Taming the Concurrency:
Controlling Concurrent Behavior
while Testing Multithreaded Software

Thesis submitted in partial fulfilment
of the requirements for the Degree
of
Master of Science
by
Evgeny Vainer

This work was carried out under the supervision of
Professor Amiram Yehudai

Tel Aviv University
December 2013
Acknowledgements

I thank my advisor for his wise advices, patience and optimism. I thank my family for their support, for believing in me and for pushing me higher and higher all the time.
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.

Abstract
4.2 IMUnit events simplification ........................................... 36
4.3 Simple schedule $e_p \rightarrow e_s$ ........................................... 38
4.4 Complex Schedules ..................................................... 40
4.5 Summary ................................................................. 45

5 Applications ............................................................. 47

6 Related Works ............................................................ 49

7 Conclusions ............................................................... 53

Bibliography ............................................................... 55
List of Tables

3.1 Test execution time ................................... 32
List of Figures

2.1 Shared Memory Access ........................................ 7
2.2 UI for working with Interleaving framework .............. 17

3.1 Unit test for ArrayBlockingQueue class ..................... 20
3.2 Unit test and gate for ArrayBlockingQueue class using Interleaving framework .......................... 21
3.3 Unit test for LongRunningTask class ......................... 22
3.4 Unit test and gate for LongRunningTask class using Interleaving framework .......................... 23
3.5 AbstractStringBuilder.append method ...................... 24
3.6 Test method for StringBuffer.append ...................... 25
3.7 Gates defined for LengthRaceCondition test ................ 25
3.8 ArrayList.addAll method .................................. 26
3.9 Test for ArrayList.addAll method .......................... 27
3.10 Gates defined for ArrayList_RaceCondition_Interleaved test .... 27
3.11 Transformation $\mathcal{T} : IMUnit Tests \rightarrow Interleaving Tests$ ........ 29

4.1 Syntax of the IMUnit scheduling language .................... 37
4.2 Transformation $\mathcal{A}_1 : IMUnit Tests \rightarrow Interleaving Tests$ for simple schedule (case 4.3) ......................... 39
4.3 Transformation $\mathcal{A} : IMUnit Tests \rightarrow Interleaving Tests$ for complex schedule (case 4.4) ......................... 40
Chapter 1

Introduction

In recent years multicore hardware has become 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 including 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 the developer’s machine or team’s test server but always fail in end user’s environment.

To overcome this problem, unit tests developers try to force context switches in the critical code regions or to delay some code block execution until the another code block execution ends. These goals are usually achieved by adding additional operations (like Sleep or Wait/Notify) to the test logic, thus making the test more complicated. This approach creates additional problems. The sleep intervals are usually chosen by trial and error, 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 some third party component. In such a case, the developers have 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

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 ad-
dition, 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 thesis is organized as follows:

• chapter 2 gives more detailed description of our idea, including some im-
plementation details

• chapter 3 provides some examples of usage of the Interleaving framework
in order to reproduce bugs in concurrent code

• chapter 4 formally proves that the Interleaving framework can be used to
reproduce all the bugs that could be reproduced using IMUnit framework

• chapter 5 provides some ideas regarding the application of the Interleaving
framework in real life projects

• chapter 6 reviews some other work in this area

• chapter 7 concludes and provides some ideas for future research
Chapter 2

Solution

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

2.1 Idea

To achieve such 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 $G = \langle L, C \rangle$

where:

$L$ - some location in code which the execution flow could reach during the test run

$C$ - 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 ($L$) and could be opened ($C$ evaluates to true) or closed ($C$ 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 $C$ - it could check anything 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 strange conditions will have no value for real tests and it's 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 $T$ could reach the location $L$. At this point the execution of $T$ is suspended and condition $C$ is evaluated. The following behavior of $T$ depends on $C$’s value:

- $C$ evaluates to true - thread $T$ is resumed and continues its execution in a regular way
- $C$ evaluates to false - thread $T$ remains suspended and will be resumed only after the value of $C$ 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 $T$ will be resumed as soon as $C$’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 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 2.1.

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
2.1. IDEA

public class SharedMemoryAccessExample {
    int multiplier = -1;
    int result = 0;

    class Worker1 extends Thread {
        public void run() {
            multiplier = 1;
        }
    }

    class Worker2 extends Thread {
        public void run() {
            result = multiplier * 10;
        }
    }

    public int Calculate() throws Exception {
        Thread t1 = new Worker1();
        Thread t2 = new Worker2();

        t1.start();
        t2.start();
        t1.join();
        t2.join();
        return result;
    }
}

Figure 2.1. Shared Memory Access

“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 expected result is 10, while the result -10 (which will be returned if line 13 executed before line 07) is a bug.

Even such a simple example of a 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 the 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, one could define the gate $G = \langle \text{line 07, thread Worker2 has finished its execution} \rangle$ 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 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 Eclipse 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:
2.2. IMPLEMENTATION

- Java is a very powerful programming language. It’s syntax and semantics have 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 the same language is used to define conditions and application and because the same runtime is 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.
CHAPTER 2. SOLUTION

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 $G = \langle L, C \rangle$, it is resumed immediately when $C$’s value becomes true. This definition assumes some mechanism that observes the value of the condition all the time and is able to resume $T$ whenever condition 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 implementation 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 $T$ reaches gate $G = \langle L, C \rangle$ its state $S$ 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 $S$ is restored and $T$ continues its execution in the regular way. This behavior fits the gate’s behavior from section 2.1, since thread $T$ can not continue it’s execution until $C$ 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 thesis.
2.2. IMPLEMENTATION

2.2.3 Location definition

Now we describe the technique we used to define the location $L$ 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 $G$ in order to evaluate the condition and suspend thread’s execution if needed.

Fortunately, we are not 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 2.2.7). 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 a regular way.

While running the test, some of the breakpoints might be hit. When this happens, the thread $T$ that hit the breakpoint is suspended by the 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 $T$’s stack and $T$ is resumed. This technique causes $T$ 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 the 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 managed to implement these capabilities without modifications made to any of those libraries. As a result, the Interleaving framework does not require 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 the code of figure 2.1. 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 that 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 in order to associate the gates with the test is to create breakpoints working set, 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 $T_1$. At this point, Interleaving will use the technique we described in section 2.2.4 to cause $T_1$ 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 Open method of the same gate was called. Let us assume that the Open method of “Worker2Done” gate was not called yet. Therefore, $T_1$ 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 $T_2$ will hit the breakpoint located at line 14, this will cause $T_2$ 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 $T_2$ to return to the test logic. In addition, this call will cause the Wait method of “Worker2Done” gate to return, releasing $T_1$ 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 $E_1 \rightarrow E_2$ to denote that event $E_1$ occurs before event $E_2$):

- The code in line 13 is executed ($A$) before the breakpoint on line 14 is hit ($B$) ($A \rightarrow B$)
- “Worker2Done” Open method is called ($C$) after the breakpoint on line 14 is hit ($B \rightarrow C$)
- “Worker2Done” Wait method returns ($D$) after its Open method is called ($C \rightarrow D$)
- condition evaluation in $T_1$ ends ($E$) after “Worker2Done” Wait method returns ($D \rightarrow E$)
- thread $T_1$ returns to test logic execution ($F$) after it completed condition evaluation ($E \rightarrow F$)
- the breakpoint in line 07 is hit before the code on the same line is executed, as a result $T_1$ will execute the code in line 07 ($G$) only after it returns back to the test logic evaluation ($F \rightarrow G$)

Events sequence above implies that $A \rightarrow G$ (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 Deadlock detection

As one could already notice, using Interleaving framework means interfering with threads scheduling. This is what the framework was created for and this is where its additional value comes from. However, threads synchronization is a very delicate area. Careless positioning of the gates inside the code or incorrect use of notification mechanisms may lead to deadlocks that otherwise would never arise in the original code.

In order to cope with this problem, every framework like Interleaving has to provide some deadlock detection mechanism that will break the test execution and notify the tester as soon as the deadlock discovered. The logic of such a mechanism is the separate topic many researches address [23, 24] and, in our opinion, is out of the scope of our research. For this reason we leave the integration of such a mechanism for the future versions of Interleaving assuming the developer is qualified and careful enough to avoid deadlock while creating the test.

2.2.7 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 2.2 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 him 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 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).
Chapter 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 [15, 16] in order to demonstrate their approaches and we continue with this tradition. The code of the test is presented in figure 3.1. 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.
@Test
public void ArrayBlockingQueue_JUnit() throws Exception {
    final ArrayBlockingQueue<Integer> q = new ArrayBlockingQueue<Integer>(1);

    Thread addThread = new Thread(
        new Runnable() {
            public void run() {
                q.add(1);
                Thread.sleep(100);
                q.add(2);
            }
        }
    );
    addThread.start();
    Thread.sleep(50);

    Integer taken = q.take();
    assertTrue(taken == 1 && q.isEmpty());
    taken = q.take();
    assertTrue(taken == 2 && q.isEmpty());
    addThread.join();
}

Figure 3.1. Unit test for ArrayBlockingQueue class

Figure 3.2 shows the same test rewritten for Interleaving framework. 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”. This ensures the add method invoked at line 13 will be executed only after the thread Τ performing the take method at line 24 is blocked.¹

In contrast to the original sleeps based test, the Interleaving test described always enforces the correct threads ordering leading to consistent results for all

¹ This test could be rewritten using ThreadBlockingGate defined in section 4.2. By using this gate type, the tester is able to remove the gate from code under test and replace it by the appropriate call inside the code of the unit test itself, thus creating more robust test that does not depend on code under test implementation. Although such test is considered to be better, the implementation provided in the example better demonstrates the concepts and entities of Interleaving framework.
3.1. EXAMPLES

```java
@Interleaved
public void ArrayBlockingQueue_Interleaved() throws Exception {
    final ArrayBlockingQueue<Integer> q = new ArrayBlockingQueue<Integer>(1);

    Thread addThread = new Thread(
        new Runnable() {
            public void run() {
                q.add(1);
                interleavings.GateManager.Open("finished_add1");
                q.add(2);
            }
        }
    );

    addThread.start();
    GateManager.Wait("finished_add1");

    Integer taken = q.take();
    assertTrue(taken == 1 && q.isEmpty());
    taken = q.take();
    assertTrue(taken == 2 && q.isEmpty());

    addThread.join();
}
```

\[ G =< \text{ArrayBlockingQueue}@317, \]
\[ \text{interleavings.GateManager.Open("started\_take2");} > \]

**Figure 3.2.** Unit test and gate for ArrayBlockingQueue class using Interleaving framework

of the test runs. In addition, our code is 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
@Test
public void LongRunningTask_JUnit() throws InterruptedException {
    Task task = new LongRunningTaskExample().new Task();
task.start();
Thread.sleep(task.MaxTime);
assertTrue(task.IsDone);
}

Figure 3.3. Unit test for LongRunningTask class

varies from run to run in hardly predictable way, and depends mostly on the en-
vironment 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 3.3 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 fast, while the “busy
wait” option consumes unnecessary machine resources.

Figure 3.4 demonstrates Interleaving version of such a test. It contains two
gates:

• $G_1$ 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)

• $G_2$ is a fictitious gate (as described in section 2.2.2) that opens $G_1$ and is
  located on the last line of the checked operation (line 33 of LongRunning-
  TaskExample.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
3.1. EXAMPLES

```java
@Test
@Interleaved
public void LongRunningTask_Interleaved() throws InterruptedException {
    Task task = new LongRunningTaskExample().new Task();
    task.start();
    //interleavings.GateManager.Wait("task done");
    assertTrue(task.IsDone);
}
```

\[G_1 = \langle \text{LongRunningTask}_\text{Interleaved}@10, \text{interleavings.GateManager.Wait("task done")}; >\]

\[G_2 = \langle \text{LongRunningTaskExample}@33, \text{interleavings.GateManager.Open("task done")}; >\]

---

**Figure 3.4.** Unit test and gate for LongRunningTask class using Interleaving framework

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 [17, 18, 42]. Figure 3.5 contains the code of the append method of AbstractStringBuilder class which StringBuffer class inherits.

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 3.6.

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
Figure 3.5. AbstractStringBuilder.append method

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” [17] and proposes a test containing two infinite loops (one loop for each thread) [18] 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 3.7.

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

```java
public AbstractStringBuilder append(StringBuffer sb) {
    if (sb == null)
        return append("null");
    int len = sb.length();
    int newCount = count + len;
    if (newCount > value.length)
        expandCapacity(newCount);
    sb.getChars(0, len, value, count);
    count = newCount;
    return this;
}
```
3.1. EXAMPLES

```java
@Test
public void LengthRaceCondition() throws Exception {
    final StringBuffer sb1 = new StringBuffer("original data");
    final StringBuffer sb2 = new StringBuffer("appended data");

    Thread worker = new Thread(new Runnable() {
        public void run() {
            sb1.append(sb2);
        }
    });
    worker.start();
    sb2.setLength(3);
    worker.join();
}
```

Figure 3.6. Test method for StringBuffer.append

\[G_1=\langle test@13, interleavings.GateManager.Wait("afterget")\rangle\]
\[G_{1fictitious}=\langle append@06, interleavings.GateManager.Open("afterget")\rangle\]
\[G_2=\langle append@10, interleavings.GateManager.Wait("afterset")\rangle\]
\[G_{2fictitious}=\langle test@14, interleavings.GateManager.Open("afterset")\rangle\]

Figure 3.7. Gates defined for LengthRaceCondition test

is responsible for opening the real one, as described in section 2.2.2

3.1.4 ArrayList concurrency

Till now all the examples we presented used SimpleGate in order to define the desired concurrent behavior. Even such a simple gate was powerful enough so that we could reproduce some concurrent bugs that are hard to reproduce using other techniques existing today. In this example we want to demonstrate the usage of another gate implementing more complicated scheduling logic.

ArrayList is a well known and widely used class existing in Java. Its current implementation is known to be not thread safe and provide no guarantees when using the same ArrayList object in multithreaded environment. Despite this fact, in some cases, the concurrent operations performed on the same instance of ArrayList do perform in the expected way since the chance for the data race to
public boolean addAll(Collection<? extends E> c) {
  Object[] a = c.toArray();
  int numNew = a.length;
  ensureCapacity(size + numNew);
  System.arraycopy(a, 0, elementData, size, numNew);
  size += numNew;
  return numNew != 0;
}

Figure 3.8. ArrayList.addAll method

happen during the execution is pretty low. This issue may confuse inexperienced developers and lead to bugs in the code they write.

Suppose somebody wants to demonstrate the unsafety of ArrayList in multi-threaded scenarios. In order to do this, he needs to perform several operations on the same instance of ArrayList class using different threads and ensure that those operations will necessarily lead to object’s state corruption. As always, doing so is not easy since the context switches that take place during the run are out of control of the test developer.

Figure 3.8 presents the source code of ArrayList.addAll method.

As it was mentioned before, this method is not thread safe. For example, if two threads execute addAll code and both of them first perform line 6 of the method and only then, simultaneously, execute line 8, the object’s state will be corrupted. In order to reproduce this problem one could write a test that is similar to the test presented on figure 3.9.

The outcome of this test depends on the context switches that took place during its execution. When working on this example we rerun this many times inside the loop and noticed that the assertion on line 28 fails for one execution of several hundreds.

Interleaving allows us to reproduce the race for all test executions. To do so, we used a new gate type we called BarrierGate. As follows from its name, the BarrierGate behavior is very similar to the functionality of the synchronization primitive called barrier. In contradiction to the SimpleGate we used earlier, the
@Test
public void ArrayList_RaceCondition_Interleaved() throws Exception {
    final ArrayList<String> tested = new ArrayList<String>();
    Thread worker1 = new Thread(new Runnable() {
        public void run() {
            ArrayList<String> data = new ArrayList<String>();
            data.add("data");
            tested.addAll(data);
        }
    });
    Thread worker2 = new Thread(new Runnable() {
        public void run() {
            ArrayList<String> data = new ArrayList<String>();
            data.add("data");
            tested.addAll(data);
        }
    });
    worker1.start();
    worker2.start();
    worker1.join();
    worker2.join();
    //assertNotNull(tested.get(tested.size() - 1));
}

Figure 3.9. Test for ArrayList.addAll method

\[ G = \langle \text{addAll@8, interleavings.GateManager.Wait("after_copy")}\rangle \]

Figure 3.10. Gates defined for ArrayList_RaceCondition_Interleaved test

BarrierGate cannot be opened by calling its Open() method (the method left unimplemented) but will open itself after a predefined number of threads called its Wait() method. For the test case above, we placed the BarrierGate on the line 8 of addAll method (figure 3.10).

When running the test, the first worker thread reaching line 8 of addAll method will be blocked by the gate until the second worker thread will reach
the gate too. At this point of time both worker threads performed arraycopy call (line 6 of addAll method) but none of them increased the internal size variable (line 8 of addAll method), thus the tested object already contains less elements then expected. After both threads reach the gate, it will open, releasing the workers to perform the rest of addAll method code. Each one of them will increase the size variable, together causing the corruption in the internal state of the ArrayList object they are working on. The assert inside the test code (line 28) will validate the state and fail because of the corruption created by worker threads. This flow will happen for every test execution consistently reproducing the desired bug.

3.2 Comparison to IMUnit

IMUnit [16] 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:
3.2. **COMPARISON TO IMUNIT**

1. the same approach described in [16] 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 substantiate the claims above, we developed a simple algorithm that allows to convert every IMUnit test to *Interleaving* notation. This algorithm is presented on figure 3.11. 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. Here we provide the general idea and the intuition behind this transformation, while the whole and formal proof is presented in chapter 4.

1. let $e_p \rightarrow e_s$ 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 $G_{e_p \rightarrow e_s} = <L, C>$ as follows:

   3.1 $L = L_{e_s}$  
   3.2 $C = L_{e_p}$ was already executed

---

**Figure 3.11.** Transformation $\mathcal{T} : IMUnit\ Tests \rightarrow Interleaving\ Tests$

The intuition behind this transformation is very simple – the execution flow could not reach $L_{e_s}$ before it passes the gate $G_{e_p \rightarrow e_s}$, but the gate remains closed until the flow executes $L_{e_p}$. 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 $[e_p] \rightarrow e_s$) 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 \( e_p \rightarrow e_s \). 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.

### 3.3 More Tests

Inspired by the observations presented in section 3.2 we tried to apply them in practice. The IMUnit package available for download at [19] comes with two hundred example unit tests that were created by the framework authors based on the real tests from different projects [8], [9], [10], [11], [12], [13]. We converted these tests to Interleaving gates notation by applying a transformation algorithm.
very similar to one presented on figure 3.11. The conversion took very little amount of time and effort and at the end we got 196\(^2\) working interleaving tests that demonstrate consistent behavior for all of the runs. In addition, the outcome of all of the converted test is equal to the outcome of the original tests. Since the origin of all the tests are different real life projects, we conclude that our notation, combined with the prototype implementation we provide, are powerful enough to be used in real life testing.

3.4 Runtime Performance

The performance of unit testing framework is a very important issue. Since many real life projects have thousands of tests, the little overhead the framework creates for each one of the tests can create a huge delay when executing the whole test set. Thus, the unit testing framework developers should aim the lowest overhead they can achieve.

Despite of the statement in the previous paragraph, the first and the most important problem we cope with in this research is the outcome reproducibility and the control we want to give the test developer over the test execution. The prototype implementation we provide with this work was created in order to demonstrate Interleaving idea, its feasibility and usability and not in order to compete with mature, production ready frameworks existing today.

The running time of the tests we measured while using Interleaving framework is not significantly different from the running time of the regular concurrent tests validating the same scenario, and in some cases even lower (please recall the example 3.1.2). We can report of the average increase in the test execution time by factor of x1.05 when using Interleaving framework compared to original IMUnit tests. Table 3.1 summarizes observed execution time for both frameworks. These results was measured on the test base mentioned in section 3.3. Please recall [16] that switching to IMUnit notation reduces the execution time of the tests by factor of x3.39 when compared to original unit test. Combing this measurements

\(^2\)there are 6 more tests we did not succeed to execute even in the original IMUnit notation
Table 3.1. Test execution time

together we can conclude that the execution time of Interleaving version of tests is at least by factor of x3 better than this of the original sleep based tests.

We strongly believe that the results presented above could be improved even more since the performance of some software products strongly depends on time and effort invested in its development and on professional level of the developers that take part in the project. However, we leave this performance research and improvements for the possible future works.
Chapter 4

Reduction from IMUnit to Interleaving

This chapter presents the algorithm we propose for transforming IMUnit unit tests into Interleaving ones. The purpose of this algorithm is to provide an easy and straightforward transformation mechanism for every unit test defined using IMUnit events notation to another unit test that uses Interleaving gates mechanism and notation. The transformed unit test should preserve the outcome of the test execution and, of course, the threads scheduling that takes place during test execution. In addition, we would like the transformed test to be as similar as possible to the original one.

The algorithm was partially presented in figure 3.11 while the purpose of this insight was to provide the reader the intuition that stands behind the transformation, that, we believe, was important for understanding the ideas presented in section 3.2. In this section, in contrast to section 3.2, we provide the full version of the transformation algorithm that covers all possible scenarios and usecases. Thus, there might be some difference in the notation we use and in the algorithm itself. In addition, we formally prove that the proposed transformation preserves the unit test properties we mentioned earlier.
CHAPTER 4. REDUCTION FROM IMUNIT TO INTERLEAVING

4.1 Proof structure

Given a unit test $UT_{IMU}$ we provide the transformation algorithm $A : IMUnit Tests \rightarrow Interleaving Tests$ such that the unit test $UT_{Int} = A(UT_{IMU})$ has the following properties:

1. Interleaving test - $UT_{Int}$ is defined using Interleaving gates and makes no use of IMUnit events or schedules.

2. Halting equality - if every possible execution of $UT_{IMU}$ halts (does not deadlock) then every possible execution of $UT_{Int}$ halts (does not deadlock).

3. Outcome equality - if the execution of $UT_{Int}$ halts, the result of the execution of $UT_{Int}$ is equal to the result of the execution of $UT_{IMU}$.

One could notice that even when using IMUnit framework, several executions of the same test may still end up with different results. This could happen if some of the threads orderings that is crucial for the test outcome is left undefined in the IMUnit scheduling created for the test. In such a case, the decision regarding the execution order of the threads is left for the OS threads scheduler and could differ in different test executions. The existence of such a test makes requirement 3 in the list above very discouraging. On one hand, such a test is still IMUnit test so the appropriate Interleaving test created by the transformation $A$ has to return the same result as the original IMUnit test. On the other hand, the outcome of the original test can change from execution to execution and depends on the OS threads scheduler. Since the decisions made by OS threads scheduler are unpredictable from the test’s point of view and can change from execution to execution, one can conclude that appropriate Interleaving test will never satisfy the outcome equality condition (3). In order to solve this contradiction we replace the outcome equality property by the following one:

4. Execution order equality - for two instructions $p$ and $s$ of $UT_{IMU}$, if $p$ is executed before $s$ for every possible execution of $UT_{IMU}$ (regardless of
decisions made by OS threads scheduler) then the instruction $p$ is executed before the instruction $s$ for every possible execution of $UT_{Int}^1$.

This property of $A$ guarantees that $UT_{Int}$ preserves the execution order of all the instructions in $UT_{IMU}$ that have well defined execution order.

Please recall that the fact that some test ends up with the constant outcome for all of its runs implies that for every pair of its instructions $p$ and $s$, such that their execution order effects the test outcome, $p$ and $s$ has well defined execution order. Otherwise the result of the test will depend on the ordering that took place during the run making the test non deterministic and its outcome irreproducible. This observation, combined with the execution order equality (4) leads to the outcome equality (3) for all of the well defined IMUnit tests.

As for the instructions that do not have well defined ordering - the transformation does not make any promises regarding their execution order in the transformed test since every possible order may be considered as correct one. In addition, since the topic of this work is reproducibility, such tests are out of our scope since the absence of well defined execution order makes the outcome of such tests irreproducible.

We split the construction of the transformation $A$ to several parts:

1. First we show that different complex events defined by IMUnit scheduling language could be expressed using simple events initiated from gates located in code under test.

2. Next we construct separate transformation for some test defined using single IMUnit schedule of form $e_p \rightarrow e_s$.

3. Finally we show the transformation for complex IMUnit schedules that are defined as conjunction of simple schedules of form (2).

For the steps (2) and (3), we show that the transformation we create satisfies the “Interleaving test” and “Halting equality” requirements as we defined them. 

\footnote{As we will show shortly, the transformation we propose does not affect the original unit test code, but changes the IMUnit instructions only, thus, the original unit test instructions exist in the transformed test too.}
In addition, for the last step we also show that the transformation satisfies the “Execution order” requirement.

For the rest of the proof we use the following notations:

- \( L \) - the set of instructions that will be executed during the test run.
- \( l_i \) - some single instruction.
- \( T(l_p) < T(l_s) \) - means that the instruction \( l_p \) is executed before instruction \( l_s \) for every possible schedule that takes place during the test run.
- \( \text{next}(l) \) - the first instruction that will be executed by some thread after it executed instruction \( l \).
- \( \text{@event}(e) \) - instruction used to initiate IMUnit event \( e \) inside the code of the test.

### 4.2 IMUnit events simplification

The formal definition of IMUnit scheduling language syntax [16] presented in figure 4.1 defines more complicated constructs for condition part of the schedule then simple events we will use in our algorithm \( \mathcal{A} \). However, all of these constructs could be represented using different combinations of these basic events in the following way:

- \( e@t \) notation means that event \( e \) is relevant for the scheduling purposes only when initiated by thread \( t \). This event type could be reduced to simple event by moving the thread’s name validation to be the part of gate logic.

- \( \text{start}@t \) and \( \text{end}@t \) events are initiated automatically by the framework when the thread \( t \) starts or ends respectively. These events are automatically initiated by IMUnit in order to overcome its limitation that does not allow test developer to initiate the events outside the code of the test. Since this limitation does not exist in Interleaving we can replace these events by simple events initiated manually just before the first line and just after the last line of the method thread \( t \) executes.
### 4.2. IMUNIT EVENTS SIMPLIFICATION

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;Schedule&gt;</code></td>
<td><code>:= { &lt;Ordering&gt; [&quot;,&quot;, ] } &lt;Ordering&gt;</code></td>
</tr>
<tr>
<td><code>&lt;Ordering&gt;</code></td>
<td><code>:= &lt;Condition&gt; &quot;\rightarrow&quot; &lt;Basic Event&gt;</code></td>
</tr>
<tr>
<td><code>&lt;Condition&gt;</code></td>
<td>`:= &lt;Basic Event&gt;</td>
</tr>
<tr>
<td></td>
<td>`</td>
</tr>
<tr>
<td></td>
<td>`</td>
</tr>
<tr>
<td></td>
<td>`</td>
</tr>
<tr>
<td><code>&lt;Basic Event&gt;</code></td>
<td><code>:= &lt;Event Name&gt; [&quot;@&quot; &lt;Thread Name&gt;]</code></td>
</tr>
<tr>
<td></td>
<td>`</td>
</tr>
<tr>
<td></td>
<td>`</td>
</tr>
<tr>
<td><code>&lt;Block Event&gt;</code></td>
<td><code>:= [&quot; &lt;Basic Event&gt; &quot;]&quot;</code></td>
</tr>
<tr>
<td><code>&lt;Event Name&gt;</code></td>
<td><code>:= { &lt;Id&gt; &quot;.&quot; } &lt;Id&gt;</code></td>
</tr>
<tr>
<td><code>&lt;Thread Name&gt;</code></td>
<td><code>:= &lt;Id&gt;</code></td>
</tr>
</tbody>
</table>

**Figure 4.1.** Syntax of the IMUnit scheduling language

- Blocking event \([e_p] \rightarrow e_s\) could be replaced by the combination of two schedulings \(e_p \rightarrow e_s\) and \(e'_p \rightarrow e_s\). Here \(e'_p\) is a new event that is initiated just after the thread of interest blocks. This behavior is implemented by ThreadBlockingGate that is a part of the Interleaving framework and is fully consistent with the blocking event semantics as defined by IMUnit.

- \(e_1 \& e_2 \rightarrow e_s\) schedule requires both events \(e_1\) and \(e_2\) to occur before event \(e_s\) occurs. Since the complex schedule is defined as conjunction of the simple schedules it contains, this construct could be replaced by the combination of two simple schedules \(e_1 \rightarrow e_s\) and \(e_2 \rightarrow e_s\).

- \(e_1 \lor e_2 \rightarrow e_s\) schedule requires one of events \(e_1\) or \(e_2\) to occur before event \(e_s\) occurs. This construct could be replaced by the schedule \(e_{e_1\lor e_2} \rightarrow e_s\), where \(e_{e_1\lor e_2}\) is a new simple event that is initiated in several places in code, immediately after each initiation of events \(e_1\) and \(e_2\).

- When more complex logical combination of events is used as a condition to some event \(e_s\), it could always be represented in the form of \((\land C_i) \rightarrow e_s\) and therefore be split to several schedules of the form \(C_i \rightarrow e_s\). Afterwards, each one of the schedules \(C_i \rightarrow e_s\) could be recursively simplified until it is represented as the combination of schedules containing simple events only.

We just showed that every complex event defined by IMUnit scheduling language could be represented as the combination of schedules containing simple
events only. Therefore for the rest of the proof and without loss of generality, we can assume that the complex schedule of $UT_{IMU}$ the transformation $A$ is applied on contains simple schedules of the form $e_p \rightarrow e_s$ only.

### 4.3 Simple schedule $e_p \rightarrow e_s$

Let $UT_{IMU}$ to be the IMUnit unit test with only one schedule of form $e_p \rightarrow e_s$ defined for it and $L$ to be the set of all the instructions in $UT_{IMU}$. The existence of such a schedule implies the following:

**Proposition 1.**

- let $L_s = \{ l \mid l = \text{@event}(e_s) \}$ be a set of 0 or more instructions in the test that initiate event $e_s$
- let $L_p = \{ l \mid l = \text{@event}(e_p) \}$ be a set of 0 or more instructions in the test that initiate event $e_p$

then the test execution halts if and only if one of the following holds

1. $L_s = \emptyset$

2. there exists $l_p \in L_p$ that is executed before any instruction that is executed after any instruction $l \in L_s$.

Please note that all the $\text{@event}(e_i)$ instructions initiating events different from $e_p$ and $e_s$ could be removed from the test without changing its outcome since the events they initiate do not appear in the scheduling defined for the test and, as follows, do not effect test execution in any way. Therefore, we can assume that every event initiation instruction initiates event $e_p$ or $e_s$. In addition, since $e_p \neq e_s$ the sets $L_p$ and $L_s$ are disjoint.

Now we want to transform $UT_{IMU}$ from IMUnit events notation to Interleaving gates notation. For the purpose of transformation we will use the gate of type Simple Gate we defined earlier in section 2.2.5. The desired algorithm $A_1$ for transforming $UT_{IMU}$ to $UT_{Int}$ is presented on figure 4.2.
1. define Simple Gate named $G_p$

2. for each $l_p \in L_p$ replace $l_p$ by $G_p$.Open() instruction

3. for each $l_s \in L_s$ replace $l_s$ by $G_p$.Wait() instruction

Figure 4.2. Transformation $A_1 : IMUnit Tests \rightarrow Interleaving Tests$ for simple schedule (case 4.3)

Lemma 2. given $UT_{IMU}$ that has one schedule of form $e_p \rightarrow e_s$, $UT_{Int} = A_1(UT_{IMU})$ satisfies interleaving test requirement.

Proof. For each event initiation instruction $l$ in $UT_{IMU}$ either $l \in L_p$ or $l \in L_s$. $A_1$ replaces all the instructions both in $L_p$ and $L_s$ so the resulting unit test $UT_{Int} = A_1(UT_{IMU})$ contains no IMUnit event initiation instructions. In addition, every line in $L_p$ and $L_s$ is replaced by some gate instruction, thus the $UT_{Int}$ is an Interleaving test as required. Please note that for the trivial case when both $L_p$ and $L_s$ are empty sets, $UT_{Int}$ still can be considered an Interleaving test.

Lemma 3. given $UT_{IMU}$ that has only one schedule of form $e_p \rightarrow e_s$, $UT_{Int} = A_1(UT_{IMU})$ satisfies halting equality requirement.

Proof. Assume that $UT_{IMU}$ halts for each possible execution. According to proposition 1 the assumption implies one of the following:

1. $UT_{Int}$ contains no Wait instructions, i.e the transformation $A_1$ did not introduce new blocking instructions to into the code of the test $\Rightarrow A_1$ could not introduce new deadlocks into the test, thus, according to assumption, $UT_{Int}$ halts for every possible execution.

2. $\exists l_p$ that will be executed before any instruction that is executed after any of $l_s \in L_s$. This implies that some instruction $G_p$.Open() in $UT_{Int}$ will be executed before any instruction following $G_p$.Wait() instruction $\Rightarrow \exists l = G_p$.Open() in $UT_{Int}$ such that its execution does not depend on code that follows $G_p$.Wait() instruction $\Rightarrow$ for every possible run of $UT_{Int}$ at least one of $G_p$.Open() instruction will be executed $\Rightarrow$ the gate $G_p$ will be opened
and all the calls for $G_p\cdot$Wait() will return $\Rightarrow A_1$ could not introduce new
deadlocks into the test, thus, according to assumption, $UT_{Int}$ halts for every
possible execution.

\[\square\]

4.4 Complex Schedules

IMUnit framework provides support for complex schedules that are defined as
a conjunction of several simple schedules. The same event $e$ may appear in
several schedules providing the tester an ability to define complicated concurrent
scenarios. Figure 4.3 presents an algorithm $A$ used to transform $UT_{IMU}$ with
complex schedule to appropriate $UT_{Int}$.

1. for each event $e$ in complex schedule defined for $UT_{IMU}$
   1.1 create SimpleGate named $G_e$

2. for each simple schedule $S = e_p \rightarrow e_s$ in complex schedule defined for $UT_{IMU}$
   2.1 for each $l_p \in L_p$ replace $l_p$ by $G_{e_p}$.$Open()$ instruction
   2.2 for each $l_s \in L_s$ replace $l_s$ by $G_{e_p}$.$Wait()$ instruction

Figure 4.3. Transformation $A: IMUnit Tests \rightarrow Interleaving Tests$ for complex
schedule (case 4.4)

There is one important point to clarify regarding the transformation $A$ -
since every event $e$ may appear in several schedulings, each event initiation line
will be replaced by the block of interleaving instructions containing Wait() and
Open() calls for appropriate gates. The execution order of these instructions is
crucial for $A$ correctness so $A$ will place all the Wait() calls before all the Open()
calls. The intuition behind this ordering is as follows: the execution of Open() method of some interleaving gate in $UT_{Int}$ is logically equivalent to initiation of
appropriate event in $UT_{IMU}$. However, the IMUnit event is initiated only after
all of the preceding events defined in the schedule were initiated. This behavior
is reproduced by calling to Wait() methods of all of the "preceding" gates before the succeeding gate is opened, ensuring that all of them are in the open state.

**Lemma 4.** given $UT_{IMU}$ that has complex schedule, $UT_{Int} = A(UT_{IMU})$ satisfies interleaving test requirement.

**Proof.** The proof is exactly as the proof of lemma 2

**Lemma 5.** given $UT_{IMU}$ that has complex schedule, $UT_{Int} = A(UT_{IMU})$ satisfies halting equality requirement.

**Proof.** For this proof we define two event sets $E_p$ and $E_s$ as follows:

- $E_p = \{e \mid \exists f \text{ s.t. } S = e \rightarrow f \in \text{scheduling of } UT_{IMU}\}$
  i.e. $E_p$ is a set of events that appear as preceding event in at least one simple schedule

- $E_s = \{e \mid \exists f \text{ s.t. } S = f \rightarrow e \in \text{scheduling of } UT_{IMU}\}$
  i.e. $E_s$ is a set of events that appear as succeeding event in at least one simple schedule

Assume by contradiction that there is a gate $G$ that causes a deadlock in $UT_{Int}$, while appropriate $UT_{IMU}$ halts for every possible run. The gate $G$ was created by $A$ based on some event $e$ in the schedule of $UT_{IMU}$. Event $e$ can be categorized into one of the three following cases:

1. $e \in E_p \land e \notin E_s$
   following the assumption that $UT_{IMU}$ halts for every possible run we can conclude that $\forall S = e \rightarrow e_i \ T(l_e) < T(next(l_e))$ which means that the execution of instruction $l_e$ does not depend on execution of any of $l_{e_i}$ instructions or any code following them. Now we look at the transformed test $UT_{Int}$. Let $TS$ to be a threads scheduling (i.e. the set of context switches and their ordering) created by OS for some of $UT_{Int}$ executions. Although $TS$ specifies the order in which $UT_{Int}$’s instructions are executed, it can
not introduce new execution order dependencies into the test code. Thus, the execution of the appropriate \( G.\text{Open()} \) instruction does not depend on execution of any of \( G.\text{Wait()} \) instructions or any code following them for any possible TS. More formally, we can formulate that the the inequality 
\[ T(G.\text{Open}()) < T(\text{next}(G.\text{Wait}())) \]
holds for every call to \( G.\text{Wait()} \) method in \( UT_{int} \). This, in turn, implies that the gate \( G \) will be opened at some point during the \( UT_{int} \) execution \( \Rightarrow \) all the \( G.\text{Wait()} \) calls will return \( \Rightarrow \) gate \( G \) will not cause a deadlock. This contradicts the assumption we made regarding gate \( G \Rightarrow \) event \( e \) does not belong to this group.

2. \( e \in E_s \land e \notin E_p \)

In this case the original test \( UT_{IMU} \) does not contain schedule that depends on \( e \ (e \notin E_p = \Rightarrow \exists S = e \rightarrow f) \Rightarrow \) transformed test \( UT_{int} \) does not contain \( G.\text{Wait()} \) calls \( \Rightarrow \) \( G \) can not cause deadlock. This contradicts the assumption we made regarding gate \( G \Rightarrow \) event \( e \) does not belong to this group.

3. \( e \in E_s \land e \in E_p \)

- first we’ll investigate \( e \in E_s \) part. The assumption that \( UT_{IMU} \) halts implies that \( \forall S = e_i \rightarrow e \ T(l_{e_i}) < T(\text{next}(e)) \). Using the reasoning similar to case (1) above we can conclude that:

\[- \forall S = e_i \rightarrow e \ T(G_{e_i}.\text{Open}()) < T(\text{next}(G_{e_i}.\text{Wait}())) \]

By the construction of \( UT_{int} \) we get:
\[- \forall S = e_i \rightarrow e \ T(\text{next}(G_{e_i}.\text{Wait}())) < T(\text{next}(G.\text{Open}())) \]

Combining these inequalities together we get that
\[- T(G_{e_i}.\text{Open}()) < T(\text{next}(G.\text{Open}())) \]

which means that all the gates \( G_{e_i} \) will be opened without any dependency on the code following \( G.\text{Open}() \) call \( \Rightarrow \) all the \( G_{e_i}.\text{Wait()} \) calls preceding \( G.\text{Open}() \) will return \( \Rightarrow \) the gate \( G \) will be opened, i.e. none of the gates created for events preceding \( e \) can prevent \( G \) from opening.
now we’ll investigate $e \in E_p$ part. The assumption that $UT_{IMU}$ halts implies that $\forall S = e \rightarrow e_i$ holds $T(l_e) < T(\text{next}(e_i))$. Using the reasoning similar to case (1) above we can conclude that:

- $\forall S = e \rightarrow e_i$ $T(\mathcal{G}.\text{Open}()) < T(\text{next}(\mathcal{G}.\text{Wait}()))$

By the construction of $UT_{Int}$ we get:

- $\forall S = e \rightarrow e_i$ $T(\text{next}(\mathcal{G}.\text{Wait}())) < T(\text{next}(\mathcal{G}_{e_i}.\text{Open}()))$

Combining these inequalities together we get that

- $T(\mathcal{G}.\text{Open}()) < T(\text{next}(\mathcal{G}_{e_i}.\text{Open}()))$

which means that the gate $\mathcal{G}$ will be opened without any dependency on the code following any of $\mathcal{G}_{e_i}.\text{Open}()$ calls $\Rightarrow$ all the $\mathcal{G}.\text{Wait}()$ calls preceding any of the $\mathcal{G}_{e_i}.\text{Open}()$ calls will return $\Rightarrow$ none of the gates based on events succeeding $e$ can prevent $\mathcal{G}$ from opening.

Following this we can conclude that even in this case gate $\mathcal{G}$ can not cause deadlock. This conclusion contradicts the assumption we made regarding gate $\mathcal{G}$ $\Rightarrow$ event $e$ does not belong to this group.

We just showed that event $e$ that the gate $\mathcal{G}$ is based on does not belong to any of the groups above $\Rightarrow e \notin E_p \land e \notin E_s$ which means that $e$ does not belongs to $UT_{IMU}$ scheduling $\Rightarrow$ gate $\mathcal{G}$ does not exist in $UT_{Int}$ thus it can not cause a deadlock, in contradiction to assumption we started with $\Rightarrow$ if $UT_{IMU}$ halts then $UT_{Int} = \mathcal{A}(UT_{IMU})$ also halts.

Lemma 6. given $UT_{IMU}$ that has complex schedule, $UT_{Int} = \mathcal{A}(UT_{IMU})$ satisfies execution order equality requirement.

Proof. Let $UT_{IMU}$ be IMUnit unit test and let $p, s$ be instructions from $UT_{IMU}$ such that for every possible execution of $UT_{IMU}$ that halts, instruction $p$ is always executed before instruction $s$ (i.e. $T(p) < T(s)$). In addition, given an $UT_{Int} = \mathcal{A}(UT_{IMU})$ there is at least one correct threads scheduling that causes instruction $p$ to be executed after instruction $s$.

We create new IMUnit unit test $D(UT_{IMU})$ using the following algorithm:
CHAPTER 4. REDUCTION FROM IMUNIT TO INTERLEAVING

• replace instruction $p$ with the following instructions block

\[
p_1: \text{after}_p \leftarrow \text{true};
\]

\[
p_2: \text{instruction } p \text{ of UT}_{IMU};
\]

• replace instruction $s$ with the following instructions block

\[
s_1: \text{instruction } s \text{ of UT}_{IMU};
\]

\[
s_2: \text{if (not after}_p \text{) deadlock();}
\]

Following the assumption we made for $UT_{IMU}$ $T(p) < T(s)$ for every possible execution $\Rightarrow$ the same holds for $D(UT_{IMU})$ $\Rightarrow T(p_1) < T(s_1)$ for every possible execution of $D(UT_{IMU})$ $\Rightarrow$ unit test $D(UT_{IMU})$ will never cause a deadlock.

However, for $UT_{Int} = A(UT_{IMU})$ there exists a threads scheduling $TS$ which causes $s$ to be executed before $p$. We define $TS'$ to be a part of $TS$ from the beginning till instruction $s$ is executed. Please note, that at the moment $TS'$ ends, instruction $p$ of $UT_{Int}$ is not executed yet. Now we look at the execution of $A(D(UT_{IMU}))$ using $TS'$ as threads scheduling. Since the effect of $TS'$ on the test execution ends after the instruction $s_1$ of $A(D(UT_{IMU}))$ is executed (and before instruction $p_1$ is executed), $TS'$ is not effected by the changes the transformation $D$ introduced into the test, thus $TS'$ is a correct partial scheduling for $A(D(UT_{IMU}))$. We now define new threads scheduling $TS''$ that is similar to $TS'$ until $TS'$ ends and executes instruction $s_2$ after $TS'$ ends. Since $s_1$ is the last instruction that is executed by $TS'$ it is absolutely legal to execute the instruction $s_2$ after that, thus $TS''$ is a correct partial scheduling for $A(D(UT_{IMU}))$. However, when executing $A(D(UT_{IMU}))$ using $TS''$ threads scheduling, the instruction $s_2$ is executed before the instruction $p_1$ causing the deadlock and preventing the execution from halting.

To summarize all the written above, given the assumption we started with we created new IMUnit unit test $D(UT_{IMU})$ that always halts, while some executions of appropriate Interleaving unit test $A(D(UT_{IMU}))$ may cause a deadlock. The existence of such a test contradicts lemma 5 that states that unit tests created
using $A$ satisfy halting equality requirement ⇒ the unit test $UT_{IMU}$ that satisfies the assumption we started with does not exist.

\[ \square \]

4.5 Summary

Now we can formulate the theorem that states that every IMUnit test can be rewritten as Interleaving test.

**Theorem 1.** Given a well defined IMUnit unit test $UT_{IMU}$ we can construct an Interleaving unit test $UT_{Int}$ that has the same outcome as $UT_{IMU}$.

**Proof.** Given $UT_{IMU}$ we define $UT_{Int} = A(UT_{IMU})$ where $A$ is a transformation defined on figure 4.3. The desired property of $UT_{Int}$ is an immediate consequence of lemmas 4, 5 and 6.

\[ \square \]

An immediate conclusion from the theorem 1 above is as follows:

**Conclusion 1.** Let IMU be all the unit test that could be created using IMU-unit framework, let Interleaving be all the unit test that could be created using Interleaving framework. Then $IMU \subseteq Interleaving$. 
Chapter 5

Applications

We believe that \textit{Interleaving} can be useful at different stages of software project lifecycle.

First, it could be used during development stage. When writing the code, unit tests are often used by developers to ensure that the code is following the specifications and is free from bugs. When dealing with concurrent code, the developers can use \textit{Interleaving} along with some unit testing frameworks in order to:

- Ensure that in every test run the tests are executed using the same threads scheduling preserving the constant outcome between the runs.

- Execute the same unit test using different threads schedulings ensuring the execution result remains constant and does not depend on the threads ordering.

- Create unit tests for synchronization mechanisms existing in the code, ensuring they behave as expected in different concurrent scenarios.

- Create complex unit tests involving several threads interaction.

Creating the scheduling for all of the cases above requires deep knowledge of the code and code flow. However, it is reasonable to assume that the developers are familiar with the code they are writing and aware of different concurrent issues that could effect its behavior. Thus, using \textit{Interleaving} the developer is able to
create more robust unit tests and to cover more aspects of the code behavior than when using standard frameworks only.

Next, *Interleaving* could be used during bug fixing process. When dealing with concurrent bug, the developer often does not have bug reproduction that could be used to understand the execution flow that leads to bug. In this situation, the only way to find such a flow is to investigate the code, checking different possible threads orderings. Creating a reproduction for these orderings could be complicated, so the investigation process could take very long time. In order to speed it up, many of the schedulings are filtered away by developers based on "thought experiment". This technique utilizes the fact that most of the concurrent bugs could be reproduced using small amount of context switches [34] so the experienced developer is usually able to "reproduce" the appropriate flow in his mind. However, there is still a chance that the developer will, by mistake, miss or filter out the buggy scheduling.

Using *Interleaving* during this investigation process could significantly reduce the time required to create the reproduction of some concurrent scenario. In addition, creating the reproduction using *Interleaving* is relatively easy. Together, this allows the developer to perform more real validations than he is able to perform today, making the investigation process more technical and reducing the chance the developer would miss the buggy scenario.

Of course, the *Interleaving* test reproducing the bug could be used later in order to validate that the bug is fixed. In addition, all the *Interleaving* tests created during the investigation process could be added to the application test set, increasing its coverage and ensuring that all the orderings checked will behave as expected in future versions.

The scenarios presented above are based on personal experience we obtained while working on different software projects during the last few years. In the future, we plan to perform a user study that will measure the performance and contribution of *Interleaving* for real life projects and its usability in the hands of real developers.
Chapter 6

Related Works

The problem of concurrent software testing has been studied by many researchers and there are plenty of papers and tools addressing different aspects of the problem. Generally, all the works on the topic could be divided to several groups, according to the approach the authors propose in order to cope with concurrency related issues:

1. Recording and replaying the error prone run for the future research and debugging

2. Automatic testing of given codebase for the problems caused by concurrency and synchronization issues

3. Manual testing of concurrency issues, by giving the tester an ability to control the scheduling as part of the test

The first topic is the well studied one. The authors of different works focus on recording different types of events during the application execution and use the collected data in order to create the exact replay of recorded run. The examples of such works are [25], [26], [27], [28]. The approach is so well studied that it is already utilized by commercial companies that provide production ready tools for record and replay of concurrent applications [14]. Although this technique is very powerful for debugging and bug fixing purposes, it is less useful for testing since
the bug prone run has to be somehow reproduced before it could be recorded for the first time.

The second group of the works is very heterogeneous. Many authors apply static analysis techniques to discover such types of concurrent bugs like deadlocks [35] or dataraces [36]. There is plenty of researches and tools [37], [38], [39] implementing different analysis techniques.

Other authors perform the analysis based on the data collected during the run. O’Callahan and Coi [20] analyze the runtime behavior of the application and apply lockset-based and happens-before techniques in order to identify potential bugs. Eraser [21] tracks application actions and uses collected data to detect possible dataraces. RaceTrack [22] 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 [29] introduces new context switches into the program code thus revealing hidden bugs. ConCrash [30] utilizes record and replay technique in order to reproduce buggy runs. AtomFuzzer [31] forces context switches inside critical regions trying to cause atomicity violation. Microsoft Chess [32, 33] reruns each test multiple times while enumerating over different possible thread schedulings.

DataCollider [43] 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 use this mechanism in order to change the execution flow induced by OS threads scheduler.

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

The third group is the smallest one and contains only few researches. All the works in this group try to utilize some information provided by developer / tester in order to reproduce the buggy state. ConAn [40, 41] and MultithreadedTC [15] split the application execution timeline to several slots providing the developer the ability to order the code blocks with respect to those slots. IMUnit [16] 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 [42] use the information provided by the developer regarding the buggy state and try to enforce the scheduling that will reach this state.
Chapter 7

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.

In order to demonstrate the feasibility of the technique we propose we implemented the prototype of our ideas in the Interleaving framework. Using this prototype we were able to reproduce several real life bugs that were considered hard to reproduce till now. In addition, the framework has good integration with Eclipse IDE and JUnit and does not require dedicated runtime environment. 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 provide the comparison of our framework against the best similar tool we are familiar with. We show that Interleaving has additional value when unit
testing the application, allowing the tester to reproduce bugs that he could not reproduce using another tool. In addition we show that the results achieved by other work are compliant with our idea and could be used as the basis when creating Interleaving unit tests.

We believe 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.
Bibliography


[19] IMUnit Project Homepage http://mir.cs.illinois.edu/imunit/


פיתוח של תוכנה מקבילה היא משימה מורכבת ומאっております, אפילו למפתחים מנוסים. משימות שמות הקWithOptions בהליך המיפוי, משיביכים לקוד, הפוכות להרכבות וור לתקע המקבילים של התוכנה. דוגמא טובה לשימוש ללא פיתוח בדיקה למבנים שונים היא פיתוח בדיקות למבנים אחרים של המבנה. בדיקה זו אמורה להכיל שפע של קוד של כיתרויות של המבנה ולא קשור בצורה ישירה לтрадיו של התוכנה. הביקורת של בדיקה זו של הביקורת לשון לבקיה של הביקורת וההתמקדות שלה על הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת ואיסוף לקח של הביקורת וההתמקדות. הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקורת של הביקורת, הביקורת ובדיקת את לקח הביקור
לנצח את המקביליות:
שליטה על התנהגות מקבילית לзорר
בדיקות של תוכנה מרובת הליכי משנה

יבגני ויינר

העבודה נעשתה בהנחיית
פרופסור עמירם יהודאי

אוניברסיטת תל אביב
טבת תשע"ד