A study on reproducible testing for distributed multithreaded Java programs.

Xiubin Cai
University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation
https://scholar.uwindsor.ca/etd/831
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®
A STUDY ON REPRODUCIBLE TESTING FOR DISTRIBUTED MULTITHREADED JAVA PROGRAMS

BY

XIUBIN CAI

A thesis submitted to the College of Graduate Studies and Research through the School of Computer Science in Partial Fulfillment of the Requirements for the Degree of Master of Science at the University of Windsor

Windsor, Ontario, Canada
2000
© 2000, Xiubin Cai
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.
ABSTRACT

Distributed Multithreaded (DM) programs are becoming more popular along with the development of network and Internet technology. Regarding the aspects of concurrency and communications such as message-passing, shared memory, and Remote Procedure Call (RPC), nondeterministic behavior in a Distributed Multithreaded (DM) program has become one of the biggest sources of difficulties in regression testing. Reproducible testing aims at providing methods and techniques to deal with this problem in testing nondeterministic programs. Such techniques cover the controlled execution of the program by using a separate control mechanism that forces the execution with a given test case.

In this thesis, we describe a reproducible testing method for DM programs. We propose an extended design notation - PMSC (Parallel Message Sequence Chart) based on MSC (Message Sequence Chart) to explicitly represent the static information of DM programs such as flow controls, thread interaction and synchronization, and object behavior. We also introduce a test case specification in Petri net, which is sufficient for describing a certain degree of deterministic behavior of concurrent programs. By constructing test constraints from the test case specification in Petri net, we can use the test constraints as a test scenario for our testing.

Based on the PMSC model and test constraints, we provide a new test control mechanism and algorithm that the test controller (TC) consists of multiple test controllers for multiple processes. A prototype is developed to evaluate the
performance of our reproducible testing approach. It shows that our approach can control a certain degree of deterministic execution with given test constraints. And the multiple test controllers can reduce the number of communications among the test controllers and the processes of the program.

**Keywords:** Non-determinism, Distributed Multithreaded (DM) program, Reproducible Testing, Static analysis, test case generation, MSC(Message Sequence Chart), Petri net.
To My Parents and My Family
ACKNOWLEDGEMENTS

The work presented in this thesis could not have been possible without the support of many people. I would like to take this opportunity to express my deepest thanks and appreciation to my supervisor, Dr. Xiaojun Chen, for her advice, comments, enthusiasm, encouragement and criticism through the development of this work. Dr. Chen has been an invaluable source of support and guidance throughout my graduate program.

I would also like to thank my committee members Dr. Indra A. Tjandra, Dr. Zhiguo Hu and Dr. Ingo Schmitt for spending their precious time to read this thesis and their comments, suggestions on this thesis work.

My special thanks go to the secretaries of the School of Computer Science, Ms. Mary Mardegan, and Ms. Margaret Garabon, for their consistent helps. Thanks go to Wanye, Hanmei, Lenong, Fang Li, and Xiaohong for their helps.

I would like to thank my husband, John Qiu, for his understanding and encouragement; and my parents, for their love and support.
# TABLE OF CONTENTS

**ABSTRACT** iii

**DEDICATION** v

**ACKNOWLEDGEMENTS** vi

**LIST OF FIGURES** xi

**Chapter 1 - INTRODUCTION** 1

1.1 Aim 1

1.2 Motivation 1

1.3 Features and Advantages of Reproducible Testing with MSC and Petri net 5

1.4 Objectives 5

1.5 Thesis Structure 6

**Chapter 2 - JAVA DISTRIBUTED MULTITHREADED (DM) ARCHITECTURE AND NON-DETERMINISTIC BEHAVIOR** 7

2.1 Java Distributed Multithreaded (DM) Constructs 7

2.1.1 Concurrency in Java – Multithreading 7

2.1.2 Java Distributed Programming 9

2.1.2.1 Java RMI (Remote Method Invocation) 9

2.1.2.2 Java IDL (Interface Definition Language) with CORBA 11

2.2 Java Distributed Multithreaded (DM) Architecture 12

2.3 Non-deterministic Behavior 14

2.4 An Example – Distributed Bakery Algorithm (DBA) 15

2.4.1 Java Implementation of DBA Algorithm 16

2.4.2 The Non-deterministic Behavior in DBA 18
Chapter 3 - EXTENDED DESIGN NOTATIONS – PMSC
(PARALLEL MESSAGE SEQUENCE CHART) 21
3.1 Overview of MSC (Message Sequence Chart) 21
3.2 Extended Design Notations – PMSC (Parallel Message Sequence Chart) 24

Chapter 4 - TEST CASE SPECIFICATION IN PETRI NET AND TEST CONSTRAINTS 28
4.1 Overview of Petri-net 28
4.2 Test Case Specification in Petri-net 30
4.3 Test Constraints 32

Chapter 5 - REPRODUCIBLE TESTING PROCESS 38

Chapter 6 - CODE INSERTION AND TEST CONTROL ALGORITHM 40
6.1 Event Types 40
6.2 Control Primitives 41
6.3 Code Insertion Module (CIM) 43
   6.3.1 Details of Code Insertion 43
   6.3.2 Rules of Code Insertion 49
6.4 The Structure of Test Control Mechanism 49
6.5 Test Control Algorithm and Mechanism 52
6.6 A Test Control Example 54

Chapter 7 - DESIGN AND IMPLEMENTATION OF A PROTOTYPE OF REPRODUCIBLE TESTING 57
7.1 System Requirements 57
7.2 Development of Code Insertion Module

7.3 Development of Test Controllers
  7.3.1 The Data Structure Implementation
  7.3.2 The Test Constraint Implementation
  7.3.3 The Test Controller Implementation
    7.3.3.1 The Major Functions of Test Controller Objects
    7.3.3.2 CORBA Objects Have Skeleton and Stub Sides
    7.3.3.3 Name Service

7.4 How to Run the System
  7.4.1 How to Run Code Insertion Module
  7.4.2 How to Run Test Controllers

Chapter 8 - EVALUATION

8.1 Analysis

8.2 The Examples
  8.2.1 Automated Banking
  8.2.2 Dining Philosophers
  8.2.3 Distributed Bakery Algorithm

8.3 The Results and Evaluation

Chapter 9 - RELATED WORK

9.1 Static Analysis Models
  9.1.1 FLAVERS (Flow Analysis for Verification of Systems)
  9.1.2 EBBA (Event-Based Behavioral Abstraction)

9.2 Test Case Generation
  9.2.1 EIAG (Event InterActions Graph)
  9.2.2 PPEG (Parallel Program Flow Graph)
9.3 Replay Control Methods 80
  9.3.1 Language-Based Replay Control Method 80
  9.3.2 Replay Control Mechanism with Communication Primitives 81
  9.3 Advantages of Our Approach 81

Chapter 10 - CONCLUSIONS AND FUTURE WORK 83
  10.1 Conclusions 83
  10.2 Future Work 84

BIBLIOGRAPHY 86

Appendix A - SOURCE CODE OF DISTRIBUTED BAKERY ALGORITHM 91
Appendix B - EXTENDED CODE OF DISTRIBUTED BAKERY ALGORITHM 96
Appendix C - SOURCE CODE OF AUTOMATED BANKING 103
Appendix D - SOURCE CODE OF DINING PHILOSOPHERS 108

VITA AUCTORIS 111
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 2.1</td>
<td>The RMI Model</td>
<td>10</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>The Structure of Java IDL with CORBA Implementation</td>
<td>12</td>
</tr>
<tr>
<td>Figure 2.3</td>
<td>Distributed Multithreaded Architecture</td>
<td>13</td>
</tr>
<tr>
<td>Figure 2.4</td>
<td>Distributed Bakery Algorithm</td>
<td>17</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>A Graphical Representation of Telephone Call in MSC</td>
<td>22</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>PMSC of MainProcess1 in DBA Algorithm</td>
<td>26</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>PMSC of SendRequest1 in DBA Algorithm</td>
<td>27</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>A Petri Net</td>
<td>29</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>Example of Test Case Specification in Petri Net</td>
<td>33</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>Example of Test Constraint</td>
<td>36</td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>Test Constraint Representation Based on Monitor and Remote Method</td>
<td>37</td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>Reproducible Testing Process</td>
<td>39</td>
</tr>
<tr>
<td>Figure 6.1</td>
<td>Communication Protocols between Processes and Test Controllers</td>
<td>44</td>
</tr>
<tr>
<td>Figure 6.2</td>
<td>The Rules of Code Insertion</td>
<td>50</td>
</tr>
<tr>
<td>Figure 6.3</td>
<td>The Structure of Test Control Mechanism</td>
<td>51</td>
</tr>
<tr>
<td>Figure 6.4</td>
<td>Test Control Algorithm</td>
<td>53</td>
</tr>
<tr>
<td>Figure 6.5</td>
<td>Test Control Scenarios</td>
<td>55</td>
</tr>
<tr>
<td>Figure 7.1</td>
<td>The Class Diagram of Code Insertion</td>
<td>60</td>
</tr>
<tr>
<td>Figure 7.2</td>
<td>The Class Diagram of Data Structures</td>
<td>62</td>
</tr>
<tr>
<td>Figure 7.3</td>
<td>The Class Diagram of Test Constraint Object Model</td>
<td>64</td>
</tr>
<tr>
<td>Figure 7.4</td>
<td>SAX DocumentHandler Interface Methods and Their Sequence</td>
<td>67</td>
</tr>
</tbody>
</table>
Chapter 1

INTRODUCTION

This chapter gives an introduction to the work of this thesis. It includes its aim, motivation and objectives. It also discusses the features and advantages of reproducible testing of distributed multithreaded programs using the techniques of MSC (Message Sequence Chart) and Petri net.

1.1 Aim

The aim of this thesis is to study and develop a new reproducible testing method which is a collection of techniques and tools for dealing with the non-deterministic execution in regression testing of distributed multithreaded (DM) programs. This method is based on MSC, which can be extended as a design notation, for describing the behavior of a DM program, and Petri net for describing test case specifications. A test control mechanism and related algorithms are provided for developing a prototype of our reproducible testing.

1.2 Motivation

Distributed multithreaded (DM) systems connect several computer systems together through a communication network. Each computer system consists of one or more processes. Each process may have multiple threads. The different threads execute concurrently and independently. DM systems are becoming more popular along the
development of network and Internet technology. The connected computer systems communicate with each other through message-passing, shared memory or Remote Procedure Call (RPC). The communication in the distributed multithreaded systems may delay. Furthermore, there are many more alternatives when considering thread interleaving. All these kinds of complications cause non-deterministic behavior in DM program. As a result, the repeated executions of the same program with the same input data may produce different sequences and different results.

The non-deterministic behavior makes testing of distributed multithreaded programs much more difficult. The problem is that we can't produce the previous running situation by a simple re-execution of the program with the same input data in regression testing. If the execution is different from the previous execution, we can not test the program with the same condition. Reproducible testing becomes one of the key issues in testing distributed multithreaded programs.

Reproducible testing is a technique that performs a controlled execution of the program by using a separate control mechanism that forces the execution with a given test case. Reproducible testing is performed in three steps:

(1) A static analysis model (SAM), which describes the program behavior and represents the static information of the program, must be generated to evaluate all possible executions of the program.

(2) Test case generation (TCM) and Feasibility Check Model (FCM) are provided to generate a feasible test case.
(3) A test case control mechanism (TCC) is performed to control the program execution with a given test case.

To deal with the non-deterministic behavior in distributed multithreaded (DM) programs, several approaches \([1,2,4,5,6,7,8,17,27]\) related to these three steps have been proposed.

Two approaches are related to the first step of reproducible testing. Naumovich \([1,2]\) provided a data flow graph model - FLAVERS (Flow Analysis for VErtification Systems). In this approach, concurrent programs are modeled by trace flow graphs (TFGs), and properties and thread interactions are described in deterministic finite state automata (FSA) over the TFG alphabet. The TFG model is useful for evaluating all possible executable paths for specific kinds of faults. Peter \([27]\) provides a program model, called EBBA (Event-Based Behavioral Abstraction). In this model, the program behavior is expressed as a collection of events and the relationship of different event types. The debugging of distributed systems is performed by comparing models of expected program behavior with the actual program behavior. We can use these models to describe the program behavior and evaluate all possible executions.

Katayama \([4,5]\) proposed a test case generation method based on the EIAG (Event Interactions Graph) model for Ada concurrent program. The method does not provide a feasibility check for the generated test case. Another test case generation technique from Pollock \([8]\) is used to generate a feasible test case based on PPEG (Parallel Program Flow Graph) model for multithreaded and share-memory based parallel programs. These models are not suitable for describing object behavior.
Several test case control mechanisms [6,7,17] have been provided for reproducible testing of concurrent Ada and distributed CORBA programs. Tai [17] uses the well-understood concurrent constructs of semaphores and monitors as examples for replay control of concurrent programs. Sohn [6,7] proposed a set of synchronized-communication primitives and a new replay control algorithm, which uses these primitives to arrange all the execution sequences of non-deterministic operations. The limitation of these approaches is that the centralized test controller may increase the number of communications among the test controller and the processes of the program since the events within a process are also sent to the global test controller.

To perform reproducible testing on DM programs is a challenging task since the non-deterministic behavior is complex and difficult to deal with. Regarding this difficulty, it is necessary for us to study and develop a reproducible testing method for DM programs. This research includes the background study on DM programs, MSC and Petri net, and proposes an extended design notation, the PMSC (Parallel Message Sequence Chart), and introduces a test case specification in Petri net and test constraints. A test control mechanism and related algorithms based on PMSC and test constraints are provided for reproducible testing of DM programs.

Java language [10] is becoming more and more popular, which enables the development of concurrent and distributed system through threads, socket communication, Remote Method Invocation (RMI) and Java Interface Definition Language (JIDL). We chose Java language as our example for reproducible testing of DM programs.
1.3 Features and Advantages of Reproducible Testing with MSC and Petri net

Message Sequence Chart (MSC) [13,14,15,16] has been widely used for the description of message flow and behavior among the processes of distributed, concurrent and communicating systems. Its clear graphical representation and object-oriented support give an intuitive understanding of the described message flows and object behavior of distributed object-oriented system. By providing an extended design notation with some extension for representing concurrent behavior based on MSC, the thread interaction and synchronization of DM programs are explicitly represented in the static analysis model.

For many years, Petri nets [22,23] have been used for modeling the behavior of various types of concurrent systems. Petri nets are graphical representations for the specification of concurrent systems. The concurrent events and behavior can be clearly described in Petri net. In addition, the model in a Petri net can describe the non-deterministic behavior of concurrent systems. By providing Petri net specification for test case description, and by constructing test constraints from the test case specification, we can use the test constraints as a testing scenario. Such test constraints are used for controlling a certain degree of deterministic execution.

1.4 Objectives

The primary objective of this thesis is to provide a method for dealing with non-deterministic execution in reproducible testing of DM programs. Based on the three steps of reproducible testing, this method includes:
• MSC-based extended design notation - PMSC (Parallel Message Sequence Chart) for Java DM programs.

• Test cases (test constraints) based on Petri-net that force a certain degree of determinism but not full determinism.

• Test control mechanism for reproducible testing based on our PMSC model and Test constraints.

Another primary objective of this thesis is to provide a prototype to perform reproducible testing of DM programs. The reason for developing a prototype of reproducible testing system is to help software testers using the prototype to automatically control the execution of DM programs under given test constraints and to evaluate the performance of our approach.

1.5 Thesis Structure

This thesis consists of ten chapters. Chapter 2 discusses Java distributed multithreaded architecture and non-deterministic behaviors in Java DM programs. Chapter 3 describes extended design notation - PMSC, which is based on MSC (Message Sequence Chart). Chapter 4 defines test case and test case representation based on Petri net. Chapter 5 represents our reproducible testing process. Chapter 6 discusses the code insertion and test case control mechanism based on our PMSC model and Test Constraints. Chapter 7 discusses the design and implementation of a prototype of reproducible testing in DM programs. Chapter 8 shows the results of the evaluation of our approach. Chapter 9 describes the comparison of this work with related work done so far by other researchers. Finally, Chapter 10 provides conclusions and recommendations for future work.
Chapter 2

JAVA DISTRIBUTED MULTITHREADED (DM) ARCHITECTURE AND NON-DETERMINISTIC BEHAVIOR

Java [10] is a general purpose, object-oriented language that has received a growing interest for developing platform-independent concurrent and distributed applications over Internet environments. This chapter overviews Java Distributed Multithreaded (DM) constructs, architecture, and the non-deterministic behavior in Java DM programs. An example of a Java DM program - Distributed Bakery Algorithm - is also given.

2.1 Java Distributed Multithreaded (DM) Constructs

The Java language enables the development of concurrent and distributed software through the constructs of multi-threading and distributed programming such as sockets, Java RMI and Java IDL. This section will introduce all these constructs in Java DM programs.

2.1.1 Concurrency in Java – Multithreading

In Java, threads are special objects that can run independently and concurrently with the main process. Threads start their execution with the start() method is called. The main
thread is already running when the program is started. When a thread is started, the thread behavior is executed in its run() method. Since only the main thread is running initially in Java DM programs, the main thread must create and start some of the other threads. These threads may then create and start other threads. In addition, one thread may be suspended by calling its join() method.

The interactions between threads occur through shared memory with synchronized thread statements (method calls) based on the monitor concept. A monitor is a portion of code in which only one thread is allowed to run at a time. Java implements a monitor with synchronized statements (methods) and locks. Each Java object has an implicit lock, which can be used by synchronized statements (methods). To execute a synchronized statement, a thread must acquire the lock of the object indicated in the synchronized block and release this lock when it exits this synchronized block.

Threads may be interrupted and awakened in monitors by calling the wait(), notify() and notifyAll() methods. When a thread calls the wait() method, it releases the lock allowing other threads to obtain this lock, thus putting itself in the waiting queue of the monitor and becoming inactive. The inactive threads in the waiting queue may be awakened only when some other threads execute one of notify() and notifyAll() methods of the lock object. The difference between these two methods is that notify() wakes up one thread while notifyAll() wakes up all the threads in the waiting queue. The wait(), notify() and notifyAll() methods must take place inside monitors for the corresponding locks.

In summary, we consider monitor, start(), join(), wait(), notify() and notifyAll() as thread communication methods.
2.1.2 Java Distributed Programming

Distributed systems require that computations running in different address spaces be able to communicate, especially on different machines. Java provides several communication mechanisms such as socket, Java RMI (Remote Method Invocation), and Java IDL (Interface Definition Language) with CORBA.

2.1.2.1 Java RMI (Remote Method Invocation)

Sockets require the processes to engage in application-level protocols to encode and decode messages for exchange, and the design of such protocols is difficult and sometimes error-prone. Instead of working directly with sockets, the interprocess communication can be developed using Java Remote Method Invocation (RMI) [21]. RMI can be used to build distributed systems and it allows one to invoke methods on other Java Virtual machines.

The RMI system is similar to but more general and easier to use than the Remote Procedure Call (RPC) mechanisms. RMI is a high level communication mechanism which provides an interface of remote objects, and allows objects to invoke a remote method on another Java Virtual machine. The remote method invocations are transparent since they are identical to local ones. RMI also provides some valuable features for developing distributed applications, such as distributed garbage collection, dynamic loading, object serialization, convenient access to streams etc. Figure 2.1 shows the RMI Model.
The user application is at the top layer of the RMI system. It allows objects to transparently communicate with remote objects, possibly on other Java Virtual Machines. The RMI system itself consists of three layers: the Stub/skeleton layer, the Remote Reference layer and the Transport layer. A remote method invocation from a client to a remote server object travels down through the layers of the RMI system to the client-side transport, then up through the server-side transport to the server.

![RMI Model Diagram]

**Figure 2.1. The RMI Model**

The Stub/Skeleton layer serves as an interface between an application and the rest of the RMI system. It consists of two parts: client-side stubs (proxies) and server-side skeletons, which allow transfer of data to the Remote Reference Layer via marshal streams. The remote reference layer is responsible for carrying out the semantics of the invocation. It transmits data to the Transport layer using connection-oriented streams. The transport layer is responsible for setting up connections, managing those connections, and keeping track of and dispatching to remote objects.
In RMI, the implementation of a remote method invocation may or may not be executed in a separate thread. Some calls which originate from the same client virtual machine will execute in the same thread; some will execute in different threads. Calls, which originate from different client virtual machines, will execute in different threads.

2.1.2.2 Java IDL (Interface Definition Language) with CORBA

Java IDL adds CORBA (Common Object Request Broker Architecture) capability to the Java platform, providing standard-based interoperability and connectivity through method invocation on remote network services. Java IDL enables distributed Web-enabled Java applications to transparently invoke operations on remote network services using the industry standard OMG IDL (Object Management Group Interface Definition Language) and IIOP (Internet Inter-ORB Protocol) defined by the Object Management Group.

Figure 2.2 illustrates the structure of Java IDL with CORBA Implementation. Interfaces for remote objects are defined in the standard CORBA Interface Definition Language (IDL). The IDL compiler compiles the IDL and generates IDL stubs for client application and an IDL skeleton for server-side object implementation. The object implementation is registered in the CORBA Name Service. The client can get the object reference through the Name Service. When a remote method is called, the client request is passed into the IDL stub that directs the request to the ORB. The ORB uses the object reference to locate the object implementation and then delivers the request to the IDL skeleton, where the request is then passed to the object implementation. Any return values
are passed back through the skeleton to the ORB. Then the ORB returns the values through the IDL stub to the client application.

The difference between Java RMI and Java IDL is that Java RMI only supports the homogeneous environment of the Java Virtual Machine, while CORBA can be run in a heterogeneously distributed environment.

![Diagram](image)

Figure 2.2. The Structure of Java IDL with CORBA Implementation

### 2.2 Java Distributed Multithreaded (DM) Architecture

A distributed multithreaded (DM) Java program consists of a set of multithreaded processes, which may run on different Virtual Machines. In each process, multiple threads can be executed concurrently and independently. The interactions between two threads are
via shared-memory communication with synchronized method calls; the communication between two processes is achieved by socket communication, Java Remote Method Invocation (RMI) and Java IDL with CORBA etc. The detailed architecture of Java DM programs is illustrated in Figure 2.3. The bars $t_1$, $t_2$, $t_3$ represents the threads and the circles $o_1$ and $o_2$ represents synchronized or remote objects.

![Figure 2.3. Distributed Multithreaded Architecture](image)

In this thesis, we make the following assumptions about our Java DM programs:

1. Threads can be dynamically created during the program execution.
2. The interaction between two threads are monitor-based synchronized block (methods), and the synchronization is realized via start(), notify(), notifyAll() and wait() methods.
3. We use Java RMI or Java IDL with CORBA implementation as the communication means between two processes. Based on Java RMI and Java IDL, the
remote method calls are treated as local method calls. There is no difference of interactions between two threads from two processes and from one single process.

(4) There may be more than one process in the program.

Formally, a Java DM program \( P \) can be defined as follows:

\[
P = (N_1, N_2, \ldots, N_n)
\]

where \( N_i (1 \leq i \leq n) \) represents a running process.

\[
N_i = (T_{i1}, T_{i2}, \ldots, T_{im})
\]

where \( T_{ij} (1 \leq i \leq n, 1 \leq j \leq m) \) represents the \( j^{\text{th}} \) thread in process \( N_i \).

We use \( \text{numProcesses}(P) \) to denote a function that returns the number of processes in \( P \) and \( \text{numThreads}(N_i) \) to denote a function that returns the number of threads in process \( N_i \).

### 2.3 Non-deterministic Behavior

In Java Distributed Multithreaded (DM) programs, non-determinism comes from the concurrent thread executions and the interactions between threads. Non-determinism can be demonstrated by running the same program with the same input and observing different results and behavior. For example, two threads update a shared variable \( V \) concurrently, in which \( V \) is protected by monitor object \( O \). Depending on the order that two threads update the variable \( V \), its value may be different. In other words, depending on the execution
sequences, we may not get the same results or observe different behavior with the same input data.

The non-deterministic behavior makes regression testing and debugging of distributed multithreaded (DM) programs much more difficult. The problem is that the program may produce different results with different execution sequences. We cannot produce the previous execution sequence by a simple re-execution of program with the same test data when we perform regression testing. Even though we get the expected result, we cannot guarantee that the program is correct.

2.4 An Example - Distributed Bakery Algorithm (DBA)

In the Distributed Bakery Algorithm (DBA) [11], several nodes compete for entering a critical section. Each node has an identification number and maintains a number, which is currently used for making a decision to enter the critical section. When a node wishes to enter the critical section, it chooses a number, broadcasts its choice to the other nodes, and then waits until it has received confirmation from each other node that the number chosen is now the lowest outstanding number. If the chosen number is the same as the number in other nodes, the node with the lowest identification number receives confirmation. The DBA algorithm includes three tasks:

- Compete for entering critical section when requested.
- Send Request to other nodes for comparison.
- Send Reply to other nodes if those nodes have lower outstanding numbers.
The DBA implementation consists of multiple processes, one for each node; each process consists of more than one thread for a different task. Hence in this thesis, we chose DBA as our typical example of Distributed Multithreaded (DM) programs. For simplicity, here we consider two nodes.

2.4.1 Java Implementation of DBA Algorithm

In Java, the nodes are implemented as separate processes which may run in different Virtual machines. The communication between two nodes is via Java IDL with CORBA implementation. Java source code of this example is given in Appendix A. Each node has two threads and two remote objects. The structure of the algorithm is shown in Figure 2.4. The rectangles represent threads, the ovals represent remote objects, the arrows represents method calls. In this figure, we have

- **Main Process** thread which accepts users' requests and executes the critical section.
- **Send Request** thread which sends the messages to Request object.
- **Request Object** which does the comparison and makes a decision whether to defer or reply to other nodes.
- **Reply Object** which sends a reply to other nodes for their requests to enter the critical section.

The three tasks that comprise the DBA algorithm for each node access shared variables that maintain state information about the node. These shared variables are implemented in a Java Node class (see Node.java in Appendix A). Each node has one instance of this class, containing the following information:
• Node Identification (ID): Node 1 has lower identification number than Node 2.
• Number: The number chosen by this node for comparison.
• Requesting: A flag to indicate whether this node is requesting its critical section. This is used to skip the comparison of sequence numbers if a node is not competing for the critical section.
• ReplyCount: Used to count received Reply messages so that the node knows when it is allowed to enter the critical section.
• Deferred: An array indicating which nodes have had their replies deferred.

The Node object is treated as a monitor. The mutual exclusion on these shared variables is protected by calling synchronized blocks (methods) on this monitor object.
2.4.2 The Non-deterministic Behavior in DBA

In this example, the non-deterministic behavior is described by observing two different results with the same input. The input in this example is the number we choose.

- *Node 1 enters critical section first.*
- *Node 2 enters critical section first.*

*Case 1:* Node 1 enters the critical section first. There are three scenarios in this case.

*Scenario 1:*
- Nodes 1 & 2 choose a number simultaneously.
- Nodes 1 & 2 propagate their number to the other node.
- Nodes 1 & 2 wait until they receive a reply from the other node.
- By comparison, both nodes have the same number but Node 1 has a lower identification number, so Node 2 is deferred.
- Node 1 receives a reply from Node 2 and enters critical section.
- Node 1 sends a reply to Node 2.
- Node 2 receives a reply from Node 1 and enters critical section.
- Node 1 & 2 again choose a higher number.

*Scenario 2:*
- Node 1 chooses a number first and propagates it to Node 2.
- Then Node 2 chooses a number and propagates it to Node 1.
- Nodes 1 & 2 wait until they receive reply from the other node.
• By comparison, Node 1 has a lower number, so Node 2 is deferred.
• Node 1 receives a reply from Node 2 and enters critical section.
• Node 1 sends a reply to Node 2.
• Node 2 receives a reply from Node 1 and enters critical section.
• Node 1 & 2 again choose a higher number.

Scenario 3:
• Node 1 chooses a number but not Node 2.
• Node 1 propagates the number to Node 2.
• Node 1 waits until it receives a reply from Node 2.
• Since Node 2 does not request entering critical section, it sends a reply directly to Node 1.
• Node 1 receives a reply from Node 2 and enters critical section.
• Node 1 again chooses a higher number.

Case 2: Node 2 enters the critical section first. There are two scenarios in this case.

Scenario 1:
• Node 2 chooses a number first and propagates it to Node 1.
• Then Node 1 chooses a number and propagates it to Node 2.
• Nodes 1 & 2 wait until they receive a reply from the other node.
• By comparison, Node 2 has a lower number, so Node 1 is deferred.
• Node 2 receives a reply from Node 1 and enters critical section.
• Node 2 sends a reply to Node 1.
• Node 1 receives a reply from Node 2 and enters critical section.
• Nodes 1 & 2 again choose a higher number.

Scenario 2:

• Node 2 chooses a number but not Node 1.
• Node 2 propagates the number to Node 1.
• Node 2 waits until it receives a reply from Node 1.
• Since Node 1 does not request entering critical section, it sends a reply directly to Node 2.
• Node 2 receives a reply from Node 1 and enters critical section.
• Node 2 again chooses a higher number.
Chapter 3

EXTENDED DESIGN NOTATIONS – PMSC (PARALLEL MESSAGE SEQUENCE CHART)

This chapter gives an overview of MSC (Message Sequence Chart) [13,14,15,16] and discusses the advantages of using MSC. Then it describes our extended design notation - PMSC (Parallel Message Chart), which is based on MSC for representing the static analysis model of Java DM program.

3.1 Overview of MSC (Message Sequence Chart)

Message sequence chart (MSC) is an easy and intuitive way for describing the behavior of a system by viewing the interaction between the system and its environment [16]. This language is standardized according to ITU recommendation Z.120 [13,14] by the ITU-TS (the Telecommunication Standardization section of the International Telecommunication Union, the former CCITT) [25].

The MSC recommendation Z.120 was approved by ITU twice in 1992 and 1996. MSC'92 [13] contains a description of an abstract syntax, a graphical syntax, and a textual syntax of the language MSC, called Basic Message Sequence Chart (BMSC). BMSC is a finite collection of instances with a detailed description of the asynchronous communication between instances. The language of BMSC includes all constructs that are necessary for specifying the pure message flow. MSC'96 [14] introduces the new
constructs of MSC including new concepts of composition and object orientation. The main constructs of MSC are instance, message, environment, action, composition, object-oriented concept etc. Figure 3.1 illustrates a graphical representation of telephone call in MSC.

![MSC Telephone Call Diagram](image)

**Figure 3.1. A Graphical Representation of Telephone Call in MSC [20]**

- **Instance**: is an abstract entity on which message inputs, message outputs and local actions may be specified. An instance contains three sub-elements: instance head, instance axis and instance end. In the graphical representation, the instance heading is represented as a named rectangle at the top, the instance end is represented as a filled-rectangle at the bottom. The vertical lines (instance axis) denote the instance process control flows. The time along each line is running from top to bottom.
• **Message**: describes an asynchronous message being sent and received between two instances. This can be represented by a horizontal arrow which starts at the sending instance and ends at the receiving instance.

• **System Environment**: is graphically represented by the frame symbol which forms the boundary of an MSC diagram. In contrast to instances, there is no ordering of communication events within the environment.

• **Actions**: describes an internal activity of an instance. An action is graphically represented by a solid rectangle containing the action name.

• **Instance Creation and Termination**: are quite common events since instances may be created or terminated dynamically during system run time. In the graphical representation, the create symbol is a dashed arrow which may be associated with textual parameters. This arrow originates from a parent instance and points at the instance head of the child instance. The termination of an instance is graphically represented by a stop symbol in the form of a cross at the end of the instance axis.

• **Composition**: In addition to the sequential composition of MSC based on conditions, some other composition mechanisms are proposed by means of process algebra operators with MSCs as arguments. The Solid rectangle blocks with keywords *seq, alt, par, loop, opt, exc* etc. in the left upper corner graphically represent six types of system composition (*sequence, alternative, parallel, repetition, optional region and exception*).
• **Object-Oriented Concepts**: The object and object behavior are explicitly described in the MSC. For example, the objects are represented as instances at the top. Each instance axis and message between them describes an object's behavior.

The major strength of MSC is that its clear graphical representation gives an intuitive understanding of the described system behavior. Another strength is that the standardization of formal semantics allows for systematic tool support and facilitates the message exchange between different tools. Due to the standardization, MSC can be used as [20]:

• *a semi-formal functional requirement specification*
• *an interface specification*
• *an overview of a service*
• *a basis for simulation and validation*
• *a basis for automatic test case generation.*

3.2 Extended Design Notations – PMSC (Parallel Message Sequence Chart)

Since Java is an object-oriented paradigm, static data flow analysis techniques [1,2,4,5] are not sufficient for describing the object behaviors. In this research work, we propose an extended design notation - PMSC (Parallel Message Sequence Chart) based on MSC for representing our static analysis model on Java DM programs, since MSC supports object-orientation and has user-friendly graphical representation. In PMSC, we
adopt some basic concepts of MSC, and extend it with some new ones for describing the Java monitor-based interaction and synchronization.

A PMSC is a collection of extended MSCs. The behavior of each thread is represented by an extended MSC. The detailed extension of PMSC model is described as follows:

- Thread method calls (*start, notify, notifyAll, wait* etc.) are represented as messages in horizontal arrows.
- The internal actions with keywords *notify, notifyAll, and wait* denote thread synchronization.
- Dotted rectangle blocks with monitors in the upper left corner represent synchronization blocks (methods).

Figure 3.2 & Figure 3.3 illustrate the PMSCs of the DBA algorithm.
Figure 3.2. PMSC of MainProcess1 in DBA algorithm
Figure 3.3. PMSC of SendRequest1 in DBA Algorithm
Chapter 4

TEST CASE SPECIFICATION IN PETRI NET AND TEST CONSTRAINTS

In this chapter, we give a brief review of Petri net and introduce an approach for test case specification using Petri net which can describe a certain degree of determinism. Then we define test constraints in term of concurrent events and their relations from the test case specification in Petri net as a test scenario for our testing approach.

4.1 Overview of Petri-net

For many years, Petri nets [22,23] have been used for modeling the behavior of various types of concurrent systems. Petri nets are a graphical representation for specifying concurrent systems. The concurrent events and behavior of the concurrent program can be clearly described in Petri net.

Formally, a Petri net is defined as a 5-tuple [23]

\[ PN = (P, T, A, W, M_0) \]

where

- \( P \) is a finite set of places. A place can contain any number of tokens. The place is marked when it contains tokens.
- $T$ is a finite set of transitions.
- $A$ is a finite set of arrows either from places to transitions or from transitions to places.
- $W$ is a weight function that associates a positive integer with each arrow of $A$. By default, the weight of an arrow is one.
- $M_0$ is the initial marking.

Graphically, a Petri net is a directed graph. Places are represented by circles, transitions by bars, and tokens by dots within the places. The state of a Petri net is represented by marking its places, which is graphically inserting a number of tokens in the places of the net. Figure 4.1 shows a Petri net.

![Petri Net Diagram](image)

**Figure 4.1. A Petri Net [22]**

In a Petri net, a transition models an event or an action. A place models a condition. A transition is enabled if the conditions in its places allow the occurrence of the modeled event or action. The presence of a token in a place denotes the existence of some
condition. A transition may fire, which represents the occurrence of an event or execution of an action. A firing sequence of a given PN with a given initial marking is a sequence of transition firings, denoted by \(<t_1, t_2, \ldots, t_n>\), such that \(t_1\) is enabled in the initial marking, \(t_2\) is enabled in the marking obtained by firing \(t_1\), and so on [22].

The model in a Petri net can describe the non-deterministic behavior of concurrent systems, which means that given an initial marking, different evolutions of the Petri net are possible. For example, the Petri net of Figure 4.1 is interpreted as two independent activities, flowing through two sequences of transitions \(<t_1, t_3, t_5>\), and \(<t_2, t_4, t_6>\). The two activities share a common resource in place \(P_3\). Initially, transitions \(t_1\) and \(t_2\) are enabled concurrently. Both activities proceed in mutual exclusion. Depending on the order of acquiring the shared resource, either activity \(<t_1, t_3, t_5>\) is enabled or activity \(<t_2, t_4, t_6>\) is enabled. The choice is non-deterministic.

### 4.2 Test Case Specification in Petri-net

Test cases can be represented in different ways. The test sequence [6,7] is a sequence of events that corresponds to flow control statements for the program. Another test case is a set of paths [4,5,8] from data flow graphs, where each path represents an event sequence in one thread. In such a context, each thread has a path and the edges between paths represent the interactions between threads. All these kinds of test cases describe event relations in total order. That means there is only one event that enables another event (only one event).
Since Petri net can describe non-deterministic behavior of concurrent systems, we can use Petri net to specify a test case with a certain degree of deterministic execution. For example, we describe two kinds of test cases for our DBA algorithm in the Petri nets of Figure 4.2. Figure 4.2 (a) shows the case in which Node 1 enters the critical section first, and Figure 4.2 (b) shows the case in which Node 2 enters the critical section first.

In a Petri net specification, the places model resources or conditions. The bars model the events that correspond to method calls in PMSC. An event is represented by:

\[(\text{threadName}, \text{objName}, \text{methodName}, \text{number})\]

where
- \text{threadName} is the thread name in which the method is called. The \text{threadName} is denoted by \text{Pid:Tid}, where \text{Pid} is the process identifier in which the thread is created, and \text{Tid} is the thread identifier within process \text{Pid}.
- \text{objName} is the name of the object in which the method call is called. The \text{objName} is denoted by \text{Pid:Oid}, where \text{Pid} is the process identifier in which the object is instantiated, and \text{Oid} is the object identifier.
- \text{methodName} is the name of the method being called.
- \text{number} is the number of appearances of \text{methodName} being called by \text{objName} in the thread \text{threadName}.

The relations between the events are represented by double-headed arrows from a set of places \(P_1\) to event \(t\) and from event \(t\) to another set of place \(P_2\). In this test case, more than one event is enabled after a set of events is fired. The arrows going from places \(P\) to
transition $t$ specify that the conditions in $P$ are satisfied, and then event $t$ is fired. The arrows going from transition $t$ to places $P$ specify that after event $t$ is fired, the resources and conditions for next event are available in places $P$.

4.3 Test Constraints

Since the non-determinism comes from thread interaction and thread synchronization, we only consider the events that correspond synchronized methods (statements), remote methods and thread interaction methods ($\text{wait()}$, $\text{notify()}$, $\text{notifyAll()}$ etc.), called concurrent events. A concurrent event is represented by

$$(\text{threadName}, \text{monitorName}/\text{remoteObjectName}, \text{methodName}, \text{number})$$

where

- $\text{threadName}$ is the thread name in which the method is called. The $\text{threadName}$ is denoted by $\text{Pid:Tid}$, where $\text{Pid}$ is the process identifier in which the thread is created, and $\text{Tid}$ is the thread identifier within process $\text{Pid}$.

- $\text{monitorName}/\text{remoteObjectName}$ is the name of the monitor or remote object in which the method is called. The $\text{monitorName}/\text{remoteObjectName}$ is denoted by $\text{Pid:Oid}$, where $\text{Pid}$ is the process identifier in which the monitor or remote object is instantiated, and $\text{Oid}$ is the monitor or remote object identifier.

- $\text{methodName}$ is the name of the method being called.

- $\text{number}$ is the number of appearances of $\text{methodName}$ being called by $\text{objName}$ in thread $\text{threadName}$.
Figure 4.2 (a) Example of Test Case Specification in Petri net – Case 1
Figure 4.2 (b) Example of Test Case Specification in Petri net – Case 2
The method name can be one of the following event types:

- synchronized block (statements)
- call for synchronized method
- call for start(), wait(), notify(), notifyAll()
- remote procedural call or remote method invocation.

From the Petri net specification, our testing process is interested in test scenarios in term of concurrent events and their relations. We represent such test scenarios as test constraints, which are:

\[(V_s, E_s)\]

where

- \(V_s\) is a set of concurrent events in the Petri net specification;
- and \(E_s\) is a set of relations among events \(V_s\). The relations are represented by a set of events \(F\) and another event \(e\), in a way that event \(e\) is enabled once all events in \(F\) have been fired. There may be more than one set of events to enable the same event.

We also define two operations over the test constraints:

1. \(Before(e)\): all sets of event \(F\) which enable event \(e\).
2. \(After(e)\): all events which happen after event \(e\).

For example, from the PMSC model in Figure 3.2 & Figure 3.3, and the test case specification in Figure 4.2, we may find that our testing process is interested in the test constraint in term of the following five concurrent events and their relations:
$e_1$: ($P_1$:mainProcess, $P_1$:node, chooseNo, 1)
$e_2$: ($P_2$:mainProcess, $P_2$:node, chooseNo, 1)
$e_3$: ($P_1$:sendRequest1, $P_2$:request, requesting, 1)
$e_4$: ($P_1$:sendRequest1, $P_1$:reply, replying, 1)
$e_5$: ($P_1$:mainProcess, $P_2$:reply, replying, 1)

Their relations are expressed by the following partial order:

![Diagram](image)

Figure 4.3. Example of Test Constraint

where nodes represent concurrent events and arrows represent a happen-before relationship between two concurrent events.

In order to use test constraints efficiently for our test control, we represent test constraints by separating events in different monitors and remote methods. Events in each monitor or each remote method are in sequence by total order. The relations between events in different monitors or remote methods are partial order. Figure 4.4 shows this representation of the example in Figure 4.3.
Figure 4.4. Test Constraint Representation
Based on Monitor and Remote Method
Chapter 5

REPRODUCIBLE TESTING PROCESS

Since there exists non-deterministic behavior in Java DM programs, new techniques should be provided to deal with the regression testing of DM programs. One way is to perform a controlled execution of the program, by having a separate execution control mechanism that ensures a given running situation. Reproducible testing is a method that reproduce a previous testing run and force an intended path to be covered during this particular testing run. Hence, reproducible testing is one of the key issues of distributed multithreaded (DM) program testing.

With our extended design notation - PMSC, test case specification in Petri net and test constraints, the structure of our reproducible testing model for distributed multithread Java programs is shown in Figure 5.1. In our model, we have the following components:

1. Test Constraint Construction Module (TCC) constructs feasible test constraints with the help of PMSC and the test case specification in Petri net model.
2. Code Insertion Module (CIM) generates an extended source program (called EP) to be tested. Based on the source program, additional program code is inserted. The inserted code will perform the communication with Test Controllers. The new source program is logically equivalent to the original program.
3. Test controllers (TCs) controls actual execution of the DM program with our test control mechanism.
(4) Run extended programs (REPs) and examine the results.

Figure 5. Reproducible Testing Process

In our approach, we focus on step 2 and 3, developing a Code Insertion Module (CIM) and Test Controllers (TCs) to monitor the testing for a given test case represented in test constraints. We assume that the PMSC model already exists in the design document and test constraints are already constructed and are feasible for our testing.
Chapter 6

CODE INSERTION AND TEST CONTROL ALGORITHM

This chapter presents the main contribution of this thesis, the test control mechanism and algorithm in reproducible testing of Java DM program. It first discusses the control primitives between the extended DM program and test controllers, then a code insertion module (CIM) is provided to transfer the original program to logically equivalent extended program. Finally, The structure and algorithm of test controllers (TCs) are designed for controlling the execution of the extended program with given test constraints.

6.1 Event Types

As we know, the non-determinism in DM programs comes from concurrent thread executions and interactions between threads. In Java, these are protected by synchronized statements (methods) on monitor objects. The synchronization between two threads in different processes is implemented by remote method invocation (RMI) or Java IDL, where this remote method is either synchronized method or non-synchronized method that contains synchronized statements (methods).

To distinguish different types of synchronization and thread interaction, we define event types (ET) that correspond to different concurrent constructs as:

\[ ET = (SYN, SYNREM, INTER, REMOTE) \]
where

- **SYN**: synchronized block (statements) or synchronized method call.
- **SYNREM**: synchronized remote method call.
- **INTER**: the wait() method call. Since each wait() method must has corresponding notify() or notifyAll() methods, we only consider the wait() call as an event.
- **REMOTE**: remote method invocation that is not a synchronized method, but it contains synchronized statements (methods).

### 6.2 Control Primitives

Similar to other reproducible testing model [6,7], to control the program execution with given test constraints, the original program must be converted into an extended program (EP) which has additional communication codes and is logically equivalent to the original program. The inserted codes are control primitives, which are special communication methods of test controllers, for controlling the communication between the processes of the extended program and Test Controllers (TCs). In our approach, there are two types of control primitives.

- **Request**: Send the request event to test controller and wait for permission for executing the corresponding statement (method).
- **Complete**: Send the complete event to test controller to notify the completeness of executing the corresponding statement (method).

For different event types, the detailed control primitives are described as follows:
(1) For *SYN* and *SYMREM* types, there are two control primitives:

- *synRequest (threadName, monitorName, methodName)* - sends a request event to the test controller. It is inserted before the corresponding synchronized block (statements), synchronized method or synchronized remote method call.

- *synComplete (threadName, monitorName, methodName)* - sends a complete event to the test controller. It is inserted before the first statement of the corresponding synchronized block (statements), synchronized method or synchronized remote method.

(2) For *REMOTE* type, there are two control primitives:

- *synRequest (threadName, remoteObjectName, methodName)* - sends a request event to the test controller. It is inserted before the corresponding non-synchronized remote method call, which contain synchronized statements (methods).

- *synComplete (threadName, remoteObjectName, methodName)* - sends a complete event to the test controller. It is inserted before the first statement of the corresponding non-synchronized remote method, which contains synchronized statements (methods).

(3) For *INTER* type, there are two control primitives:

- *waitRequest (threadName, monitorName, wait)* - sends a request event to the test controller. It is inserted before the `wait()` method call.

- *waitComplete (threadName, monitorName, wait)* - sends a complete event to the test controller. It is inserted after the `wait()` method call.

Here, *threadName* is the thread name in which the event is triggered, *monitorName/remoteObjectName* is the name of the monitor or remote object in which the
event is triggered, methodName is the name of the method being called. The event, which is used by test controllers to control the execution, consists of a 4-tuple (threadName, monitorName/remoteObjectName, methodName, number). The first three elements are passed from the control primitives, and the last element is constructed in test controllers.

The communication protocols between processes of extended programs (EPs) and test controllers are shown in Figure 6.1. For the SYNREM and REMOTE event types, the communication is involved in two processes. Thus, the request control primitive is sent from the test controller of calling process to the test controller of the process with remote method being called. The complete control primitive is sent directly from the process with remote method being called to its controller.

6.3 Code Insertion Module (CIM)

To create an extended program, the Code Insertion Module (CIM) inserts the control primitives before and after each corresponding statement or method call. In addition, to pass information to or receive information from remote objects, some changes should be made on the specification of remote objects. Appendix B shows an example of the extended program for the DBA algorithm in Appendix A. In this section, we discuss the details of code insertion and give the rules of code insertion in BNF form.

6.3.1 Details of Code Insertion

Assume test controllers TC1,...,TCn on Process P1,...,Pn. The details of code insertion for different event types are described as follow:
(1). Synchronized block on Process $P_i$ (SYN type):

\[ \text{synchronized (o) \{ P \}} \text{, where P is a set of methods or statements.} \]

is translated into

\[ TC_i\text{.synRequest(Thread.currentThread().getName(), o.getMonitorName(), synblock);} \]

\[ \text{synchronized (o) \{} \]

44
TCi.synComplete(Thread.currentThread().getName(), o.getMonitorName(), synblock);

P;
}

The translation is done by inserting the synRequest primitive before the synchronized block (statements) and inserting the synComplete primitive before the first statement of synchronized block (statements). Additional code should be added into the program:
- Each thread should be explicitly given a name, which consists of ProcessId and threadId. The name is retrieved by calling getName() method.
- Each class with synchronized method should have an attribute for the monitor name, and a getMonitorName() method to retrieve this name.

(2). Call for synchronized method on Process Pi (SYN type):

   o.synMethod(...), where synMethod(...) is a synchronized method

is translated into

   TCi.synRequest(Thread.currentThread().getName(), o.getMonitorName(), synMethod);

   o.synMethod(...);

and each synchronized method

   synMethod(...) { P }

is translated into
synMethod(...) {
    TCi.synComplete(Thread.currentThread().getName(), getMonitorName(),
    synMethod);
    P;
}

The translation is done by inserting the synRequest primitive before the synchronized
method call and inserting the synComplete primitive before the first statement of the
synchronized method.

(3). Process Pi call for synchronized remote method on Process Pj (SYNREM type):
    o.synMethod(...), where synMethod(...) is a synchronized remote method

is translated into
    TCi.synRequest(Thread.currentThread().getName(), o.getMonitorName(),
    synMethod);
    o.synMethod(..., Thread.currentThread().getName());

and each synchronized method
    synMethod(... ) { P }

is translated into
    synMethod(..., String threadName) {
        Thread.currentThread().setName(threadName);
    }
TCj.synComplete(Thread.currentThread().getName(), getMonitorName(),
synMethod);

P;
}

The translation is done by inserting the synRequest primitive before the synchronized remote method call and inserting the synComplete primitive before the first statement of the synchronized remote method. Additional code is added into the program:
- To retrieve the monitorName of remote object o on process Pi, the specification of getMonitorName() method should be added to the interface of remote object o.
- When the synMethod(...) is called, a separate thread on process Pj is created to perform the invocation of this remote method. To retrieve the threadName of process Pi, the specification of synMethod(...) is changed to: synMethod(..., Thread.currentThread().getName()).

(4). Process Pi call for o.remMethod() on process Pj, where o is a remote object and m is not a synchronized method but contains synchronized statements (methods) (REMOTE type)

is translated into

TCi.synRequest(Thread.currentThread().getName(), o.getName(),
remMethod);
o.remMethod(..., Thread.currentThread().getName());

and remMethod(...) { P } is translated into
47
remMethod(..., String threadName) {
    Thread.currentThread().setName(threadName);
    TCi.remComplete(Thread.currentThread().getName(), getRemoteName(),
             remMethod);
    P;
}

The translation is done by inserting the remRequest primitive before the remote method call and inserting the remComplete primitive before the first statement of the remote method. Additional code should be added into the program:
- Each class with non-synchronized remote method should have an attribute for the remote object name, and have a getRemoteName() method to retrieve this name.
- The specification of getRemoteName() method should be added to the interface of remote object o.

(5). Call for wait() within monitor o on process Pi (INTER event)

is translated into

    TCi.waitRequest(Thread.currentThread().getName(), o.getMonitorName(), wait);
    wait();
    TCi.waitComplete(Thread.currentThread().getName(), o.getMonitorName(), wait);
The translation is done by inserting the `waitRequest` primitive before the `wait` method call and inserting the `syncComplete` primitive before the first statement of the `wait` method call.

### 6.3.2 Rules of Code Insertion

The translation from an original program to the logically equivalent extended program needs to follow some rules. BNF or a context-free grammar is a natural notation for describing the syntax of a programming language. Many programming language constructs such as if-else, while-loop, assignment, expression, statement, variable declaration etc. can be simply described in BNF form. For example, the definition of `<assign>` may be given by

\[<assign> \rightarrow <var> = <expression>\]

where the symbol on the left side (LHS) of the arrow is the abstraction being defined. The text to the right of the arrow is the definition of the LHS.

The normal way to describe the rules of translation is to embed some semantic actions in the right side of BNF grammar. Figure 6.2 shows the rules of code insertion in which the inserted code is described as an `insert` action by enclosing it between braces.

### 6.4 The Structure of Test Control Mechanism

To deal with the non-deterministic behavior during the program execution, we convert
Figure 6.2 The Rules of Code Insertion

\[
\text{<synBlock> } \rightarrow \{ \text{insert(<synRequest>)} \}
\text{ synchronized (<objectName>){}
\{ insert(<synComplete>) \}
\text{<stmt-list> }
\}
\]

\[
\text{<synMethodCall> } \rightarrow \{ \text{insert(<synRequest>)} \}
\text{<objectName>.<methodName>(<argument-list>);}
\]

\[
\text{<synRemMethodCall> } \rightarrow \{ \text{insert(<synRequest>)} \}
\text{<objectName>.<methodName>(<argument-list>,
\{ insert(<getThreadName>) \});}
\]

\[
\text{<remMethodCall> } \rightarrow \{ \text{insert(<remRequest>)} \}
\text{<objectName>.<methodName>(<argument-list>,
\{ insert(<getThreadName>) \});}
\]

\[
\text{<symMethodDef> } \rightarrow \text{<scope> synchronized <type> <methodName>(<parameter-list>)
\{ insert(<synComplete>)}
\text{<stmt-list> }
\}
\]

\[
\text{<symRemMethodDef> } \rightarrow \text{<scope> synchronized <type> <methodName>(<parameter-list>,
\{ insert("String threadName") (}
\{ insert(<setThreadName>)
\{ insert(<synComplete>) \}
\text{<stmt-list> }
\}
\]

\[
\text{<remMethodDef> } \rightarrow \text{<scope> <type> <methodName>(<parameter-list>,
\{ insert(String threadName) (}
\{ insert(<setThreadName>)
\{ insert(<remComplete>) \}
\text{<stmt-list> }
\}
\]

\[
\text{<waitMethodCall> } \rightarrow \{ \text{insert(<waitRequest>)} \} \text{ wait(); insert(<waitComplete>)}
\]

\[
\text{<synRequest> } \rightarrow \text{Tci.synRequest(Thread.currentThread().getName(), o.getMonitorName(),
\text{<methodName>};}
\]

\[
\text{<synComplete> } \rightarrow \text{Tci.synComplete(Thread.currentThread().getName(), o.getMonitorName(),
\text{<methodName>};}
\]

\[
\text{<remRequest> } \rightarrow \text{Tci.remRequest(Thread.currentThread().getName(), o.getRemoteName(),
\text{<methodName>};}
\]

\[
\text{<remComplete> } \rightarrow \text{Tci.remComplete(Thread.currentThread().getName(), o.getRemoteName(),
\text{<methodName>};}
\]

\[
\text{<waitRequest> } \rightarrow \text{Tci.waitRequest(Thread.currentThread().getName(),o.getRemoteName(), "wait");}
\]

\[
\text{<waitComplete> } \rightarrow \text{Tci.waitComplete(Thread.currentThread().getName(),o.getRemoteName(), "wait");}
\]

\[
\text{<getThreadName> } \rightarrow \text{Thread.currentThread().getName()};
\]

\[
\text{<setThreadName> } \rightarrow \text{Thread.currentThread().setName(threadName);}
\]

\[
\text{<scope> } \rightarrow \text{ public | private | \epsilon}
\]

\[
\text{<type> } \rightarrow \text{<className> | String | int | boolean | void | ...}
\]

\[
\text{<objectName> } \rightarrow \text{<string>}
\text{<methodName> } \rightarrow \text{<string>}
\]

\[
\text{<className> } \rightarrow \text{<string>}
\text{<stmt> } \rightarrow \text{<stmt-list>}
\]

\[
\text{<parameter-list> } \rightarrow \text{\epsilon | <type> <string> | <type> <string> <parameter-list>}
\]

\[
\text{<argument-list> } \rightarrow \text{\epsilon | <string> | <string> <argument-list>}
\]
an original program to an extended program (EP). The extended program (EP) is different from the original program but it is logically equivalent to the original one. Given the test constraints, each process of the extended program communicates with its test controller through the control primitives (see the process of REPs and TCs in Figure 5.1).

The Extended Programs (EPj) consist of a set of processes (EP1, ..., EPn) (n≥2). Each process (Pi) consists of a set of threads (Tli, ..., Tlm) (m≥1), a set of monitor objects (Mli, ..., Mlk) (m≥0) and a set of remote method calls (Rli, ..., Rll) (l≥0). The Test Controllers (TCj) consist of a set of test controllers (TC1, ..., TCn). Each controller (TCj) controls the execution of process (Pi).

The process of test controller (TCi) controls the execution of process (Pi) of extended program (EPi) by using the two types of communication primitives and managing of two queues. Figure 6.3 shows the structure of our test control mechanism.

![Figure 6.3 The Structure of Test Control Mechanism](image-url)

The test control mechanism can be described in the following steps:
(1) Process $P_i$ sends a request event $e$ to its test controller $TC_i$. If event $e$ corresponds to a remote method call defined in process $P_j$, then $TC_i$ sends this event to test controller $TC_j$ of process $P_j$.

(2) $Tci$ checks the condition that enables the request event. If it is satisfied, it returns to process $P_i$ directly. Otherwise, this event is added to $waitQueue$ until the condition is satisfied. The $waitQueue$ is a queue to store the deferred events. When the condition is satisfied, this event is deleted from $waitQueue$ and returns to process $Pi$.

(3) Process $P_i$ executes the approved event.

(4) Process $P_i$ sends the complete event to its test controller $Tci$.

(5) Test controller $Tci$ adds this complete event to its $completeQueue$. The $completeQueue$ is a queue to store the complete events. These events will become the conditions for approving other events.

6.5 Test Control Algorithm and Mechanism

Our test control algorithm is illustrated in Figure 6.4. To control the execution of test constraints, we propose a set of test controllers ($TC$s) where each process has one TC. In this algorithm, the test controller ($TC_i$) on Process $Pi$ provides two types of control primitives (Request, Complete) to communicate with corresponding process ($P_i$). Each test controller uses two queue - "$waitQueue$" and "$completeQueue$".

Whenever a thread on Process $P_i$ requests executing an event, it calls the Request primitives ($synRequest(), remRequest(), waitRequest()$) of its test controller ($TC_i$). The $TC_i$ identifies the controller that this event belongs to (e.g. it is for controlling remote
Figure 6.4. Test Control Algorithm

Initialize variables completeQueue, waitQueue, testConstraint, and controller[1..n];
controller[1..n] = array of test controller objects;
testConstraint = the given test constraint object;
completeQueue = to queue the completed events;
waitQueue = to queue the waiting events;

Request Primitive {
    construct the requested event e from argument;
    if (e is not belong to current process) then
        call the Request Primitive of the corresponding test controller that e belong to;
    else {
        if (e is not one element of testConstraint) then
            return;
        check the conditions that enable event e;
        if ( (exist a set F in testConstraint that enable event e ) \&\&
            (F is a subset of completeQueues of controller[1..n])) then
            return;
        AddToWaitQueue(e);
        wait for notify;
    }
    return;
}

Complete Primitive {
    construct the completed event e from argument;
    if (e is not belong to current process) then
        call the Complete Primitive of the corresponding test controller that e belong to;
    else
        if (e is one element of testConstraint) then
            AddToCompleteQueue(e);
        return;
}

Controller Process {
    while (true) {
        for (each event e in waitQueue) {
            if ( (exist a set F in testConstraint which enable event e ) \&\&
                (F is a subset of completeQueues of controller[1..n])) then {
                notify the waiting event e;
                DeleteFromWaitQueue(e);
            }
        }
    }
}
monitor or remote method call). If it belongs to other controller ($TC_j$), then the event is sent to $TC_j$ by calling the Request primitives of $TC_j$. Otherwise, the test checks the condition that will enable this event. If the condition is satisfied, the Request call is returned and the thread continues its execution. Otherwise, this event is added to the $waitQueue$ and waits for later approval.

Whenever a thread on Process $Pi$ starts executing an event, it calls the $complete$ primitives ($synComplete()$, $remComplete()$, $waitComplete()$) of its test controller ($TC_i$). The $TC_i$ adds this event to the $completeQueue$. The events in the $completeQueue$ will become conditions for approving other events. The reason that the event is sent back after it starts execution is to protect the next event execute first because of the different speed of the processes.

The process of test controller keeps checking each event in the $waitQueue$. If the condition that will enable this event is satisfied, then this event is deleted from $waitQueue$ and returns back to the calling thread to continue execution.

6.6 A Test Control Example

Using our proposed Code Insertion and Test Control Algorithm, we can perform a controlled execution of DM Java programs with given test constraints. For example, given an example of test constraints in Figure 4.4, Figure 6.5 shows the scenario of this control mechanism.
Figure 6.5 (a) Test Control Scenario of Thread P1:main

Figure 6.5 (b) Test Control Scenario of Thread P2:main
Figure 6.5 (c) Test Control Scenario of Thread P1:sendRequest

Figure 6.5 (d) Test Control Scenario of Thread P1:sendRequest
Chapter 7

DESIGN AND IMPLEMENTATION OF A PROTOTYPE OF REPRODUCIBLE TESTING

The design and implementation of reproducible testing systems (RTS) is based on the mechanism and algorithm explained in Chapter 6. It mainly involves the Code Insertion Module and Test Controllers in the process of reproducible testing. Section 6.3 discusses the details of translation from DM programs to logically equivalent extended programs, and Section 6.5 describes the algorithm in which multiple test controllers control the execution of extended programs. The entire system is developed in distributed environment. CORBA provides standard-based interoperability and connectivity through method invocation on remote objects, which can be shared by multiple processes, but locations are transparent. XML (Extended Markup Language) [36] provides a platform-independent way to document test constraints, and allows us to generate test constraint objects directly from the XML documents. According to the test control mechanism and algorithm, this chapter presents a prototype of reproducible testing system for Java DM programs.

7.1 System Requirements

The prototype of Reproducible Testing System (RTS) is implemented using Java, CORBA and XML (Extended Markup Language). There are several reasons to choose Java, CORBA and XML.
(1) Java is a general-purpose, object-oriented programming language, which provides Distributed Multithreaded (DM) constructs such as concurrency, socket communication, Java RMI and Java IDL, etc. It also has many convenient and powerful features for file processing, GUI and string manipulation. Other advantages of Java include platform independence and reusability. Especially, by providing the API with CORBA and XML implementation, Java definitely is the best choice for implementing DM programs and the tool of reproducible testing.

(2) CORBA provides standard interfaces written in OMG IDL, which make multiple processes and remote objects able to run on different platforms across different networks, and allows them to be written in different programming languages.

(3) XML provides a platform-independent way to structure information. The language independent interfaces make it easier for developers to deal with XML. The test cases (test constraints) in an XML document are structured and stored in plain text, which enable different processes on a network to share the information. The Java XML API can easily construct test constraint objects from test constraints in XML documents.

The environment for this implementation requires Windows95/NT or UNIX operating system, JDK1.2 with Java IDL and Java SAX (the Simple API for XML) API, CORBA VisiBroker 4.0 for Java.

7.2 Development of Code Insertion Module
In this thesis, a tool, called Code Insertion Module (CIM) has been developed for automatically generating logically equivalent extended DM programs from Java source programs. It is implemented in three classes: StaticInfo class, Statement class, and CodeInsertion class.

Class StaticInfo stores the static information of Java DM programs, such as the names of process, threads, monitors, remote objects, and synchronized or remote methods. This information is stored in a textual file called static.info in the program directory.

Class Statement stores the information of different types of statements in the program. We classify the statements in the following different types: CLASS (class definition), MAIN (static main method), INSTANT (object instantiation), METCALL (synchronized or remote method call), SYNBLOCK (synchronized block), WAIT (thread wait method call), METDEF (definition of synchronized or remote method), and OTHER (other statements). The attributes of Statement class are type which holds statement type, line which holds the code of the statement, and className, objectName & methodName which are identified by this statement.

Class CodeInsertion provides two methods. One is the getStatement() method that reads each statement from the programs with the separator "{"","}" or ";" and stores the information that this statement holds in the Statement object. Another method is the parse() method that does the code insertion according to different types of statements. The details of code insertion have already been discussed in section 6.3.
Figure 7.1 shows the class diagram of Code Insertion.

![Class Diagram of Code Insertion]

Figure 7.1. The Class Diagram of Code Insertion

7.3 Development of Test Controllers

In this thesis work, we developed a prototype of test control system that has multiple test controllers working concurrently. Each test controller controls one process in DM programs. Test controllers are implemented as CORBA objects which are running on different processes. The control primitives that are specified in the CORBA IDL file allow communication among the test controllers and the processes of the DM programs. The test constraints in an XML document are transparent and make it easy to generate test constraint objects.
The goal for developing a prototype of test controllers is to help software testers to use the prototype to automatically control the execution of DM programs under given test constraints. Without the prototype, the execution of DM programs is non-deterministic and error prone.

The implementation of this prototype includes the following packages:

- **DataStructures**: Implementation of `LinkedList` for storing information of test constraint objects and other event queues.
- **TestConstraints**: Implementation of Java SAX API for generating test constraint objects from the XML document.
- **TestControllers**: CORBA object implementation for test controllers.

### 7.3.1 The Data Structure Implementation

In DataStructures package, there are two interfaces, `List` and `ListItr`; and three classes, `ListNode`, `LinkedList` and `LinkededListItr`. Figure 7.2 shows the Data Structure class diagram. Class `ListNode` implements a node in `LinkedList`. It uses an attribute `element` to store objects in `LinkedList`. Class `LinkedList` implements interface `List`, which provides the functions of `isEmpty()` and `makeEmpty()`. Class `LinkedListItr` implements interface `ListItr`, which maintains the current position and provides manipulation of `LinkedList` such as insert, remove, retrieve, find, first and advance etc.
7.3.2 Test Constraint Implementation

Test constraints, which are discussed in Chapter 4, consist of a set of events and their relations. The test constraint representation is illustrated in Figure 4.4. Test constraints stored in an XML document are a tree of elements. The format looks like:

```xml
<?xml version = "1.0"?>
<TestConstraint>
  <Monitor>
    <Constraint>
```
<Event>
  <ThreadName>P2:main</ThreadName>
  <MonName>P2:node</MonName>
  <MetName>chooseNo</MetName>
  <Number>1</Number>
</Event>

<Event>
  <ThreadName>P1:main</ThreadName>
  <MonName>P1:node</MonName>
  <MetName>chooseNo</MetName>
  <Number>1</Number>
</Event>

<Event>
  <ThreadName>P1:sendRequest</ThreadName>
  <MonName>P2:request</MonName>
  <MetName>requesting</MetName>
  <Number>1</Number>
</Event>

</Constraint>
</Monitor>
</TestConstraint>

There are three steps to use the SAX API in our test constraint object construction:

1. Creating a test constraint object model (like TestConstraint, Monitor, Constraint, Relation and Event classes) to hold test constraint information.
2. Creating an SAX parser to parse the XML document.
3. Creating a DocumentHandler to turn the XML document into instances of test constraint object model.
Step 1: Creating a test constraint object model

There are five classes to represent the information in our XML test constraint document: TestConstraint class, Monitor class, Constraint class, Relation class and Event class. These classes are a simple mapping from the elements in the XML document. Figure 7.3 shows the class diagram of the Test Constraint object Model.

Figure 7.3. The Class Diagram of Test Constraint Object Model

The following describes the details of these classes:

64
• The TestConstraint class is a container of Monitor objects, which extends LinkedListIter class. The TestConstraint class has methods that allow one to add Monitor object, get Monitor objects, and find out if an event e is the test constraint. The TestConstraint element in the XML document maps to the TestConstraint class.

• The Monitor class, Constraint class and Relation class have a similar construction and functionality, except that the Constraint class has an extra attribute to hold the event.

• The Event class simply holds 4 String objects: the threadName, monName, metName and number. The event element maps to the Event class. The threadName, monName, metName and number elements map into String class.

**Step 2: Creating an SAX parser**

Creating an SAX parser is easy; in addition, the programmer needs to create an XML Document handler class for the parser. Here is the code to create an SAX parser:

```java
import java.io.*;
import org.xml.sax.*;
import com.sun.xml.parser.*;

public class SaxTestConstraintConverter {
    private String parserClassName = "com.sun.xml.parser.Parser";
    private org.xml.sax.Parser parser;
    private SaxTestConstraintHandler handler;

    public SaxTestConstraintConverter (String fname) {
        try{
            InputSource is = Resolver.createInputSource(new File(fname));

            SaxTestConstraintHandler handler = new SaxTestConstraintHandler();
```

parser.setDocumentHandler(handler);
parser.setErrorHandler(handler);
parser.parse(is);
} catch (Throwable t) { t.printStackTrace(); }
}
}

where SaxTestConstraintHandler class is the implementation of DocumentHandler.

**Step 3: Creating a DocumentHandler**

The SAX parser that was created in Step 2 reads an XML document and fires events as it encounters start tags, end tags, data sections, etc. An interface called DocumentHandler is used to notify some objects when these events occur. An implementation of the DocumentHandler interface is provided to deal with method invocation when events are fired.

Figure 7.4 shows the sequences of method calls that the SAX parser makes on the DocumentHandler interface implementation class. The sequence of method calls is numbered from 1 to 26. The figure shows how the SAX parser turns the XML document into a series of events, which are in turn translated into a series of method calls in the DocumentHandler class.

- The SaxTestConstraintHandler class implements the DocumentHandler interface.

The major functions are described as follow:
<table>
<thead>
<tr>
<th>XML Document</th>
<th>List of method calls on the DocumentHandler Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;TestConstraint&gt;</code></td>
<td>1: startDocument()</td>
</tr>
<tr>
<td><code>&lt;Monitor&gt;</code></td>
<td>2: startElement(&quot;TestConstraint&quot;, attrs)</td>
</tr>
<tr>
<td><code>&lt;Constraint&gt;</code></td>
<td>3: startElement(&quot;Monitor&quot;, attrs)</td>
</tr>
<tr>
<td><code>&lt;Event&gt;</code></td>
<td>4: startElement(&quot;Constraint&quot;, attrs)</td>
</tr>
<tr>
<td><code>&lt;ThreadName&gt;</code></td>
<td>5: startElement(&quot;Event&quot;, attrs)</td>
</tr>
<tr>
<td>P2:main</td>
<td>6: startElement(&quot;ThreadName&quot;, attrs)</td>
</tr>
<tr>
<td><code>&lt;MonName&gt;</code></td>
<td>7: characters(char[], start, length)</td>
</tr>
<tr>
<td>P2:node</td>
<td>8: endElement(&quot;ThreadName&quot;)</td>
</tr>
<tr>
<td><code>&lt;MetName&gt;</code></td>
<td>9: startElement(&quot;MonName&quot;, attrs)</td>
</tr>
<tr>
<td>chooseNo</td>
<td>10: characters(char[], start, length)</td>
</tr>
<tr>
<td><code>&lt;Number&gt;</code></td>
<td>11: endElement(&quot;MonName&quot;)</td>
</tr>
<tr>
<td>1</td>
<td>12: startElement(&quot;MetName&quot;, attrs)</td>
</tr>
<tr>
<td><code>&lt;Event&gt;</code></td>
<td>13: characters(char[], start, length)</td>
</tr>
<tr>
<td><code>&lt;Relation&gt;</code></td>
<td>14: endElement(&quot;MetName&quot;)</td>
</tr>
<tr>
<td><code>&lt;Event&gt;</code></td>
<td>15: startElement(&quot;Number&quot;, attrs)</td>
</tr>
<tr>
<td><code>&lt;Event&gt;</code></td>
<td>16: characters(char[], start, length)</td>
</tr>
<tr>
<td><code>&lt;Event&gt;</code></td>
<td>17: endElement(&quot;Number&quot;)</td>
</tr>
<tr>
<td><code>&lt;Event&gt;</code></td>
<td>18: endElement(&quot;Event&quot;)</td>
</tr>
<tr>
<td><code>&lt;Relation&gt;</code></td>
<td>19: startElement(&quot;Relation&quot;, attrs)</td>
</tr>
<tr>
<td><code>&lt;Event&gt;</code></td>
<td>20: startElement(&quot;Event&quot;, attrs)</td>
</tr>
<tr>
<td><code>&lt;Event&gt;</code></td>
<td>21: endtElement(&quot;Event&quot;)</td>
</tr>
<tr>
<td><code>&lt;Relation&gt;</code></td>
<td>22: endElement(&quot;Relation&quot;)</td>
</tr>
<tr>
<td><code>&lt;Constraint&gt;</code></td>
<td>23: endElement(&quot;Constraint&quot;)</td>
</tr>
<tr>
<td><code>&lt;Monitor&gt;</code></td>
<td>24: endElement(&quot;Monitor&quot;)</td>
</tr>
<tr>
<td><code>&lt;TestConstraint&gt;</code></td>
<td>25: endElement(&quot;TestConstraint&quot;)</td>
</tr>
<tr>
<td></td>
<td>26: endDocument()</td>
</tr>
</tbody>
</table>

Figure 7.4. SAX DocumentHandler Interface Methods and Their Sequence

67
• **startDocument()**: This method is called when the SAX parser starts reading the document. It does nothing here.

• **endDocument()**: This method is called when the SAX parser finishes reading the document. It does nothing here.

• **startElement()**: This method is called when the SAX parser encounters a start element tag. It uses the `currentElement` variable to remember that the element tags of `ThreadName`, `MonName`, `MetName` and `Number` are just open so that the `characters(...)` can put the data read from the SAX parser in the right object. If `startElement()` encounters the element tags of `Monitor`, `Constraint`, `Relation` and `Event`, it creates new objects of the corresponding classes: `Monitor`, `Constraint`, `Relation` and `Event`.

• **endElement()**: This method is called when the SAX parser encounters an end element tag. If the element tag is closed, the corresponding object must be added to the test constraint object.

• **characters()**: This method is called when the SAX parser encounters data. It puts the data in the right object.

### 7.3.3 Test Controller Implementation

A test controller is implemented by a CORBA object which itself is a thread. Multiple test controllers are running in different processes, and can be distributed on different machines. The test controller object implements six control primitives (`synRequest`, `synComplete`, `remRequest`, `remComplete`, `waitRequest` and `waitComplete`), which can be remotely invoked by extended programs and other test controllers. The methods of these
six control primitives are specified in one IDL file, called Tcc.idl, which can be compiled into a package Tcc. The thread itself performs the control execution of Extended Programs (DM). Figure 6.3 illustrates the algorithms of our test controllers. For each CORBA object implementation of test controller, it runs in the process of the extended program to be tested.

7.3.3.1 Major Functions of Test Controller Objects

The ControllerImpl class contains the implementation of the test controller and it extends CntrollerPOA. It also provides a method called buildTestConstraints() that gets an input file name of an XML file and creates test constraint objects from the XML file.

The major functions in ControllerImpl are (1) Creating test constraint object from the XML file; (2) Getting the object references of other test controllers; (3) Implementing the six control primitives; and (4) Implementing the process of the test controller.

7.3.3.2 CORBA Objects Have Skeleton and Stub Sides

The IDL compilation generates both a client stub and a skeleton implementation. On the client side, we need to write a client program that accesses a remote object through the client stub. On the object implementation side, we need to write the implementation of the remote object specified in the IDL file. The skeleton passes the client request to the actual object implementation.
7.3.3.3 Name Service

VisiBroker 4.0 provides a name service with a hierarchical namespace for the CORBA objects' service name binding and name resolving. In the name service, there are two functions that are used to bind and resolve the CORBA objects. When each CORBA object provides its service, it needs to bind within a namespace in the name service. And when each of the CORBA objects is invoked by a client, the object name should be resolved in the name service first and then the client can invoke the method.

The Name class encapsulates the implementation of the binding and resolving functions of the name service as shown in Figure 7.5.

```java
package Naming;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.CosNaming.*;

public class Name {
    private NamingContext root;
    private POA myPOA;

    public Name(NamingContextExt rootExt, POA myPOA) throws Exception {
        this.myPOA = myPOA;
        this.root = (NamingContext) rootExt;
    }

    public void binding(String objName, Servant r) throws Exception {
        NameComponent[] rName = { new NameComponent (objName, ")" ) }; //
        root.rebind(rName, myPOA.servant_to_reference(r));
    }

    public org.omg.CORBA.Object resolve(String objName) throws Exception {
        NameComponent[] rName = { new NameComponent (objName, ")" ) }: //
        return root.resolve(rName);
    }
}
```

**Figure 7.5. The Name Class**

70
7.4 How to Run the System

There are two steps in running the system. First, the code insertion module must be running to generate the extended programs. Then the extended programs and test controllers must be running to control the non-deterministic execution.

7.4.1 How to Run Code Insertion Module

To run the Code Insertion Module, go to the CodeInsertion directory, and then type the following command (all commands are bold)

% Java CodeInsertion

CodeInsertion will ask testers for a package (directory) name, such as Bakery used to hold the source code of tested DM programs. The static information of the program is given in file static.info. After getting the information from this file, the system will read the source files in the given package, and generate the extended programs in a new package (directory) such as BakeryExt. After all the files of extended programs are generated, the DM system is ready to be tested.

7.4.2 How to Run Test Controllers
In the directory of the DM program to be tested, run the *NameServe* first, and then run the multiple processes of extended programs and test controllers. For example, the commands for the execution of Distributed Bakery Algorithm (DBA) are as follows:

```
%nameserv <NameService>
%vbj -DSVCnameroot=<NameService> MainProcess 0 <testconstraintfile>
%vbj -DSVCnameroot=<NameService> MainProcess 1 <testconstraintfile>
```
Chapter 8

EVALUATION

In this chapter, we present an analysis and experimental results to evaluate our reproducible testing approach. In section 1, we analyze the performance of our reproducible testing by comparing to Sohn's approach. In section 2, we describe the examples we used to examine the performance of our approach. In section 3, we show the results and summarize our evaluation.

8.1 Analysis

The performance of reproducible testing can be measured by the number of synchronization and interaction events that are controlled by remote (or global) test controllers. The controlled execution is done by adding control primitives in the program. For a DM program, given the number of total synchronized method calls (#Sync. Met. Call), the number of synchronized remote method calls (#Sync. Rem. Call), and the number of events that have happen-before relationship with other events in different processes (#Hap. Relation), we can calculate the number of added control primitives (#Control among test controllers) among test controllers and the number of communications among threads (#Communication among threads):

\[
#\text{Control among test controllers} = #\text{Sync. Rem. Call} + #\text{Hap. Relation}
\]

\[
#\text{Communication among threads} = #\text{Sync. Met. Call}
\]
Our approach provided a mechanism and algorithm with multiple test controllers for multiple test processes. In this approach, the synchronization and interaction events within a process are sent to its local test controller. Only the synchronization and interaction events involved in two processes are sent to the remote test controller of the process with synchronized method being called. The number of added control primitives for our approach is $\#\text{Sync. Rem. Call} + \#\text{Hap. Relation}$. Sohn's approach [6,7] provided a mechanism and algorithm with one global test controller. In this approach, all the synchronization and interaction events are sent to this global test controller. The number of added control primitives for Sohn's approach is $\#\text{Sync. Met. Call}$. Therefore, there are more events need to be controlled by the global test controller in Sohn's approach than by the remote test controller in our approach.

### 8.2 The Examples

To evaluate the performance of our approach, we chose three Java DM programs (the Automated Banking, the Dining Philosophers, and the Distributed Bakery Algorithm) that involve non-deterministic behavior. Each program has two processes, and at least one process has two or more threads.

#### 8.2.1 Automated Banking

The Automated Banking example is introduced by D. Lea [37]. In this example, two users have access to two accounts – the checking account and the savings account. The available operations are deposit to, and withdraw from an account. All users are
implemented by threads and the accounts are represented by objects accessed by their users. There are two processes (CheckingAccount process and SavingsAccount process), each of which has an account and a thread. An additional thread is added to CheckingAccount process that automatically transfers money from the checking account to the savings account if the amount in the checking account exceeds a certain threshold. Figure 8.1 shows the structure of this example.

![Diagram of Automated Banking Example]

**Figure 8.1. The Structure of Automated Banking Example**

The Java implementation of the Automated Banking example is given in Appendix C. There are three threads and four synchronized method calls. Only one synchronized method call is remote method call.

### 8.2.2 Dining Philosophers

The Dining Philosophers problem [11] is a well-known example, often used as a case study for concurrent programming. Our dining philosophers example involves three philosophers sitting in a circle. Each of the philosophers alternates between eating and thinking. To eat, a philosopher must acquire two forks, each of which it shares with one of its neighbors. To avoid deadlock, all philosophers must pick the fork from their left first,
then the fork from their right. All philosophers are implemented by threads and the forks are represented by monitor objects shared by the philosopher threads. To implement it as a Java DM program, we provide two processes, one has two philosopher threads and two fork objects, and another one has one philosopher thread and one fork object. Figure 8.2 shows the structure of the Dining Philosophers example.

![Diagram](image)

**Figure 8.2. The Structure of Dining Philosophers Example**

The Java implementation of the Dining Philosophers example is given in Appendix D. There are three threads and six synchronized method calls. Only two synchronized method calls are remote method calls.

### 8.2.3 Distributed Bakery Algorithm

We have already discussed the Distributed Bakery Algorithm as an example of our approach in Chapter 2. The structure of this example was shown in Figure 2.4. There are four threads and fourteen synchronized method calls. Only six synchronized method calls are remote method calls.
8.3 The Results and Evaluation

The main purpose of our approach is to improve the efficiency of reproducible testing. The execution time is used as the criterion. Suppose the number of added control primitives among test controllers is N, the execution time of a DM program without test controllers is T1, and the execution time of each added control primitive among test controllers is T2, then the total execution time with test controllers are T1 + N*T2.

Figure 8.3 presents the data of running these examples with our approach and without our approach. In this figure, for each example of Java program, the first column gives the name of the program. The second column gives the overall number of synchronized method calls in the program. The third column gives the number of synchronized remote method calls. The fourth and the fifth column give the time that is used to run the programs with our approach and the time that is used to run the programs without our approach. The results of this experiment show that our approach represents an efficient way to control the execution of non-deterministic behavior.

<table>
<thead>
<tr>
<th>Program</th>
<th>#Sync. Met. Call</th>
<th>#Sync. Rem. Call</th>
<th>Running Time with our approach (milliseconds)</th>
<th>Running Time without our approach (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automated Banking</td>
<td>4</td>
<td>1</td>
<td>671</td>
<td>502</td>
</tr>
<tr>
<td>Dining Philosophers</td>
<td>6</td>
<td>2</td>
<td>713</td>
<td>567</td>
</tr>
<tr>
<td>Distributed Bakery Algorithm</td>
<td>14</td>
<td>6</td>
<td>1075</td>
<td>891</td>
</tr>
</tbody>
</table>

Figure 8.3. The Experimental Data
Chapter 9

RELATED WORK

This chapter gives a brief review of earlier work on reproducible testing of distributed and concurrent programs, including related work on static analysis models, test case generation and some existing test control methods for reproducible testing. Then we describe the advantages of our reproducible testing approach.

9.1 Static Analysis Models

In this section, we review two types of static analysis models for describing the behavior and static information of distributed and concurrent programs.

9.1.1 FLAVERS (Flow Analysis for Verification of Systems)

Naumovich, Avrunin and Clarke [1,2] provide the FLAVERS (Flow Analysis for Verification of Systems) static analysis method for modeling Java concurrent programs. This approach extracts the behavior of monitors into several low-level thread synchronization primitives and constructs the TFG (Traceable Flow Graph) for Java concurrent programs. But the thread communications are modeled by feasibility constraints instead of explicitly represented in the TFG model. A deterministic Finite State Automaton (FSA) over the TFG alphabet is used to describe feasibility constraints and
user specified properties. The TFG model is useful for evaluating all possible executable paths for specific kinds of faults.

9.1.2 EBBA (Event-Based Behavioral Abstraction)

Peter [27] provides a program behavior model, called EBBA (Event-Based Behavioral Abstraction). In this model, the behavior is expressed by a collection of events and the relationship of different event types. A high-level debugging approach, based on the EBBA model, is proposed to compare models of expected program behaviors and the actual program behaviors. This model is also useful for describing the behavior and the static information of the program.

9.2 Test Case Generation

9.2.1 EIAG (Event InterActions Graph)

Katayama, Furukawa and Ushijima [4,5] proposed the EIAG (Event InterActions Graph) as a static analysis model for Ada concurrent programs. The EIAG consists of Event Graphs that denote concurrency events in each process, and Interactions that denote interactions between the event Graphs. Copaths are defined as test cases for concurrent programs, which consist of sets of paths on Event Graphs satisfying Interactions. Test criteria and algorithms for automatically generating test cases are provided based on the EIAG. The disadvantage of this approach is that there is no way in EIAG to describe the interactions based on the shared memory mechanism.
9.2.2 PPEG (Parallel Program Flow Graph)

Yang and Pollock [8] provided automatic generation of test cases for multithreaded and parallel programs for shared memory multiprocessors. A PPEG (Parallel Program Flow Graph) is defined to represent the control flow model of concurrent programs. The proposed algorithms can generate feasible test case. Temporal testing method rather than reproducible testing method is provided to deal with non-determinism. In temporal testing, programs are executed repeatedly with the same inputs by altering the scheduled execution time of all process creation and synchronization events along the test case.

9.3 Replay Control Methods

In this section, we review two replay control methods to control the non-deterministic executions in reproducible testing of distributed and concurrent programs.

9.3.1 Language-Based Replay Control Method

Carver and Tai [17] proposed a language-based deterministic execution method using concurrent constructs such as queued semaphores and monitors for replay of concurrent programs. Using the synchronization and mutual exclusion constructs of the semaphores and monitors, they can control the execution sequence of synchronization events by transforming the concurrent program into another program in the same language. It performs in two steps:
(1) Collecting the sequences of synchronization events of a concurrent program by transforming it into a new program and executing the new one.
(2) Controlling the execution of a concurrent program by transforming it into different programs that can replay the collected synchronization sequences.

9.3.2 Replay Control Mechanism with Communication Primitives

Sohn, Kung and Hsia [6,7] proposed another deterministic execution method using several communication primitives and an efficient replay algorithm for reproducible testing on concurrent programs like Ada [6] and distributed object-oriented programs like CORBA [7]. In this approach, an extended program (EP) is generated by inserting the communication primitives before and after the concurrent statements in the original program. One additional replay control mechanism and algorithm uses these communication primitives to control the execution of the extended program (EP) that is logically equivalent to the original one.

This approach works more generally in distributed and object-oriented environments. It can reduce the system overhead using only one global replay controller. The disadvantage of this approach is that the centralized test controller may increase the number of communications among the test controller and the processes of the program since the events within a process are still sent to the global test controller.

9.3 Advantages of Our Approach
In our approach, the extended design notation - PMSC takes advantage of MSC which offers clear graphical representation, object-oriented and standardization of systematic tool support. With the extension of thread interactions and synchronization in PMSC, it can explicitly describe the behavior and represent the static information of DM programs such as flow controls, thread interactions, object behaviors, and monitor synchronization etc.

Since Petri net can describe the non-deterministic behavior of concurrent systems, our test case specification in Petri net is sufficient for describing a certain degree of deterministic execution of DM programs. By constructing test constraints from test case specification, our approach only focuses on the test constraints with a set of concurrent events and their relations. Those events correspond only to concurrent constructs such as synchronized methods, remote methods, and thread interactions. The test constraints overcome the limitation of forcing the unnecessary control of non-concurrent events.

We provide a new test control mechanism in which the test controller (TC) consists of multiple test controllers for multiple processes. The process of the test controller controls the execution of corresponding processes by using different types of communication primitives. The advantage of this mechanism is that it reduces the number of communications among the test controllers and the processes of the program.
Chapter 10

CONCLUSIONS AND FUTURE WORK

This chapter draws the conclusions and describes two possible directions for future work related to this research.

10.1 Conclusions

In this thesis, Distributed Multithreaded (DM) architecture and non-deterministic behavior are first presented. Then we proposed an extended design notation - PMSC based on MSC for this architecture, which can explicitly represent concurrent and object behavior of DM programs. The PMSC is used as a basis for test case generation and test control in reproducible testing of DM programs. We also introduced a method for describing test case specification using Petri net. The Petri net specification can be used to describe test scenarios with a certain degree of determinism. In addition, by constructing test constraints from the test case specification in Petri net, we can use the test constraints with a set of concurrent events and their relations a test scenario for our testing approach.

This thesis presents significant contributions to reproducible testing of DM programs and provides a test control method with code insertion and multiple test controllers for monitoring and controlling a certain degree of deterministic execution of DM programs.

The main contributions of this thesis are:
1. Presenting system architecture and non-deterministic behavior of Java Distributed Multithreaded (DM) programs.

2. Proposing an extended design notation - PMSC which is based on MSC for Java DM programs.

3. Introducing a test case description based on Petri net and test constraints to describe test scenarios with a certain degree of determinism.

4. Providing a test control mechanism and algorithm for reproducible testing of Java DM programs based on the PMSC and our test constraints.

5. Providing the design and implementation of the prototype of reproducible testing system.

6. Showing the results and evaluating our approach.

10.2 Future Work

A method was developed for the reproducible testing of DM programs based on MSC and Petri net. This method includes the extended design notation – PMSC (Parallel Message Sequence Chart) based on MSC, the test case (test constraints) based on Petri net specification, and the test control mechanism for reproducible testing based on PMSC and the test constraints. According to the process of reproducible testing, some extensions to this research remain for further exploration.

There are two possible types of extensions. One is the automatic generation of the PMSC model from Java code. Since MSC has the standard formal semantics and systematic tool support, a tool that supports additional semantics of PMSC should be built to translate the behavior and static information in the program to the PMSC model.
Another possible extension is the test case (test constraints) generation. By analyzing the test case specification in Petri net and the PMSC model, an algorithm should be provided to construct feasible test constraints with a set of concurrent events and their relation. Such test constraints can control a certain degree of non-deterministic execution.
BIBLIOGRAPHY


[12]. X.J. Chen, L. Logrippo; "Deriving Use Cases for Distributed Systems from Knowledge Requirements", Univ. of Windsor.


[25]. The Telecommunication Standard of International Telecommunication Union (ITU-TS), Home Page; http://www.itu.int/


[35]. A. Bechini, and K.-C. Tai; "Design of a Toolset for Dynamic Analysis of Concurrent Java Programs"; 6th International Workshop on Program Comprehension, 24-26 June 1998, Ischia, Italy.


Appendix A

SOURCE CODE OF DISTRIBUTED BAKERY ALGORITHM

/* Class MainProcess - This is the main process which provides the precondition and postcondition for distributed bakery algorithm (DBA). */
package Bakery;
import java.util.*;
import java.io.*;
import org.omg.CORBA. *
import org.omg.PortableServer.*;
import org.omg.CosNaming.*;
import Naming.*;

public class MainProcess extends Thread{
    public final static String REQUEST="Request", REPLY="Reply";
    private Node node = null;
    private Name name = null;
    private SendRequest[] sendRequests = null;

    public MainProcess(Name name, Node node) {
        this.name = name;
        this.node = node;
        sendRequests = new SendRequest[ node.getTotal() ];
    }

    public static void main(String args[]){
        if (args.length != 2 ) {
            System.out.println("Usage: MainProcess id total");
            System.exit(1);
        }
        int id = Integer.parseInt(args[0]);
        int total = Integer.parseInt(args[1]);

        try { // 1. Initialize the ORB
            ORB orb = ORB.init(args, null);
            // get a reference to the root POA
            POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
            // get a reference to the Naming Service root_context
            org.omg.CORBA.Object rootObj = orb.resolve_initial_references("NameService");
            NamingContextExt root = NamingContextExtHelper.narrow(rootObj);
            // Create policies for our persistent POA
            org.omg.CORBA.Policy[] policies = {
                rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT) };
            // Create myPOA with the right policies
            POA myPOA = rootPOA.create_POA( "bakery_poa", rootPOA.the_POAManager(), policies );

            // 2. Instantiate Service Objects

        }
    }
}
// Create Name Object
Name name = new Name(root, myPOA);
Node node = new Node(id, total);
// Create MainProcess object
MainProcess process = new MainProcess(name, node);
process.start();
// Create the Request and Reply objects
RequestImpl request = new RequestImpl(node, name);
ReplyImpl reply = new ReplyImpl(node, process);

// 3. Activate the objects with the ID on myPOA
myPOA.activate_object_with_id((REQUEST+id).getBytes(), request);
myPOA.activate_object_with_id((REPLY+id).getBytes(), reply);
// Activate the POA manager
rootPOA.the_POAManager().activate();
// Associate the objects with the name in the name service
name.binding(REQUEST+id, request);
name.binding(REPLY+id, reply);
System.out.println("Servant is ready.");
// Wait for incoming requests
orb.run();
}
catch (Exception e) { e.printStackTrace(); }
}

class MainProcess {
    public void run () {
        try {
            while (true) {
                Thread.sleep(10000);
                precondition();
                doCS();
                postcondition();
            }
        }
    }
    catch (Exception e) { e.printStackTrace(); }
}

private void precondition() throws Exception {
    node.chooseNumber(); // Choose a number
    node.setReplyCount(0); // send Requests to other nodes

    int num = node.getNumber();
    int id = node.getID();
    for (int i = 0; i < node.getTotal(); i++) {
        if (i != id) {
            DBA.Request req = DBA.RequestHelper.narrow(name.resolve(REQUEST+i));
            sendRequests[i] = new SendRequest(req, num, id);
            sendRequests[i].start();
        }
    }
}
// Waiting for entering critical section
synchronized (this) {
    if (node.getReplyCount() != node.getTotal() - 1) {
        wait();
    }
}
}

private void postcondition() throws Exception {
    node.setRequesting(false);
    for (int i = 0; i < node.getTotal(); i++) {
        if (node.getDefer(i)) {
            node.setDefer(i, false);
            DBA.Reply rep = DBA.ReplyHelper.narrow(name.resolve(REPLY + i));
            rep.replaying();
        }
    }
}

private void doCS0 {
    System.out.println("Do Critical Section in Node " + node.getID());
}
} // end of MainProcess

/*@ Class SendRequest -- The thread which performs sending requests to other nodes */
class SendRequest extends Thread {
    DBA.Request request = null;
    int number = 0;
    int id = 0;

    public SendRequest(DBA.Request request, int number, int id) {
        this.number = number;
        this.id = id;
        this.request = request;
    }

    public void run() {
        try {
            System.out.println("send request");
            request.requesting(number, id);
        } catch (Exception e) {
            e.printStackTrace(System.err);
        }
    }
} // end of SendRequest

/*@ Class Node - This is global variable class used by the DBA algorithm */
class Node {
    private int id, total, number = 0, highNumber = 0, replyCount = 0;
    private boolean requesting = false;
    private boolean[] defer = null;

}
public Node(int id, int total) {
    this.id = id;
    this.total = total;
    defer = new boolean[total];
    for (int i=0; i<total; i++)
        defer[i] = false;
}

public int getId() { return id; }
public int getTotal() { return total; }

public synchronized void setRequesting(boolean requesting) {
    this.requesting = requesting;
}

public boolean getRequesting() { return requesting; }
public void setNumber(int number) { this.number = number; }
public int getNumber() { return number; }
public void setHighNum(int number) { this.highNumber = number; }
public int getHighNum() { return highNumber; }
public void setReplyCount(int count) { this.replyCount = count; }
public int getReplyCount() { return replyCount; }
public void setDefe(int i, boolean def) { defer[i] = def; }
public boolean getDefe(int i) { return defer[i]; }
public synchronized void chooseNumber () {
    requesting = true;
    number = highNumber + 1;
}
} // end of class Node

/* Class RequestImpl - This is the implementation of the DBA Request process. */
class RequestImpl extends DBA.RequestPOA {
    private Node node = null;
    private Name name = null;
    RequestImpl (Node node, Name name) {
        this.node = node;
        this.name = name;
    }

    public synchronized void requesting (int number, int id) {
        int highNum = Math.max(node.getHighNum(), number);
        node.setHighNum(highNum);

        synchronized (node) {
            int num = node.getNumber();
            boolean isDefe = (node.getRequesting() && (num < number ||
                (num == number && node.getId()<id)));
            if (isDefe)
                node.setDefe(id, true);
            else

94
try {
    DBA.Reply reply = DBA.ReplyHelper.narrow(name.resolve(MainProcess.REPLY+id));
    reply.replying();
} catch (Exception e) {
    e.printStackTrace(System.err);
}

} // end of RequestImpl

/** Class ReplyImpl - This is the implementation of the reply process */
public class ReplyImpl extends DBA.ReplyPOA {
    private Node node;
    private MainProcess process;
    public ReplyImpl(Node node, MainProcess process) {
        this.node = node;
        this.process = process;
    }

    public synchronized void replying() {
        node.setReplyCount(node.getReplyCount()+1);

        synchronized (process) {
            if (node.getReplyCount() == (node.getTotal()-1))
                process.notify();
        }
    }
} // end of ReplyImpl

/** DBA.idl - The Specification of Distributed Bakery Algorithm */
module DBA {
    interface Request {
        void requesting (in long number, in long id);
    }
    interface Reply {
        void replying ();
    }
};
Appendix B

EXTENDED CODE OF DISTRIBUTED BAKERY ALGORITHM

/* Class MainProcess - This is the main process which provides the preconition and postcondition
   for distributed bakery algorithm (DBA). */
package Bakery;
import java.util.*;
import java.io.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.CosNaming.*;
import Naming.*;
import Controller.*;

public class MainProcess extends Thread {
    public static String ProcessId;
    public String monitorName;
    public ControllerImpl controller;
    public void setController(ControllerImpl controller){
        this.controller = controller;
    }
    public void setMonitorName (String str){
        monitorName = MainProcess.ProcessId + ";" + str;
    }
    public String getMonitorName (){    
        return monitorName;
    }

    public final static String REQUEST="Request", REPLY="Reply";
    private Node node = null;
    private Name name = null;
    private SendRequest[] sendRequests = null;

    public MainProcess(Name name, Node node) {
        this.name = name;
        this.node = node;
        sendRequests = new SendRequest[ node.getTotal()];
    }

    public static void main(String args[]){
        ProcessId = "P"+ Integer.parseInt(args[0]);  // set static attribute ProcessId
        if (args.length != 3 ){
            System.out.println("Usage: MainProcess id total, fileName");
            System.exit(1);
        }
        int id = Integer.parseInt(args[0]);
        }
int total = Integer.parseInt(args[1]);
String fileName = args[2];

try {
    // 1. Initialize the ORB
    ORB orb = ORB.init(args, null);
    // get a reference to the root POA
    POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
    // get a reference to the Naming Service root_context
    org.omg.CORBA.Object rootObj = orb.resolve_initial_references("NameService");
    NamingContextExt root = NamingContextExtHelper.narrow(rootObj);
    // Create policies for our persistent POA
    org.omg.CORBA.Policy[] policies = {
        rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
    };
    // Create myPOA with the right policies
    POA myPOA = rootPOA.create_POA("bakery_poa", rootPOA.the_POAManager(), policies);

    // 2. Instantiate Service Objects
    // Create Name Object
    Name name = new Name(root, myPOA);
    // Create Controller object
    ControllerImpl controller = new ControllerImpl(ProcessId, fileName, name);

    Node node = new Node(id, total);
    node.setMonitorName("node");
    node.setController(controller);

    // Create MainProcess object
    MainProcess process = new MainProcess(name, node);
    process.setName(MainProcess.ProcessId+":process");
    process.setMonitorName("process");
    process.setController(controller);
    process.start();

    // Create the Request and Reply objects
    RequestImpl request = new RequestImpl(node, name);
    request.setMonitorName("request");
    request.setController(controller);

    ReplyImpl reply = new ReplyImpl(node, process);
    reply.setMonitorName("reply");
    reply.setController(controller);

    // 3. Activate the objects with the ID on myPOA
    myPOA.activate_object_with_id((REQUEST+id).getBytes(), request);
    myPOA.activate_object_with_id((REPLY+id).getBytes(), reply);
    myPOA.activate_object_with_id((TCC.Controller.CONTROL+ProcessId).getBytes(), Controller);
    // Activate the POA manager
    rootPOA.the_POAManager().activate();
    // Associate the objects with the name in the name service
    name.binding(REQUEST+id, request);
    name.binding(REPLY+id, reply);
name.binding(TCC.Controller.CONTROL+ProcessId, controller);
System.out.println("Servant is ready.");
// Wait for incoming requests
orb.run();
}
catch (Exception e) { e.printStackTrace(); }
}

public void run () {
try {
   while (true) {
      Thread.sleep(10000);
      precondition();
      doCS0();
      postcondition();
   }
} catch (Exception e) { e.printStackTrace(); }
}

private void precondition() throws Exception {
   // Choose a number
controller.synRequest(Thread.currentThread().getName(),
   node.getNumber(), "chooseNumber");
node.chooseNumber();

node.setReplyCount(0); // send Requests to other nodes
int num = node.getNumber();
int id = node.getId();
for (int i = 0; i < node.getTotal(); i++) {
   if (i != id) {
      DBA.Request req = DBA.RequestHelper.narrow(name.resolve(REQUEST+i));
      sendRequests[i] = new SendRequest(req.num, id);
      sendRequests[i].setName(ProcessId+"sendRequest"+i);
      sendRequests[i].setController(controller);
      sendRequests[i].start();
   }
}

// Waiting for entering critical section
controller.synRequest(Thread.currentThread().getName(), getMonitorName(), "synblock");
synchronized (this) {
   controller.synComplete(Thread.currentThread().getName(), getMonitorName(),
      "synblock");
   if (node.getReplyCount() != node.getTotal()-1) {
      controller.waitForRequest(Thread.currentThread().getName(), getMonitorName(), "wait");
      wait();
      controller.waitForComplete(Thread.currentThread().getName(), getMonitorName(), "wait");
   }
}
}
private void postcondition() throws Exception {
    controller.synRequest(Thread.currentThread().getName(), node.getMonitorName(),
        "setRequesting");
    node.setRequesting(false);
    for (int i = 0; i < node.getTotal(); i++) {
        if (node.getDefer()) {
            node.setDefer(i, false);
            DBA.Reply rep = DBA.ReplyHelper.narrow(name.resolve(REPLY+i));
            controller.remRequest(Thread.currentThread().getName(), rep.getMonitorName(),
                "replying");
            /* rep.replying(); */
            rep.replying(Thread.currentThread().getName());
        }
    }
}
private void doCS() {
    System.out.println("Do Critical Section in Node " + node.getID());
}
} // end of MainProcess

/* Class SendRequest – The thread which performs sending requests to other nodes */
class SendRequest extends Thread {
    ControllerImpl controller;

    public void setController(ControllerImpl controller) {
        this.controller = controller;
    }

    DBA.Request request = null;
    int number = 0;
    int id = 0;

    public SendRequest(DBA.Request request, int number, int id) {
        this.number = number;
        this.id = id;
        this.request = request;
    }

    public void run() {
        try {
            System.out.println("send request");

            controller.remRequest(Thread.currentThread().getName(),
                request.getMonitorName(), "requesting");
            /* request.requesting( number, id); */
            request.requesting(number, id, Thread.currentThread().getName());
        } catch (Exception e) { e.printStackTrace(System.err); }
    }
} // end of SendRequest
/* Class Node - This is global variable class used by the DBA algorithm */
class Node {
    private String monitorName;
    private ControllerImpl controller;
    public void setController(ControllerImpl controller) {
        this.controller = controller;
    }
    public void setMonitorName(String str) {
        monitorName = MainProcess.ProcessId + ":" + str;
    }
    public String getMonitorName() {
        return monitorName;
    }

    private int id, total, number = 0, highNumber = 0, replyCount = 0;
    private boolean requesting = false;
    private boolean[] defer = null;

    public Node(int id, int total) {
        this.id = id;
        this.total = total;
        defer = new boolean[total];
        for (int i = 0; i < total; i++)
            defer[i] = false;
    }

    public int getId() { return id; }
    public int getTotal() { return total; }

    public synchronized void setRequesting(boolean requesting) {
        controller.synComplete(Thread.currentThread().getName(), getMonitorName(), "setRequesting");
        this.requesting = requesting;
    }

    public boolean getRequesting() { return requesting; }
    public void setNumber(int number) { this.number = number; }
    public int getNumber() { return number; }
    public void setHighNum(int number) { this.highNumber = number; }
    public int getHighNum() { return highNumber; }
    public void setReplyCount(int count) { this.replyCount = count; }
    public int getReplyCount() { return replyCount; }
    public void setDefe(int i, boolean def) { defer[i] = def; }
    public boolean getDefe(int i) { return defer[i]; }
    public synchronized void chooseNumber() {
        controller.synComplete(Thread.currentThread().getName(), getMonitorName(), "chooseNumber");
        requesting = true;
        number = highNumber + 1;
    }
}
} // end of class Node
/* Class RequestImpl - This is the implementation of the DBA Request process. */
class RequestImpl extends DBA.RequestPOA {
    private String monitorName;
    private ControllerImpl controller;
    public void setController(ControllerImpl controller) {
        this.controller = controller;
    }
    public void setMonitorName(String str) {
        monitorName = MainProcess.ProcessId + ":" + str;
    }
    public String getMonitorName() {
        return monitorName;
    }
    private Node node = null;
    private Name name = null;
    RequestImpl (Node node, Name name) {
        this.node = node;
        this.name = name;
    }
    /* public synchronized void requesting (int number, int id) */
    public synchronized void requesting (int number, int id, String threadName) {
        Thread.currentThread().setName(threadName);
        controller.removeComplete(Thread.currentThread().getName(), getMonitorName(), "requesting");
        int highNum = Math.max(node.getHighNum(), number);
        node.setHighNum(highNum);
        controller.synRequest(Thread.currentThread().getName(), node.getMonitorName(), "synblock");
        synchronized (node) {
            controller.synComplete(Thread.currentThread().getName(), node.getMonitorName(), "synblock");
            int num = node.getNumber();
            boolean isDefer = (node.getDefer() && (num < number || (num == number && node.getID() < id)));
            if (isDefer)
                node.setDefer(id, true);
            else
                try{
                    DBA.Reply reply = DBA.ReplyHelper.narrow(name.resolve(MainProcess.REPLY + id));
                    controller.removeRequest(Thread.currentThread().getName(), reply.getMonitorName(),
                    "replying");
                    /* reply.replying() */
                    reply.replying(Thread.currentThread().getName());
                } catch (Exception e) {
                    e.printStackTrace(System.err);
                }
        }
    }
} // end of RequestImpl
/* Class ReplyImpl - This is the implementation reply process */
public class ReplyImpl extends DBA.ReplyPOA {
    private String monitorName;
    private ControllerImpl controller;
    public void setController(ControllerImpl controller){
        this.controller = controller;
    }
    public void setMonitorName (String str){
        monitorName = MainProcess.ProcessId + ":" + str;
    }
    public String getMonitorName (){
        return monitorName;
    }
    
    private Node node;
    private MainProcess process;
    public ReplyImpl(Node node, MainProcess process){
        this.node = node;
        this.process = process;
    }

    /* public synchronized void replying() */
    public synchronized void replying(String threadName){
        Thread.currentThread().setName(threadName);
        controller.remComplete(Thread.currentThread().getName(),getMonitorName(), "replying");
        node.setReplyCount( node.getReplyCount()+1);
        controller.synRequest(Thread.currentThread().getName(), process.getMonitorName(), "synblock");
        synchronized (process){
            controller.synComplete(Thread.currentThread().getName(), process.getMonitorName(), "synblock");
            if( node.getReplyCount() == (node.getTotal()-1))
                process.notify();
        }
    }
} // end of ReplyImpl

/* DBA.idl - The Specification of Distributed Bakery Algorithm */
module DBA {
    interface Request {
        /* void requesting (in long number, in long id); */
        void requesting(in long number, in long id, in string threadName);
        string getMonitorName();
    };
    interface Reply {
        /* void replying 0; */
        void replying(in string threadName);
        string getMonitorName();
    };
};
Appendix C

SOURCE CODE OF AUTOMATED BANKING

/* BankAccount class - This class is the super class of CheckingAccount and SavingAccount */
package AutomatedBanking;

interface BankAccount {
    public long balance();
    public void deposit(long amount);
    public void withdraw(long amount);
}

/* CheckingAccount class - this class implement a checking account which operations are a deposit to,
withdraw from the account, and transfer to the savings account if the amount in the checking account
exceeds a threshold. */
package AutomatedBanking;

import java.util.*;
import java.io.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.CosNaming.*;

public class CheckingAccountImpl extends ACCOUNT.CheckingAccountPOA implements BankAccount,
Runnable {
    public static final String CHECKINGACCOUNT = "CheckingAccount",
    SAVINGSACCOUNT = "SavingsAccount";
    protected long balance = 0;
    protected ACCOUNT.SavingsAccount savings;
    protected long threshold;

    public CheckingAccountImpl(long t) { threshold = t; }

    // called only upon initialization
synchronized void initSavings(ACCOUNT.SavingsAccount s) {
        savings = s;
    }

    protected boolean shouldTry() { return balance < threshold; }
    public synchronized void tryTransfer() { // called an additional thread
        if (shouldTry()) balance += savings.transferOut();
    }
    public synchronized long balance() { return balance; }
    public synchronized void deposit(long amount) {
        balance += amount;
    }
}
public void withdraw(long amount) {
    deposit(-amount);
}

public static void main(String args[]) {
    try {  // 1. Initialize the ORB
        ORB orb = ORB.init(args, null);

        // get a reference to the root POA
        POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));

        // get a reference to the Naming Service root_context
        org.omg.CORBA.Object rootObj = orb.resolve_initial_references("NameService");
        NamingContextExt root = NamingContextExtHelper.narrow(rootObj);

        // Create policies for our persistent POA
        org.omg.CORBA.Policy[] policies = {
            rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
        };

        // Create myPOA with the right policies
        POA myPOA = rootPOA.create_POA("banking_poa", rootPOA.the_POAManager(), policies);

        // 2. Instantiate Service Objects
        CheckingAccountImpl checking = new CheckingAccountImpl(5000);
        NameComponent[] rName = { new NameComponent(SAVINGSACCOUNT, ","),
            ACCOUNT.SavingsAccountHelper.narrow(((NamingContext) root).resolve(rName))
        };
        checking.initSavings(rName);
        Thread checkThread = new Thread(checking);
        checkThread.start();
        TransferThread transfer = new TransferThread(checking);
        transfer.start();

        // 3. Activate the objects with the ID on myPOA
        myPOA.activate_object_with_id(CHECKINGACCOUNT.getBytes(), checking);

        // Activate the POA manager
        rootPOA.the_POAManager().activate();

        // Associate the objects with a name in the name service
        NameComponent[] rrName = { new NameComponent(CHECKINGACCOUNT, ","),
            ((NamingContext) root).rebind(rrName, myPOA.servant_to_reference(checking))
        };

        System.out.println("The servant is ready.");
        // Wait for incoming requests
        orb.run();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
public void run() {
    try {
        while(true) {
            deposit(1000);
            Thread.sleep(10000);
            withdraw(500);
            Thread.sleep(10000);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
} // end of class CheckingAccount

class TransferThread extends Thread {
    CheckingAccountImpl checking;

    public TransferThread (CheckingAccountImpl check) {
        checking = check;
    }

    public void run() {
        while(true) {
            checking.tryTransfer();
        }
    }
} // end of class TransferThread

/*============================================================================*/
/* SavingsAccount class - this class implement a savings account which operations are a deposit to, */
/* withdraw from the account, and transfer in the savings account if the amount in the checking account */
/* exceeds a threshold. */
/*============================================================================*/

package AutomatedBanking;

import java.util.*;
import java.io.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.CosNaming.*;

public class SavingsAccountImpl extends ACCOUNT.SavingsAccountPOA implements BankAccount, Runnable {
    public static final String CHECKINGACCOUNT = "CheckingAccount",
    SAVINGSACCOUNT = "SavingsAccount";
    protected long maxTransfer;
    protected long balance = 0;

    public SavingsAccountImpl (long max) { maxTransfer = max; }

    public synchronized long transferOut() { // called only from the checking account
        long amount = balance;
        if (amount > maxTransfer)
            amount = maxTransfer;
if (amount >= 0)
    balance -= amount;
return amount;
}
public synchronized long balance() { return balance; }
public synchronized void deposit(long amount)
    balance += amount;
}
public void withdraw(long amount) {
    deposit(-amount);
}
public static void main(String args[])
    // 1. Initialize the ORB
    try {
        ORB orb = ORB.init(args, null);
        // get a reference to the root POA
        POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA");
        // get a reference to the Naming Service root_context
        org.omg.CORBA.Object rootObj = orb.resolve_initial_references("NameService");
       NamingContextExt root = NamingContextExtHelper.narrow(rootObj);
        // Create policies for our persistent POA
        org.omg.CORBA.Policy[] policies = {
            rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
        };
        // Create myPOA with the right policies
        POA myPOA = rootPOA.create_POA( "banking-po", rootPOA.the_POAManager(), policies );
        // 2. Instantiate Service Objects
        SavingsAccountImpl savings = new SavingsAccountImpl(500);
        Thread saveThread = new Thread(savings);
        saveThread.start();
        // 3. Activate the objects with the ID on myPOA
        myPOA.activate_object_with_id(SAVINGSACCOUNT.getBytes(), savings);
        // Activate the POA manager
        rootPOA.the_POAManager().activate();
        // Associate the objects with a name in the name service
        NameComponent[] rName = { new NameComponent(SAVINGSACCOUNT, "")};
        ((NamingContext)root).rebind(rName, myPOA.servant_to_reference(savings));
        System.out.println(myPOA.servant_to_reference(savings)+ " is ready.");
        // Wait for incoming requests
        orb.run();
    }
catch (Exception e) {
    e.printStackTrace();
}

public void run() {
    try {
        while(true) {
            deposit(1000);
            Thread.sleep(10000);
            withdraw(500);
            Thread.sleep(10000);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
} // end of class SvingsAccount

/* ACCOUNT.idl - The specification of Automated Banking */
module ACCOUNT {
    interface SavingsAccount {
        long long transferOut();
    };
    interface CheckingAccount {
        void tryTransfer();
    };
};
Appendix D

SOURCE CODE OF DINING PHILOSOPHERS

/* ForkImpl class - This class implements the Fork interface */
package DiningPhilosophers;
public class ForkImpl extends FORKS.ForkPOA {
    public int id;
    public ForkImpl(int id) { this.id = id; }
}

/* Philosopher class - This class implement a philosopher */
package DiningPhilosophers;
public class Philosopher extends Thread {
    protected int id;
    protected FORKS.Fork left, right;

    public Philosopher (int id, FORKS.Fork l,FORKS.Fork r){
        this.id = id;
        left = l;
        right = r;
    }
    public void run() {
        while (true) {
            think();
            synchronized (left) {
                synchronized (right) { eat(); }
            }
        }
    }
    private void eat () { System.out.println("Philosopher " + id + " is eating!"); }
    private void think () { System.out.println("Philosopher " + id + " is thinking!"); }
} // end of class Philosopher

/* Process class - this class create the philosopher thread, and fork object */
package DiningPhilosophers;
import java.util.*;
import java.io.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.CosNaming.*;

public class Process extends Thread {
    public static final String FORK = "Fork";
    private NamingContextExt root;
    private int iid;

    public Process (NamingContextExt root, int id) {

108
this.root = root;
idd = id;
}
public static void main(String args[])
{
    if (args.length != 1) {
        System.out.println("Usage: Process id");
        System.exit(1);
    }
    int idd = Integer.parseInt(args[0]);
    ForkImpl fork1, fork2, fork3;

    try {
        // 1. Initialize the ORB
        ORB orb = ORB.init(args, null);
        // get a reference to the root POA
        POA rootPOA = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));

        // get a reference to the Naming Service root_context
        org.omg.CORBA.Object rootObj = orb.resolve_initial_references("NameService");
        NamingContextExt root = NamingContextExtHelper.narrow(rootObj);

        // Create policies for our persistent POA
        org.omg.CORBA.Policy[] policies = {
            rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
        };

        // Create myPOA with the right policies
        POA myPOA = rootPOA.create_POA("dining_poA", rootPOA.the_POAManager(), policies);

        // 2. Instantiate Service Objects
        fork1 = new ForkImpl(1);
        fork2 = new ForkImpl(2);
        fork3 = new ForkImpl(3);
        Process process = new Process(root, idd);
        process.start();

        // 3. Activate the objects with the ID on myPOA
        if (idd == 1) {
            myPOA.activate_object_with_id((FORK+1).getBytes(), fork1);
            myPOA.activate_object_with_id((FORK+2).getBytes(), fork2);
        } else
            myPOA.activate_object_with_id((FORK+3).getBytes(), fork3);

        // Activate the POA manager
        rootPOA.the_POAManager().activate();

        // Associate the objects with a name in the name service
        if (idd == 1) {
            NameComponent[] rName1 = { new NameComponent(FORK+1, "");
            ((NamingContext)root).rebind(rName1, myPOA.servant_to_reference(fork1));
            NameComponent[] rName2 = { new NameComponent(FORK+2, "");
            ((NamingContext)root).rebind(rName2, myPOA.servant_to_reference(fork2));
        }
} else {
    NameComponent[] rName3 = { new NameComponent(FORK+3,"")};
    ((NamingContext)root).rebind(rName3, myPOA.servant_to_reference(fork3));
}

System.out.println("Servant is ready.");
// Wait for incoming requests
orb.run();
}
catch (Exception e) {
    e.printStackTrace();
}
}

public void run() {
    Philosopher phil1, phil2, phil3;
    FORKS.Fork fork1, fork2, fork3;

    try {
        Thread.sleep(30000);
        NameComponent[] rName1 = { new NameComponent(FORK+1,"")};
        fork1 = FORKS.ForkHelper.narrow(((NamingContext)root).resolve(rName1));
        NameComponent[] rName2 = { new NameComponent(FORK+2,"")};
        fork2 = FORKS.ForkHelper.narrow(((NamingContext)root).resolve(rName2));
        NameComponent[] rName3 = { new NameComponent(FORK+3,"")};
        fork3 = FORKS.ForkHelper.narrow(((NamingContext)root).resolve(rName3));

        if (idd == 1) {
            phil1 = new Philosopher(1, fork1, fork2);
            phil1.start();
            phil2 = new Philosopher(2, fork2, fork3);
            phil2.start();
        } else {
            phil3 = new Philosopher(3, fork3, fork1);
            phil3.start();
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}

} // end of class Process

/* FORKS.idl - The specification of Fork in Dining Philosophers problem */
module FORKS {
    interface Fork {
    }
};
Xiubin Cai was born in 1968 in Fujian, China. She graduated from Beijing Science & Technology University, Beijing, P.R.C in 1990, where she received a Bachelor's degree in Computer Engineering. She is currently a candidate for a Master's degree in Computer Science at University of Windsor and expects to graduate in the summer of 2000.