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

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

[1] FIT, Brno University of Technology, Božetěchova 2, 61266, Brno, Czech Republic, e-mail: {krena, iletko, 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 paper presents a tool for concurrency testing (abbreviated as ConTest) and some of its extensions. The extensions (called *plug-ins* in this paper) 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 paper is designed to hide 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 encountered when testing and debugging 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 they manifest 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 paper, we consider testing and runtime analysis (capable of catching bugs even when they do not appear directly 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 test coverage for concurrent Java programs, on top of which specialised plug-ins with various functionalities can be built. In Section 3, we describe two ConTest plug-ins for runtime detection of common concurrent bugs (data races and atomicity violations). In Section 4, we describe a plug-in for healing bugs that escaped to the field. To evaluate these plug-ins, we performed several experiments. Due to space limitations, we only provide a short description of the experiments in Sections 3 and 4.

2 The ConTest Tool and Infrastructure

ConTest [4] is an advanced tool for testing, debugging, and measuring test coverage for 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()`, `sleep()`, or `wait()`. 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 [12], 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 because 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 a 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. As an Eclipse plug-in, it defines an extension point, which allows ConTest's own plug-ins to be easily made into Eclipse plug-ins themselves.

3 Plug-ins for Detecting Synchronisation Errors

This section describes two plug-ins we implemented as ConTest extensions for detecting common concurrency problems at runtime. Namely, the plug-ins are targeted at detecting data races and atomicity violations.

3.1 The 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 [13] 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 lock l. The detection of races is based on the consideration that every shared variable (detected by ConTest at runtime) 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 refined by intersecting it 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 algorithm for Java, Eraser+ uses several improvements to the original algorithm as described in [7]. However, despite the implemented support of join synchronisation and variable initialisation by another thread, Eraser+ can still produce false alarms, especially when synchronisation mechanisms other than basic Java locks or the join synchronisation are used.

3.2 AtomRace: Detecting Data Races and Atomicity Violations

Our second plug-in uses the AtomRace algorithm [8], invented in consideration of the needs (low overhead, no false alarms) of self-healing programs. 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 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 that 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 [9] 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 [3] to identify the lines of code where critical patterns that use 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 [9]. 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).

```
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 program simulating bank accounts

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 particularly 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 may confirm that a suspicion raised by an algorithm such as Eraser+ is not a false alarm.

3.3 Experiments

We evaluated the Eraser+ and AtomRace plug-ins[3] on the four case studies listed in Table 1, with the results listed in Table 2. Below, we first describe the case studies and explain the races that we identified in them. Then, we provide 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 is a program that simulates a *simple bank system* in which the total bank balance is accessed by several threads without a proper synchronisation. The 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 thread simulating an account, 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 occurs a bug following the *load-and-store* bug pattern [7] on it. To see this, note that the `+=` operation is broken into a sequence of three operations on the bytecode level. Thus, 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 the final balance to possibly be wrong. The problematic method is called many times during execution of the test case.

Our second case study is the *web crawler*, which 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,

[3] `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

a worker thread serves it. The method which causes problems in this case 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 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 represents an occurrence of the *test-and-use* bug pattern [7], and such a situation causes an unhandled `NullPointerException`. Contrary to the previous race example, this race shows up only very rarely.

The third case study is a development version of an open-source *FTP server* produced by Apache and mentioned in [6]. It contains several types of data races. The server works 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 still remain accessible from the `run` loop. This situation leads to an unhandled `NullPointerException` within the loop. The problem corresponds to the repeated test-and-use bug pattern mentioned in [7], but, in this case, more than one variable is involved. In the program there are also present several further occurrences of the load-and-store bug pattern. However, none of them were considered as harmful because they only influence values of internal statistics values.

Our fourth and final case study is *TIDorbJ* developed by Telefónica I+D. It is a CORBA 2.6 compliant ORB (Object Request Broker), which is available as open source software running on the MORFEO Community Middleware Platform [14]. In particular, we used the basic *echo concurrent* test shipped with *TIDOrbJ*. The test starts a server process for handling incoming requests and a client process that constructs several client threads, each sending several requests to the server. The server constructs several threads that 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

```
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

thread in the `catch` branch while another thread is about to invoke a method on `forwardReference`. An unhandled `NullPointerException` is caused if such a situation occurs. This race is very rarely manifest, because it is yielded by an exception produced within the `try`–`catch` block.

Another data race in *TIDOrbJ* has then been identified in the `IIOPProfile` class on the variable `m_listen_point`. The variable is first tested for being `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), an instance of the test-and-use bug pattern from [7] appears here.

Finally, several other data races that we identified in *TIDOrbJ* were classified as not harmful since they do not lead to an exception. An example of these data races is an instance of the load-and-store bug pattern from the `IIOPCommLayer` class. The data race is on the `recover_count` variable, which decreases by one each time a catch block in the `sendRequest` method executes due to an exception produced while sending a data element. When the `recover_count` variable reaches zero, the algorithm does not try to recover by resending the data. The

```
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. A data race in TIDorb Java

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).

Table 1 gives some more numerical data about the case studies under consideration. In particular, the numbers of classes and lines of code that the case studies consist of are given in columns two and three of the table. Then, the table also gives the number of monitored atomic sections for each case study. Finally, the last two columns give the average number of threads that arose during the tests as well as the average number of monitored instances of shared variables for each case study.

Results of Experiments. Table 2 summarises results of the tests that we performed with both Eraser+ and AtomRace. In particular, we performed five testing runs with approximately ten different noise settings for each case study. Of course, more runs with different input values and different places where some noise is injected could discover more bugs, but we chose to stay with constant input values and a relatively small number of executions. The results were obtained under Sun's Java version 1.5 on a machine with 2 AMD Opteron 2220 processors at 2.8 GHz.

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 test run of the applications without ConTest, with ConTest but without the plug-ins, and with the Eraser+ and AtomRace plug-ins. The column marked as *Data Races* presents the number of data races we found in the test cases during all tests presented in this paper (some of them were already known, some not).

Table 1. Case studies on which the Eraser+ and AtomRace plug-ins have been tested

Example	Classes	kLOC	atom. sect.	threads	inst. of shared vars.
Bank	3	0.1	2	9	28
Web crawler	19	1.2	8	33	320
FTP server	120	12	14	304	23 123
TIDOrbJ	1120	84	310	49	438 014

Table 2. Data races detected by the Eraser+ and AtomRace plug-ins (Note: Two of the Eraser+ warnings on TIDOrbJ have not yet been classified as true or false errors.)

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

Our further experimental results given in Table 3 illustrate the impact of noise injection, which enforces different and still legal thread interleavings, on both of the considered algorithms. The table shows average numbers of true data races reported during one execution of a test case (out of 100 executions) on a machine with 2 AMD Opteron 2220 processors at 2.8 GHz. The columns *CT 100* and *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 and 0.2, respectively. The columns *ARV 100* and *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 and 0.2, respectively. As can be seen, the efficiency of Eraser+—which even detects data races 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. Again, one can increase the amount of injected noise but then the number of detected races increases only a little or even decreases. A higher amount of injected noise can also cause a different code coverage, e.g., when a server application is not able to serve all incoming requests, the code responsible for solving such a situation is executed and examined by the detection algorithm.

A Brief Summary. To sum up the abilities of data race and atomicity violation detection plug-ins, we can say that 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.

Table 3. The influence of noise injection on Eraser+ and AtomRace

	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

On the other hand, AtomRace does not suffer from false alarms but only very rarely reports data races that 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 using suitable noise injection heuristics. In the end, judging from our experiments, in practice, AtomRace seems to be able to detect all bugs detected by Eraser+, and sometimes even more. Moreover, AtomRace is able to detect atomicity violations that 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 [7]. This healing plug-in cannot remove bugs from the code but it can prevent the code from failing on them.

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 [7]. 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 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 regarding the designer's intent may be needed as was discussed, e.g., in [6].

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 (alternatively, `wait()` or `sleep()` with a minimum or zero waiting/sleeping time) that causes the running thread to lose 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. Yet another approach is to inject `yield()` or `wait()` to a thread that is trying to enter a critical section in which there is already another process. 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. 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 [11].

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 that vary in the number of available processor cores. We used a computer with (1) one core based on Intel Pentium 4 2,8 GHz (with hyper-threading), (2) two cores based on Intel Core 2 Duo E8400, (3) four cores based on two AMD Opteron 2220, and (4) eight cores based on two Intel Xeon 5355. Our 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 that occur often during an execution. An example of such a data race is the bank account test case shown in Figure 1. We found a similar situation also in the *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 4. The results were obtained for eight account threads doing ten 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 ratio of runs in which a problem gets manifested (i.e., `Bank_Total` ends up with a wrong value)

Table 4. Efficiency of healing techniques in the *Bank* test case

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

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 is 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 probability of a race manifestation 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 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 4 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 that occur only very rarely. Such bugs depend on a very special timing among the involved threads. An example of this kind of scenario is the web crawler test case shown in Figure 2. Table 5 shows the 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 that influence the scheduler can be successfully used to avoid the bug, as shown in Table 5.

Table 5 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. It can be seen that the techniques that 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 than techniques influencing threads that enter the problematic section first. Of course, additional synchronisation again suppresses the bug completely.

Table 5. Efficiency of healing the web crawler

Proc	Orig	Yield	Prio	YiPrio	OTYield	OTWait	NewMut
1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0
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

5 Related Work

For monitoring and influencing a concurrent Java program, one could consider using AspectJ as an alternative to ConTest. However, as discussed in [12], ConTest differs from aspect-oriented programming in making a clear separation between a target program and tools, so that different tools can all target the same target program without interfering with one another. Moreover, ConTest works on the bytecode level and comes with various ready-to-use methods for noise injection, test coverage measurement, etc. Another alternative that one could think of is the JVM Tool Interface (JVMTI), which is a popular C programming interface provided by JVM implementations in order to write monitoring and development tools that inspect and control the execution of the applications running in the JVM. Despite providing a rich interface, JVMTI lacks some of the interface methods that are important for developing concurrency bugs detection tools. For example, it does not provide a method for the event where a thread takes a lock (it supports only the event of a contended lock when the thread has to wait since another thread is holding the lock).

On top of ConTest, various other algorithms for dynamic detection of concurrency-related bugs than Eraser+ or AtomRace could be implemented, such as [2, 15, 5]. Such dynamic analysis algorithms differ in their detection power (ability to warn about unseen bugs), number of false alarms, and their overhead—a deeper discussion of these algorithms is beyond the scope of this tool paper. We have concentrated on Eraser+ and AtomRace because of their simplicity and relatively low overhead (and in the case of AtomRace, absence of false alarms). Experiments with other dynamic analyses implemented in the same framework are, however, an interesting issue for future work.

Most existing works concentrate on detecting concurrency-relate bugs. There are far fewer works on their self-healing. The approach closest to our healing plug-in is probably that of ToleRace [10] healing asymmetric races (i.e., read-write races, not write-write races) by using local copies of shared variables (which cannot be variables referring to external resources, such as files, whose local copy cannot be created).

6 Conclusions

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 consider additional types of concurrency-related bugs.

Acknowledgement. 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. 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.

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. R. O'Callahan and J.-D. Choi. Hybrid Dynamic Data Race Detection. In *Proc.of PPoPP'03*. ACM, 2003.
3. T. Copeland. *PMD Applied*. Centennial Books, 2005.
4. O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java Program Test Generation. *IBM Systems Journal*, 41(1):111–125, 2002.
5. T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A Race and Transaction-aware Java Runtime. In *Proc. of PLDI'07*. ACM, 2007.
6. 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.
7. B. Křena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar. Healing Data Races On-The-Fly. In *Proc. of PADTAD'07*. ACM, 2007.
8. B. Křena, Z. Letko, and T. Vojnar. AtomRace: Data Race and Atomicity Violation Detector and Healer. In *Proc. of PADTAD'08*. ACM, 2008.
9. S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *Proc. of ASPLOS-XII*. ACM, 2006.
10. R. Nagpaly, K. Pattabiramanz, D. Kirovski, and B. Zorn. ToleRace: Tolerating and Detecting Races. In *P*roc. of STMCS'07, 2007.
11. Y. Nir-Buchbinder, R. Tzoref, and S. Ur. Deadlocks: from Exhibiting to Healing. In *Proc. of RV'08*, volume 5289 of *LNCS*. Springer, 2008.
12. 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.
13. 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.
14. 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.
15. Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. *SIGOPS Oper. Syst. Rev.*, 39(5):221–234, 2005.