# Obtaining Trustworthy Test Results in Multi-threaded Systems

**Ayla Dantas, Matheus Gaudencio, Francisco Brasileiro, Walfredo Cirne**

[1]Laboratório de Sistemas Distribuídos
Universidade Federal de Campina Grande (UFCG)
Av. Aprígio Veloso, 882 – Bloco CO – Campina Grande – PB – Brazil

{ayla, matheusgr, fubica, walfredo}@lsd.ufcg.edu.br

***Abstract.*** *Testing multi-threaded systems is quite a challenge. The non-determinism of these systems makes their implementation and their test implementation far more susceptible to error. It is common to have tests of these systems that may not pass sometimes and whose failures are not caused by application faults (bugs). For instance, this can happen when test verifications (assertions) are performed at inappropriate times. Unreliable tests make developers waste their time trying to find non-existing bugs, or else make them search for bugs in the wrong place. Another problem is that developers may cease to believe that certain test failures are caused by software bugs even when this is the case. In this paper, we propose an approach to avoid test failures that are caused, not by application bugs, but by test assertions performed either too early or too late. Our basic idea is to use thread monitoring to support test development. Moreover, we put forward a testing framework that follows the proposed approach, and we also evaluate the effectiveness of this framework for testing a multi-threaded system.*

## 1. Introduction

Software testing is a fundamental activity of software engineering [Bertolino 2007], and its purpose is to secure the quality of the final product. Test activities usually include the following steps: 1) designing test cases; 2) executing the software with these test cases; and 3) examining the results produced by these test executions [Harrold 2000].

When testing multi-threaded systems, it is not always easy to determine when the results produced by the system execution can be examined. This happens when the tests being run exercise the system's asynchronous operations. In some cases, we have tests that may fail because we did not examine the operation results at the right time - for instance, when the test assertions are performed either too early (when the system is still carrying out the desired operation), or too late (when the state being verified is no longer valid).

The main purpose of our work is to suggest a way of preventing this particular problem and increase test reliability. As tests do not indicate the absence of bugs, but their presence [Dijkstra 1969], we want to make sure that behind a test failure there will always be a bug that needs to be searched.

One of the facts that has motivated our work is that multi-threaded applications have become increasingly common, especially with the advent of multicore processors.

On the other hand, this has contributed to expand still further the complexity of software development. As a result, it has established the need for more resourceful programming tools capable of pinpointing defects in a more systematic fashion and assisting programs debugging, identifying performance bottlenecks, and facilitating testing activities [Sutter and Larus 2005].

In this paper's context, testing is especially challenging due to the inherent nondeterminism of multi-threaded applications. Different thread interleavings may happen for different executions of the same test case [MacDonald et al. 2005]. However, one expects the system to work correctly for all possible executions. Failures that only occur in some executions must have been caused by intermittent bugs, and not by test defects, as it would be the case sometimes. Because these bugs are difficult to find, when the test itself is the problem, developers will waste too much time trying to find bugs that do not exist, or searching for bugs in the wrong place. Another negative effect is that developers may become suspicious of test results. For instance, in the event of a test failure, developers may not believe the failure has been caused by a bug, even when that is exactly the case. This has also been observed elsewhere [Stobie 2005].

Our main purpose is to help test developers to avoid failures in multi-threaded systems tests caused by assertions performed at inappropriate times. To avoid that, a possible solution is to make the tests wait until a certain stable state of the system threads has been obtained, as discussed in another work [Goetz and Peierls 2006]. Nevertheless, this solution has been described by Goetz and Peierls as an 'easier said than done' technique. Because of that, developers tend to use, in practice, explicit delays (such as `Thread.sleep(timeInterval)` ), which can lead to spurious test failures.

We propose in this work an approach to avoid such spurious test failures caused by test assertions performed too early or too late. This approach is based on the monitoring of application threads and on simple operations made available to test developers. In order to support this approach, we have developed a testing framework which uses Aspect-Oriented Programming [Kiczales et al. 1997] to monitor and provide a better control of application threads during tests. Moreover, we have evaluated the application of this framework to some tests of OurBackup [Oliveira 2007] – a peer-to-peer backup system.

This paper is organized as follows. In Section 2, we give an overview of the main research directions on multi-threaded systems testing. Besides, we also introduce in this section the specific problem we are dealing with. In Section 3 we present our approach and in Section 4 we present more details about the framework to support this approach. In Section 5 we present an evaluation of our work. In Section 6 we discuss some related work, and finally, in Section 7, we present our conclusions and point out directions for future research.

## 2. Multi-threaded systems testing

There are several works that deal with tests in multi-threaded systems. The purpose of some of them is to find concurrent defects, such as unintentional race conditions or deadlocks. The research areas in this field include: data race detection [Naik et al. 2006, Savage et al. 1997], static analysis [Rinard 2001], replay techniques [Ronsse and De Bosschere 1999, Choi and Srinivasan 1998], techniques to force some thread interleavings [MacDonald et al. 2005], and techniques to generate differ-

ent interleavings in order to reveal concurrent faults [Stoller 2002] (such as the ConTest tool [Copty and Ur 2005]).

However, none of these works deals with the problem we are handling. In the present work, our purpose is to help test developers in better predicting the moment to perform test assertions. This is most demanding when the test involves asynchronous operations. Our intention is to avoid intermittent failures in tests and, more specifically, failures which are not caused by application faults.

Before performing a test assertion, we have to guarantee that the system has completed the operation being tested. However, this is not always easy, as it has also been observed by Goetz and Peierls [Goetz and Peierls 2006]. The authors say that a possible strategy is to wait until the thread blocks. A similar observation can also be found in the HarnessIt tool documentation [United Binary ]. According to this documentation, test methods on multi-threaded objects should not exit before all child threads created by the target object have completed their processing.

In order to better understand what has been discussed, we will present an overview of the tests that involve asynchronous operations. We will also discuss some common techniques used in these tests.

### Waiting Before Assertions

When we have asynchronous operations in a test, the following test phases are generally found:

1. *Exercise the system*
2. *Wait*
3. *Perform assertions*

Let us consider, for example, a case in which phase 1 exhibits the following operation: `mySystem.initialize()` . This operation initializes the system, and, in order to do that, it creates other threads, which may also create some more threads (see Figure 1). These threads will perform operations and then stand waiting until other stimuli are provided.

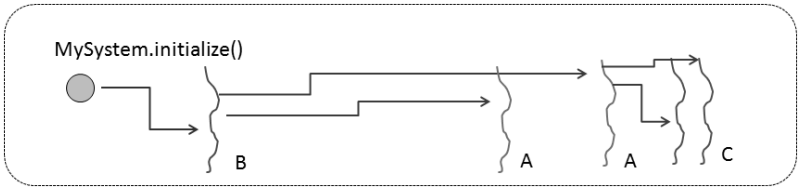


**Figure 1. Example: Threads created by an asynchronous operation being tested**

Considering the same example, suppose that in the assertions part (phase 3), we have the following code: `assertTrue(mySystem.isCorrectlyInitialized())` .

This code verifies whether the system has been correctly initialized. In case it has not, the test will fail. Nevertheless, before verifying whether the system has been correctly initialized, we have to wait until the initialization process has finished. Defining how much time to wait before performing assertions, especially when multiple threads

are created, may be challenging. Moreover, depending on the approach used, some test failures may occur.

When there are asynchronous operations in a test, some of the common forms to implement the *wait* phase before assertions are:

1. *Explicit Delays.* Ex:
```
Thread.sleep(t);
```
2. *Busy Waits.* Ex:
```
while (!stateReached())
  Thread.sleep(t);
```
3. *Control of threads.* Ex:
```
myThread.join();
```

### Explicit Delays and Busy Waits

It is common to use explicit delays in tests because they are easy to implement. However, they may bring in two problems: test failures due to insufficient delay times; or tests that take too long because of extremely large delay times chosen by test developers [Meszaros 2007].

Busy waits may also be common, especially when the condition being verified in each iteration is easily obtained from the system. However, guards using busy waits have some drawbacks [Lea 2000], such as: i) they can waste an unbounded amount of CPU time spinning without necessity; and ii) they can miss opportunities to fire if they are not scheduled to execute during times when the condition is momentarily true.

### Control of Threads

A more secure way to know when an assertion can be performed is through the control of the threads created in the test. We must assure that they have finished or are in a stable state (such as all threads waiting) before assertions are performed.

When we have the instances of the thread objects in the test, this is simpler, and operations such as `Thread.join()` can be invoked. When this operation is used in a test, the test will only proceed when the given thread instance has finished its job. However, it is not always possible to have access to the instances of all threads created by the test. Sometimes we will need to monitor the application threads in order to know when to perform assertions. Besides, some thread control mechanisms may also be needed, because some threads may wake up and mess up test results when assertions are still being performed.

In the following section, we propose an approach based on the control of threads to provide this support for test developers.

## 3. The *Thread Control For Tests* Approach

In the Distributed Systems Lab (LSD) at Federal University of Campina Grande (UFCG), the problem of test failures that were not caused by bugs could be easily observed. It was in general caused by assertions performed at inappropriate times. For instance, when the

test was run in a slower machine, delay times being used could be insufficient and the test could fail.

As an attempt to avoid this problem in the development of OurGrid [Cirne et al. 2006], an open source peer-to-peer grid middleware, we have improved our tests by providing to test developers an operation called `waitUntilWorkIsDone` [Dantas et al. 2006]. This operation made the test thread wait until all other threads started by the test were waiting or have finished.

Many tests have benefited from this operation and did not fail anymore due to insufficient time to execute asynchronous operations before the assertions. However, as the system evolved, we have noticed that test developers began requiring some specific thread configurations (besides *the all threads waiting or finished* configuration). Moreover, another problem noticed was that some periodic threads could wake up during assertions and mess up test results.

To solve the limitations of busy-waits, explicit delays and also of our previous work [Dantas et al. 2006], we propose in the present work a general approach to test multi-threaded systems. This approach is called *Thread Control For Tests*, 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 (the expected state);
2. invoke the system operations, **exercising the system**;
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 disturb the system;
5. make the system **proceed** normally so that the test can finish.

Concerning these phases, we propose that phases 1, 3 and 5 should be provided by a testing tool. Our basic idea is that this tool must make some simple operations available to test developers. In order to do that, the tool should be able to monitor system threads and it should notify the test thread as soon as an expected system state is reached. Besides, the testing tool must also avoid system perturbations during assertions (phase 4).

In the *Thread Control For Tests* approach, a system state to be reached is defined in terms of thread states. Such expected state can be defined in a general way, such as *wait until all threads created by the test are waiting or have finished*. We may also specify a wait condition in a more specific way, such as *wait until certain threads are in certain states*. To illustrate this latter case, let us consider the example previously given and illustrated by Figure 1. An expected system state (or *system configuration*) for this case could be when *Thread B* has finished and all instances of *Thread A* are waiting.

As we can observe, in order to support this approach, we must provide mechanisms to monitor state transitions. Figure 2 illustrates some examples of operations to be monitored, considering the Java language [Gosling 2000]. It also shows some possible states of system threads which are achieved before or after these operations are performed.

Taking the Java language as an example, we can consider that an application thread is running after it starts to execute its corresponding `run` method. We may also say that it has finished, when this method execution has ended. All these transitions should be

monitored in order to detect as soon as possible when an overall expected system state has been reached. Once this has happened, the test thread should be informed. Besides that, after this state is reached, some monitored transitions that could disturb test assertions should be blocked until they are explicitly allowed by the test thread to proceed.

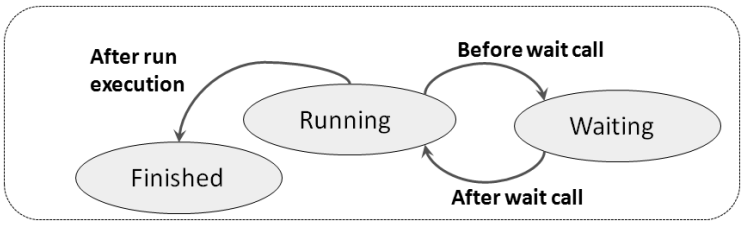


**Figure 2. Some monitored operations and related states**

While defining a *system configuration* in terms of the identification of certain threads and their associated states, one must define the states of interest. Moreover, one must also take into account before or after which points in the execution flow of a program a state transition happens. This should be considered while implementing test tools to support this approach. In order to aid the development of such tools, we have formalized some of these concepts in the following:

Let the set of system threads instantiated by a test case execution be $T = \{t_1, ..., t_{|T|}\}$. Each thread, at a given time, is at a state $s \in S$, where $S$ is the collection of all possible states in which system threads can be. The states set $S$ is defined by the testers considering states where threads should or should not be while assertions are being performed. The state in which a given thread $t_i \in T$ is at a given time is given by a function which we denote as $f : T \rightarrow S$.

Furthermore, let the system overall state at a given moment be a set of pairs $O = \{(t_1, f(t_1)), ..., (t_{|T|}, f(t_{|T|}))\}$. This overall state changes whenever a thread changes its state. For a thread $t_i$, a state transition happens after or before certain points are reached in the execution of the system. Let then $P$ represent the set of these moments at which state transitions happen.

In order to implement a test tool that follows our approach, one must define the set of possible states $S$, the possible state transitions $P$ and also a transition function $g : S \times P \rightarrow S$.

To use a test tool with these features, a test developer must define an expected state ($E$) for a given system. This state is defined in terms of a subset $J \subseteq T$ of the system threads and the desired states in which they can be found. An expected state is reached when for each thread $t_j \in J$, $\exists(t_j, f(t_j) \in O$ so that $f(t_j)$ is at one of these states.

To illustrate that, considering the example shown in Figure 2, we can state that $\{Finished, Running, Waiting\} \subset S$. We may also say that $\{$*After run execution*, *Before wait call*, *After wait call*$\} \subset P$.

This formalization is intended to guide the development of tools that support this approach, such as the testing framework that will be presented in the following section. However, an important attribute of the testing tool is to make the expected state definition as simple as possible, abstracting this formalism from the tool users (test developers).

Moreover, supporting tools that follow our approach should define their states and transitions of interest. These tools should be as general as possible in order to increase the reuse of such tools by several applications.

In the following section, while discussing our testing framework to support the *Thread Control For Tests* approach, we will show the common states and transitions under consideration. In addition to that, we will also present some technical details on how this framework can be extended to incorporate other states and transitions of interest.

## 4. The *ThreadControl* Testing Framework

In order to monitor several execution points of interest and also avoid many changes to the applications' code, a technique that can be used is Aspect-Oriented Programming (AOP) [Kiczales et al. 1997]. AOP is a programming paradigm that aims at addressing the *aspects* (or system *concerns*) whose implementation *crosscuts* traditional modules. AOP proposes that those aspects that could cause code tangling and scattering should be kept separate from the functional code.

An example of an AOP language is AspectJ [Kiczales et al. 2001], which is a general-purpose aspect-oriented extension to Java. AspectJ supports the concept of *join points*, which are well-defined points in the execution flow of a program. It provides ways of identifying particular join points (*pointcuts*) and a mechanism to define additional code that runs at join points (*advice*). Pointcuts and advice are defined in *aspects*.

Considering these AOP concepts and the *Thread Control For Tests* approach, we have identified execution points (*pointcuts*) that represent main changes on the system state. Once those execution points are reached, there come the *advices* (which resemble methods) to update the system overall state and to delay state changes (just in case an assertion is being performed). When an expected state with a given configuration of threads has been reached, the test thread will be notified in order to perform its assertions. The assertions are performed without any disturbance from any other monitored 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 the system state is not allowed to proceed before completion of these assertions.

Our testing framework is called *ThreadControl* and it is implemented in Java and AspectJ. However, other generic and similar frameworks can be built in other languages. The source code of the *ThreadControl* framework is available as open source at `http://www.dsc.ufcg.edu.br/~ayla/threadControl`.

If a framework like *ThreadControl* is used in a test with asynchronous operations, the test developer should simply include in the test some calls to framework operations. The main operations provided are the following:

- `prepare` : it is used to specify desired system configurations (in terms of thread states) for which the system should wait before performing assertions;
- `waitUntilStateIsReached` : it allows the assertions to be performed at a secure moment, when the state specified through the `prepare` operation has been reached. If a monitored thread tries to disturb the system after this moment, it will not be allowed to proceed;

- `proceed` : this is the operation responsible for making the system proceed its normal execution. Any threads that were not allowed to proceed because assertions were being performed are then released. This way, the test may terminate or continue with other assertions.

As we can observe, these operations correspond to phases 1, 3 and 5 of the phases proposed in the previous section. An illustration of the use of these operations is shown in the code below. This code is based on the system initialization example from Section 2. It shows a test case that uses the JUnit framework [Massol 2004].

```
1  public void testSystemInitialization(){
2    threadControl.prepare(expectedSystemConfiguration);
3    mySystem.initialize();
4    threadControl.waitUntilStateIsReached();
5    assertTrue(mySystem.isCorrectlyInitialized());
6    threadControl.proceed();
7  }
```

In this example, the system is exercised through the `mySystem.initialize` method call, which is not synchronous. In order to avoid executing the assertion at line 5 before the initialization has completed, we simply included a call to the `waitUntilStateIsReached` operation from the framework (see line 4). However, before it is invoked, there must have been a call to `prepare` in order to specify the system configuration to wait for (line 2).

To provide the operations above through the testing framework, we have implemented an aspect called `ThreadControlAspect` and some auxiliary classes. This code is combined (weaved) with the application code through the AspectJ compiler before executing the tests.

The pointcuts defined in this aspect correspond to state transitions. To identify what must be done before or after these points in the execution flow of the program, we define `before` and `after` advices. They are responsible for managing the current overall state of the system. Besides, they also have code to control threads that are about to change the system state while assertions are being performed.

In order to know when a state transition happened, we had to select some execution points that would lead to these transitions in the testing framework. Initially, we included support for the following Java operations involving threads: calls to `Thread.start` , executions of the `run` method from classes that implement the `Runnable` interface, calls to `Thread.sleep` , calls to `Object.wait` , calls to `Object.notifyAll` and `Object.notify` . The possible states for threads that have been considered were: *unknown, started, running, waiting, sleeping, notified, possibly notified and finished*. Considering the formalization presented in the previous section, these states correspond to the $S$ set for this framework. The determination of the states to which the threads should go when some execution points (such as the operations above) are reached corresponds to the implementation of the transition function $g$.

Some other operations from `java.util.concurrent` package (released in Java 5) are also managed by the framework, such as: calls to `take` and `put` methods on classes that implement the `BlockingQueue` interface, and also calls to `acquire` ,

`release` and `drainPermits` methods from the `Semaphore` class. During the implementation of the support for these methods, one should bear in mind the need for monitoring the internal elements of these structures (e.g. the elements of the queue or the semaphore permits). Depending on these elements, we will be able to determine with more precision in which state the threads are.

Other operations, including application specific operations, can also be managed. In order to do that and extend the framework, new pointcuts and corresponding advices should be added to the `ThreadControlAspect`. The pointcuts should correspond to the points in the execution that lead to state transitions. The advices should inform to auxiliary classes when a state transition has happened. Besides that, they should perform verifications in order to decide whether or not a state transition is allowed to proceed.

All the state management of the framework is done by the auxiliary classes, according to notifications sent by the aspect. We do not rely on the thread state information provided by the `Thread.getState` Java operation as this information may not be reliable. We have observed that such information may take some time to be updated according to some initial experiments we have performed. This was also observed by [Goetz and Peierls 2006], which also states that this operation is of limited usefulness for testing, although it can be used for debugging.

To illustrate how the framework can be extended to consider other transitions, we present below a code snippet of the `ThreadControlAspect` consisting of one of its advices:

```
1  after(Object o): waitCalls(o) {
2    verifyAndBlockThreadIfNecessary(thisJoinPoint);
3    threadWatcher.threadFinishedToWaitOnObject(
4      Thread.currentThread(), o, false,
5      WaitType.WAIT_ON_OBJ_LOCK);
6    verifyAndBlockThreadIfNecessary(thisJoinPoint);
7  }
```

This advice states what must be done after the call to `Object.wait` method returns. In general, the instances of auxiliary classes of the framework should be informed, as illustrated by lines 3-5. However, before informing about the state transition and after this notification, the framework verifies if the state transition is allowed to proceed (lines 2 and 6). If it is not, the current thread will be blocked until a `proceed` call is invoked by the test. It is important to notice that operations to inform about state changes should be synchronized to avoid data race conditions.

As we could observe, the framework presented follows the approach previously proposed. However, this framework does not constitute the only possible implementation. Solutions based on design patterns are also possible. We have decided to use AOP in order to avoid too many code changes in the application under test, making it also applicable to existing systems. By using AOP, it was possible to monitor several execution points in a transparent and modularized way, which did not require changes in the base code of the application. Should we need to remove this monitoring code used just for testing purposes, we simply compile the application without the aspect that introduces this functionality.

## 5. Evaluation

In order to evaluate our approach and its supporting framework, we have initially observed in some systems from our lab the problem of test failures which were not caused by bugs. In this section, we present some experiments we have performed using test cases of OurBackup [Oliveira 2007], a peer to peer backup system. Our main objective was to evaluate the use of the *Thread Control Approach* in a real system compared to the use of alternative approaches.

OurBackup was chosen as the real system of our evaluation because it presented several tests with asynchronous operations. Besides, we have observed that running those tests several times could in some rare executions lead to failures due to the problems of assertions performed at inappropriate moments.

The approach currently used by OurBackup to wait before assertions is based on a combination of busy waits and explicit delays. Explicit delays were used in addition to busy waits because the guard condition used in the busy waits was based on the `Thread.getState` operation, which is not always reliable, as previously discussed. In order to use the current approach being used there, some code conventions must be followed while implementing application threads. As following such conventions is not always possible for existing systems, many developers tend to use in similar occasions the *Explicit delays* approach [Goetz and Peierls 2006, Meszaros 2007]. Taking this into consideration, we have also decided to compare the use of this approach with the *Thread Control for Tests Approach*.

In order to do that, we have selected some tests from OurBackup that presented asynchronous operations. Then, we have evaluated three versions of these tests:

- *CurrVersion*: The current version of the tests, which uses as wait approach a combination of busy waits and explicit delays;
- *ThreadControlVersion*: A version of the tests using the *ThreadControl* testing framework;
- *ExplicitDelaysVersion(delayTime)*: A version of the tests using explicit delays. This version has been configured with different delay times as their values would influence both the tests execution times as well as the occurrence of failures caused by insufficient delays.

After implementing these three versions, we re-executed these tests several times. In these initial experiments, each test case was executed 650 times, using the same machine in a controlled environment. Then we measured the occurrence of failures on each of the 12 tests that had been selected. For the *ExplicitDelaysVersion* scenarios, we used, as the delay time parameter, a percentage of the mean time taken for each wait phase of the *ThreadControlVersion* corresponding tests. Therefore, we ended up with 4 variations of the *ExplicitDelaysVersion*. One of these variations uses, as delay times for each waiting phase, 100% of the mean delay times obtained from the execution of the *ThreadControlVersion* . We named this variation *ExplicitDelaysVersion (100%)*. The same procedure was used for the other three variations: *ExplicitDelaysVersion (80%)*, *ExplicitDelaysVersion (120%)* and *ExplicitDelaysVersion (150%)*. Table 1 shows the number of failures per test obtained from these 650 runs considering the different test variations. .

**Table 1. Test failures for each wait approach**

| | ThreadControlVersion | CurrentVersion | ExplicitDelays(100%) | ExplicitDelays(80%) | ExplicitDelays(120%) | ExplicitDelays(150%) |
|---|---|---|---|---|---|---|
| T1 | 0 | 0 | 150 | 596 | 25 | 34 |
| T2 | 0 | 0 | 244 | 632 | 63 | 35 |
| T3 | 0 | 0 | 340 | 642 | 56 | 58 |
| T4 | 0 | 0 | 188 | 525 | 41 | 58 |
| T5 | 0 | 0 | 69 | 301 | 17 | 14 |
| T6 | 0 | 0 | 202 | 612 | 25 | 21 |
| T7 | 0 | 0 | 109 | 647 | 13 | 8 |
| T8 | 2 | 5 | 291 | 641 | 9 | 9 |
| T9 | 0 | 0 | 123 | 619 | 12 | 11 |
| T10 | 0 | 0 | 274 | 650 | 9 | 6 |
| T11 | 0 | 0 | 391 | 650 | 32 | 40 |
| T12 | 0 | 0 | 132 | 650 | 6 | 7 |

As we can observe, depending on the approach used to wait before assertions we can obtain failures in a test because the verifications are not performed at the appropriate time. We could make this conclusion by observing the relation between the delay time and the number of tests with failures in the *ExplicitDelaysVersion* scenarios. Observing that fact, the developers of the *CurrentVersion* have used delay times considered to be enough in their tests. However, this practice does not guarantee that the test will pass in different machines. To illustrate this problem, we have performed 1000 executions of the same tests for the *CurrentVersion* and the *ThreadControlVersion* on a machine under a constant and heavy load. The failures obtained are shown in Table 2.

**Table 2. Test failures in a machine under heavy load**

| | ThreadControlVersion | CurrentVersion |
|---|---|---|
| T1 | 0 | 1 |
| T2 | 0 | 2 |
| T3 | 0 | 55 |
| T4 | 0 | 2 |
| T5 | 0 | 0 |
| T6 | 0 | 0 |
| T7 | 0 | 0 |
| T8 | 0 | 2 |
| T9 | 0 | 0 |
| T10 | 0 | 2 |
| T11 | 0 | 0 |
| T12 | 0 | 160 |

As we can notice, more tests have failed in this new run. Analyzing the test code, we could see that the causes for many of these failures are related to performing the assertions too early or even too late. Using our framework we could avoid failures caused by this problem. However, we cannot guarantee that all failures are due to this problem. For instance, while debugging the failure found in the controlled scenario (see Table 1) for the *ThreadControlVersion*, we discovered that this failure (which rarely occurs) was associated to a bug already known by the OurBackup development team. The objective of our work is to make developers focus on the real application bugs when a test failure happens and not on a possible problem with the test.

To avoid such failures while using alternative approaches, developers tend to increase too much the delay times used. Nevertheless, this leads to longer test execution times. This common practice of increasing delay times has motivated us to also measure in our initial experiments of the first scenario the total time taken for the tests execution considering each approach. The execution mean times of these tests, considering the 650

runs in a controlled environment, are shown in Figure 3. Our main conclusion from this initial experiment is that, for most of the tests, using the *ThreadControl* framework did not impose a great burden on the performance of the tests execution. Indeed, it could even improve the tests performance, especially considering that it is not always possible to define the most appropriate delay time to use when dealing with a heterogeneous test environment.

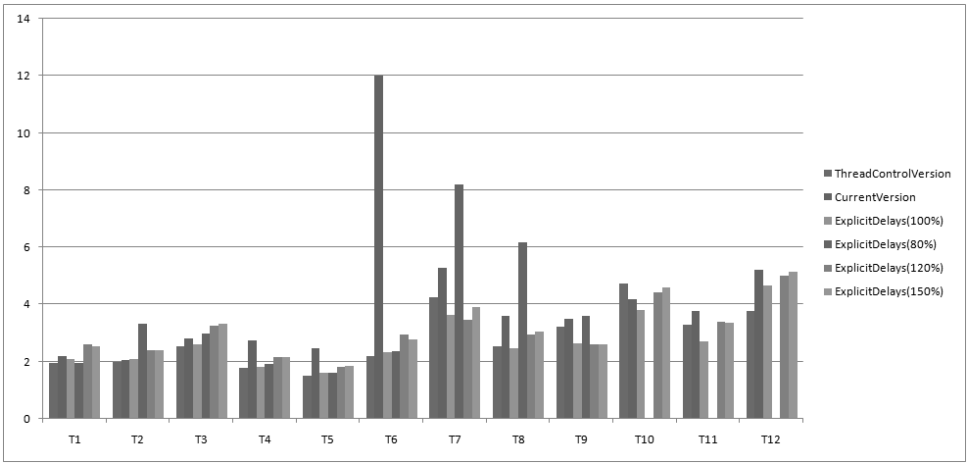


**Figure 3. Mean execution time for the tests (in seconds)**

Another conclusion from our experiments is that our approach was able to avoid the test failures from OurBackup tests that were not due to bugs. This fact was expected, according to the testing framework implementation, but the experiments gave us an idea of its use in practice. We could notice that while using other approaches these failures happen and their frequency would depend on the delay times chosen by the testers. This has also been observed in the development of OurGrid, in which developers tried to avoid the problem by just increasing the delay times used. One negative effect of such practice, besides making the test runs take longer, is that developers may abandon some of these tests or they may be rarely run, even though such tests can be the only ones that exercise the system in a given way.

Moreover, we have observed that the use of our testing framework has helped test developers with their work. By using this framework, we have spared developers from guessing the appropriate delay times to use. It has also led to test executions that took less time than the time required for the executions of the same tests using other approaches.

Although bringing such benefits, it is important to notice that the *Thread Control For Tests Approach* also presents some drawbacks. The main disadvantage is related to the fact that monitoring threads does affect thread interleavings. This may diminish the chances of a problematic interleaving to be executed. In order to avoid that, techniques and tools to generate different interleavings ([Stoller 2002, Copty and Ur 2005]) should also be used in addition to our approach. Another drawback of our approach is that it is not as easy to implement as explicit delays. However, we believe that this effort is worthwhile as some test failures due to the problems in the test will be avoided, leading to more reliable tests.

## 6. Related Work

In a previous work [Dantas et al. 2006], we have discussed how AOP has been used to help with the testing process of the OurGrid project. In that work, we proposed the `waitUntilWorkIsDone` operation in which there was only one system state configuration to wait for (e.g. all threads started by the test have finished or are waiting). In the approach *Thread Control For Tests*, we extend this initial idea by making the definition of other expected system states and also of other possible transitions. Besides, changes in the system state during assertions are also avoided.

The book by Meszaros [Meszaros 2007] is another related work. The author discusses how difficult it is to develop tests with asynchronous code and presents the problem of longer test runs due to waiting phases based on explicit delays. Besides presenting the problem, Meszaros mainly suggests that tests with asynchronous operations should be avoided at all costs in unit and component tests. We agree that at these levels, such tests should be avoided. Nevertheless, while testing the system as a whole, considering its most important operations, we cannot avoid tests with asynchronous operations. In cases like this, we must face the challenges posed by this kind of tests, using, for instance, approaches such as the one we propose here.

Works based on randomized scheduling [Sen 2007, Stoller 2002, Edelstein et al. 2002] during test executions may benefit from our work. This happens because they depend on reliable tests. While exercising different schedulings, it is important to avoid problems in tests that would lead to failures not caused by bugs.

The works by Copty and Ur [Copty and Ur 2005], Rocha et al. [Rocha et al. 2005] and MacDonald [MacDonald et al. 2005] investigate the suitability of AOP for implementing testing tools. In Copty and Ur's work [Copty and Ur 2005], AspectJ has been used to implement the Contest tool. This tool is used to force different thread schedulings during several re-executions of test suites in order to find concurrent problems. It could be used in addition to our approach to minimize the effect caused by monitoring on thread interleavings. Rocha et al. [Rocha et al. 2005] present J-Fut: a tool for functional testing of Java programs. This tool uses AOP to instrument and analyze the program under test. The use of the AOP technique for both tools showed to be suitable and has allowed a clear separation between the testing and the application code. This is something we have also noticed in our work. In [MacDonald et al. 2005], AOP is combined with another technology called CSSAME to run test cases in a deterministic fashion. It basically controls the program execution, allowing all paths of a race condition to be tested. In our work, AOP is used for controlling executions in order to avoid system changes while assertions are being performed.

Regarding threads monitoring, the work by Moon and Chang [Moon and Chang 2006] presents a thread monitoring system for multi-threaded Java programs, which is able of tracing and monitoring running threads and synchronization. One of the main differences between that work and ours is that it does not focus on testing, but mostly on debugging and profiling. Moreover, it uses a different technique for thread monitoring: code inlining. The authors argue that the use of the inlining approach is efficient for monitoring Java programs. We believe the use of this technique can also be explored in the future in our testing framework as an alternative to AOP.

The work by Pugh and Ayewah [Pugh and Ayewah 2007] has also some ideas similar to ours. Basically, this work describes the MultithreadedTC framework, which allows the construction of deterministic and repeatable unit tests for concurrent abstractions. The objective of the framework is to allow us to demonstrate that a code does provide specific concurrent functionality. In order to do that, it uses the idea of a clock that advances when all threads are blocked. This resembles the `waitUntilStateIsReached` operation of our framework. However, this work differs from ours in its focus on exercising a specific interleaving of threads in an application, which is not our focus. While using our approach, threads may run in different interleavings, but will achieve a common state in which assertions should be performed and should pass independently of the interleaving (unless there is a bug). Another difference between this work and ours is that, in order to use the MultithreadedTC framework, an explicit control of the threads behavior is necessary in the test code as the concurrency functionality is being analyzed. In our case, while using our framework, we do not need the thread instances in the test code, but only a way to refer to them (such as their class names) and to their expected state before assertions. Another similarity between this work and our framework is the fact that both borrow metaphors from JUnit [Massol 2004], making it easier the acceptance of the tool and benefiting from existing features of current IDEs (integrated development environments). This was illustrated in this paper by the `testSystemInitialization` test case shown in Section 4.

## 7. Conclusions and Future Work

We have presented in this paper an approach for determining when to make assertions while testing multi-threaded systems. We have also presented a testing framework to support this approach. We conclude from our evaluation that the approach proposed could be successfully used to determine the adequate moment to perform assertions and to avoid major changes in the system state while verifications are being performed. As a result, we could avoid the occurrence of test failures due to assertions being performed at inappropriate times. Furthermore, we believe this approach can reduce the overall execution time for test runs compared with other approaches currently used, such as explicit delays and busy waits. Using these approaches, the execution time would depend on intervals chosen by developers, which could either be appropriate or not, depending on the environment. On one hand, if a small interval is chosen, test failures can occur more often. On the other hand, if the interval is overestimated, this would lead to long test runs.

We also conclude that it is important to prevent failures caused by problems in the test from happening. This would avoid time wasting trying to find bugs in the wrong place or investigating bugs that do not exist. Besides, developers may also think that some rare failures are caused by a problem in the test, not in the application, even when this is the case. Another problem is that developers may exclude some important tests from the application test suite because these tests present the above mentioned problems, even when they are also able of finding bugs that only manifest themselves in some executions.

Such scenario with these problems may be too common because many developers tend to use the explicit delays approach. This happens because it is not always possible to include monitoring code in the application. However, techniques such AOP can make code monitoring possible in a transparent way. Besides, such monitoring code can be easily removed from the application when the system is deployed.

Another conclusion is that our approach also presents the advantage of avoiding the *miss of opportunity to fire* effect. This way, when a given state is being expected before assertions, threads that could mess up the test execution while assertions are being performed are not allowed to proceed.

In the future, in order to reinforce our current results, we plan to explore the *ThreadControl* testing framework in other systems. We want to measure how much test execution time can be saved using this framework and how many failures which are not caused by bugs could be avoided in those systems. Besides, we also plan to extend these ideas in tests that use distributed components that may run on different machines.

## 8. Acknowledgments

We would like to thank all previous and current members of the OurBackup and OurGrid teams for their patience in explaining their problems with tests and giving us more details about previous solution attempts being used. Special thanks go to Eduardo Colaço, Flávio Vinicius, Alexandro Soares, Paolo Victor and Marcelo Iury. We would also like to thank CNPq, the Brazilian research agency that has supported this work.

## References

Bertolino, A. (2007). Software testing research: Achievements, challenges, dreams. In *FOSE '07: 2007 Future of Software Engineering*, pages 85–103, Washington, DC, USA. IEEE Computer Society.

Choi, J. and Srinivasan, H. (1998). Deterministic replay of Java multithreaded applications. *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 48–59.

Cirne, W., Brasileiro, F., Andrade, N., Costa, L., Andrade, A., Novaes, R., and Mowbray, M. (2006). Labs of the World, Unite!!! *Journal of Grid Computing*, 4(3):225–246.

Copty, S. and Ur, S. (2005). Multi-threaded Testing with AOP is Easy, and It Finds Bugs. *Proc. 11th International Euro-Par Conference, LNCS*, 3648:740–749.

Dantas, A., Cirne, W., and Saikoski, K. (2006). Using AOP to Bring a Project Back in Shape: The OurGrid Case. *Journal of the Brazilian Computer Society (JBCS)*, 11:21–36.

Dijkstra, E. (1969). Notes on structured programming.

Edelstein, O., Farchi, E., Nir, Y., Ratsaby, G., and Ur, S. (2002). Multithreaded Java program test generation. *IBM Systems Journal*, 41(1):111–125.

Goetz, B. and Peierls, T. (2006). *Java concurrency in practice*. Addison-Wesley.

Gosling, J. (2000). *The Java Language Specification*. Addison-Wesley Professional.

Harrold, M. (2000). Testing: a roadmap. *Proceedings of the Conference on The Future of Software Engineering*, pages 61–72.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. (2001). Getting Started with AspectJ. *Communications of the ACM*, 44(10):59–65.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J. (1997). Aspect–Oriented Programming. In *European Conference on*

*Object–Oriented Programming, ECOOP'97*, LNCS 1241, pages 220–242, Finland. Springer–Verlag.

Lea, D. (2000). *Concurrent Programming in Java (tm): Design Principles and Patterns*. Addison-Wesley.

MacDonald, S., Chen, J., and Novillo, D. (2005). Choosing Among Alternative Futures. *Proc. Haifa Verification Conference, LNCS*, 3875:247–264.

Massol, V. (2004). *JUnit in Action*. Manning.

Meszaros, G. (2007). *XUnit Test Patterns: Refactoring Test Code*. Addison-Wesley.

Moon, S. and Chang, B. (2006). A thread monitoring system for multithreaded Java programs. *ACM SIGPLAN Notices*, 41(5):21–29.

Naik, M., Aiken, A., and Whaley, J. (2006). Effective static race detection for java. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 308–319, New York, NY, USA. ACM Press.

Oliveira, M. I. S. (2007). Ourbackup: Uma solução p2p de backup baseada em redes sociais. Master's thesis, Universidade Federal de Campina Grande.

Pugh, W. and Ayewah, N. (2007). Unit testing concurrent software. *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 513–516.

Rinard, M. (2001). Analysis of multithreaded programs. *Proceedings of the 8th International Symposium on Static Analysis*, pages 1–19.

Rocha, A., Simão, A., Maldonado, J., and Masiero, P. (2005). Uma ferramenta baseada em aspectos para o teste funcional de programas Java. In *Simpósio Brasileiro de Engenharia de Software, SBES 2005*, Uberlândia-MG.

Ronsse, M. and De Bosschere, K. (1999). RecPlay: a fully integrated practical record/replay system. *ACM Transactions on Computer Systems (TOCS)*, 17(2):133–152.

Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. (1997). Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411.

Sen, K. (2007). Effective random testing of concurrent programs. *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 323–332.

Stobie, K. (2005). Too darned big to test. *Queue*, 3(1):30–37.

Stoller, S. D. (2002). Testing concurrent Java programs using randomized scheduling. In *Proc. Second Workshop on Runtime Verification (RV)*, volume 70(4) of *Electronic Notes in Theoretical Computer Science*. Elsevier.

Sutter, H. and Larus, J. (2005). Software and the concurrency revolution. *Queue*, 3(7):54–62.

United Binary, L. Harnessit test framework: Multithreaded testing considerations. At http://unittesting.com/documentation/Appendices/MultiThreadedTesting.html.