

Improving Automated Testing of Multi-threaded Software

Ayla Dantas, Francisco Brasileiro, Walfredo Cirne
Universidade Federal de Campina Grande (UFCG)
Laboratório de Sistemas Distribuídos
{ayla, fubica, walfredo}@dsc.ufcg.edu.br

Abstract

This paper discusses an approach to avoid incorrect results in the execution of automatic tests of multi-threaded systems. We argue that such incorrect results have two main sources. First, it is typically difficult to determine when all threads have finished processing and thus when it is safe to perform the test assertions. Second, background threads can change the system state while assertions are being performed, thus producing non-deterministic results. The main contributions of this work are: (i) a generic approach that ensures that test assertions are performed in a safe moment; (ii) implementation details of such an approach using Aspect-Oriented Programming (AOP); and (iii) an evaluation of the proposed approach.

1. Introduction

A common problem while automatically testing multi-threaded software is to have tests that fail even though the failure cause is not a bug. While developing distributed systems in our lab, we have noticed that a common cause for this problem is that the operations to be tested are asynchronous and many times the test verifications are performed at the wrong moment. For instance, the test being executed does not wait enough time between stimulating the system and checking if the system behaved as expected (assertion). To solve that, test programmers tend to increase this wait time, often making test execution time longer than necessary and yet not assuring the problem is solved. Another problem is that the assertion may be executed too late, when the condition being verified is not valid anymore. Such false test failures have the bad impact of making developers no longer trust test results. In this paper we propose an approach to avoid these problems. It is intended to help test developers by making their test execution results more deterministic, eliminating the need for estimating an appropriate time to wait before performing assertions. Our main motivation is that multi-threaded applications are becoming

more and more popular with the advent of multicore processors [9] and testing such systems is a challenging task. One of the difficulties is avoiding test failures that are actually **false negatives**, which are test runs that fail even though no problem actually exists with the system [4]. This leads to a lot of time waste trying to find bugs that do not exist or searching bugs in the wrong places. The approach proposed in this work is based on the monitoring of the system threads during tests execution for a better identification of the correct moment to perform assertions in tests. Besides, a better control of threads must also be provided to avoid system changes while assertions are being performed. We propose the use of Aspect-Oriented Programming [6] as a technique to perform threads monitoring and control in a transparent and modularized way. Moreover, we also evaluate our approach comparing it with two other ones that are in common use while testing asynchronous operations in multi-threaded systems.

The remaining of this paper is organized as follows: Section 2 contextualizes the problem of determining when to make assertions and presents two approaches commonly used. Section 3 presents the new approach that we propose, called the *ThreadControlForTests*, and also a framework to support it; Section 4 evaluates this approach; Section 5 discusses some related work; and finally, Section 6 presents some final remarks.

2. Determining When to Assert

There are lots of challenges in testing and debugging multi-threaded systems. Most of the research in this area focuses on finding problems such as data race and deadlocks using tools like Contest [2]. However, in order to better benefit from tools like this and to have trustworthy test results, it is important to have tests whose verifications are true independently of the order in which the system threads executed.

An automatic test of a system that is not concurrent normally comprises two parts: stimuli and verifications (assertions). In multi-threaded systems, between these parts it is

usually necessary to include a *wait* phase in case there are asynchronous operations whose results must also be verified through assertions. In the following, we present some approaches frequently used to define how long the tests should wait before performing assertions.

Explicit delays. The simpler way to make a test thread wait before performing its assertions is to include sleep or timed wait calls before assertions (e.g. `Thread.sleep(TIMEOUT)` in Java) [1]. They make this thread wait for a certain amount of time supposed to be enough for the execution of the system operations to be tested. However, the time to wait depends on the machine where the test is executed and on the load competing with the test. Therefore, using a small interval would lead many tests to fail due to lack of time for the system to finish the stimulated operations. On the other hand, using a long interval would lead to long test execution runs.

The Busy-wait Approach. Another way to verify in a test that a certain system state was reached and that an assertion can be performed is to use busy-waits. Busy-waits in tests are normally explicit delays inside loops where a condition is tested until it assumes a given value. However, guards using busy-waits have some drawbacks [7]: i) they can waste an unbounded amount of CPU time spinning without necessity and, ii) they can miss opportunities to fire if they happen not to be scheduled to execute during times when the condition is momentarily true. Another disadvantage that we can also observe is the necessity of operations provided by the system so that the test can verify if the condition tested in the loop guard has become true.

3. A Generic Approach: ThreadControlForTests

While developing the OurGrid system, an open source peer-to-peer grid middleware, we have improved our tests by providing to test developers a class with an operation called `waitUntilWorkIsDone` [3]. This operation made the test thread wait until all other threads started by the test were waiting or have finished. Lots of tests have benefited from this operation and did not fail anymore because of lack of time to execute asynchronous operations before the assertions. However, as the system evolved, we have noticed that test developers wanted some specific thread configurations to wait for. Besides, another problem noticed was that some periodic threads could wake up during assertions and lead to false negative results.

To solve the limitations of busy-waits, explicit delays and also of our previous work, we propose in this work a general approach to test multi-threaded systems and implementation details of a framework to support it. This approach is called `ThreadControlForTests`, and it proposes that tests involving asynchronous operations should present

the following phases: 1) prepare the environment stating the system state in which the assertions should be performed; 2) invoke the system operations, making it run and produce results to be verified; 3) wait until the expected system state has been reached; 4) perform the assertions, with no fear that a system thread will change the system expected state and perturb the system; 5) make the system proceed normally so that the test can finish.

From these phases, we propose that phases 1, 3 and 5 should be provided by a testing framework able to monitor system threads and notify the test as soon as an expected state is reached. Besides, the framework must also avoid system perturbations during assertions (phase 4). In the `ThreadControlForTests` approach, a state to be reached is defined in terms of threads configurations, which define some system threads and the states they should be in order to perform the assertion.

System state. In order to verify if a given state (a given set of threads and their states) has been achieved, we need a way to describe such state. A system state to wait for can be described using a set of Thread class names and the states in which they should be. We have considered that state changes for each thread are caused by method calls and executions. Based on Java 1.4 basic thread states and their lifecycle, we propose the thread states and transitions illustrated by Figure 3. Each node there represents a state and the arrows represent the method calls or executions after/before which a thread change happens. For instance, once the execution of the `run` method from a class implementing the `Runnable` interface has finished, we can consider that a Thread has finished. A given *system state* to wait for can be the moment at which given Threads A and B have finished, for example. It is important to notice that other states and transitions, besides the ones shown in Figure 3 can also be used. This would happen when, for testing purposes, it is interesting to define application-specific states and transitions.

The ThreadControl Framework. To enable the use of the generic approach for testing, besides defining appropriate states and transitions, it must be provided a way to monitor system execution during tests and to avoid interruptions during assertions. This can be obtained by several changes in the system source code, which may make it difficult to understand and is also very error prone.

This can be avoided through the use of Aspect-Oriented Programming (AOP) in the implementation of a testing framework to aid the `ThreadControlForTests` approach. AOP [6] is a programming paradigm that aims to clearly address the *aspects* or system *concerns* whose implementation *crosscut* traditional modules. AOP proposes that those aspects that could cause code tangling and scattering should be written separately from the functional code. An example of an aspect language is AspectJ [5], which is a general-

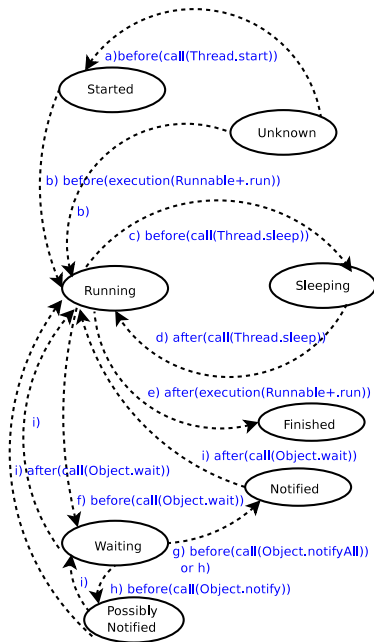


Figure 1. Each thread life cycle

purpose aspect-oriented extension for Java. AspectJ supports the concept of *join points*, which are well-defined points in the execution flow of a program [10]. It also has a way of identifying particular join points (*pointcuts*) and a mechanism to change the system behavior at join points (*advice*). Pointcuts and advice are defined in aspects.

Considering such AOP concepts and the ThreadControlForTests approach, we have identified execution points (pointcuts) that represent main changes on the system state. Once those execution points are reached, there are advices (which are basically methods) to update the system overall state and to delay state changes (just in case an assertion is being performed). If an expected state with a given configuration of threads has been reached, the test thread is notified to perform its assertions without the disturbance of any other system thread (e.g. a periodic thread). The other threads do not interrupt the assertions executions because when they are about to change their state the advice code blocks them. Therefore, the test thread can perform assertions and any other thread trying to change its state is not allowed to proceed before assertions have finished.

Our testing framework is called ThreadControl. It is implemented in Java and AspectJ, but generic and similar frameworks can be built in other languages. Through a framework like ThreadControl, the test developer must simply invoke some method calls in the test. One of them is `prepare`, which is used to specify desired system configurations (threads states) for the moments in which assertions should be performed. Other methods also pro-

vided by the framework enable the test thread to wait until the system is in the expected configuration (specified in the parameter given for the `prepare` method). The `waitUntilStateIsReached` method allows the assertions to be performed at a secure moment. Another method call that should also be included in the tests is the one responsible for making the system proceed its normal execution so that the test may terminate or continue with other assertions: `proceed`.

In order to implement the ThreadControl framework using AOP we have implemented the ThreadControlAspect and the ThreadWatcher class. The pointcuts are defined according to the arrows illustrated by Figure 3. The advices are responsible for managing the current state of system threads and for controlling them. The source code of the ThreadControl framework is available as open source at <http://www.dsc.ufcg.edu.br/~ayla/threadControl>.

4. Evaluation

In this section we compare the ThreadControlForTests Approach with the ones currently used while testing asynchronous systems: the Explicit delays and Busy-wait. They are compared considering the following criteria: 1) Avoidance of state changes during assertions and miss of opportunity to fire; 2) Necessity of code changes in the system to enable the approach; 3) Ease of use; 4) Time to test; 5) Occurrence of false negatives.

We can notice that only the ThreadControlForTests Approach avoids state changes during assertions, avoiding miss of opportunity to fire. Regarding the necessity of direct code changes in the system, it is only necessary while using the Busy-wait Approach, as the condition verified may not be already exposed in the system's API. In the ThreadControlForTests Approach using AOP, code changes are transparent and happen at compile time. Considering the *Ease of use* aspect, we have considered the number of lines necessary to implement a test following each of the approaches. Basically, the Explicit Delays Approach is the simplest one to use. The other approaches' complexity depends on the number of states to wait for in all tests (considering the ThreadControlForTests Approach) and on the number of changes in the base system (for the Busy-wait Approach).

In order to evaluate the time to test and the occurrence of false negatives we have performed some initial experiments with two tests. They were chosen based on common testing scenarios of two systems in our lab: OurGrid and OurBackup (a peer-to-peer backup system). Three versions of these tests were developed: one using sleep calls (the explicit delays approach), another using busy-waits and a third one using the generic approach through the ThreadControl framework. Both tests explore simple

Table 1. Test 1 results

Approach	Frequency of false negatives	Mean time before assertion(ms)
Generic	0%	80
Simplistic (t=80)	99.7%	84
Simplistic (t=83)	3.5%	86.67
Simplistic (t=100)	2.6%	102.52
Busy-wait (t=1)	0%	87.5
Busy-wait (t=80)	0%	201.47

asynchronous operations and only the second one explores miss of opportunity to fire. In both cases, the ThreadControlForTests approach did not lead to any false negatives and was also the approach in which test runs took less time.

For the first test, we have noticed that false negatives only happen when the explicit delays are used and their frequency depends on the timeout used by the test. For instance, using 80ms as timeout, the frequency of false negatives is 99.7%, and increasing it to 83ms the frequency of false negatives goes to 3.5% (see Table 1). However, increasing the timeout used by the test leads to greater test execution times. For instance, while using only 100ms as timeout, the time taken by the test execution was 28.15% longer than with the ThreadControlForTests approach. For the Busy-wait approach, reducing the timeout leads to a faster test execution, but also to a lot of CPU consumption. This also happened with the second test while using busy-waits. The only difference was that false negatives also happened in this case, but their frequency was reduced when smaller timeouts were being used. For instance, with a timeout of 83ms, false negatives appeared in 5.3% of the test runs. If we reduced the timeout to 1ms, the frequency of false negatives decreases to 0.1%.

5. Related Work

The closest related work is [3], in which we discuss how AOP has been used to help in the testing process of the OurGrid project. In that work we have proposed the `waitUntilWorkIsDone` operation in which there was only one system state configuration to wait for (all threads started by the test have finished or are waiting). In the ThreadControlForTests Approach, we extend this initial idea by making it possible the definition of other system states to wait for and we also avoid system state changes during assertions. The work by Coptly and Ur [2] is another work related to ours. It investigates the suitability of AOP for testing tools by implementing a testing tool (ConTest) using AspectJ. This tool is used to force different thread schedulings during several re-executions of test suites in order to find concurrency problems. We believe our approach

must be used in addition to test tools like this to assure that a failure while re-executing a test that has passed before is in fact a bug and not a timing problem. Regarding threads monitoring, we may cite the work of Moon and Chang [8], which uses the technique of code inlining for monitoring.

6. Conclusions and Future Work

We have proposed in this paper an approach for determining when to make assertions while testing multi-threaded systems and a framework to support it. We conclude that the approach proposed here can be used to securely determine the adequate moment to perform assertions and to avoid major changes in this state while verifications are being performed, avoiding false negative of tests. In addition to this, our initial experiments have shown that, besides being non-deterministic, the two approaches in common use today lead to longer test execution times, depending on the timeouts chosen by test developers.

In the future, we plan to explore the ThreadControl framework in other systems. We want to measure how much test execution time can be saved using it and how many false negatives we could avoid in those systems. Besides, we also plan to extend these ideas in tests that use distributed components that may run on different machines.

References

- [1] Xunit Patterns: Slow Tests. <http://xunitpatterns.com/Slow%20Tests.html>, 2007.
- [2] S. Coptly and S. Ur. Multi-threaded Testing with AOP is Easy, and It Finds Bugs. *Proc. 11th International Euro-Par Conference, LNCS*, 3648:740–749, 2005.
- [3] A. Dantas, W. Cirne, and K. Saikoski. Using AOP to Bring a Project Back in Shape: The OurGrid Case. *Journal of the Brazilian Computer Society (JBSCS)*, 11:21–36, 2006.
- [4] E. Dustin, J. Rashka, and J. Paul. *Automated Software Testing: Introduction, Management, and Performance*. Addison-Wesley Professional, 1999.
- [5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting Started with AspectJ. *Communications of the ACM*, 44(10):59–65, 2001.
- [6] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming, ECOOP'97*, LNCS 1241, pages 220–242, Finland, June 1997. Springer-Verlag.
- [7] D. Lea. *Concurrent Programming in Java (tm): Design Principles and Patterns*. Addison-Wesley, 2000.
- [8] S. Moon and B. Chang. A thread monitoring system for multithreaded Java programs. *ACM SIGPLAN Notices*, 41(5):21–29, 2006.
- [9] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.
- [10] A. Team. The AspectJ Programming Guide. At <http://www.eclipse.org/aspectj>.