AtomRace: Data Race and Atomicity Violation Detector and Healer

Zdeněk Letko
Brno University of Technology
Božetěchova 2, Brno
CZ 612 66, Czech Republic
xletko00@stud.fit.vutbr.cz

Tomáš Vojnar
Brno University of Technology
Božetěchova 2, Brno
CZ 612 66, Czech Republic
vojnar@fit.vutbr.cz

Bohuslav Křena
Brno University of Technology
Božetěchova 2, Brno
CZ 612 66, Czech Republic
krena@fit.vutbr.cz

ABSTRACT

The paper proposes a novel algorithm called *AtomRace* for a dynamic detection of data races. Data races are detected as a special case of atomicity violations on atomic sections specially defined to span just particular read/write instructions and the transfer of control to and from them. A key ingredient allowing AtomRace to efficiently detect races on such short atomic sections is a use of techniques for a careful injection of noise into the scheduling of the monitored programs. The approach is very simple, fully automated, avoids false alarms, and allows for a lower overhead and better scalability than many other existing dynamic data race detection algorithms. We illustrate these facts by a set of experiments with a prototype implementation of AtomRace. Further, AtomRace can also be applied to detect atomicity violations on more general atomic sections than those used for the data race detection. They can be defined by the user or obtained by some static analysis.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Verification

1. INTRODUCTION

Concurrent, or multi-threaded, programming has become popular. New technologies such as multi-core processors have become widely available and cheap enough to be used even in common computers. Thus, true concurrency moves from computing centres to everyday life. However, as concurrent programming is far more demanding, its increased use leads to a significantly increased number of bugs that appear in commercial software due to errors in synchronization of its concurrent threads. This stimulates a more intensive research in the field of detecting and removing of such bugs.

This article proposes an architecture for detecting and on-the-fly healing of data races and atomicity violations in Java. The architecture is based on a novel algorithm, which we call *AtomRace*. AtomRace detects only true bugs and does not produce false alarms—at least in the case of *data races*. In the case of *atomicity* problems, the same result is achieved if the algorithm is provided with a correct definition of atomic sections of the code. If they are not provided, they can be approximated either via static or dynamic analysis, of course, with a certain loss of precision. The algorithm scales well and produces only a moderate overhead which allows it to be used not only during testing but also in the field.

The article is organized as follows. The rest of this section contains a short overview of state of the art in data race and atomicity violation detection and healing followed by a short introduction of our approach. Section 2 describes the proposed architecture, including the way how the AtomRace algorithm is incorporated into a self-healing machinery. The AtomRace detection algorithm is introduced in Section 3. Finally, a few experiments document the main outcomes of our solution.

1.1 Data Race and Atomicity Detection and Healing Techniques

Verification problems for programs written in general programming languages are usually undecidable, which is, of course, the case of race detection too. Therefore, tools for data race and atomicity violation detection are based on detecting data races dynamically when only one execution trace is analyzed, or statically, using various approximative and/or semi-algoritmic solutions. Up to now, many different approaches of this form have been proposed. Below, we briefly summarise some of them.

Most current dynamic analysis tools are based on tracking the so-called *locksets* using the observation that if every shared variable is protected by a lock, there is no possibility of operations on this variable being simultaneous, and therefore a race is not possible. A popular such algorithm is *Eraser* [25], later improved with an ownership model [30]. However, a problem of lockset-based algorithms is a high number of false alarms when other than a lock-based synchronization is used. One way how to reduce the number of false alarms is to use the Lamport's *happens-before* relation [14]. A combination of the happens-before relation with the lockset-based approach is, e.g., used in [3, 21, 6, 32, 13]. The works [21] and [32] are based on using the

PADTAD'08, July 20–21, 2008, Seattle, Washington, USA.

so-called *vector clocks* [17] monitoring the happens-before relation. Probably, the most advanced algorithm combining the happens-before relation with locksets is the *Goldilocks* algorithm [6], which not only shrinks the so-far computed locksets while monitoring an evolving run of a program, but also allows them to grow.

Some further works are then motivated by the fact that data race freedom does not imply correct synchronization. A concept of *high-level data races* has been described in [1], together with a method of detecting them using the so-called *view consistency*. In [29], a principle of *method consistency* extends the view consistency to accommodate the scope of methods as a consistency criterion. In [9], a similar notion of *method atomicity* is studied. As the granularity on the level of methods may be too coarse, yet other approaches have then been proposed based on the principle of *serializability* [31, 16, 28], which exploits the idea that two consecutive accesses from one thread to a shared variable should not be interleaved with an unserializable access from another thread. In particular, the AVIO tool [16] introduces a notion of *access interleaving invariants* (AI invariants) identifying a number of patterns how concurrent threads can access a shared variable, which are then classified according to their possible undesirable effects.

On the other hand, numerous static analyses have been introduced to detect data races and atomicity violations. To infer violations in the synchronization, they use, e.g., primarily flow insensitive *type-based systems* [8, 23, 24] or mostly flow sensitive *static versions of lockset algorithms* [7, 19, 12]. Further, there also exist works for detecting data races using specialised *model checking* techniques—cf., e.g., [10, 5].

All the previous work focused only on detecting data races and atomicity violation. We have focused in our previous work also on healing them on-the-fly [13]. ToleRace [18] also tries to detect and remove the detected races from the application by duplication of shared data inside a critical section and so provides an illusion of atomicity when the shared data is updated. If a conflict among copies occurs, ToleRace can in some cases solve it and so hide or tolerate the problem.

1.2 The AtomRace Approach

AtomRace is a new dynamic data race and atomicity violation detection algorithm. As for detecting data races, it is based *directly* on the definition of a (low-level) data race which says that a data race occurs if two or more threads access a shared variable and at least one access is for writing and there is no explicit synchronization which prevent these accesses from being simultaneous. Thus, a data race can be detected by finding a situation when such an access scenario occurs. In AtomRace, this is detected as a special case of an *atomicity violation* when atomic sections are defined simply as sequences of instructions *BeforeAccessEvent*, i, *AfterAccessEvent* where i is a read or write instruction on shared data and *BeforeAccessEvent*/*AfterAccessEvent* are special instructions that are added by instrumentation before/after i. Of course, a data race happens only when at least one of two colliding atomic sections is based on a write instruction. The probability of spotting a collision of this kind in a regular program is low, however, we exploit *noise injection* techniques [27, 4] that may significantly increase this probability—which is actually proven to be the case by our experiments. Note that this mechanism of detecting data races does not generate false alarms. It is due to AtomRace, unlike many other approaches, does not depend on the kind of synchronization primitives used in the analysed program. It naturally supports all types of synchronization (including user defined synchronization primitives) because it does exploit the semantics of such mechanisms, but directly checks the correctness of a program execution.

Further, the atomic sections monitored by AtomRace may be extended to span more subsequent instructions on a shared variable to detect not only data races but also other kinds of problems in synchronization. Such sequences may be obtained from the AI invariants, may be predefined by the user, or obtained by some static analyses directed by looking for standard patterns of code sections to be performed atomically (e.g., testing a shared variable to be non-null and sub-sequently dereferencing it, etc.).

AtomRace can be used not only for detecting synchronization problems, but we also extend it to heal the detected problems either by adding synchronization or influencing the Java virtual machine scheduler [13]. Additional advantage of AtomRace is a low overhead introduced to the monitored application and the fact that it does not generate false alarms when detecting data races. The number of false alarms produced by AtomRace during more general atomicity violation detection depends on the correctness of the predefined atomic sections.

2. ARCHITECTURE

The proposed architecture is depicted in Figure 1 and consists of three modules. The *execution monitoring* module watches the program and triggers predefined events occurred during the execution. Additional information describing the event are collected and the event is then passed to the analysis engine. The *analysis engine* uses AtomRace algorithm to decide if a problem occurs. Finally, the *healing logic* can influence the behavior of the program to prevent the problem manifestation. In the following, we look at the modules in more detail.

Execution monitoring must provide the analysis engine with the following information about each event: (1) the thread which the event belongs to, (2) the shared variable (if any) that was accessed within the event, (3) type of the event determined by the executed instruction, and (4) localization of the event in the application code. For each access of shared variable, two events are needed: *beforeAccessEvent* which is invoked right before the access instruction and *afterAccessEvent* which is invoked right after the access instruction.

The **analysis engine** uses AtomRace algorithm to detect data races and atomicity violations in the stream of events provided by the monitoring module. In the case of atomicity violation detection, the correct atomicity is determined in advance and is available to AtomRace from the external repository. The detection algorithm will be described later in Section 3.

The **healing logic** module controls the influencing of the execution as was studied in [13]. It can be done by safe but not very efficient influencing of Java scheduler. The healing techniques based on influencing the scheduling do not guarantee that a detected problem will really be completely removed, but they can decrease the probability of its manifestation. Moreover, due to the nature of the approach, the

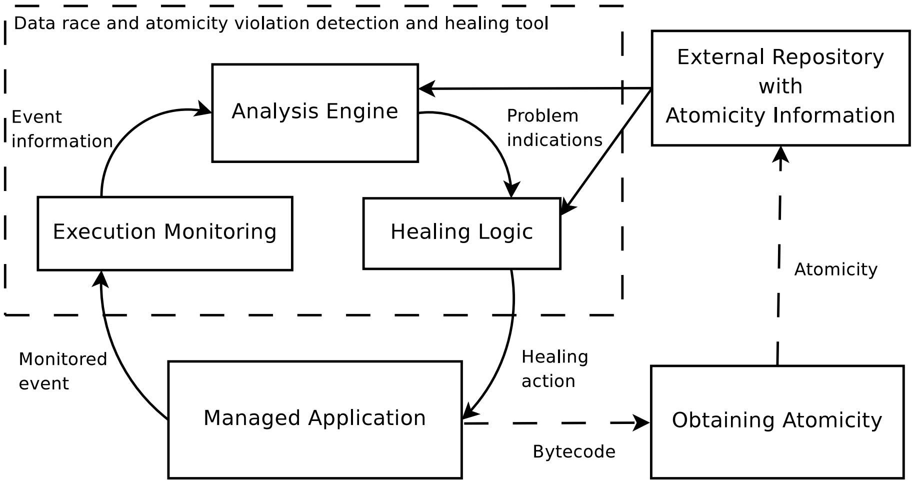

Figure 1: Architecture diagram

healing is safe from the point of view that it does not cause new, perhaps even more serious problems (such as deadlocks). Or, it can be done by more effective but potentially dangerous adding a new synchronization lock. This class of self-healing techniques injects *additional healing locks* to the application. Every time a critical variable on which a possibility of a data race was detected is accessed, the accessing thread must first lock a specially introduced lock. Such an approach guarantees that the detected problem cannot manifest anymore. The healing has to follow the restriction given by correct atomicity to reach the goal. Again, the atomicity information is available in the external repository.

A correct identification of the atomic sections to be monitored is crucial for our detection and healing mechanisms to work properly. Such atomic sections can be defined either manually by the user or obtained automatically via static and/or dynamic analysis in the **obtaining atomicity** module. We have proposed two concrete static analyses for deriving the atomic sections to be monitored. The so-called *pattern-based* static analysis identifies blocks of code that are likely to be intended to execute atomically based on looking for some typical programming constructions, for which such an assumption is usually correct. Two examples of such patterns are the so-called load-and-store and test-and-use patterns [13]. The second static analysis builds on the *access interleaving (AI) invariants with the serializability notion* from [16]. Inspired by the notion of *AI invariants*, we may statically identify couples of two immediately consequent accesses to a shared variable v in the interprocedural CFG as candidates for atomic sections. A dynamic analysis (described also in [16]) can be then used to remove the candidate sections which do not correspond (or do not seem to correspond) to code sections that should really be executed atomically. Finally, let us note that like in the case of data races, it does not make sense to consider atomic section over `final` variables whose values do not change. Further, it is not needed to monitor atomic sections laying within the `<clinit>` method, which is guaranteed by the JVM to be executed atomically.

We expect the following practical usage of the AtomRace. At the end of the application development, the program is analyzed and the set of atomicities that should be followed is determined and stored to the external repository. The formal verification like static analysis or model checking is then used in order to check whether it is safe to enforce particular atomicities for the bug healing. For instance, one should check that enforcing the atomicity of a section of code by introducing an addition lock does not cause a deadlock. The atomicity repository is then distributed together with the application to the customer. In the field, the program is executed together with AtomRace which watches the execution and automatically heals detected bugs.

3. THE ATOMRACE ALGORITHM

AtomRace is an algorithm for detecting data races and atomicity violations at runtime. Data race detection in AtomRace is fully automated and self-contained. For atomicity violation detection, AtomRace expects the atomic sections that should be monitored to be given to it as a part of its input. In fact, data race detection is implemented in AtomRace as a special case of atomicity violation detection on atomic sections that are specifically defined for this purpose.

AtomRace does not track the use of any concrete synchronisation mechanisms—instead, it solely concentrates on the consequences of their absence or incorrect use. That is why AtomRace can deal with programs that use any kind of synchronisation, including non-standard synchronisation mechanisms defined just for the concrete case. AtomRace may miss data races or atomicity violations, but, on the other hand, it does not generate any false alarm in the case of data race detection nor in the case of atomicity violation detection (wrt. the atomic sections provided to it).

We expect AtomRace to work on Java bytecode[1] instrumented as follows: for each shared variable v (corresponding to a field of a certain class in Java) that is to be monitored, we assume each access to v at a location *loc* to be preceded by a code fragment that generates an event *be-*

[1]We refer to Java here, but the basic principles of the algorithm can be used in the context of other programming languages too. Only the treatment of the special cases discussed at the end of the section is Java-specific.

foreAccessEvent(v, loc) and to be followed by a code fragment that generates an event *afterAccessEvent*(v, loc). We view the code fragments generating these events as an implementation of special pseudo-instructions denoted as *beforeAccess*(v, loc) and *afterAccess*(v, loc), and we allow atomic sections to span from/to such instructions. Moreover, we also allow the code to be instrumented to generate events *atomExitEvent*(v, loc). As before, we view the code generating such events as an implementation of a special pseudo-instruction referred to as *atomExit*(v, loc) in the following. This kind of events is used in special cases of the control flow (like exception handling) when it does not make sense to continue with checking the current atomic section.

3.1 Data Race Detection

A data race is defined as a sequence of two accesses to the same shared variable from different threads provided that (1) these accesses are not separated by any synchronisation, and (2) at least one of them is a write access. In AtomRace, such a situation is detected by looking for a violation of *primitive atomic sections* of the form *beforeAccess*(v, loc); *read/write*(v); *afterAccess*(v, loc) where *read/write*(v) is any instruction reading or writing a shared variable v. It is clear that if such a primitive atomic section based on a read instruction is broken by a write instruction, or if a primitive atomic section based on a write instruction is broken by a read or write instruction, a data race happens because there is for sure no synchronisation used neither between *beforeAccess*(v, loc) and *read/write*(v) nor between *read/write*(v) and *afterAccess*(v, loc).[2] In order to significantly increase the probability of detecting data races via violating the described primitive atomic sections, which are very short, we use techniques of noise injection discussed in Section 3.3.

Data race detection based on the above idea can be implemented in a very simple way within handling the events generated by *beforeAccess*(v, loc) and *afterAccess*(v, loc) as we show in Figure 2. For each shared variable v, we define a variable *Access*(v) which is `null` in the case v is not being currently accessed by any thread, and which contains a couple (t, loc) otherwise, where t is the thread that is accessing v at the location *loc*. In the latter case, to simplify the description, we use *Access*$(v).t$ and *Access*$(v).loc$ to refer to the thread and location stored in *Access*(v), respectively. Given a location $loc \in Loc$, we use a function $getMode : Loc \rightarrow \{read, write\}$ to obtain the way v is accessed at *loc*. We use $t_{current}$ to refer to the currently executing thread.

Let us, however, note that the Algorithm in Figure 2 is a little simplified. In reality, it has to be refined a bit to cope with some special situations that may arise in Java. First, variables defined as `volatile` [22] are intended for use in cases where data races are tolerable and so, they should not be monitored by the algorithm. Next, Java uses a special `<clinit>` method to assign implicit values to static variables when they are used for the first time. This initialisation requires a write access which should, however, not be taken into account when looking for data races. Finally, the algorithm should not track shared variables declared as `final` because their values cannot be changed during the execution.

[2]Intuitively, the primitive atomic sections start *after* the instruction preceding a given read/write instruction and stop *before* the instruction following it.

Initialisation:
$\forall v \in SharedVariables : Access(v) =$ `null`;

Computation:
switch $(AtomRaceEvent)$ {
case : beforeAccessEvent(v, loc)
    if $(Access(v) ==$ `null`$)$ **then**
        $Access(v) = (t_{current}, loc)$;
    else
        if $((getMode(Access(v).loc) ==$ `write`$)$ $||$
            $(getMode(loc) ==$ `write`$))$ **then**
            RACE DETECTED

case : afterAccessEvent(v, loc)
    if $(Access(v).t == t_{current})$ **then**
        $Access(v) =$ `null`;
}

Figure 2: Data race detection in AtomRace

Another feature of AtomRace that deserves a comment is its ability to give the user a very valuable *diagnostic information* in case a data race is detected. Namely, the user can be informed about the particular shared variable (i.e., in Java, about the class instance and the field name) on which a data race was detected and about the two program locations whose concurrent execution lead to the data race. Such a piece of information is also very useful within the self-healing process.

3.2 Atomicity Violation Detection

We now extend the above presented algorithm such that it allows us to deal with more general *atomic sections*. However, as above, we still assume an atomic section to be associated with a single shared variable only. For a shared variable v, we view an atomic section as a code fragment which is delimited by a single entry point and possibly several end points. The intended meaning of an atomic section over a variable v is that when a thread t starts executing the atomic section, no other thread should access v before t reaches an end point of the atomic section (with the exception of some kinds of accesses that may be explicitly allowed for the given atomic section). Quite naturally, we assume the entry point of an atomic section to correspond to some *beforeAccess*(v, loc) instruction and the end points to correspond to some *afterAccess*(v, loc') or *atomExit*(v, loc') instructions.

To allow a specification of which accesses from other threads *should not be considered to break an atomic section* when it is being executed by some tracked thread, we associate a (possibly empty) subset of the set $\{read, write\}$ with each end point of each atomic section. This subset indicates which kind of operations *can be* performed by other threads on v while a tracked thread is running between the entry point of a given atomic section and a given end point of this atomic section. As discussed in [15], we can use this information, e.g., to allow not only checking of pure atomicity, but to allow for handling not purely atomic, but *serializable accesses* (in the sense of [16]) as well.

When dealing with *several atomic sections* associated with the same variable v, we require that these sections *do not overlap* in any other way than possibly on their entry and end points. More precisely, the only allowed overlap is that one atomic section has an entry point $beforeAccess(v, loc)$ while the other has an end point $afterAccess(v, loc)$ for the same location loc. Due to this requirement, a process can only be in one atomic section at a time (with the exception of leaving one section and at the same time entering another).

As shown in Figure 3, detection of atomicity violation can again be implemented in a simple way within handling the events $beforeAccessEvent(v, loc)$, $afterAccessEvent(v, loc)$, and $atomExitEvent(v, loc)$. For the purpose of describing the algorithm, we expect the set of atomic sections associated with a variable v that are supposed to be tracked and that satisfy the conditions described above to be encoded in a "flattened" way as a set $Atomic(v)$ of triples $(loc_{entry}, loc_{end}, A)$ where $A \subseteq \{read, write\}$ and loc_{entry}, loc_{end} are locations corresponding to entry and end points of particular branches of the atomic sections encoded by $Atomic(v)$. We use the notation $Atomic(v).A(loc_1, loc_2)$ to refer to the set A in $(loc_1, loc_2, A) \in Atomic(v)$.

For each shared variable v whose atomic sections we intend to monitor, we maintain the set $Access(v)$ used already in Figure 2. Moreover, we also build a set $SuspectAccess(v)$ which contains types of accesses to v that came from other threads than the one whose execution in an atomic section over v we are currently monitoring. The algorithm works in such a way that if a thread t is entering an atomic section over a variable v over which no atomic section is currently being monitored, we start monitoring accesses to v from other threads, and once t is leaving the atomic section via some end point, we check that no undesirable access to v from a thread other than t has happened. Note that we always monitor atomicity of an atomic section associated with a certain variable v just for a single thread—the one that entered a critical section over v while no other thread was currently executing an atomic section over v.

Note that the algorithm shown in Figure 3 is a bit simplified wrt. the above described functionality. In particular, we have left out the treatment of overlapped atomic sections, which can, however, be added in a straightforward way.

The algorithm can be easily extended to cope with *circular atomic sections*, i.e. atomic sections where $loc_{entry} = loc_{end} = loc$, but we do not want the atomic section to terminate just after firing the statement at the location loc. In such a case, we tag such a section in a special way, and the algorithm does not leave the atomic section during the first occurrence of *afterAccessEvent* at loc.

Finally, the algorithm can also be extended to support recursive atomic sections by counting how many times the section has been entered and left and by terminating the atomic section only when these numbers are equal.

3.3 Race and Atomicity Violation Exhibition

The AtomRace algorithm was originally developed for detecting data races and atomicity violations for healing the problems at runtime. Therefore, the aims of the algorithm slightly differ from the previous approaches. The first difference is not to find as many potential problems as possible (with the chance of false alarms) but to report only true alarms in order to invoke the expensive healing mechanisms only when some problem really occurs. The next essential aim of the algorithm is to cause as small overhead as possible due to the intent to use it in the field. This is achieved by using less shared data structures than in many other race detection algorithms. Despite that, this algorithm can be also useful in bug hunting within the application testing if suitable *noise injection* is used.

Initialisation:
$\forall v \in SharedVariables :$
  $Access(v) = \texttt{null}, SuspectAccess(v) = \emptyset;$

Computation:
switch $(AtomRaceEvent)$ {
case : beforeAccessEvent(v, loc)
  if $(Access(v) == \texttt{null})$ **then**
    if $(\exists l_{end}, A : (loc, l_{end}, A) \in Atomic(v))$ **then**
      $Access(v) = (t_{current}, loc);$
  else
    if $(Access(v).t \mathrel{!}= t_{current})$ **then**
      add $(t_{current}, loc)$ **to** $SuspectAccess(v);$

case : afterAccessEvent(v, loc), atomExitEvent(v, loc)
  if $(Access(v).t == t_{current}$ &&
    $\exists A : (Access(v).loc, loc, A) \in Atomic(v))$ **then**
    if $(SuspectAccess(v) \mathrel{!}= \emptyset)$ **then**
      $A = Atomic(v).A(Access(v).loc, loc);$
      foreach $(t_s, l_s) \in SuspectAccess(v)$ **do**
        if $(getMode(l_s) \notin A)$ **then**
          ATOMICITY VIOLATION DETECTED
    $Access(v) = \texttt{null};$
    $SuspectAccess(v) = \emptyset$;
}

Figure 3: A simplified version of the AtomRace algorithm for detecting atomicity violation

As we have already mentioned, the problem with using AtomRace to find as many data races and atomicity violations as possible is that the considered atomic sections may be very short and the probability of observing a real conflict on them may be very low. This problem is also common to many algorithms based on building happens-before relations. However, the probability may be significantly increased by suitably influencing the execution of the program what is exactly the purpose of noise injection. In general, noise injection is a technique that forces different legal interleavings for particular executions of a test in order to increase the concurrent coverage. In fact, it simulates the behaviour of various possible schedulers. The noise can be injected at any instrumented point (e.g., during the execution of $beforeAcces(v, loc)$ or $afterAccess(v, loc)$) of the tested software. When such a point is reached, the noise heuristics decides—randomly or based on a specific bug-finding technique—if it injects some kind of delay or other kind of influencing the execution (like a context switch) there or not.

The introduction of noise can help the detection of races and/or atomicity violations in two ways: firstly, different legal thread interleavings are enforced. Secondly, randomly chosen atomic sections are executed for a longer time period and therefore the probability that a conflict will occur on them is increased. Both of this helps to see conflicts that would not be seen otherwise. Of course, the probability of seeing a data race and/or atomicity violation can then be rapidly increased also if a true multiprocessor computer is used for testing.

In our prototype implementation of AtomRace, we use the ConTest infrastructure [4, 20, 27] for instrumentation, handling the generated events, as well as for noise injection. Based on our experience with ConTest and influencing the Java scheduler for self-healing purposes [13], we have proposed three noise heuristics for increasing the probability of detecting data races and/or atomicity violations. All of them are based on injecting calls of `Thread.sleep()` inside atomic sections to increase their duration followed by calls of `Thread.yield()` to force a thread switch. The probability of noise injection in the given location is driven by the parameter that ranges from 0 (=never) to 1000 (=every time). The duration of sleep is given by the number of milisecond that sleep should last. The three heuristics we have implemented are the following:

- *A random heuristics.* This is the simplest heuristics that can be used during a normal testing when there is no suspicion that something wrong is happening in the program. It injects noise to randomly chosen atomic sections.
- *A variable-based heuristics.* This heuristics can be used when some concrete variable is suspected to be accessed with a wrong synchronisation. The noise is injected to the sections associated with instances of the suspected variable only.
- *A heuristics based on program locations.* This approach allows the user to identify atomic sections which are suspected to be problematic. The noise is injected to the given program locations only.

The second and third heuristics can be also used for testing and debugging. If AtomRace detects a race or an atomicity violation in one run of the tested software, the developer can use a noise injection focused on the suspected variable or the problematic program locations to increase the probability of a repeated manifestation of the detected problem.

4. PROTOTYPE IMPLEMENTATION AND EXPERIMENTS

We have implemented a prototype of a data race and atomicity violation detection tool based on the AtomRace algorithm. The tool is implemented in Java on top of the ConTest infrastructure [4], mentioned already at the end of the previous section, which provides us with a listeners architecture [20], a possibility of a static bytecode instrumentation, and noise injection heuristics [27]. Our static analyses have been implemented in FindBugs [11, 2]—a Java bytecode static analysis tool. In fact, we have implemented the entire architecture depicted in Figure 1, including execution monitoring (which is done by ConTest) and healing.

ConTest instruments not only accesses to shared variables (fields in Java) and accesses to array cells but also various other constructions in the bytecode related, e.g., to dealing with monitors, to entering and exiting of methods, basic blocks, etc. ConTest also adds instrumentation important for computing code coverage. However, for AtomRace, the instrumentation added before and after the access to shared variables and array cells is the most important (other instrumentation points are used just to generate the special *atomExitEvent*). At each instrumentation point, ConTest provides us automatically with most of the data that we need for our purposes. Only some special pieces of information have to be specially computed on-the-fly as, e.g., the fact whether the array cell that we are currently handling belongs to a shared or local array.

We have also fully implemented all three noise generating heuristics presented in Section 3.3. The difference between our and ConTest injection heuristics is in the place where the noise is injected. ConTest injects noise with the intention to see different access interleavings and/or better synchronisation coverage, but it cannot take into account the atomic sections used by AtomRace. Our heuristics put the noise directly between *beforeAccessEvent* and *afterAccessEvent* and therefore it increases the duration of the monitored atomic sections.

We have only partially implemented some of the simple static analyses for obtaining atomicity that we have mentioned in Section 2. Currently, we support only the load-and-store atomicity pattern for the pattern-based static analysis and we have implemented an intraprocedural static analysis for statically inferring AI invariants (method calls are replaced with *atomExitEvent*s in this case, and thus interprocedural AI invariants are not checked by the algorithm).

4.1 Experiments with Data Races

We have evaluated AtomRace on several examples including a small toy example of a program simulating a bank with a simple concurrent access to the accounts, an IBM web crawler algorithm with 19 classes and 1200 lines of code embedded in IBM WebSphere product, and the Java TIDorb project with 1400+ classes. Java TIDorb is a CORBA-compliant ORB (Object Request Broker) product that is a part of the MORFEO Community Middleware Platform [26]. We have used several tests created by the developers and available in the project repository. In the following, we present results achieved for one of them only—namely, the *echo concurrent* test. It starts the server for handling incoming requests and then starts a client which constructs several client threads (10 in our case) each sending requests (40 in our case) to the server.

In the bank example, AtomRace worked well, and the (in advance known) data race present in the code was correctly detected. The data race possible in the web crawler example, which was also known in advance, manifests only very rarely. Here, let us recall that AtomRace was primarily designed to detect only the races that manifest along the current execution path without producing any false alarm. This is important for self-healing when the healing engine should not influence executions where a race does not occur. On the other hand, this feature is a limitation for using AtomRace for bug hunting. Accordingly, AtomRace did not detect any data race in the web crawler example until a variable-focused noise was used. This noise injection technique needs the variable to which a noise should be focused to be specified. For this purpose, we have used a modification of the Eraser [25] algorithm introduced in [13]. The modified Eraser algorithm pinpointed the variable on which a race can occur. This information was used by AtomRace to focus the noise injection on all instances of the problematic variable. With this kind of noise injection, the race was

	None	CT 50	CT 150	Rand 50/50	Loc 1000/50	Var 500/50	Var 500/150	Var 750/50	Var 1000/50
2 proc.	3,36 / 0,89	45,26 / 2,09	62,12 / 2,36	16,77 / 2,22	0,83 / 0,22	257,46 / 3,42	253,97 / 3,52	362,92 / 3,84	674,35 / 3,34
4 proc.	1,58 / 0,37	31,79 / 1,9	40,87 / 2,42	11,07 / 2,27	11,38 1,71	261,38 / 3,63	271,05 / 3,73	387,91 / 3,82	667,46 / 3,38
8 proc.	2,55 / 0,49	35,47 / 2,31	55,74 / 2,35	10,87 / 2,18	10,74 / 2,07	270,44 / 3,68	268,68 / 3,7	401 / 3,88	666,79 / 3,25

Table 1: Noise injection influence on data race detection efficiency

then successfully detected.

In the case of the TIDorbj echo concurrent test, AtomRace detected data races on four different variables, which were *not* known in advance. Because the data races exhibit relatively rarely (but not as rarely as in the case of the web crawler), this case study offers a good opportunity to study effects of using different computing environments[3] as well as different noise injection strategies on the AtomRace-based data race detection. The results are shown in Table 1.

The columns of Table 1 correspond to different noise injection strategies used with different parameters—up to the column labelled with $None$, which corresponds to the case when no noise injection heuristics is used.[4] The rows of the table correspond to different processor architectures used for the testing. The entries of the table consist of two values written in the form $races/coverage$. The $races$ value represents the average number of data races detected during one execution, and the *coverage* value represents the average number of distinct variables over which a race has been detected (out of the four that were found across all the runs).

The $CT\ x$ heuristics that appears in Table 1 is the *random noise heuristics* provided by ConTest where x represents the per mile of locations where the noise is injected (1000 means everywhere, 0 nowhere). As can be seen from the table, introducing even a small percentage of noise can rapidly increase the probability of a successful race detection.

The $Rand\ x/y$ column shows the results that we have obtained using our own random heuristics described in Section 3.3. The x value represents the per mile of places where a noise was injected, and the y value represents the duration of the injected noise (in nanoseconds). It can be seen that our random heuristics is not as successful as that of ConTest.

The $Loc\ x/y$ heuristics represents the location-based heuristics described in Section 3.3. The meaning of x and y is the same as in the previous heuristics. We have provided the heuristics with the list of all locations where a race has previously been detected. The results are not so convincing in comparison with the use of the ConTest heuristics but in this case, only seven distinct locations in the code were influenced, and so the performance degradation was smaller.

[3]The success of concurrency testing often highly depends on the testing environment, therefore, we have run our tests on three different machines: a PC with two Intel Xeon 1,7GHz processors, a PC with two Dual Core AMD Opteron 2220 processors, and a PC with two Quad Core Intel Xeon 5355 processors.

[4]We have to, however, note that some noise is generated already due to the presence of our code at the instrumented points of the bytecode, and even this noise can already significantly increase the probability of a race manifestation compared to the really original code.

Finally, results obtained using the variable-based heuristics from Section 3.3, referred to via $Var\ x/y$, are listed in the table. This heuristics focused the noise on a given set of variables which were those four on which the race has already been detected in previous runs. In this case, the percentage of successfully detected races increased dramatically and therefore this heuristics is very suitable for forcing race occurrences. It can be seen that changes in the noise duration have a minor influence on the percentage of detected races.

4.2 Experiments with Atomicity Violations

We have also evaluated capabilities of AtomRace for detecting atomicity violations. Recall that, in AtomRace, atomicity violations are detected in a similar way as data races, but the atomic sections span more than only one instruction. This leads to a higher probability to see a conflict on a problematic variable, which was the case also in our various experiments such as those whose results are presented in Table 2. The results presented in this table were in particular collected while experimenting with the simple bank simulating program presented in the previous subsection. We concentrated on monitoring an instance of the load-and-store bug pattern, which appears in the form of a statement `BankTotal += sum;` manipulating a global variable from different threads without a synchronisation in the program.

The rows of Table 2 correspond to experiments with different numbers of processors in the machines used for testing. The second column of the table shows the ratio of runs where the error manifested by an erroneous final value of `BankTotal` when the program was run without ConTest and the detector. The next column (labelled with "*data race*") shows the ratio of runs where a data race was successfully detected by AtomRace. For the case when an atomic section covering the `BankTotal += sum;` statement was specified, the column labelled with "*atomicity violation*" shows the ratio of a successful detection of a problem in the synchronisation, which is notably increased compared to the previous case. Finally, the last column shows how this ratio further increases when we combine the atomicity violation detection with the CT 50 heuristics explained in the previous subsection.

AtomRace also successfully detected atomicity violations in the web crawler test case where the test-and-use bug pattern can be identified on the problematic variable. A similar situation and results were then as well achieved in TIDorb where both the load-and-store and test-and-use bug patterns were identified. In all executions, AtomRace correctly detected atomicity violations exactly at the time when they were really occurring.

processors	ratio of problem manifestation	AtomRace: successful problem detection ratio		
		data race	atomicity violation	atomicity violation + CT 50
2	0	0,04	0,09	0,96
4	0	0,28	0,84	1
8	0,10	0,16	0,70	1

Table 2: Atomicity violation detection

4.3 Self-Healing and Performance

Most of the healing capabilities of AtomRace are due to the results presented already in [13], which were based on using a modified Eraser algorithm. Compared with that algorithm, AtomRace does not produce any false alarms and therefore the performance degradation caused by the healing actions is lower as witnessed by our experiments presented in detail in [15]. The memory and processor consumption of AtomRace are also lower because it does not need to compute and maintain a lock set for each shared variable. Finally, the bug reports produced by AtomRace directly pinpoint a problematic variable together with at least two program locations manipulating it in a conflicting way, and therefore it is easier to check what happens in the code—which we really appreciated in the TIDorb case study where we were faced with data races unknown at that time.

Our current implementation does not contain any performance optimisation. Due to the fact that ConTest implicitly instruments also final and volatile field accesses, we have to determine them on-the-fly. A similar situation is also with local array cells. Therefore, the performance overhead generated by our implementation highly depends on the frequency of executions of instrumented points. Our experiments showed us that in the case of Java TIDorb, an execution of the fully instrumented code with only ConTest activated and without any further noise injection caused in average a 4-times longer execution, and ConTest together with AtomRace then caused even a 12-times longer execution. For the bank and web crawler case studies, the overhead was of course lower because there were less instrumentation points. An execution of the fully instrumented bank case study with only ConTest activated and without any further noise injection caused in average a 5% longer execution while ConTest together with AtomRace caused a 25% longer execution.

5. CONCLUSIONS

We have presented a novel algorithm called AtomRace which detects data races and atomicity violations at runtime. The algorithm does not produce any false alarms about data races nor about atomicity violation (in the latter case provided that the algorithm is not instructed to monitor code sections which are not really expected to be atomic). Because AtomRace minimises the amount of work with auxiliary data structures shared among the monitored threads, it is—compared to other existing solutions—faster and more scalable. We have also described that AtomRace can be easily incorporated into a self-healing architecture. In fact, this use of AtomRace, within which all the described features of AtomRace are especially welcome (with the fact that AtomRace may miss data races or atomicity violations being less stressed), was one of the main motivations for its proposal. However, the experiments that we have conducted show that AtomRace combined with suitable noise injection mechanisms, which we have also proposed, can as well be quite successfully used for testing of concurrent software aimed primarily at bug finding.

The first experimental results obtained from a prototype implementation of AtomRace provide us also with a lot of inspiration for future work. First, the atomicity inferring mechanisms need to be improved. Our experiences showed that the AI invariant-based approach requires some number of correctness assertions to be present in the code. These are needed for an efficient dynamic pruning of the set of candidate atomic sets obtained from a static analysis. The pattern-based approach for detecting candidate atomic sections worked well, but our current implementation supports only the load-and-store pattern. For the future, we would like to also implement and test dealing with the test-and-use pattern and investigate some more patterns too. Another weakness of our prototype implementation based on ConTest is the overhead it introduces. We believe that this overhead can be significantly reduced. For instance, we instrument and then handle events from many locations in the code whose monitoring is not really needed (like, e.g., dealing with final variables, local arrays, locations important for various ConTest analyses such as code coverage analysis, etc.). These issues can be tackled by a partial instrumentation already available in ConTest, but not yet used by us. To properly use partial instrumentation, one needs to devise a suitable static analysis to decide which locations must be instrumented. Finally, there is also a space for proposing new noise injection heuristics capable of increasing the probability of race detection without significantly degrading the performance.

6. ACKNOWLEDGMENTS

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, and by Czech Science Foundation under the contracts GA102/07/0322 and GP102/06/P076.

7. REFERENCES

[1] C. Artho, K. Havelund, and A. Biere. High-level data races, 2003.

[2] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Using findbugs on production software.

In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion*, pages 805–806, New York, NY, USA, 2007. ACM.

[3] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 258–269, New York, NY, USA, 2002. ACM Press.

[4] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.

[5] T. Elmas, S. Qadeer, and S. Tasiran. Precise data-race detection and efficient model checking using locksets. Technical Report MSR-TR-2005-118, Microsoft Research, March 2006.

[6] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware java runtime. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 245–255, New York, NY, USA, 2007. ACM Press.

[7] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. *SIGOPS Oper. Syst. Rev.*, 37(5):237–252, 2003.

[8] C. Flanagan and S. N. Freund. Type-based race detection for java. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 219–232, New York, NY, USA, 2000. ACM Press.

[9] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. *SIGPLAN Not.*, 39(1):256–267, 2004.

[10] T. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *Proceedings of the 15th International Conference on Computer-Aided Verification (CAV)*, pages 262–274. Lecture Notes in Computer Science 2725, Springer-Verlag, January 2003.

[11] D. H. Hovemeyer. *Simple and effective static analysis to find bugs*. PhD thesis, College Park, MD, USA, 2005. Director-William W. Pugh.

[12] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta. Fast and accurate static data-race detection for concurrent programs. In *CAV*, pages 226–239, 2007.

[13] B. Křena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar. Healing data races on-the-fly. In *PADTAD '07: Proceedings of the 2007 ACM workshop on Parallel and distributed systems: testing and debugging*, pages 54–64, New York, NY, USA, 2007. ACM.

[14] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[15] Z. Letko. Dynamic data race detection and self-healing in java programs. Master's thesis, Faculty of Information Technology, Brno University of Technology, 2008.

[16] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 37–48, New York, NY, USA, 2006. ACM Press.

[17] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel and Distributed Algorithms*. 1988.

[18] R. Nagpaly, K. Pattabiramanz, D. Kirovski, and B. Zorn. Position paper - tolerace: Tolerating and detecting races. In *STMCS: Second Workshop on Software Tools for Multi-Core Systems (STMCS)*, 2007.

[19] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. *SIGPLAN Not.*, 41(6):308–319, 2006.

[20] Y. Nir-Buchbinder and S. Ur. Contest listeners: a concurrency-oriented infrastructure for java test and heal tools. In *SOQUA '07: Fourth international workshop on Software quality assurance*, pages 9–16, New York, NY, USA, 2007. ACM.

[21] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 167–178, New York, NY, USA, 2003. ACM Press.

[22] T. Peierls, B. Goetz, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

[23] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: context-sensitive correlation analysis for race detection. *SIGPLAN Not.*, 41(6):320–331, 2006.

[24] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller. Automated type-based analysis of data races and atomicity. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 83–94, New York, NY, USA, 2005. ACM.

[25] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multi-threaded programs. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 27–37, New York, NY, USA, 1997. ACM Press.

[26] J. Soriano, M. Jimenez, J. M. Cantera, and J. J. Hierro. Delivering mobile enterprise services on morfeo's mc open source platform. In *MDM '06: Proceedings of the 7th International Conference on Mobile Data Management*, page 139, Washington, DC, USA, 2006. IEEE Computer Society.

[27] R. Tzoref, S. Ur, and E. Yom-Tov. Instrumenting where it hurts: an automatic concurrent debugging technique. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 27–38, New York, NY, USA, 2007. ACM.

[28] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT*

symposium on Principles of programming languages, pages 334–345, New York, NY, USA, 2006. ACM Press.

[29] C. von Praun and T. Gross. Static detection of atomicity violations in object-oriented programs, 2003.

[30] C. von Praun and T. R. Gross. Object race detection. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 70–82, New York, NY, USA, 2001. ACM Press.

[31] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. *SIGPLAN Not.*, 40(6):1–14, 2005.

[32] 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.