

# ConTest Listeners: A Concurrency-Oriented Infrastructure for Java Test and Heal Tools <sup>\*</sup>

Yarden Nir-Buchbinder  
IBM Haifa Research Center  
yarden@il.ibm.com

Shmuel Ur  
IBM Haifa Research Center  
ur@il.ibm.com

## ABSTRACT

With the proliferation of the new multi-core personal computers, and the explosion of the usage of highly concurrent machine configuration, concurrent code moves from being written by the select few to the masses. As anyone who has written such code knows, there are many traps awaiting. This increases the need for good concurrency-aware tools to be used in the program quality cycle: monitoring, testing, debugging, and the emerging field of self-healing.

Academics who build such tools face two main difficulties; writing the instrumentation infrastructure for the tool, and integrating it into real user environments to obtain meaningful results. As these difficulties are hard to overcome, most academic tools do not make it past the toy stage.

The ConTest Listener architecture provides instrumentation and runtime engines to which writers of test and heal tools, especially concurrency-oriented, can easily plug their code. This paper describes this architecture, mainly from the point of view of a user intending to create a testing/healing tool. This architecture enables tool creators to focus on the concurrent problem they are trying to solve without writing the entire infrastructure. In addition, once they create a tool within the ConTest Listeners framework the tool can be used by the framework users with no additional work, enabling access to real industrial applications. We show how to create tools using the architecture and describe some work that has already taken place.

## Keywords

Java, concurrency, testing, self-healing, instrumentation

<sup>\*</sup>This work is partially supported by the European Community under the Information Society Technologies (IST) programme of the 6th FP for RTD - project SHADOWS contract IST-035157. The authors are solely responsible for the content of this paper. It does not represent the opinion of the European Community, and the European Community is not responsible for any use that might be made of data appearing therein.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SOQUA '07, September 3-4, 2007, Dubrovnik, Croatia

Copyright 2007 ACM ISBN 978-1-59593-724-7/07/09 ...\$5.00.

## 1. INTRODUCTION

As concurrent programming in Java becomes more and more widespread, there is a growing demand and supply of tools for monitoring, testing, debugging and, recently, self-healing concurrency, which is fueled by the notorious difficulty in finding and debugging concurrent bugs. These tools range from comprehensive industrial solutions to academic products that are often no more than specific ideas.

Many such tools face a common challenge: how to hook into the target program (i.e., the program being tested/monitored/debugged) while it runs, so that the tool can execute defined activities when the target program performs defined operations.

Scientifically, this is hardly a challenge: techniques for Java *instrumentation* have been known for many years, and implemented many times by many parties ([16, 5, 1] among many others). However, therein lies the problem; many parties have faced this *practical* challenge and been forced to implement their own solution. More often than not, this task is completely tangential to what is actually of interest for the tool developers; namely, how to deal with the concurrency, the bugs, or the performance. Furthermore, it requires completely different expertise. For example, source instrumentation usually requires working with syntax trees. Instrumentation of classes (which is often the preferred approach, for several reasons) requires knowledge of the Java bytecode language – not a common skill among Java developers. Then there are typical caveats and know-how, both to instrumentation and to working as an external layer with a target program – you’ll see many of them in this paper. So inventors and developers of testing tools have been forced to invest precious resources in implementing instrumentation, often “reinventing the wheel”, compromise the quality of their solution, or remain with a theoretical solution.

A second kind of challenge that tool developers face is *deployment*: they need to find validators who are willing to try the tool; then they need to convince them to install the tool, learn how to use it, and report the results back to the developers. It is not easy to find validators such as these, since industrial clients are usually reluctant to any change in their build process. Many academic tools settle for validation on small, often artificial examples.

We have an industrial concurrent testing tool, ConTest [9], which has its own instrumentation component. We recently made this component available in an open architecture for tool developers, dubbed *ConTest Listeners*. ConTest and the listener architecture are available from IBM alphaWorks (<http://www.alphaworks.ibm.com/tech/contest>). Section

2 of this paper gives a very brief description of ConTest, and our motivation in developing the open architecture. The architecture is oriented toward, although by no means limited to, concurrency-related tools. It enables tool developers to write only the code that they want to execute in response to target program's operations they are interested in. We relieve them of the burden of forcing this code to be invoked at the right time, or even of knowing anything about the instrumentation mechanism.

Tool developers can deliver their listener-based tool as an extension to ConTest. This greatly relieves the effort of finding validators: there is a ready user base, and no installation effort on the user part. From the users' point of view, they are just adding a new feature to ConTest, as opposed to installing a new point tool.

Section 3 is the main section of this paper. It gives a technical description of the architecture from the client's - i.e., the tool developer's - perspective; in other words, exactly what we provide, and what the tool developer should know. While the description is specific to our architecture, one can learn some general lessons from it; virtually all the tricky details and caveats we list apply to anybody who wishes to implement an instrumentor or write code that operates on an arbitrary target program. This highlights the need for such architecture: we spare many of these difficulties from the tool developers. Where we can't, we at least make sure the developer sees the caveat documented, with guidelines how to avoid it.

Section 4 compares our approach with the better known architecture in the field that may be used to achieve similar goals; namely, aspect-oriented programming (AOP) exemplified through AspectJ. While AOP is more general-purpose than our architecture, our architecture has advantages in the aspects of ease-of-use and multi-tool support. Section 5 describes work that other parties are doing with the architecture. Future work is discussed in section 6.

## 2. HISTORICAL CONTEXT

The ConTest project in IBM [9] started in IBM research in 2000 as a tool to help find concurrency-related bugs. The approach was to modify the application so that when a test executes, the interleaving forced by ConTest is likely to expose bugs. ConTest has no false alarms (or alarms of any kind), since it only causes different timing scenarios to happen in tests and tries to help the tests reveal the bugs hiding in the code. With time and increasing usage, many features were requested and added to the tool, such as code coverage, deadlock report and detection, and more. A notable feature added to ConTest is **synchronization coverage** [4], which helps to assess the quality of tests with regards to synchronization.

In 2006, an EU project called SHADOWS started, dealing with the area of self-healing. One of its goals was to identify concurrency problems at run time and fix them. The idea is that an oracle exists, either as the result of the test or by some other means, and the interleaving can be correlated to good or bad behavior. Then, at runtime, the interleavings corresponding to bad behavior are studied and eliminated. This research is done by a number of partners, making it desirable that they all use the same infrastructure while writing their own components. In particular, while ConTest can be used to generate the interleavings, we wanted to enable other parties to implement their own

heuristics for interleaving generation, using the ConTest infrastructure. Therefore, ConTest was redesigned so that the listener architecture could be extracted and made public.

As the development team of ConTest, we had an important asset to bring to the listener architecture, in addition to a ready instrumentor: knowledge, collected during several years of working with developers, of the various requirements and tricky details that occur in concurrent testing tools. In other words, we know what tool developers need, and what industrial users want, because we are tool developers ourselves, and have been working with industrial users for several years. This knowledge, too, was made public and explicit in the architecture design and documentation; some selected pieces of it are described in the next section.

ConTest Listeners are part of a long term plan to make concurrent testing tool research more accessible and form a community of testers. Toward this end, five years ago we started a yearly workshop, PADTAD, dealing with testing and debugging concurrent applications. In [12] [10], we described an architecture for collaboration between different testing tools and created a benchmark of concurrent programs with known bugs to evaluate the testing tools. An important component in such collaboration architecture is a common instrumentor, and the architecture presented here answers this need.

## 3. THE ARCHITECTURE'S INTERFACE

To help keep track of the explanation, we personalize the involved parties. A developer named Tooly is developing a Java testing tool called TestBest. Devvie is a developer or tester who works on an application called BizWiz. Devvie has some Java code of BizWiz - maybe just class files, if she's a tester - and some running tests. Devvie is a user of TestBest (hoping to deliver a high-quality BizWiz to the end user, named Miss Hughes). TestBest will be referred to as *the tool*, and BizWiz as the *target program*.

Tooly wants his tool to perform specific activities upon certain events of the target program. For example, whenever the target program code takes or releases locks or writes to variables, the tool needs to update internal data structures. Using ConTest Listeners, all Tooly needs to do is to write a class or a group of classes that implement the functionality to run when these events (or some of them) happen, and a little XML file to register his code with the ConTest Runtime Manager, which we provide. We also provide the instrumentation engine, which goes over Java classes (byte-code) and plants calls to the Runtime Manager. In principle, TestBest product is composed of two parts:

1. Our instrumentation engine
2. A group of classes comprising the runtime code (perhaps grouped together in `testbest.jar`), consisting of classes written by Tooly, the ConTest Runtime Manager, and the XML file.

To use TestBest, Devvie instruments her classes using the instrumentation engine, and then runs her tests, adding `testbest.jar` to the classpath. The instrumented code calls the Runtime Manager upon the appropriate events, and the Runtime Manager calls Tooly's classes. All this process is completely transparent to Devvie, until the stage where the tool does whatever it is that it's made to do.

This description was tagged "in principle", because our preferred method for Tooly to hand out his tool is as an *extension to ConTest*, which may be one of many extensions. He provides Devvie with a jar (consisting only of his own classes), and the XML file. Devvie obtains ConTest independently, and puts the XML in a predefined location. Part of ConTest is the aforementioned instrumentation engine, and the rest of the process for Devvie is as described above. This mode of operation resembles architectures working with plugins such as Eclipse [8] or Mozilla FireFox [13]. Tooly's extension operates alongside ConTest's runtime features, such as scheduling intervention, and alongside other extensions that Devvie installs; if, for some reason, this is harmful for Tooly's code, he can instruct Devvie to configure ConTest's features off, or even to run his extension alone, making it identical to the description above.

A special way to distribute an extension is as an Eclipse plugin [8]. ConTest itself can be obtained as a plugin; Eclipse has its own notion and mechanism of plugin extension. In our case the two notions of extension coincide – we harness the Eclipse mechanism so that a ConTest extension as described above can be defined as an extension to ConTest plugin. This is handy if Devvie develops the target program within Eclipse, or if the target program is itself a plugin and Devvie wants to use a ConTest extension to test it.

### 3.1 Listener Interfaces

Tooly writes his runtime functionality – the one to be triggered by target program's events – as class(es) implementing a set (or a subset) of interfaces we define – the listener interfaces. Below, we often refer to "the extension", or the "ConTest extension", meaning Tooly's whole product, or set of classes; "the listener code" is those methods in these classes that implement the listener interfaces. "Listener writers", or "extension writers", are what Tooly exemplifies.

One listener interface, for example, is `BeforeMonitorEnterListener`. It defines one method, `beforeMonitorEnterEvent` (`String codeLocation`, `Object lock`), which is invoked before each entry of the target program to a synchronized block or method. Tooly's implementation of this function is run by the target program's thread that does the synchronized block, just before taking the lock. This is true of all listener code; it always runs in target program threads. Similarly, there is an interface `AfterMonitorEnterListener`.

Other events that can be listened to include these:

1. before/after exit from synchronized blocks
2. before/after calls to `wait`, `notify`, `join`, `interrupt`, `Thread.start`
3. before/after calls to various methods of `java.util.concurrent` package
4. before/after reads and writes of nonlocal variables (static and object members). For instance, if there is `a=b+c`, and all three are members, we have (i) a read of `b`, (ii) a read of `c`, and (iii) a write of `a`. Before an invocation of `foo(b)` there is a read of `b`.
5. before/after reads and writes of array cells
6. start and end of threads (the code will be invoked by the thread itself that has started or is about to finish.

Compare with `Thread.start` listener, which is run by the parent thread.)

7. method entry (at the beginning of each instrumented method)
8. basic block entry – refers to bytecode basic blocks, which differ from source basic blocks in subtle points that are beyond the scope of this paper.

Note that the target program may subclass `Thread`, say by class `MyThread`. We then consider `MyThread.join` an instrumentation target, just like `Thread.join`, and similarly for other instrumentation target methods. However, if `MyThread` overrides `join`, then it is not an instrumentation target. One may expect that `MyThread.join` will have the same semantics as `Thread.join`, but this is up to the developer who wrote `MyThread`; she may have overridden `join` precisely to *avoid* actual joining, and in this case the join listeners should not be invoked. If her implementation *does* join a thread (possibly wrapping it in some other operations), it probably performs it by calling `super.join`, which is a call to `Thread.join` – it is this inner call that we should instrument. These distinctions are of challenge to the instrumentor.

As seen above, the method for monitor-enter gets as input the synchronized object and the code location of the synchronization. That is, the instrumentation infrastructure passes this information to the Runtime Manager, which passes it to the listener code. Similarly, other listeners get relevant context information of the target program. Listeners for method invocation get the parameter values of the method (e.g., wait timeout), and a reference to the invoking object. The listeners for read and write of variables get the variable name, a reference to the containing object, and the value read/written. Code location is given to relevant listeners (all the above except thread begin and end), and includes source file name and directory, method name, line number, and a number to differentiate multiple operations in the same line. For performance reasons this info is combined into one string.

Both the list of listenable events, and the context information given, result from the kinds of tasks we want to make possible to achieve with listeners, as presented in Section 5. They reflect the architecture's *concurrency-oriented* nature; the listenable methods are those strongly related to concurrency and to a thread's life cycle. Member access is listenable because members can be shared among threads and participate in data races (local variables cannot, so there is no listener for their access). Similarly for array cells. Basic block entry listener can be used to keep track of loops.

### 3.2 Contractual Aspects

**No completeness:** By default, listeners can't assume that they see *all* instances of the event to which they listen; Devvie controls what she instruments, and she may want to instrument only some of her classes. She also has options to exclude certain code pieces within a class from instrumentation, or to instrument only a given set of code locations. Such flexibility, however, may be unacceptable to some test/heal features and algorithms. If his extension assumes completeness, Tooly should explicitly instruct Devvie to instrument everything. See the Further Work section below about refining this requirement.

Many listeners form pairs of *before* and *after* the event, so that there are sequences of execution by threads: before-event-after. However, the *after* listeners are not invoked if the event itself threw an exception, e.g., a null pointer exception in a variable write. An exception to this rule is the listeners around method calls (*wait*, etc.); if their invocation throws an exception, the *after* listeners are invoked, and get a reference to this exception. After they complete running, the exception is thrown to the target program. These specifications were the result of a mix of considerations: what we could achieve in instrumentation, what seemed to be helpful information for tools, and what could be handled at runtime without too high a performance penalty.

**No atomicity:** The Runtime Manager adds no synchronization around listeners. Tooty should be aware that the sequence before-event-after is not atomic: context switches may happen at any stage, and other listeners (or the same ones!) can intervene, running in a different user's thread.

**Naughty listeners:** A listener-based tool typically runs in the background, doing activities like collecting information and writing to files. Obviously it can be expected to slow down the target program (therefore listener programs are usually destined to work at development, testing, and debugging stages, and not in production, although this may come about as well). Sometimes this slow-down is an integral part of its functionality, as is the case with noise-makers. The tool is free to intervene more aggressively: it may attempt to synchronize on target program locks, and even write to target program variables (this can be done by Java reflection API, using the field name and a reference to the target program object, both of which are given to the listener).

This may raise questions about interactions between different extensions; the before-member-write listener of TestBest may be told that value 2 is about to be written to some member, and then the same listener of another extension, say ThreadMill, may write 3 to this member, rendering TestBest's data false. From the architecture's point of view this is of little concern, since it is secondary anyway to ThreadMill's effect on the target program. We assume extension writers know what they are doing. Naturally, Tooty should make whatever effect the extension has on the target program's behavior clear to Devvie and relevant for the purpose of extension. Devvie can judge (or Tooty can warn her, if it's not clear) if it may conflict with other extensions. The general use case for extensions is that whatever the program does with the extension is legal behavior that could have happened (theoretically, if not practically) without the listener. Hence, generally listeners can coexist.

**Exceptions:** Here is an interesting case where an effect of a listener on the target program is unclear and irrelevant. The listener interface methods don't declare throwing an exception, so the listener code can't throw checked exceptions. Technically, it can throw unchecked exceptions (`RuntimeExceptions`), like any Java code, but this should be avoided. The reason is that it runs within the target code, but the target code is ignorant of it. Suppose the target code contains this:

```
String num = readInputFrom("in.txt");
try {
    intMember = Integer.parseInt(num);
} catch (NumberFormatException e) {
    System.out.println("problem in in.txt");
}
```

}

Now, suppose some listener code that is invoked after the write to `intMember` throws a `NumberFormatException` (say, due to some problem in a configuration file that the listener internally reads). This exception is caught by Devvie's code above, and leads to a faulty error message.

So we encourage extension writers not to throw any exception from listener code. Encouragement hardly suffices, however, as a runtime exception may percolate up from some third-party (or Java core) methods used by the listener code. If this happens, we prefer that the target program abort with an appropriate message rather than risk it catching the exception wrongly, or ignoring it (in this we adopt the *fail-fast* approach [15]). So the Runtime Manager catches any exception thrown by a listener, wraps it in an `Error`, and rethrows it. The target code may catch `Errors` as well, but Java discourages programmers from doing that – certainly from making programmatic assumptions about the origin of an `Error`. So this handling, while not completely fool-proof, leads to the desired behavior (target program's abnormal termination) if the target program adheres to this recommendation of Java.

**Target program beginning and termination:** The instrumentor plants calls to static methods of our class `Manager` in the target program. The static initializer of `Manager` bootstraps the Runtime Manager, including reading the extension registry XML files, and instantiation of the listeners. So listener instances, by default, are constructed upon the first hit of an instrumentation point. Note that the static initializer of a class is considered a method, and hence it's entry is an instrumentation point; this instrumentation point will necessarily be hit before any code of this class. If Devvie instruments the class of `main()`, then ConTest's bootstrapping and the listener's creation will be the first thing to run after Java's bootstrapping.

We could have been expected to define a listener for the JVM termination event, where the extension can do final result calculations, cleanup, etc. This is problematic, however, since the JVM may terminate abruptly, due to an external signal, with no time for final operations. The extension code can use the Java *shutdown hook* mechanism to perform final operations. It usually works, but the API of this mechanism states some limitations on its reliability upon abrupt termination. So extension writers should strive not to rely on it, and perform summary calculations and output at least on an occasional basis. In any case, the architecture can't be expected to give stronger guarantees than those given by the Java mechanism.

Rather than doing calculations on the fly during test runtime, a more common model is to *only log* necessary data, and do calculations on this data off-line, after the test has ended (say, using a program that Devvie is instructed to run after testing). In addition to being less sensitive to the above mentioned shutdown problem, this approach is usually more desirable as it has less performance impact on the test. One would typically prefer a very heavy calculation done off-line over a much smaller calculation done "at the expense of the test".

**Target object Safety:** Many listener methods get a reference to a target program's object as a parameter. Two caveats lurk for Tooty as a result. The first caveat appears when Tooty wishes to keep a collection of such objects or some information mapped to them. Typically he wants to

keep them for at least as long as they live (and might be seen again by his listeners). But he can't know when they are no longer alive, and so has to keep them indefinitely in his data structure. The problem then is that the Java garbage collector cannot collect them, and the listener clogs the JVM memory. Java contains a built-in solution for this problem: `WeakHashMap`. If Tooty puts target program objects in such a map, the information will exist as long as the object is referenced by the target program, and thereafter can be collected by the garbage collector (and disappear from the map).

An example to the second caveat lurks in the following innocent-looking implementation of a listener method:

```
beforeMonitorEnterEvent(
    String programLocation, Object lock) {
    System.out.println("took lock:" + lock);
}
```

When this code runs, it invokes `lock.toString()`. Now `lock` may be an instance of an instrumented target program class, and this class may have overridden `toString()`. So now the listener code runs instrumented code, causing more listener code to be invoked. This breaks the assumption that listener code is invoked only in response to actions done naturally by the target program. In the worst case it may lead to an infinite loop of listener calls.

Tooty can safely use *final* methods on target objects, such as `getClass()`. We provide some utility functionality for safe printing of object's identity, using only safe methods.

Similarly, it is dangerous for a listener to invoke `hashCode()` of a target program's object. But this method is indirectly invoked when storing a target program's object in a `HashMap` or a `HashSet` (or a `WeakHashMap`, as we've just recommended). To solve it, one can use `IdentityHashMap`, which uses the listener-safe `System.identityHashCode`. But in view of the previous caveat, one needs a map combining the features of both identity and weak keying. Java does not provide such a utility, so we wrote `WeakIdentityHashMap` and offer it to listener writers.

### 3.3 Other Test/Heal Utilities

The architecture provides some utilities that may be useful to test or heal tools:

- A **configuration file** - where extension writers (such as Tooty) are invited to add properties to be controlled by their user (Devvie), and a reader for this file.
- A uniform **screen output** mechanism - normally Tooty should refrain from sending output to standard output/error, since this output may intervene with the target program output and cause confusion. For example, the target program's output may be inspected automatically by the test environment to determine whether a test succeeded. If Tooty does need to output error or warning messages, we provide a uniform mechanism that mitigates the disadvantages; the output is clearly marked as coming from ConTest, and can be turned off by Devvie. There is one kind of output that we actually encourage Tooty to give (using this mechanism): at extension start up, Tooty should inform Devvie of configuration parameters (such as the extension's being active at all!) This is sometimes vital to verify that the settings are as intended. This mechanism can easily be extended to a yet more elaborate logging mechanism.

- A uniform **file output** mechanism - using it, each extension can have its own subdirectory in a common output directory of ConTest. Furthermore, the mechanism defines the notion of a **trace** - a file scoped to one test run. Tooty can define such a trace and print to it as a stream without worrying about file creation and closing, as the architecture manages it for him. All files belonging to a given test run, of all extensions, would get a uniform component in their name (a timestamp, or a Devvie-controlled identifier), thus enabling Devvie to combine information for a given test from all extensions. The architecture carefully handles buffering and flushing behavior of the files, which are tricky matters given the problem of JVM termination and the desire to have minor impact on the program under test.

- **Test reset** - many test features require the notion of a *single test*, or a *single program run*, well-defined in time, as a scope to some data collecting or processing, such that at each new test run, the data collection starts afresh. The extension writer can let the architecture manage it for the extension. By default, one invocation of the JVM is considered one test. However, there is a typical case where this is not the desired behavior: when testing a server application. Typically, a server runs an infinite loop, where, in each iteration, it gets a client request, processes it, and waits for the next request. In a typical test setup, the server JVM runs for many hours (conceptually, forever); each client request, or a small group of requests, is considered a separate test.

ConTest contains a mechanism for the user (Devvie) to say "a test has ended, now starts a new one". This can be done manually via the keyboard, programmatically by the target program calling some ConTest API, or (the most useful) by a message given to a network socket on which ConTest listens. Extensions can inherit this mechanism by implementing a special listener, `TestResetListener`, invoked when Devvie gives this message in any of the available ways. In this event, the extension code typically flushes some of its data structures. In addition, the file output mechanism responds to this event by closing the files scoped to the current test run and opening new ones.

- **Deferred activation** - Devvie can request that ConTest and its extensions don't start activity upon JVM startup but wait for some condition. One use case for this feature is when the tool is finding bugs at the target program's startup. These bugs, being exposed by the tool, cause the program to abort. But Devvie wants to continue testing and let the program reach more interesting places; she will fix the startup bugs later (or maybe she is a tester, and needs to continue testing before the developer fixes the early bugs). Currently, Devvie can specify this in the configuration file, in terms of a class name or a method name which, when encountered for the first time, triggers ConTest to start activity. Naturally, a much more elaborate specification mechanism can be defined if desired.

Usually Tooty doesn't need to take care of the deferred activation because the architecture manages it for him. The architecture does so simply by not invoking lis-

teners until the condition has occurred. After a test reset, the listeners are deactivated until the condition happens again. Tooley can request (using a flag in the XML registration file) that his listener be active from the start, despite Devvie’s request. One reason to do this is if his algorithm requires some sense of completeness; for instance, if it must know about *all* the locks taken so far by the program, without which it would not be correct.

- **Special random utility** - many test features need to make frequent random decisions. Java’s `Random` class is problematic for this purpose as it synchronizes on a lock at each request for a random value. This is required for good distribution guarantees in a multi-threaded context, but if done often, can have disastrous effects on performance. Furthermore, for test applications that may want to decide randomly whether to take a lock, this behavior of `Random` defeats the purpose. We provide a utility `QuickRandom` which avoids synchronization and thus solves these problems, at the price of somewhat weakened accuracy guarantees; the API provides a means of playing with the tradeoff. Another important feature of our random utility is its interaction with the **replay** property of the architecture; Devvie can request that the configuration of a given test run would be recorded and later replayed, usually for debugging. The most important recorded and replayed aspect is the random seed, guaranteeing that the same sequence of random choices made by all the listeners is followed, provided that they use the architecture’s replay.

#### 4. COMPARISON WITH AOP

AOP is another approach in which a programmer specifies activities to be done when certain kinds of operations are done by a target program. In that, it resembles our listener architecture. The best-known AOP tool for Java is AspectJ [2].

In AOP, the developer can specify pointcuts in which some action, called advice, is performed. A pointcut is a set of locations called join points. Whenever the program execution reaches a join point described in the pointcut, a piece of code associated with the pointcut – the advice – is executed. Using AOP, the programmer can decide where and when additional code should be executed. Then the classes are modified – either at compile time, or after compilation, or at classload. At run-time, when the program reaches the join point, the JVM activates the advice code.

When instrumenting the join points with the advice, one possibility is to put the code that needs to be executed directly there. The second option is to insert a call to a method that executes the relevant advice. It is usually more desirable for a variety of reasons to insert a call than to execute the code directly.

To use AOP, the tool writer must perform these steps:

1. Identify the places in the code to apply the advice.
2. Create a pointcut that points to all the correct join points.
3. Create an advice to apply at the join points.

4. Modify the project using the AOP tool. This is usually a change to the build process.

Our experience in [6] showed that for the purposes of concurrent testing, the second step may be difficult, as the language given by the tool to describe the pointcuts may not be sufficient to describe the desired join points. Specifically, there was no support for synchronization instrumentation, which is probably the most important instrumentation point for concurrency tools. However, we used the fact that AspectJ is open-source to add some of the missing components in [7].

One can think of our listener architecture as a special case of AOP, where all the possible relevant join points are instrumented in advance. A listener can subscribe to some of the join points. When a join point executes, all the listeners that subscribe to it are activated. The information available to the listeners is predefined by the architecture.

The main advantage of AOP over ConTest Listeners is that it is a more general-purpose tool. For example, assume that you want to change all I/O to be from a specific file and not from the input. With listeners this cannot be done as all the original program instructions are executed. With AOP this is straightforward. The difference reflects the fact that our architecture is oriented toward concurrency.

Another possible advantage of the AOP approach is when performance is of essence. With the listener architecture all points are instrumented, regardless of whether there is a listener listening on them. With AOP it is possible to weave only into the relevant locations. However, this will improve soon for the listener architecture – see Further Work below.

The greater power of AOP, however, comes with a price of complication. ConTest Listeners are simpler for the tool developer; To use AspectJ one has to learn a new language, whereas implementing listener interfaces is a common task for Java developers.

A more important disadvantage of AOP results from the fact that it’s typical use-case is a programmer writing aspects for her own code. This is evident in the case of applying multiple tools. Suppose three vendors *A*, *B* and *C* want to supply different race detectors. Each provides their own instrumentor. Now if Devvie wants to apply all three, she has a problem: if she instruments with tool *A* first, then with *B* and then with *C*, then *B* will operate not only on Devvie’s code but also on *A*’s code, and *C* would operate on *B*’s code as well. By contrast, ConTest Listener architecture was designed so that different extensions would work independently on a well-defined target code, with a single instrumentation action. As we discussed in section 2.2 (under “naughty listeners”), listeners *can* “step on each other’s toes”, but this is the exception rather than the rule, and if an extension is problematic in this sense it’s obvious for its writer.

#### 5. CURRENT AND UPCOMING USAGE

ConTest Listeners are sufficient for implementing the classic algorithms and tools of concurrent testing: race detection, deadlock detection, atomicity checking, noise making, basic interleaving replay [5], and high-level data race detection [1].

In fact, as the architecture emerged from ConTest most of ConTest runtime functionality has been refactored to be listener code. So ConTest itself uses the architecture for

heuristics-controlled noise-making, lock discipline checking [11], deadlock analysis, concurrent coverage and code coverage.

The first external users of ConTest Listeners were a group in the University of Brno. They have created an extension that implements race detection based on the Eraser algorithm [14] and then performs healing. Other groups are planning to use the architecture for high-level data race detection and atomicity checking.

A second current use for ConTest Listeners is healing concurrent bugs by changing the synchronization. For example, one way to heal a data race on a shared variable is by introducing an additional lock to ensure that a lock is held whenever this variable is accessed. Another way is to introduce delays that make "bad" interleavings less likely. Those and other options were implemented with listeners. Note that these healing actions can also introduce new bugs. For example, introducing a new lock for avoiding data races can cause a deadlock. Moreover, overhead by introduced synchronization can make the application performance significantly drop.

Another potential use of the ConTest Listeners is by a group in the Università di Milano Bicocca, who plan to use listeners to monitor all method entries and the value of the variables in these locations, and to follow *define-use* of variables. Another group in the UIUC is planning to use the architecture for high-level race detection and atomicity checking. We think that the architecture is especially attractive for academia researchers, and hope to have more extensions to add to our growing list.

## 6. FUTURE WORK

Our instrumentor is not very flexible in terms of the locations it instruments. While the user can specify instrumenting only a subset of the code locations, it currently cannot be done on a semantic basis. Generally, all types of events will be instrumented, even if there is no registered listener for them, with a resultant cost in performance. This would be easy to remedy in our architecture, and we plan to do it shortly; the instrumentation engine will read the XML files that register the listeners, and instrument only the events of types that have listeners registered.

Many test/heal tools, for reasons that have to do with performance but also correctness, would prefer to instrument only a subset of the locations, even of the event types on which they act. Therefore, we will allow listeners to specify a subset of instrumentation points: by type, e.g., thread start, by location, e.g., file and line numbers, or more specifically, e.g., accesses to a specific variable. Note that some of these options are available in aspect-oriented tools. Syntax will be defined to enable listeners to refine instrumentation, according to the options mentioned. An extension writer who decides to take advantage of this feature will "pay" for the improved performance with complexity of usage, so it will be in between our current offering and the standard AOP approach.

Another performance overhead we have is more challenging, in terms of software engineering, to reduce. We took a maximalist approach in deciding which context parameters to pass to the listeners; in particular, most parameters are passed to both the *before* and *after* listeners of an event, with identical values. Typically the listener would use only a subset of the parameters given to the method it implements,

and many parameters would not be used by any registered listener. The runtime penalty of producing these parameters and passing them would nevertheless be paid, and is probably significant. Furthermore, it is unlikely that any Java runtime compiler (JIT) would know to eliminate redundant parameters generation and passing. We can let each listener declare prior to instrumentation (say, in the XML file) which parameters it needs. The challenge is that to take advantage of this information, the listener interfaces themselves have to be changed per a given set of registered listeners. A possible approach is that we provide a tool that, after reading the required parameters, modifies the architecture's classes, eliminating redundant parameters.

Scott Stoller from UIUC has pointed to us in private communication an interesting possible performance improvement. If listener code wishes to store data scoped to target program's objects, we provide `WeakIdentityHashMap` as described in section 3. While it is quite fast (using identity hash code, rather than the object implementation of `hashCode`, it still may be expensive in performance for listeners that use it intensively. Stoller suggested that the instrumentation engine can add a member of a listener-defined data class to each instrumented class, so that the listener can assign and obtain this member directly. This would probably be quicker than using a map, and will spare interaction with the garbage collector.

We plan to change the architecture so that instrumentation is done during classload rather than as a static modification to the class files. One advantage is that it makes things simpler for Devvie; she no longer has the separate instrumentation step, and simply runs the tests with an additional JVM flag. But the really interesting advantage is that we can reload a class in mid-JVM run, and instrument it differently, achieving a similar effect to the work described in [3]. An example of a testing tool that need this feature is a coverage tool that for performance reasons, having measured that an instrumentation point was accessed, removes this instrumentation.

We also plan to supply a standard interface to the listeners that will make it possible to stop the program and show the user the status of various objects and threads.

Listeners are still limited in the way they can intervene; while they can go as far as changing values of target program members (through reflection), they can't change the original operations of the target program. Even for a method that have listeners, the listeners can't eliminate the method invocation, and can't change the method's parameters; similarly, synchronization listeners cannot eliminate the synchronization. All these are kinds of things that advanced tools may want to do. Interestingly, the ConTest infrastructure can perform these operations with little or no modification to the instrumentation engine, and we may provide listener writers with the API to do it. Note, however, that it will be a radical architectural change; if done uncarefully, it would lead to a similar problem as the one described above for AOP-based tools, of different extensions "stepping on each other's toes". But we believe that with careful design it can be done without breaking the architecture's basic premise that extensions can coexist. It will, however, no longer be a pure *listener* architecture.

## 7. REFERENCES

- [1] C. Artho, K. Havelund, and A. Biere. High-level data

- ances. In *VVEIS'03, The First International Workshop on Verification and Validation of Enterprise Information Systems*, April 2003.
- [2] AspectJ Team. The AspectJ programming guide. Version 1.5.3. Available from <http://eclipse.org/aspectj>, 2006.
- [3] J. Baker and W. Hsieh. Runtime aspect weaving through metaprogramming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 86–95, 2002.
- [4] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur. Applications of synchronization coverage. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 206–212. ACM SIGPLAN, June 2005.
- [5] J.-D. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 48–59, 1998.
- [6] S. Copty and S. Ur. Multi-threaded testing with aop is easy, and it finds bugs! In *Proc. 11th International Euro-Par Conference*, pages 740–749. LNCS, 2005.
- [7] S. Copty and S. Ur. Toward automatic concurrent debugging via minimal program mutant generation with aspectj. In *TV06*, 2006.
- [8] Eclipse Organization. Welcome to eclipse (plugin developer guide). <http://eclipse.org/documentation>, 2005.
- [9] O. Edelstein, E. Farchi, Y. Nir, G. Ratzaby, and S. Ur. Multithreaded java program test generation. *IBM Systems Journal*, 41(3):111–125, 2002.
- [10] Y. Eytani, K. Havelund, S. D. Stoller, and S. Ur. Towards a framework and a benchmark for testing tools for multi-threaded programs: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(3):267–279, 2007.
- [11] E. Farchi, Y. Nir-Buchbinder, and S. Ur. Cross-run lock discipline checker for java. In *Parallel and Distributed Systems: Testing and Debugging (PADTAD - 3)*, November 2005.
- [12] K. Havelund, S. D. Stoller, and S. Ur. Benchmark and framework for encouraging research on multi-threaded testing tools. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 286.1, 2003.
- [13] Mozilla FireFox. developer center. [http://developer.mozilla.org/en/docs/Main\\_Page](http://developer.mozilla.org/en/docs/Main_Page).
- [14] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.
- [15] J. Shore. Fail fast. *IEEE Software*, 21(5):21–25, Sept/Oct 2004.
- [16] S. D. Stoller. Model-checking multi-threaded distributed java programs. In *Proceedings of the 7th International SPIN Workshop on Model Checking*, pages 224–244, 2000.