# Automatic Repair of Concurrency Bugs

Jeremy S. Bradbury, Kevin Jalbert
*Software Quality Research Group*
*Faculty of Science (Computer Science)*
*University of Ontario Institute of Technology*
*Oshawa, Ontario, Canada*
{*jeremy.bradbury, kevin.jalbert*}*@uoit.ca*

***Abstract*—Bugs in concurrent software are difficult to identify and fix since they may only exhibit abnormal behaviour on certain thread interleavings. We propose the use of genetic programming to incrementally create a solution that fixes a concurrency bug automatically. Bugs in a concurrent program are fixed by iteratively mutating the program and evaluating each mutation using a fitness function that compares the mutated program with the previous version. We propose three mutation operators that can fix concurrency bugs: synchronize an unprotected shared resource, expand synchronization regions to include unprotected source code, and interchange nested lock objects.**

***Keywords*-concurrency; genetic programming; mutation.**

## I. Introduction

Concurrency bugs are difficult and expensive to fix since there are often many interleavings for a concurrent program. We propose using a search-based technique, specifically genetic programming, to apply and assess possible fixes for common concurrency bugs. In particular we are interested in fixing deadlocks and data races. A deadlock occurs in *"...a situation where two or more processes are unable to proceed because each is waiting for one of the others to do something..."* [1]. A data race occurs when *"...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"* [1].

There has been a great deal of research in the area of search-based software engineering [2]. Furthermore, the use of genetic programming to aid in identifying a solution that fixes a bug is not a novel idea [3]–[8]. Our proposed approach adapts the original idea of automatically fixing sequential software using genetic programming to specifically target concurrent software. Bugs in a concurrent program are fixed by iteratively mutating the program and evaluating each mutation using a fitness function that compares the mutated program with the previous version. We have focused on deadlock and data race bugs and have identified three mutation operators that can insert potential fixes into a concurrent program. We believe that this approach can be very successful with respect to concurrency bugs since the possible set of fixes is relatively small and the possible fixes are limited to source code locations associated with access to shared data and the protection of shared data. In the previous research using genetic programming to fix bugs in sequential programs, the general approach has proved successful with a larger set of possible actions and thus suggests that the correction of concurrent software is also feasible.

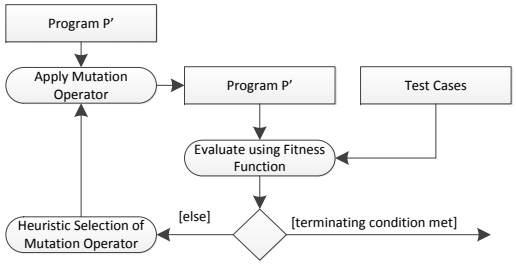


Figure 1. Automatic Repair Process

To the best of our knowledge there has been no previous work using genetic programming to patch bugs in concurrent software, However, there has also been work that involves the correction of concurrency bugs using self-healing [9]. The self-healing approach is applied dynamically and is focused on coping with a concurrency bug, while our approach evolves the source code in order to correct the bug.

## II. Our Automatic Repair Process

Our process expects a concurrent program $P$ as input (see Figure 1). A mutation operator is applied to $P$, which results in a mutated program, $P'$. We have identified three mutation operators that can correct data race and deadlock bugs. We have previously explored variations of these operators in previous work on mutation testing of concurrent software [10]. The first two operators specifically target data race bugs while the third operator targets deadlock bugs:

1) *Synchronize an unprotected shared resource.* One cause of a data race is that a shared resource is unprotected. By synchronizing around a shared resource data races can be fixed.

**Program $P$:**

```
...
obj.write( var1 );
...
```

**Program $P'$:**

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

2) *Expand synchronization regions to include unprotected source code.* Data races can sometimes be caused if the synchronization region does not fully encapsulate access to the shared resources. Expanding the synchronization region can also fix the data race.

**Program $P$:**

```
synchronized ( lock ){
  obj.write( var1 );
}
obj.write( var2 );
```

**Program $P'$:**

```
synchronized ( lock ){
  obj.write( var1 );
  obj.write( var2 );
}
```

3) *Interchange nested lock objects.* Common deadlocks occur due to the ordering of lock acquisition. By interchanging nested lock objects common deadlocks can be fixed.

**Program $P$:**

```
synchronized ( lock1 ){
  synchronized ( lock2 ){
    obj.write( var1 );
  }
}
```

**Program $P'$:**

```
synchronized ( lock2 ){
  synchronized ( lock1 ){
    obj.write( var1 );
  }
}
```

After the application of a mutation operator, the new program, $P'$, is then evaluated using our fitness function:

$$fitness(P) = \sum_{i=0}^{n} \frac{interleavings\ without\ a\ bug}{total\ \#\ of\ interleavings\ tested}$$

$$n = \#\ of\ Test\ Cases$$

Our fitness function differs from the function used by Weimer et al. in which the fitness is the weighted value of the number of tests that pass in addition to a weighted sum of the number of tests that fail. Since we are focused on concurrent programs we have to provide coverage of the interleaving space and need to run each test many times in order to provide confidence that the test will not fail for some interleaving. Therefore, we have chosen to calculate the percent of interleavings that pass for each test case and use the sum of these values as our measure of fitness. We evaluate our fitness function by running all tests with $P'$ a number of times using a concurrency testing tool (e.g., IBM's ConTest [11]) to aid in exploring different thread interleavings. If the $fitness(P') > fitness(P)$ then a given mutation is kept, otherwise it is discarded.

We next check if the process can terminate or if more mutations (i.e., bug fixes) are required. The terminating conditions are: (1) a program $P'$ has reached a user-defined threshold of success meaning that enough tests succeed with enough interleavings to provide confidence in the program correctness, (2) the process has progressed a predefined number of iterations without significant progress.

If no terminating condition is met then we need to select a mutation operator for the next iteration of the algorithm. We do not select the mutation operators randomly. Instead, we select mutations based on the bugs present in the program. For example, if most of the test cases failed due to a deadlock then selecting the third mutation operator has the best chance of fixing the deadlock otherwise we select one of the other two operators.

For simplicity, our process was described with one mutation per iteration. Ideally, we can mutate the program to generate and evaluate multiple mutants per iteration in order to find a solution more quickly.

## III. Conclusion & Future Work

Genetic programming has been used to fix sequential programs [3]–[5], [7], [8] and additional work has focused on refinement of this approach [6]. Our research is an adaptation of this previous work that targets concurrent programs with deadlock and data race bugs. Currently, we are in the process of implementing our work. After the implementation is complete we plan to evaluate and refine many aspects of our approach including: the mutation operators, the fitness function, the termination threshold and the heuristic selection of mutation operators.

## References


[1] B. Long, P. Strooper, and L. Wildman, "A method for verifying concurrent Java components based on an analysis of concurrency failures," *Concurr. Comput.: Pract. Exp.*, vol. 19, no. 3, pp. 281–294, 2007.

[2] M. Harman, "Why the virtual nature of software makes it ideal for search based optimization," in *Proc. of FASE*, 2010, pp. 1–12.

[3] S. Forrest, T. Nguyen, W. Weimer, and C. L. Goues, "A genetic programming approach to automated software repair," in *Proc. of GECCO*, 2009, pp. 947–954.

[4] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *Proc. of CEC*, 2008, pp. 162–168.

[5] A. Arcuri, "On the automation of fixing software bugs," in *Proc. of ICSE*, 2008, pp. 1003–1006.

[6] J. L. Wilkerson and D. Tauritz, "Coevolutionary automated software correction," in *Proc. of GECCO*, 2010, pp. 1391–1392.

[7] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proc. of ICSE*, 2009, pp. 364–374.

[8] W. Weimer, S. Forrest, C. L. Goues, and T. Nguyen, "Automatic program repair with evolutionary computation," *CACM*, vol. 53, no. 5, May 2010.

[9] Z. Letko, T. Vojnar, and B. Krena, "AtomRace: Data race and atomicity violation detector and healer," in *Proc. of PADTAD*, 2008, pp. 1–10.

[10] J. S. Bradbury, J. R. Cordy, and J. Dingel, "Mutation operators for concurrent Java (J2SE 5.0)," in *Proc. of Mutation*, 2006, pp. 83–92.

[11] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur, "Multithreaded Java program test generation." *IBM Systems Journal*, vol. 41, no. 1, pp. 111–125, 2002.