Aspect-Oriented Programming for Guided Testing

SiyuanLiu[a], Yang Yang[b]

Power Construction Corporation of China, Haidian District, 22# Chegongzhuangxilu St, Beijing, China

liusy@powerchina.cn[a], yangy@powerchina.cn[b]

Keywords: AOP, PUT, specification-based testing, distributed systems, nondeterminism, AspectJ, test guide.

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

Introduction

With the advances of modern computers and computer networks, distributed and/or multithreaded software systems are becoming more and more popular. Improving qualities of these systems is an important task we are facing. However, distributed and multithreaded systems are very often much more complex to analyze or test due to the nondeterminism involved. Unlike traditional sequential systems, an input sequence given to the system may have several different execution paths depending on the interactions among different threads in each process and/or different processes possibly running on different machines across networks. Reproducible testing [1] is one of the possible approaches to performing testing in such an environment. With this approach, a test scenario consists of an external sequence of events (i.e. test case) and some additional information. The test case describes the external input and the observations, while the additional information, called test constraint, describes some constraints on the execution paths, such as partial or total order of the execution of some statements in the program. We introduce an additional process called test guide into the testing procedure so that given a test scenario, the system under test (i.e. the set of processes instantiated from the program under test) can be guided to take a certain execution path based on the given input sequence and the test constraint. Then the external observations can be compared with the desired ones.

Objective

To realize guided testing, we have two tasks to accomplish: one is to provide the test guide, and the other is to establish the communication between the test guide and the processes in the system under test, so that each process/thread can communicate with the test guide at the desired checkpoints.

With this added communication, the test guide will be able to decide, according to the test constraint, whether a thread should proceed, wait for other threads, or resume from waiting state, based on the overall test constraint and the current status information of other threads either in this process or in other processes. In this way, the execution is guided to take a desired path.

The test guide can be a generic tool that is developed once for all as long as its communication protocol with any system under test is predefined. Such a tool was developed using Java Remote Method Invocation (RMI) and reported in [2].

To establish the communication between the test guide and the processes in the system under test, there are various ways: We can automatically insert auxiliary code into the program under test, or alter the execution environment (such as Java Virtual Machine for program under test writing in Java). In [2], it has been discussed a method and related tool support for automatically inserting additional code into the source code of any given Java program with Remote Method Invocation (RMI). In [3, 4]. In this work, we present another solution to realize such communication by making use of the features in Aspect-Oriented Programs. We provide an automated translation from a test constraint to a set of aspects using AspectJ. The translated AspectJ program is then woven into the program under test. This solution does not require the availability of the source code of the program under test as in [3], and it is not dependent on the specification and implementation of Java runtime environment as in [2].

Reproducible testing with distributed bakery algorithm

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

Case 1: Process 1 enters critical section first.

Case 2: Process 2 enters critical section first.

Proposed Approach

We gave the classification of reproducible testing approaches and Aspect-Oriented Programming used in testing. Furthermore we will define the constraints manually created to identify point of interests to be tested [2, 5, 6, 7]. In this work, we put the emphasis on approaches which make the test reproducible and controllable. Our work does not require the availability of the program code, but it needs user’s knowledge to fix feasible test constraints to avoid deadlock during the control of the test. We provide an automated translation from a test constraint to a set of aspects using AspectJ, The translated AspectJ program is then woven into the program under test. The advantage of AspectJ is that it is simple and practical to use, and it provides support of a range of crosscutting concerns for modular implementation. Furthermore, AspectJ code can be written once for all, and automatically woven into the code of any program under test.

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

Testing Architecture

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

The structure of our testing method is shown in Fig.1. A test case is usually defined as a sequence of input/output pairs. For simplicity, we consider here that each test case is an input/output pair. An extension of the current work to handle sequences of input/output pairs is straightforward. Each test case is associated with a test constraint which describes a partial order among internal events, which refers to the execution of a process at a checkpoint in the source code. We assume that program under test, test case, and test constraints are given. AspectJ code is automatically generated from the test constraint, and woven into the given program under test, to form the extended program under test. The execution of the extended program under test is augmented by the communication with the test guide, which is written once for all. The test guide takes the test constraint as input to make sure it is satisfied by properly delaying the communication with the processes in the system under test. As we noted in the Introduction, we assume that the given test constraint can uniquely determine one output, and this actual output is compared by a test oracle with the expected one given in the test case.

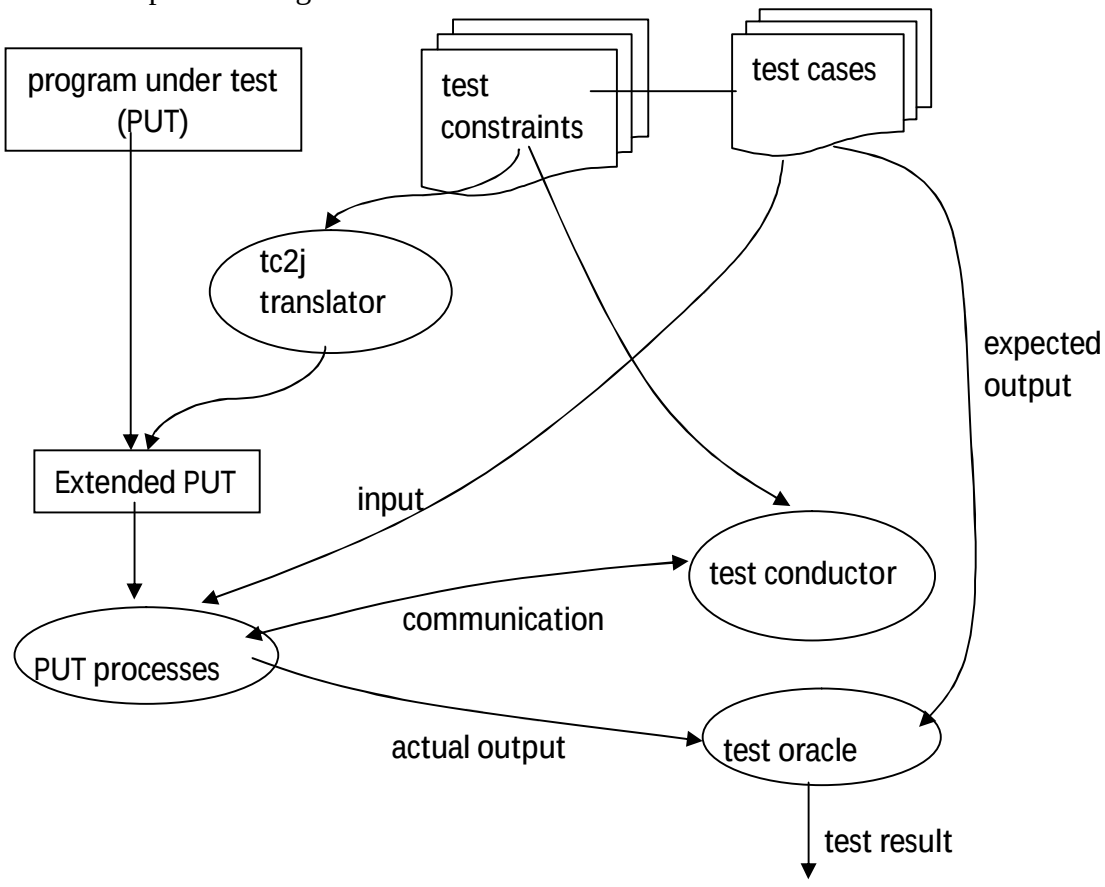

Fig.1: Testing Architecture

Event description and test constraint

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

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

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

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

cName is the class name of the method being invoked.

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

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

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

e1: (1, 2, SocketThread, handleRequest, 1)

e2: (1, main, DBakery, pickNum, 3)

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

A test constraint file describes a partial order among a set of internal events. Both the AspectJ generator and the test guide read this file to extract useful information. A possible test constraint file for the distributed bakery example is given in Fig. 2

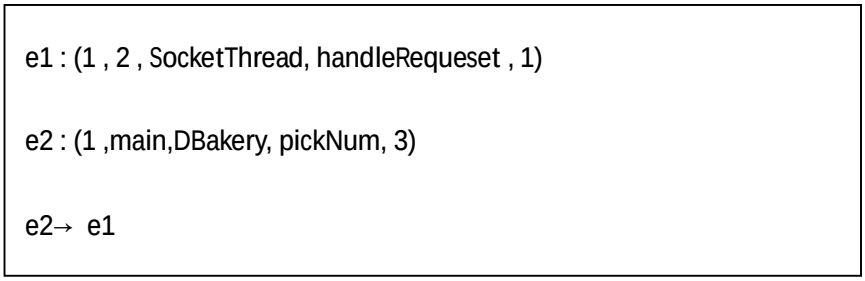

Fig. 2: A test constraint file

Generating AspectJ Code from Test Constraints

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

Data Structure of the Test Guide

The class TestGuide first declares an integer to count the numbers of constraint relations in the second part of the test constraint file. Then it declares 3 arrays of this length. The first array, named preEvent of type string, is used to record the names of the events appeared before the happen-before relation "→". The second array, named postEvent of type string, is used to save the names of the events appeared after the happen-before relation "→". For example, with e2→e1, e2 is saved in the array preEvent[i], and e4 is saved in postEvent[i]. For each relation, the two events are saved separately in preEvent and postEvent but with the same array index i. The third array named "condition" is of type Boolean. Its elements are set to false initially.

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

Algorithm of the Test Guide

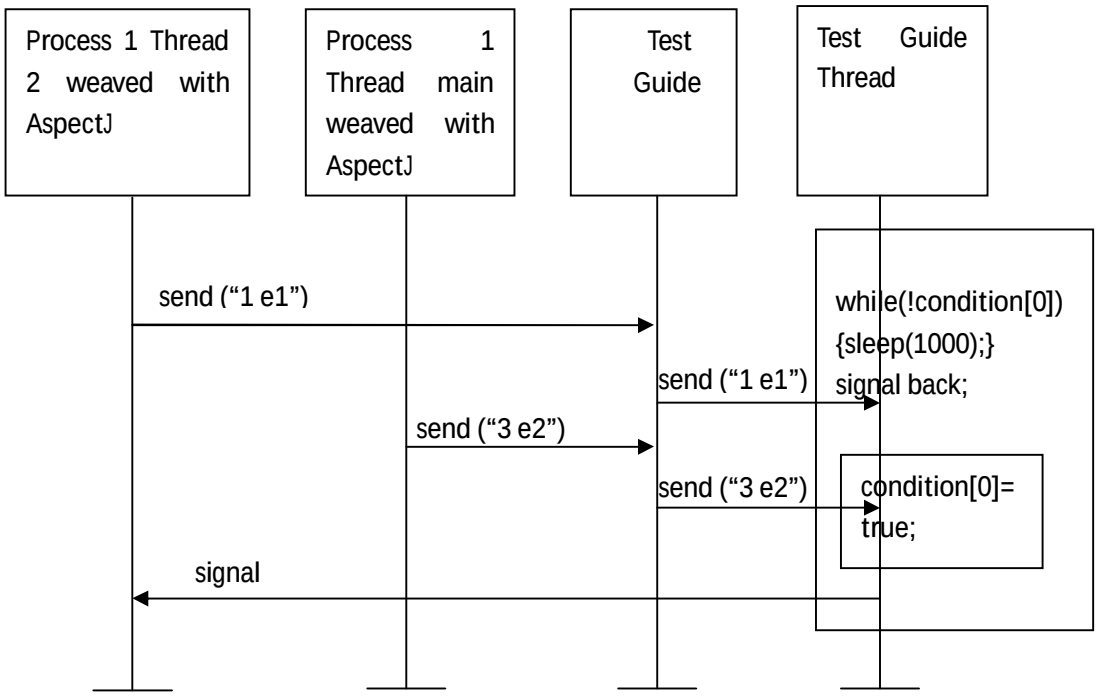

Fig.3: sequential chart of {e2→e1}

During the execution, it may happen a situation where one postEvent has two prevent to correspond. To make sure that the test guide thread will not give back a signal immediately after the cond of (eName', cond) comes true, we use a counter named countPreEvent. countPreEvent will count the number of the preEvent of a postEvent before the TestGuideThread functions. Each cond of (eName', cond) turned to true will lead to countPreEvent--. TestGuideThread will not signal the extended PUT until countPreEvent becomes 0.A sequential chart showing how the test guide handles incoming events is given in Fig. 3.

Conclusions

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

Acknowledgements

This work was financially supported by Digital Hydropower Project of Power Construction Corporation of China (CHC-KJ-2009-01).

References

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

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

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

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

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

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

[7] X. Wang AND J. Zhang, Analysis and Research on Distributed Network Protocol Testing Controllability Problem. 5th International Conference on Information Engineering for Mechanics and Materials .Atlantis Press (2015).