UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

COOPERATIVE TESTING OF MULTITHREADED JAVA APPLICATIONS

Miguel Ângelo Marques Simões

DISSERTAÇÃO

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Engenharia de Software

2014

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

COOPERATIVE TESTING OF MULTITHREADED JAVA APPLICATIONS

Miguel Ângelo Marques Simões

DISSERTAÇÃO

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Engenharia de Software

Dissertação orientada pelo Prof. Doutor Francisco Cipriano da Cunha Martins
e co-orientada pelo Prof. Doutor Eduardo Resende Brandão Marques

2014

Agradecimentos

Agradeço aos meus orientadores, professores Eduardo Marques e Francisco Martins, pela constante disponibilidade, apoio e suporte para o desenvolvimento deste trabalho de modo a torná-lo o melhor possível. Aprendi imenso no desenrolar deste trabalho e grande parte devo-o a eles.

Agradeço imenso à minha família, pois sem ela não teria sido possível chegar onde cheguei. Aos meus pais, Fernando Simões Ferreira e Maria de Fátima Figueiredo, agradeço todo o apoio, motivação e também pelos sacrifícios que fizeram para me permitir chegar até aqui. Agradeço também à minha e irmã e sobrinho, Tânia e Rodrigo, pelo apoio constante e que nunca me deixaram de animar. A todos eles, muito obrigado.

Por fim, mas não menos importante, quero agradecer a todas as pessoas, que ao longo do meu percurso académico me ajudaram e apoiaram. Em especial a quatro pessoas que já me acompanham à algum tempo e tornaram tudo mais fácil. Ao João Caetano e Filipe Lemos pela companhia e ajuda em muitos trabalhos tanto da licenciatura, como de Mestrado, e por uma amizade pela qual tenho muito valor. E também à Lara Caiola e César Santos por toda a amizade e apoio que deram durante o mestrado e principalmente no desenvolvimento da tese. Todos eles foram importantes para chegar aqui.

Aos meus pais, avós, irmã e sobrinho.

Resumo

Para desenvolver programas concorrentes, o modelo de programação prevalente é o uso de *threads*. Uma *thread* define uma linha de execução sequencial num programa, em que partilha o espaço de memória com todas as outras *threads* que estão a executar o programa ao mesmo tempo. Além de memória partilhada, as *threads* de um programa também interagem mediante primitivas de concorrência como *locks* ou barreiras. O uso de *threads* é suportado por alguma das linguagens de programação mais conhecidas e utilizadas, como por exemplo, o Java ou C.

Em programas *multithreaded*, erros de programação são fáceis de cometer, com o uso de memória partilhada e primitivas de interacção. Por exemplo, a partilha de memória entre as *threads* pode levar a condições de corrida sobre os dados (*data races*), e inerente inconsistência de valores para os mesmos. Outro problema típico é a ocorrência de impasses (*deadlocks*) no uso de primitivas de concorrência, como *locks*, que não permitem que uma ou mais *threads* progridem na sua execução. Quando um *bug* se manifesta, um programador normalmente tenta determinar manualmente a sua origem, através de técnicas de *debugging*, e/ou executa o conjunto de testes associados ao programa, observando em quais testes são reportados erros. Frequentemente, estas duas abordagens falham para programas *multithreaded*.

A razão para estas dificuldades é que numa execução com múltiplas *threads* as mudanças de contexto acontecem de forma não-determinista, o que torna impossível observar e controlar de forma precisa o fluxo de execução e escalonamento de *threads* associado. Ao mesmo tempo, os *bugs* tipicamente apenas se manifestam num sub-conjunto de todos os escalonamentos possíveis, e com frequência apenas em escalonamentos bastante específicos e difíceis de reproduzir. Um teste de software pode ser repetido imensas vezes, sem expor um *bug*. Se e quando este é detectado, pode ser extremamente difícil identificar e replicar o fluxo de execução ocorrido. Até padrões de *bugs* relativamente simples são difíceis de detectar e replicar com precisão (por exemplo, ver [9]).

Esta tese trata do problema geral de testar programas concorrentes escritos em Java [16], de modo a que *bugs* possam ser descobertos e replicados de forma determinista. A nossa abordagem passa pela execução de testes de software com semântica cooperativa [23, 24]. A observação fundamental subjacente ao uso de semântica cooperativa é que as mudanças

de contexto relevantes entre *threads* acontecem apenas nos pontos de potencial interferência entre estas, tais como o acesso a dados partilhados ou o uso das primitivas para concorrência, chamados *yield points* cooperativos. Uma execução é considerada cooperativa se o código entre dois *yield points* consecutivos da mesma thread executa de forma sequencial como uma transação, sem qualquer interferência de outras *threads*.

A semântica cooperativa pode ser explorada para testes de software deterministas, em que a reproducibilidade de execução é garantida, e a cobertura sistemática do espaço de estados de escalonamento possa ser efectuada. A ideia base é executar as *threads* com semântica cooperativa, por forma a que a execução repetida de um teste possa explorar o espaço de estados de forma controlada e customizada, e até eventualmente exaustiva. Esta técnica é usada em algumas ferramentas [3, 7, 17] para programas concorrentes escritos em C/C++.

Nesta tese nós apresentamos o desenho, implementação e avaliação de uma ferramenta de testes cooperativa para programas concorrentes escritos em Java. A ferramenta desenvolvida chama-se Cooperari e está disponível em https://bitbucket.org/edrdo/Cooperari . Tanto quanto pudemos determinar pela avaliação do estado da arte, trata-se da primeira ferramenta de testes cooperativos para programas em Java. As contribuições desta tese são as seguintes:

- A ferramenta instrumenta o *bytecode* Java da aplicação, por forma a interceptar *yield points* na execução, e a tornar a execução cooperativa. Os *yield points* suportados concernem operações sobre monitores Java (operações de *locking* e primitivas de notificação), operações do ciclo de vida de uma *thread* definidos na API java.lang.Thread, e ainda acessos (de leitura e escrita) a campos de objectos ou posições de vectores. A instrumentação dos *yield points* é definida recorrendo à linguagem AspectJ [14].

- Para a execução cooperativa definimos um ambiente de execução, que compreende um escalonador (*scheduler*) cooperativo e uma implementação cooperativa de monitores Java e operações do ciclo de vida de uma *thread*. O escalonador cooperativo condiciona o escalonador nativo da máquina virtual Java (que se mantém activo), por forma a se obter uma semântica cooperativa na execução. Aquando da intercepção de um *yield point*, o escalonador cooperativo toma controlo da execução, e deterministicamente seleciona a próxima thread que irá executar.

- Desenvolvimento de duas políticas de cobertura. A escolha feita num passo de escalonamento é determinado pela política de cobertura do espaço de estados de escalonamentos. A primeira delas simplesmente escolhe a próxima *thread* a executar de forma pseudo-aleatória, sem manter estado entre várias execuções de um programa/teste; para execução determinista, o gerador de números aleatórios é inicializado com uma semente fixa. A outra política mantém um histórico de decisões

de escalonamento anteriores, por forma a evitar decisões repetidas de escalonamento para o mesmo estado do programa.

- Implementação de mecanismos de monitorização em tempo de execução para a deteção de *data races* e alguns tipos de *deadlock*. É assinalado um erro quando na monitorização do acesso aos dados é verificado que para a mesma variável ou posição do vector está a ser acedido por duas threads, em que uma delas está a escrever e a outra está a escrever ou ler. Nós detectamos *deadlocks* verificando se existe ciclos nas dependências dos *locks* ou se alguma *thread* está bloqueada eternamente pois está à espera de uma notificação.

- A ferramenta está integrada com o JUnit [20], o popular ambiente de testes para programas Java. Os programadores de testes podem usar a ferramenta JUnit da forma usual, recorrendo apenas a anotações adicionais às classes de teste. Na nossa ferramenta adicionamos novas anotações no JUnit, como por exemplo a definição da pasta que se deve instrumentar, o critério de avaliação para os testes e ainda uma que define que a execução irá ser cooperativa (execução por nós desenvolvida). Após a execução de um teste, a nossa ferramenta detecta se ocorreu um erro, se sim termina a execução, se não irá executar mais testes até cobrir a política ou então ter atingido o número máximo de execuções.

- A ferramenta foi avaliada com exemplos de programas *multithreaded* retirados dos repositórios ConTest e SIR, referenciados na literatura. No caso geral, a ferramenta consegue detectar e replicar deterministicamente os *bugs* dos programas em causa, em contraste a testes com execução preemptiva que na vasta maioria dos casos não o consegue fazer.

O trabalho levou à publicação do seguinte artigo em conferência internacional com revisão de pares:

Cooperari: A Tool for Cooperative Testing of Multithreaded Java programs, Eduardo R. B. Marques, Francisco Martins, and Miguel Simões. In: Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages (PPPJ'14), Krakow, Poland, Sept. 23–26, 2014, pp. 200-206, 2014, ISBN 978-1-4503-2926-2, ACM New York, NY.

Palavras-chave: testes de software, semântica cooperativa, threads, Java

Abstract

Software bugs are easy to introduce in multithreaded programs, resulting in well-known errors, such as data races in the use of shared memory among threads, or deadlocks when using multithread primitives like locks. These bugs are hard to detect, replicate, and trace precisely, and are typically manifested only for a subset of all possible thread interleavings, many times even for a very particular thread interleaving. At the same time, given the non-determinism and irreproducibility of scheduling decisions at runtime, it is generally hard to control and observe thread interleaving and explore the associated state-space appropriately, when programming/executing software tests for a multithreaded program.

This thesis presents Cooperari, a tool for deterministic testing of multithreaded Java programs, based on the use of cooperative semantics. In a cooperative execution, threads voluntarily suspend at interference points (e.g., lock acquisition, shared data access), called yield points, and the code between two consecutive yield points of the same thread always executes serially as a transaction. A cooperative scheduler takes over control at yield points and deterministically selects the next thread to run. A program test runs multiple times, until it either fails or the state-space of thread interleavings is deemed as covered by a configurable policy that is responsible for the scheduling decisions. Beyond failed assertions in software tests, some types of deadlock and data races are also detected by Cooperari at runtime.

The tool integrates with the popular JUnit framework for Java software tests, and was evaluated using standard benchmark programs. Cooperari can find, characterize, and deterministically reproduce bugs that are not detected under unconstrained preemptive semantics.

Keywords: Software testing, Cooperative semantics, Threads, Java

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The still-prevalent programming model for enabling concurrency in software programs at large is the use of threads. A thread defines a sequential line of execution in a program, that shares a common memory address space with all other threads in the same program that execute concurrently. Beyond shared memory, threads in a multithreaded program communicate using a number of well-established primitives such as locks, barriers, or atomic memory updates. The thread model is supported by some of the most well-known and used programming languages, like Java or C.

Programming mistakes are easy to make in multithreaded programs. The use of shared memory and other multithreading primitives requires careful attention and craft by a software developer. For instance, data races or deadlocks in the use of multithreading primitives are well-known problems. When a bug is manifested, a software developer typically tries to manually debug the program at stake and/or execute test suites associated to the program and observe which tests happen to fail. Frequently, both of these approaches cannot uncover bugs.

The reason for these difficulties is that a multithreaded system's scheduler typically performs context switches in a non-deterministic and irreproducible manner. As a result, it becomes impossible to control and observe thread interleaving appropriately. At the same time, multithreaded bugs are typically manifested only for a subset of all possible schedules, frequently relying on a very particular thread interleaving. Software tests may be repeated many times without exposing the bug (i.e., reproducing the schedules of interest). If and when the bug is exposed, it is in any case hard to trace the associated program behaviour and replicate it again. Also, the use of a debugger typically interferes with the program schedule itself, making some bugs, so-called "heisenbugs", also hard to observe and reproduce manually. Even simple bug patterns may be elusive to detect and replicate precisely (e.g., see [9]).

1.2 Problem statement and approach

This thesis deals with the general problem of testing multithreaded programs written in the Java programming language [16], such that program bugs are uncovered in deterministic/reproducible manner, and can also be traced in program execution without ambiguity.

Our approach is to conduct software tests that execute with cooperative semantics [23, 24]. The key observation underlying cooperative semantics is that a context switch can be made at any point during program execution, but the relevant context switches are only those that cause thread interference, e.g., access to shared data or the use of multithreading primitives, so-called cooperative yield points. A program's execution is cooperative if the code between two yield points of the same thread executes serially as a transaction without any interference from other threads.

Cooperative semantics may be exploited for software testing to attain a reproducible and systematic coverage of the state-space of thread schedules, e.g., as in a number of tools [17, 3, 7] that work for multithreaded C/C++ programs. The idea is to schedule threads cooperatively, such that (deterministic) context switches are only done at yield points. Repeated executions of a program (test) may then potentially explore the state-space of thread schedules in a customized manner.

1.3 Contributions

The general contribution of this thesis is the design, implementation and evaluation of a tool for cooperative testing of multithreaded Java applications. The tool is called Cooperari and is available for download at https://bitbucket.org/edrdo/Cooperari. To the best of our knowledge, it is the first cooperative testing tool for the Java programming language. In detail, the contributions are summarized as follows:

- Cooperari instruments Java application byte code such that yield points defining thread interference are intercepted and executed cooperatively. The instrumentation of yield points is specified using AspectJ [14].
- For cooperative execution, a runtime system is defined comprising a cooperative scheduler, and a cooperative implementation of multithreading primitives of Java monitors and thread lifecycle operations.
- Runtime monitoring mechanisms have been implemented for the detection of data races and some forms of deadlock.
- The tool integrates with JUnit [20], the popular testing framework for Java. Test programmers can use the traditional JUnit framework with little extra effort.

- Two coverage policies for the exploration of the state-space of thread schedules have been implemented: a memoryless pseudo-random choice of threads, and a history-dependent policy that maintains a record scheduling decisions that persists across test trials,
- The tool has been evaluated using standard benchmark examples from literature.

In significance of these contributions, the following peer-reviewed publication describing Cooperari was presented at an international conference during the thesis period:

Cooperari: A Tool for Cooperative Testing of Multithreaded Java programs, Eduardo R. B. Marques, Francisco Martins, and Miguel Simões. In: Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages (PPPJ'14), Krakow, Poland, Sept. 23–26, 2014, pp. 200-206, 2014, ISBN 978-1-4503-2926-2,ACM New York, NY.

1.4 Thesis structure

The remainder of this thesis is structured as follows:

- Chapter 2 ("Background and related work") provides background on Java multithreading and multithreaded program bug patterns, and surveys related work.
- Chapter 3 ("The Cooperari testing tool") describes the main features of Cooperari from a user's perspective.
- Chapter 4 ("Design and implementation") describes the design and implementation of Cooperari.
- Chapter 5 ("Evaluation") provides an evaluation of Cooperari using benchmark program examples.
- Chapter 6 ("Conclusion") makes a final discussion and highlights directions for future work.

Chapter 2

Background and related work

2.1 Multithreaded Java programs

In this section we provide an overview of the core multithreading primitives in the Java language and API [16], as defined by the java.lang.Thread and java.lang.Object classes, plus the built-in support for locks using synchronized blocks.

Multithreaded programming introduces new kind of errors, such as race conditions, deadlocks, and memory consistency. To avoid them we need to properly synchronize our Java objects or methods to allow mutual exclusive access of the critical section to two threads.

The Java scheduler is typically preemptive. Preemptive schedulers are driven by the notion of prioritized computation, i.e the thread with the highest priority should always be the one currently using the processor. If the scheduler has a timer interrupt it will decide which thread is next to run non-deterministically and resume that thread for some constant period of time. When the thread has executed for that time period it will be suspended, i.e. originates a context switch, and the next thread scheduled will be resumed.

In Fig. 2.1a we have a implementation of a semaphore. A semaphore represents a non-negative integer that can only be atomically incremented or decremented, using up() and down() respectively. Lines 6–9, 12, 14 are synchronized blocks to the object. Before entering their critical section, the thread needs to acquire the lock on the object. The critical section will be only executed by one thread at a time for the object that is locked, ensuring that we have mutual exclusion. Synchronization is also used to prevent simultaneous access by multiple threads to the same data, for example the access to the variable value (lines 7, 8 and 12).

Line 9 is the termination of a synchronized block, when the thread reaches this point, the lock on the object at stake is released. The lock can also be released if any error or exception occurs. After the lock release, if there are other threads waiting to acquire that lock, one of them is subsequently chosen non-deterministically by the JVM and acquires the lock.

```
1  class Semaphore {
2    private int value;
3    Semaphore(int initial) {
4      value = initial ;
5    }
6    int getValue() {
7      return value;
8    }
9    void down() throws
           InterruptedException {
10     synchronized (this) {
11       while (value == 0) {
12         wait();
13       }
14       value = value - 1;
15     }
16   }
17   void up() {
18     synchronized (this) {
19       value = value + 1;
20       if (value == 1) {
21         notify();
22       }
23     }
24   }
25 }
```

(a) Semaphore implementation

```
1  static class Client extends Thread {
2    private Semaphore sem;
3    Client(Semaphore sem) { this.sem = sem; }
4    public void run() {
5      try {
6        sem.down();
7        // do some work ...
8        sem.up();
9      }
10     catch(InterruptedException e) { }
11   }
12 }
13   static final int N = ...;
14   @Test public final void test() {
15   Semaphore sem = new Semaphore(N-1);
16   Client [] c = new Client[N];
17   for (int i=0; i < N; i++) {
18     c[i] = new Client(sem);
19     c[i].start();
20   }
21   for (int i=0; i < N; i++) {
22     c[i].join();
23   }
24   assertEquals(N-1, sem.getValue());
25 }
```

(b) Semaphore test

Figure 2.1: Semaphore example

The purpose of condition based methods, wait() (line 7) and notify() (line 14) is to coordinate access to synchronized code blocks between two threads that require each other to perform some functionality. In the example, this means that if the value of the semaphore is 0 then the thread needs to wait until the value is bigger then 0 (line 7). The notification is only made (line 14), if the semaphore value is equal to 1 (line 13).

When the wait() (line 7) is executed it releases the lock on the object, and suspends the thread until a notification is delivered by other threads for the object through notify(), notifyAll(), or spurious wakeup [16] event takes place. When the notify() (line 14) is made, one of the threads that are suspended in the wait call will be resumed, this choice is made non-deterministically by the JVM.

The test shown in Fig. 2.1b creates a semaphore with an initial value of N−1 (line 15), that is shared by N Client threads (line 18). When a thread starts its execution (line 19), it will execute the run() method (line 4). All critical section (line 7), is surrounded by a semaphore (line 6 and 8), the semaphore is used to guarantee mutual exclusion.

At the end of the test, a join() (line 23) is made, to make the test to wait until all threads terminate their execution, and then an assertion is made to verify if the semaphore's value is equal to the initial one N−1 (line 20), if so the test passes. Otherwise, the test fails giving a assertion error.

2.2 Bug patterns

```
void up() {
  value = value + 1;
  if (value == 1) notify () ;
}
```

(a) Unsynchronized data access

```
void up() {
  value = value + 1;
  synchronized (this) {
    if (value == 1) notify () ;
  }
}
```

(b) Partially unsynchronized data access

```
void up() {
  synchronized (this) {
    value = value + 1;
  }
  synchronized (this) {
    if (value == 1) notify () ;
  }
}
```

(c) Two-stage access pattern

```
void up() {
  synchronized (this) {
    value = value + 1;
  }
  if (value == 1)
    synchronized(this) {
      notify () ;
    }
}
```

(d) Two-stage access and unsynchronized access

Figure 2.2: Multithreaded bug examples

In Figure 2.2, we illustrate some multithread bugs with some variations of the up() method of the semaphore example back from the previous section.

In (a), the up() method does not acquire any lock. This may result in data races that will render the semaphore's value inconsistent. Furthermore, if the notify () method is called, it will necessarily throw an exception, since calling this method requires a lock on the target object. Fragment (b) fixes the later issue, but still lets the semaphore value be accessed without any synchronisation.

In (c), we have a more subtle bug, known as a two-stage access bug pattern [9]. The operations are synchronized, but in two steps. Imagine the case where the semaphore value is 0, and one thread in hanging on down(), Now, two other threads may execute the first synchronised step and take the semaphore value from 0 to 2, if they yield just before entering the second synchronized step. The fragment in (d) a variation of (c), with an additional unsynchronised access to to the semaphore value.

Simple bug patterns like this have been characterised in literature, e.g., see [9] for a taxonomy of concurrent bugs patterns that occur in practice. Some buggy code may execute for quite some time without notice but are eventually manifest—this is actually the case of (a) and (b)— while others are very hard to occur and detect—the case of (c) and (d). In Chapter 3, we present the use of Cooperari over example (d) in detail.

2.3 Cooperative semantics

The use of cooperative scheduling rests on the premise of semantics preservation. An execution is cooperative if it ensures that the code between two yield points of the same thread executes serially as a transaction without any interference from other threads. The execution of a program under cooperative semantics should be equivalent to that obtained using a traditional preemptive scheduler. This happens if the considered yield points characterize all possible thread interference, a property Yi et al. call cooperability [24, 23]. The authors define a formal framework to reason on cooperative semantics of Java programs, along with tools COPPER and SILVER for formal analysis of programs.

COOPER is a dynamic analysis tool that detects cooperability violations by observing an execution trace of the target program, and uses a graph-based algorithm to verify that this observed trace is serializable. It reports an error if the yield annotations are not sufficient to capture all thread interference. After that, it verifies if the program is correct using cooperative reasoning. To place yield annotations automatically in the program, they developed SILVER, a yield inference tool. One of SILVER main objectives is to insert yield annotations as less as possible, to do so it only adds one when it is strictly necessary, to reduce the number of possible executions paths for analysis. .

2.4 Cooperative testing tools

CHESS [17] is a tool for multithreaded Windows programs. It operates as a stateless model checker, enumerating all possible thread schedules, while a test repeatedly executes. CHESS relies on thin wrappers for multithreading primitives to identify nondeterministic choices at yield points, and various techniques to curb state-explosion such as iterative preemption bounding.

Cloud9 [3] is a parallel symbolic execution engine for multithreaded C POSIX programs. During state-space exploration for symbolic assertion checking, Cloud9 relies on cooperative scheduling and a cooperative/symbolic implementation of a portion of the POSIX system calls, in particular for the pthreads API.

CONCURRIT [7] is a tool for C++ multithread programs, employing a DSL for imposing thread schedule constraints. Tests execute repeatedly, with the guidance of a model checker that covers all schedules defined by the DSL constraints. The execution is cooperative, relying on yield calls at execution points in user-level code specified by the DSL (specific details are given in [5]).

Compared with these tools, Cooperari works for Java programs, and allows a partial exploration of possible thread schedules through a deterministic pseudo-random search and a structural program coverage criterion. Like CHESS and Cloud9, Cooperari enables cooperative execution of multithreading primitives, while CONCURRIT just works for user-placed yield points.

2.5 Randomized scheduling

Randomized scheduling works by causing interference to the native multithreaded scheduler, trying to "shake" it to produce meaningful context switches in hotspots where bugs may occur. The base technique can be effective to expose bugs, but works non-deterministically.

Rstest [22], transforms a Java program, by inserting calls ("noise") with a scheduling function at selected points. The scheduling function either does nothing or causes a non-deterministic context switch. They use static analysis to reduce the number of inserted calls to the scheduling, and it does not reduce the probability of detecting deadlocks and assertion violations. Rstest analyses bytecode and monitors for violations of the classification of objects as unshared, i.e., if the object is accessed by at most one thread. Rstest does not have heuristics.

ConTest [6, 15, 18], inserts calls to the scheduling function at all concurrent events, in particular, the events whose order determines the result of the program. ConTest executes sleep instructions before and after concurrent events. ConTest contains two tools: Erase+ is able to detect data races; AtomRace is able to detect not only data races, but also more general bugs cause by violation of atomicity preemptions. ConTest developed a plug-in that is able to heal bugs (that remain in an application even when it is deployed), automatically at runtime. The plug-in does not remove bugs from the code but it prevents their manifestation.

PCT [4] is a randomized scheduler for finding concurrency bugs, it uses a randomized algorithm for concurrency testing. PCT repeatedly runs a given test program with supplied inputs. In each test run, it randomly schedules the threads of the program. All the runs are independent, and so it can increase the probability of finding bugs.

CALLFUZZER [13] employs biased random scheduling to verify if warnings reported by a predictive program analysis are real bugs, a technique known as active testing. It works in two phases: first uses static or dynamic analysis to identify potential concurrency bugs, in the second phase active testing uses a randomized thread scheduler to verify if warnings reported by a predictive program analysis are real bugs, to take care of them they implemented a Hybrid data race, Atomizer algorithm to detect atomic violations and iGoodlock to detect deadlocks. CALFUZZER pauses a thread when it reaches a statement involved in a potential concurrency bug and determines if this statement is benign or harmful (e.g., causes an exception). The tool is available for Java language.

2.6 Other approaches

MultithreadedTC [19] is a framework that allows a test designer to exercise a specific interleaving of threads in an application. It features a personal clock (not the computer clock) that allows test designers to coordinate threads even in the presence of blocking and

timing issues. The clock advances only when all threads are blocked; test designers can delay operations within a thread until the clock has reached a desired tick. IMUnit [12] achieves the same effect as MultithreadedTC, but through annotations that define a kind of temporal logic, and works for Java. The tool also incorporates a translation tool to convert Java sleep-based tests into event-based tests.

Chapter 3

The Cooperari testing tool

In this chapter we provide an overview of the testing framework provided by the Cooperari tool. We start with a description of the Cooperari framework and its main features (Section 3.1). We then discuss the use of Cooperari resorting to two examples (Section 3.2).

3.1 The Cooperari framework

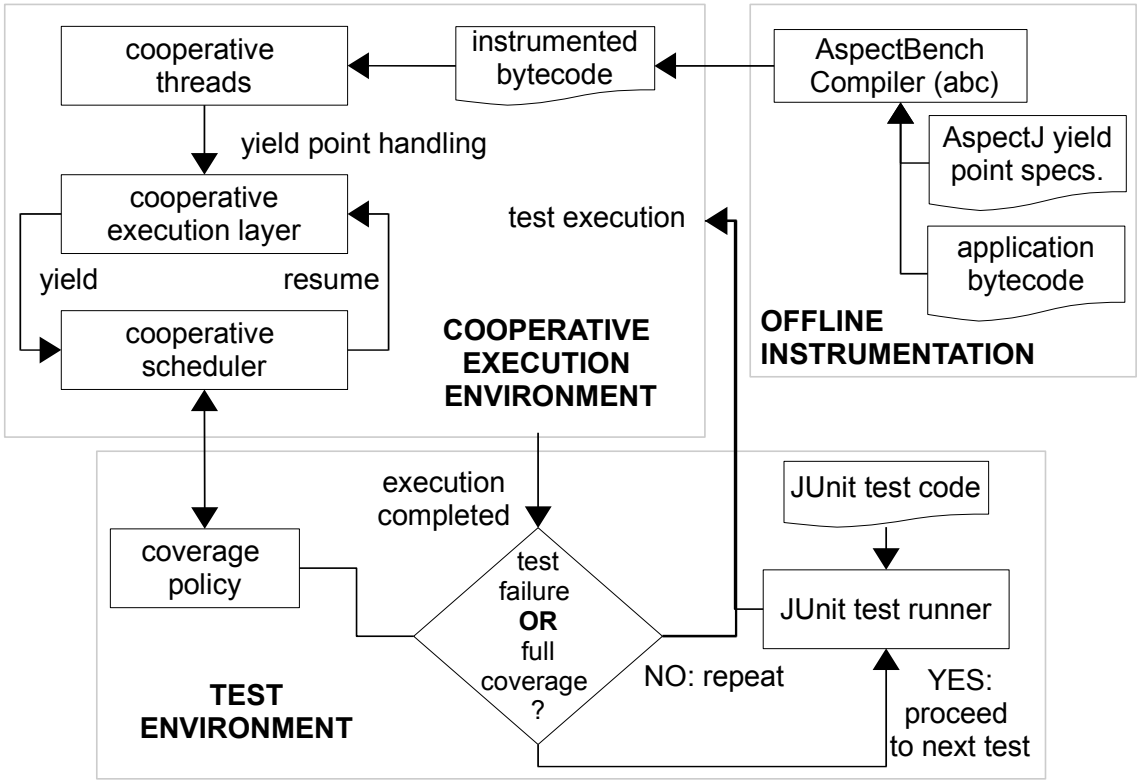

Figure 3.1: The Cooperari framework

The overall architecture of Cooperari is depicted in Figure 3.1. It comprises the fol-

lowing three main parts :

1. the offline instrumentation of JVM bytecode in support of cooperative execution;
2. a runtime execution environment that is responsible for handling the actual execution of an application with cooperative semantics;
3. and a testing environment that triggers the execution of application tests.

Next, we provide an overview of how these parts work and relate to each other.

3.1.1 Yield point instrumentation

To attain cooperative execution, Cooperari instruments the JVM bytecode of Java classes that are of interest. The instrumentation takes care of transforming potential thread interference points into thread yield points for cooperative execution.

To accomplish instrumentation, Cooperari makes use of AspectJ [14], in particular of the AspectJ AspectBench Compiler (abc) [1], as illustrated in Figure 3.1, top-right. The basic idea is to define (aspect-oriented style) code that is triggered when a thread interference point is reached. The instrumented code will then seek to render the program execution cooperatively. In what concerns the Cooperari user, the instrumentation is automatic, transparent, and no AspectJ knowledge is required.

The yield points defined through bytecode instrumentation overall comprise:

- **Java lock/monitor operations**: lock acquisition, lock release, plus calls to `wait()`, `notify()`, and `notifyAll()` in class `java.lang.Object`;
- **Thread lifecycle methods**: methods that affect the behavior or govern the interaction of threads in class `java.lang.Thread`, including `start()`, `stop`, `join`, `interrupt`, and `sleep`.
- Read or write accesses to object fields or arrays;

3.1.2 Cooperative execution environment

Cooperari enables a special execution environment for application code to run cooperatively, shown in Figure 3.1 (top-left). The main components of this environment are a set of cooperative threads, a cooperative execution layer, and a cooperative scheduler.

The cooperative threads runs the application code that has been instrumented. When a thread reaches a yield point, instrumented bytecode gets executed such that control is delegated to the cooperative execution layer in place of normal execution. The two core actions performed at this point by the cooperative execution layer are causing the current thread to voluntarily stop execution (yield) and invoking the cooperative scheduler.

The cooperative scheduler will then select the thread to be resumed next, according to a specified policy, discussed further on. This type of operation guarantees a cooperative execution: only one thread is active at any time, and that the code between two yield points executes serially as a transaction.

In complement to the core tasks for cooperative execution, the cooperative execution layer also embeds monitoring modules for detecting deadlocks and race conditions. Their use is discussed in the discussion of the examples in Section 3.2.

3.1.3 Testing environment

The Cooperari framework is completed by a testing environment (Figure 3.1, bottom), whose role is to execute application tests, written using the JUnit testing library for Java.

Each individual test trial triggers the launch of a cooperative execution environment, corresponding to one execution of application code. Trials of each test are repeated until either a trial fails (e.g., due to a failed test assertion, or to a detection of a deadlock) or the configurable coverage policy determines that no further trials should be executed. The coverage policy configured for a test suite also defines the scheduling function to be used by the cooperative scheduler. At this point we implemented two policies: a pseudo-random (deterministic) choice of thread to run coupled with a bound on the maximum number of test executions, and a history-dependent policy that tries to avoid repeated scheduling decisions.

3.2 Using Cooperari

3.2.1 Dining Philosophers

We first discuss the use of Cooperari with the classic dining philosophers problem. The formulation is well-known, and illustrates deadlocks due to resource sharing in a concurrent setting. A group of N philosophers sits at a round table where N forks are placed in between each plate. To eat, a philosopher has to have both forks, first he grabs his left fork and then his right one. After eating, a philosopher puts both forks down. The philosophers get stuck if they all simultaneously grab the left fork, leaving no right fork available to be picked by anyone. No philosopher may thus progress, defining a deadlock.

In line with the dining philosophers' formulation, the code in Figure 3.2 defines a Philosopher Java class, and a TestPhilosophers test suite.

In Figure 3.2a, we can see that a Philosopher object is created using supplied left and right fork objects, and that the run() method defines the behavior of the philosopher. To grab a fork, a philosopher acquires a lock to the corresponding object, making use of a **synchronized** block within the run() method. The fork acquisition is defined at lines 12–16 for the left fork and 13–15 for the right fork. The two **synchronized** blocks are nested,

```
1  package philosophers;
2  class Philosopher implements Runnable {
3   private Object left , right ; // forks
4   private boolean thoughts = false;
5   private boolean food = false;
6   Philosopher(Object left , Object right ) {
7    this. left = left ;
8    this. right = right ;
9   }
10  public void run() {// 1. think
11   thoughts = true;
12   synchronized(left) {// 2. get left fork
13    synchronized(right) {// 3. get right
14     food = true;// fork and eat
15    } // 4. release right fork
16   } // 5. release left fork
17  }
18  boolean hadThoughts() { return thoughts;}
19  boolean hadFood() { return food; }
20 }
```

(a) Philosopher code

```
1  @RunWith(CTestRunner.class)
2  @CTestOptions(coverage=RANDOM,
       instrument="philosophers")
3  public class TestPhilosophers {
4   static final int N = ...;
5   @Test public void testDinner() {
6    Object f[] = new Object[N];
7    for (int i = 0; i < N; i++)
8     f[i] = new Object();
9    Philosopher[] p = new Philosopher[N];
10   for (int i=0; i < N; i++)
11    p[i] = new Philosopher(f[i], f[(i+1)%N]);
12   runThreads(p);
13   for (int i=0; i < N; i++) {
14    assertTrue(p[i].hadThoughts());
15    assertTrue(p[i].hadFood());
16   }
17  }
18 }
```

(b) Dining philosophers test

Figure 3.2: Dining philosophers example

implying that the right fork is only acquired after the left one, and, in the end, that the right fork is released before the left one. If the code in run() runs to completion, the thoughts and foods fields, initially set to **false**, will have been set to **true**.

The code of Figure 3.2b is that of a JUnit test class, TestPhilosophers, with a test method called testDinner. The test method first defines the round table setup comprising N forks and N philosophers (lines 7–11), represented by variables f and p, respectively. After this setup, a call to runThreads(p) is made to execute the N philosopher threads; runThreads is an utility method in the Cooperari API that performs the launch of a set of threads, followed by a wait for the termination of all of them. When all the threads complete, a sequence of JUnit-style assertions (14–16) verifies that each philosopher had a round of thought and food.

A normal execution does not uncover the dining philosophers deadlock in a vast majority of executions. For example, we observed that a deadlock is reached only once every 20000 executions of testDinner for N=2, and failed to observe deadlock at all for N=3.

To execute the dining philosophers test cooperatively, the TestPhilosopher class must be appropriately annotated. The @RunWith(CTestRunner.**class**) annotation (line 1) sets JUnit to use Cooperari's custom test runner (CTestRunner). In complement, the @CTestOptions annotation (line 2) defines options for cooperative execution: the coverage =RANDOM option specifies that pseudo-random, deterministic scheduling decisions should be made at yield points, and the instrument="philosophers'' option indicates the Java package whose classes should be instrumented for cooperative execution. The Cooperari test runner will repeatedly execute each test in the suite, testDinner alone in this case, until it either fails or reaches a maximum number of trials (by default 1000). The output of an

execution is as follows, considering N=3 in the test code.

```
Instrumented code must be generated, please wait.
Changes have been detected.
Instrumentation completed in 5486 ms.
testDinner: executed 3 times in 68 ms [failed]
Failure trace for testDinner written to
'log/TestPhilosophers_testDinner.trace.txt'
There was 1 failure:
1) testDinner(TestPhilosophers)
DeadlockError: L0/T0/Philosopher.java:13
> L1/T2/Philosopher.java:12
> L2/T0/Philosopher.java:12
> L0/T1/Philosopher.java:12
```

The execution above starts by performing bytecode instrumentation. This takes more than 5 seconds, but the instrumentation will be triggered just once, so long as the source bytecode does not change. The actual test execution is done afterwards, reporting a failure after 3 trials of testDinner in 68 milliseconds. A DeadlockError exception is reported, referring to a lock acquisition cycle involving the three threads, identified as T0/1/2, and three fork objects, identified as L0/1/2.

The test execution output also informs the location of a cooperative trace file, where the detailed thread schedule can be inspected. The contents are shown in Figure 3.3. In the listing, the lock acquisition yield points are identified by monitorenter, the name of the JVM bytecode instruction that is executed for that purpose. In the beginning, all threads are initially suspended (steps 1–3). After startup, thread 1 is allowed to run for two steps (4–5): the first step takes the thread to the point of left fork acquisition, and the second one (with the left fork lock now effectively acquired) to the point of right fork acquisition. Thread 1 then yields, and the same pair of steps is allowed in succession for threads 2 (6–7) and 0 (8–9). A deadlocked state is thus reached, as every philosopher holds the left fork, but neither is able to acquire the right one.

```
<step> <thread> <yield point>
1 0 [begin]
2 1 [begin]
3 2 [begin]
4 1 monitorenter(L0) Philosopher.java:12
5 1 monitorenter(L2) Philosopher.java:13
6 2 monitorenter(L1) Philosopher.java:12
7 2 monitorenter(L0) Philosopher.java:13
8 0 monitorenter(L2) Philosopher.java:12
9 0 monitorenter(L1) Philosopher.java:13
```

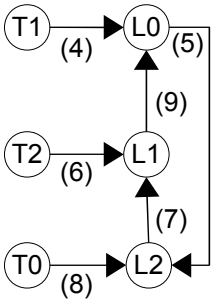

Figure 3.3: Cooperative execution for the dining philosophers

3.2.2 Semaphore bug example

We now go back to the semaphore example, introduced in Chapter 2. The example concerns a bug in the semaphore implementation in method up(), actually variant (d) method in the Figure 2.2 in Section 2.2. We show the full (but this time buggy) semaphore

class again and an associated test in Fig. 3.4. The example will illustrate different types of yield points and deadlocks, as well as data race detection by Cooperari. The test shown creates a semaphore with an initial value of N−1, shared by N Client threads. Each thread proceeds (in the run() method) by decrementing the semaphore, doing some work, and incrementing the semaphore back before terminating. At the end, the test passes if the semaphore's value is equal to the initial one, N−1.

```
1  class Semaphore {
2   private int value;
3   Semaphore(int initial) { value = initial ; }
4   int getValue() { return value; }
5   void down() throws InterruptedException {
6    synchronized (this) {
7     while (value == 0) { wait(); }
8     value−−;
9    }
10  }
11  void up() {
12   synchronized (this) { value++; }
13   if (value == 1) {
14    synchronized (this) { notify(); }
15   }
16  }
17 }
```

(a) Buggy semaphore

```
1  static class Client implements
       Runnable {
2   private Semaphore sem;
3   Client(Semaphore sem) { this.sem =
       sem; }
4   public void run() {
5    try {
6     sem.down();
7     // do some work
8     sem.up();
9    }
10   catch(InterruptedException e) { }
11  }
12 }
13 static final int N = ...;
14 @Test public final void test() {
15  Semaphore sem = new
       Semaphore(N−1);
16  Client [] c = new Client[N];
17  for (int i=0; i < N; i++) c[i] = new
       Client(sem);
18  runThreads(c);
19  assertEquals(N−1, sem.getValue());
20 }
```

(b) Semaphore test

Figure 3.4: Buggy semaphore example and test

As explained earlier in Section 2.1 the semaphore class employs the condition-based wait() and notify() methods associated to Java monitors, the core synchronization primitives used by Java applications and thread-safe Java API classes (along with notifyAll()).

In the semaphore code of Fig 3.4 (a), the up() operation only calls notify() when the semaphore increments to 1 (lines 13–14). The code would be correct if a single **synchronized** block covered all instructions in up(). Instead, two blocks of the kind are used, a "two-stage access" pattern [9]. Moreover, the notification event relies on an unsynchronized read access to value (line 13), which may race with simultaneous write accesses. It is then possible that two or more increments in up() take the semaphore value from 0 to a value greater than 1. A required notification, in case some thread is blocked in down(), may be skipped and, as a result, a waiting thread may block forever. Skipped notifications would also be possible if the read access to value was part of the second **synchronized** block, but the purpose of the example is also to illustrate data races.

Cooperari detects the race condition and the wait deadlock, considering the yield points defined by lock acquisition and release, wait(), notify(), and the read/write ac-

cesses to the semaphore value. The output of an execution for N=3 is as follows:

```
Race: T0 at Semaphore.java:12 over Semaphore.value
Race: T1 at Semaphore.java:13 over Semaphore.value
Failure trace for test written to
    'log/examples.semaphore.TestSemaphore_test.trace.txt'
test: executed 36 times in 578 ms [failed]
1) test(examples.semaphore.TestSemaphore)
   WaitDeadlockError: { T2/Semaphore.java:7 }
```

The output reports a race over the semaphore field and a test failure due to a WaitDeadlockError for a thread identified as T2. A fragment of the cooperative trace is shown in Figure 3.5, where each step is annotated with pending reads and writes (r/w) for the semaphore value. In the execution at stake, thread 0 and 1 (T0, T1) completed down(), thus value will be 0, and began executing up(), whilst thread 2 (T2) is executing down(). At step 23, T0 and T1 are suspended at the first lock acquisition point in up(), and T2 is blocked at the call to wait() in down(). In steps 23–27 and 28–31, T0 and T1 are in succession able to acquire the lock, increment the semaphore value, relinquish the lock, and suspend again before reading value. When they do, just before terminating (steps 32 and 33), value is 2, hence they fail to deliver the notification to T2. Thus, T2 will block forever on wait(). The deadlock is detected at this point. As for the race, it is signaled at step 29, for a pending write by T0 and a pending read by T1.

```
<step> <thread> <yield point>                  # r/w
15 0 monitorenter(L0) Semaphore.java:12        # {}/{}
 ...
22 1 monitorenter(L0) Semaphore.java:12        # [value=0]
23 2 wait(L0) Semaphore.java:7                 #
24 1 get(Semaphore.value) Semaphore.java:12 # 1/{}
25 1 set(Semaphore.value) Semaphore.java:12 # {}/1
26 1 monitorexit(L0) Semaphore.java:12         # [value=1]
27 1 get(Semaphore.value) Semaphore.java:12 # 1/{}
28 0 get(Semaphore.value) Semaphore.java:12 # 0,1/{}
29 0 set(Semaphore.value) Semaphore.java:12 # 1/0 [race]
30 0 monitorexit(L0) Semaphore.java:12         # [value=2]
31 0 get(Semaphore.value) Semaphore.java:13 # 0,1/{}
32 0 <end> # {}/{};read 2; no call to notify ()
33 1 <end> # read 2; no call to nofity ()
```

Figure 3.5: Cooperative execution for the semaphore test

Chapter 4

Design and implementation

4.1 Yield point instrumentation

This chapter describes the main aspects in the design and implementation of Cooperari. We first describe the instrumentation of yield points for cooperative execution (Section 3.1.2). We then turn to the main components of the runtime system for cooperative execution: the cooperative scheduler (Section 4.2), the cooperative implementation of multithreading primitives (Section 4.2.3), and state-space exploration during test execution (Section 4.3). The chapter ends with a description of the runtime detection mechanisms for deadlocks (Section 4.4.1) and races (Section 4.4.2).

As overviewed in Chapter 3, Cooperari employs AspectJ specifications for specifying yield points and instrumenting corresponding application bytecode. We do so in conjunction with the AspectBench Compiler (abc) [1]. The motivation for using abc is the built-in support by the tool for AspectJ extensions related to lock acquisition and release instructions, described in [2]. We next describe the general pattern of instrumentation and the instrumentation of several types of yield points.

4.1.1 Instrumentation pattern

Let us first introduce some aspect-oriented programming terminology to explain the use of AspectJ. AspectJ code is organized in *aspects*, each of which may contain a collection of *advices*. An advice intercepts the execution of program at special points, called *join points*, and induces the execution of some code in association to the advice. For this, the specification of an advice comprises a set of *pointcut* conditions to intercept join points, and a block of code that is executed whenever a join point is intercepted.

The pattern of instrumentation employed by Cooperari is illustrated by the AspectJ code listed in Figure 4.1. Three advices are shown for: lock acquisition, specified by pointcut lock() (lines 1–5); the second for lock release, specified by pointcut unlock() (lines 6–10); and calls to the Thread.holdsLock() (11–16). In the first two advices, the lock() and unlock() pointcuts employed are AspectJ extensions for intercepting lock ac-

```
1  void around(Object o) : lock() && args(o) {
2    CThread t = getCThread(thisJoinPoint, o);
3    if (t != null) t.cMonitorEnter(o);
4    else proceed(o);
5  }
6  void around(Object o) : unlock() && args(o) {
7    CThread t = getCThread(thisJoinPoint, o);
8    if (t != null) t.cMonitorExit(o);
9    else proceed(o);
10 }
11 boolean around(Object o) :
12   call (boolean Thread.holdsLock(Object)) && args(o) {
13   CThread t = CThread.intercept(thisJoinPoint, o);
14   if (t != null) return t.cHoldsLock(o);
15   else return proceed(o);
16 }
```

Figure 4.1: Sample AspectJ instrumentation code

quisition/release instructions [2]. In all three cases, the pointcut specifications are complemented by the use of the around keyword, specifying that advice code should run in place of the join point, i.e., program execution will be diverted to the advices' code in place of the original one that associates to the join points[1].

Looking at the advices' code in Figure 4.1, we see that they follow the same pattern. Each of them first determines if the current thread is subject to cooperative semantics, through a call to getCThread(). The arguments given to this function are used only to initialize profile information of the yield point, if one is at stake. If the current thread is subject to cooperative semantics (t != **null**), then execution will be diverted to a call in the Cooperari runtime, e.g., cMonitorEnter() (line 3), the first advice will execute in place of the monitorenter JVM instruction for lock acquisition. Otherwise (if t == **null**), the proceed AspectJ keyword (lines 4, 9, 15) specifies that the join point should execute normally. The latter case is due to the possibility that instrumented code may run non-cooperatively at some stages of execution, particularly within a JUnit runner thread before or after the invocation of method runThreads(), responsible for creating a cooperative thread environment.

4.1.2 Lock/monitor yield points

Table 4.1 shows the defined aspects for monitor acquisition, release, and condition-based synchronization.

The Object.wait(**long**) and Object.wait(**long**,**int**) call is executed within the cooperative framework, but not deterministically, because its completion depends on the elapsed time measured by the JVM. In complement, the tool intercepts thread state methods, like Thread.holdsLock(), to maintain the cooperative semantics coherent, even if this call is not a yield point.

[1] Among several other AspectJ variations, it is also possible to trigger advice code before or after a join point is reached.

Yield point		Pointcut definition
monitorenter JVM instruction		lock()
monitorexit JVM instruction		unlock()
Thread.holdsLock()		call (**boolean** Thread.holdsLock(Object))
Object.wait	Object.wait()	call (**void** Object.wait())
	Object.wait(**long**)	call (**void** Object.wait(**long**))
	Object.wait(**long**,**int**)	call (**void** Object.wait(**long**, **int**))
Object. notify ()		call (**void** Object. notify ())
Object. notifyAll ()		call (**void** Object. notifyAll ())

Table 4.1: Lock/monitor yield points

4.1.3 Thread-lifecycle yield points

Thread lifecycle methods, are methods that affect the behavior of the threads. Thread.start(), Thread.stop(), Thread.join() and Thread.interrupt() are the core methods for thread creation and thread interruption. Table 4.2 shows the aspects defined to cover this methods and how the pointcuts were defined.

As the Object.wait() call; Thread.sleep and Thread.join(**long**), Thread.join(**long**, **int**) also belong to time-based functions, so they do not execute deterministically, because they depend on the elapsed time of the JVM.

Such as Thread.holdsLock(), Thread.getState() call is intercepted to maintain the co-operative semantics coherent.

Yield point		Pointcut definition
Thread.sleep	Thread.sleep(**long**)	call (**void** Thread.sleep(**long**))
	Thread.sleep(**long**,**int**)	call (**void** Thread.sleep(**long**,**int**))
Thread.interrupt()		call (**void** Thread.interrupt())
Thread.interrupted()		call (**boolean** Thread.interrupted())
Thread.isInterrupted()		call (**boolean** Thread.isInterrupted())
Thread.stop()		call (**void** Thread.stop())
Thread.join	Thread.join()	call (**void** Thread.join())
	Thread.join(**long**)	call (**void** Thread.join(**long**))
	Thread.join(**long**,**int**)	call (**void** Thread.join(**long**,**int**))
Thread.getState()		call (Thread.State Thread.getState())

Table 4.2: Thread-lifecycle yield points

4.1.4 Data access yield points

There is a main difference between this yield points and the ones described in sections 4.1.2 and 4.1.3. The difference is that all the other aspects use the keyword around, but in the data access yield points we use the keywords before and after. This keywords refer that before and after a write or read access we have a yield point, to check if in

some point of the program, we have a simultaneous write/read or write/write on the same variable by different threads. In table 4.3, we demonstrate the yields that were defined to cover all the data accesses.

Yield point	Pointcut definition
field read	get()
field write	set()
array read	arrayget()
array write	arrayset()

Table 4.3: Data access yield points

4.2 Cooperative execution environment

4.2.1 Overview

The creation of a cooperative execution environment is triggered by a call to the runThreads() Cooperari API function, typically invoked from a test suite. The call takes an argument, which can either be an array of objects of type java.lang.Runnable. Objects of this kind define method run() with no arguments, which is supposed to define the execution flow of a thread. If the array supplied to runThreads() has length n, then n cooperative threads will be started and executed to completion.

A cooperative execution environment comprises the interaction of cooperative threads launched trough runThreads() with a cooperative scheduler and a cooperative implementation of multithreading primitives. This happens thanks to the bytecode instrumentation, described earlier in Section 4.1, which intercepts thread execution appropriately at yield points.

4.2.2 The cooperative scheduler

The cooperative scheduler is a special thread, whose role is to decide which application thread should be active at any given time. To ensure cooperative semantics, only one cooperative thread can be active at any time, and thread context switches are only performed at instrumented yield points. Hence, a cooperative thread voluntarily suspends its execution when it reaches a yield point. The cooperative scheduler assumes control at this point, deciding the next thread to run, and resuming the execution of the chosen thread. The built-in JVM scheduler is conditioned in this process by the actions of the cooperative scheduler and cooperative threads.

The main implementation of the cooperative scheduling scheme is shown in Figure 4.2. It comprises the implementation of primitives cYield() and cResume() in a class

```
1  private boolean yield;
2  public void cYield() {
3    ... // yielding
4    yield = true;
5    syncYield();
6    while (yield) {
7      LockSupport.park();
8    }
9    syncResume();
10   ... // resumed
11 }
12 public void cResume() {
13   ...
14   yield = false;
15   LockSupport.unpark(this);
16 }
```

(a) Thread yield and resumption

```
1  CoveragePolicy policy;
2  List<CThread> ready;
3  ...
4  public void cStep() {
5    ...
6    syncYield();
7    CThread t =
8      policy.decision(ready);
9    t.cResume();
10   syncResume();
11   ...
12 }
```

(b) Scheduler step

Figure 4.2: Implementation of cooperative scheduling

called CThread 4.2a, representing the operation of cooperative threads, and a primitive cStep in a class CScheduler 4.2b, for the cooperative scheduler operation.

A thread yield is accomplished through a voluntary call to cYield() (Figure 4.2a) by a cooperative thread. The thread starts by enabling a yield condition variable (line 4) and synchronizes with the scheduler with a call to syncYield() (line 5), a barrier handshake. Essentially, this makes the scheduler aware of the thread yield. The thread then calls the LockSupport.park() Java API method, preventing its execution by the built-in JVM scheduler; the JVM scheduler could otherwise break the cooperative scheduling intent.

In synchrony with the thread yield procedure, the cooperative scheduler works as follows within cStep() (Figure 4.2b). It begins by acknowledging a thread yield through a call to syncYield() (line 6). It then employs the coverage policy in place, policy, to select the next thread t to run amongst the ready thread set ready (lines 7–8). Finally, the scheduler executes t.cResume() to resume the chosen thread.

The resumption process in cResume(), executed by the scheduler thread, disables a thread's yield condition (line 14) and allows the thread to be picked up again by the JVM scheduler through a call to the LockSupport.unpark() Java API method (line 15). These two steps are exactly the reverse the thread yield procedure. When the thread resumes again, it will complete the execution of cYield() in which it blocked. The resumption process ends with a synchronization handshake between thread and scheduler, syncResume() in 4.2a and 4.2b.

4.2.3 Cooperative implementation of multi-threading primitives

The multithreaded primitives where implemented in CThread class, helped by a CMonitor class for representing monitors, and a COperation base class that implements the behavior of primitives. Each primitive executes an action before its suspension and another after

resumption.

For instance, the CThread.cMonitorEnter() method (Fig. 4.1) is used by Cooperari for lock acquisition. It is implemented by first initializing a COperation object for the new monitor acquisition, after the initialization a cYield() is invoked to suspend the thread execution and when it is resumed (through cResume()) the thread completes the lock acquisition on the object.

```
1  class CMonitor {
2    // Reference count.
3    int refCount;
4    // Owner thread
5    CThread owner;
6    // Wait count
7    int waitCount;
8    // Notification epoch
9    long nEpoch;
10   // Notification queue
11   Queue<Integer> nQueue;
12 }
```

(a) Monitor data

```
1  void init (Object o) {
2    m = getMonitor(o);
3    m.refCount++;
4  }
5  CState getState() {
6   return
7     m.owner == null ?
8       CREADY : CBLOCKED;
9  }
10 void complete(CThread t) {
11   m.owner = t;
12 }
```

(b) Lock acquisition

Figure 4.3: Implementation of monitor operations

Fig. 4.3 demonstrates part of the support for monitor operations. The CMonitor class (Fig. 4.3a) is used for distinct types of COperation corresponding to monitor acquisition, release, notification, and wait wakeup. In Fig. 4.3b is showed the support for lock acquisition. As illustrated, each operation comprises initialization (lines 1–4), state report (lines 5–9) and completion methods (lines 10–12). A thread can only resume its execution, if the operation is on a ready state (CREADY, line 8). In the lock acquisition operation, a monitor for o is created in the getMonitor(o) (line 2) call, if there are no pending locks on o. If there already exists a monitor on the object o, the reference counter for the monitor object is incremented (line 3). The same counter is decremented in the completion of lock release, and the monitor is deleted when the reference counter gets the value 0. A CREADY state is signaled when the monitor has no owner, letting the thread be considered for execution. The thread may go back to a CBLOCKED state, if another thread is chosen instead by the scheduler and acquires the lock, this happens if there exists more than one thread blocked in the lock acquisition on the same object. In the completion stage of the operation (lines 10–12), the owner of the monitor is the thread that was chosen by the scheduler to run (line 11).

4.3 Test execution and state-space exploration

4.3.1 Overview

When Cooperari is used for a JUnit test suite, the suite must be configured with a coverage policy using a pre-defined value for the coverage setting of the @CTestOptions.

The role of the coverage policy is to guide thread scheduling in each test trial, in interface with the cooperative scheduler, and maintain state across multiple trials of each test to determine for how long test trials should be repeated, according to some criterium of state-space exploration. Two coverage policies are implemented so far in Cooperari: a pseudo-random choice of threads combined with a fixed bound on the number of test trials, and a history-dependent policy that tries to avoid repeated scheduling decisions for the same program state.

Test execution proceeds as follows. After executing a test trial, the Cooperari test runner evaluates if the test failed. If so, no more trials are executed for that test. Otherwise, the coverage policy is queried to determine if the associated coverage criteria has been satisfied. If not, the trial is repeated, otherwise no more trials are executed and the test is considered as passed (no failures will have been detected in this case). This functionality is embedded in Cooperari's extension of the default JUnit test runner.

Meanwhile, during each trial, more precisely during the execution of a cooperative execution environment through runThreads(), the cooperative scheduler will interface with the coverage policy for the test suite at each scheduling step to determine the thread to run at each yield point, the policy.decision() call back in Figure 4.2 (Section 4.2.2).

4.3.2 The random coverage policy

```
1  public class RandomCoveragePolicy implements CoveragePolicy {
2    private final int _maxRuns; // Maximum number of test executions.
3    private int _runs;   // Executions so far.
4    private Random _pRNG; // Pseudo−random number generator.
5    public RandomCoveragePolicy(Class<?> testClass, Method m,
           CTestOptions options) {
6      _runs = 0;
7      _maxRuns = options.maxRuns();
8      _pRNG = new Random(0); // uses a fixed seed for repeatable tests
9    }
10   public void onTestFinished() { _runs++; }
11   public boolean done() { return _runs >= _maxRuns; }
12   public CThread decision(List<CThread> readyThreads) {
13     return readyThreads.get( _pRNG.nextInt(readyThreads.size()) );
14   }
15    ...
16 }
```

Figure 4.4: Code for the random coverage policy

The simplest coverage policy implemented within Cooperari is the random coverage policy, activated using the coverage=RANDOM setting for @CTestOptions. The policy employs a pseudo-random generator for thread scheduling decisions, and a runtime parameter that bounds the maximum number of test trials, configured by the maxRunsPerTest setting in @CTestOptions (1000 by default if omitted). During a scheduling step, a thread is chosen randomly, without any recall of past decisions. The decision is deterministic across test sessions however, since the random number generator is always initialized

with a fixed seed. After each test trial, the number of test trials executed so far is compared with the maximum number of trials to determine if there is no need for more test trials.

The code for the random policy is shown in Figure 4.4. For the most part, the code should be self-explanatory, and illustrates the general traits of the extensible framework for coverage policies. The decision() method is used by the cooperative scheduler to obtain the next thread to run at each yield step. The onTestFinished() and done() methods are respectively used by the Junit runner to notify a test trial conclusion, and query if the coverage criterium has been satisfied. These methods are defined originally in abstract form by the CoveragePolicy Java interface in the Cooperari API. Regarding determinism for the policy at stake, observe that the pseudo-random generator, an instance of java.util.Random, is always initialized with fixed seed 0. This will guarantee a deterministic sequence of scheduling decisions (so long as the remaining system runs deterministically as well, of course).

4.3.3 The history-dependent policy

The second policy implemented in Cooperari is a history-dependent one. This policy maintains a history of scheduling decisions across test trials in order to avoid repetition of behavior for the same program state. The history record works in conjunction with a program state abstraction and an equivalence relation between such program state abstractions, a form of dynamic partial order reduction [10], that helps reducing the state-space of possible thread schedules and tries to avoid scheduling decisions that commute.

The policy works as follows. The state of ready threads is represented by program state $s = \{(n_1, pc_1), \ldots, (n_k, pc_k)\}$, a set where each each $(n, pc) \in s$ defines $n > 0$ threads suspended at location pc, where pc corresponds to the stack trace information obtained via Thread.getStackTrace() augmented with information for the current yield point. History is maintained as a set of pairs (s, pc), where s is an abstract program state and pc is a program location representing a past scheduling decision. At each step, for state s, the policy deterministically tries to find a thread t at location pc such that (s, pc) is not part of the history set. If so, it decides on scheduling t. If all the choices have already been made, then the selection proceeds as in the random coverage policy.

Note that the history-dependent policy cannot guarantee an enumeration of all thread schedules. Firstly, no backtracking mechanism ensures a visit to states where past unexplored scheduling decisions lie. To have a backtracking mechanism, a model checker could integrate orthogonally with the remaining infrastructure of Cooperari, e.g., see [7, 17]. Secondly, the program state abstraction scheme may also filter out relevant thread schedules, given that it is strictly based in structural program information (information for the stack trace and yield points of thread), ignoring data values for instance.

4.4 Runtime monitoring mechanisms

Cooperari uses runtime monitoring to detect two common problems in multithreaded programs: deadlocks and data races. A deadlock occurs whenever one or more threads is hanged on a synchronization point and unable to progress. A data race occurs whenever a thread writes a data item that is being accessed (read or written) simultaneously by another thread. We describe the support for detecting these events within Cooperari.

```
public class DeadlockDetector {
  private final
    ResourceGraph<Monitor> _graph = new
          ResourceGraph<>();
  private final
    IdentityHashMap<CThread,LinkedList<Monitor>>
          _lockChain = new
          IdentityHashMap<>();
  ...
  public void onMonitorEnter(CThread t,
        Monitor m) {
    LinkedList<Monitor> chain =
          _lockChain.get(t);
    if (chain == null) {
      chain = new LinkedList<>();
      chain.add(m);
      _lockChain.put(t, chain);
    } else {
      Monitor from = chain.getLast();
      _graph.addEdge(from, m);
      List<Monitor> deadlock =
            _graph.findCycle(m);
      if ( !deadlock.isEmpty()) {
        // Deadlock detected, error handling
              omitted
        ...
      } else {
        chain.addLast(m);
      }
    }
  }
  public void onMonitorExit(CThread t) {
    LinkedList<Monitor> chain =
          _lockChain.get(t);
    Monitor m = chain.removeLast();
    if (!chain.isEmpty()) {
      _graph.removeEdge(chain.getLast(), m);
    } else {
      _lockChain.remove(t);
    }
  }
}
```

(a) Deadlock detection code

```
public class RaceDetector {
  private final HashMap<Data, Status>
        _monitoring = new HashMap<>();
  ...
  private void beginWrite(Data d) {
    Status status = _monitoring.get(d);
    if (status == null) {
      // No previous info for data item.
      status = new Status();
      _monitoring.put(d, status);
    }
    status.incrementWriters();
  }
  private void endWrite(Data d) {
    Status status = _monitoring.get(d);
    if (status.decrementWriters() == 0) {
      // Call returns reference count (readers
            + writers).
      // If 0, information can be discarded.
      _monitoring.remove(d);
    }
    if (status.raceCondition()) {
      log( new RaceError(CThread.self(),
            d._object, d._key) );
    }
  }
  ...
}
```

(b) Race detection code

Figure 4.5: Runtime monitoring of deadlocks and data races

4.4.1 Deadlock detection

Cooperari is able to detect deadlocks using two mechanisms. The first mechanism consists of a simple check conduced by the cooperative scheduler. Whenever a scheduling

step is invoked, the state of all cooperative threads is inspected. If all these threads are in a blocked state, then by definition there is a deadlock. This mechanism only works if all the threads are really blocked, however a deadlock will not be detected for a strict subset of the threads, except for deadlocks involving lock acquisition in the manner described next.

The second mechanism detects deadlocks due to cyclic lock acquisitions, as in the dining philosophers example of Chapter 3. As locks are acquired and released, a resource graph [11] is maintained to detect any cyclic dependency expressed by lock acquisitions.

The graph works in conjunction with a chain of lock acquisitions maintained per each thread. Edges are added to the graph in the initialization stage of lock acquisitions, and removed in the completion stage of lock releases. For a thread t locking monitor m the monitoring information is updated as follows:

1. If t owns no locks, i.e., the lock chain of t is empty, the edge t $\rightarrow$ m is added to the graph;
2. If the lock chain of t ends with m', i.e., the last lock obtained by t is m', an edge m' $\rightarrow$ m is added to the graph.

In both the two cases above, m is also appended to the lock chain of t. In reverse manner, when t releases m, the added edge and the lock chain's tail are removed. Deadlocks are easily monitored by checking for the existence of cycles in the graph.

In line with the above reasoning, the code listing in Figure 4.5a depicts the relevant fragments for deadlock detection. The lock chain and resource graph are represented respectively by variables _chain and graph. The code executed for lock acquisition and release is defined by methods onMonitorEnter() and onMonitorExit().

4.4.2 Race detection

Cooperative scheduling naturally exposes races conditions, whenever they may happen, as illustrated by the semaphore example in Chapter 3.

For race detection, Cooperari records information about pending read and write yield points for each data item, which can either be an object field or an array position. When a thread yields on a (read or write) data access, we increment a read or write counter for the data item, and the same counter is decremented when the thread resumes. A race is signaled whenever a completing write detects a pending read or write, or when a completing read detects a pending write. The information for a data item is discarded if both read and write counters reach 0, meaning that there are no pending operations for that data item.

The listing in Figure 4.5b shows a fragment of the code that implements the race detection scheme. The listing comprises the methods that are executed whenever a write

access for a data item initiates, beginWrite(), and completes, endWrite(). The code to deal with read accesses is similar.

Chapter 5

Evaluation

This chapter describes the evaluation of Cooperari using a set of multithreaded Java program, mostly taken from the ConTest [8] and the SIR [21] benchmark suites. The examples at stake were previously employed in the evaluation of tools for verification and testing of multithreaded Java programs.

We begin with a summary description of the benchmarks and their preparation for evaluation (Section 5.1). We then provide and discuss the results for the benchmarks (Section 5.2), covering the effort of bytecode instrumentation, the ability of the system to detect bugs versus a normal testing environment, and a comparison between the two coverage policies implemented in the system so far.

5.1 Benchmark programs

The evaluation covered 12 multithreaded program examples, identified in Table 5.1. The examples are from the ConTest suite [8] and the SIR repository [21], as identified in the table, plus the dining philosophers and semaphore examples of Chapter 3. Some ConTest examples are also in the SIR repository, and we used the SIR versions in those cases.

The SIR/Contest test subjects' original code was used, and associated test programs were converted into JUnit tests that employ Cooperari. All tests are parametric in the number of threads. The bugs at stake comprise deadlocks for Bank, Clean, Dining Philosophers, Piper, and Semaphore, and failed test assertions for the remaining examples. Monitor-based synchronization primitives are employed in all benchmarks except for Apache Common Lang, Merge Sort, and Reorder, where plain data races lead to failed test assertions.

Benchmark	Description	Bug(s)
Alarm Clock [21]	A NullPointerException is thrown in some executions due to data races and and bad use of monitors.	Data Race
Apache Common Lang [21]	The hash code of an of object may yield different results for different threads due to a data race.	Data Race
Bank [8]	A bank account object is changed by multiple threads, and the balance becomes inconsistent due do a data race. The threads may also deadlock.	Deadlock
Clean [8, 21]	Bad use of monitor notifications leads to deadlock.	Deadlock
Dining Philosophers	Example from Section 3.2.1.	Deadlock
Linked List [8, 21]	A non-synchronized operation for a linked list operation leads to data inconsistency.	Data Race
Merge Sort [8]	Data races during array sorting conducted by multiple threads.	Data Race
Piper [8, 21]	Producer-consumer simulation of an airplane ticket reservation system. The same seat may be allocated to more than one thread, while others deadlock.	Deadlock
Reorder [8, 21]	Data race between threads leads to an inconsistent state for an object.	Data Race
Semaphore	See Section 3.2.2.	Deadlock
Two Stage [8, 21]	It's a reader and a writer, that share the same data, and the data can be inconsistent.	Data Race
Wrong Lock [8, 21]	A wrong lock is obtained, leading to an unsynchronized access/race for a data value.	Data Race

Table 5.1: Program descriptions

5.2 Results

5.2.1 Test setup

The results for our benchmark evaluation were conducted using a standard Java 7 JVM, with a dual-core 3 GHz CPU and 4 GB of RAM. The times we report were measured through command-line scripts.

5.2.2 Bytecode Instrumentation time

The first set of results concerns the time for bytecode instrumentation in each of the benchmarks, as performed by the abc compiler guided by Cooperari's yield point specifications in AspectJ. These results are shown in Table 5.2, listing the lines of code (LOC) and the instrumentation time in seconds per benchmark, as well as the LOC-time ratio.

Overall, the instrumentation time is less than 30 seconds in all cases, and the time-LOC ratio is between approx. 5 and 15 LOC/s. A yield point count and a yield point-time ratio, in place of LOC and LOC/s, would be more appropriate measures in principle, but we could not obtain that information from the abc runtime, and we did not also conduct a manual count of the yield points. Still, the numbers already indicate that bytecode instrumentation can be a costly process. Recall however that the instrumentation is conducted only once until the test subjects' code changes. Hence, the instrumentation effort can be

Benchmark	LOC	Time(s)	LOC/s
Alarm Clock	210	18.9	11.11
Apache Common Lang	398	26.0	15.31
Bank	77	11.7	6.58
Clean	63	11.4	5.53
Dining Philosophers	29	5.6	5.18
Linked List	150	13.3	11.28
Merge Sort	98	12.1	8.10
Piper	102	14.0	7.29
Reorder	48	11.0	4.36
Semaphore	29	5.6	5.18
Two Stage	70	13.3	5.26
Wrong Lock	63	12.5	5.04

Table 5.2: Bytecode instrumentation time

amortized by the repetition of the tests over time.

5.2.3 Test execution

We executed the benchmark tests, using three distinct configurations. Two configurations employ cooperative execution: the first one uses the history-dependent coverage policy, the second uses the random coverage policy. The remaining configuration employs unconstrained scheduling of threads. The unconstrained execution is enabled using coverage=NONE in the CTestOptions annotation for a JUnit test class; the runThreads() Cooperari API method executes unmodified bytecode in this case.

For each of the configurations and benchmarks, we varied the number of threads from 2 to 32 and took the measures listed in Table 5.3. For each case, the results indicate the number of test trials executed, on top for each entry, and the execution time in seconds, at bottom. The times and test trials in the unconstrained execution are the average of 10 executions for each case. The number of trials for the cooperative execution cases does not vary, but the times are again the average of 10 executions. Entries in italic for some 2-thread settings (Alarm Clock, Piper, and Semaphore) indicate that the bug at stake is guaranteed not to occur, hence 1000 test trials are expected. Bold entries indicate that the bug may occur but is not reproduced after 1000 trials.

The first key observations relate to the comparison of effectiveness between cooperative and unconstrained test execution. Generally, we can conclude that cooperative execution exposes bugs that unconstrained execution cannot, as the former only failed to expose bugs after 1000 trials for the dining philosophers' 32-thread case and the semaphore's 16 and 32 thread cases; these two benchmarks require a very precise schedule for deadlock, as discussed in Chapter 3. In contrast, unconstrained execution worked only for the Clean and Piper examples. Moreover, cooperative testing requires a relatively small number of trials in many of the benchmarks.

Regarding execution times, the overhead imposed by cooperative execution is notice-

Benchmark	Hist. dep. coverage					Random coverage					Unconstrained execution				
	2	4	8	16	32	2	4	8	16	32	2	4	8	16	32
Alarm Clock	*1000*	7	3	1	2	*1000*	8	13	8	6	*1000*	———	**1000**	———	
	64.9	1.4	1.1	0.7	1.2	62.3	1.4	2.2	1.9	1.8	51.0	51.4	51.6	52.0	52.7
Apache Common Lang	1	1	1	1	1	1	1	1	1	1	———		**1000**	———	
	0.2	0.3	0.4	0.6	0.8	0.2	0.2	0.4	0.5	0.8	0.3	0.6	0.9	1.4	2.2
Bank	1	1	1	3	1	1	1	1	2	1	———		**1000**	———	
	0.1	0.2	0.4	1.0	1.0	0.1	0.2	0.3	1.1	1.0	1.4	1.5	1.9	2.6	3.7
Clean	3	4	2	1	1	21	2	1	1	1	25	3	1	1	1
	0.4	2.0	1.7	1.4	1.6	2.9	0.9	1.1	1.3	1.6	0.1	0.1	0.1	0.1	0.1
Dining Philosophers	2	9	48	581	**1000**	4	13	165	**1000**	**1000**	———		**1000**	———	
	0.1	0.2	1.0	18.8	189.4	0.1	0.2	2.4	23.8	46.2	0.3	0.4	0.8	1.3	2.3
Linked List	2	1	1	1	1	2	4	1	1	1	———		**1000**	———	
	0.1	0.2	0.4	1.0	1.2	0.1	0.2	0.1	0.2	0.4	0.1	0.6	0.9	1.4	1.7
Merge Sort	99	117	95	54	4	99	11	51	14	35	———		**1000**	———	
	4.6	6.8	17.7	26.9	3.2	4.2	2.5	4.6	3.5	9.5	0.3	0.4	0.7	1.3	2.3
Piper	*1000*	2	2	1	1	*1000*	3	1	1	1	*1000*	2	1	1	1
	7.7	0.3	0.3	0.6	0.8	7.2	0.1	0.2	0.4	0.7	0.7	0.1	0.1	0.1	0.1
Reorder	50	13	4	7	20	2	23	26	17	10	———		**1000**	———	
	0.4	0.2	0.2	0.5	1.5	0.1	0.3	0.7	0.8	0.9	0.3	0.4	0.8	1.4	2.3
Semaphore	*1000*	37	137	**1000**	**1000**	*1000*	8	249	**1000**	**1000**	*1000*	———	**1000**	———	
	6.5	0.7	2.7	34.4	73.1	6.0	0.2	2.4	32.1	63.0	0.3	0.5	0.8	1.2	1.9
Two Stage	52	57	11	15	28	324	141	157	92	46	———		**1000**	———	
	1.1	1.9	1.1	0.3	3.2	3.6	2.5	3.6	0.3	4.0	0.3	0.4	0.7	1.3	2.4
Wrong Lock	5	1	2	7	1	5	1	2	3	1	———		**1000**	———	
	0.1	0.1	0.2	0.9	0.5	0.1	0.1	0.2	0.5	0.5	0.3	0.4	0.8	1.3	2.2

Table 5.3: Benchmark results

able, particularly in the 1000-trial runs, e.g., approximately 20 times slower than unconstrained execution for the 2-thread setting in the semaphore example. This is due to the execution of instrumented code and the internal Cooperari support for cooperative execution. The issue is mitigated however by the execution of a smaller number of trials in all other cases.

Finally, we can compare the use of the history-dependent and random policies. The history-dependent policy tries to avoid repeated scheduling decisions, and in doing so it limits their state-space. The random coverage however seems to have the benefit of exploring different slices of that state-space earlier in some cases. In terms of test trials, the history-dependent policy clearly performs better in the Dining Philosophers and Two Stage benchmarks, but the random coverage policy has comparable performance otherwise. The other observation is that the history-dependent policy certainly involves more computational effort, due to its internal book-keeping, and as result, the test execution times are generally higher for a similar number of test trials with the random policy.

Chapter 6

Conclusion

We presented Cooperari, a testing tool for multithreaded Java software, based on cooperative semantics. The tool was implemented with the following core aspects: the instrumentation of thread interference points for cooperative execution; an execution environment established by a cooperative scheduler, a cooperative implementation of multithreading primitives, and runtime detection of deadlocks and races; and an environment for reproducible tests, in association to custom coverage policies for the exploration of the state-space of thread schedules. We validated the initial prototype using two state-space coverage criteria and a set of standard benchmark examples.

Our main conclusion is drawn from the benchmark evaluation. The use of cooperative semantics provides an appropriate approach for reproducible and deterministic testing of multithreaded code. By comparison, unconstrained preemptive semantics provides a much weaker alternative for robust testing, as it failed to expose bugs in most cases.

In the future, we plan to extend and analyze the tool and the use of cooperative semantics in a number of ways, as follows:

- Conducting a more in-depth analysis of the use and the effectiveness of cooperative testing, e.g., performing a comparison to other related approaches such as randomized scheduling, and considering larger, real-word applications;
- Covering a wider set of multithreading Java primitives, e.g., atomic operations, barriers, futures, amongst others in the feature-rich `java.util.concurrent` API;
- Implementing and analyzing further coverage policies for state-space exploration, e.g., taking the stateless model-checking approach of other tools [3, 7, 17];
- Reaching a more stable state of development, such that it can be used reliably as a pedagogical tool in the software testing course at FCUL ("Verificação e Validação de Software");
- And developing an Eclipse IDE plugin, for simpler use and configuration of Cooperari software tests.

Bibliography

[1] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Abc: An extensible aspectj compiler. In *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, AOSD '05, pages 87–98. ACM, 2005.

[2] Eric Bodden and Klaus Havelund. Aspect-Oriented Race Detection in Java. *IEEE Trans. Software Eng.*, 36(4):509–527, 2010.

[3] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 183–198, New York, NY, USA, 2011. ACM.

[4] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 167–178. ACM, 2010.

[5] Jacob Burnim, Tayfun Elmas, George Necula, and Koushik Sen. CONCURRIT: Testing Concurrent Programs with Programmable State-Space Exploration. In *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism*, HotPar, page 16. USENIX Association, 2012.

[6] Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Framework for testing multi-threaded java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.

[7] Tayfun Elmas, Jacob Burnim, George Necula, and Koushik Sen. CONCURRIT: a domain specific language for reproducing concurrency bugs. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, PLDI, pages 153–164, 2013.

[8] Yaniv Eytani and Shmuel Ur. Compiling a benchmark of documented multi-threaded bugs. In *IPDPS*. IEEE Computer Society, 2004.

[9] Eitan Farchi, Yarden Nir, and Shmuel Ur. Concurrent Bug Patterns and How to Test Them. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS, page 286. IEEE Computer Society, 2003.

[10] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 110–121. ACM, 2005.

[11] Richard C. Holt. Some deadlock properties of computer systems. *ACM Comput. Surv.*, 4(3):179–196, 1972.

[12] Vilas Jagannath, Milos Gligoric, Dongyun Jin, Qingzhou Luo, Grigore Rosu, and Darko Marinov. Improved multithreaded unit testing. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE, pages 223–233. ACM, 2011.

[13] Pallavi Joshi, Mayur Naik, Chang-Seo Park, and Koushik Sen. CalFuzzer: An Extensible Active Testing Framework for Concurrent Programs. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV, pages 675–681. Springer-Verlag, 2009.

[14] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An overview of AspectJ. In *ECOOP 2001—Object-Oriented Programming*, pages 327–354. Springer, 2001.

[15] Bohuslav Krena, Zdenek Letko, Yarden Nir-Buchbinder, Rachel Tzoref-Brill, Shmuel Ur, and Tomás Vojnar. A Concurrency Testing Tool and Its Plug-Ins for Dynamic Analysis and Runtime Healing. In *Proceedings of the 9th International Workshop*, RV, pages 101–114. Springer Berlin Heidelberg, 2009.

[16] Doug Lea. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[17] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI, pages 267–280. USENIX Association, 2008.

[18] Yarden Nir-Buchbinder and Shmuel Ur. ConTest listeners: a concurrency-oriented infrastructure for Java test and heal tools. In *Fourth International Workshop on Software Quality Assurance: In Conjunction with the 6th ESEC/FSE Joint Meeting*, SOQUA, pages 9–16. ACM, 2007.

[19] William Pugh and Nathaniel Ayewah. Unit testing concurrent software. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE, pages 513–516. ACM, 2007.

[20] Junit. http://junit.org.

[21] Software-artifact Infrastructure Repository. http://sir.unl.edu.

[22] Scott D. Stoller. Testing Concurrent Java Programs using Randomized Scheduling. *Electr. Notes Theor. Comput. Sci.*, 70(4):142–157, 2002.

[23] Jaeheon Yi, Tim Disney, Stephen N. Freund, and Cormac Flanagan. Cooperative types for controlling thread interference in Java. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA, pages 232–242. ACM, 2012.

[24] Jaeheon Yi, Caitlin Sadowski, and Cormac Flanagan. Cooperative reasoning for preemptive execution. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP, pages 147–156. ACM, 2011.