2009

Aspect-Oriented Programming for Test Control

Siyuan Liu
University of Windsor

Aspect-Oriented Programming for Test Control

by

Siyuan Liu

A Thesis
Submitted to the Faculty of Graduate Studies
through Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada
2009

Aspect-Oriented Programming for Test Control

by

Siyuan Liu

APPROVED BY:

2 September 2009

Declaration of Co-Authorship

I hereby declare that this thesis incorporates material that is result of joint research, as follows:

This thesis also incorporates the outcome of a joint research under the supervision of professor Dr. Jessica Chen. The collaboration is covered in Chapter 2, Chapter 5 and Chapter 6 of the thesis. In all cases, the key ideas, primary contributions, experimental designs, data analysis and interpretation, were performed by the author, and the contribution of co-authors was primarily through the provision of some key ideas and constructive criticism.

I am aware of the University of Windsor Senate Policy on Authorship and I certify that I have properly acknowledged the contribution of other researchers to my thesis, and have obtained written permission from each of the co-author(s) to include the above material(s) in my thesis.

I certify that, with the above qualification, this thesis, and the research to which it refers, is the product of my own work.

Abstract

Distributed and multithreaded systems are usually much more complex to analyze or test due to the nondeterminism involved. A possible approach to testing nondeterministic systems is to direct the execution of the program under test to take a certain path for each test, so that a unique output (or output sequence) can be observed. Considering specification-based testing, we assume that a test case is given together with a test constraint for directing the internal nondeterministic choices. To instruct the program under test to execute according to a given test constraint, the program under test needs to communicate with the tester. In this thesis, we propose to use the features in Aspect-Oriented Programs to realize such communication. This solution does not require the availability of the source code of the program under test. Assuming that the program under test consists of a set of Java multithreaded processes communicating with each other using sockets, we provide an automated translation from a test constraint to a set of aspects using AspectJ.

Keywords: specification-based testing, distributed systems, nondeterminism, AspectJ.

Acknowledgements

This thesis work could not have been possible without the help of many people. First and most important, I would like to take this opportunity to express my appreciation to my supervisor, Dr. Jessica Chen. I am very thankful for her advice, patience and encouragement. And thanks to her pleasantry while solving my questions which keeps my interests through learning.

I would also like to thank my committee members, Dr. Jianguo Lu, Dr. Chunhong Chen and Dr. Alioune Ngom for spending their precious time to read my thesis and their comments, suggestions on this work.

Thanks also to my peers Yan Wang and Lihua Duan for their valuable advices and helps.

Table of Contents

List of Figures

1. Introduction

1.1 Reproducible Testing

With the advances of modern computers and computer networks, distributed and/or multithreaded software systems are becoming more and more popular. Improving qualities of these systems is an important task we are facing. However, distributed and multithreaded systems are very often much more complex to analyze or test due to the nondeterminism involved. Unlike traditional sequential systems, an input sequence given to the system may have several different execution paths depending on the interactions among different threads in each process and/or different processes possibly running on different machines across networks. Reproducible testing [5] is one of the possible approaches to performing testing in such an environment. With this approach, a test scenario consists of an external sequence of events (i.e. test case) and some additional information. The test case describes the external input and the observations, while the additional information, called test constraint, describes some constraints on the execution paths, such as partial or total order of the execution of some statements in the program. We introduce an additional process called test guide into the testing procedure so that given a test scenario, the system under test (i.e. the set of processes instantiated from the program under test) can be guided to take a certain execution path based on the given input sequence and the test constraint. Then the external observations can be compared with the desired ones.

1.2 Major Issues

There are three major issues related to this approach, in additional to the traditional study on software testing: how to define test constraints, how to obtain test constraints, how to realize guided testing.

To define a test constraint, we can use a partial order among the internal events which are defined on the checkpoints (i.e. program statements) of each process (see [7]). Clearly, each test constraint determines a set of execution paths satisfying this constraint. A test constraint may not be feasible. That is, the set of execution paths satisfying a certain test constraint can be empty. How to determine that a given test constraint is feasible is discussed in [8, 37] and how to select feasible test constraints remains an interesting problem beyond the scope of the present work. Even though a test constraint is feasible, it may not be proper, in the sense that there are at least two execution paths that satisfy the constraint but lead to different outputs. How to determine that a test constraint is proper and how to select proper test constraints so that unique output can be guaranteed for reproducible testing are interesting issues beyond the scope of the present work. In the following, we will only consider test constraints that are feasible and proper.

To obtain a test constraint, there are two typical ways: one is to run the system and record the total order of the checkpoints so that we can re-run the system with the same order. This so-called replay control technique is especially useful for regression testing. Another way is through the analysis of the requirement documents or design documents. From these documents, we may derive some particular execution paths that we are interested in. Very often, these are typical/representative scenarios in which possible errors or bugs may reside. In the following, we assume that test constraints are given.

To realize guided testing, we have two tasks to accomplish: one is to provide the test guide, and the other is to establish the communication between the test guide and the processes in the system under test, so that each process/thread can communicate with the test guide at the desired checkpoints. With this added communication, the test guide will be able to decide, according to the test constraint, whether a thread should proceed, wait for other threads, or resume from waiting state, based on the overall test constraint and the current status information of other threads either in this process or in other processes. In this way, the execution is guided to take a desired path.

1.3 Objective

The test guide can be a generic tool that is developed once for all as long as its communication protocol with any system under test is predefined. Such a tool was developed using Java Remote Method Invocation (RMI) and reported in [4].

To establish the communication between the test guide and the processes in the system under test, there are various ways: We can automatically insert auxiliary code into the program under test, or alter the execution environment (such as Java Virtual Machine for program under test writing in Java). In [4], it has been discussed a method and related tool support for automatically inserting additional code into the source code of any given Java program with Remote Method Invocation (RMI). In [9, 10], further discussions are given on how to introduce interceptors into different middleware layers of Java RMI to realize the communication between the test guide and the processes in the system under test. In this thesis, we present another solution to realize such communication by making use of the features in Aspect-Oriented Programs. We provide an automated translation from a test constraint to a set of aspects using AspectJ. The translated AspectJ program is then woven into the program under test. This solution does not require the availability of the source code of the program under test as in [9], and it is not dependent on the specification and implementation of Java runtime environment as in [4].

For most of the network applications, a distributed multithreaded system can be considered as a set of processes executed simultaneously, with each process possibly having several threads running concurrently. Different processes may run on the same machine or on different machines. The communication among the processes can be realized via CORBA, DCOM, RMI, stream sockets, (virtual) distributed shared memory etc. In our present work, a system under test consists of a set of processes, each running a Java program possibly with multithreading, communicating with each other using stream socket.

1.4 Thesis Structure

The rest of the thesis is organized as follows. We first introduce in Chapter 2 the distributed bakery algorithm whose implementation is used as a program under test. Chapter 3 reviews some previous work in AOP for testing and automated reproducible testing. In Chapter 4 we explain the mechanism and test architecture of our guided testing. Chapter 5 gives the testing specification. We introduce our test case and describe test checkpoints as events generated from test constraints. In Chapter 6, we give a brief introduction to AspectJ, and then describe the translation from test constraints to AspectJ code. Chapter 7 shows how test guide works to realize guided tests. Chapter 8 concludes this thesis work and mentioned possible future works.

2. Reproducible Testing with Distributed Bakery Algorithm

2.1 Introduction to Software Testing and Testing Non-deterministic Systems

With respect to the context in which software testing intends to operate, to provide stakeholders with information about the quality of the product or service becomes an empirical investigation conducted by software testing. To let people understand the risks of using the software product, software testing also provides an objective, independent view to it. Software testing is the process of executing a program or system with the intent of finding errors. Software testing can also be viewed as the process of validating and verifying a software application, which verifies whether the program meets the business and technical requirements.

Software testing can be carried out at any time in the software development process depending on the testing method employed, while most of the test effort is made when the requirements have already been defined or the coding process has been finished.

Though testing cannot identify all the defects of software applications completely, it compares the state and behavior of the application with the design oracles or mechanisms so that it is easy to recognize a problem. These oracles may include specifications, comparable products, inferences about intended or expected purpose, user expectations, relevant standards or other criteria.

One major purpose for software testing is to detect software failures so that errors may be uncovered and corrected. This is a non-trivial pursuit. Testing cannot show whether a product works properly in all conditions but can only show the product’s improper behavior under specific conditions. The scope of software testing often includes examining the code as well as executing the code in various environments. It exams whether the code does what it is supposed to do and does it properly. Information derived from software testing can also be used to correct the defects of the process by which software is developed.

Software testing methods are traditionally divided into black box testing and white box testing. These are the two approaches to describe the point of view of a designer when designing test cases.

Black box testing treats the software as a "black box"—without knowing any internal implementation. There is no knowledge provided to the users about the internal structure of the test object. Black box testing takes an external perspective of the test object to derive test cases. These tests can be functional or non-functional, but usually functional. The test designer selects from different inputs including both valid ones and invalid ones to test the program and determines the correct output. Black box testing methods include: equivalence partitioning, boundary value analysis, all-pairs testing, fuzz testing, model-based testing, traceability matrix, exploratory testing and specification-based testing.

Black box testing is applicable to all levels of software testing: unit testing, integration testing, functional testing, system testing and acceptance testing. The higher the level of testing we consider, the more complex the black box may become.

Advantages of black box testing include that it is efficient when used on large systems, and that tester can be non-technical with no need of having detailed functional knowledge of system. Disadvantages of black box testing include that test cases are tough and challenging to design when there is no clear functional specifications, and that it is very time consuming to identify all necessary inputs: There are chances of missing important paths during testing.

Specification-based testing is a special case of black box testing. It is introduced to test the functionality of software based on the applicable requirements. Thus, the tester inputs data into the test object, and only sees the output from it. This level of testing usually requires the tester to have the whole test cases. The tester then can simply verify the output value for a given input, and judge whether it is the same output as the expected value specified in the test case.

White box testing is possible when the tester has access to the internal data structures including the code of the program under test.

Types of white box testing include (i) API (application programming interface) testing which is the testing of the application using public and private APIs; (ii) Code coverage which creates tests to satisfy some criteria of code coverage (e.g., to create tests to make sure all statements in the program are executed at least once); (iii) fault injection methods; (iv) mutation testing methods; and (v) static testing.

Testing Non-deterministic Systems

Distributed systems' usages are more and more popular due to the well developed networking and web support, from web banking system and transportation system to large industrial system. The complexity of distributed and concurrent systems is growing as well, from the heterogeneity in terms of the adopted hardware to the nondeterminism which poses a great deal of difficulty in system analysis.

Concurrent programs and sequential programs behave differently. Executing a concurrent program many times with a predefined input may lead to different sequences of synchronization events and produce different results. This is because a distributed and concurrent system usually has many different execution paths because of the interactions among different processes with various kinds of process cooperation and their different running speed, which finally leads to different interleavings of the execution paths. As a result, testing turns out to be non-repeatable. This nondeterministic behavior makes testing concurrent programs notoriously difficult.

One approach to dealing with such difficulty is non-deterministic testing, which run a program with a fixed sequence of inputs many times in hope that faults will be exposed in one of these executions. This technique is called *test replay*. Nondeterministic testing is the most widely used approach in practice due to its simplicity. However, it is conducted in an ad hoc manner. To avoid this time consuming and inefficient test, some research has focused on how to efficiently insert noise makers (e.g. random delays) into selected locations of programs so that different synchronization events are likely to be executed by repeated executions and thus increase the chance of finding faults.

Another approach is to introduce a test guide and a series of test constraints. The latter is used to define points of interests in a program to be tested. Extra codes are inserted to the original program at these points to communicate with the test guide. The test guide gets instruction from test constraints and automatically guides the test via the communication with the processes.

2.2 Distributed Bakery Algorithm

Suppose that there are n distributed processes sharing a same resource e.g. a printer. Each process has a critical section to access this shared resource. The n processes should communicate among themselves so that no two processes enter their critical sections simultaneously. Lamport's Distributed Bakery algorithm provides a general solution to this problem where n processes communicate in a peer-to-peer manner. It works in this way: Each process is given an integer id. Each process maintains a current number which is initially set to zero.

When a process wishes to enter its critical section, it increases its current number by one, and assigns its ticket number to be this current number, just like people take a ticket number at a bakery store. Then it sends a request with its ticket number to all other processes.

When a process receives a request from another process with a ticket number k, it updates its current number to be k if k is bigger than its current number. Thus, the current number a process maintains is what it knows so far the biggest ticket number among all ticket numbers maintained by various processes. Note that it is possible that two processes have different current numbers, e.g. a process A has increased its current number to 4 but this number 4 has not yet arrived at process B so B's current number is still 3. It is also possible that several processes (locally) pick up the same ticket number.

A process A will send a reply to process B if one of the following three conditions holds (i) A does not need to access its critical section now (myTicketNum=0); (ii) B's ticket number is smaller than A's current number; (iii) B's ticket number is equal to A's current number and B has smaller process id. A process will enter its critical section only after it has received replies from all other processes, i.e. it has the smallest ticket number and among all processes having the smallest ticket number, it has the smallest process id. This guarantees the distributed mutual exclusion to access the critical sections.

```
class DBakery {
private int myTicketNum = 0;
private int currentNum = 0;
private int replyCount = 0;
public void DBakery() {
accept requests for socket connection from all processes with bigger id;
send requests for socket connection to all processes with smaller id;
for each socket of the connection with process pid
create a thread with name pid of class SocketThread and execute its run() method;
}
public void dBakeryAlgorithm() {
pickNum();
send requests with myTicketNum to all other processes to enter critical section;
replyCount = 0;
wait until replies from all other processes are received;
doCS();
myTicketNum = 0;
check req. queue: send replies to all processes whose requests have been deferred;
}
public void synchronized pickNum() {
currentNum++;
myTicketNum = currentNum;
}
public synchronized getCurrentNum() {
return currentNum;
}
public synchronized setCurrentNum(int i) {
currentNum = i;
}
public synchronized void addReply() {
replyCount++;
if (replyCount = MyProc.totalProcessNum -1)
notify that all replies are received
}
......
}
```

Figure 1: Code structure of the example: part 1

```
public class MyProg {
public static int processId, totalProcessNum;
public static void main(String[ ] args) {
change args[0] into an integer and assign it to MyProg.processId;
change args[1] into an integer and assign it to MyProg.totalProcessNum;
DBakery d = new DBakery();
while (true) do {
receive a user's command;
d.dBakeryAlgorithm();
}
}
}
class SocketThread extends Thread {

private DBakery d;
public void SocketThread(String s, DBakery d) {
super(s);
this.d = d;
}
public void run() {
while (true) do {
receive a message from its socket named s;
convert s into integer and assign it to pid;
if it is a request message, call handleRequest(receivedNumber, pid);
if it is a reply message, call d.addReply();
}
}
public void synchronized handleRequest(int n, int id) { int highNum = max(d.getCurrentNum(),
n); d.setCurrentNum(highNum);
if (d.getTicketNum()=0) or
(highNum > n) or (highNum = n and MyProg.processId > id)
send reply;
else
add this request to the request queue with (socketName, id)
}
}
```

Figure 2: Code structure of the example: part 2

Figure 1 and 2 show the draft of a sample Java code for the distributed bakery algorithm using sockets. Each process executing this piece of code is started providing two arguments: the id of the current process processId, and the total number of processes totalProcessNum. Suppose that there are two processes in the system under test executing the program in Figure 1 and 2. The first process is started with

```
>java MyProg 1 2
```

showing that there are 2 processes in total and the current process id is 1. Similarly, the second process is started with

```
>java MyProg 2 2
```

Each process executing MyProg has totalProcessNum number of threads: the main thread, which is the initial thread created when MyProg is started, and totalProcessNum -1 number of threads to handle the messages received from each of the other processes.

In the main thread, we use object d to execute method doBakeryAlgorithm each time the user gives an input command, e.g. to print a document, which is executed in a critical section expressed by method doCS(). Before calling doCS(), this process first picks up a ticket number by calling method pickNum() and waits until it has received replies from all other process. After calling doCS(), it needs to send out the deferred replies if there are any.

There are totalProcessNum -1 sockets for each process, one for each of the other processes. For the socket of the connection with process pid, we use the string of pid as the name of the socket as well as the name of the thread created for this socket. Thus, thread named pid is dedicated to handle the messages received from process pid using socket pid. There are two kinds of messages received: (i) requests from process pid to enter its critical section; and (ii) replies from process pid to the previous requests of the present process.

2.3 Reproducible Testing with Distributed Bakery Algorithm

In the following part, we show the nondeterministic behavior of distributed bakery algorithm. Suppose there are two processes with id 1 and 2 competing for the critical section. Process 1 will receive a message from its user and enter its critical section to print out A. Process 2 will receive a message from its user and enter its critical section to print out B. Suppose also that the input is to send a user's message to B followed by a user's message to A. When the two processes access the shared printer almost at the same time, the printing order varies due to their coming order which can be influenced by the internal scheduling of the machines. In fact, with the same input given above, we may have the following two situations and nondeterministically receive different output: AB or BA.

• Case 1: Process 1 enters critical section first.

• Case 2: Process 2 enters critical section first.

Figure 3 illustrates two scenarios with output AB and BA respectively. Initially, currentNum in both process 1 and process 2 are 0. After receiving the user's input, process 2 picks up number 1, and sends a request with number 1 to process 1. After process 2 received user's input, process 1 also receives user's input.

In the first scenario (Figure 3(A)), process 1 picks up a number before the request from process 2 arrives, and thus its ticket number is 1. When the request from

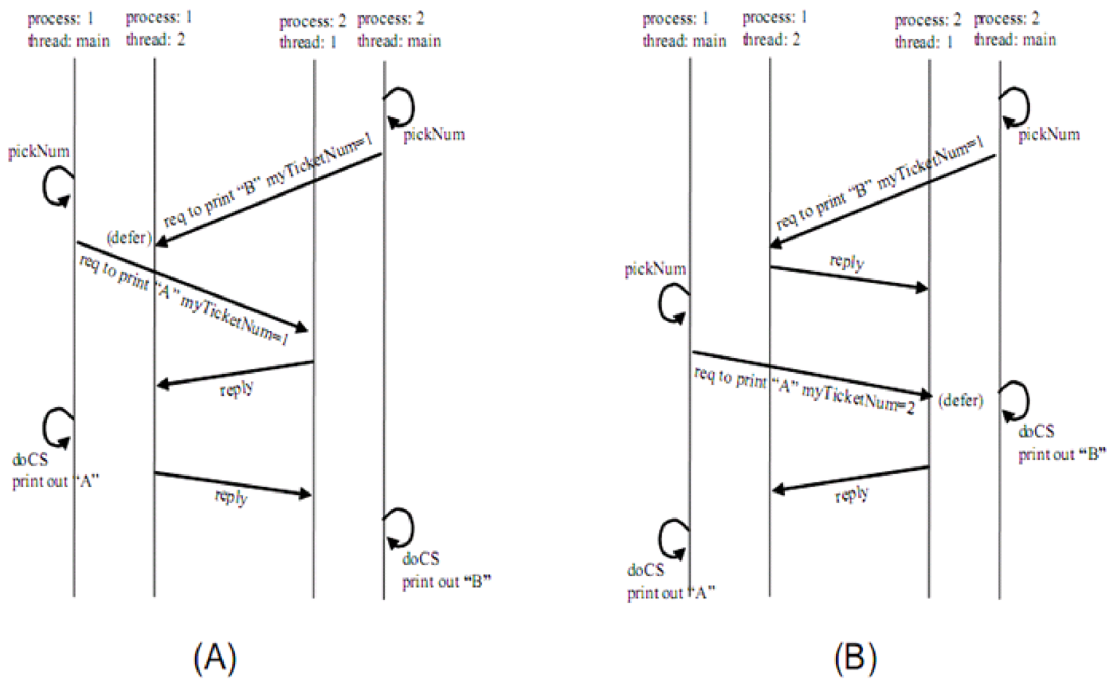

Figure 3 : Two possible scenarios

process 2 arrives, since the ticket number of process 1 and that of process 2 are both 1, and process 1 has smaller id, the reply to process 2 is deferred until process 1 finished its execution in its critical section. Thus, we have output A followed by B.

In the second scenario (Figure 3(B)), process 1 picks up a number after the request from process 2 arrives, and thus its ticket number is 2. When the request from process 2 arrives, since process 1 at this moment does not require the access to its critical section (myTicketNum=0), it sends out reply without delay. On the contrary, the request from process 1 is deferred until process 2 finished its execution in its critical section. Thus, we have output B followed by A.

Of course, there are also many other scenarios in which AB or BA are printed out. There are three major scenarios for Case 1 and two major scenarios for Case 2.

Scenario 1:

• Process 1 and process 2 pick up a number simultaneously.

• Process 1 and process 2 send their number to each other.

• Process 1 and Process 2 wait until they receive a reply from the other.

• By comparison, both of the processes have the same number, but process 1 has a smaller process ID which is 1, so process 2 is deferred.

• Process 2 replies to process 1.

• Process 1 receives the reply and then enters its critical section.

• Process 1 finishes critical section and replies to process 2.

• Process 2 receives the reply from process 1 and then enters its critical section.

Scenario 2:

• Process 1 picks a number first and sends it to process 2.

• Process 2 picks a number and sends it to process 1.

• Process 1 and Process 2 wait until they receive a reply from the other.

• By comparison, process 1 has a smaller ticket number, so process 2 is deferred.

• Process 2 replies to process 1.

• Process 1 receives the reply and enters its critical section.

• Process 1 finishes its critical section and replies to process 2.

• Process 2 receives the reply from process 1 and enters its critical section.

Scenario 3:

• Process 1 picks a number but not process 2.

• Process 1 sends the picked number to process 2.

• Process 1 wait until it receives a reply from process 2.

• Since Process 2 does not wish to enter critical section, it sends a reply back to Process 1.

• Process 1 receives the reply and enters its critical section.

Case 2:

Scenario 1:

• Process 2 picks a number first then sends it to process 1.

• Process 1 picks a number then send it to process 2.

• Process 1 and Process 2 wait until they receive a reply the other one.

• By comparison, process 2 has a smaller ticket number, then process 1 is deferred.

• Process 1 replies process 2.

• Process 2 receives the reply then enters critical section.

• Process 2 finishes critical section then replies process 1.

• Process 1 receives the reply from process 2 then enters critical section.

Scenario 2:

• Process 2 picks a number but not process 1.

• Process 2 sends the picked number to process 1.

• Process 2 wait until it receives a reply from process 1.

• Since Process 1 does not wish to enter critical section, it sends a reply back to Process 2.

• Process 2 receives the reply then enters critical section.

Of course, there are many other scenarios. The above two just show that nondeterminism exists in lots of places during a distributed program execution which leads to different execution paths.

While we may observe either AB or BA, the chance to observe output AB is very low. We give more details about this in the next sections on how to carry out guided testing in order to observe AB without repeating the test many times.

3. Aspect Oriented Programming

3.1 Characteristic of AOP

AOP is a programming paradigm which is an extension to Object-Oriented Programming (OOP). It has improved certain areas where OOP fails. Object-Oriented (OO) had a dramatic effect on how to develop software when it entered the mainstream of software development. Developers could visualize systems as groups of objects and the interaction between those objects. This allows them to form more complicated systems and to develop them in less time than ever before. The only problem with OOP is that the essential static model makes changes in requirements a profound impact on development timelines.

Aspect-Oriented Programming (AOP) complements OO programming by allowing the static OO model to be modified dynamically for creating a system which can grow to meet new requirements. An application can adopt new characteristics as it develops just like the state changes of objects in the real world during their lifecycles.

AOP allows dynamic modification of our static model to include the code required to fulfill the requirements without having to modify the original static model. Better still, we can keep this additional code in a single location in the aspect rather than scattering it all across the existing model, as we would have to if we were using OOP on its own.

AOP aims to solve these OOP problems by allowing crosscutting concerns to be cleanly captured in one self-contained unit of code. It increases modularity by enabling improved separation of concerns. In order to do so, we need to break down a program into different parts. By providing abstractions that can be used to implement, abstract and compose these concerns, all programming paradigms support some level of encapsulation of concerns into separate, independent entities. But some concerns called *crosscutting concerns* define these forms of implementation because they "cut across" multiple abstractions in a program.

Concerns are implemented in AOP by using blocks of code called aspects. Aspects contain a part called advice which are used to implement the crosscutting concerns. The places where the advice should be applied to the OOP codes are called joinpoints. A weaver is used to weave the AOP code with OOP code so the appropriate advices can be inserted at the places within the OOP code specified by the joinpoints. Typical examples of such crosscutting concerns implemented using AOP are: security, synchronization and tracing.

3.2 A Simple Example of Using AspectJ

AspectJ is a simple and practical aspect-oriented extension to Java. With just a few new constructs, AspectJ provides support of a range of crosscutting concerns for modular implementation.

It is possible to define additional implementation to run at certain well-defined points in the execution of the program due to dynamic crosscutting in AspectJ. It is based on a small but powerful set of constructs: join points are well-defined points in the program flow; Pointcut is a language construct to identify certain join points and certain values at those join points; advice are method-like constructs used to specify extended additional code to be executed at certain join points; and aspects are units of modular crosscutting implementation, composed of pointcuts, advice, and ordinary Java member declarations. Aspect can alter the behaviour of the base code (the non-aspect part of a program) by applying advice (additional behaviour) over a quantification of join points (points in the structure or execution of a program).

AspectJ is powerful, practical, and simple to use. The programs written using it are easy to understand. Here we give a simple AspectJ example.

In this java program, the main method will call the greeting method in class Hello, and the print out is simply a string of “Hello!”. With the additional AspectJ code woven into it, the output will print a string of “AOP>>” before the original string “Hello!”. The combined output is “AOP>> Hello!”. Here the AspectJ code defines a pointcut which is the calling of greeting method in class Hello. The advise specified before this pointcut expresses that the output stream should print “AOP>>” first. As a consequence, what the aspect does here is to print “AOP>>” before the calling of greeting method in class Hello.

```
public class Hello {
        void greeting(){
            System.out.println("Hello!");
        }
        public static void main( String[] args ){
            new Hello().greeting();
        }
}
public aspect With {
        before() : call( void Hello.greeting() ) {
            System.out.print(“AOP>> ");
        }
}
>ajc Hello.java With.aj
>java Hello
  AOP>> Hello!
```

In summary, AspectJ is defined as follows.

- Join points are well-defined points in the program flow and it is a place where the aspect can join the executions.
- Pointcut is a language construct to identify certain join points
  - e.g., call(void Hello.greeting())
- Advice is the code to be executed at certain join points
  - e.g., before() : call(void Hello.greeting()) {

    System.out.print("AOP>> "); }
- Aspect is a module containing pointcuts, advice, etc.
  - e.g., public aspect With {

    before(): call(void Hello.greeting())

    { System.out.print("AOP>> "); } }

If there is any arguments in a method e.g. greeting(int i), the pointcut in the corresponding aspect should be written with the arguments, e.g. call (void Hello.greeting(..)).

4. Related Work

4.1 AOP Used in Distributed Program Testing

Hughes et al. addressed the problem regarding the difficulty and expense of testing. Testing is a vital stage in the development cycle of any application but people often neglect to perform it successfully [22]. Due to the co-ordination required, it is especially serious for us to successfully test several distributed components simultaneously in distributed applications. Thus a framework implemented using Aspect-Oriented Programming and Reflection is proposed. It aims to ease the testing of distributed systems and other varieties of systems which encounter similar problems. The authors introduced the technique of automating the insertion/removal of monitoring code, from which the proposed Aspect Testing Framework simplifies the problem of testing complex distributed systems such as AGnuS. Furthermore, the ability of easily adding, modifying and removing communication code makes the tailoring of the communication to fit any monitoring interface easier. Thus, the time required to thoroughly test complex distributed systems is significantly reduced by facilitating the re-use of interface code.

The user can simply create a template using the tag <METHODNAME> to generalize a method name in the joinpoint and then apply the proposed framework. The framework will use the reflection API to examine the classes and present to the user all the potential joinpoints. The programmer can then select the class, methods, fields etc. to substitute the content of joinpoints with. Our work in testing is similar at this point.

In aspect-oriented refactoring, Mets ä et al. aimed at weaving test control points into the system under test by using aspects to keep the original system design oblivious to testing [28]. Aspects were intended to be used to carry out performance profiling tasks, and control points for test execution, etc. Their purpose was to declare system structures that could be used for injecting test code into the system by defining testability pointcuts. Control points for test execution and monitoring were also provided. Pointcuts can be structural, temporal, functional, or non-functional implemented depending on the purpose of the test case. These pointcuts could be derived from the client requirements, design constraints, and architectural requirements.

Copty and Ur in 2003 examined the possibility of implementing the instrumentation part of a multi-threaded testing tool using AOP [13]. They performed a detailed examination of all the requirements we are interested in and checked their satisfiability with AspectJ. However, as a drawback, how to define synchronization blocks as places for instrumentation is not worked out with AspectJ.

4.2 Different Ways to Realize Reproducible Testing

4.2.1 Execution Replay of Nondeterministic Programs

In 1990, Leu et al. presented a class "control driven" [27]. It realizes execution replay on distributed memory architectures which is a complement to the first technique "control driven execution replay" proposed by Leblanc in the context of shared memory architectures. In contrary to all other proposed approaches, their technique is adapted to non-blocking primitives, and is not dependent on any form of message passing communication.

Later on, Carver and Tai proposed a so called deterministic execution debugging and testing to solving the problems caused by nondeterministic execution behavior [7]. They are the first people who introduced the idea of deterministic testing of concurrent programs. They presented a language-based approach, and used examples of semaphores and monitors to implement process synchronization in concurrent programs to show their approach to developing synchronization-sequence replay tools for concurrent programs. First, it collects the sequence of synchronization events of a concurrent program by transforming it into a new program and executing the new one, and then controls the execution of concurrent program by transforming it into different programs that can replay the collected synchronization sequences.

Bates described a high-level debugging approach, Event-Based Behavioral Abstraction (EBBA) [3]. His behavior-modeling algorithm is used to match actual behavior to models of expected program behaviors and automates many behavior analysis steps. In EBBA, the behavior is expressed as a sequence of events and the relationship of different event types.

4.2.2 Specification-based Test Control over Events in Nondeterministic Programs

The major issue in guided testing is to establish communications between test guide and program under test.

Sohn et al. proposed a dynamic state-based reproducible testing approach for component software in which each component can change the system state nondeterministically during the concurrent execution in a Common Object Request Broker Architecture (CORBA) environment [33]. In this approach, by inserting the communication primitives before and after the concurrent statement on the original program, it generates an extended program which is logically equivalent to the original one. A replay controller for a given state sequence from a statechart-like model of each component is used, which is designed to force the order of the extended program's execution based on the given state sequence.

Cai and Chen presented their work in test control methods for distributed concurrent systems, and the frame work of their automated test control toolkit which can help users to realize some particular execution paths desired [4]. In this approach, artificially controlling the partial order of synchronization events in distributed multithreaded programs was proposed. This framework is implemented in java and adopts CORBA as its underlying middleware for communication among processes. The authors designed a parser to automatically insert code into the original PUT to let it communicate with a test controller, which controls the order of remote calls according to monitor constraints. The extended PUT should request for permission from the controllers whenever it makes a remote method call, although a monitor constraint does not contain any events regarding remote method calls. This work realized guided testing through controlling of some important synchronization events derived from the test documents.

In order to inject the interceptors into the underlying middleware system, Chen and Wang modified existing Java library for the control of remote calls and provided a solution to intercept Java remote method calls and responses [9]. They implemented a distributed testing environment, with some of the components residing with each process in the AUT. Local test drivers and local path controllers are included in the local testing components. They also used a centralized communicator to coordinate among testing components and adopted Java RMI for the communications between testing components. With the help of the interception service implemented by Java RMI, they realized the control of the order over the input and the remote call events.

Furthermore, this work requires neither the availability of the program code nor the test user's knowledge to intrude into the underlying system as they discussed about the AUT.

4.3 Proposed Approach

We gave the classification of reproducible testing approaches and Aspect-Oriented Programming used in testing. Furthermore we will define the constraints manually created to identify point of interests to be tested [3, 4, 7-9, 22, 27, 28, 33]. In this work, we put the emphasis on approaches which make the test reproducible and controllable. Our work does not require the availability of the program code, but it needs user's knowledge to fix feasible test constraints to avoid deadlock during the control of the test. We provide an automated translation from a test constraint to a set of aspects using AspectJ, The translated AspectJ program is then woven into the program under test. The advantage of AspectJ is that it is simple and practical to use, and it provides support of a range of crosscutting concerns for modular implementation. Furthermore, AspectJ code can be written once for all, and automatically woven into the code of any program under test.

To realize guided test, a test guide is also built to guide a PUT to take a certain path. The execution of the PUT is augmented by weaving additional AspectJ code which realizes the communication between test guide and the PUT.

5. Testing Architecture

The structure of our testing method is shown in Figure 4. A test case is usually defined as a sequence of input/output pairs. For simplicity, we consider here that each test case is an input/output pair. An extension of the current work to handle sequences of input/output pairs is straightforward. Each test case is associated with a test constraint which describes a partial order among internal events, which refers to the execution of a process at a checkpoint in the source code.

We assume that program under test, test case, and test constraints are given. AspectJ code is automatically generated from the test constraint, and woven into the given program under test, to form the extended program under test. The execution of the extended program under test is augmented by the communication with the test guide, which is written once for all. The test guide takes the test constraint as input to make sure it is satisfied by properly delaying the communication with the processes in the system under test. As we noted in the Introduction, we assume that the given test constraint can uniquely determine one output, and this actual output is compared by a test oracle with the expected one given in the test case.

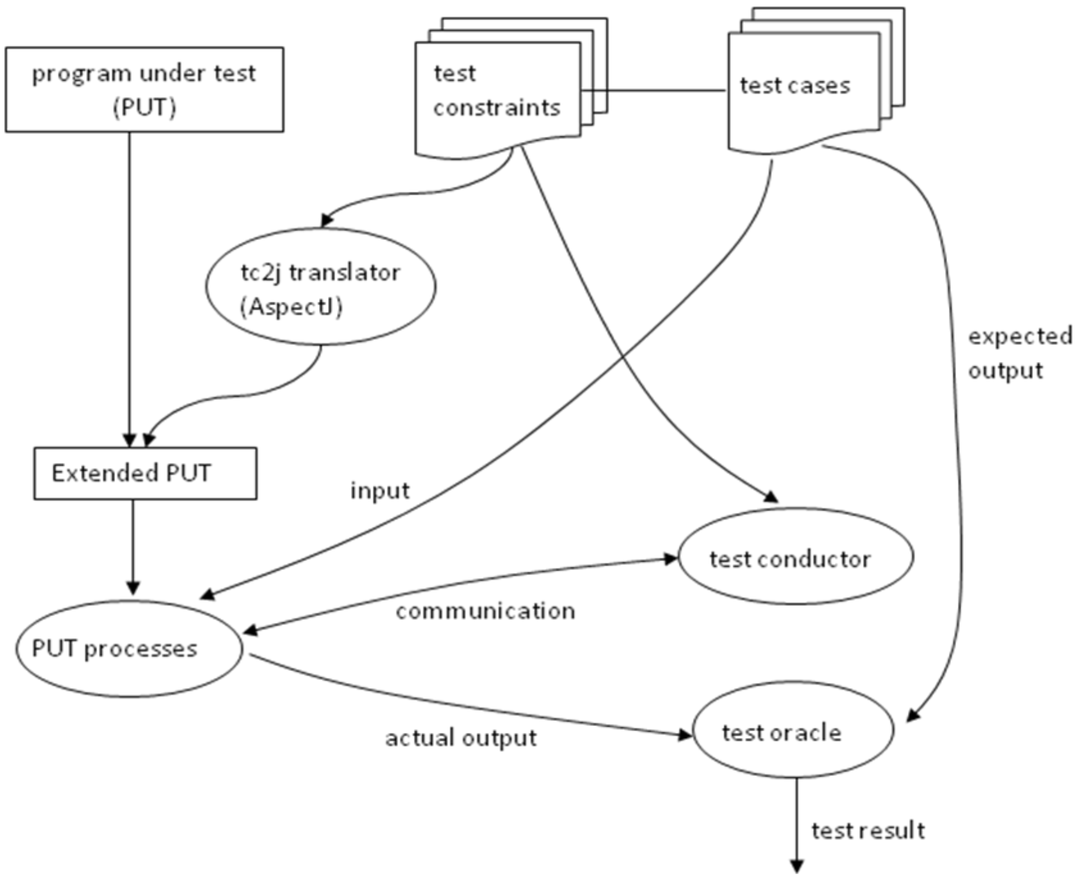

Figure 4: Testing Architecture

6. Test Specification

6.1 Test Case Specification

In the distributed bakery example, we would like to make sure that output A comes out before output B, and the test requirement we would like to satisfy can be informally expressed as follows:

• (input) process 1 and 2 each requests once to enter its critical section;

• (test constraint)

– process 2 picks up a number before process 1 does;

– process 1 enters critical section before process 2 does.

• (expected output) the output is AB.

The most intuitive way to realize this test is to let process 2 ask the test guide for permission to enter its critical section: the test guide will grant it permission if process 1 has already exited from its critical section. Figure 5(A1) shows such a scenario. Note that the augmented communication between the test guide and the processes in the system under test are given in dashed arrows. Here we have three augmented messages added to scenario (A) (see Figure 3(A)):

• m1: after process 1 exited from its critical section, it sends a message to acknowledge the test guide about it.

• m2: before process 2 enters its critical section, it sends a message to ask for permission from the test guide.

• m3: the test guide sends a message to process 2 to grant it the permission.

Of course, the test guide will delay message m3 if it has not yet received message m1

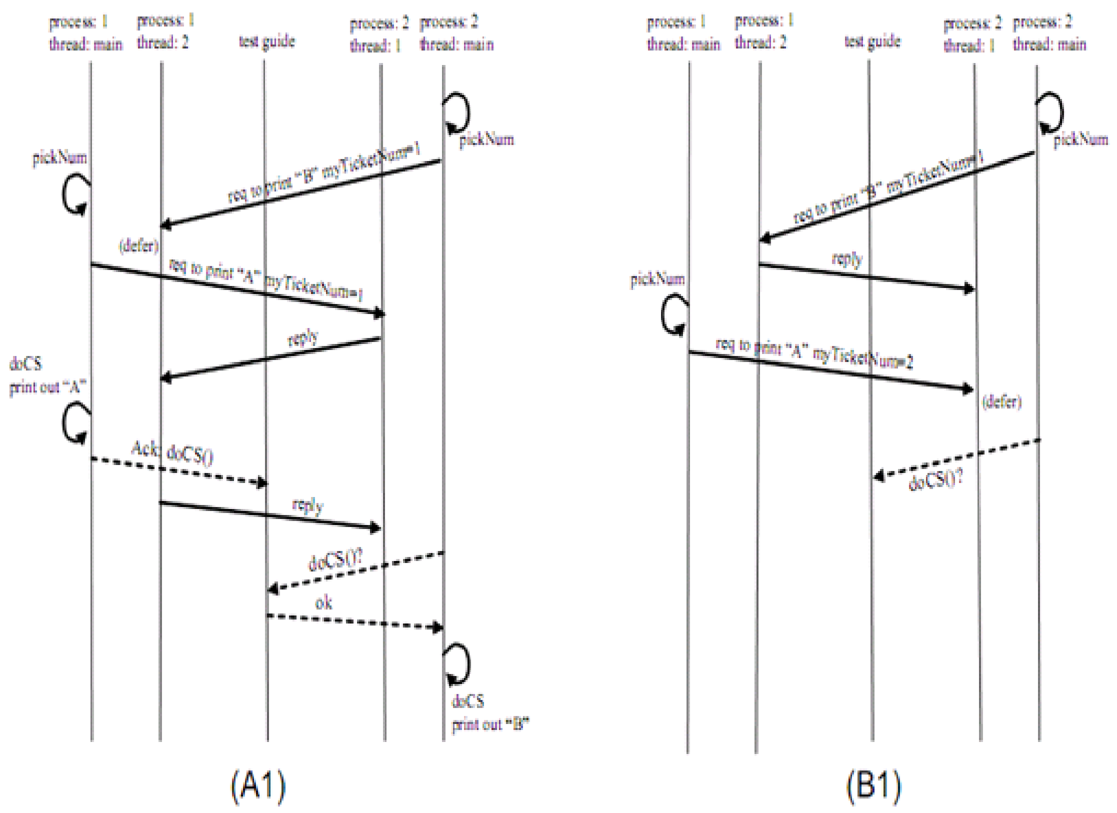

Figure 5 : Scenarios with test guide

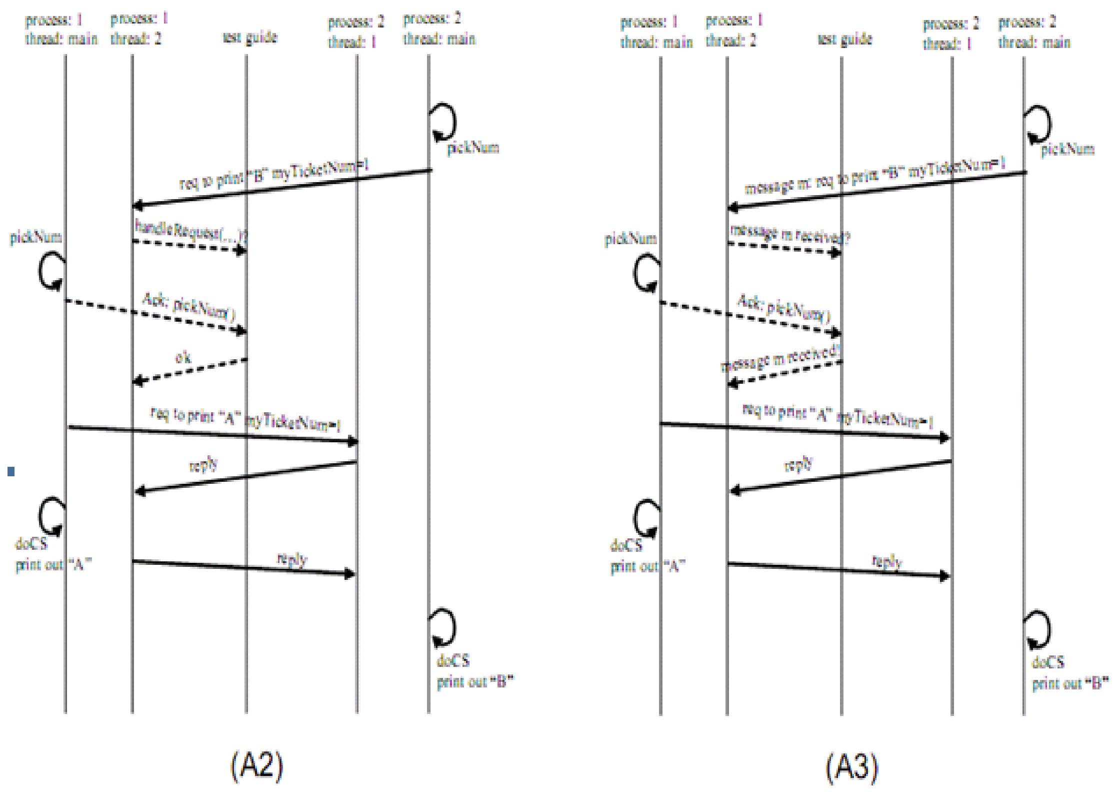

Figure 6 : Scenarios with test guide

from process 1. In another word, the test guide will make sure that e1 → e3 where "→" denotes the happen-before relation, and we use ei to denote the first occurrence of the internal event of mi. More generally, a test guide will make sure that a given test constraint is satisfied, where a test constraint is a partial order of the happen-before relation among the internal events. Figure 5(A1) suggests that {e1 → e3} might be used as a test constraint for the above test requirement for observing AB.

However, Figure 5(B1) shows another scenario when we cannot observe AB by using {e1 → e3} as test constraint. This scenario is obtained by adding the three messages to scenario (B) (see Figure 3(B)). Here, when process 2 asks for permission to enter its critical section, the test guide has to delay its response until it has received a message from process 1. Process 1, on the other hand, is waiting for the reply from process 2 for its request to enter its critical section, which will not arrive until process 2 exits from its own critical section. Thus, we cannot observe output AB.

This example shows that not all test constraints are feasible. The fact that {e1 → e3} is not a feasible solution suggests that process 2 should ask the test guide for permission to proceed at an earlier checkpoint rather than right before entering its critical section. Figure 6(A2) shows such a scenario which leads to a feasible test constraint: {e5 → e6} where

• m4: before invoking method handleRequest(...), process 1 will send a message to the test guide to ask for permission.

• m5: after process 1 completed method pickNum(), it sends a message to the test guide to acknowledge about it.

• m6: the test guide sends a message to process 1 to grant it the permission to proceed with handleRequest(...).

m5 → m6 makes sure that in process 1, pickNum() will be called ahead of handleRequest(). As a consequence, myTicketNum of process 1 has value 1 (instead of 2) when process 1 sends a request to process 2 to enter its critical section. This is essential for process 1 to enter its critical section ahead of process 2, and thus produces output AB.

Another possible scenario is shown in Figure 6(A3), which leads to an alternative test constraint: {e8 → e9} where

• m7: when process 1 received message m to request to print B with myTicketNum=1, it will send a message to the test guide to ask for permission to proceed with the received message.

• m8: after process 1 completed method pickNum(), it sends a message to the test guide to acknowledge about it.

• m9: the test guide sends a message to process 1 to process the received message.

Similar to {e5 → e6}, {e8 → e9} is a feasible test constraint. The only difference is that the permission request is sent upon the receipt of a message instead of upon the invocation of a method to process the message.

6.2 Test Constraints

To realize the communication between the test guide and the processes in the system under test, we need to (i) determine the checkpoints, i.e. the places in the program under test, where the additional messages are needed; (ii) determine the messages between the test guide and the processes of the system under test. In this section, we give detailed explanations of these two issues. The formal description of test constraints follows.

6.2.1 The Checkpoints of Interest

The checkpoints we are interested in are those that are essential to the nondeterministic choices along an execution path that may cause different output. Different outputs produced from a system with the same input is caused by the different orders of accessing shared objects, which is in turn caused by different execution speed among the threads and processes.

In addition to the order of accessing shared objects which is directly related to the output, there are orders of the executions of some other statements, such as those for process communication and thread cooperation, that are indirectly related to the output.

Process communication can be either the lower-level send and receive operations or the higher-level operations such as remote method invocations. When a remote method is invoked at the caller’s side, there is a new thread implicitly used as a proxy at the callee’s side to actually invoke the method. This implicitly used thread may also access the shared objects. From the user’s viewpoint, the remote method invocation can also be a checkpoint of interest, either at the caller’s side or at the callee’s side. Process communication with send/receive operations as considered in our setting is not strongly related to the uniqueness of the output. Users may however find it convenient to use the send and receive operations as checkpoints.

Thread cooperation refers to the coordination among multiple threads using wait/notify mechanism provided by various programming languages. When the wait/notify mechanism is used, there is usually a condition involved, upon the truth value of which the waiting thread would be notified. This condition itself is a shared object between the waiting thread and the notifying thread. The access to this shared object could be implicit.

In general, we allow users to use any checkpoint in the program under test as long as it is a method invocation. Theoretically, this simplification does not reduce the generality of the proposed method since all statements or blocks of statements can be rewritten into a method. Practically however, we would like to release this restriction to allow users to use any statement as a checkpoint. The extension of the present work in this aspect is left for future work. In particular, with our current running prototype, we exemplified how to use various methods provided by Java API (Abstract Programming Interface) for thread synchronization, process communication, and thread cooperation as checkpoints.

6.2.2 Event Description and Test Constraint

At the user-defined checkpoints, we would like to realize the desired communication between the test guide and the processes in the system under test. There are three typical kinds of messages we are interested in: (i) a permission request to the test guide to invoke a method; (ii) a response from the test guide to a request; (iii) an acknowledgement to the test guide for having completed a method invocation. The last one is necessary to enable the test guide to send responses to other permission requests.

An event is defined as an event name:event body pair. An event body is a tuple <pid, tName, cName, mName, type, num> where:

• pid is the id of the current process. A same program can be executed by different processes. We assume that when a process is started, a unique process id is given (e.g. from the command line) and it is kept in a special static variable MyProg.processId (simply written as processId below) which can be accessed anyway in the program.

• tName is the name of the current thread. We assume that thread names are all explicitly given in the source code, and can be obtained by invoking Thread.currentThread().getName() (simply written as getThreadName() below).

• cName is the class name of the method being invoked.

• mName is the name of the method being called. Considering operator overloading, a method should be distinguished jointly by the method name and the types of its parameters. Here we simply use method name to distinguish a method. An extension of the current work to handle more sophisticated message definition is straightforward.

• type has values 1, 2, 3, representing the types of the messages: type=1 specifies the message as a permission request to the test guide to invoke a method; type=2 specifies the message as a response from the test guide to the previous request; type=3 specifies the message as an acknowledgement to the test guide for having completed a method invocation.

• num is the number of appearances of this mName being called by thread tName in process pid.

Special symbol “*” is used in the case we are not interested in specifying a particular element. For instance, a “*” as a class name indicates that we are interested in the specified method defined in any class.

With the above formalization, the previous example messages can be expressed as:

• e1: (1, main, DBakery, doCS, 3, 1)

• e2: (2, main, DBakery, doCS, 1, 1)

• e3: (2, main, DBakery, doCS, 2, 1)

• e4: (1, 2, SocketThread, handleRequest, 1, 1)

• e5: (1, main, DBakery, pickNum, 3, 1)

• e6: (1, 2, SocketThread, handleRequest, 2, 1)

• e7: (1, 2, *, readLine, 1, 1)

• e8: (1, main, DBakery, pickNum, 3, 1)

• e9: (1, 2, *, readLine, 2, 1)

A test constraint is defined as a partial order of the happen-before relation among a set of internal events. With the internal messages defined above, event e1 should happen before event e2 can also be interpreted as e1 should be received before e2 is sent, where e1 is the type 3 event sent to the test guide to acknowledge the completion of e1, and e2 is the type 1 event sent to the test guide request to start e2. Thus, the internal events can be defined as the sending/receiving of the corresponding internal messages.

A test constraint file describes a partial order among a set of internal events. Both the AspectJ generator and the test guide read this file to extract useful information. A possible test constraint file for the distributed bakery example is given in Figure 7.

```
e4 : (1 , 2 , SocketThread, handleRequeset , 1 , 1 )

e5 : (1 ,main,DBakery, pickNum, 3 , 1 )

e5  →  e4
```

Figure 7: A test constraint file

Note that internal events of type 2 can be omitted from the specification since all the related information is carried in the corresponding events of type 1 so that the test guide can handle the response properly. In Figure 7, internal event e6 is omitted.

The AspectJ generator reads the first part of the test constraint file to get all the event name and event body pairs. It does not read the second part of the file i.e. the happen-before relation among internal events. For each internal event, the AJGenerator extracts the pid, tName, cName, mName, type, num tuple.

The test guide reads the second part of the test constraint file to understand the required happen-before relationship among the specified events.

7. Generating AspectJ Code from Test Constraints

For each internal event, the AJGenerator extracts the information of its event name, process id, thread name, class name, method name, event type and num. The AspectJ code is generated for the points of the execution where the specified method of the specified class is invoked by the specified thread and process. Note that for the communication with the test guide, only the event name is suffcient. For convenience, the type of the event is sent together with the event name in our prototype implementation.

For a type 1 event called eName with process id pid, thread name tName, class name cName, method name mName, number of occurrences num, the generated ApectJ code is shown in Figure 8. It automatically generates a unique joinpoint named methodCall0 for any invocation of method mName which is defined in class cName. An array named methodCalledTime[] is used to record the time of appearance of a <pid, tid, cName, mName, type> tuple. According to this piece of code, during the execution of the extended program under test, right before any invocation of method mName in class cName, the current process and thread with a response methodCalledTime[i] is checked. If the current process ID is pid, the current thread name is tName, and the methodCalledTime[i] reaches num then event name eName, together with the type 1 is sent to the test guide, and the execution is held, waiting for response from the test guide. Here, in and out are the input and output stream respectively for the socket connection with the test guide, which uses a predefined listening channel to establish connections with the processes in the system under test.

Note that different processes and different threads will use different channels for communication. Thus, the response from the test guide can be any message and it is only read, not used.

For a type 3 event called eName with process id pid, thread name tName, class name cName, method name mName, number of occurrences num, the generated ApectJ code is shown in Figure 9. It is similar to the code for type 1 events except that it sends out the acknowledgement message to the test guide without waiting for response.

The code generated from the test constraint file in Figure 7 is given in Figure 10.

```
public pointcut methodCall0(): call(* *.mName(..)) && within(cName);
before(): methodCall0() {
try {
if (MyProg.processId==pid &&
Thread.currentThread().getName().equals(tName)) {
methodCalledTime[0]++;
if(methodCalledTime[0]==num){
kkSocket[0] = new Socket("chenpc06",4111);
out[0] = new PrintWriter(kkSocket[0].getOutputStream(),true);
in[0] = new BufferedReader(new InputStreamReader(kkSocket[0].getInputStream()));
out[0].println("1 eName");
while ((in[0].readLine()) != null) {
break;
}}
}}
catch(Exception e) {}
}
```

Figure 8: Generated AspectJ code for type 1 event

```
public pointcut methodCall0(): execution(* *.mName(..)) && within(cName);
after(): methodCall0() {
```

```
try {
if (MyProg.processId==pid &&
Thread.currentThread().getName().equals(tName)) {
methodCalledTime[0]++;
if(methodCalledTime[0]==num){
kkSocket[0] = new Socket(“chenpc06”,4111);
out[0] = new PrintWriter(kkSocket[0].getOutputStream(),true);
out[0].println(”3 eName”);
} } }
catch(Exception e) {}
}
```

Figure 9: Generated AspectJ code for type 3 event

```
import java.io.*;
import java.net.*;
public aspect ControlSequ{
Socket[] kkSocket = new Socket[2];
PrintWriter[] out =new PrintWriter[2];
BufferedReader[] in =new BufferedReader[2];
int[] methodCalledTime = new int[2];
public pointcut methodCall0(): call(* *.handleRequest(..))&& within(SocketThread);
before(): methodCall0(){
try{
if(MyProg.processId==1){
if(Thread.currentThread().getName().equals("2")){
methodCalledTime[0]++;
if(methodCalledTime[0]==1){
kkSocket[0] = new Socket("chenpc06",4111);
out[0] = new PrintWriter(kkSocket[0].getOutputStream(),true);
in[0] = new BufferedReader(new InputStreamReader(kkSocket[0].getInputStream()));
out[0].println("1 e4");
while ((in[0].readLine()) != null)
{break;}
out[0].close();in[0].close();kkSocket[0].close();}}
}}
catch(Exception e){}
}
public pointcut methodCall1(): execution(* *.pickNo(..))&& within(DBakery);
after(): methodCall1(){
try{
if(MyProg.processId==1){
if(Thread.currentThread().getName().equals("main")){
methodCalledTime[1]++;
if(methodCalledTime[1]==1){
kkSocket[1] = new Socket("chenpc06",4111);
out[1] = new PrintWriter(kkSocket[1].getOutputStream(),true);
out[1].println("3 e5");
out[1].close();kkSocket[1].close();}
} } }
catch(Exception e){}
} }
```

Figure 10: Example of generated AspectJ code

8. Design and Implementation of Test Guide

8.1 Characteristic of Test Guide

Besides generating AspectJ code from test events, we also need a partial order relation between those events, following which we could realize the guided test. We can obtain this relation from the second part of the test constraint file. The test guide here is used to direct the test. It functions quite straightforwardly: It reads the relation from the constraint file and maintains two arrays. For each happen-before requirement e.g. ei→ej, the former event ei is stored in the first array and the second one ej is stored in the second array. Because there are so many relations, to avoid mixing them, each relation is stored in the two arrays with the same index. For example, for e5→e4, e5 and e4 are stored in preEvent[i] and postEvent[i] separately with the same index i. An array of boolean variable is also introduced to mark whether a preEvent has happened in order to help decide whether a postEvent can be enabled. All test guide has to do is to hold an postEvent until its preEvent has happened. Figure 13 and 14 illustrate how the PUT communicates with the test guide with type 1 and type3 event respectively.

In this thesis work, the prototype of the test guide we developed has multiple threads. The test guide uses main thread to keep listening for requests for the channel establishment at a predefined port. For each channel established, a new communication thread is created to receive messages and to send back responses if necessary. The test guide uses java stream socket for communication. The test constraints are saved in a simple text file with simple format, as illustrated in Figure 7, which makes the retrieval of test constraints very easy. The test guide gets information from the second part of this constraint file.

The goal of developing such a test guide is to help software testers to use the prototype to automatically control the execution of distributed multithreading programs under given test constraints.

Type 1 event e4: (1, 2, SocketThread, handleRequest, 1, 1)

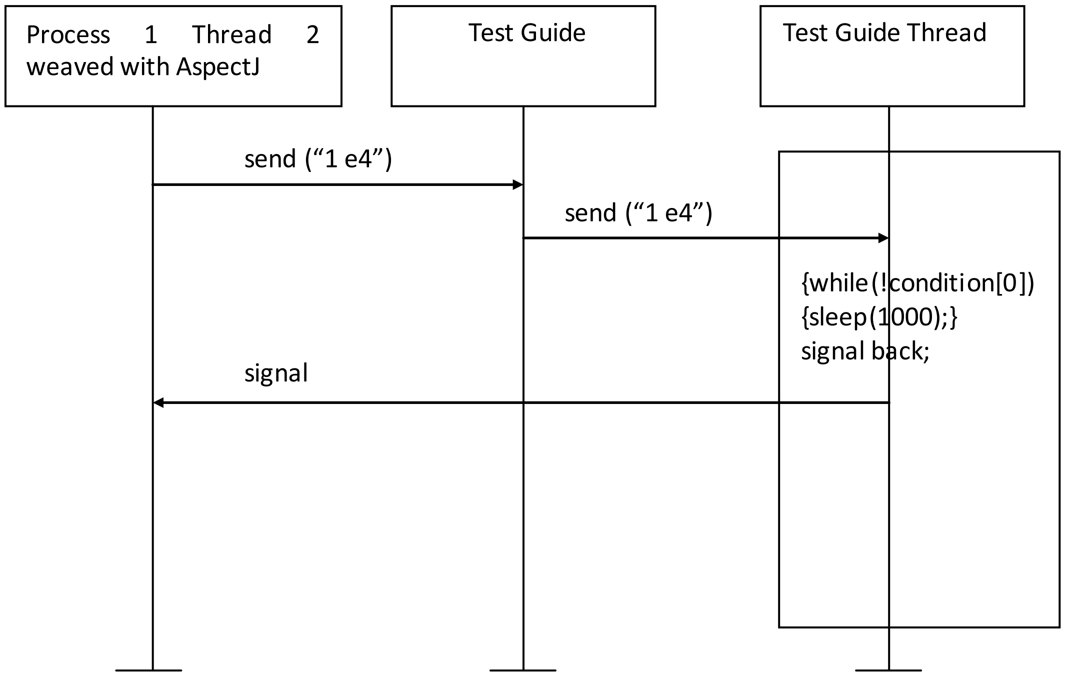

Figure 11: sequential chart of e4

Type 2 event e5: (1, main, DBakery, pickNum, 3, 1)

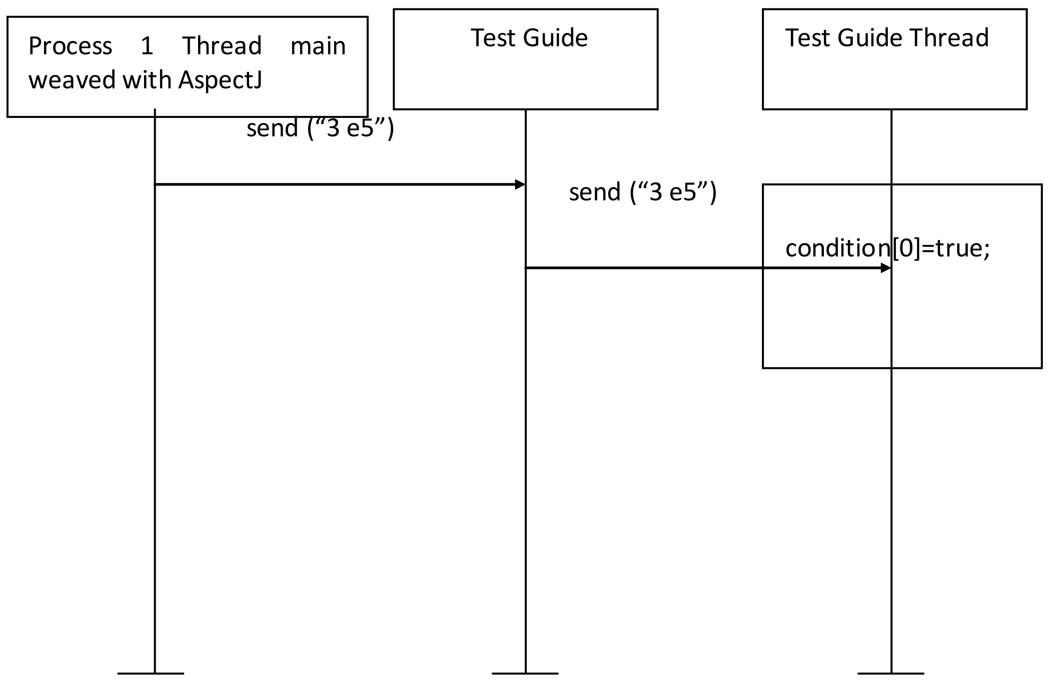

Figure 12: sequential chart of e5

8.2 Data Structure of the Test Guide

There are two types of events as we mentioned in Chapter 7. Now we explain how to handle these events using the test guide.

The class TestGuide first declares an integer to count the numbers of constraint relations in the second part of the test constraint file. Then it declares 3 arrays of this length. The first array, named preEvent of type string, is used to record the names of the events appeared before the happen-before relation "→". The second array, named postEvent of type string, is used to save the names of the events appeared after the happen-before relation "→". For example, with e5→e4, e5 is saved in the array preEvent[i], and e4 is saved in postEvent[i]. For each relation, the two events are saved separately in preEvent and postEvent but with the same array index i. The third array named "condition" is of type Boolean. Its elements are set to false initially.

The class TestGuideThread is a subclass of TestGuide which is used to handle incoming messages from the aspect of PUT. If it receives a type 3 event, it will search the array of preEvent to match the event name. If an event name is matched, the index of that preEvent is recoded, and the element with the same index of array condition is updated to true. If it receives a type 1 event, it will do the same thing as it receives a type 3 event. But instead of updating that condition to true, it will keep checking that condition until it becomes true. Then the TestGuideThread will signal back the corresponding aspect of PUT via stream socket.

In general, the test guide keeps a list B of (eName, cond) pairs for all blocked type 1 events, i.e. those events whose responses are held. Here eName is the name of the event being blocked, and cond is a condition. Whenever eName is enabled, the threads waiting on cond will be notified to send out corresponding responses. Based on the information about which events have happened, the test guide decides whether a request can be responded. cond will only be updated via type 3 events.

We give an example to show how our approach works in DBA. In the situation mentioned in Figure 6(A2), when process 2 wishes to enter the critical section, it will first pick a number, and then send the number with its pid to process 1. Process 1 will assign a thread of class SocketThread to handle this received message. This event is recognized by the AspectJ code weaved into process 1 as e4 which is the request to call handleRequest method in SocketThread class by pid 1 thread 2 for the first time. e4 is a type 1 event defined in the test constraints. AspectJ of current PUT will send this event to TestGuide and block current PUT, waiting for response from TestGuide. Since e5→e4 is required according to the test constraints file, TestGuide accepts the connection of this aspect and assigns a thread of class TestGuideThread to handle the message passed by this aspect. The TestGuideThread object will recognize this type 1 event automatically and keep it waiting. It will not send a signal back until the message of e5 arrives. At this time, process 1 receives an order to enter the critical section. It picks a number first. This is recognized by its aspect as event 5 which is the accomplishment of executing pickNum method in DBakery class by pid 1 thread main for the first time. e5 is a type 3 event defined in the test constraints. The aspect of current PUT sends e5 to TestGuide but will not block current PUT because type 3 event is an acknowledgment. The TestGuide accepts the connection of this aspect and assigns a thread of class TestGuideThread to handle the message passed by this aspect. The TestGuideThread recognizes this type 3 event automatically and updates the condition, which e4 is waiting for to become true. Then the TestGuideThread object used to handle e4 signals the aspect which sent e4, and that aspect will no longer block the PUT. Because process 1 picked a number before handling the request of process 2, it will pick the same number as process 2 does. Though they have the same number, process 1 has a smaller pid which means it has higher priority over process 2. Process 1 will enter critical section first. Finally, output AB is generated.

A sequential chart showing how the test guide handles incoming events is given in Figure 15.

8.3 Algorithm of the Test Guide

During the execution, it may happen a situation where one postEvent has two prevent to correspond. To make sure that the test guide thread will not give back a signal immediately after the cond of (eName', cond) comes true, we use a counter named countPreEvent. countPreEvent will count the number of the preEvent of a postEvent before the TestGuideThread functions. Each cond of (eName', cond) turned to true will lead to countPreEvent--. TestGuideThread will not signal the extended PUT until countPreEvent becomes 0. A sequential chart is showed in Figure 16 to illustrate this. When a communication thread receives a type 1 event named eName, it works as Figure 13 shows. When a communication thread receives a type 3 event named eName, it works as Figure 14 shows. Here, we assume a test constraint file like this:

e4: (1, 2, SocketThread, handleRequest, 1, 1)

e5: (1, main, DBakery, pickNum, 3, 1)

e10: (2, main, DBakery, pickNum, 3, 1)

{e5→e4} & {e10→e4}

A counter countPreEvent is used to count the preEvent of e4. In this case, countPreEvent is 2.

```
1: while (receive eName)
2:    for all (eName', cond) in list B do
3:        if eName' is enabled then
4:            notify those threads waiting on cond
5:        else
6:            update list B and put current thread to waiting state
7:        end if
8:    end for
9: end while
```

Figure 13: Handling type 1 event named eName

```
1: while (receive eName)
2:    for all (eName', cond) in list B do
3:        if eName' is enabled then
4:            notify those threads waiting on cond
5:        end if
6:    end for
7: end while
```

Figure 14: Handling type 3 event named eName

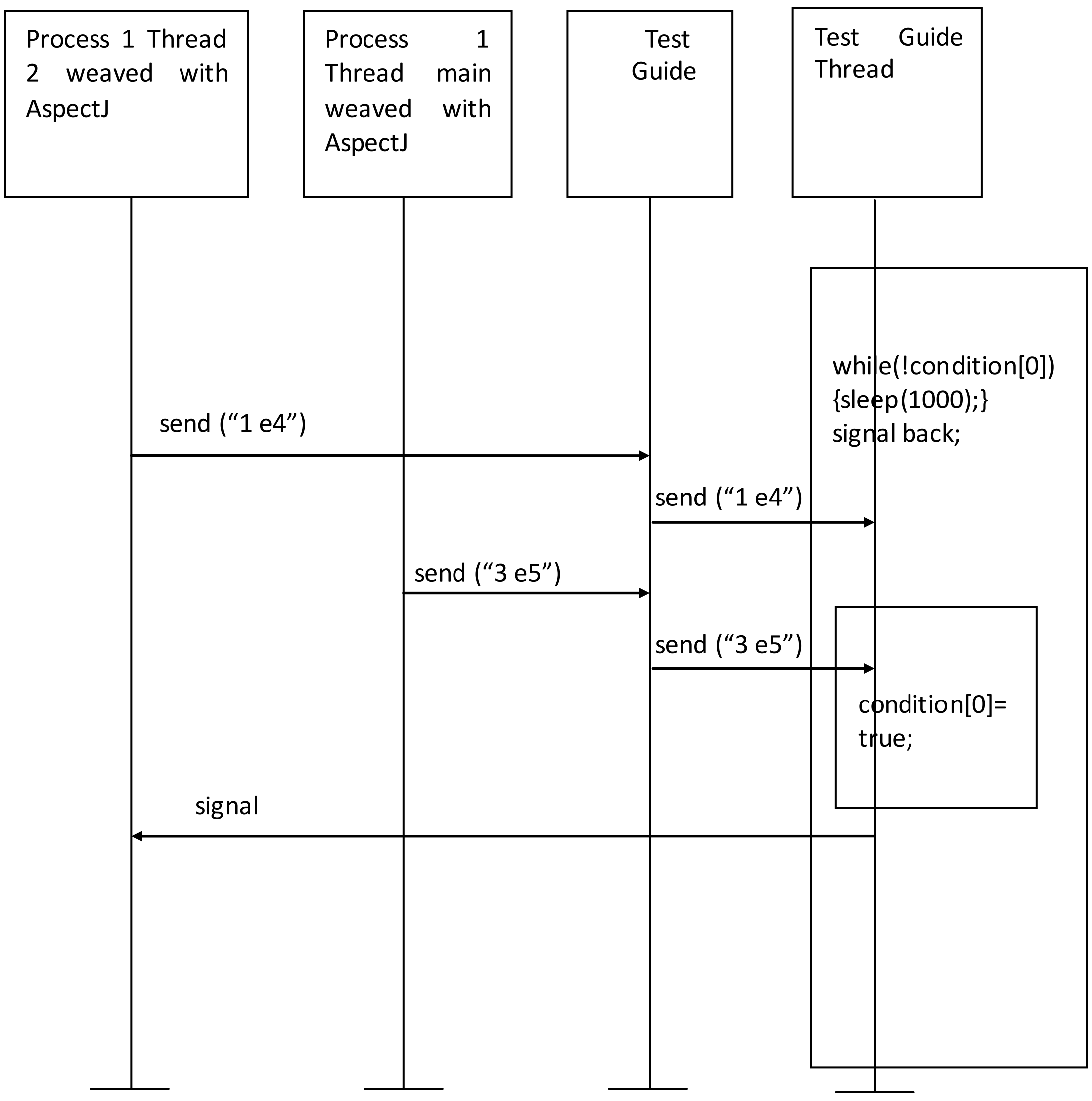

Figure 15: sequential chart of {e5→e4}

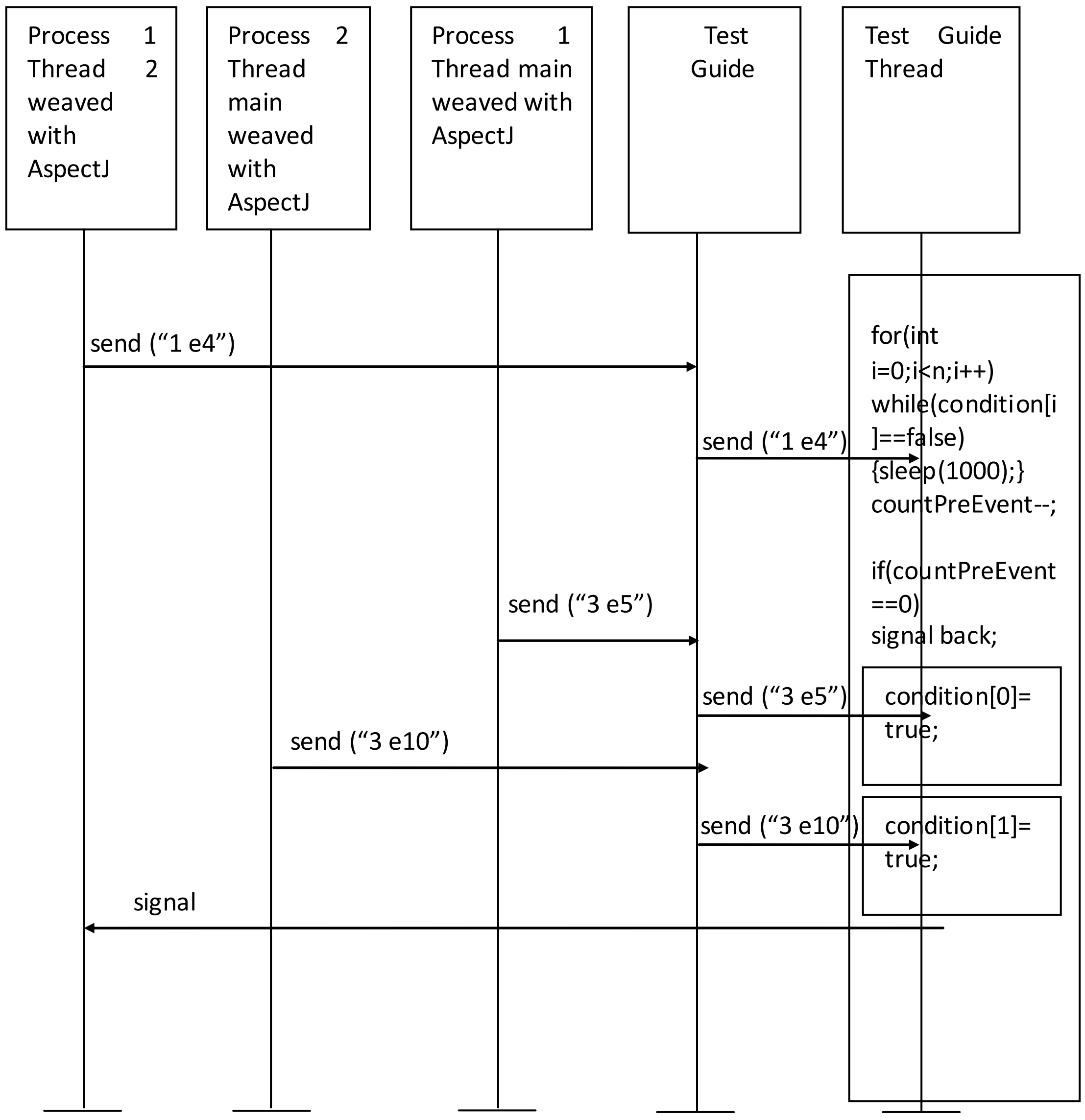
Process 1 Thread 2 weaved with AspectJ
Process 2 Thread main weaved with AspectJ
Process 1 Thread main weaved with AspectJ
Test Guide
Test Guide Thread
for(int i=0;i<n;i++) while(condition[i]==false) {sleep(1000);} countPreEvent--;
if(countPreEvent ==0) signal back;
send ("1 e4")
send ("1 e4")
send ("3 e5")
send ("3 e5")
condition[0]= true;
send ("3 e10")
send ("3 e10")
condition[1]= true;
signal

Figure 16: sequential chart of {e5→e4}&{e10→e4}

9. Conclusions and Future Work

In this Thesis work, we have proposed an approach to automated reproducible testing for distributed Java applications, via AspectJ. With AspectJ code weaved into the PUT, we could easily gain control over certain point of interest without modifying the original PUT. With a set of certain feasible test constraints, a generator AJgererator is introduced to generate a corresponding AspectJ class which will be weaved into PUT. Test guide also reads the relations from the test constraint file and saves the relations for further judgment. The extended PUT and Test Guide communicate with each other to generate a unique output.

We have introduced test scenarios, and discussed the feasible scenarios. We also overviewed the major functions of AspectJ and discussed how to generate AspectJ code from test constraints and how test guide functions to realize guided test. We have implemented this test guide and it runs well on the distributed bakery algorithm example.

Finally, we would like to mention that this AspectJ used in our work could only generate a pointcut (point of interest to test) from a method: either before or after a method call or execution. The communication we implemented between extended PUT and test guide is through Java stream sockets. It remains interesting to design a way to make any statements as checkpoints. Also, we are interested in extending this work for PUT communicating with test guide in ways other than stream sockets.

Bibliography

[1] AspectJ. AspectJ WWW site. At URL http://www.eclipse.org/aspectj/.

[2] BANIASSAD, E. AND CLARKE, S. 2004. Theme: An Approach for Aspect-Oriented Analysis and Design. *Proceedings of the 26th International Conference on Software Engineering*. 158-167.

[3] BATES, P. 1995. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Transactions on Computer Systems*. 13(1), 1-31.

[4] CAI, X. AND CHEN, J. 2000. Control of nondeterminism in testing distributed multi-threaded programs. *In Proc. of the 1st Asia-Pacific Conference on Quality Software (APAQS 2000)*. 29–38.

[5] CARVER R. H. and TAI K. C., 1986. Reproducible testing of concurrent programs based on shared variables. *inProc. 6th Int. Conf. Distributed Computing Systems.* 428-433.

[6] CARVER, R.H. AND TAI, K.C. 1998. Use of sequencing constraints for specification-based testing of concurrent programs. *IEEE Transactions on Software Engineering*. 24(6), 471-490.

[7] CARVER, R.H. AND TAI, K.C. 1991. Replay and testing for concurrent programs. *IEEE Software*. 66-74.

[8] CARVER, R.H. AND TAI, K.C. 1991. Static Analysis of Concurrent Software for Deriving Synchronization Constraints. *Proc. IEEE Int'l Conf. Distributed Computing Systems*. 544–551.

[9] CHEN, J. AND WANG, K. 2003. Constructing a Reproducible Testing Environment for Distributed Java Applications Quality Software, *Proceedings Third International Conference*. 402- 409.

[10] CHEN, J. 2003. Building Test Constraints for Testing Middleware-Based Distributed Systems. *In Software Engineering and Middleware, Lecture Notes in Computer Science.* 2596, 216-232.

[11] CHOI, J.-D. AND SRINIVASAN, H. 1998. Deterministic replay of java multithreaded applications. *In Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*. 48 – 59.

[12] COOPER, R. AND MARZULLO, K. 1991. Consistent detection of global predicates. *SIGPLAN Notices*. 167-174.

[13] COPTY, S. AND UR, S. 2005. Multi-threaded testing with AOP is easy, and it finds bugs! *In Proc. 11th International Euro-Par Conference, LNCS* 3648. 740-749.

[14] COTTENIER, T., BERG, A. AND ELRAD, T. 2007. Joinpoint inference from behavioral specification to implementation. In *Proc. of the 21st European Conference on Object-Oriented Programming (ECOOP)*. 4609, 476-500.

[15] DAMODARAM-KAMAL, S.K. AND FRANCIONI, J.M. 1993. Nondeterminacy: Testing and debugging in message passing parallel programs. *In ACM/ONR Workshop on Parallel and Distributed Debugging*, 118–128.

[16] EDELSTEIN, O., FARCHI, E., NIR, Y., RATSABY, G. AND UR, S. 2002. Multithreaded Java program test generation. *IBM Systems Journal*. 41(1), 111–125.

[17] FOWLER, R. AND LEBLANC, T. 1989. An Integrated Approach to Parallel Program Debugging and Performance Analysis on Large-scale Multiprocessors. *SIGPLAN Notices*. 24(1), 163 - 173.

[18] HANSEN, B. 1978. Reproducible Testing of Monitors, *Software Practice and Experience*. 721-729.

[19] HAREL, D. AND GERY, E. 1997, Executable Object Modeling with Statecharts. *IEEE Computer*. 30(7), 31–42.

[20] HARTMAN, A., KIRSHIN, A., AND NAGIN, K. 2002. A test execution environment running abstract tests for distributed software. *Proceedings of SEA*. 448—453.

[21] HAVELUND, K. AND PRESSBURGER, T. 2000. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer, STTT*. 2(4), 366-381.

[22] HUGHES, D., GREENWOOD, P., AND BLAIR, L. 2003. Aspect Testing Framework. *FMOODS/DAIS Student Workshop*.

[23] ITOH, E., FURUKAWA, Z., AND USHIJIMA, K. 1996. A prototype of a concurrent behavior monitoring tool for testing concurrent programs. *In Proc. of Asia-Pacific Software Engineering Conference (APSEC'96)*. 345-354.

[24] KENKATESAN, S. AND DATHAN, B. 1995. Testing and debugging distributed programs using global predicates. *IEEE Transactions on Software Engineering*. 21(2), 163-177.

[25] KICZALE, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. G. 2001. An overview of AspectJ. *Lecture Notes in Computer Science*. 2072, 327–355.

[26] LEBLANC, T. AND MELLOR, J. 1987. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*. 36(4), 471-482.

[27] LEU, E., SCHIPER, A., AND ZRAMDINI, A. 1990. Execution Replay on Distributed Memory Architectures. *IEEE Symposium on Parallel and Distributed Processing*. 106–112.

[28] METSA, J., KATARA, M., AND MIKKONEN, T. 2008. Comparing Aspects with Conventional Techniques for Increasing Testability. *Software Testing, Verification and Validation, 2008 1st International Conference*. 387-395.

[29] METSA, J., KATARA, M., AND MIKKONEN, T. 2008. Testing Non-functional Requirements with Aspects: An Industrial Case Study. *In Proceedings of the Seventh International Conference on Quality Software (QSIC 2007) IEEE Computer Society*. 5–14.

[30] OBERHUBER, M. 1995. Elimination of nondeterminacy for testing and debugging parallel programs. *In Mireille Ducassee, editor, Proceedings of 2nd Int. Workshop on Automated and Algorithmic Debugging.*

[31] PESONEN, J., KATARA, M., AND MIKKONEN, T. 2006. Production-testing of embedded systems with aspects. In Hardware and Software, Verification and Testing, LNCS. 3875, 90–102.

[32] SEBASTIAN BENZ. 2008. AspectT: Aspect-Oriented Test Case Instantiation. *Proceedings of the 7th international conference on Aspect-oriented software development, ACM, SESSION: Aspects and generative programming*. 1-12.

[33] SOHN, H., KUNG, D., AND HSIA, P. 1999. State-based reproducible testing for CORBA applications. *In Proc. of IEEE International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE'99)*. 24-35.

[34] SPEZIALETTI, M. 1989. A generalized approach to monitoring distributed computations for event occurrences. *Ph.D. Dissertation, Univ. Pittsburgh*. 212.

[35] STOLLER, S.D. 2000. Model-checking multi-threaded distributed java programs. *In Proceedings of the 7th International SPIN Workshop on Model Checking*. 224 – 244.

[36] STOLLER, S.D. 2002. Model-checking multi-threaded distributed Java programs. *International Journal on Software Tools for Technology Transfer*. 4(1), 71–91.

[37] TAI K-C, CARVER R. H., OBAID E. E., 1991. Debugging Concurrent Ada Programs by Deterministic Execution. *IEEE Transactions on Software Engineering,* 17(1), 45-63.

[38] WANG, Y., KING, G., AND WICKBURG, H. 1999. A method for built-in tests in component-based software maintenance. *In IEEE International Conference on Software Maintenance and Reengineering*. 186–189.

Appendix A

CODE OF TEST GUIDE & TEST GUIDE THREAD

```
import java.net.*;
import java.util.StringTokenizer;
import java.io.*;

public class TestGuide extends Thread{
    public static String[] preEvent,postEvent;
    public static boolean[] condition;
    public static int n;
    public static void main(String[] args) throws Exception {

            ServerSocket serverSocket = null;
            int portNum=4111;
            boolean listening = true;
            int i=0;
            String line;
            n=0;
            BufferedReader br = new BufferedReader(new FileReader("constraint.txt"));
            BufferedReader b = new BufferedReader(new FileReader("constraint.txt"));
            while( (b.readLine())!= null)
                n++;
            b.close();
            preEvent=new String[n];
            postEvent=new String[n];
            condition=new boolean[n];
        while( (line = br.readLine())!= null)
            {
                    StringTokenizer st    = new StringTokenizer(line,"{,},-,> ");
                    preEvent[i] = st.nextToken();
                    postEvent[i] = st.nextToken();
                    condition[i]=false;
                    i++;
            }
        br.close();

        try {
            serverSocket = new ServerSocket(portNum);
        } catch (IOException e) {
            System.err.println("Could not listen on port: 4111.");
            System.exit(-1);
        }

        while (listening)
        new TestGuideThread(serverSocket.accept()).start();

        serverSocket.close();
    }
}
```

```
import java.net.*;
import java.util.StringTokenizer;
import java.io.*;

public class TestGuideThread extends TestGuide {
        private Socket socket = null;
        public TestGuideThread(Socket socket) {
                this.socket = socket;
        }
        public void run() {

        try {
                PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
                BufferedReader in = new BufferedReader( new
        InputStreamReader(socket.getInputStream()));
                String inputLine, eventName, type;
                int countPreEvent=0;

                while ((inputLine = in.readLine()) != null) {
                        StringTokenizer st     = new StringTokenizer(inputLine," ");
                        type = st.nextToken();
                        eventName = st.nextToken();
                        for(int i=0;i<n;i++)
                        if(postEvent[i].equals(eventName))
                                countPreEvent++;
                        if(type.equals("3")){
                                for(int i=0;i<n;i++)
                                {
                                        if(preEvent[i].equals(eventName))
                                        {
                                                condition[i]=true;
                                                System.out.println("condition true");
                                }
                        }}

                        if(type.equals("1"))
                        {       for(int i=0;i<n;i++)
                                {
                                        if(postEvent[i].equals(eventName))
                                                {
                                                while(!condition[i])
                                                {sleep(1000);
                                                System.out.println("waiting condition ");}
                                                countPreEvent--;// there might be 2 preEvent for one postEvent
                                                if(countPreEvent==0)
                                                out.println("Go");
                                                }
                                                }
                                }
                        }

                out.close();
                in.close();
                socket.close();
        } catch (Exception e) {
                e.printStackTrace();
        }

        }
}
```

Appendix B

CODE OF AJGENERATOR

```
import java.io.*;
import java.util.*;

public class AJgenerator {

        public static void main(String[ ] args) throws Exception{
        int n=0,c=1;
        String line,eventName,pid,tid,className,methodName,type,counter;
        String computerName="chenpc06",portNum="4111";
        BufferedReader br = new BufferedReader(new FileReader("event.txt"));
        BufferedReader b = new BufferedReader(new FileReader("constraint.txt"));
        while( (b.readLine())!= null)
                c++;
        b.close();
        BufferedWriter bw = new BufferedWriter(new FileWriter("ControlSequ.aj"));
        bw.write(
                        "import java.io.*;\n"+"import java.net.*;\n"+"public aspect ControlSequ{\n"+
                        "Socket[] kkSocket = new Socket["+c+"];\n"+
                        "PrintWriter[] out =new PrintWriter["+c+"];\n"+
                        "BufferedReader[] in =new BufferedReader["+c+"];\n\n"+
                        "int[] methodCalledTime = new int["+c+"];\n"
        );

        while( (line = br.readLine())!= null)
        {
                        StringTokenizer st     = new StringTokenizer(line,"(,/,,)");
                        eventName = st.nextToken();
                        pid = st.nextToken();
                        tid = st.nextToken();
                        className = st.nextToken();
                        methodName = st.nextToken();
                        type = st.nextToken();
                        counter = st.nextToken();
                //Use string tokenizer to split event e1(tid,className,methodName,1)

                  if(type.equals("1")){
                                bw.write(
                                        "public pointcut methodCall"+n+"():" +" call(*
                                *."+methodName+"(..)"+")"+
                                        "&& within("+className+");"+"\n"+
                                        "before(): methodCall"+n+"(){\n"+
                                        "try{\n");

                        bw.write(

                                        "if(MyProg.processId=="+pid+"){\n"+
                                        "if(Thread.currentThread().getName().equals(\""+tid+"\")){\n"+
                        "methodCalledTime["+n+"]++;\n"+
                        "if(methodCalledTime["+n+"]=="+counter+"){\n"+
                        "kkSocket["+n+"] = new
                    Socket(\""+computerName+"\","+portNum+");\n"+
                        "out["+n+"] = new
                    PrintWriter(kkSocket["+n+"].getOutputStream(),true);\n"+
                        "in["+n+"] = new BufferedReader(new
                    InputStreamReader(kkSocket["+n+"].getInputStream()));\n\n"+
                        "out["+n+"].println(\"1 "+eventName+"\");\n"+
                        "while ((in["+n+"].readLine()) != null)\n"+
                        "{break;}\n"+
                        "out["+n+"].close();"+
                        "in["+n+"].close();"+
                        "kkSocket["+n+"].close();"+
                        "}}\n}}\ncatch(Exception e){}\n"+
                        "}\n"
                        );}
            If(type.equals("3"))
                {bw.write(
                        "public pointcut methodCall"+n+"():" +" execution(*
                *."+methodName+"(..)"+")"+
                        "&& within("+className+");"+"\n"+
                        "after(): methodCall"+n+"(){\n"+
                        "try{\n"+
                        "if(MyProg.processId=="+pid+"){\n"+
                        "if(Thread.currentThread().getName().equals(\""+tid+"\")){\n"+
                        "methodCalledTime["+n+"]++;\n"+
                        "if(methodCalledTime["+n+"]=="+counter+"){\n"+
                        "kkSocket["+n+"] = new
                Socket(\""+computerName+"\","+portNum+");\n"+
                        "out["+n+"] = new
                PrintWriter(kkSocket["+n+"].getOutputStream(),true);\n"+
                        "out["+n+"].println(\"3 "+eventName+"\");\n"+
                        "out["+n+"].close();"+
                        "kkSocket["+n+"].close();"+
                        "}\n}\n}\n}\n"+
                        "catch(Exception e){}\n}\n\n"
                        );
                        }

               n++;
        }
        bw.write("}\n");
        br.close();
        bw.close();
        }

}
```

Appendix C

CODE OF DISTRIBUTED BAKERY ALGORITHM

```
import java.io.IOException;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.UnknownHostException;

class DBakery extends Thread {

        ServerSocket serverSocket = null;
        boolean listening = true;
        public static int myTicketNum = 0;
        public static int currentNum = 0;
        public static int replyCount = 0;
        public static int deferedNumber = 0;
        public int totalProcessNum;
        public static int pid;
        public Socket[] sSocket;
        public PrintWriter[] out;
        public static PrintWriter[] deferedQueue;
        public static boolean NumPicked=false;
        public DBakery(){}
    public DBakery(int totalProcessNum,int id) throws IOException{

                pid = id;
                  this.totalProcessNum = totalProcessNum;
                  sSocket=new Socket[totalProcessNum];
                  out=new PrintWriter[totalProcessNum];
                  deferedQueue = new PrintWriter[totalProcessNum-1];
                try {
                        serverSocket = new ServerSocket(5555+pid-1);
                } catch (IOException e) {
                        System.err.println("Could not listen on port: 5555.");
                        System.exit(-1);
                }

        for(int i=0;i<pid-1;i++)
                {
//                        send requests for socket connection to all processes with smaller id;
                        try { sSocket[i] = new Socket("chenpc06", 5555+i);//chenpc06 is the name of this
PC
                                out[i] = new PrintWriter(sSocket[i].getOutputStream(), true);
                  } catch (UnknownHostException e) {
                          System.err.print("Don't know about host: chenpc06");
                          System.exit(1);}

                          new SocketThread(sSocket[i],pid).start();
                                // for each socket s, create a thread i of class SocketThread and execute its
run();
                }
        for(int j=pid-1;j<totalProcessNum-1;j++)
                new SocketThread(sSocket[j]=serverSocket.accept(),pid).start();
//      accept requests for socket connection from all processes with bigger id;


        }
        public void dBakeryAlgorithm() throws Exception{

                pickNo();
                NumPicked=true;
                for(int i=0;i<pid-1;i++)
                        out[i].println(myTicketNum+" "+pid);
                for(int i=pid-1;i<totalProcessNum-1;i++)
          {
                        try {
                        out[i] = new PrintWriter(sSocket[i].getOutputStream(), true);
                        out[i].println(myTicketNum+" "+pid);
                        } catch (UnknownHostException e) {
                                System.err.print("Don't know about host: chenpc06");
                                System.exit(1);
                          } catch (IOException e) {
                                System.err.print("Couldn't get I/O for the connection to: chenpc06");
                                System.exit(1);
                          }
              }

                // send requests with currentNum to all other processes in order to enter critical
section;
                replyCount = 0;
                // wait until replies from all other processes are received;
                while(replyCount != totalProcessNum -1)
                        Thread.sleep(1000);

                doCriticalSection();

                replydefer();
                NumPicked=false;

        }

        public synchronized void pickNo() {
                myTicketNum = ++currentNum;
        }
        public static int getCurrentNum() {
                return currentNum;
        }
        public static synchronized void setCurrentNum(int i) {
                currentNum = i;
        }
        public synchronized int addReply() {
                replyCount++;
                if (replyCount == totalProcessNum -1 )
```

```
                    return 0;
                  else return 1;
        }
        public synchronized void doCriticalSection()
        {System.out.println(pid+":"+Thread.currentThread().getName()+"     is     executing     critical
section");
    }

        public static void replydefer()
        {System.out.println("deferedNumber: "+deferedNumber);
                for(int i=0;i<deferedNumber;i++)
                {deferedQueue[i].println("reply");
                }
        }


        public static synchronized void addDefer(PrintWriter defer)
        { deferedQueue[deferedNumber]=defer;
        deferedNumber++;
        System.out.println("deferedNumber: "+deferedNumber+" added");
                }
}


import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.*;
class SocketThread extends DBakery {


        private int i ;
        private Socket socket = null;
        PrintWriter out = null;
        BufferedReader in = null;

        public static int pid;
        public SocketThread(){}
        public SocketThread(Socket socket,int id) {
                this.socket = socket;

                pid=id;
        }

        public void run() {


                try {
                      out = new PrintWriter(socket.getOutputStream(), true);
                        in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

                        String inputLine;
                    while ((inputLine = in.readLine()) != null) {

                      if (inputLine.equals("reply"))
                      {    System.out.println("received a reply");
                              i = addReply();
                                if(i==0)
                                      break;
                      }
                      else{
                              StringTokenizer st     = new StringTokenizer(inputLine," ");
                              String ticketNum = st.nextToken();
                              String id = st.nextToken();
                              handleRequest(Integer.parseInt(ticketNum),Integer.parseInt(id));

                      }
                        }

                        out.close();
                        in.close();
                        socket.close();

                } catch (IOException e) {
                        e.printStackTrace();
                }

        }
        public void handleRequest(int n, int id) {
                System.out.println(pid+":"+Thread.currentThread().getName()+" in handleRequest");
                int highNum = Math.max(getCurrentNum(), n);
                setCurrentNum(highNum);
                //compare highNum and n,
                if (highNum > n || (highNum == n && pid > id))
                { out.println("reply");
                }
                else if(!NumPicked)
                {out.println("reply");}
                              else //defer the reply -- keep the request in a deferred request queue;
                                      {
                                      try{
                                              addDefer(out); }
                                              catch(Exception e){}

                                              }
        }
        }


import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class MyProg extends JFrame implements ActionListener{

        public static int processId;
        public static int totalProcessNum;
        public static DBakery d;
        JButton MyButton;
        JButton MyButton2;

public MyProg() {
        super("DBakery process-"+processId);
        Container c = getContentPane();
        c.setLayout(new FlowLayout());
        MyButton = new JButton("run dbakey");
        MyButton.addActionListener(this);
        MyButton2 = new JButton("quit");
        MyButton2.addActionListener(this);

        c.add(MyButton);
        c.add(MyButton2);
        setSize(240, 120);
        setVisible(true);
}
public void actionPerformed( ActionEvent e )
{try{
        if (e.getSource() == MyButton)
                d.dBakeryAlgorithm();
        else
                System.exit(1);
}catch(Exception ex){}

}
        public static void main(String[ ] args) throws Exception    {
                processId = Integer.parseInt(args[0]);
                totalProcessNum = Integer.parseInt(args[1]);

               d = new DBakery(totalProcessNum,processId);

                                MyProg m=new MyProg();

            }
}
```

Vita Auctoris

Siyuan Liu was born in 1985 in Hubei, China. He graduated from Wuhan University of Technology, Wuhan, China in 2007, where he received a Bachelor's degree in Computer Science. He is currently a candidate for a Master's degree in the School of Computer Science at University of Windsor and expects to graduate in fall 2009.