A Tool for Automatically Repairing Concurrency Bugs

Jeremy S. Bradbury ● Kevin Jalbert

Faculty of Science ● University of Ontario Institute of Technology ● Oshawa, Ontario, Canada

{jeremy.bradbury, kevin.jalbert}@uoit.ca

UOIT
CHALLENGE INNOVATE CONNECT

1. Motivation

- It is challenging to develop high quality concurrent software
  - Multiple threads of execution can be interleaved in many different ways
  - Testing concurrent software is tedious since concurrency bugs appear intermittently
  - When a concurrency bug is detected, it can be unclear on how to actually fix the bug
- **Related work** has used genetic programming to automatically repair bugs in sequential code [1], [2]

Research Goal: Extend related work and automatically repair concurrency bugs (data race and deadlock bugs) and improve performance using genetic programming.

2. Approach

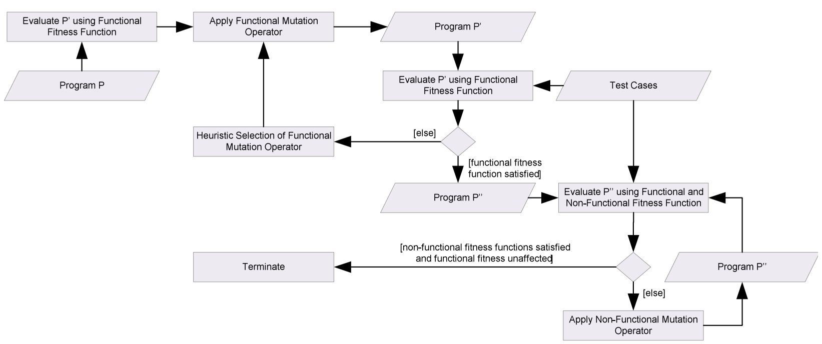

Figure 1: Overview of the Automatic Repairing Process

3. Evaluating Fitness

- Fitness is defined as follows:

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

$$non-functional\ fitness(P) = \sum_{i=0}^{n} \frac{average\ CPU\ time}{interleavings\ tested};\ n = \#\ of\ Test\ Cases$$

- If the new program *P'* has a higher functional fitness then program *P* then we keep it, otherwise *P'* is discarded
- Proper evaluation of the program is achieved by running test cases multiple times using the testing tool **ConTest** [3]
  - ConTest instruments a concurrent program with random thread delay, thus increasing the likelihood that each program execution will explore a different thread interleaving due to the new thread schedule
- After the functional fitness is satisfied then the satisfying the non-functional fitness can begin. The goal is to minimize the average CPU time taken for thread interleavings without lowering the functional fitness

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

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

[3] IBM's ConTest website (http://www.haifa.ibm.com/projects/verification/contest/index.html)

[4] J.S. Bradbury, K. Jalbert. "Automatic repair of concurrency bugs," in *Proc. of SSBSE 2010 - Fast Abstracts*, Sept. 2010.

4. Functional Mutation Operators

- To ensure that the mutating program executes correctly in accordance with respect to the test cases a set of functional mutation operators are needed
- These mutation operators will negatively effect the non-functional requirements (performance)
- The following mutation operators aim to improve the functional requirements (correctness):

1. Synchronize an unprotected shared resource

- One cause of a data race is that a shared resource is un-protected. By synchronizing around a shared resource data races may be fixed.

```
…
obj.write ( var1 );
…
```
→
```
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 may also fix the data race.

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

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

5. Non-Functional Mutation Operators

- To ensure that the mutated program executes efficiently with respect to the overhead of synchronizations, a set of non-functional mutation operators are needed
- These mutation operators can negatively effect the functional requirements
- The following mutation operators aim to improve the non-functional requirements:

1. Remove unnecessary synchronization regions

- Synchronization regions create overhead due the time required in acquiring/releasing the lock and delays due to waiting for the lock. Removing unnecessary synchronization regions will improve performance.

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

2. Shrink synchronization regions

- Reducing the number of statements encapsulated in a synchronization region will allow the lock to be released quicker. The less time a thread holds the lock the less thread contention will exist, thus improving performance.

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

6. Conclusions & Future Work

- Preliminary prototype is capable of automatically mutating a concurrent program to improve the fitness
  - Functional and non-functional improvements are achieve by using the mutation operators
- Our approach for automatically repairing concurrent software [4] requires further evaluation to assess the ability of the approach to fix real concurrency bugs
- Future work includes:
  - Completion of the implementation
  - Evaluation and optimization of the overall approach, the mutation operators and the fitness function

CASCON 2010 Technology Showcase ● Markham, Ontario, Canada ● November 2010

SQR GROUP

Funding provided by: NSERC CRSNG