Automatically Repairing Concurrency Bugs with ARC

David Kelk, Kevin Jalbert, Jeremy S. Bradbury

Software Quality Research Laboratory
University of Ontario Institute of Technology
Oshawa, Ontario, Canada
{david.kelk, kevin.jalbert, jeremy.bradbury}@uoit.ca

Abstract. In this paper we introduce ARC – a fully automated system for repairing deadlocks and data races in concurrent Java programs. ARC consists of two phases: (1) a bug repair phase and (2) an optimization phase. In the first phase, ARC uses a genetic algorithm without crossover to mutate an incorrect program, searching for a variant of the original program that fixes the deadlocks and data races. As this first phase may introduce unneeded synchronization that can negatively affect performance, a second phase attempts to optimize the concurrent source code by removing any excess synchronization without sacrificing program correctness. We describe both phases of our approach and report on our results.

Keywords: bug repair, concurrency, concurrency testing, evolutionary algorithm, SBSE.

1 Introduction

As computers and even mobile devices now ship with more than one core per chip, programs must parallelize to take advantage of improvements in processing power [23]. On a multicore system concurrency provides a potentially significant benefit with respect to performance, however, writing concurrent source code can be difficult and error-prone, especially when one considers the set of possible thread interleavings of a concurrent program. Further exacerbating the issue of writing correct concurrent source code is the fact that concurrency bugs can be difficult to find because they may occur in only a small set of possible thread interleavings [21]. Even when a concurrency bug has been detected, its repair is often non-trivial because many concurrency bugs are the result of the interaction of different code fragments executing in different threads within a program.

The use of search-based software engineering (SBSE) [12] techniques to automatically repair bugs in sequential programs is a well researched idea [17,24]. To address the challenges of detecting and fixing concurrent programs we propose ARC (**A**utomatic **R**epair of **C**oncurrency bugs) – an automatic technique to repair deadlocks and data races in concurrent Java programs. ARC requires no formal specifications or annotations. Only the Java source code and a test suite

capable of demonstrating known deadlocks and data races are necessary. ARC consists of two phases: (1) a bug repair phase and (2) an optimization phase. At its core, ARC works by using a genetic algorithm without crossover ($GA\neg C$) to evolve variants of an incorrect concurrent Java program into a variant that fixes all known bugs.

A common problem for automatic bug repair techniques is the size of the search space of possible bug fixes. Applying these techniques to concurrent programs introduces thread interleavings which increase the difficulty of searching the space. To counteract this challenge, ARC incorporates techniques to constrain the search space and make it tractable. First, we limit the algorithm to only fixing deadlocks and data races in concurrent Java programs (instead of all types of concurrency bugs). Second, ARC only targets modifications to concurrency mechanisms as potential bug fixes. For example, `Synchronize` statements maybe added, removed, and manipulated. The possible bug fixes are comprised of the combined application of 12 mutation operators to evolve the program[1]. Third, the Chord tool [22] is used to perform a static analysis of a concurrent program to target specific shared classes, methods and variables where bug fixes can be applied (i.e., localization of bug fixes within the source code). This shared list is further refined by the ConTest testing tool [7]. Fourth, the evaluation of potential bug fixes are evaluated using ConTest which injects noise (i.e., random delays) that assist in exploring the interleaving space during testing.

Next in Section 2 we cover background material related to concurrency bugs and genetic algorithms. The motivation for ARC along with an example problem are presented in Section 3. In Section 4 we describes how ARC evolves fixes for data races and deadlocks (Phase 1). Improving nonfunctional properties of the program, such as its execution time (Phase 2), is described in Section 5. We evaluate ARC in Section 6 on a set of programs from the IBM concurrency benchmark [8, 9, 13]. In Section 7 we discuss related works in the field of automatic program repair and explain how ARC is novel when compared to previous approaches. Finally, we conclude and present future work in Section 8.

2 Background

2.1 Concurrency Bugs

Data races and deadlocks are two of the most common concurrency bugs. A data race can be defined as: *"...two or more concurrent threads access a shared variable and when at least one access is a write, and the threads use no explicit mechanism to prevent the access from being simultaneous."* [20]. A deadlock can be defined as: *"...a situation where two or more processes are unable to proceed because each is waiting for one of the others to do something in a deadlock cycle ...For example, a deadlock can occur when one thread in a program holds a lock that another thread desires and vice-versa"* [20].

[1] A number of the mutation operators used to repair bugs in ARC are converse to the mutation operators in the ConMAn mutation testing tool [4].

Buggy Program:

```
write(int var1){
  ... // Expensive loop
  data = var1;
  ... // Database query
}

int public read(){
  return data;
}
```

Fixed Program:

```
synchronize write(int var1){
  ... // Expensive loop
  data = var1;
  ... // Database query
}

int synchronize read(){
  return data;
}
```

Fig. 1. A developer first synchronizes the `read` function, yet the bug still exists. Synchronizing the `write` method as well fixes the bug.

1st **Optimization on Fix:**

```
public write(int var1){
 ... // Expensive loop
 synchronized(this){
   data = var1;
   ... // Database query
 }
}

int synchronize read(){
 return data;
}
```

2nd **Optimization on Fix:**

```
public write(int var1){
 ... // Expensive loop
 synchronized(this){
   data = var1;
 }
 ... // Database query
}

int synchronize read(){
 return data;
}
```

Fig. 2. A developer shrinks the critical region to exclude the expensive loop (Optimization 1). Next, a developer shrink the critical region again to exclude the database query (Optimization 2).

2.2 Genetic Algorithms

Genetic algorithms [11] (GAs) are a heuristic search technique modelled on natural evolution. They are population based and uses mutation, crossover and a fitness function to evolve solutions to problems. Proposed solutions are encoded in individuals of the population. Each individual is evaluated by a fitness function, an equation that determines how close the individual is to the solving the problem. The more fit a individual's solution is, the greater the chance it will pass it's genetic material (i.e., itself) into the next generation. Crossover mixes the individuals to produce new ones while mutation injects fresh information in to the population so it does not become stagnant.

ARC uses a genetic algorithm without crossover (GA¬C) and selection. A population of proposed solutions is generated in the first generation and mutated each generation. Two ending conditions exist. First, a fix is found for the data races and deadlocks or a fixed number of generations pass with no solution found.

3 Motivating Example

To illustrate the challenges of concurrency bug repair we consider an example of a data race and how ARC might fix it. In the left part of Figure 1 the `read` and `write` method access a shared variable. A very simple data race exists because there is no atomic access to the `data` variable during the concurrent reading or writing. A possible repair involves synchronizing both accesses as shown in the right part of Figure 1. Note that synchronizing one method alone does not fix this bug. Section 4 describes this in more detail.

The solution in the right part of Figure 1 is far from ideal. The solution found by ARC forces other threads to wait unnecessarily while the write method works in the loop and database sections. An optimization is to shrink the critical region guarded by the synchronize statements to only guard access to the shared variable as shown in Figure 2. Section 5 describes how ARC attempts to optimize fixes found in the first phase of operation by removing and shrinking synchronization blocks.

4 Phase 1: Fixing Deadlocks and Data Races

ARC requires two inputs: An incorrect concurrent Java program and JUnit tests exercizing the errors. The test suite is the oracle that determines if bugs still exists in the program. One limitation of ARC (and of other related automatic bug fixing techniques mentioned in Section 7) is that it can only fix bugs detectable by the test suite. Given an incorrect program ARC performs a static analysis to identify the variables, classes and methods involved in concurrency. It then invokes the GA¬C to find fixes for the data races and deadlocks. Each generation is broken down into a number of steps, shown in Figure 3 and described here.

Update Population. First, the members of the population must be created. If ARC has just started, the original incorrect program is replicated and copied into each member of the GA¬C in the first generation. For succeeding generations N the program for the same member from generation N - 1 is used.

Generate Mutants (Update Representation). After the population is updated, ARC generates all mutants for all members using the operators in Table 1. These operators are implemented in TXL [6] using pattern matching and replacement rules. An example mutant created by the EXSB operator is shown in Figure 4.

Apply a Mutation to an Individual. Once all mutants are generated for each individual, ARC selects a type of mutation (e.g. EXSB) and then an instance of it (e.g. 4th mutant generated) from those available. The selected mutation is copied into the source for the member. It is possible that the mutant is not valid. For example, a new synchronization block could have been added that synchronizes on a variable that is out of scope. ARC attempts to compile the project. If an error is detected, the mutation is rolled back and another is selected. This continues until a successful compilation or ARC runs out of mutants. In this latter case, ARC raises and exception and ends.

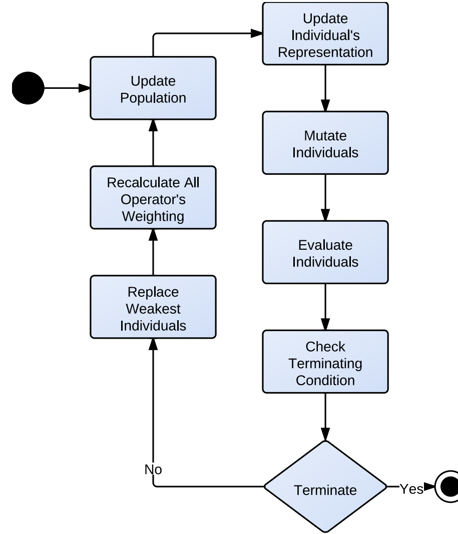

Fig. 3. Detailed view of the GA¬C used in Phase 1, the repair phase.

Table 1. Set of mutation operators used by ARC.

Operator Description	**Acronym**
Add a synchronized block around a statement	ASAT
Add the synchronized keyword to the method header	ASIM
Add a synchronized block around a method	ASM
Change the order of two synchronized blocks order	CSO
Expand synchronized region after	EXSA
Expand synchronized region before	EXSB
Remove synchronized statement around a synchronized statement	RSAS
Remove synchronization around a variable	RSAV
Remove synchronized keyword in method header	RSIM
Remove synchronization block around method	RSM
Shrink synchronization block after	SHSA
Shrink synchronization block before	SHSB

Program P**:**

```
obj.write(var1);
synchronized(lock){
  myHash.remove(var1);
}
```

Program P'**:**

```
synchronized(lock){
  obj.write(var1);
  myHash.remove(var1);
}
```

Fig. 4. An example of the EXSB (expand synchronization before) mutation operator.

Evaluate Individuals. A mutation may be beneficial, destructive or benign. We must evaluate it to determine it's effect on the program. A key prob-

lem in evaluating mutants is the unpredictability of thread interleavings. If a concurrency bug appears in only a few possible interleavings, how can we gain confidence that a proposed fix actually works?

ARC uses IBM's ConTest tool [7] to instrument the software under repair by injecting noise into the selection of interleavings. This causes threads to randomly delay at different times during execution, increasing the chance that different interleavings are explored. By running the instrumented version of the program multiple times we gain more confidence that a larger set of the interleavings are explored. Choosing the number of times ConTest run to test each proposed fix is the most crucial parameter in ARC. Confidence in any proposed fix must be carefully balanced against the time required to find the fix.

The number of successful ConTest executions are used to determine fitness:

$$functional\ fitness(P) = (s \times sw) + (t \times tw)$$

$$\text{where: } s = \#\ of\ successful\ executions,\ sw = success\ weighting,$$
$$t = \#\ of\ timeout\ executions,\ tw = timeout\ weighting$$

When determining fitness we consider both a successful execution and a timeout as positive factors. Timeout executions are positive because given more time they may become successful executions. Timeouts are weighted less than a successful execution since the success of a timeout execution is not guaranteed.

If ARC finds an individual that achieves 100% successful executions, we need to ensure it is truly a fix. It is possible that a proposed solution could still contain a bug that escaped detection because the interleaving exhibiting it were not selected by ConTest. To increase our confidence that a fix is correct we take the base number of ConTest runs (say 10) and multiply it by an additional safety factor (say 20). We run the proposed fix through ConTest ($10 \times 20 = 200$) more times to give us additional confidence the fix holds. If a data race or deadlock is found during these additional runs, the fix is rejected and the search continues. This continues until a correct program is found or the GA¬C runs out of generations.

Replace Weakest Individuals. We believe the *competent programmer hypothesis* [1] applies when fixing concurrency errors. That is, that programmers strive to create correct programs. Programs with bugs in them are nearly correct so the distance in the search space from an incorrect program to a correct program is small(er) and tractable. Even with a smaller search space, evolutionary algorithms may evolve candidate solutions that stray down paths leading to little or no improvement. To encourage individuals to explore more fruitful areas of the state space we include the option to restart or replace the lower w percentage (say 10%) of individuals if they under-perform for too long. Two replacement strategies are used. First, the under-performing member is replaced with a random individual from the upper x percent of the population. Second, the member is replaced by the original incorrect program.

Recalculate Operator Weighting. ARC leverages historical information on how successful different mutation operators have been and about the relative dominance of data races and deadlocks. We give additional weight to operators

that have raised the fitness of the population or reduced the frequency[2] of data races or deadlocks. The weighting is designed to ensure the chance of selecting an operator is always greater than zero regardless of their performance. Separate weightings are used for data races and deadlocks. A sliding window of n generations is used to adapt the operator weighting to recent history.

5 Phase 2: Optimizing Fixes From Phase 1

ARC may introduce unnecessary synchronization during the fixing process. If a fix is found, an optional second phase begins that attempts to improve the running time of ARC by shrinking and removing unnecessary synchronization blocks. The same strategy is used from part one. A new fitness function and a subset of the TXL operators (RSAS, SHSA, etc. from Table 1) are used.

$$non-functional\ fitness(P) = \frac{worst\ score}{[sig_t \times unc(t)] + [sig_c \times unc(c)]}$$

$$where:\ unc(x) = \frac{(x_{max} - x_{min})}{x_{avg}}$$

$$sig_t = \begin{cases} t/c & if\ t > c \\ c/t & if\ c > t \end{cases}$$

$$sig_s = \begin{cases} c/t & if\ t > c \\ t/c & if\ c > t \end{cases}$$

The fitness function depends on the real time, t, required for an execution and the number of voluntary context switches made, c. Voluntary context switches is the number of times a thread voluntarily gives up control of the CPU. By minimizing unnecessary synchronization both of these values should decrease. At the beginning of phase 2 we run the correct, unoptimized program (without ConTest) a number of times to acquire the unoptimized running time and number of context switches. These values are used in the fitness function to evaluate relative improvements.

Removing and reducing synchronization runs the risk of introducing new errors into the program. Before every non-functional evaluation we need to ensure that no bugs are present. We re-run the first phase's correctness check (eg, 10 ConTest runs, then 200 more). If any deadlocks or data races are encountered the proposed optimization is rejected and this individual is reset to the previous generation. After ARC validates the proposed optimization additional runs are conducted without using ConTest to obtain the running time and voluntary context fixes.

Unlike phase 1 there is no early stopping criteria as there is no correct running time. Lower is always better. Phase 2 always uses its full allotment of generations. For this reason, running phase 2 is user-configurable. Optimization of this phase is future work.

[2] For example, if EXSB reduces the occurrence of deadlocks from 80 of 100 ConTest runs to (say) 60 of 100 runs, it will be selected more frequently in future generations to combat deadlocks.

Table 2. The set of IBM benchmark programs used to evaluate ARC.

Program	SLOC	Classes	Bug Type	Can Fix?
account	165	3	Data Race	Yes
accounts	75	2	Data Race	Yes
bubblesort2	104	2	Data Race	Yes
deadlock	109	2	Deadlock	Yes
lottery	157	2	Data Race	Yes
pingpong	143	4	Data Race	Yes
airline	93	1	Data Race	No
buffer	319	5	Data Race	No

6 Evaluation

In order to evaluate ARC's ability to repair concurrency bugs we selected 8 programs from the IBM Concurrency Benchmark [8,9,13]. We chose six programs containing bugs ARC can fix and two ARC cannot fix as a sanity check.

ARC is designed to be flexible and contains a number of configurable parameters. Table 3 describes the configuration used in our evaluation. The parameter values are influenced by community standards (e.g., evolution population, evolution generations) and through experience gained using ARC (e.g., ConTest runs, validation multiplier). Of importance, the GA¬C population size and generation size are both 30. Every member at every generation is evaluated by being run through ConTest 10 times. Any potential fix is evaluated 150 more times. Testing was conducted on a Linux PC with a 2.33 GHz processor, 4 gigabytes of RAM running Linux Mint 13.

6.1 Experimental Results

Each program in Table 2 was run through ARC 5 times using the parameters described in Table 3. Results are summarized in Table 4. ARC was able to fix the 6 fixable programs and was not able to fix the 2 non-fixable. For the 6 repairable programs the time taken to find a fix ranged from about 2 minutes to 100 minutes. The most time consuming aspect of ARC is the numerous ConTest executions. Second is the waiting necessary to determine the difference between a successful execution and a timeout caused by a deadlock. The *Timeout Multiplier* in Table 3 allows ARC to wait up to 20 times the instrumented execution time for the program to complete.

Almost all fixes are found in the first or second generation. The static analysis by Chord and the dynamic analysis by ConTest significantly shrink the state space. For example, the account program contains 3 classes, approx. 9 methods and 6 variables. After the analysis, this is reduced to 2 classes, 3 methods and 3 variables. A population of 30 may exceed the number of mutations available, leading to the search space probably being exhaustively covered. If the correct program is 1 or 2 mutation steps from the incorrect one, it should be found

Table 3. The set of parameters that ARC uses along with their descriptions and values.

Parameter	**Description**	**Value**
Project Test MB	The amount of memory allocated	2000
ConTest Runs	Test suite executions per gen. per member	10
Validation Mult.	Multiplier on ConTest runs when validating potentially correct programs	15
Timeout Mult.	Time multiplier for ConTest before timeout	20
Evolution Gen	Maximum number of generations of the GA¬C	30
Evolution Population	Population size for the GA¬C	30
Replace Lowest %	Lowest n% of population replaced in GA¬C	10
Replace With Best %	Replace under-perfomers with best individuals n% of the time	75
Replace min turns	Minimum time under-performing	3
Replace Interval	Every n generations, under-performers are replaced	5
Ranking Window	Size of sliding window for operator weighting	5
Success Weight	Fitness score for successful executions	100
Timeout Weight	Fitness score for timeout executions	50
Improv. Window	Number of generations to consider for convergence	10
Avg. Fit. Delta	Minimum average fitness improvement required	0.01
Best Fit. Delta	Minimum best fitness improvement required	1

Table 4. Summary of the results of running the programs through ARC 5 times.

Program	**Average Generations to Find Fix**	**Average Time Taken (HH:MM:SS)**
account	5.0	00:08:08
accounts	1.0	00:44:00
bubblesort2	2.2	01:40:20
deadlock	1.0	00:02:12
lottery	2.4	00:38:00
pingpong	1.0	00:12:32

quickly. ARC works as a proof of concept and must be further evaluated – more runs and on larger programs.

Threats to Validity. The main threat to validity for our experimental evaluation of ARC is external validity – our ability to generalize results. All of the programs used in our experiment are small and are not representative of large-scale concurrent software. In the future, we plan to address this threat by conducting further experiments with larger concurrent software systems.

7 Related Work

We will now discuss two areas of related research: the use of SBSE to repair sequential bugs and the existing work on healing and fixing concurrency bugs.

Sequential Bug Repair. Several approaches to sequential program repair have been proposed in the literature. For example, Arcuri and Yao as well as Wilkerson and Tauritz use co-evolutionary competition between programs with bugs (or between test cases) [2, 3, 25]. Both of these approaches require formal specifications and use genetic programming to evolve fixes.

Alternatively GenProg is another approach to sequential program repair but requires no formal specifications [24]. Instead, GenProg uses test cases to demonstrate a bug and describe the desired functionality that must be preserved. To address the limitations of the previous approach GenProg introduces two innovative features that allow the repair of real bugs in real programs: (1) It assumes the bug is written correctly in another part of the program and (2) It determines the error path on which the bug occurs and target only those statements for repair.

Concurrency Bug Repair. For concurrent programs, there are several examples of related work. One example is the use of ConTest to heal data races [15, 18]. Healing a program is not the same as repairing a program – *"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"* [18].

AFix [14] is a framework for fixing single-variable atomicity violations in C++ programs. This approach combines dynamic bug analysis, patch creation and merging and dynamic testing. One limitation of AFix is that it only considers bug ides that involve manipulating mutex locks. We can not compare AFix with ARC because AFix work only on C++ programs which ARC works only on Java programs. Our analysis of the bugs capable of being fixed by AFix and ARC indicates that ARC can be applied to a wider variety of bugs and bug combinations (i.e., programs with different kinds of concurrency bugs present) and ARC offers a wider variety of possible bug fixes. Others have evaluated AFix and reported that, *"Our evaluation of AFix on large real systems also shows that the AFix sometimes incurs the degraded performance and, worse, frequent deadlocks"* [19].

Finally, Axis [19] is another concurrency bug repair tool that uses a branch discrete control theory called supervision based on place invariants to fix any number of correlated atomicity violations with minimal harm to concurrency. The Axis approach does not appear to be able to fix deadlocks.

8 Conclusions and Future Work

In this paper we have introduced ARC, a framework to automatically repair deadlocks and data races in concurrent Java programs. The goal of ARC is not only to ensure that a concurrency bug is repair but also to maximize the performance of the program once the bug has been fixed. To achieve this goal ARC consists of two phases:

1. *Phase 1:* a bug repair phase that employs a genetic algorithm without crossover to mutate an incorrect program, searching for a variant of the original program that fixes the deadlocks and data races.
2. *Phase 2:* an optimization phase attempts to optimize the concurrent source code by removing any excess synchronization without sacrificing program correctness. Excess and unneeded synchronization may be introduced in Phase 1 and can negatively affect performance.

To evaluate ARC, we conducted experiments using a set of 8 programs from the IBM concurrency benchmark. ARC was able to fix the data races and deadlocks in all 6 of the fixable programs. Although ARC was successful with the set of programs from the IBM concurrency benchmark we still need to evaluate ARC's scalability on larger open source projects. To assist with scalability we plan to leverage some of the different heuristics for seeding noise and different optimizations supported by ConTest [16]. These optimizations will hopefully reduce the testing time required to evaluate variants of the original program that are produced during both of ARC's phases.

Finally, we plan to further investigate the mutation operators used to repair concurrency bugs in ARC. Through experimentation we plan to optimize the existing set of mutation operators to maximize their capabilities while removing unnecessary operators. We also plan to experiment with new mutation operators that will increase the set of possible bug fixes. Potential additions to our current set of mutation operators include splitting or merging synchronization blocks and adding synchronize blocks with locks not used elsewhere in the program. Furthermore, we would like to expand ARC's operators to deal with new anti-patterns [5, 10] and give ARC the ability to fix additional types of bugs.

Acknowledgment

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

1. Acree, A.T., Budd, T.A., DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Mutation analysis. Tech. rep., GIT-ICS-79/08, Georgia Institute of Technology (1979)
2. Arcuri, A.: On the automation of fixing software bugs. In: Proc. of Int. Conf. on Soft. Eng. (ICSE'08). pp. 1003–1006 (2008)
3. Arcuri, A., Yao, X.: A novel co-evolutionary approach to automatic software bug fixing. In: Proc. of IEEE Congress on Evolutionary Computation (CEC'08). pp. 162–168 (2008)
4. Bradbury, J., Cordy, J., Dingel, J.: Mutation Operators for Concurrent Java (J2SE 5.0). In: Proc. of the Work. on Mutation Analysis (Mutation'06). pp. 83–92 (2006)
5. Bradbury, J., Jalbert, K.: Defining a Catalog of Programming Anti-Patterns for Concurrent Java. In: Proc. of the Int. Work. on Software Patterns and Quality (SPAQu'09). pp. 6–11 (2009)

6. Cordy, J., Halpern, C., Promislow, E.: TXL: A rapid prototyping system for programming language dialects. In: Proc. of the Int. Conf. on Computer Languages. pp. 280–285 (1988)
7. Edelstein, O., Farchi, E., Nir, Y., Ratsaby, G., Ur, S.: Multithreaded Java program test generation. IBM Systems Journal 41(1), 111–125 (2002)
8. Eytani, Y., Tzoref, R., Ur, S.: Experience with a concurrency bugs benchmark. In: Proc. of Software Testing Benchmark Work. (TESTBENCH'08) (2008)
9. Eytani, Y., Ur, S.: Compiling a benchmark of documented multi-threaded bugs. In: Proc. of Work. on Parallel and Distributed Sys.: Testing, Analysis, and Debugging (PADTAD'04) (2004)
10. Fiedor, J., Křena, B., Letko, Z., Vojnar, T.: A uniform classification of common concurrency errors. Lecture Notes in Computer Science 2012(6927), 519–526 (2012)
11. Galletly, J.: An overview of genetic algorithms. Kybernetes 21(6), 26–30 (1992)
12. Harman, M.: Why the Virtual Nature of Software Makes it Ideal for Search Based Optimization. In: Proc. of Int. Conf. on Fundamental Approaches to Software Engineering (FASE'10). pp. 1–12 (2010)
13. Havelund, K., Stoller, S., Ur, S.: Benchmark and framework for encouraging research on multi-threaded testing tools. In: Proc. of Work. on Parallel and Distributed Sys.: Testing, Analysis, and Debugging (PADTAD'03). pp. 22–26 (2003)
14. Jin, G., et al.: Automated atomicity-violation fixing. In: Proc. of ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation (PLDI '11). pp. 389–400 (2011)
15. Krena, B., Letko, Z., Tzoref, R., Ur, S., Vojnar, T.: Healing Data Races On-The-Fly. In: Proc. of Work. on Parallel and Distributed Sys.: Testing, Analysis, and Debugging (PADTAD'07). pp. 54–64 (2007)
16. Kena, B., Letko, Z., Vojnar, T., Ur, S.: A platform for search-based testing of concurrent software. In: Proc. of Work. on Parallel and Distributed Sys.: Testing, Analysis, and Debugging (PADTAD'10). pp. 48–58 (2010)
17. Le Goues, C., Dewey-Vogt, M., Forrest, S., Weimer, W.: A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In: Proc. of Int. Conf. on Soft. Eng. (ICSE'12). pp. 3–13 (2012)
18. Letko, Z., Vojnar, T., Krena, B.: AtomRace: Data Race and Atomicity Violation Detector and Healer. In: Proc. of Work. on Parallel and Distributed Sys.: Testing, Analysis, and Debugging (PADTAD'08) (2008)
19. Liu, P., Zhang, C.: Axis: automatically fixing atomicity violations through solving control constraints. In: Proc. of Int. Conf. on Soft. Eng. (ICSE'12). pp. 299–309 (2012)
20. Long, B., Strooper, P., Wildman, L.: A method for verifying concurrent Java components based on an analysis of concurrency failures. Concurrency and Computation: Practice & Experience 19(3), 281–294 (Mar 2007)
21. Musuvathi, M., Qadeer, S., Ball, T.: CHESS: A Systematic Testing Tool for Concurrent Software. Tech. rep., Microsoft Research (2007)
22. Naik, M., Aiken, A.: Conditional must not aliasing for static race detection. In: Proc. of ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'07). pp. 327–338 (Jan 2007)
23. Sutter, H., Larus, J.: Software and the concurrency revolution. Queue 3(7), 54–62 (2005)
24. Weimer, W., et al.: Automatically finding patches using genetic programming. In: Proc. of Int. Conf. on Soft. Eng. (ICSE'09). pp. 364–374 (2009)
25. Wilkerson, J., Tauritz, D.: Coevolutionary automated software correction. In: Proc. of Genetic and Evolutionary Computation Conf. (GECCO'10). pp. 1391–1392 (2010)