation ($\mathcal{F} \to \mathcal{G}$)

Events sequence above implies that $\mathcal{A} \to \mathcal{G}$ (i.e. the code in line 13 will always be executed before the code in line 07), resulting in consistent bug reproduction, no matter what was the threads scheduling created by JVM/OS for current test execution.

2.2.6 User interface

One of the things we always kept in mind while creating the *Interleaving* framework is its usability. Providing the developers with a tool that is based on concepts they are familiar with significantly reduces the learning curve and eases the migration. Till now we described two examples of such a reuse in our framework:

- using Java programming language in order to describe gates' conditions
- using breakpoint mechanism in order to define gates' locations

Another example of this approach is the user interface of the *Interleaving* framework. All the operations the test developer has to perform while creating and executing interleaved test could be done using standard Eclipse IDE environment and

no additional plugins/windows are required. In our opinion such an integration is very important, since the developer fills comfortable with the environment and can focus on his actual job, instead of spending time on learning new concepts. Figure 2 shows an Eclipse IDE window while creating and executing the test, and describes how different parts of this window come into play while working with the *Interleaving* framework:

1. JUnit test runner window shows the last test run result. Since each *Interleaving* test is also a regular JUnit test, this window displays the results of interleaved tests executed during the run together with the regular tests results.
2. Breakpoints window is used to show the developer all the breakpoints defined for the test run. The breakpoints could be grouped into the Working Sets while each working set corresponds to some interleaved test and contains all the breakpoints relevant to this test. This way the developer can easily manage the gates defined for some test.
3. The gate's condition is shown in the Breakpoint's condition part of Breakpoints window. This window allows the developer to enter the gate's condition using Java programming language providing it with the full set of features he is used to while writing the code (like syntax highlighting or Intellisence). The content of this control shows the condition of the breakpoint selected in the above part of the same window (2), thus providing the developer very convenient view of the gate he works on.
4. The code window could be used to examine the test code/code under test while the gates locations are marked by breakpoints icon on the margins of the window (5), thus providing the developer with an easy way to understand the context the gate is used in.
5. Eclipse IDE allows the developers to define new breakpoints by simply clicking on the margins of the code window. In *Interleaving* terminology this operation defines a new gate whose location is defined by the newly created breakpoint. Afterwards, the gate's condition has to be defined and the breakpoint has to be moved to an appropriate working set. Both of these operations were mentioned earlier and could be performed using breakpoints window (2, 3).

3. Evaluation

The evaluation of our work consists of two parts. First, we looked for different examples of concurrent bugs that are hard to reproduce using standard testing tools and created the gates sets that reproduce the buggy behavior in a consistent way. A few such examples are presented in this chapter. Some of them are real bugs taken from the bugs repositories, while others are synthetic examples we created in order to demonstrate the expressiveness and the power of our approach. The second part of the evaluation is done via the comparison to other works. We show that our framework is at least as powerful as some other tools presented in recent papers, and in some cases more powerful.

3.1 Examples

3.1.1 BlockingQueue

We start with an example of the real unit test for ArrayBlockingQueue class in java.util.concurrent (JSR-166) [7]. This unit test was used by several authors [8, 9] in order to demonstrate their approaches and we continue with this tradition. The code of the test is presented in figure 3. It contains two Thread.sleep calls used by the developer to enforce the desired threads ordering. Although this technique works for most of the runs, there still might be a run in which the threads will be executed in a different order ending up with an incorrect result.

```
1  @Test
2  public void ArrayBlockingQueue_JUnit() throws
   Exception {
3    final ArrayBlockingQueue<Integer> q = new
     ArrayBlockingQueue<Integer>(1);
4
5    Thread addThread = new Thread(
6      new Runnable() {
7        public void run() {
8          q.add(1);
9          Thread.sleep(100);
10         q.add(2);
11       }
12     }
13   );
14
15   addThread.start();
16   Thread.sleep(50);
17
18   Integer taken = q.take();
19   assertTrue(taken == 1 && q.isEmpty());
20   taken = q.take();
21   assertTrue(taken == 2 && q.isEmpty());
22
23   addThread.join();
24 }
```

Figure 3. Unit test for ArrayBlockingQueue class

Figure 4 shows the same test rewritten for *Interleaving*. In addition to the code shown, the test's set up contains one gate placed inside ArrayBlockingQueue.take method, just before it blocks (line 317), which opens the gate named "started_take2".

In contrast to the original unit test, *Interleaving* test always enforces the correct threads ordering leading to consistent results for all of the test runs. In addition, our code is

```
1   @Test
2   @Interleaved
3   public void ArrayBlockingQueue_Interleaved()
    throws Exception {
4     final ArrayBlockingQueue<Integer> q =
5       new ArrayBlockingQueue<Integer>(1);
6
7     Thread addThread = new Thread(
8       new Runnable() {
9         public void run() {
10          q.add(1);
11          interleavings.GateManager.Open(”finished_add1”);
12          interleavings.GateManager.Wait(”started_take2”);
13          q.add(2);
14        }
15      }
16    );
17
18    addThread.start();
19    GateManager.Wait(”finished_add1”);
20
21    Integer taken = q.take();
22    assertTrue(taken == 1 && q.isEmpty());
23    taken = q.take();
24    assertTrue(taken == 2 && q.isEmpty());
25
26    addThread.join();
27  }
```

$\mathcal{G} = <$ ArrayBlockingQueue@317, interleavings.GateManager.Open(”started_take2”); $>$

Figure 4. Unit test and gate for ArrayBlockingQueue class using Interleaving framework

easier to understand since the desired threads scheduling is specified in the code in a clearer, declarative way.

3.1.2 Unspecified Time

The next example is the synthetic one, but it demonstrates a very common scenario. Suppose the tester needs to check the class that performs some long time operation in a different thread. The amount of time the operation could take varies from run to run in hardly predictable way, and depends mostly on the environment the test is run on. In order to create such a test, the developer needs to execute the operation, wait until the job is finished and only then check its status. Figure 5 contains sample code that demonstrates this approach.

In this example, we assume that the operation time is upper bounded by some constant. If it is not the case, the test could "busy wait" until the operation is done. Both methods are not perfect - in the former case the test always takes the maximal possible time even when the operation ends very

```
1   @Test
2   public void LongRunningTask_JUnit() throws
    InterruptedException
3   {
4     Task task = new LongRunningTaskExample().new
      Task();
5
6     task.start();
7
8     Thread.sleep(task.MaxTime);
9     assertTrue(task.IsDone);
10  }
```

Figure 5. Unit test for LongRunningTask class

```
1   @Test
2   @Interleaved
3   public void LongRunningTask_Interleaved() throws
    InterruptedException
4   {
5     Task task = new LongRunningTaskExample().new
      Task();
6
7     task.start();
8
9     //interleavings.GateManager.Wait(”task_done”);
10    assertTrue(task.IsDone);
11  }
```

$\mathcal{G}_1 = <$ LongRunningTask_Interleaved@10, interleavings.GateManager.Wait(”task_done”); $>$
$\mathcal{G}_2 = <$ LongRunningTaskExample@33, interleavings.GateManager.Open(”task_done”); $>$

Figure 6. Unit test and gate for LongRunningTask class using Interleaving framework

fast, while the "busy wait" option consumes unnecessary machine resources.

Figure 6 demonstrates *Interleaving* version of such a test.

It contains two gates:

- $\mathcal{G}_1$ is located just before the assertTrue call. The gate remains closed until the task is done (optionally this gate could be removed from the test set up and replaced by the commented line)
- $\mathcal{G}_2$ is a fictitious gate (as described in section 2.2.2) that opens $\mathcal{G}_1$ and is located on the last line of the checked operation (line 33 of LongRunningTaskExample.java)

Using this technique the test gets the best of the two worlds – it takes as little time as the checked job takes, and the test thread is blocked while the operation performs. In addition,

in case the operation class would not provide us with MaxTime and IsDone members, the developer has no convenient way to check this scenario without using *Interleaving* capabilities.

3.1.3 StringBuffer

Our next example deals with the real bug that exists in StringBuffer class in the current version of JRE [10, 11, 22]. Figure 7 contains the code of the append method of AbstractStringBuilder class which StringBuffer class inherits.

```
1   public AbstractStringBuilder append(StringBuffer sb)
    {
2     if (sb == null)
3       return append("null");
4
5     int len = sb.length();
6     int newCount = count + len;
7     if (newCount > value.length)
8       expandCapacity(newCount);
9
10    sb.getChars(0, len, value, count);
11    count = newCount;
12    return this;
13  }
```

Figure 7. AbstractStringBuilder.append method

This method contains a potential data race while working with the length of the received argument. If the length of sb changes after line 05 was performed, but before line 10 is executed, the method could end up with an exception. One can easily write the test that tries to reproduce this scenario. The example of such a test is shown in the figure 8.

Unfortunately, running this test as is will not reproduce the bug. The reason for this is that the context switch between the worker thread and the test thread should happen in a very specific and very short time window - after the worker thread performed line 05 of append method but before it reaches line 10 of it. This timing window is pretty tight and it is very unlikely for the context switch to happen there in regular runs. The sleeps technique used in the BlockingQueue (section 3.1.1) example also fails to reproduce the bug. Usage of this technique requires one of the sleep calls to be located inside the append method, causing code under test modification which is undesirable in most cases. In order to reproduce the bug we tried to execute this test in some different setups - we executed the test many times inside the loop, we executed several instances of the test simultaneously, we ran it on different machines under different loads - all with no success. The bug appeared in very few runs in a very inconsistent way. The inability to reproduce the bug was noticed by java developers too. The appropriate bug reports mention that the bug "can be reproduced rarely" [10]

```
1   @Test
2   public void LengthRaceCondition() throws Exception
    {
3     final StringBuffer sb1  = new
      StringBuffer("original data");
4     final StringBuffer sb2  = new
      StringBuffer("appended data");
5
6     Thread worker = new Thread(new Runnable() {
7       public void run() {
8           sb1.append(sb2);
9       }
10    });
11
12    worker.start();
13    sb2.setLength(3);
14    worker.join();
15  }
```

Figure 8. Test method for StringBuffer.append

and proposes a test containing two infinite loops (one loop for each thread) [11] in order to reproduce it.

Using *Interleaving* framework we reproduced the buggy behavior in all of the runs by adding only two gates to the test and without changing the code at all. The first gate is located in line 13 of the test and opens after the worker thread passed line 05 of the append method, while the second is located in line 10 of the append method and opens after the test performed line 13 of its code. The formal gates definition is presented in the figure 9.

$\mathcal{G}_1$ =<test@13,
interleavings.GateManager.Wait("afterget");>
$\mathcal{G}_{1fictitious}$ =<append@06,
interleavings.GateManager.Open("afterget");>
$\mathcal{G}_2$ =<append@10,
interleavings.GateManager.Wait("afterset");>
$\mathcal{G}_{2fictitious}$ =<test@14,
interleavings.GateManager.Open("afterset");>

Figure 9. Gates defined for LengthRaceCondition test

Please recall that in our implementation all the gates are manual, i.e. every conceptual gate consists of two parts - the real gate and some fictitious gate that is responsible for opening the real one, as described in section 2.2.2

3.2 Comparison to IMUnit

IMUnit [9] is another framework that provides test developers the ability to define the ordering of some events during test execution. The scheduling definition for this framework consists of two parts:

1. initiation of events of interest somewhere inside the code

2. declarative definition of desired events ordering for the test using some special syntax

The framework controls tests execution and ensures the desired ordering in the following manner – while executing the test, the flow could reach some event of interest (1) defined by the test developer. At this moment, the execution of the thread is suspended until all of the preceding events defined for the test (2) occurred. In addition to the framework, the authors provide a tool that allows relatively easy migration from the "sleep based" tests to IMUnit notation. Using this tool the authors succeed to convert a large amount of concurrent tests to be used with IMUnit, a result that implies the good expressive power of IMUnit notation. We will show that IMUnit events are a special case of *Interleaving* gates and every IMUnit test could be easily rewritten for our framework. One can immediately conclude that:

1. the same approach described in [9] can be used to convert the tests to our notation.
2. the expressive power of *Interleaving* notation is at least as good as that of the IMUnit notation.

Moreover, we will show that the StringBuffer bug mentioned earlier (section 3.1.3) can not be reproduced using IMUnit but can easily be reproduced using *Interleaving*, which implies the greater expressiveness of the *Interleaving* framework.

In order to substatntiate the claims above, we developed a simple algorithm that allows to convert every IMUnit test to *Interleaving* notation. This argorithm is presented on figure 10. We also provide the formal proof that the transformation this algorithm applies to the test code does not affect the test result and preserves the scheduling enforced by the framework. Due to space limitations we will not present this proof here, but only describe the intuition and the general idea. The whole and formal proof will be provided in [24].

1. *let $e_p \to e_s$ be the IMUnit scheduling defined for the test (which means that event e_p should happen before event e_s)*
2. *let L_{e_p} and L_{e_s} to be the lines of code where events e_p and e_s are initiated, respectively*
3. *define gate $\mathcal{G}_{e_p \to e_s} =< \mathcal{L}, \mathcal{C} >$ as follows:*
   3.1 $\mathcal{L} = L_{e_s}$
   3.2 $\mathcal{C} = L_{e_p}\ was\ already\ executed$

Figure 10. Transformation T(unit test) from IMUnit to Interleaving

The intuition behind this transformation is very simple – the execution flow could not reach L_{e_s} before it passes the gate $\mathcal{G}_{e_p}$, but the gate remains closed until the flow reaches L_{e_p}. This implies that L_{e_p} will always be executed before L_{e_s}, enforcing the desired scheduling. In the full proof we also show how to transform other types of scheduling (like $[e_p] \to e_s$) and how to deal with complex scheduling that contains multiple simple scheduling.

Using this simple algorithm one can easily understand why all the tests created with IMUnit notation are a subset of all the tests that could be created using *Interleaving*. The reason for that is that while using IMUnit the events can be initiated from the test code only, which implies that appropriate gates in the transformed test will also be placed inside the code of the test (while *Interleaving* mechanism that uses breakpoints allows the developer to put the gate almost everywhere - inside the code under test or even in third parties code). This limitation significantly reduces the set of bugs IMUnit is capable to reproduce. For example, the StringBuffer bug mentioned above (section 3.1.3) can not be reproduced using IMUnit because of this issue.

Another conclusion that is immediate from the algorithm above is that every IMUnit event could be represented using a gate with very simple and constant condition. This fact also limits the expressive power of the framework. In order to overcome this limitation IMUnit defines its own scheduling specification language that allows the developer to specify more complex condition like $[e_p] \to e_s$. The problem with this approach is that every new condition complicates this language specification and that test developers have to be familiar with this language and all of its capabilities. *Interleaving*, in contrast, does not limit the tester to a predefined set of conditions but allows him to define every logic he desires using the power of the Java programming language - the language the developer is already familiar with. For example, a condition code can check the internal state of current "this" object or even the values of local variables on the stack, things that are impossible while using IMUnit notation.

4. Related Works

The problem of concurrent software testing has been addressed by many researchers. Several approaches have been developed in order to cope with it. O'Callahan and Coi [12] analyze the runtime behavior of the application and apply lockset-based and happens-before techniques in order to identify potential bugs. Eraser [13] tracks application actions and uses collected data to detect possible dataraces. RaceTrack [14] is another tool that utilizes this approach but applies different algorithms in order to identify data races.

Another set of tools interfere with the threads scheduler work, forcing the execution of uncommon executions flows. ConTest [15] introduces new context switches into the program code thus revealing hidden bugs. ConCrash [16] utilizes record and replay technique in order to reproduce buggy runs. AtomFuzzer [17] forces the context switches inside critical region trying to cause atomicity violation. Microsoft Chess [18, 19] reruns each test multiple times while enumerating over different possible thread schedulings. All

the techniques above are fully automated and do not make any use of the knowledge the developer has regarding his code.

ConAn [20, 21] and MultithreadedTC [8] split the application execution timeline to several slots providing the developer the ability to order the code blocks with respect to those slots. IMUnit [9] introduces the concept of events that occur during the test run and enforces events ordering specified for the test. This technique is very close to the one we propose. The comparison of our work to IMUnit was presented earlier in the paper (section 3.2). Park and Sen [22] use the information provided by the developer regarding the buggy state and try to enforce the scheduling that will reach this state.

DataCollider [23] is the only tool we are aware of that makes use of the breakpoints mechanism. It breaks the execution on access to random memory locations and analyzes the program state in order to identify data races. Unlike *Interleaving*, it does not uses this mechanism in order to change the execution flow induced by OS threads scheduler.

5. Conclusions

Testing concurrent applications is a very challenging task. One of the reasons for this is lack of control over threads scheduling during test execution and inability to reproduce the bug as the result of this. We propose a novel technique that allows the unit test developer to specify the desired threads scheduling as part of test setup. This scheduling will be enforced during the test execution consequently reproducing the bug on every test execution.

Our technique utilizes the breakpoints mechanism which allows us to preempt the flow in arbitrary points in the code, including code under test and third party libraries, without the need for code modification. We also allow the test developer to define the decision logic for every particular context switch using Java programming language. All this makes our framework very powerful but still easy to learn and use.

We implemented the prototype of our ideas in the *Interleaving* framework and used it to reproduce some concurrent bugs that are very hard to reproduce using other tools. The framework has good integration with Eclipse IDE and JUnit and does not require dedicated runtime. Although the framework is implemented using Java, the technique itself is not bound to a specific language and can be implemented for other platforms too.

We believe that our technique is promising and could be combined with other works to achieve even better results. For example, the declarative notation of IMUnit could be combined with the freedom the *Interleaving* provides to initiate the events from every place in the code. Moreover, the idea of using breakpoints for execution flow interception could be used for other purposes like invariants validation or code instrumentation.

References

[1] Long, B., Strooper, P.: A classification of concurrency failures in java components. In: Proc. of IPDPS 2003, p. 287.1. IEEE Computer Society, Washington, DC, USA (2003)

[2] Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: Proc. of ASPLOS 2008, pp. 329-339. ACM, New York, NY, USA (2008)

[3] Yang, C-S. D.: Structural Testing of Shared Memory Parallel Programs. PhD thesis, University of Delaware (1999)

[4] Radnoci, R.: Methods for Testing Concurrent Software. Master thesis, University of Skï¿œvde (2009)

[5] Eytani, Y., Havelund, K., Stoller, S. D., Ur, S.: Toward a Benchmark for Multi-Threaded Testing Tools.

[6] Souza, S. R. S., Brito, M. A. S., Silva, R. A., Souza, P. S. L., and Zaluska, E.: Research in concurrent software testing: a systematic review. In: Proc. of PADTAD 2011. ACM, New York, NY, USA (2011)

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

[8] Pugh, W., Ayewah, N.: Unit testing concurrent software. In: ASE (2007)

[9] Jagannath, V., Gligoric, M., Jin, D., Luo, Q., Rosu, G., Marinov, D.: Improved multithreaded unit testing. In: FSE 2011

[10] Bug 4810210 Java Bug Database. http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4810210

[11] Bug 4813150 Java Bug Database. http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4813150

[12] O'Callahan, R., Choi, J.-D.: Hybrid Dynamic Data Race Detection. In: PPoPP (2003)

[13] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A Dynamic Data Race Detector for Multithreaded Programs. In: ACM Transactions on Computer Systems, v.15 n.4, pp.391-411 (1997)

[14] Yu, Y., Rodeheffer, T., Chen, W.: RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In: SOSP (2005)

[15] Edelstein, O., Farchi, E., Nir, Y., Ratsaby, G., Ur, S.: Multithreaded java program test generation. In: IBM system journal, v.10 n.1, pp. 111-125, 2002

[16] Luo, Q., Zhang, S., Zhao, J.: A Lightweight and Portable Approach to Making Concurrent Failures Reproducible. In: Proc. of FASE 2010. pp. 323-337, Springer-Verlag, Berlin, Heidelberg (2010)

[17] Park, C. S., Sen, K.: Randomized Active Atomicity Violation Detection in Concurrent Programs. In: SIGSOFT (2008)

[18] Ball, T., Burckhardt, S., Musuvathi, M., Qadeer, S.: First-class Concurrency Testing and Debugging. Position Paper, Workshop on Exploiting Concurrency Efficiently and Correctly, 2008

[19] Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P. A., Neamtiu, I.: Finding and Reproducing Heisenbugs in Concurrent Programs. In: Proc. of OSDI 2008. pp. 267-280. USENIX Association, Berkeley, CA, USA.

[20] Long, B.: Testing concurrent java components. PhD thesis, The Universiy of Queesland (2005)

[21] Long, B., Hoffman, D., Strooper, P. A.: Tool support for testing concurrent java components In: IEEE TSE (2003)

[22] Park, C. S., Sen, K.: Concurrent Breakpoints. In: Proc. of PPoPP 2012. ACM, New York, NY, USA

[23] Erickson, J., Musuvathi, M., Burckhardt, S., Olynyk, K.: Effective Data-Race Detection for the Kernel. In: Proc. of OSDI 2010. pp. 151-162

[24] Vainer, E.: Master thesis. Tel Aviv University, in preparation (2013).