A Concurrency Testing Tool and its Plug-ins for Dynamic Analysis and Runtime Healing

FIT BUT Technical Report Series

Bohuslav Křena, Zdeněk Letko, Tomáš Vojnar, Yarden Nir-Buchbinder, Rachel Tzoref-Brill, and Shmuel Ur

Technical Report No. FIT-TR-2009-01
Faculty of Information Technology, Brno University of Technology

Last modified: April 7, 2009

A Concurrency Testing Tool and its Plug-ins for Dynamic Analysis and Runtime Healing

Bohuslav Křena[1], Zdeněk Letko[1], Tomáš Vojnar[1], Yarden Nir-Buchbinder[2], Rachel Tzoref-Brill[2], and Shmuel Ur[2]

[1] FIT, Brno University of Technology, Božetěchova 2, 61266, Brno, Czech Republic,
e-mail: {iletko, krena, vojnar}@fit.vutbr.cz

[2] IBM, Haifa Research Lab, Haifa University Campus, Haifa, 31905, Israel,
e-mail: {yarden, rachelt, ur}@il.ibm.com

Abstract. This report presents a tool for concurrency testing (abbreviated as ConTest) and some of its extensions. The extensions (called *plug-ins* in this report) are implemented through the listener architecture of ConTest. Two plug-ins for runtime detection of common concurrent bugs are presented—the first (Eraser+) is able to detect data races while the second (AtomRace) is able to detect not only data races but also more general bugs caused by violation of atomicity presumptions. A third plug-in presented in this report is designed for hiding bugs that made it into the field so that when problems are detected they can be circumvented. Several experiments demonstrate the capabilities of these plug-ins.

1 Introduction

Concurrent programming is very popular nowadays despite its complexity and the fact that it is error-prone. The crucial problem of testing and debugging of concurrent programs is the huge number of possible execution interleavings and the fact that they are selected nondeterministically at runtime. The interleaving depends among other factors on the underlying hardware, with the result that concurrent bugs hide until used in a specific user configuration. Applications of model checking in this area are limited by the state space explosion problem, which is quite severe when considering large applications and their huge interleaving space, while static analysis tools suffer from many false alarms, and are also complicated by the need to analyse non-sequential code.

In this report, we consider testing and runtime analysis (capable of catching bugs even when they do not directly appear in the witnessed run) supported by techniques that make concurrent bugs appear with a higher probability. In particular, in Section 2, we present ConcurrentTesting (abbreviated as ConTest), a tool for testing, debugging, and measuring coverage of concurrent Java programs, on top of which specialised plug-ins with various functionalities can be built. In Section 3, we show two ConTest plug-ins for runtime detection of common concurrent bugs (data races and atomicity violations). In Section 4, we show a plug-in for healing bugs that escaped to the field. To evaluate these plug-ins, we performed several experiments in Sections 3 and 4.

2 ConTest: A Tool and Infrastructure for Concurrency Testing

ConTest [3] is an advanced tool for testing, debugging, and measuring coverage of concurrent Java programs. Its main goal is to expose concurrency-related bugs in parallel and distributed programs, using random noise injection. ConTest instruments the bytecode—either off-line or at runtime during class load—and injects calls to ConTest runtime functions at selected places. These functions sometimes try to cause a thread switch or a delay (generally referred to as *noise*). The selected places are those whose relative order among the threads can impact the result; such as entrances and exits from synchronised blocks, accesses to shared variables, and calls to various synchronisation primitives. Context switches and delays are attempted by calling methods such as `yield()` or `sleep()`. The decisions are random so that different interleavings are attempted at each run, which increases the probability that a concurrency bug will manifest. Heuristics are used to try to reveal typical bugs. No false alarms are reported because all interleavings that occur with ConTest are legal as far as the JVM rules are concerned. ConTest itself does not know that an error occurred. This is left to the user or the test framework to discern, exactly as they do without ConTest.

ConTest is implemented as a listener architecture [9], which enables writing other tools easily using the ConTest infrastructure. These tools are referred as *ConTest plug-ins*. Among the tools that can be written as ConTest plug-ins are those for concurrency testing, analysis, verification, and healing. The ConTest listener architecture provides an API for performing actions when some types of events happen in the program under test. The events that can be listened to include all events that ConTest instruments as described above. Each plug-in that extends ConTest defines to which event types it listens. ConTest can run any number of different plug-ins in a single execution.

A plug-in registers to ConTest through an XML mechanism. ConTest takes care of the instrumentation and of the invocation of the plug-in code for a specific event when this event occurs at runtime. ConTest also provides various utilities that can be useful when writing plug-ins. ConTest supports partial instrumentation, i.e., it can be instructed to include (or exclude) specific program locations in the instrumentation. This can be useful, for example, when concentrating on specific bug patterns. The `Noise` class provides noise injection utilities, such as the `makeNoise()` method that performs noise according to ConTest preferences, and more specific methods that perform noise of certain types and strengths. The noise injection utility is useful not only as it spares the developer the need to implement noise injection, but also because it takes care of risks that may arise when using more complicated types of noise. For example, if the noise is implemented by `sleep()` or `wait()`, these calls can take interrupts invoked by the target program, and this may interfere with the semantics of the program. The ConTest API takes care of this scenario. ConTest's own noise injection can be disabled if it is not required or interferes with the plug-in function. Additional utilities include methods for retrieving lock information, efficient random number generation, utilities for safe retrieval of target program threads and objects names and values in the code of plug-ins, and a hash map suitable for storing target program objects.

Eclipse is the most popular IDE for Java. Eclipse itself has an open architecture so that tools can be implemented as plug-ins and plug-ins can extend other plug-ins through extension points. ConTest is available both as a stand-alone tool and as an Eclipse plug-in. ConTest plug-ins can be implemented for both modes.

3 Plug-ins for Detecting Data Races and Atomicity Violations

This section describes two plug-ins we implemented as ConTest extensions for detecting common concurrency problems at runtime.

3.1 Eraser+ Plug-in for Data Race Detection

Our first plug-in uses a slightly enhanced version for the Java context of the well-known Eraser algorithm [10] for data race detection (denoted Eraser+). The plug-in registers to events *beforeAccess(v, loc)* and *afterAccess(v, loc)* generated by accesses to class fields *v* at program locations *loc* as well as to events *monitorExit(l)* and *monitorEnter(l)* generated by acquire/release operations on standard Java lock *l*. The detection of races is based on the consideration that every shared variable (detected by ConTest) should be protected by a lock. Since Eraser has no way to know which lock protects which variable, it deduces the protection relation during execution. For each shared variable *v*, Eraser identifies the set $C(v)$ of candidate locks. This set contains those locks that have protected *v* during the computation so far. Initially, $C(v)$ contains all locks. At each *beforeAccess(v, loc)* caused by a thread t, $C(v)$ is pruned by an intersection with the set of locks held by t. The set of locks currently held by a thread t is managed within the *monitorEnter(l)* and *monitorExit(l)* events. If $C(v)$ becomes empty, a race condition over *v* is reported.

To reduce false alarms and optimise the Eraser algorithm for Java, Eraser+ uses several improvements to the original algorithm described above. Firstly, we enhanced the original algorithm with the so-called *ownership model* [13, 12] which delays building of the set $C(v)$ of candidate locks to reflect the typical initialisation pattern used in Java programs where a variable is initialised by a different thread than the one which later uses the variable. We also added a support for the *join synchronisation*. Those enhancements were described in [5]. However, despite the implemented enhancements, Eraser+ can still produce false alarms, especially when synchronisation mechanisms other than Java basic locks or the join synchronisation are used.

3.2 AtomRace Plug-in for Data Race and Atomicity Violation Detection

Our second plug-in uses the AtomRace algorithm [6], invented with consideration of the needs of self-healing of programs (low overhead, no false alarms). AtomRace can detect not only data races but also atomicity violations. In fact, data races are viewed by AtomRace as a special case of atomicity violations. AtomRace does not track the use of any concrete synchronisation mechanisms; instead, it concentrates solely on the consequences of their absence or incorrect use. Thus, AtomRace can deal with programs that use any kind of synchronisation (even non-standard). AtomRace may miss data races or atomicity violations. On the other hand, it does not produce any false alarms.

The plug-in registers to events generated by accesses to class fields (i.e., *beforeAccess(v, loc)* and *afterAccess(v, loc)*) and by encountering method exit points (denoted *methodExit(loc)*). AtomRace detects data races by making each access to a shared variable *v* at a location *loc* a *primitive atomic section* delimited by *beforeAccess(v,loc)* and *afterAccess(v,loc)*. If an execution of such a primitive atomic section is interleaved by executing any other atomic section over *v*, and at least one of the accesses is for writing, a data race is reported. Of course, such primitive atomic sections are very short and the probability of spotting a race on them is very low. Therefore, we make the execution of these atomic sections longer by inserting some noise. In addition, AtomRace can deal with more general *atomic sections* when appropriate. For a shared variable *v*, it views an atomic section as a code fragment which is delimited by a single entry point and possibly several end points in the control flow graph. When a thread *t* starts executing the atomic section at some *beforeAccess(v,loc)*, no other thread should access *v* in a disallowed mode (read or write) before *t* reaches an end point of the atomic section at some *afterAccess(v,loc')* or *methodExit(v,loc')*. This way, AtomRace is able to detect atomicity violations. AtomRace can also detect *non-serialisable accesses* in the sense of [7] if atomic sections are defined over two subsequent accesses to the same variable.

When AtomRace deals with general atomic sections, it must be provided with their definition in advance, whether defined manually by the user or obtained automatically via static and/or dynamic analyses. We implemented a *pattern-based* static analysis that looks for typical programming constructions that programmers usually expect to be executed atomically. Occurrences of such patterns are detected in two steps. First, the PMD tool is used [2] to identify the lines of code where some critical patterns of using certain variables appear from the abstract syntax tree of the Java code under test. Then, FindBugs [1] analyses the ConTest-instrumented bytecode, and the occurrences of critical patterns detected by PMD are mapped to the variable and program location identifiers used by ConTest. Moreover, a dataflow framework implemented in FindBugs finds all possible execution paths in the control flow graph, starting from a concrete location denoting the start of an atomic section, and hence finds all possible exits of the section (including those related to exceptions). Further, another static analysis was implemented to support detection of non-serialisable accesses introduced in [7]. FindBugs obtains the initial set of *access interleaving (AI) invariants*. AtomRace then removes non-relevant AI invariants from the set during testing (we assume that invariants broken when a test passes successfully are not relevant).

As noted above, the atomic sections monitored by AtomRace may be too short to identify a conflict. However, we can profit from the ConTest noise injection mechanism to increase the length of their execution and hence to increase the probability of spotting a bug—to the extent that the detection becomes really useful according to our experiments. We implemented three injection schemes: First, the noise may be injected into the atomic sections randomly, when no a-priori knowledge on what to concentrate is available. Second, if we have already identified suspicious code sections or suspicious variables via previous analysis (e.g., using Eraser+ or static analysis), we may inject noise into the appropriate code sections or into sections related to the suspicious variables only. This way we significantly reduce the overhead and we may confirm that

```
public static void Service(int id, int sum) {
   accounts[id].Balance += sum;              // thread safe
   BankTotal += sum;                         // data race
}
```

Fig. 1. A problematic method in a bank account simulating program.

a suspicion raised by an algorithm such as Eraser+ is not a false alarm. The heuristics can also be used for replaying an already detected data race scenario.

3.3 Experiments

We evaluated the Eraser+ and AtomRace plug-ins[3] in four case studies, with the results listed in Table 1. Below, we first describe the case studies and explain the races that we identified in them. Then, we get back to some experimental evidence of how our algorithms found the problems. Finally, we discuss the influence of noise injection in more detail.

Case Studies. The first case study (*Bank*) simulates a simple program that maintains bank accounts while accessing the total bank balance by several threads without proper synchronisation. A bug is related to the global balance variable `BankTotal`, and the problematic method is depicted in Figure 1. The `Balance` variable is unique for each account thread, and hence, there is no race possible if a correct thread `id` is used as a parameter of the method. The `BankTotal` variable is shared among all threads, and there is a *load-and-store* bug pattern [5] on it (as the `+=` operation is broken to a sequence of three operations on the bytecode level, two threads may read the same value from `BankTotal`, modify it locally, and store the resulting value back to `BankTotal` while overwriting each other's result). The data race causes that the final balance may get wrong. The problematic method is called many times during the execution of the test case.

Web crawler is a part of an older version of a major IBM production software. The crawler creates a set of threads waiting for a connection. If a connection simulated by a testing environment is established, a worker thread serves it. The problematic method is shown in Figure 2. This method is called when the crawler is being shut down. If some worker thread is just serving a connection (`connection != null`), it is only notified not to serve any further connection. This notification is done within the `finish()` method depicted in Figure 2 by a thread performing the shutdown process. A problem occurs if the `connection` variable is set to `null` by a worker thread (a connection was served) between the check for `null` and an invocation of the `setStopFlag()` method. This case corresponds to the *test-and-use* bug pattern [5], and such a situation causes an unhandled `NullPointerException`. Contrary to the previous race example, this race shows up only very rarely.

FTP server is a development version of an open-source software, produced by Apache, mentioned in [4]. It contains several types of data races. The server works

[3] Available at `http://www.fit.vutbr.cz/research/groups/verifit/tools/racedetect/`.

```
public void finish() {
    if (connection != null) connection.setStopFlag();    // data race
    if (workerThread != null) workerThread.interrupt();
}
```

Fig. 2. A problematic method in the IBM web crawler program.

as follows. When a new client connects to the server, a new thread for serving the connection is constructed and enters the serving loop in the `run` method which is depicted in Figure 3. The `close` method, also depicted in Figure 3, can be run by another thread concurrently with the `run` method. When the `close` method is executed during processing of the `do-while` loop in the `run` method, the `m_request`, `m_writer`, `m_reader`, and `m_controlSocket` variables are set to `null` but remain still accessible from the `run` loop. This situation leads to an unhandled `NullPointerException` within the loop. The problem does not correspond to any of the patterns examined in [5]. In the test case, there are then present several further occurrences of the load-and-store bug pattern. However, none of them were considered as harmful because they influence only values of internal statistics.

TIDorbJ developed by Telefónica I+D is a CORBA 2.6 compliant ORB (Object Request Broker) which is published as open source software at the MORFEO Community Middleware Platform [11]. We, in particular, used the basic *echo concurrent* test shipped with *TIDOrbJ*. The test starts a server process for handling incoming requests and a client process which constructs several client threads, each sending several requests to the server. The server constructs several threads which serve the requests. If there are not enough server threads available, the client threads produce a timeout exception and retry later. Using this test, we identified some harmless data races in *TIDorbJ* as well as some races that led to a code modification upon our reporting them to Telefónica. The most harmful and interesting races are described in the following paragraphs.

The first data race that we identified in *TIDOrbJ* is depicted in Figure 4. In this case, the problematic variable `forwardReference` can be set to `null` by one thread in the `catch` branch while another thread is about to invoke a method on `forwardReference`. Such a situation causes an unhandled `NullPointerException`. This race is again very rarely manifested because it is yield by an exception produced within the `try–catch` block.

Another data race in *TIDOrbJ* has been identified in the `IIOPProfile` class on the variable `m_listen_point`. The variable is first tested for `null` and then set to a new value within a method defined as `synchronized`. However, since the test for `null` is out of the synchronized method (and not repeated inside the method), a slight variation of the test-and-use bug pattern from [5] occurs here.

Other data races in *TIDOrbJ* were classified as not harmful because they do not lead to an exception. An example of one of these data races can be an instance of the load-and-store bug pattern from the `IIOPCommLayer` class. The data race is on the `recover_count` variable which is decreased by one each time a catch block in the `sendRequest` method is executed due to an exception produced during sending a data element. When the `recover_count` variable reaches zero, the algorithm is not trying

```
public void run() {
  ...
  // initialise m_request, m_writer, m_reader, and m_controlSocket
  ...
  do {
    String commandLine = m_reader.readLine();
    if (commandLine == null)
      break;
    ...
    m_request.parse(commandLine);
    if (!hasPermission()) {
      m_writer.send(530, "permission", null);
      continue;
    }
    service(m_request, m_writer);
  } while (!m_isConnectionClosed);
}

public void close(){
  synchronized(this){
    if (m_isConnectionClosed)
      return;
    m_isConnectionClosed = true;
  }
  ...
  m_request = null;  m_controlSocket = null;  // still accessible from
  m_reader = null;   m_writer = null;         // the run() method above
}
```

Fig. 3. Problematic methods in the Apache FTP server.

to recover by resending the data. The data race can cause the recovery process to be executed more times than required by the value of the `recover_count` variable (hence the system does not fail, yet its performance may be lowered unnecessarily).

Results of Experiments. Let us now get back to Table 1 summarising results of multiple testing runs that we performed with both Eraser+ and AtomRace (five runs with approximately ten different noise settings for each case) on the described case studies. We see that Eraser+ produces false alarms while AtomRace does not. On the other hand, Eraser+ was able to detect data races even when the problem did not occur in a given execution. Interestingly, in repeated test runs, AtomRace managed to identify all bugs found by Eraser+, and more. The table also gives the time needed for one testing run of the applications without ConTest, with ConTest but without the plug-ins, and then the times with the Eraser+ and AtomRace plug-ins.

Our experiences with the detection abilities of the algorithms are as follows. Both algorithms were able to detect the problem in the simple *Bank* example. Eraser+ detects the violation in all executions, AtomRace in all executions where the problem occurs.

```
class IIOPCommunicationDelegate extends CommunicationDelegate{
   ...
   public void invoke(RequestImpl request) {
      try {
         if ( this.forwardReference == null ) {
            ...
         } else {
            this.forwardReference.invoke(request);
         }
      } catch (org.omg.CORBA.COMM_FAILURE cf) {
         this.forwardReference = null;
         throw cf;
      }
   }
}
```

Fig. 4. Another data race in TIDorb Java.

						Eraser+				AtomRace			
Example	Classes	kLOC	Time appl. only (sec.)	Time ConTest (sec.)	Data Races	Warnings	True Races	False Alarms	Time (sec.)	Warnings	True Races	False Alarms	Time (sec.)
Bank	3	0.1	1.005	1.007	1	1	1	0	1.009	1	1	0	1.008
Web crawler	19	1.2	3.01	3.02	1	1	0	1	3.04	1	1	0	3.03
FTP server	120	12	11.04	11.42	15	12	12	0	13.58	15	15	0	12.67
TIDOrbJ	1120	84	3.51	5.28	5	15	5	8	10.80	5	5	0	9.29

Table 1. Data races detected by Eraser+ and AtomRace plug-ins

In the Web crawler test case, Eraser+ detects a problem over the problematic variable but far before the problematic part of the code. This was caused by the fact that accesses to the problematic variable were synchronised by the *wait-notify* mechanism which is unsupported by Eraser+. Therefore, the warning is considered as a false alarm. AtomRace again detects violations in all executions where the problem occurs.

In the more complicated *FTP server* test case, Eraser+ missed three data races. Among the missed data races was, for example, the race on the `m_isConnectionClosed` variable shown in Figure 3. The data races were missed because of the Eraser+ owner-transfer enhancement [5, 13, 12]. This enhancement causes that Eraser+ produces less false alarms but under some circumstances described in [5, 12], it can miss the first occurrence of a problem. When we turned off this feature, Eraser+ found all three missing data races and produced five another false alarms. AtomRace again works perfectly.

Finally, in the *TIDOrbJ* test case, the Eraser+ algorithm produces 15 warnings. Out of them, 5 were classified as true races, 8 were classified as false alarms, and 2 are not yet known to be real or false alarms. AtomRace announced just the 5 real data races.

Noise Injection. Both detection algorithms search for data races along the given execution path. The introduction of noise into the execution can enforce different and still legal thread interleavings and so different execution paths. This way, ConTest noise injection heuristics can increase detection efficiency during multiple executions of the same test. This works for both algorithms as can be seen from Table 2. The table shows average numbers of true data races reported during one execution of a test case (out of 100 executions). Columns *CT 100* (*CT 200*) show results for test cases when the ConTest noise injection was activated, and noise was inserted into particular locations of the bytecode with probability of 0.1 (0.2), respectively. Columns *ARV 100* (*ARV 200*) show results for the AtomRace variable based noise injection where noise was inserted into particular (primitive) atomic section with probability of 0.1 (0.2), respectively. As can be seen, the efficiency of Eraser+ which detects even data races which do not manifest during the execution increases only a little. In the case of AtomRace, the efficiency increases to values obtained by Eraser+ and beyond—still without any false alarms.

	Data	Eraser+			AtomRace				
	races	no noise	CT 100	CT 200	no noise	CT 100	CT 200	ARV 100	ARV 200
Bank	1	1	1	1	0.39	0.99	1	1	1
Web Crawler	1	0	0	0	0	0.01	0.03	0.04	0.04
FTP server	15	5.70	5.88	6.05	3.40	5.79	5.27	5.94	6.00
TIDOrbJ	5	1.80	1.96	2.23	0.37	2.38	2.39	3.63	3.82

Table 2. Noise injection influence on the detection algorithms.

A Brief Summary. To sum up, the Eraser+ algorithm is able to detect data races even in many executions where the problem does not occur because, in fact, it does not detect data races but violations in a locking policy. On the other hand, it produces false alarms, and it is problematic to suppress these warnings without avoiding false negatives.

On the other hand, AtomRace does not suffer from false alarms but only very rarely reports data races which do not occur during the execution (e.g., when the thread is interleaved between *beforeAccess(v)* and the immediate access to *v*). Hence, to give useful results, AtomRace needs to see more different interleavings than Eraser+. However, as our experiments indicate, this can be achieved via using suitable heuristics. In the end, judging from our experiments, AtomRace seems to be able to in practice detect all bugs detected by Eraser+, and sometimes even more. Moreover, AtomRace is able to detect atomicity violations which could not be detected by Eraser+.

4 Plug-in for Bug Healing at Runtime

When data races or atomicity violations are detected during software development, they can be corrected manually by a programmer. Some bugs, however, may (and often really do) remain in an application even when it is deployed. This motivates another ConTest plug-in that we developed and that is able to heal such bugs automatically at runtime [5]. The plug-in cannot remove bugs from the code but it can prevent their manifestation.

Note that the healing techniques discussed in this section focus on healing bugs that may be classified as occurrences of the *test-and-use* and *load-and-store* bug patterns described as typical patterns of errors in atomicity in [5]. Other bugs than those corresponding to instances of these patterns cannot be healed automatically by our plug-in. An example of such a bug is, e.g., the problem detected in the *FTP server* test case depicted in Figure 3. Fixing (or healing) such problems is not trivial, and to make it acceptable, a significant amount of information on the designer's intent may be needed as was discussed, e.g., in [4].

Our first method of self-healing is based on *affecting the scheduler*. The scheduler is affected during execution of the *beforeAccess(v, loc)* listener for a problematic variable *v*, where *loc* is the beginning of an atomic section defined over *v*. The scheduling may, for example, be affected by injecting a `yield()` call that causes the running thread to loose its time slice; but the next time it runs, it has an entire time slice to run through the critical code. Another similar approach is to increase the priority of the critical thread. Such healing approaches, of course, do not guarantee that a bug is always healed, but at least they significantly decrease the probability of a manifestation of the bug. It is also evident that this healing technique cannot be used when the atomic section is long. On the other hand, such a healing is safe (i.e., it cannot introduce a new bug) as it does not change the semantics of the application.

Our second self-healing method injects *additional healing locks* to the application. A healing lock for a variable *v* is acquired at *beforeAccess(v, loc)* whenever *loc* is a starting point of any atomic section related to *v* and then released at *afterAccess(v, loc)* or *methodExit(v, loc)* at *loc* corresponding to the end point of the entered atomic section. This approach guarantees that the detected problem cannot manifest anymore. However, introducing a new lock can lead to a deadlock, which can be even more dangerous for the application than the original problem. Moreover, frequent locking can cause a significant performance drop in some cases. However, one can consider either using some light-weight static analysis showing that adding locks is safe (as there is obviously no danger of nested locking, which is often the case) and/or consider combining the healing with a deadlock avoidance method as suggested in [8].

Both of the above described methods heal atomicity violations and data races after their first occurrence. However, one can go a step further and heal even violations which are just about to happen for the first time. This can be achieved by tracking executions of atomic sections and blocking all threads which are about to access the problematic variable *v* when some thread is already executing an atomic section related to *v*.

4.1 Experiments

We evaluated the healing plug-in on the same case studies as the detection plug-ins. The healing efficiency was tested using assertions (oracles) introduced into the original code of the test cases. These assertions allow one to detect whether the known bug manifested, e.g., if a `NullPointerException` was thrown within the problematic block of code. Manifestation of the bugs depends on timing and the used hardware architecture. Therefore, all tests have been done on several architectures which vary in the number of available processor cores: one core Intel Pentium 4 2,8 GHz, two cores Intel Core 2 Duo E8400, four cores 2xAMD Opteron 2220, and eight cores 2xIntel Xeon 5355. Our

Proc	Orig	Yield	Prio	YiPrio	OTYield	OTWait	NewMut
1	0.908	0.734	0.821	0.735	0.711	0.598	0
2	0.297	0.094	0.705	0.444	0.068	0.041	0
4	0.545	0.673	0.648	0.658	0.415	0.242	0
8	0.710	0.681	0.783	0.755	0.651	0.573	0

Table 3. Efficiency of healing techniques in the *Bank* test case.

experiences show that data races described in the previous section can be divided into two groups of problems from the healing point of view.

Frequently Manifesting Bugs. The first group includes data races and atomicity violations which occur often during an execution. As an example of such a data race, the bank account test case shown in Figure 1 can be used. We found a similar situation also in *FTP server* (in a module responsible for gathering server statistics) and *TIDOrbJ* (in the exception handling block shown in Figure 4).

In the *Bank* test case, the problematic piece of code is called during each operation with accounts. The healing efficiency for this test case on computers with a different number of cores is shown in Table 3. In particular, the results were obtained for 8 account threads doing 10 account operations each, without any computation in between these operations. The first column of the table shows the number of cores. The other columns of the table describe the percentage of runs in which a problem is manifested (i.e., `Bank_Total` ends up with a wrong value) out of 6000 executions of the test for a particular setting. The second column of the table shows how often the bug manifests without any healing. The *Yield* column refers to calling `Thread.yield()` when entering a problematic atomic section, the *Prio* column refers to increasing the priority of the thread entering such a section, and the *YiPrio* column refers to a combination of both of these techniques. The *OTYiyeld* (*OTWait*) columns refer to calling `Thread.yield()` (or `Thread.wait(0, 10)`, respectively) if a thread is inside a critical section when another thread wants to enter it. The *NewMut* column shows the results of adding healing locks.

As can be seen from the table, the race manifestation probability highly depends on the used configuration. In most cases, healing by affecting the scheduler cannot be effectively used for suppressing such races. However, results given in the second row show that some methods (e.g., `Yield`, `OTYiled`, and `OTWait`) on some configurations can help. Some other methods (e.g., `Prio`) can, on the other hand, make the problem even worse. In general, it can be said that methods based on influencing the scheduler are not suitable for healing very frequently occurring bugs. On the other hand, such bugs should be easy to find during development, e.g., by testing. The last column of Table 3 shows that only additional synchronisation can heal the problem in a satisfactory way.

Rarely Manifesting Bugs. The second group of data races and atomicity violations are those which occur only very rarely. Such bugs depend on a very special timing among the involved threads. As an example of this kind of scenario, the web crawler test case

Proc	Orig	Yield	Prio	YiPrio	OTYield	OTWait	NewMut
4	0.0195	0.0018	0.0018	0.0023	0.0003	0	0
8	0.0194	0.0022	0.0035	0.0035	0.0002	0	0

Table 4. Efficiency of healing the web crawler.

shown in Figure 2 can be used. Table 4 shows results of applying our healing plug-in in this test case when only 30 worker threads were used, and so there were only 30 possible manifestations of the bug in one execution. In this case, healing techniques which influence the scheduler can be successfully used for avoiding the bug as can be seen from Table 4.

Table 4 shows the percentage of runs in which a problem manifested (meaning that a `NullPointerException` occurred in the problematic piece of code) out of 6000 executions of the test for a particular setting. Rows for one- and two-core systems are omitted because the problem did not manifest in any of the test executions we did at these configurations. It can be seen that the techniques which influence threads that are about to access a variable when some other thread is inside an atomic section defined for this variable (`OTYiedl` and `OTWait`) provide a better healing efficiency. Of course, additional synchronisation again suppresses the bug completely.

5 Conclusions and Future Work

We presented ConTest, a tool and infrastructure for testing concurrent programs, and its plug-ins for detecting and healing data races and atomicity violations. In the future, we plan to improve the efficiency of the methods (to decrease the overhead and increase the ratio of bug finding), improve the static analyses we use (e.g., for detecting occurrences of bug patterns), and support detection and healing of errors based on more different bug patterns than so far, including more involved bug patterns like, e.g., errors in synchronisation which involve more variables.

Acknowledgements

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 report. 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. This work is partially supported by the Czech Ministry of Education, Youth, and Sport under the project *Security-Oriented Research in Information Technology*, contract CEZ MSM 0021630528, and by Czech Science Foundation under the contract GA102/07/0322.

References

1. N. Ayewah, W. Pugh, D. Morgenthaler, J. Penix, and Y. Zhou. Using FindBugs on Production Software. In *Proc. of OOPSLA'07*. ACM, 2007.
2. T. Copeland. *PMD Applied*. Centennial Books, 2005.
3. O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java Program Test Generation. *IBM Systems Journal*, 41(1):111–125, 2002.
4. M.E. Keremoglu, S. Tasiran, and T. Elmas. A Classification of Concurrency Bugs in Java Benchmarks by Developer Intent. In *Proc. of PADTAD'06*. ACM, 2006.
5. B. Křena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar. Healing Data Races On-The-Fly. In *Proc. of PADTAD'07*. ACM, 2007.
6. B. Křena, Z. Letko, and T. Vojnar. AtomRace: Data Race and Atomicity Violation Detector and Healer. In *Proc. of PADTAD'08*. ACM, 2008.
7. S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *Proc. of ASPLOS-XII*. ACM, 2006.
8. Y. Nir-Buchbinder, R. Tzoref, and S. Ur. Deadlocks: from Exhibiting to Healing. In *Proc. of RV'08,* volume 5289 of *LNCS*. Springer, 2008.
9. Y. Nir-Buchbinder and S. Ur. ConTest Listeners: A Concurrency-oriented Infrastructure for Java Test and Heal Tools. In *Proc. of SOQUA'07*. ACM, 2007.
10. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multi-threaded Programs. In *Proc. of SOSP'97*. ACM, 1997.
11. J. Soriano, M. Jimenez, J. Cantera, and J. Hierro. Delivering Mobile Enterprise Services on Morfeo's MC Open Source Platform. In *Proc. of MDM'06*. IEEE, 2006.
12. J. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and Precise Data Race Detection for Multithreaded Object-Oriented Programs, 2002.
13. C. Flanagan and S. N. Freund. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. In *Proc. of POPL'04*, 2004. ACM Press.