Automatically Repairing Concurrency Bugs with ARC

MUSEPAT 2013 • Saint Petersburg, Russia

David Kelk, Kevin Jalbert, **Jeremy S. Bradbury**

Faculty of Science (Computer Science)
University of Ontario Institute of Technology
Oshawa, Ontario, Canada

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

http://www.sqrlab.ca

SQR LAB

UNIVERSITY OF ONTARIO INSTITUTE OF TECHNOLOGY

Funding provided by:

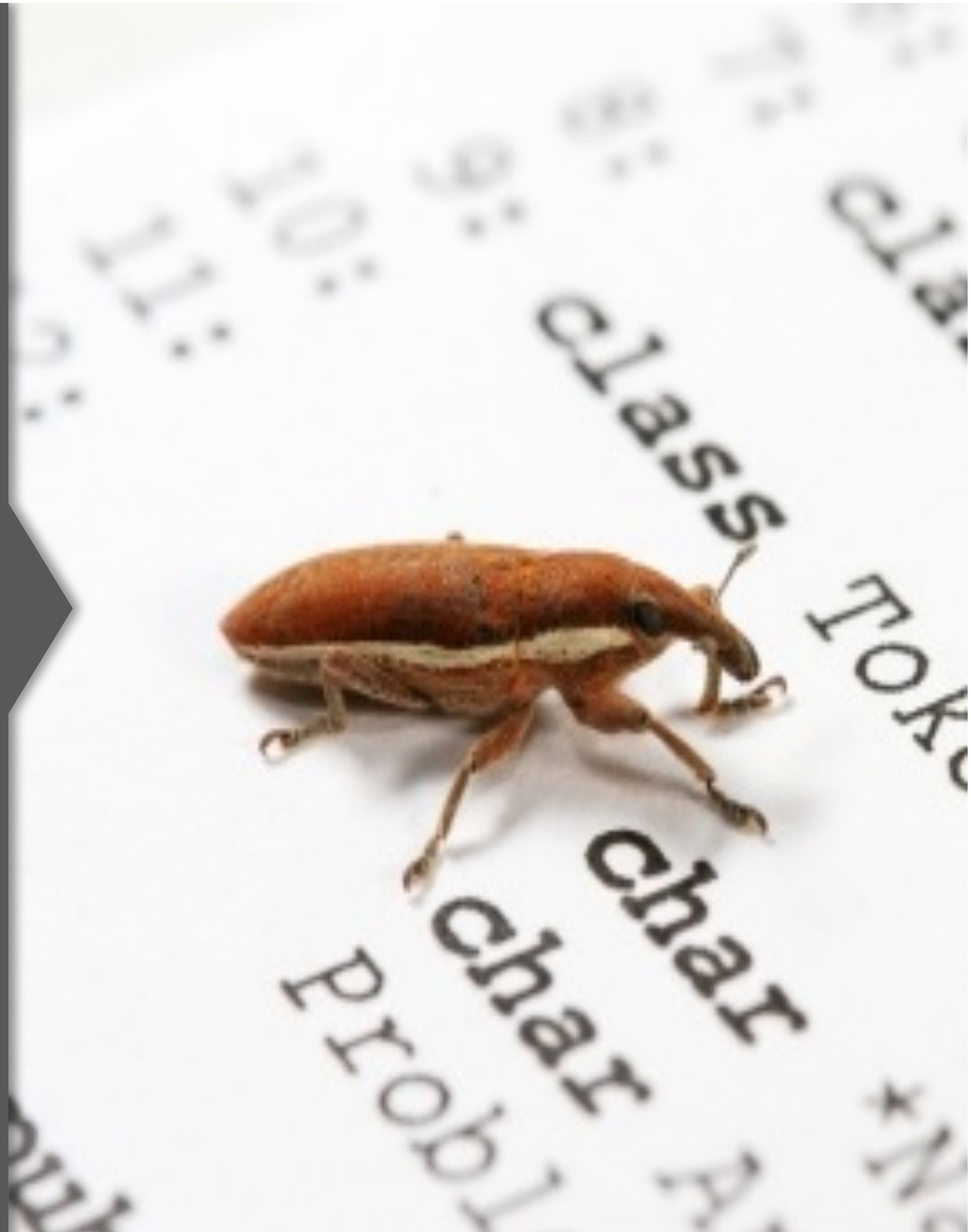
NSERC CRSNG

What do we mean by concurrency bugs?

- There are many different kinds of concurrency bugs
- We focus on two of the most common kinds – data races and deadlocks

What do we mean by bug repair?

- We view bug repair as a source code modification that fixes a concurrency bug while minimizing the effect on performance



How do we approach bug repair?

- We use Search-Based Software Engineering (SBSE)
- Many software engineering problems can be expressed as optimization problems
- SBSE positions these problems in a search-based context
  - Applies meta-heuristic optimization techniques

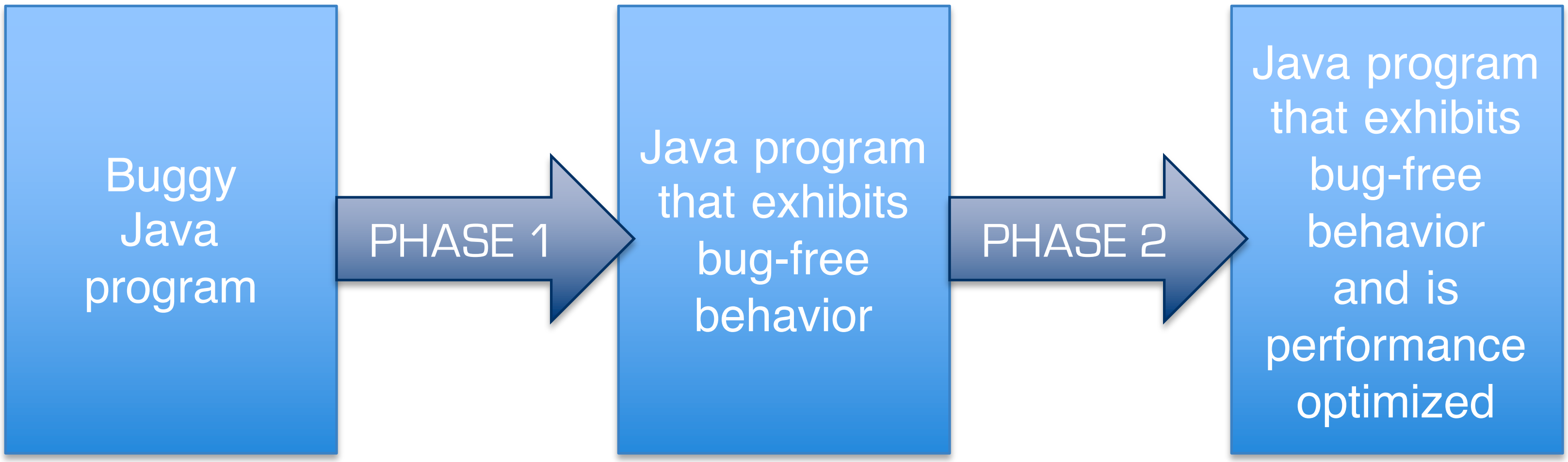
Example SBSE techniques include hill climbing, particle swarm optimizations, genetic algorithms (GAs)...

Automatic Repair of Concurrency Bugs

- Several SBSE approaches have been proposed to fix bugs in single threaded programs [LDFW12, Arc11]
  - genetic programming is used to evolve patches, while testing evaluates fitness
- These techniques cannot be applied directly to fix concurrency bugs due to the nondeterministic nature of thread scheduling
- We adapt this work to handle concurrency bugs by modifying the fitness function and it's evaluation

[LDFW12] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in Proc. of ICSE 2012, Jun. 2012.
[Arc11] A. Arcuri, "Evolutionary repair of faulty software," in Applied Soft. Computing, vol. 11, 2011, pp. 3494–3514.

ARC – Automatic Repair of Concurrency



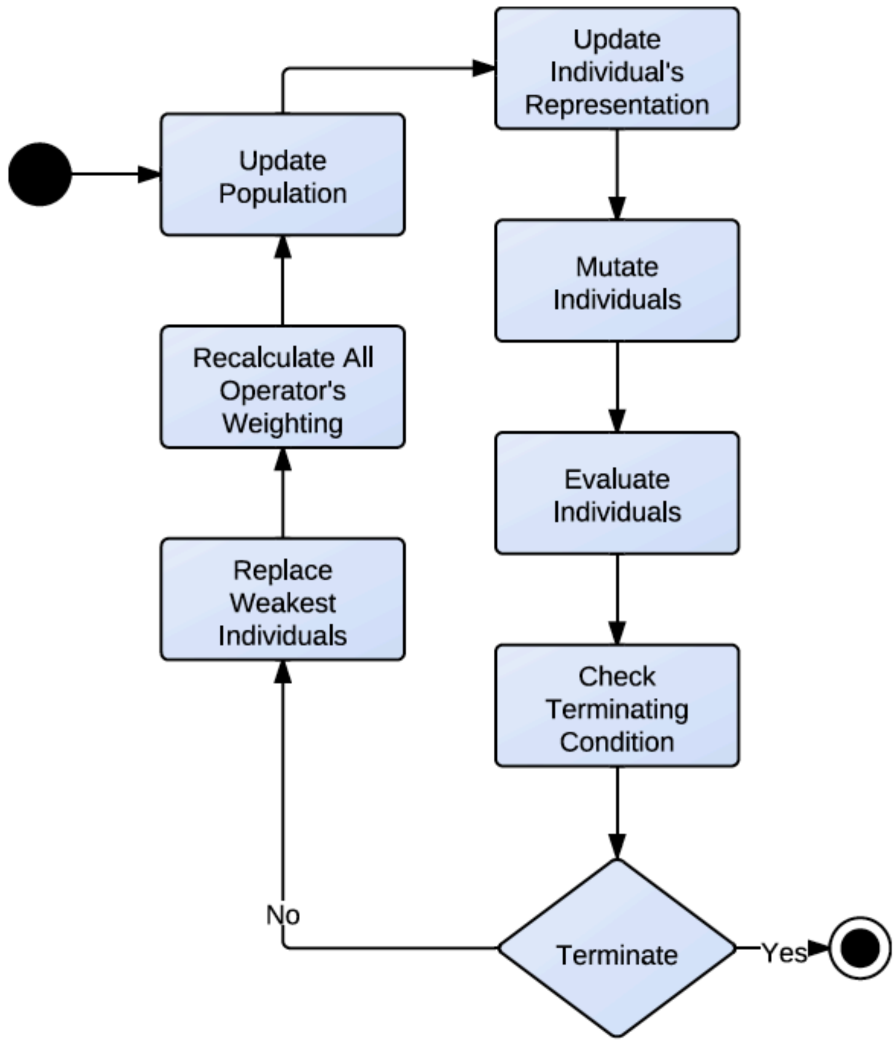
PHASE 1:

- Repairing Deadlocks and Data Races

PHASE 2:

- Optimizing the Performance of Repaired Source Code

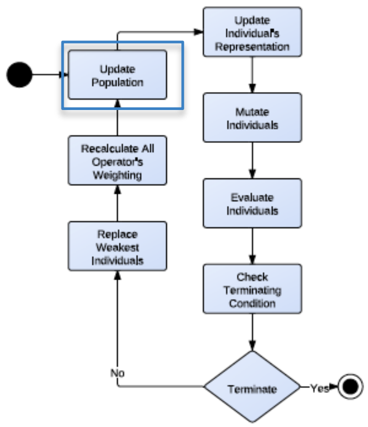
ARC PHASE 1
Repairing Deadlocks and Data Races



INPUT

1. Java program with concurrency bugs
2. Set of JUnit tests
    - The test suite is the oracle (hence, the approach is only as good as the tests!)

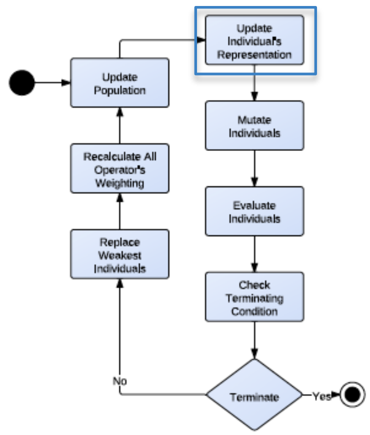
ARC PHASE 1
Repairing Deadlocks and Data Races



1. Update Population

- Create the population for the genetic algorithm (GA)
- The first generation is a set of copies of the original buggy program
- Subsequent generations will be updated based on the GA (described in future steps)

ARC PHASE 1
Repairing Deadlocks and Data Races



2. Generate mutants

- Use mutation operators to generate mutants for all members of the population
- The generated mutants are optimized using the static analysis tool Chord [NA07]
  - Allows mutation operators to target specific shared classes, methods and variables when generating mutants

[NA07] 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 2007)*, pp. 327–338, Jan. 2007.

Mutation Operator[1] 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

[1] All mutation operators are written in the TXL source transformation language – http://www.txl.ca.

Mutation 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

Program P:

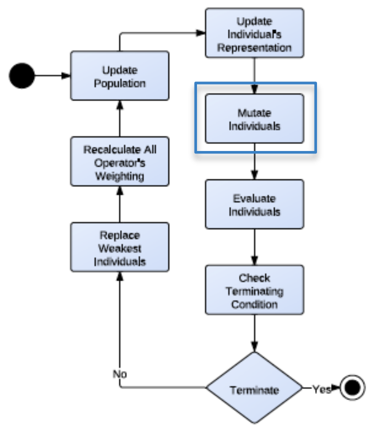
```
obj.write(var1);
synchronized(lock) {
    myHash.remove(var1);
}
```

EXSB

Program P':

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

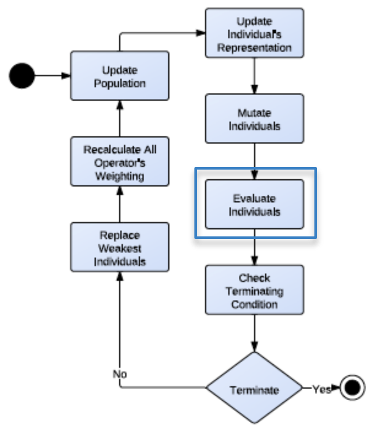
ARC PHASE 1
Repairing Deadlocks and Data Races



3. Apply mutation to an individual in population

- During execution the GA selects a type of mutation (i.e., a mutation operator) – random on first generation
- From the set of mutations created a random instance is used

ARC PHASE 1
Repairing Deadlocks and Data Races



4. Evaluate individuals

- Fitness function is used to evaluate mutants selected in previous step

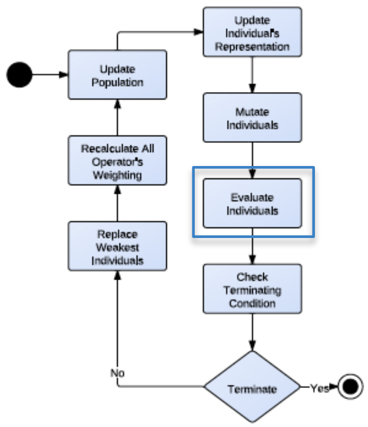
functional fitness(P) = (*s* x *sw*) + (*t* x *tw*)

where:

s = # of successful executions
sw = success weighting
t = # of timeout executions
tw = timeout weighting

- The mutated individual is maintained only if the fitness function is improved

ARC PHASE 1
Repairing Deadlocks and Data Races



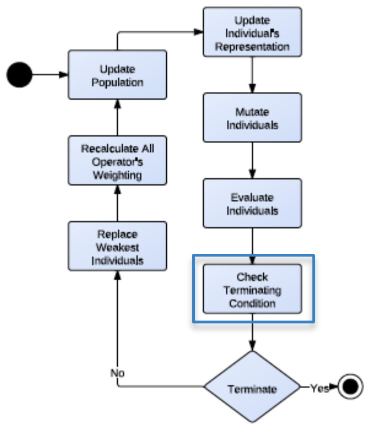
4. Evaluate individuals

- In order to evaluate the fitness function for a given individual we need to evaluate the function over many different interleavings/executions
- We use IBM's ConTest [EFN+02], which instruments the program with noise, to ensure that many interleavings are evaluated

[EFN+02] Edelstein, O., Farchi, E., Nir, Y., Ratsaby, G., Ur, S.: Multithreaded Java program test generation. IBM Systems Journal 41(1), 111–125 (2002)

ARC PHASE 1

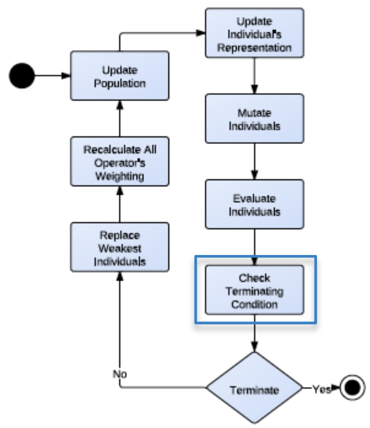
Repairing Deadlocks and Data Races



4. Check terminating condition

- An individual that produces 100% successful executions is a potential fix
- However, we perform an additional step to increase confidence that the individual is in fact correct
    - We evaluate the individual with ConTest again using a safety multiplier (e.g., 20) to increase the number of interleavings explored – a fix is only accepted if we achieved 100% success for this additional evaluation

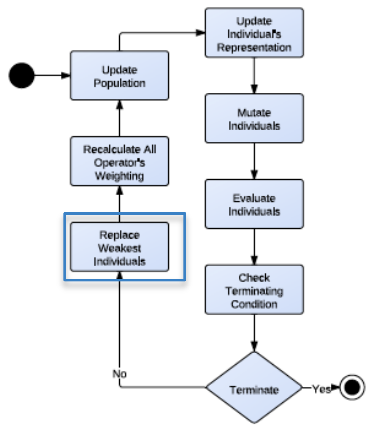
ARC PHASE 1
Repairing Deadlocks and Data Races



4. Check terminating condition

- Our approach will not always find a successful solution that repairs all of the concurrency bugs in a program
- If after a user-defined number of generations a solution is not reached our algorithm will terminate

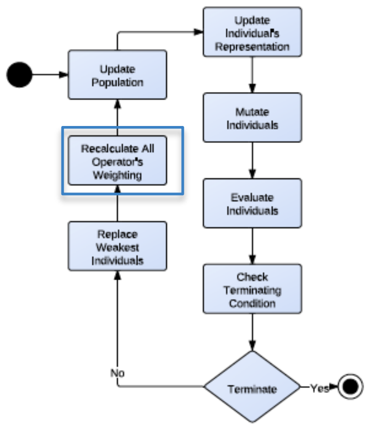
ARC PHASE 1
Repairing Deadlocks and Data Races



5. Replace Weakest Individuals *(Optional)*

- To encourage individuals to explore more fruitful areas of state space we can replace individuals
  - We can restart with original population
  - We can replace (e.g., 10%) of underperforming individuals with random high-performance individuals or with original program

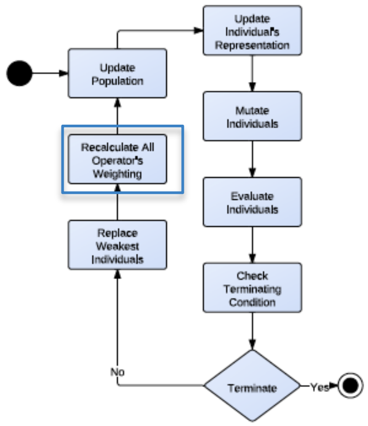
ARC PHASE 1
Repairing Deadlocks and Data Races



6. Calculate Operator Weighting

- We leverage historic information from previous generations to weight the operators and increase the likelihood that useful operators are selected first/more frequently
- Strategy 1: weight based on % of dead locks/data races uncovered
- Strategy 2: Weight based on a mutation operator's fitness function success

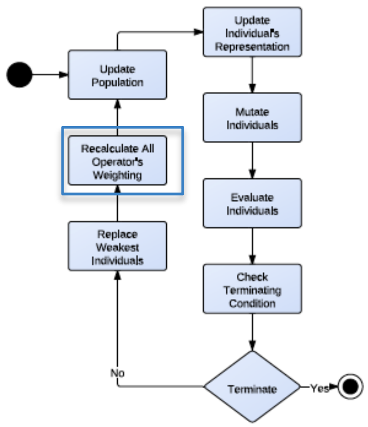
ARC PHASE 1
Repairing Deadlocks and Data Races



6. Calculate Operator Weighting

- Strategy 1: weight based on % of dead locks/data races uncovered
  - For example, some operators are geared towards fixing deadlocks, others data races and some both.
  - We increase the likelihood that specific operators are selected based on the number of deadlock and data races in our historic evaluations

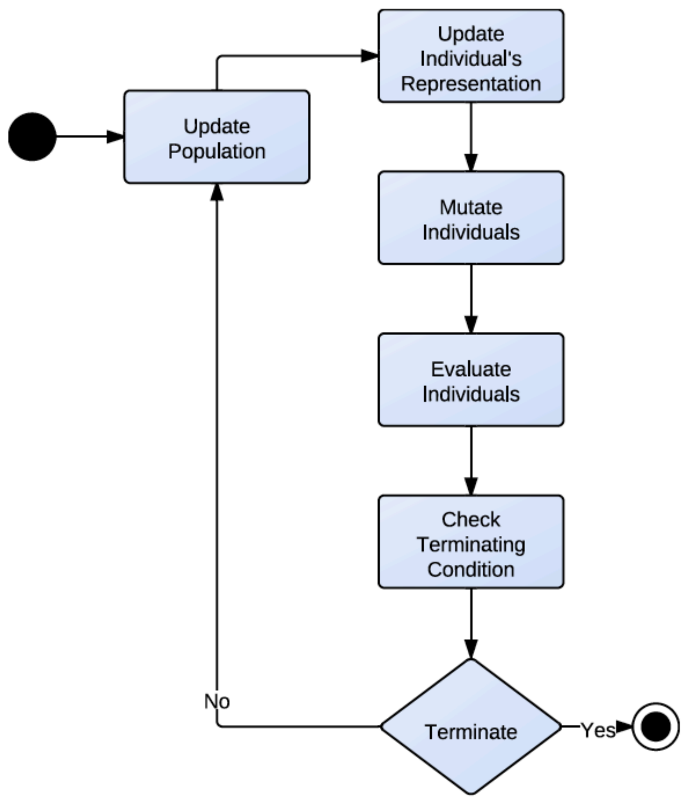
ARC PHASE 1
Repairing Deadlocks and Data Races



6. Calculate Operator Weighting

- Strategy 2: Weight based on a mutation operator's fitness function success
  - For example, operators that have historically increased the fitness are weighted proportional to their success

ARC PHASE 2
Optimizing Repaired Source Code



- ARC may introduce unnecessary synchronization during Phase 1.
- If Phase 1 is successful, an *optional* second phase attempts to improve the running time of the program-under-repair by shrinking and removing unnecessary synchronization blocks
- A new non-functional fitness function and a subset of the TXL operators (e.g., RSAS, SHSA) are used

Evaluation – Setup

- We selected a set of 8 programs from the IBM Concurrency Benchmark [EU04] that have deadlock or data race bugs
  - 6 programs that exhibit bugs ARC was designed to fix
  - 2 programs that ARC was not designed to fix (sanity check)
- Each program was analyzed using 5 executions of ARC

[EU04] 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 2004)*, 2004.

Evaluation – Results[1]

Program	Bug Type	Bug Repaired?	# Generations to Repair Bug (Avg.)	Time Required to Repair Bug (Avg.)
account	Data Race	✔	5.0	08m 08s
accounts	Data Race	✔	1.0	44m 00s
bubblesort2	Data Race	✔	2.2	1h 40m 20s
deadlock	Deadlock	✔	1.0	02m 12s
lottery	Data Race	✔	2.4	38m 00s
pingpong	Data Race	✔	1.0	12m 32s
airline	Data Race	✖	-	-
buffer	Data Race	✖	-	-

[1]Our evaluation was conducted on a Linux PC with a 2.33 GHz processor, 4 gigabytes of RAM running Linux Mint 13.

Related Work

- The use of ConTest to heal data races [LVK08] – healing influences the schedule and does not repair the source code
- AFix [JSZ+11] fixes single-variable atomicity violations in C++ programs – combines dynamic bug analysis, patch creation and merging and dynamic testing
- Axis [LZ12] uses branch discrete control theory and invariants to fix any number of correlated atomicity violations

[LVK08] 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 2008)*, 2008.

[JSZ+11] Jin, Guoliang, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblital: Automated atomicity-violation fixing. In: *Proc. of ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation (PLDI 2011)*, pp. 389–400, 2011.

[LZ12] Liu, P., Zhang, C.: Axis: automatically fixing atomicity violations through solving control constraints. In: *Proc. of Int. Conf. on Soft. Eng. (ICSE 2012)*, pp. 299–309, 2012.

Conclusions

- We automatically repair deadlocks and data races in concurrent Java programs using ARC
  - The goal of ARC is not only to ensure that a concurrency bug is repaired but also to optimize the performance of the program once the bug has been fixed
- 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

Challenges & Future Work

- Scalability – Although ARC was successful with the IBM Concurrency Benchmark programs we still need to evaluate ARC on larger projects
- Performance – The use of testing with noise making to evaluate the variants of programs produced during ARC is costly
    - further heuristics and optimizations need to be explored

Challenges & Future Work

- Flexibility – ARC is currently only capable of fixing deadlocks and data races
  - We place to explore other mutation operators that will increase the kinds of bugs that can be fixed
- Readability [FLW12] – automatic repair always has the potential to decrease the readability and maintainability of the source code
  - We have not studied the readability of the fixes produced by ARC

[FLW12] Zachary P. Fry, Bryan Landau, and Westley Weimer. "A Human Study of Patch Maintainability." In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*, 177–187, 2012.

Automatically Repairing Concurrency Bugs with ARC

MUSEPAT 2013 • Saint Petersburg, Russia

David Kelk, Kevin Jalbert, **Jeremy S. Bradbury**

Faculty of Science (Computer Science)
University of Ontario Institute of Technology
Oshawa, Ontario, Canada

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

http://www.sqrlab.ca

SQR LAB

UNIVERSITY OF ONTARIO INSTITUTE OF TECHNOLOGY

Funding provided by:

NSERC CRSNG