CORE: A Framework for the Automatic Repair of Concurrency Bugs

by

David Kelk

A thesis submitted in partial fulfilment of
the requirements for the degree of

Doctor of Philosophy

in

Computer Science

University of Ontario Institute of Technology

Supervisors: Dr. Jeremy Bradbury and Dr. Mark Green

January, 2015

Acknowledgements

Dr. Jeremy Bradbury and Dr. Mark Green, my supervisors, were key to seeing this thesis through to completion. They nurtured and helped develop CORE from an idea to a working version though a course project and then a published paper at MUSEPAT'13. Their insights and feedback kept this thesis on course and on time. Dr. Bradbury's work on mutation and familiarity with TXL helped me through the toughest parts of the project - the mutation operators. Dr. Green made suggestions large and small (*spell check the document* and *put the axis titles on the graph* saved me from a great deal of embarrassment), all of which were important.

My office mates and friends helped me get through graduate school. Kevin Jalbert introduced me to GitHub, Python and many of the other tools used in CORE. Richard Drake shared my interest in non-mainstream movies.

For transforming my distaste for highschool into a love of universities I will be eternally grateful to Paul Delaney and Dr. Michael Derobertis. Its been a long and twisted road - but I made it.

Beverly Myatt from CSD was instrumental in helping me navigate the difficulties of TAing courses. Bethesda Software deserves a special mention for many long, sleepless nights.

As always, my mother, Karen McLean. She always listened with interest when I babbled on about my thesis. I couldn't have done it without you.

Abstract

Desktop computers now contain 2, 4 or even 8 processors. To benefit from them programs must be written to work in parallel. If writing good code is hard, writing good parallel code is much harder. Parallelization adds process communication and synchronization to the list of difficulties faced by programmers. It also adds new kinds of bugs not found in single-threaded code such as deadlocks and data races.

In this thesis we develop the CORE (COncurrent REpair) framework. It automatically fixes deadlocks and data races in parallel Java programs. It uses a search-based software engineering approach to mutate and evolve the source code. In these mutants synchronization blocks are added, removed, expanded, shrunk or the synchronization variable is changed. Each potential fix is model checked or run through a thread noising tool that forces different thread interleavings to be explored.

Efficiently fixing data races and deadlocks in parallel Java programs is realized by combining two techniques. First, different forms of static and dynamic analyses are brought together to constrain the search space. Second, a genetic algorithm without crossover was implemented that uses both noising and model checking to determine fitness. These techniques are unified in the CORE framework. Different kinds of analysis better constrain the search space of the problem. Intelligent use of noising, model checking and incremental model checking are combined efficiently into a modern framework that help to increase the overall quality of concurrent software.

This thesis created three projects within the CORE framework, ARC-OPT, CORE-MC and CORE-IMC. First, static analysis from Chord and dynamic analysis from ConTest with fitness evaluation by thread noising from ConTest were combined in ARC-OPT. Second, JPF was integrated into the framework to analyze the source. Fitness was evaluated by JPF and ConTest. This version was called CORE-MC. Third, function header scanning for in-scope locks and incremental modelling support was added to CORE-MC to create CORE-IMC. Each project builds upon the previous and each was evaluated against a suite of test programs.

Co-Authorship

ARC (Automatic Repair of Concurrent programs) began as a course project developed by the author and Kevin Jalbert. It cumulated in a paper co-authored by the author, Kevin Jalbert and our supervisor, Jeremy Bradbury and published in the proceedings of the 1st International Conference on Multicore Software Engineering, Performance, and Tools (MUSEPAT 2013) [43]. This paper forms the basis of Chapter 4.

ARC was prior work and is not a contribution of this thesis. Once this thesis was decided on, two months of effort was put into improving and optimizing the code to produce ARC-OPT and the CORE framework. What distinguishes ARC-OPT was the addition of static analysis of the code to be fixed. It is the sole work of the author and the first contribution of this thesis.

Using search-based software engineering to enhance model checking applications was advocated for in a paper co-authored with my supervisors Jeremy Bradbury and Mark Green. It was published in the proceedings of the 1st International Workshop on Combining Modelling and Search-Based Software Engineering (CMSBSE 2013) [11].

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Summary

Over the last five years rapid progress has been made in the field of automatically fixing bugs in sequential software programs [4, 5, 7, 29, 34, 49–51, 63, 74, 87–89, 91]. Equal progress hasn't been made on the automatic repair of deadlocks and data races in concurrent software programs. Numerous techniques exist to find concurrency bugs [38, 39, 42, 60–62, 64–66, 68, 70, 73, 85] and techniques exist to try and suppress them [12, 14, 45, 53, 58, 59, 83, 86, 92]. However, only a few try to fix them [8, 12, 40, 41, 54, 56, 57] and when they do, they have limitations. Some [8, 12] are limited to finite state machines like circuit design and communications protocols. Most [40, 41, 56, 57] fix only a subset of concurrency problems such as data races and atomicity violations. Only one [54] attempts to fix deadlocks. No techniques exist to automatically and completely fix the broad class of all kinds of data races and deadlocks in a consistently reliable way.

Automatic Repair of Concurrency Bugs (ARC) [43] is a program developed by Dr. Jeremy Bradbury, Kevin Jalbert and the author. It uses a genetic algorithm [90]

Table 1.1: As the programs become larger and the bugs harder to find, the running time of ARC grows unacceptably long.

Scenario	**Pop.**	**Gen.**	**ConTest Runs**	**ConTest Run Time**	**Total Time**
Easy to find bug in a toy program	20	30	3	3 sec.	90 min.
Moderately hard to find bug in a moderate size program	30	50	10	1 min.	10.4 days
Hard to find bug in a toy program	30	50	1500	2 sec.	52 days

without crossover, GA¬C, to evolve fixes for deadlocks and data races in concurrent Java programs. Source code is mutated by adding, removing, growing, shrinking and reordering Java's *synchronize*() blocks. No other code structures are affected. State space explosion is constrained by only targeting the concurrently used classes and variables found by the ConTest [45] thread noising tool. ARC uses ConTest's thread interleaving randomization to repeatedly explore different thread interleavings and assign a fitness score to every mutant program. ConTest is run a set number of times on each mutant program to explore the concurrent state space. Choosing the number of ConTest runs requires some experience as it must be large enough to regularly and reliably find the data race or deadlock.

ARC's running time is proportional to the population P of the GA¬C, times the number of generations N, times the number of ConTest runs per member per generation CR, times the running time of ConTest CT: $RunTime = O(P \times N \times CR \times CT)$. Table 1.1 shows the running time of ARC for three different scenarios[1].

The running time of ConTest is the largest contributor to the running time of ARC. All of the additional work ARC does (copying source, compiling source, calculating

[1]It takes 4 days to run the unit tests for Python 3.

fitness, etc) is at most the same order of magnitude as the running time of ConTest, but usually of a smaller magnitude. Consider that code is copied once, compiled once and then is run through ConTest CN times for every member for every generation. Values in Table 1.1 reflect the time needed for ARC to run, using all generations. When the running time reaches into days[2], one must consider the trade-offs involved in parameter selection very carefully.

ARC has weaknesses that need to be addressed. First, the search space needs to be better constrained. The more information we have about the classes, methods and variables used concurrently leads to smaller search spaces and faster running times. Second, the incomplete exploration of all possible thread interleavings by running ConTest introduces uncertainty. If the buggy interleaving wasn't run, a bug escapes detection. The StringBuffer test program is an example in which ConTest didn't find the data race after noising the program 1,000 times. If ARC doesn't find a race or deadlock it declares the program being noised as correct and then ends. ARC declares the buggy program 'fixed'. Can we tolerate ARC missing a data race or deadlock in a 'fixed' program? Even if every proposed fix is rigorously tested[3], how do we know it is error free?

This thesis walks the threefold path to address these problems. ARC [43] was a research prototype. It worked, but it was slow and inelegant. The first excursion fixed the deficiencies in ARC[4] and added the static analysis of the program being fixed by Chord, leading to the first contribution, optimized ARC (ARC-OPT). At the same

[2]ARC usually breaks on the ConTest noising when the number of ConTest runs, CR, goes above approx. 1000 per member.

[3]Every proposed solution is executed by ConTest $V * CN$ more times, to try and validate the proposed fix.

[4]ARC is as it appears in [43]. It is joint work completed by the author and Kevin Jalbert under Dr. Bradburys supervision. The check-in on May 7, 2012 in the GitHub repository is the ARC referenced herein. It is also the final check-in for ARC. Development on ARC-OPT and the CORE framework began after this date.

time ARC-OPT was refactored to more readily interface with other programs, was cleaned up, fixed and optimized[5].

In the second stage, ARC-OPT was leveraged and evolved into CORE-MC (Model Checker). ConTest was replaced by the Java Pathfinder (JPF) [84] model checker [22]. Model checking is used to determine if a proposed fix truly eliminates the deadlocks and data races. Exhaustive model-checking of proposed solutions generated by CORE-MC provides certainty about results; a data race exists, a deadlock exists, or there are no data races and no deadlocks. At the same time, JPF provides information to better constrain the search space by returning the classes, methods and variables it found were used concurrently. After every execution of JPF, the output generated is scanned for any new classes, methods and variables found. They are added to the lists maintained by CORE-MC.

Initially JPF was to replace ConTest as the evaluation engine of each mutant program. Model checking is slow and had a devastating effect on the speed of CORE-MC. The state space explosion problem was also an issue. JPF would often crash with an 'out of memory' error or simply fail because the target program had more than 128 threads[6]. A hybrid approach was adopted in which JPF was run for a limited time and to a limited search depth. If JPF found a bug it assigned fitness and moved on. If it failed for any reason, CORE-MC fell back on ConTest to noise the program. If JPF found no bugs to the given depth, the mutant could have potentially fixed the bug(s). ConTest was again called upon to validate it.

In the third stage, incremental modelling techniques [48,81] were added to CORE-MC to create CORE-IMC. In incremental modelling the results of a model-checking run are recorded and used to speed up future runs. That is, the search tree from the

[5]When completed, the average running time of the test suite was reduced from 34 minutes in ARC to 13 minutes in ARC-OPT.

[6]The version of JPF used in this thesis has a hard-coded limit of 128 threads.

previous model checking run is loaded from disk and used by the model-checker to speed up the current run. A more in-depth description can be found in Section 2.9.3. At the same time, the scanning of function headers found to be used concurrently was added to CORE-IMC. We found that the analysis tools would identify a method used concurrently but would not identify any in-scope variables for that method that were usable as locks in the synchronize statements. Function headers were scanned for all non-primitive variables. These were added to the lists maintained by CORE-IMC as in-scope and valid lock candidates.

Note that the techniques added are cumulative, with ARC coming first and CORE-IMC last. At each stage, the resulting software (ARC, ARC-OPT, CORE-MC and CORE-IMC) was evaluated against a test suite and compared to the other stages in the thesis.

Similar to GenProg, the CORE framework is a pragmatic software-engineering and heuristic search based approach to fixing data races and deadlocks, not a theoretical one. Genetic algorithms are often used when deterministic algorithms aren't known - as in this case. There are no deterministic algorithms that fix data races and deadlocks in concurrent Java programs. CORE isn't guaranteed to find a fix if one exists and it isn't guaranteed to find the best fix. When genetic algorithms are used, a fix that is *good enough* is better than no fix at all.

CORE can introduce data races or deadlocks into parts of the code not protected by test cases. Conversely, CORE can fix unknown data races and deadlocks in code covered by test cases. This occurred in the AccountSubType test program (Section 4.5). In practice, CORE found fixes for all fixable test programs and usually found them in the first generation. See Section 4.3.1 for details. To the best of our knowledge, the fixes found by the CORE framework didn't introduce any new deadlocks or data races.

Note that initially the author believed the major contribution of this thesis was the genetic algorithm. That is, the part of the framework that evaluates every mutant program and assigns fitness. I no longer believe this to be the case. Instead, the most important part of the framework is the combination of the different analyses that constrain the search space. This has a major impact on determining the difficulty of getting from 'here' (a buggy program) to 'there' (a fixed program.)

The rest of this chapter is structured as follows. Section 1.2 states the thesis, defines terms and outlines the goals of the work. It is followed by the limitations and key assumptions in section 1.2.1. Motivation is described in section 1.3 and is followed by the thesis contribution in 1.4. Finally, the organization of the rest of this thesis is described in section1.5.

1.2 Thesis Statement and Scope of Research

Thesis statement: Efficiently fixing data races and deadlocks in parallel Java programs is realized by combining two techniques. First, different forms of static and dynamic analyses are used to constrain the search space. Second, a genetic algorithm without crossover is implemented that uses both noising and model checking to determine fitness. These techniques are brought together in the CORE framework. Different kinds of analysis better constrain the search space of the problem. Intelligent use of noising, model checking and incremental model checking are combined efficiently into a modern framework that helps to increase the overall quality of concurrent Java software.

Recall that ARC uses ConTest to both noise the program to be fixed and to find classes, methods and variables used concurrently. This thesis created three projects

within the CORE framework. First, the bugs in ARC were fixed and ARC was optimized. It was then augmented with static analysis from Chord to create optimized ARC, ARC-OPT. Second, we added JPF to both model check mutant programs and to find additional classes, methods and variables used concurrently. This portion was called the CORE model checker, CORE-MC. Third, we added incremental modelling support and the scanning of function headers for lockable variables to CORE-MC, creating CORE incremental model checker, CORE-IMC. The addition of different analysis tools and techniques (ConTest, Chord, JPF and function header scanning) in each step constrain the search space. Incremental modelling is faster than fully model checking each candidate solution in every generation. Finally, an empirical evaluation was performed at every stage.

A *formal software analysis* is "A mathematically well-founded automated technique for reasoning about the semantics of software with respect to a precise specification of intended behaviour for which the sources of unsoundness are defined [22]."

Model checking "Takes as input a state transition system model M representing a system S's behaviour, and a property P to be checked against the system, and then exhaustively explores all paths through M while checking that P is true at each reachable state. In concurrent systems, this exhaustive exploration of paths considers all possible interleavings of concurrent transitions [22]."

Bugs in software are errors causing it to behave incorrectly. They can come from incorrect or incomplete specifications, design or coding.

Bug repair is the correction of a software bug to bring the program into agreement with its expected output, design and/or specifications.

Genetic Algorithms are part of the family of nature-inspired, evolutionary, heuristic search techniques. They are population based and contain mutation and crossover.

A fitness function provides a score for each proposed solution. More fit (higher scoring) solutions are preferentially passed on to the next generation. This process continues until a solution is found, a certain number of generations pass or a predetermined number of fitness evaluations is exhausted.

An *Evolutionary fitness function* determines the value of a proposed solution to the problem at hand. Generally high absolute values indicate better (more fit) solutions which are preferentially passed into the next generation of the evolutionary search.

Concurrent processing occurs when many calculations are carried out simultaneously on different processors in a computer. Any problem that can be broken down into smaller, independent problems can be parallelized. Each sub-problem is solved independently on different processors at the same time.

A *Deadlock* occurs within program(s) when a proces, A, has to wait for a resource held by process B, where B is waiting for a resource held by C. If none of these processes can advance, they are in a deadlock. A classic example is the dining philosophers problem[7].

Data races occur when two or more threads can nondeterministically change the value of a variable with no read of that variable between them. The threads of control race to change the variable - leading to unpredictable or incorrect behaviour.

Incremental computation is "basically an attempt to avoid repeating lengthy analyses of a system specification after the specification has undergone some relatively minor change." [79].

1.2.1 Design Decisions and Limitations

1. Existing tools and code were used: Ant, Java, JUnit, Python, ConTest, Chord, Java PathFinder, etc.

[7]Dijkstra, Edsger W.: http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1000.PDF

2. Target platforms are the Linux and Mac implementations of Unix. Cross-platform support (to Windows) isn't guaranteed due to the limitations of Python.

3. Programs evaluated must be written in Java, must be concurrent and be invocable by the Ant build system.

4. An existing model checker (Java Pathfinder) is used. Preference is given to Java model checkers with a good API.

5. Programs chosen for evaluation must at a minimum noise (with ConTest) in a reasonable amount of time on a reasonable desktop computer. Being model-checkable (with JPF) is nice, but not required.

6. CORE may take hours or possibly days to run, especially if it exhausts all generations without finding a fix.

7. Programs may be fixed within the first generation of the GA¬C. This is a common problem for all heuristic bug fixing techniques and is elaborated on in Section 4.3.

1.3 Motivation

The main motivation for this research is to improve the quality of concurrent Java programs. In general, we argue that quality of automatic data race and deadlock fixing is increased by both constraining the search space of potential fixes and by creating an efficient mutant evaluation engine. Search spaces are constrained by using a combination of dynamic analysis (ConTest in ARC), static analysis (Chord in ARC-OPT), model checking (JPF in CORE-MC) and function header scanning (in CORE-IMC). Thread randomization/noising is introduced in ARC and refined in

ARC-OPT. Model checking of the mutant programs is added in CORE-MC and is augmented with incremental modeling in CORE-IMC.

In every generation the framework applies a single mutation to each member of its population. Mutations by definition are small changes. Incremental modelling techniques reuse the results of modelling runs to speed further runs. These two techniques share a natural synergy. In a population based heuristic search, using the results of model checking of the previous generation should considerably speed up model checking of the current generation. Time saved compared to many full model checking invocations could be substantial. It is possible that for any execution of CORE-IMC, only one full model checking run will be necessary. If information about the models is updated each generation, all other model checker invocations should be incremental.

A secondary motivation was to make CORE useful and usable by practitioners. Many discussions have occurred on how to make tools useful for real world software developers. CORE should be as automated as possible. It should help with the day to day problems encountered by practitioners. It should be part of a development environment familiar to practitioners and have a test suite of programs for repeatable evaluations.

In its current form the framework is almost self-contained and portable. Only installations of Java and Python are required. CORE's source code is always available as it is written in Python - an interpreted language. In its current form the framework requires some effort to set up a concurrent Java program to be fixed. One must be able to write a driver program to run the JUnit test cases that demonstrate the bug(s). The Ant build system and JPF must be understood well enough to write a compile script in Ant for the program and create a configuration file for JPF for the

program[8].

1.4 Contribution

There are no frameworks that fix data races and deadlocks in concurrent programs in a consistently reliable way. ARC does so for Java programs. Its main limitations are the time it takes and the uncertainty inherent in using ConTest.

The novel contribution of this thesis is the combination of analyses that constrain the search space of potential mutant programs and the development of an efficient genetic algorithm engine to evaluate these mutant programs.

Search space constraint is managed by integrating these methods.

1. Dynamic analysis from ConTest
2. Static analysis from Chord
3. Model checking from Java Pathfinder
4. Scanning of concurrently used function headers for in-scope lock variables

An efficient mutation engine is realised by combining the following methods.

1. Noising from the ConTest tool
2. Model checking from Java Pathfinder
3. Augmenting Java Pathfinder with incremental model checking

These methods are realised in the CORE framework in ARC-OPT, CORE-MC and CORE-IMC. Using model checking and incremental model checking to support the repair of parallel Java programs with data races and deadlocks is novel. Using them in a mutation powered, population based heuristic search is also novel.

[8]Much of this can be done by studying and modifying existing examples.

1.5 Organization of Thesis

The rest of this thesis is structured as follows. Chapter 2 introduces the concepts used in this thesis: heuristic search, model checking and incremental modelling being the most important. Chapter 4 describes the CORE framework for the bug fixing software and the evaluations done on it. The following chapters describe the specific implementations: ARC and ARC-OPT in Chapter 5, CORE-MC in 6 and CORE-IMC in 7. Conclusions and future work are presented in Chapter 8.

Chapter 2

Background

2.1 Introduction

This chapter reviews the background material required for this thesis. Heuristic search (Section 2.2) is described, with an emphasis on the search techniques used by CORE. A brief introduction to Search-based Software Engineering (Section 2.3) follows. Rapid progress has been made on repairing single threaded programs (Section 2.4). The same cannot be said for concurrent repair. Concurrency (Section 2.6) introduces complications like deadlocks and data races. Existing techniques that attempt to find, suppress and repair concurrent bugs (Section 2.7) are surveyed. CORE uses formal modelling techniques (Section 2.8), specifically model checking (Section 2.9) to determine the correctness of a candidate program. JPF (Section 2.9.1) was selected because of its maturity and the existing body of work on it (Section 2.9.2). Finally, incremental modelling (Section 2.9.3) is described, with an emphasis on the approach used in CORE.

Table 2.1: High level view of Evolutionary Programming.

Evolutionary Programming
initialize population with random genetic material
generation = 0
while solution not found and generations remain
evaluate fitness of each member of the population
mutate population
assemble next generation population from parents and children
evaluate stopping condition
generation = generation + 1
end while
output fittest member of population

2.2 Heuristic Search: Evolutionary Programming and Genetic Algorithms

"Heuristic search is what you use when you don't know what you're doing[1]*."*

Heuristic search is a family of search techniques modelled after biological evolution. One of the earliest techniques was Evolutionary Programming (EP)[2]. An overview of the algorithm is provided in Table 2.1. It is population based, where every member is considered to be a separate species. Member 1 is a cat, member 2 is a cactus and so-on.

EP is mutation driven. Mutations are changes to a member of the population. In the biological world they can be caused by changes in DNA replication or from radiation. This is represented in digital evolutionary strategies by randomly changing the values of variables by a normally distributed (Gaussian) amount. The amount of change is also low as EP assumes the children are similar to their parents. Mutation

[1] Dr. Mark Green, personal conversation.

[2] `http://www.aip.de/~ast/EvolCompFAQ/Q1_2.htm`, retrieved 21 Oct. 2013.

introduces random changes into a member of the population that might bring it closer to a good solution, do nothing, or move it farther away.

In the mutation step, the parent is replicated 1 or more times and the mutations are applied to each. EP has no fixed rules for the number of children produced by each parent. After this the parents and children compete to move into the next generation. Here as well, EP has no rules for the size of the population. It can vary from generation to generation. Figure 2.1 gives an overview of EP.

Here is an example of EP. A common question asked by any number of people is, 'Why is our universe the way it is?' One attempt at an answer is the theory of Cosmological Natural Selection (CNS) [75–77]. It suggests that the purpose of universes is to create black holes, each of which contains a child universe with its own black holes. In this scenario, universes are the members of the population. Their fitness is determined by the number of black holes (offspring universes) they produce[3]. CNS requires that the changes in the physical constants from universe to universe are small - like normally distributed mutations.

As trips through black holes are one-way and non-returnable, we can consider all child black holes to be separate species[4]. In CNS the number of universes explodes exponentially. EP models this well with its flexible child and population counts.

The fitness of a universe is determined by how many black holes it produces. Assuming that the laws of physics are the same across all realities, the parameters that can vary include the masses of particles (proton, neutron, electron and neutrino, among others) and the strengths of fundamental forces (gravity, electromagnetism, nuclear strong and nuclear weak). Table 2.2 gives an example of what the N^{th} generation of black hole universes might look like after the evaluation and mutation

[3]Universes that are good at making black holes are also hospitable to human life.

[4]It isn't possible for anything to travel from one sibling black hole to another to compare their properties as this would require exiting an event horizon.

Table 2.2: N^{th} generation of the evolution of black hole universes in Cosmological Natural Selection (CNS) by evolutionary programming (EP). m_p and m_e are the mass of the proton and electron. F_{em} and F_w are the relative strengths of the electromagnetic and weak forces.

	Black Hole #	$\boldsymbol{m_p}$ $\times\mathbf{10^{-27}}$ **kg**	$\boldsymbol{m_e}$ $\times\mathbf{10^{-31}}$ **kg**	$\boldsymbol{F_{em}}$ 1	$\boldsymbol{F_w}$ $\times\mathbf{10^{-11}}$	**Score**
Evaluated	1	1.67	9.11	1	1	600
	2	1.72	9.21	1.003	1.04	430
	3	1.6	9.16	1.0001	0.89	170
Mutated	1	1.67	**9.31**	1	1	
	2	1.72	9.21	1.003	**1.08**	
	3	1.6	9.16	**0.9999**	0.89	

steps. In the three universes, the mass of the electron, strength of the weak force and strong force respectively are mutated by small amounts. For the purposes of this example, the optimal black-hole producing realities receive a score of 1000 points. As generations and ending conditions are not well defined, they are omitted.

Genetic algorithms (GA) [10,90] are another commonly used heuristic search technique. Like evolutionary programming, they are population based and use mutation. Unlike EP, all of the members of the population are the same species and crossover is used. In EP, mutation rates are high as they are the only drivers of change. In GAs, mutation rates are low due to the use of crossover. Further, in GAs, mutation sizes are not normally distributed. An overview of the GA algorithm is given in Table 2.3.

Crossover is the splicing together of two parents to produce children. It is usually applied with a probability of 60% or more [10,13]. It is common for parents to produce 2 children to replace (or compete with) themselves.

Continuing our Cosmological Natural Selection example for GAs, black holes can merge[5]. In this specific example, both parents are consumed and only one sibling is

[5]Black hole mergers are not explicitly mentioned in CNS.

Table 2.3: High level view of a genetic algorithm.

Genetic Algorithm
initialize population with random genetic material
generation = 0
while solution not found and generations remain
evaluate fitness of each member of the population
mutate population
create children by crossing over parents
assemble next generation population from parents and children
evaluate stopping condition
generation = generation + 1
end while
output fittest member of population

Table 2.4: Crossover applied to the first two black hole universes from Table 2.2. They have merged and replaced black hole 1. Black hole 2 was removed in the crossover step.

	Black Hole #	m_p $\times 10^{-27}$ **kg**	m_e $\times 10^{-31}$ **kg**	F_{em} 1	F_w $\times 10^{-11}$	**Score**
	1	1.67	9.11	1	1	600
Evaluated	2	1.72	9.21	1.003	1.04	430
	3	1.6	9.16	1.0001	0.89	170
	1	1.67	**9.31**	1	1	
Mutated	2	1.72	9.21	1.003	**1.08**	
	3	1.6	9.16	**0.9999**	0.89	
	1	1.67	9.31	**1.003**	**1.08**	
Crossover	3	1.6	9.16	0.9999	0.89	

produced. In Table 2.4, universes 1 and 2 are crossed over, while 3 is left alone. When we compare the mutated and crossover rows of Table 2.4, we observe the following.

1. M_p and M_e are selected from universe 1.
2. F_{em} and F_w are selected from universe 2.
3. These values (M_{p1}, M_{e1}, F_{em2} and F_{w2}) are combined together (crossed over) to form a new universe.
4. This universe replaces universe 1.
5. Universe 2 is removed, as it has merged with universe 1.

Replacing parents with their offspring is a simple strategy. Other strategies have parents and children compete to pass into the next generation. Alternatively each member of the population may have a finite lifetime, but once it does expire, it is removed from the population. In competitive selection, the higher scoring (more fit) solutions are preferentially passed to the next generation. Note that low fitness members also have a chance to pass to the next generation as well. This helps to preserve genetic diversity.

The rest of the algorithm is straightforward. We check to see if the goal has been achieved. If not, we repeat the process while we have generations remaining. Otherwise we exit and if no solution was found, optionally present the best solution found so far.

Fitness, mutation, crossover and preferential treatment for fitter members drive the genetic algorithm. As preferential treatment is given to the fit, those members spread in the population, bringing up the average fitness. Crossover randomly combines two realities to produce a new one. This could fortuitously create an even more fit member. Mutation prevents stagnation. By randomly injecting new values into

the "genetic mix," a population doesn't converge too early on good, but not optimal values.

Genetic programming (GP) [44] is an extension of genetic algorithms in which the solution is stored in a tree structure. This makes it easy to store and manipulate programs, for example. All of the sequential program repair papers described here use GP.

One significant weakness of heuristic search techniques such as genetic algorithms is the selection of parameters. How big a population does one use? For how many generations should it run? What rate should be selected for crossover and mutation? Studies [6] indicate that parameter selection has a significant impact on the quality of the solution generated. To complicate matters, there is no optimal parameter set for all problems. Optimal parameters are problem dependent.

2.3 Search-Based Software Engineering

Search Based Software Engineering (SBSE) [32, 35, 47] is a field of computer science in which heuristic search techniques, like genetic algorithms and evolutionary programming, are used to solve a wide array of software challenges. They include (but are not limited to) project planning, maintenance, reverse engineering, source code comprehension [33, 46], source code refactoring, component selection [9] and program repair. The largest area of SBSE - and of interest to CORE - is concerned with testing software [47]. SBSE techniques exist to generate, improve and optimize[6] test suites.

Many software engineering problems are optimization problems or can be expressed as such; optimize for understanding or optimize the test suite size. SBSE rephrases them as search problems: search for the optimal understanding, or search

[6]Improving and optimizing are different: The former could increase coverage (for example,) while the latter removes redundant tests.

for the optimal test suite size. Any optimization problem representable as a member of an evolvable population (like optimizing universes to produce black holes) and expressible in terms of a score (fitness function: number of black holes produced by said universe) can be adapted to search based techniques.

SBSE is often used when existing algorithms take too long, or are not known. The randomness inherent to heuristic search means that from one invocation of the search to another, one doesn't receive the same answer or the same quality of answer. This is often acceptable because near optimal solutions or optimizations are better than no solution at all. For example, scheduling shifts for all of the vehicles and workers at an airline taking into account all local labour and safety laws (pilot fatigue, vehicle maintenance, vacations, ...) is a hard problem. It may take a scheduling program a week or more to come up with a solution. If a search-based approach determines a solution within a few hours that is 3% worse[7], that may be acceptable under the circumstances.

Fitness functions can be programmed to examine trade-offs or problems with multiple, conflicting objectives [28]. For example, in the next release problem [9], customers have a list of features they would like to see implemented. Each feature takes time and costs money. A multi-objective heuristic search creates a series of scenarios examining customer satisfaction against the cost of implementing subsets of these features. Optimizing the conflict between them isn't a yes-no problem. Many different choices could be made with different consequences for customers and company.

One limitation faced by SBSE approaches is the search space. This is all of the possible solutions to the search based problem that must be examined. Using the simplified cosmological example above, the mass of the electron and proton are constrained to be within two orders of magnitude of their values in our universe and

[7]And still complies with all laws and regulations.

the electromagnetic and weak forces are constrained to be within one magnitude. The mass of the proton, m_p, for example, can take on $100 \times 100 = 10,000$ different values while the force of gravity can take on $10 \times 10 = 100$ different values. The search space is then, $10000 \times 10000 \times 100 \times 100$ or 10^{12} different possible universes. This is large by human standards, but easy for a computer to navigate. To be clear, heuristic search techniques don't test all 10^{12} combinations. In the example above, the simplified universe search used 3 members (realities). If we let it run for 20 generations, there are at most 3×20 or 60 realities evaluated.

Flush with the success of explaining our universe we try to use genetic programming to evolve the equations of the Standard Model of Physics ... and fail completely. The search space is all mathematical statements buildable from elementary mathematics. The equations describing the interactions of matter and forces are the stopping conditions. This is beyond the ability of GP[8]. Simply put, the search space is too large. It is important to recognize that heuristic search has limits. One of them is that it cannot create complicated things starting from nothing. A second is that a heuristic search fails when the search space is too large or the distance in the search space from 'here' (the starting point) to 'there' (an acceptable solution) is too large.

2.4 Automatic Single-Threaded Program Repair

If programs can't be evolved, how can we use heuristic search techniques to repair them? It is possible because we don't have to start from scratch. We assume the *competent programmer hypothesis* [2]: A programmer is competent and strives to create correct programs. If the program has a bug in it, it is still *mostly correct*. All it takes is a few changes (insertions, deletions, ...) in the proper spot(s) to correct the

[8]GP can't even evolve a sorting algorithm from scratch [7].

error. The search space - the difficulty in getting from *here* to *there* - is then orders of magnitude smaller.

Automatic sequential program repair has made great strides in the last five years. It started with an approach capable of fixing bugs in toy algorithms [7]. No techniques were applied to limit the search space so the approach didn't work on programs larger than roughly 10 lines of code. Further research has led to an approach capable of fixing programs up to 40 lines [5].

At the same time a technique was developed by a different research group preferentially targeting the part of the code where the error occurs. This approach also assumed the error was written correctly somewhere else in the program. Part of the fixing process involved copying the correctly implemented code to the region of the error. Targeting the search this way allowed this approach to fix bugs in large programs - up to 21,000 lines of code in early tests [49–51, 87, 88].

2.5 Concurrency

Until recently, sequential processing was the norm in desktop computers. One processor did all the work. By rapidly switching tasks, it provided the illusion of multiple programs running at once. This illusion is propagated by the incredibly fast speeds (gigahertz) at which these processors operate. There is a ceiling though, to increases in processor speed: processors must shrink as they get faster. Heat generation and quantum mechanics[9] place a fundamental limit on how small (and thus how fast) a processor can be. To continue the rapid gains of previous decades it is necessary to place multiple cores inside desktop computers. This is the norm today[10]. Two to four processors are commonplace and eight can be found in high-end systems.

[9] Quantum tunnelling for example, becomes a problem as components become smaller and smaller.
[10] October, 2013

Parallel processing occurs when many calculations are carried out simultaneously on different processors in a computer. Fundamentally, any problem that can be broken down into smaller, independent problems can be parallelized. Each sub-problem is solved independently on different processors at the same time.

Most software today is written for sequential processing. This leads to the common situation where one processor in a desktop computer is saturated with work while the other(s) are idle. Software hasn't yet caught up with the changes in hardware. Writing parallel programs is much more difficult than writing sequential ones. Each unit of computation talks to its neighbours and reads and writes data shared among them. There is a rich literature on concurrent processing. We briefly survey some terminology[11] and describe how CORE misuses it in the next section.

A *critical section* is a piece of code available to multiple processes in a concurrent program. It contains instructions that should only be executed by one (or a few) process at a time - such as updating a bank balance. *Mutual exclusion* is the requirement that two or more processes are not in a critical section at the same time.

Locks enforces the mutual exclusion in code. Simple locks allows only one process access to the code at a time. *Spin locks* are a type of lock in which other processes simply wait (spin) until the lock is free. This is efficient if the lock is only held for a short time, but wasteful if the lock is held longer. When the lock is held longer than it would take the operating system to reschedule the spinning process, it is wasteful.

Semaphores keep track of how many resources are free. A *binary semaphore* has one available resource while a *counting semaphore* has a number N of resources available.

Monitors are locks with additional properties. If a process, prA, is working in the critical section of a monitor and cannot proceed for some reason, it can temporarily

[11] coughWikipediacough

relinquish its lock and go to sleep. That is, it gives up its remaining processor time to another process, prB, that can make progress. When prA is able to make progress again, it reacquires the lock then continues its work. Note that while prA has the lock, it has exclusive access to the resource.

Recursive locks or *mutex locks* are locks that allows a process to acquire the lock more than once, or recursively acquire the lock. Normally, if a process acquires a lock and then attempts to acquire it again, it deadlocks. Mutex locks allow a process to acquire it more than once. What makes mutex locks difficult to work with is that the lock must be released the same number of times that it was acquired in order for it to be free again. When acquiring and releasing the lock depends on complex logic, it is possible that the acquiring and releasing of the mutex lock fall out of step.

Reader writer locks or *multiple readers, single writer locks* are locks that allow multiple processes to read the data protected by the lock, but only one process to write to that data. Reader writer locks have to prevent readers from reading while a writer is writing. This can cause the writer process to wait forever or *starve* if there are numerous readers and there is no provision to allow the writer process to write. Conversely, allowing the writer process to write can hurt performance when writing is frequent.

2.5.1 Java Synchronization

Java has built in support for concurrency[12]. The two synchronization constructs CORE modifies are *synchronized methods* and *synchronized statements*. Synchronized methods (Figure 2.1) contain the *synchronized* keyword in the method declaration. This prevents more than one process from being within the object's method at the same time. Once one process has the method, all other processes must wait for it to

[12]http://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html, retrieved August 2014

Figure 2.1: In Java, synchronized access to an object's methods is enforced by adding the *synchronized* keyword to the method header.

```
public class bankAccount {

  public synchronized boolean withdraw(...) {
    ...
  }

  ...
}
```

Figure 2.2: In Java, lines can be locked by synchronizing on any non-primitive variable.

```
public class bankAccount {
  private object myLock;
  private currency balance;

  public bankAccount() {
    balance = 0.0;
    myLock = new object();
    ...
  }

  public boolean withdraw(currency amount) {
    synchronized (myLock) {
      balance -= amount
      ...
    }
  }

  ...
}
```

Table 2.5: The algorithm for a physicist to write on the blackboard is shown on the left. Two physicists working at the same blackboard end up in deadlock on the right.

Working Algorithm	**Deadlock / Starvation**

function DoPhysics {
  Pick up chalk
  Pick up eraser
  Write
  Put down eraser
  Put down chalk
}
	Phys1: Thinks
	Phys2: Thinks
	Phys1: Pick up chalk
	Phys2: (Waiting to pick up chalk)
	
	Deadlock: Both are stuck

finish.

Lines or blocks of code are also synchronized by surrounding them in a synchronized block (Figure 2.2). In this case a lock is needed. Any non-primitive type can be used as a lock Java also allows an object to be used as a lock before it is created. In both cases, when a synchronized method or block is exited, Java ensures that all other processes waiting on them see the correct state of the object.

CORE mutates code by adding, removing, growing, shrinking and swapping these synchronized blocks. When this thesis refers to synchronize blocks, it is referring to what has been described here (Figures 2.1 and 2.1).

2.6 Deadlocks, Data Races and Synchronization Blocks

Concurrent programming leads to new kinds of software bugs not found in sequential software. Of concern here are *data races* and *deadlocks*. Each is illustrated with a modern example; working physicists (with respect to dining philosophers). Imagine

two physicists writing at a blackboard[13]. They think, then write, then think again and write endlessly. There is only one piece of chalk and one eraser in the room. A physicists needs both before she can write. Once she picks up one she won't put it down until she has both and can finish writing. It doesn't matter in which order she picks up the chalk and eraser. In concurrent terms, each physicist is a process. When they have both items (data accessors) they write (perform some computation). Table 2.5 shows two possible outcomes for this situation. On the left, a physicist successfully picks up both items. On the right, each picks up one item. As they are unable to pick up the other, they are stuck, or *deadlocked*. Each is blocking the other from proceeding.

A graduate student comes in with a question. Both physicists answer, interweaving their replies with each other[14]. As string theory and loop quantum gravity describe the universe in different and incompatible ways, the graduate student eventually leaves even more confused. This is the essence of a *data race*: processes (physicists) have unrestrained access to a resource (graduate student) or output. Data is lost by being overwritten before it is read or the ordering of the data is incorrect or confused. Unexpected values could be read, leading to errors or incorrect results.

2.7 Existing Work on Finding, Suppressing and Repairing Deadlocks and Data Races

The success in fixing sequential programs has not been replicated reliably for concurrent programs. Many techniques exist to find concurrent bugs like data races

[13]One works on string theory, the other on loop quantum gravity - two different and incompatible ways of including gravity in our fundamental theories of reality.

[14]Its simple really. Just integrate the Hamiltonian from 0 to null infinity in 5-space using hyperbolic coordinates while obeying the weak energy condition in Einstein-Bonnet gravity.

and deadlocks. For instance CHESS [60] is a model checking framework for finding concurrency bugs in Windows programs. Deadlock detection in Java programs using multiple static analysis is described in [62]. Static detection of deadlocks, data races and reachability bugs is often inaccurate. An approach to improving on this by using heuristically directed model checking is described in [70].

A method for detecting data races in Java programs using ConTest is described in [53]. In related work [45], attempts are made to suppress bugs by influencing the scheduling of threads. Scheduling is altered to try and decrease the probability of a bug occurring. This approach alters scheduling but doesn't fix the code itself. A method of fixing data races is implemented, but ensuring the fix doesn't introduce new problems, like deadlocks, requires model checking and is left by the authors as future work.

Implicit atomicity arbitrarily groups consecutive dynamic memory operations into atomic blocks to enforce memory ordering at a coarse grain. It tries to hide data races (and atomicity violations) by reducing the number of interleaving opportunities between memory operations. Atom-Aid [59] implements this idea, reducing the probability that data races and atomicity violations will occur.

AFix [40] is a system for fixing single-variable atomicity violations in C/C++ programs. It works in multiple steps. First it uses CTrigger - an in house tool - to dynamically detect atomicity violations. AFix then develops patches for each bug. Once all patches are created, they are merged and optimized. Thread noising techniques are used to test the fixes created by AFix. Encouraging results are reported from this approach.

2.8 Modelling

This section provides a very brief introduction to modelling. In particular we look at the B method. A *model* describes a system and what it does in a mathematically precise and unambiguous way. *Model proving* is an automatic, formal verification technique that systematically checks if a formal property holds in the model. A model is *correct* when it satisfies all the properties (invariants, pre-conditions, ...) obtained from the specification. Note that correctness is only as good as the model and the engine doing the checking[15].

B [1, 71] is a formal method for specifying, designing and implementing software. It is a modelling language designed to ease the transition between programming and modelling. Its structure is familiar to anyone who has worked with code. In modelling one writes mathematical variables, sets, or relations and the constraints between them. These constraints describe the desired behaviour of those sets and variables.

The largest hurdle to overcome when modelling for the first time is changing mindsets. When developing code, one is telling the computer *what to do.* When modelling one is telling the computer what to do by describing *what holds true.* True statements are the desired behaviour. The prover's (analogous to a compiler) role is to check the model for any holes, ambiguities or contradictions to the "truth" the modeller is creating and then display them. Finding any holes is similar to failing to compile. One has to find out what went wrong, correct the mistake and try again.

Figure 2.3 is an example of a simple B model. The desired outcome is to add two integers, with the restriction that the first integer must be between 0 and 100 inclusive. This model (called a MACHINE in B parlance) declares two variables, Num1 and Num2 in the VARIABLES clause. All variables are typed within the

[15]The author has encountered and reported a bug in the Event-B prover.

INVARIANT clause. Additional restrictions and relationships among variables, like Num1 < Num2 are also specified here. Invariants must *always hold true*. Initial values are given to variables in the INITIALISATION clause.

OPERATIONS are analogous to functions. SetNumbers accepts two inputs. PRE conditions are invariants that must hold true for the operation to be called. Here the types of input arguments InArg1 and InArg2 must be integers. Values are assigned to Num1 and Num2 in parallel, $||$. Operation AddNumbers adds the numbers and returns them in the locally declared OutResult variable. Note the (arbitrary) restriction of Num1's value in the precondition of AddNumbers.

The power of modelling is proving that the model is correct and defect free. Correctness by design is enforced by the generation of proof obligations by the B engine that the models must fulfil. Proof obligations are the formal contracts (evidence and guarantees) showing the model is correct. Some of the properties that a B model checker [15] looks for include:

- Assertions are always satisfied.
- Initialization satisfies all invariants.
- Results of operations preserve all invariants.
- Types of variables are preserved.
- User-defined sets are non-empty.
- Assignments to variables do not cause them to overflow (eg: x is always <= MAXINT) or underflow (x is always >= MININT).

All proofs must evaluate to true for the model to be correct. (Analogously, a program must be correct for it to compile.) A good model checker automatically

discharges (proves) many proof obligations using internal rules and heuristics found in first order logic. Others require the modeller to work interactively with the model checker to either prove or refute the remaining proof obligations.

When the model discharges all proof obligations we say it is *correct by design*. That is, the prover has exercised all of its ability to find holes, ambiguities and contradictions within the model and failed.

One proof obligation is unfulfilled for the machine in figure 2.3:

> Check that the invariant (Num1 < Num2) is preserved by the operation
> - ref 3.4 => InArg1 + 1 <= InArg2

The INVARIANT clause has the requirement, Num1 < Num2. This isn't enforced by the preconditions of the SetNumbers operation. (In B, invariants are not automatically added to preconditions upon evaluation.) In effect, the prover is saying, 'In SetNumbers it is possible to keep adding one to InArg1 until is is equal or larger than InArg2. Once they are assigned to Num1 and Num2 respectively, the invariant is broken.' Adding InArg1 < InArg2 to the PRE clause of SetNumbers satisfies the invariant.

In B, C source code can be generated from the models. If the models are correct by design, then the code is as well. Can this be done in the opposite direction? That is, given source code, can we create a model from it and formally check its properties. We can with tools like Java Pathfinder (JPF), as described in the next section.

2.9 Model Checking

Model verification proves that a model satisfies all of its proof obligations. It is different from *model checking* in the following respect. Model verification examines the

consistency between a model and its requirements, while model checking enumerates and examines all of the states a model of the program can reach and checks them against formalized requirements or other desired properties. Further, model checking does so for all thread interleavings. Both are formal techniques. Only model checking exhaustively explores the state space - the space of all states reachable by the program.

Many of the properties checked for are built into the model checking environment itself. This removes the burden from the user of having to implement these checks in every project. Properties that model checkers can test for include correctness, reachability, safety, liveness, fairness, exceptions, deadlocks and data races.

Attempting to model check a program can end in three different ways. First, the model has all of the desired properties and passes all checks, that is the model satisfies the formalized requirement or property. Second, the model checker runs out of memory. This is the *state space explosion problem.* Techniques to deal with this are described below. Third, a counter-example is found describing a deficiency in the model. Information in the counter-example should give some indication of what went wrong. Even with the description it may take some effort to determine the source of the defect. It could be a *modelling error*: The model doesn't reflect the design, so it needs to be corrected. Otherwise the design could be ambiguous or incorrect and could need to be improved. Lastly, it could be a *property error.* An invariant or precondition in the model doesn't reflect the design document so it must be changed.

Before discussing the relevance of model checking to CORE, we need to know what state spaces are. Data and control are the two looked at here. *Data state space* is the total number of different values the variables and objects in the program can take on over the program's life. Table 2.6 shows a fragment of a program for which we want to calculate the data state space. Variable a is assigned a random value in

Table 2.6: Calculating the size of a state space.

Program Fragment	**States Per Line**
int a = 0, b = 0, c = 0, d = 0;	
a = random(1, 100);	100 states
b = random(1, 3);	3 states
c =random(1, 3);	
if (c == 1)	
d = 7 * random(1, 100);	100 states
else if (c == 2)	
d = 5 * random(1, 10);	10 states
else	
d = 10;	1 state

the range of 1 to 100, variable b in the range 1 to 3. Taken together a and b can take on 300 different states: {{a = 1, b = 1}, {1, 2}, ...{1,100}, {2, 1}, ... {2, 100}, ...{3, 100}}. Separately the if-elseif-else structure imposes a restriction on the value of d. It can be either random(1, 100) or random(1, 10) or 10. As they are mutually exclusive, these states are summed, not multiplied together. State sizes are 100, 10 and 1 for c= 1, 2 and 3 respectively giving us 111 states. To find the total number of data states, multiply 300 by 111 to get 33,300 states for this fragment. Two examples of state are {a = 42, b = 2, c = 1, d = 93} and {65, 3, 2, 7}. *Control state space* enumerates the number of different paths through the code. Here the control flow is 3, one each for the if, else if and else branches.

For the data state space, model checkers exhaustively explore all 33,300 states. When scaled to realistic programs it becomes obvious that their *state spaces are huge.* Computers have finite amounts of memory and processing power so techniques have to be implemented to deal with this data state space problem.

Table 2.7: Exploring the interleavings of the working physicists. In the left column, ordering is enforced so no deadlocks occur. No enforcement exists for the middle or right columns. Luckily the center column doesn't deadlock, but the right column does.

Enforced Ordering	**No Deadlock**	**Deadlock**
Ph1: Thinks	Ph1: Thinks	Ph1: Thinks
Ph2: Thinks	Ph2: Thinks	Ph2: Thinks
Ph1: Pick up chalk	Ph2: Thinks	Ph2: Thinks
Ph1: Pick up eraser	Ph1: Pick up chalk	Ph1: Thinks
Ph1: Writes	Ph1: Pick up eraser	Ph1: Thinks
Ph1: Put down eraser	Ph1: Writes	Ph1: Thinks
Ph1: Put down chalk	Ph1: Put down eraser	Ph2: Thinks
Ph2: Pick up chalk	Ph2: Pick up eraser	Ph2: Thinks
Ph2: Pick up eraser	Ph1: Put down chalk	Ph2: Pick up chalk
Ph2: Writes	Ph2: Pick up chalk	Ph1: Pick up eraser
Ph2: Put down eraser	...	
Ph2: Put down chalk		Deadlock
...		

Data races and deadlocks are of interest, so we explore how a model checker looks for them in greater depth. To find deadlocks a model checker has to explore the *interleavings* of concurrent threads, the interleavings state space. That is, every single combination of the ways in which statements from separate threads can be mixed together must be examined. Table 2.7 illustrates different situations encountered by the working physicists. In the left column, ordering has been enforced by some kind of synchronization mechanism causing physicists to write in turn. Efficiency is sacrificed for safety. Locking is removed in the central column. By luck or design this interleaving doesn't contain a deadlock. On the right is an interleaving leading to a deadlock.

The long list of '*Thinks*' from the right column of Table 2.7 emphasizes the requirement that ALL interleavings must be explored - even when redundant. After some thought we realize a number of statements in the working physicists *DoPysics*

Table 2.8: Two examples of interleavings with partial order reduction. Ordering is enforced in the left column. The lack of ordering in the right column causes a deadlock.

No Deadlock	**Deadlock**
Ph2: Pick up chalk	
Ph2: Pick up eraser	
Ph2: Write	Ph1: Pick up chalk
Ph1: Pick up eraser	Ph2: Pick up eraser
Ph1: Pick up chalk	
Ph1: Write	Deadlock
...	

function from Table 2.5 have no effect on the existence of deadlocks. *Put down eraser*, *put down chalk*, *think* and even *write* can all be removed from the list of statements that must be interleaved. Only *pick up chalk* and *pick up eraser* determine if there is a deadlock and only *explain to graduate student* determines if there is a data race. *Partial order reduction* is a family of techniques that examines these dependencies and removes the statements not affecting the outcome. There is a significant savings in memory using this technique. Programs that are orders of magnitude larger can be model checked when partial order reduction is enabled. Table 2.8 contains a deadlock example, while Table 2.9 contains a data race example, using partial order reduction.

2.9.1 Java PathFinder

Java Pathfinder (JPF) [84] is the model checker used in CORE. This section gives a brief description of JPF and surveys some scholarly work done on or with it.

Initially model checking was most often used during the software design phase. Designs are simpler than programs and have smaller state spaces. The authors of JPF created it in an attempt to nudge the formal methods community more towards model checking code. They cite a number of reasons:

Table 2.9: Exploring interleavings with partial order reduction leading to potential data races. As in previous cases, ordering in the left column prevents a data race. There is no ordering in the right column, leading to a data race. 'Leaves' is included for understandability.

No Data race	**Data race**
Ph1: Explain to student	
Ph1: Explain to student	Ph1: Explain to student
Stu: Leaves	Ph2: Explain to student
Ph2: Explain to student	
Ph2: Explain to student	Data race
...	

- Errors exist in programs regardless of model checking the designs.
- Critical section errors and deadlocks are introduced at a deeper level of detail than in the design document.
- Formal methods (such as JPF) should support debugging and error location along with model checking.

JPF is a model checker and a verification, analysis and testing environment for Java. It has a number of features in it to combat the state space explosion problem. Partial order reduction, described above, by static analysis, is one of them. Two other tools are integrated into JPF to perform runtime analyses: *Eraser* [36] detects data races while *LockTree* detects deadlocks. *State space collapse* is an optimization in JPF in which every item (object, variable, ...) in a state is placed in a table with a unique index. Objects are then compared by computing and comparing indexes instead of the objects themselves. This optimization increased the number of states stored in memory and the number of states compared per second by two orders of magnitude each.

Predicate abstraction is another optimization technique used by JPF in which a

program with a large or infinite state space is represented or abstracted by a finite number of predicates. A predicate is a function that takes a variable number of arguments and returns true or false. Model checking is performed on the predicates instead of the actual program. If a counter-example is found on the predicates, they can be checked against the actual program. If the counter-example is false, the abstraction is improved by adding or modifying predicates so the counter-example doesn't occur again[16].

2.9.2 Existing Work on Java Pathfinder

Most of the existing work is related to improving JPF's model checking capabilities. For example, an experience parallelizing JPF is reported in [21]. In their study, the authors parallelized random state space searching, reporting increased speeds of 2 to 1000x.

Delta Execution [18] is similar to incremental modelling. Whereas incremental modelling works across model checking runs, delta execution works within a run. It reuses both the storage and model checking results of heap states. Only the uncommon parts of the heap - the deltas - are stored and executed separately. For example, if 3 states are $\{q, r, s, w\}, \{a, r, s, t\}$ and $\{z, r, s, b\}$ the common state, $\{\_, r, s, \_\}$, is stored and model checked once. Only the deltas, $\{q, \_, \_, w\}, \{a, \_, \_, t\}$ and $\{z, \_, \_, b\}$ are stored and model checked independently. Exploration times are improved by a factor of up to $11\times$ in evaluations.

Other techniques used by JPF include *mixed execution* [19]. This technique improves the execution time of deterministic blocks in JPF by translating the state from the JPF virtual machine to the host Java virtual machine. Deterministic blocks don't

[16] http://chicory.stanford.edu/satyaki/research/PredicateAbstraction.html

require Java Pathfinder's virtual machine layer, so effort is saved as only one virtual machine – not two – is used. Further, *lazy translation* is used to translate only the parts of the state that an execution dynamically depends on. Average improvements of 36% are realised in experiments.

JPF-SE [3] performs symbolic execution of the code. In their words:

> "Programs are instrumented to enable JPF to perform symbolic execution; concrete types are replaced with corresponding symbolic types and concrete operations are replaced with calls to methods that implement corresponding operations on symbolic expressions. Whenever a path condition is updated, it is checked for satisfiability using an appropriate decision procedure. If the path condition is unsatisfiable, the model checker backtracks."

Object graphs are collections of objects in which the nodes are the objects and the edges are the connections between objects. These can be test inputs to programs. A method of automatically creating object graphs meeting user constraints is described in [30] along with optimizations made to the process. Average speed increases of 16x are reported in the experiments performed.

Basset is a JPF-based project to model check the actor based languages, Scala and ActorFoundry. An adaptation layer replaces the actor libraries with simpler ones so the model checking is focused on the application code. Specifically, features such as automatic thread migration and balancing (among others) are removed. Increased speeds averaging 30% are reported in experiments.

In [82] the concurrency libraries introduced in Java 1.5, java.util.concurrent are replaced by model classes. This has two effects. First, model classes don't need to be model checked. Second, they are represented in the model checking space by a

single integer. When the code is checked, the model Java interface is used to delegate the execution of the libraries to the host virtual machine on which JPF runs. Model Java interface code isn't model checked by the JPF virtual machine, so effort is saved. Average improvements of 40% are reported in experiments.

Heuristic techniques like particle swarm optimization [27] are often used in directed model checking approaches. The search algorithm guides/directs the exploration to an area likely to contain an error or counter-example, saving effort over an exhaustive search for it. A counter-example is usually created with less effort than using a regular model checking search algorithm. It is also possible the error path is shorter than with exhaustive techniques. Counter-example generation cannot prove a model correct of course. In [80] an Estimation of Distribution algorithm is used to find counter-examples.

Counter-example generation with genetic algorithms [52] was implemented using a new memory operator. In the language of this proposal, the authors used a modified form of incremental modelling to save memory. Only the previous state is remembered, anything older is thrown away.

2.9.3 Incremental Model Checking

Model checking involves the examination of all states and transitions in a program. Table 2.10 gives state transition examples for our working physicists. In the starting state, physicist 1 has chalk in hand and physicist 2 is thinking. If the total number of possible transitions, or things to do is 6 {pick up chalk, pick up eraser, put down chalk, put down eraser, think, write} and there are two physicists then there are 12 total transitions from this state. Four interesting cases are shown in the table. The last entry of course leads to a deadlock.

Incremental model checking involves using the output of a model checking run as

Table 2.10: Examples of a current state and transitions to a next state for working physicists. In the first case, when the physicist is only holding the chalk, she cannot think or put it down, so there is no change in state.

Current State	**Transition Fn**	**New State**
Phys. 1 has chalk, Phys. 2 is thinking	Think(Phys1), or PutDownChalk(Phys1)	Transition disabled. No state change.
Phys. 1 has chalk, Phys. 2 is thinking	PickUpEraser(Phys1)	Phys. 1 has chalk and eraser, Phys. 2 is thinking
Phys. 1 has chalk, Phys. 2 is thinking	Think(Phys2)	Phys. 1 has chalk, Phys. 2 is thinking
Phys. 1 has chalk, Phys. 2 is thinking	PickUpEraser(Phys2)	Phys. 1 has chalk, Phys. 2 has eraser

input to the next run. This speeds up the second run by re-using calculations from the first run.

Incremental model checking is a two step process - first recording and secondly, re-using the results of model checking runs. An implementation similar to [48] is described. In the first run, a program is model checked the standard way. Decisions made during the run are recorded in a file in a *Hash(Start state)-transition function call-Hash(end state)* format similar to Table 2.10. Note that transitions that cannot be taken and transitions leading to error states are not recorded.

Changes are then made to the program code. There are two linked restrictions that should be adhered to for incremental modelling to work well. First, these changes *should minimally alter the state space*: variables and their types shouldn't be changed, added or removed from classes, method or method calls. Second, the hash of the states should be the same. When the variables and types don't change, the hash doesn't change. This is so the *Hash(Start state)-transition function call-Hash(end*

state) match up between the two versions of the program. For the parts of the code where the state space has changed, it must be fully model checked again.

The output of the *previous run* and the modified source are the inputs to the second model checking run. Changes from the previous run are stored to a *change list* for the current run. States in the current model checking run are hashed and searched for in the previous run and change list. Three cases result. First, if the state is in the change list, this state has already been visited in this run. We don't have to evaluate this state again or visit any of it's child states. This is standard model checking practice.

Second, the state is found in the previous run. We don't have to re-evaluate this state again, but we cannot assume that all child states reached from this one will be the same. A change "further down" may be different so all child nodes must still be examined. Effort is saved, though, by not re-evaluating this state. *This is the innovation of incremental modelling* and the source of the performance boost. If changes between program versions are small - like mutation induced changes - this case should be very common.

In the third case, the state isn't in either the previous run or change list. This is a new state added since the previous run. It must be evaluated and all child states explored. It is eventually added to the change list. As changes to the program are usually small - and mutations are definitely small changes - this case should happen rarely.

Table 2.11 summarizes these three cases. Every entry of *false* is effort saved. At the end of the incremental run, the previous run data list and change list for the current run are combined together and written to file. This becomes the incremental data for the next run.

A brief survey of incremental model checking (IMC) in the literature rounds out

Table 2.11: In incremental modelling, a state seen on a previous run doesn't need to be evaluated again. Its child states must still be checked. Every entry of false in the table is effort saved.

Case	**Evaluate**	**Evaluate Children**	**New in Incremental Modelling**
Already visited transition	false	false	No
Existing transition from previous run	false	true	Yes
New or modified transition in current run	true	true	No

this section. IMC is implemented in Java Pathfinder in [48]. Average speed increases of 40% are realised. IMC has been applied by the same group to the verification of network protocols in J-Sim [78]. In [16] IMC is applied to an inter-procedural algorithm analysing recursive state machines for null pointer dereferencing. Blast [37] is an implementation of a lazy-abstraction algorithm that incrementally model checks temporal safety properties during software development.

Test cases are used by CORE to demonstrate the data race or deadlock. IMC has been used in a framework [25] to create tests for the modified parts of a program that are checked against the modified specification. Counter-example generation using IMC and the estimation of distribution algorithm is described in [81].

Figure 2.3: Example B machine that adds two integers and returns the result. Note that $Num1$ must be between 1 and 100 inclusive.

MACHINE *AddTwo*

VARIABLES *Num1*, *Num2*

INVARIANT
$Num1 \in INTEGER \wedge Num2 \in INTEGER \wedge Num1 < Num2$

INITIALISATION
$Num1 := 0 \parallel Num2 := 1$

OPERATIONS
SetNumbers(*InArg1*, *InArg2*) =
PRE
$InArg1 \in INTEGER \wedge InArg2 \in INTEGER$
THEN
$Num1 := InArg1 \parallel$
$Num2 := InArg2$
END;

$OutResult \longleftarrow AddNumbers =$
PRE
$Num1 \geq 0 \wedge Num1 \leq 100 \wedge OutResult \in INTEGER$
THEN
$OutResult := Num1 + Num2$
END;
END

Chapter 3

Literature Survey

3.1 Introduction

This chapter surveys relevant research literature on concurrent bug detection, suppression and fixing.

3.2 Literature Survey

3.2.1 Falcon: Fault Localization in Concurrent Programs

Falcon [66] is a framework for detecting atomicity and order violations. It proposes a pattern-based analysis of concurrent programs to find and rank the two types of violations. It records memory access sequences from threads on concurrently used variables and then performs a statistical analysis on them to assign a suspiciousness score to each. They are ranked and presented to the user.

The Soot Analysis framework is used to perform a static thread-escape analysis of the Java bytecode to both determine which variables could be shared and then to instrument the program to record all shared accesses of these variables at runtime.

A test case is selected and the program is run many times by Soot to try and expose faulty interleavings. Memory access sequences are then associated with the pass/fail results of the testing. Suspiciousness values are computed from both the memory access sequences and the testing results.

Falcon was evaluated against a suite of programs with known bugs. It consistently placed the bug in the first or second rank. On average Falcon slowed the execution of each program by a factor of 8 to 10.

CORE and Falcon share some common steps - static analysis, determining variables used concurrently and multiple runs of the program by test case(s). CORE is more general, in that it fixes both data races and deadlocks. Falcon's output could also be used as input to CORE to help find faults to fix.

3.2.2 AtomAid: Detecting and Surviving Atomicity Violations

Data races occur when two or more threads access the same data at the same time without synchronization and at least one of the accesses is a write. Being data race free doesn't guarantee correct programs as correctness depends on a stronger condition called atomicity, that "requires that every concurrent execution of a set of operations is equivalent to some serial execution of the same operations. Atomicity violations, sometimes called high-level data races, can cause erroneous behaviours when a consistency requirement exists between multiple pieces of shared data [59]."

Implicit atomicity is the grouping of dynamic memory operations into atomic blocks to enforce coarse grained memory ordering. These systems look for a series of operations performed by one thread and place them in the same block - effectively making the entire block of operations atomic. No special annotations are needed.

Implicit atomicity generators suppress atomicity violations by reducing the number of thread interleavings leading to an error. The authors of [59] created Atom-Aid, that "creates implicit atomic blocks intelligently instead of arbitrarily, dramatically reducing the probability that atomicity violations will manifest themselves [59]."

Atom-Aid studies the program as it is running to detect likely atomicity violations. This allows it to adjust blocks dynamically, allowing the program to find and survive these bugs. It is also able to report the possible locations of atomicity violations to help with the debugging process. In tests Atom-Aid was able to suppress 99.8 to 99.9% of threads leading to atomicity violations.

CORE and Atom-Aid could be complimentary tools. Atom-Aid seeks to suppress bugs and to aid in debugging by reporting where the atomicity violations occur. CORE could use the bug reports generated by Atom-Aid to better target the buggy areas of code.

3.2.3 AtomRace: Data Race and Atomicity Violation Detector and Healer

AtomRace [53] is similar to Atom-Aid in that it attempts to detect and suppress data races and more generally, atomicity violations in Java programs. To find atomicity violations AtomRace must be given a list of atomic sections to be monitored. If the list isn't supplied, AtomRace can attempt to generate one by invoking static or dynamic analysis tools. AtomRace uses ConTest to noise the program in an attempt to find violations. It suppresses errors by either adding synchronization or by influencing the Java virtual machine scheduler to avoid buggy interleavings.

AtomRace has a second way of attempting to heal/suppress atomicity violations. It can try to add locks (called *healing locks*) to the program to prevent them. The

approach guarantees that no new atomicity violations will be added as long as the list of atomic sections is correct. It is possible that the added lock(s) could produce a deadlock. The authors recommend using a model checker to search for this possibility. Like Atom-Aid above, AtomRace also records information to help developers fix bugs.

The authors envision a scenario in which programs are protected in real time. First, the atomic sections are supplied and the program is analysed by AtomRace. A list of potential locations for noising or lock insertion is produced. These possibilities are model checked and bad entries are removed from the list. AtomRace and the modified list are distributed with the program and run with it to provide bug suppression in real time.

CORE and AtomRace are similar in that they both use ConTest to noise programs to expose bugs. CORE does more, as it attempts to fix both data races and deadlocks, where AtomRace only suppresses data races. AtomRace uses a model checker in an ad-hoc way to look for deadlocks. CORE integrates it fully into the fixing process - using it to look for both data races and deadlocks and updating the search strategy with information received from it. Both require information on concurrently used sections of code to target their search. CORE generates the classes, methods and variables used concurrently from the tools it uses. AtomRace generates or is supplied with the atomicity blocks. Similar to Atom-Aid, AtomRace generates debugging information for the developer. CORE could use this information to help its own search.

3.2.4 Bypassing Races in Live Applications with Execution Filters

In [92] the authors describe *Loom*, a framework allowing users to quickly create and apply patches for data races to live running software. First, the Loom update engine is compiled into the target buggy program and the program is executed. A data race is detected by users or by a race detection tool. Developers analyse the race and use Loom to create a rough patch. Most likely more code than necessary is synchronized by this patch. Loom fills the gap between bug detection and bug fixing. For example, if two lines in two methods are racing, a Loom patch might be to make both methods mutually exclusive. This patch is written in a Loom specific language and then is applied to the running program without needing to restart it. Overhead on the managed program is minimal, $< 5\%$ in tests.

Loom is limited in its scope. It doesn't detect data races or perform any analysis on them. It is up to the programmers to find and triage races to create the rough patch required by Loom. Further, Loom doesn't perform any analysis or checking on the patch when it is running. That is, users have no idea whether the patch is a good one or not. Further, multiple Loom patches can be applied to a program. They could introduce a deadlock because Loom doesn't perform any kind of global analysis or reasoning about the patch interactions. Finally, the authors of Loom note that in suppressing certain interleaving leading to known races, Loom could expose races in other interleavings that become more likely due to the restrictions.

3.2.5 Kivati: Fast Detection and Prevention of Atomicity Violations

Kivati [14] is a framework that detects and avoids atomicity violations in C programs on Linux. It claimed to be the first to do so with low overhead (19%) on commodity x86 processors. This low overhead is achieved in part by using hardware watchpoints found on processors like the x86.

Kivati's detection and prevention algorithm begins with a static analysis of the program to determine the regions it believes are atomic. Annotations are added to the source to identify these regions. When the program is run, Kivati checks variable accesses against the list of believed atomic regions. If two accesses lead to an atomicity violation, Kivati reorders the accesses to avoid it. The framework then records the variables and threads involved so developers can check them. Kivati also maintains a list of false positives that it can safely ignore.

The framework can also be run in bug finding mode. When this is done the overhead is slightly larger as Kivati pauses threads that it believes are in an atomic section of code. By pausing the thread however, it increases the chance of causing an atomicity violation with another thread.

3.2.6 ColorSafe: Architectural Support for Debugging and Dynamically Avoiding Multi-variable Atomicity Violations

ColorSafe [58] is a framework for detecting and avoiding single and multiple variable atomicity violations. It does so by grouping variables together and giving them the same colour - or group ID. Most atomicity violation detection techniques limit themselves to single variable detection because multiple variable detection is too difficult.

ColorSafe handles this naturally by treating all variables of the same color as a single variable. This way, multiple variable atomicity violations are handled naturally.

It operates in two modes, debugging and deployment. In debugging mode it detects and records atomicity violations. It attempts to weed out false positives by applying heuristics. Developers can add their own groups by writing annotations in their code. In deployment mode detection is relaxed to find more violations. When it detects what it believes to be a violation, ColorSafe applies 'ephemeral transactions' to the thread of the violating colour. Ephemeral transactions don't change program state. Their purpose is to delay a thread until the atomicity violating danger has passed.

ColorSafe has one large barrier to adoption though. It requires hardware support for all of its core operations: hierarchical multilevel memory tagging, buffers and logic to handle history items, cache coherence protocol support and transactional memory. Processors with these specifications weren't available to the authors, so they evaluated ColorSafe by simulating the missing hardware with the Pin instrumentation system.

3.2.7 Deterministic Dynamic Deadlock Detection and Recovery

Sammati [31] is a runtime system that automatically and deterministically detects and recovers from deadlocks in multithreaded applications. It is designed to work with POSIX threads as a pre-loadable library. Sammati doesn't require annotations, access to source code or recompilation of the program. It preserves existing lock semantics while deterministically eliminating deadlocks without deadlocking itself. At a high level, memory updates from a critical section are delayed. They take place after all of the locks protecting the critical region have been released. Deadlock are

searched for at the acquisition of each lock. Recovering from a deadlock requires Sammati to select a victim lock and discard all of the updates that would have been performed after the release of that lock.

For Sammati to work efficiently, the following five aspects of their framework must be implemented efficiently.

- Identify critical sections and their updates,
- Isolate and delay these updates until all lock are release,
- Preserve the existing lock semantics,
- Apply the updates when all locks have been released and
- Perform deadlock detection and recovery.

Sammati cannot recover thread local storage data. This data was initialized and manipulated from shared libraries making it too difficult for Sammati to determine where the data was and how to manipulate it. The authors described how they overcame this by using the LLVM compiler infrastructure to instrument store instructions at compile time. Sammati can then efficiently track thread local data. This requires access to the programs source code and the recompilation of that code.

3.2.8 Automated Atomicity-violation Fixing

AFix [40] is a framework that attempts to automate the entire bug fixing process for single-variable atomicity violations. It begins by performing a dynamic analysis of the program using CTrigger, an in-house bug detection tool. CTrigger catalogues all potential atomicity violations and then attempts to reproduce them by noising the program.

It uses a combination of static analysis and static code transformation to generate patches for each bug. AFix examines the call graph and looks for execution paths into the critical region that don't have a lock. It inserts the missing locks along all of these paths. Locks are similarly released on all paths leaving the critical region. AFix guarantees that this locking policy will not introduce new atomicity violations into the program. AFix then attempts to statically merge and harmonize patches where it can. For example, if patch A is completely subsumed by patch B, A is deleted. Overlapping patches can be harmonized - reducing the number of locks used and reducing the chance of deadlock.

AFix doesn't think globally about lock design [56] so it can introduce deadlocks into a program when multiple patches are introduced. Two phases of deadlock detection are used to try and find them. In the end the developer is responsible for trying to fix any deadlocks with debugging information generated by the detection algorithms. Performance profiling is available to help developers manually modify patches to increase efficiency. Their testing phase also has two steps. First, CTrigger runs the patched program with injected noise to search for faulty interleavings. Second, a "general interleaving test implemented by us [40]" is used for the same purpose.

CORE and AFix are similar in that they both use static analysis and noising. Where AFix introduces new locking variables, CORE re-uses existing variables in the synchronized blocks. AFix fixes single variable atomicity violations and may introduce deadlocks. Neither approach thinks globally about lock design. CORE fixes all types of data races and deadlocks fixable by modifying synchronized statements. CORE can also introduce data races and deadlocks as well in code not protected by test cases.

3.2.9 Axis: Automatically Fixing Atomicity Violations Through Solving Control Constraints

Axis [56] is described and positioned as a better AFix. Unlike AFix, it tries to automate the entire process of fixing single and multi-variable atomicity violations. It begins by performing a dynamic analysis of a program using Pecan, an in-house bug detection tool that catalogues all potential atomicity violations.

Unlike AFix, Axis is built upon a theoretical foundation that maximizes concurrency for each atomicity violation. It does this by using a branch of control theory called the 'supervision based on place invariants' (SBPI) as the theoretical foundation. Atomicity violations are fixed by modelling the concurrent properties of the program as Petri nets and then solving a set of control constraints on these nets. A constraint solver is used to add missing locks around the critical region.

Like AFix, Axis doesn't reason globally about lock design either. Instead, the program is instrumented by another tool (like Gadara [86]) that dynamically detects and avoids deadlocks introduced by Axis. Gadara imposes runtime overhead on the program, that never exceeded 10% in tests. It uses discrete control theory to detect deadlock situations and then dynamically delays thread executions to avoid them.

3.2.10 Automatic Repair for Multi-threaded Program with Deadlock/Livelock using Maximum Satisfiability

One way to try and avoid deadlocks is to use trylocks instead of regular locks. A trylock tries to acquire a lock, and if it fails, executes a failure block where remedial steps can be taken - like releasing locks and trying again. While this helps, livelocks can occur. Livelocks occur when threads are active, but not making progress. (Imagine two people trying to pass each other in a narrow corridor. They both step the

same way over and over - repeatedly getting in each others way. They are locked as neither is advancing, but live because both are moving.)

In this paper [54], the authors propose a method to fix deadlocks, livelocks and deadlivelocks (deadlocks hereafter) in concurrent programs. First, a static analysis of the program is performed to find cyclic lock dependencies that could result in deadlocks. These dependddencies are transformed into a boolean satisfiability problem (SAT). Weighted partial maximum satisfiability is used to find the minimal fixes for each deadlocks.

Similar to CORE, the framework includes a number of constraints to reduce the search space. The only code changes that are made are to turn locks into trylocks and the reverse. When a trylock becomes a lock, the false branch is disabled. When a lock becomes a trylock, the false branch rolls back the locking. This roll back can be complicated so the framework prefers to change trylocks to locks. When a trylock is created, the roll back path is minimized in terms of function calls, shared variable updates and lock acquisitions.

Static analysis can lead the framework to try and fix false positives. It can only fix deadlocks fixable by changing locks to trylocks and the reverse. Control flow isn't changed. The framework cannot automatically handle complicated rollbacks. Human intervention may be required. They also assume the competent programmer hypothesis implicitly when they assume each lock is properly unlocked. No locks are added or removed.

3.2.11 Fully automatic and precise detection of thread safety violations

This paper [67] describes a framework that detects data races, atomicity violations and deadlocks. It consists of a test generator that generates and runs tests on different threads to exercise the class under test (CUT). It records all test combinations that result in the program deadlocking or generating an exception. Said deadlock or exception is marked as real real when it cannot be triggered by any single threaded sequence of calls.

The framework only detects bugs from one object by concurrently calling its methods. Multi-object bugs aren't covered. Further, it only detects bugs when the program deadlocks or generates an exception. As it is designed to work on Java programs, this amounts to querying the Java virtual machine.

3.2.12 Grail: context-aware fixing of concurrency bugs

A Petri net is a graph with two types of nodes - places and transitions. It is similar to control flow graphs except that transitions are nodes instead of edges. Program execution is simulated on a Petri net by imagining that a token on the place node represents the current statement being executed. For concurrent programs, multiple Petri nets are used - one for each thread. The state of a Petri net (and thus the program) is the position of all of the tokens at a given time.

Grail [55] is a framework that fixes data races, deadlocks and atomicity violations. Unlike many other frameworks, Grail creates fixes that are both optimal, where each added lock synchronizes the minimal amount of code, and correct, so that the sum of local fixes won't introduce any new bugs.

Like many approaches, it fixes bugs by adding additional locks. Input consists of

the program and bug reports in terms of the lines of code and memory states involved. Grail models the program as a Petri net followed by modelling the bug(s) as a Petri net as well. Mixed integer programming is used to find the constraints that transform a faulty execution into a correct one. A framework called Supervision Based on Place Invariants is then invoked to implement the fix. In evaluations, Grail outperforms Axis and AFix by 40% or or more.

Chapter 4

CORE Framework

4.1 Introduction

This chapter describes the CORE framework and how it was evaluated. CORE's bug fixing algorithm (Section 4.2) is described in detail. This is followed by a discussion on the way in which heuristic, search-based, bug fixing frameworks work (Section 4.3). Finally we finish this chapter by describing how CORE was evaluated (Section 4.5).

4.2 The CORE Framework

CORE (COncurrent REpair framework) is a contribution to filling the gap in concurrent program repair. It is a search based software engineering application driven by a genetic algorithm without crossover (GA¬C) that evolves fixes for deadlocks and data races in concurrent Java programs. The relationships between the four instances of the framework are shown in Figure 4.1.

The focus of the following sections is on explaining the inner working of CORE. At a high level CORE divides the fixing process into 3 phases: analysis, repair and

Figure 4.1: Instances of the CORE framework. ARC is prior work. ARC-OPT, CORE-MC and CORE-IMC are described and evaluated in this thesis.

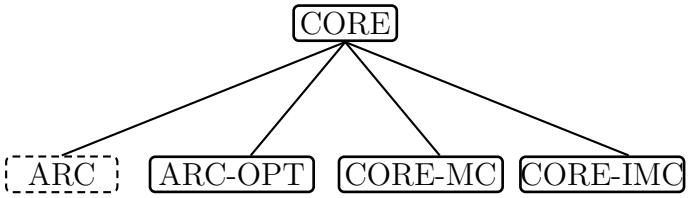
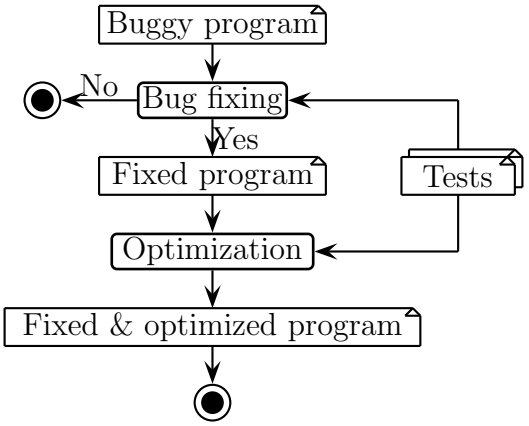
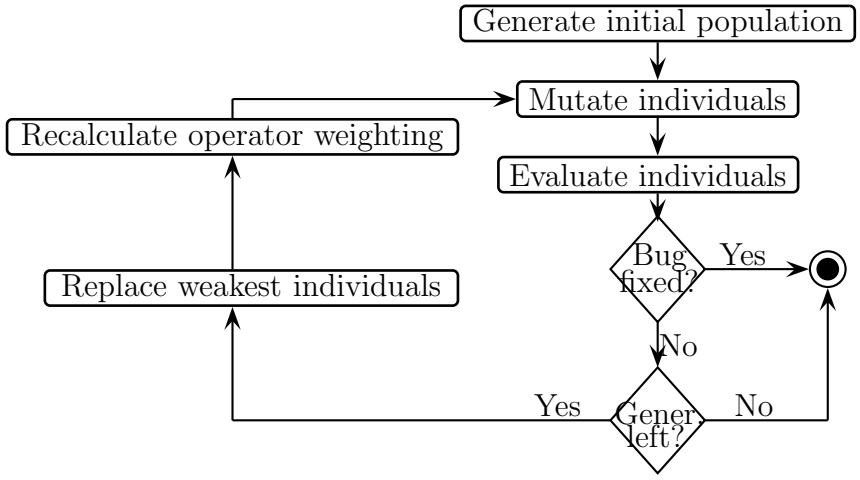


Figure 4.2: High-level overview of the two phases of operation of the CORE framework.



optimization (Figure 4.2). Work done during the analysis phase is critically important to CORE's success. This is where the search space is narrowed by finding the classes, methods and variables used concurrently. After the analysis the GA¬C proper begins its work. An outline of the GA¬C algorithm is in Figure 4.3. Optimizations added to the different instances of the framework (ARC-OPT, CORE-MC and CORE-IMC) are explained in their respective chapters. Only the base framework common to all is detailed here.

Figure 4.3: The CORE framework uses an Evolutionary Strategy to fix data races and deadlocks in Java programs.



4.2.1 ConTest

Before describing the inner workings of the CORE framework, we make one detour to describe the ConTest [24] thread noising tool. ConTest is a concurrent testing tool that forces different thread interleavings to occur. When a program instrumented by ConTest runs, the instrumentor makes calls to the noising heuristic module. This module randomly delays the execution of the thread in questions, increasing the chance that different thread interleavings will be executed. This variable threading increases the chance that one or more tests will fail. ConTest can detect data races, deadlocks and other exceptions.

ConTest has other features not used by CORE. These include simulating variable network loads by using the same kind of noising technique, supporting parallel and sequential coverage models and replaying previous tests for debugging and regression support.

4.2.2 Genetic Algorithm Details

CORE is powered by a genetic algorithm without crossover (GA¬C). When ARC was initially conceived - long before this thesis started - it wasn't clear to the author how to include crossover into the algorithm. If crossover occurred at an arbitrary line of code, how is variable scoping and bracket matching maintained, for example. Once this thesis was decided on, the emphasis was on tool integration, state space constraint, code optimization and evaluation. Crossover wasn't part of the goals of the thesis (Section 1.2).

Genetic algorithms operate on a population of entities. In CORE, each member of the population (its representation) consists of the source code to be mutated, along with the details of the mutations applied and how successful each mutation was. For example, a member of the population had an EXSA (EXpand Synchronization block After) mutation applied in the first generation and ran successfully (bug free) 30 out of 50 times. The other 20 times a deadlock or data race occurred. In the second generation, an ASAT (Add Synchronization Around a Statement) mutation was applied and ran successfully 10 times out of 50. When CORE completed its run, it left behind all of the mutants programs for each member for each generation. Additional information including which mutation was applied and how many runs succeeded for each member for each generation are recorded in the log file.

Fitness functions are used to determine how successful each mutant program is. In CORE each program is scored in a range of 0 to 1000 points, in proportion to the number of successful runs it achieves. For example, if 30 out of 50 ConTest runs are successful, the score is 600. If 60 out of 80 are successful, the score is 750. Fitness determined by model checking is similar. The score range is still 0 to 1000 and the score is determined by the search depth at which an error is first encountered compared to the maximum search depth. For example, if an error is encountered at

a search depth of 40 and the maximum search depth is 50, the fitness score is 800.

4.2.3 Setup: Time-out Generation and Search Space Pruning

Before the main algorithm begins, the CORE framework determines the time-out value and gathers targeting information. The target program is instrumented by ConTest and is run a large number of times (usually $10 \times 15 = 150$ times.) to determine its average running time. The average running time is multiplied by a configurable value (say, by 4) to become the time-out interval for the run. Any program run that exceeds this duration is stopped and recorded as deadlocked or timed out based on the feedback it receives from the Java runtime. During this process ConTest generates a list of *classes* and *variables* found to be used concurrently. CORE uses this list to better target the concurrent sections of code.

Static analysis by Chord, analysis by JPF and function header scanning respectively were added to ARC-OPT, CORE-MC and CORE-IMC and are described in Sections 5.3.1, 6.2.1 and 7.3.

4.2.4 Generate Mutants

In the first step of Figure 4.3, the members of the population are created or updated. During the first generation the incorrect program is copied into each member of the GA¬Cs population. In later generations the program from the previous generation for each member is used.

CORE works its way through each member of the population in turn. First, it exhaustively generates all mutants for a member. Mutants are created by TXL [17] scripts. TXL is a pattern matching and replacement language. The complete list of operators used is in Table 4.1. Note that operators that create mutants may have

Table 4.1: Set of mutation operators used by the CORE framework.

Operator Description	**Acronym**
Add a Synchronized block Around a sTatement	ASAT
Add the Synchronized keyword In the Method header	ASIM
Add a Synchronized block around the code in a Method	ASM
Change the Synchronization order of two nested synch. blocks	CSO
EXpand a Synchronized region down by one line (After)	EXSA
EXpand a Synchronized region up by one line (Before)	EXSB
Remove a Synchronized statement Around a Synchronized block	RSAS
Remove Synchronization Around a line of code (Variable)	RSAV
Remove Synchronized keyword In Method declaration	RSIM
Remove a Synchronization block around the code in a Method	RSM
SHrink Synchronization block by one line from the end (After)	SHSA
SHrink Synchronization block by one line from the beginning (Before)	SHSB

Table 4.2: Using the Add Synchronization Around a Method (ASM) operator places a synchronization block around all of the code in the method.

Before	**After**
function DoPhysics {	function DoPhysics {
	synchronize(LockOne){
pick up chalk	pick up chalk
...	...
}	**}**
	}

multiple instantiations, depending on the class/method/variable targeting information available. Tables 4.2, 4.3 and 4.4 show examples of how three of CORE's operators, ASM, EXSA and CSO, mutate code.

4.2.5 Mutate Individuals

After a member's mutants are generated, the CORE framework selects a type of mutation (e.g. EXSB) and then an instance of it (e.g. 4^{th} mutant generated) from those available. The program source is copied to the compilation directory followed

Table 4.3: The EXpand Synchronization After the block (EXSA) operator extends the synchronization block down to encompass the next line of code.

Before	**After**
function DoPhysics {	function DoPhysics {
synchronize(LockOne){	**synchronize(LockOne){**
pick up chalk	pick up chalk
}	pick up eraser
pick up eraser	**}**
...	...
}	}

Table 4.4: The Change Synchronization Order (CSO) operator flips the order of the locking variables for two nested synchronization blocks.

Before	**After**
function DoPhysics {	function DoPhysics {
pick up chalk	pick up chalk
synchronize(LockOne){	**synchronize(LockTwo){**
synchronize(LockTwo){	**synchronize(LockOne){**
pick up eraser	pick up eraser
...	...
}	**}**
}	**}**
...	...
}	}

by the mutant file. It is possible that the mutant isn't valid. For example, a new synchronization block could have been added that synchronizes on an out of scope variable. CORE 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 occurs or CORE runs out of mutants. If CORE runs out of mutants it resets the program to the previous generation, assigns a fitness of 0 and moves on to the next member of the population. If a reset isn't possible or if this problem occurs in the first generation, CORE writes an entry to the log explaining this and exits.

4.2.6 Evaluate Individuals

Once a compilable mutant is created, it must be evaluated. The fitness score was described in Section 4.2.2 above.

How do we assure ourselves that a proposed fix really fixes all of the known bugs? For ARC and ARC-OPT, it is possible that a proposed solution could still contain a bug in a thread interleaving that escaped detection. For CORE-MC and CORE-IMC the bug could exist at a search depth deeper that the model checker explored. To increase confidence in a fix, take the base number of ConTest runs and multiply it by an additional safety factor. All proposed fixes are run through ConTest this many more times to give us additional confidence that the fix works. This is done for all versions of the framework. 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 CORE runs out of generations.

4.2.7 Check Ending Condition

If any proposed fix passes the tests from the previous section, that member of the population is declared correct and its program is written to the output directory for the user. If no member program passes all test, the search continues until CORE runs out of generations.

4.2.8 Replace Weakest Individuals

We believe the *competent programmer hypothesis* [2] applies when fixing concurrency bugs. Programmers strive to create correct programs. Programs with bugs in them are nearly correct so that the distance in the search space from an incorrect program to a correct one is small and solvable. Even with a smaller search space, evolutionary

algorithms may evolve candidate solutions that stray down paths leading to little or no improvement. In some cases the evolution may make the program worse. To encourage members to explore the higher fitness parts of the state space there is an option to restart or replace the lowest x percentage (say 10%) of individuals if they perform poorly for too many generations. Two replacement strategies are used. First, the member is replaced by a random individual from the upper y (say 25%) percent of the population. Second, the member is replaced by the original incorrect program.

4.2.9 Recalculate Operator Weighting

CORE leverages historical information on how successful different mutation operators have been and about the relative dominance of data races versus deadlocks. Operators raising the fitness of the population or reducing the frequency[1] of data races or deadlocks are given additional weight in the selection process. This weighting never reduces the selection chance to zero. There is separate weighting for each bug type (deadlock fixing and data race fixing) within which the relative success of operators is recorded. Heuristic searches are dynamic. A successful operator at the beginning of a search may become detrimental at the end. To avoid this, a sliding window of the previous n generations is used to adapt weightings to what has happened recently.

4.2.10 Example

With ICHEP[2] rapidly approaching our working physicists decide to evolve a solution to their chalk grabbing difficulties. After rolling numerous dice they create Table 4.5. It shows one evolutionary path to the fix, focusing on constructive steps for clarity.

[1]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.

[2]International Conference on High Energy Physics

Table 4.5: Evolving a fix for the working physicists deadlock in CORE. The original code is in the left column. A mutant that doesn't improve fitness is in the middle column. By expanding the synchronized region up one line a fix is found in the right column.

Original	**Partial Fix**	**Full Fix**
	Add synch. around statement	Expand synch. before

function DoPhysics {
  pick up chalk
  pick up eraser
  write
  put down eraser
  put down chalk
}
	function DoPhysics {
	  pick up chalk
	  synchronize (eraser){
	  pick up eraser
	  }
	  write
	  put down eraser
	  put down chalk
	}
		function DoPhysics {
		  synchronize (eraser){
		  pick up chalk
		  pick up eraser
		  }
		  write
		  put down eraser
		  put down chalk
		}

On the left is the original pseudo-code. In the middle column the *Add Synchronization Around Statement* (ASAT) operator is applied to a random statement. CORE considers all statements containing concurrent variables. Any concurrent variable could be a lock a developer created but didn't use. As mutations are simple single steps, synchronizing this statement is a valid mutation.

Observe that the mutation leading to the middle column doesn't affect the bug at all. It still appears as frequently as in the left column. Both columns have the same fitness. We keep this mutation because it increases the diversity of the population. Quoting [72]: "... neutral mutations that leave fitness unchanged are considered to be beneficial – improving the system's robustness and its ability to discover evolutionary improvements." This has an analogy in nature: One mutation by itself may do nothing. In cooperation with a second (or third, ...) they could become beneficial or destructive. In this example the mutation is beneficial. Expanding the synchronization block upward by one statement (last column) creates a fix for the

Table 4.6: Evolving optimizations for the deadlock fix found by CORE. The fix found in the left column is to synchronize all of the code. It works but serializes the code. An attempted optimization in the middle column reintroduces the deadlock, so it is rejected. In the right column the synchronize block is shrunk by two lines, leading to a better (but not optimal) solution.

Evolved Fix	**Optimization creating a deadlock**	**Successful Optimization**
Add synch. around a method	Shrink synch. before	Shrink synch. after *-Applied twice*

function DoPhysics {
synchronize(chalk) {
pick up chalk
pick up eraser
write
put down eraser
put down chalk
}
}
	function DoPhysics {
	pick up chalk
	synchronize(chalk) {
	pick up eraser
	write
	put down eraser
	put down chalk
	}
	}
		function DoPhysics {
		synchronize(chalk) {
		pick up chalk
		pick up eraser
		write
		}
		put down eraser
		put down chalk
		}

deadlock.

As CORE adds, grows and reorders concurrent blocks, it could add unnecessary synchronization. These blocks hurt performance by serializing code that should be running concurrently. Phase 2 attempts to fix this by evolving the program to optimize efficiency. It uses a different set of operators specialized to shrink and remove concurrency blocks. A different fitness function is used. It derives a score based on execution time and the number of voluntary context switches[3]. Lower numbers in both lead to higher scores.

Table 4.6 demonstrates how a fix found in the bug fixing phase[4] is optimized. In the left column is the fix from the first phase. An optimization is applied that reintroduces a deadlock in the center column. This can happen when the synchronization

[3]A *voluntary context switch* can occur when a thread of execution must wait on a lock. It gives up it's remaining time on the processor to another thread.

[4]Intentionally different from the fix found in table 4.5.

blocks are shrunk or removed. For every proposed optimization CORE must check for the reintroduction of deadlocks and data races. This is done the same way as in the fixing phase. CORE rejects the proposed optimization in the middle column. The right column contains the final solution found by CORE. Note that it isn't perfect[5]. *Once again we emphasize that heuristic search is not guaranteed to find the optimal solution.* CORE found a *good enough* solution for eliminating the bug. Running CORE a second, third or fourth time might produce a better solution. Alternatively, comments written by CORE in the source code (not shown for brevity) could help a developer optimize the fix by hand.

Note that CORE isn't designed to fix gross misunderstandings of concurrency. We assume the competent programmer hypothesis. When concurrency is missing, for example, or the code contains conceptual errors, CORE isn't able to fix it.

4.3 COREs Search Strategy

With the framework explained in depth, we spend this section comparing and contrasting CORE's approach to other frameworks that suppress or fix bugs in both sequential and multi-threaded programs. First we look at GenProg [49]. It is a well known and successful framework for the automatic fixing of sequential bugs in programs. We contrast it with RSRepair [69] and discuss how the decisions made to constrain the search space affect their performances (Section 4.3.1). In the following sections we look more in-depth at frameworks that find, suppress or fix parallel bugs, AtomAid [59], AtomRace [53], Falcon [66] and AFix [40] (Section **??**) and compare and contrast them with the CORE framework.

[5]Write could be removed from the synchronize block.

Table 4.7: Average generation a fix was found for the test programs from the papers written on GenProg.

Paper	**Year**	**Average Gen.**
A GP Approach to Automated Software Repair	2009	3.6
Automatic, Efficient and General Repair of Software Defects Using Lightweight Analysis	2010	2.1
Automated Program Repair through the Evolution of Assembly Code	2010	1.6
Automatic Program Repair with Evolutionary Computation	2010	2
GenProg: A Generic Method for Automatic Software Repair	2011	1.8

4.3.1 Does Heuristic Search Make a Difference?

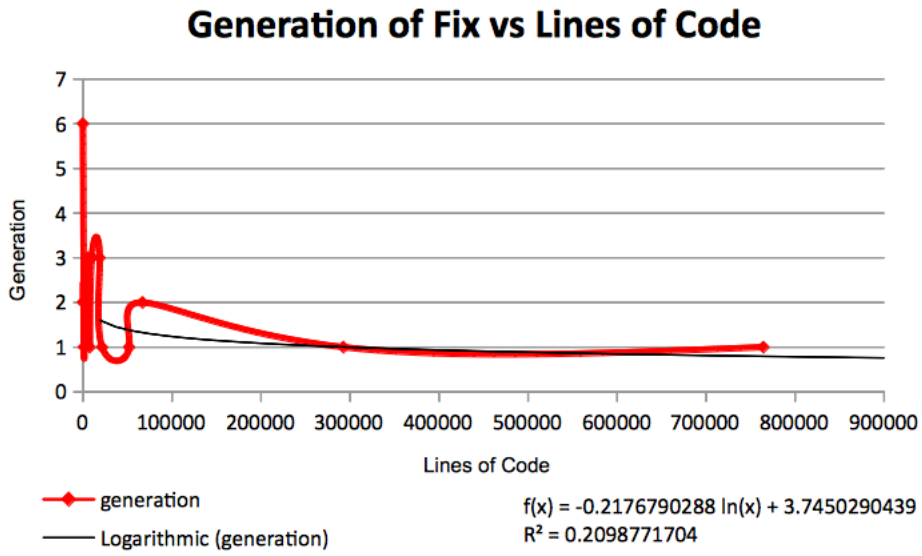
Does genetic programming work well on automated program repair? This is the question posed by a recent paper [69]. Specifically, does the genetic programming approach used in the current state of the art in single threaded program repair, GenProg [49], do better than random? The authors of [69] conducted a study where they duplicated GenProg but removed fitness-based selection and replaced it with random selection. Overall their framework, RSRepair (Random Search Repair?) performed better than GenProg. RSRepair fixed more bugs and was faster.

A careful analysis of an earlier version of GenProg [50] revealed that randomly constructed patches fixed 62% of the programs used in the paper[6]. That is, the fix for the program was found in GenProg's initial randomly generated population more than half of the time. Twenty percent of fixes were found after one generation of the evolutionary process, 6% after two and the rest after more.

On average the fix is found in the randomly created population before any genetic

[6]Critical analysis of a paper for the 5010 course from 2011, unpublished.

Figure 4.4: There is no relationship between the generation of a fix and the size of the programs in lines of code in [50].



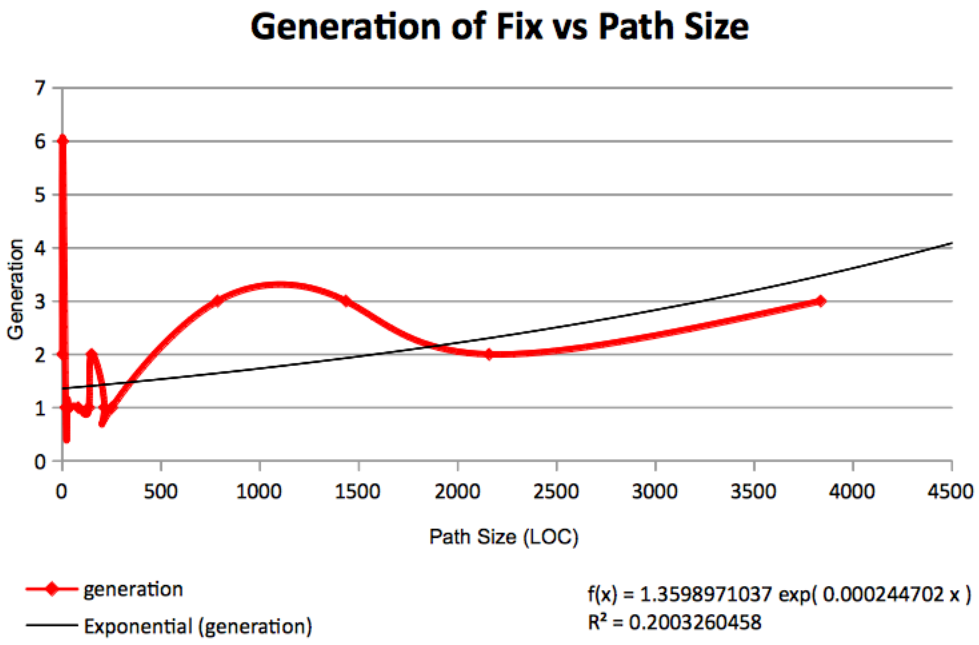
programming techniques are applied or after one generation of the search. This observation applies to most of the GenProg related papers and is shown in Table 4.7[7].

GenProgs creators are aware of this issue:

> "One concern about our results to date is the role of evolution. Most of our repairs result from one or two random modifications to the program, and they are often found within the first few generations or occasionally, not at all. We have conducted some experiments using a brute force algorithm ... and random search .... Both these simpler alternatives perform as well or better than the GP on many, but not all, of our benchmark programs. ... However, thus far GP outperforms the other two search strategies in cases where the weighted path is long ...." [87]

[7]Many programs are reused from one paper to another.

Figure 4.5: There is no relationship between the generation of a fix and the critical path size of the programs in [50].



Further analysis of [50] turned up two more interesting facts.

1. Figure 4.4 shows there is no relationship ($R^2 = 0.2$) between the generation a fix was found in and the program size in lines of code. In other words, big programs are not harder to fix.

2. Figure 4.5 shows there is also no relationship ($R^2 = 0.2$) between the generation the fix was found in and the size of the critical execution path (weighted path) containing the bug.

GenProg and RSRepair operate in a similar way - they stop as soon as they fix one bug. For the programs in [49, 69] each has between 2 and 44 bugs and each has between 70 and 8000 test cases. On average each bug has roughly 100 test cases.

Test cases consist of both positive tests preserving existing functionality and negative tests demonstrating the bugs to be fixed. Presumably most tests preserve existing functionality[8]. Every member is tested against 10% of the relevant tests[9]. Full testing is only invoked when this subset is passed.

In GenProg, fitness is based on the total number of tests passed, with negative tests having twice the weight of positive tests. It may be the case that GenProg is preferentially selecting mediocre programs. A fix may require changes that harm functionality protected by the positive test cases. When this happens the program receives a lower score. When many tests are concentrated in a small area, it is possible that one change will cause many of the positive tests to fail. Contrast this with the principle that mutational diversity is a good thing. This change (insertion, deletion, ...) may be a necessary for a fix in a future generation. But, due to the lower fitness GenProg will selectively remove this diversity - the lower scoring mutant - from the population. RSRepair won't. It is impartial to decreases in fitness leading to increased diversity. This may explain why RSRepair does better.

For GenProg and RSRepair there is a rough relationship between the number of bugs in a program and the fix rate. At 44 bugs the chance of finding a fix for *one*[10] bug is 100%. When the number of bugs shrinks to 2, success drops to 7 or 13%. (R^2 is 0.57 for GenProg and 0.35 for RSRepair.) As the program is almost correct and has many positive tests, GenProg and RSRepair start at or near a local fitness optima that must be escaped. When there are many bugs it is likely that one random change will fix one of them and complete the algorithm. Conversely, when there are few bugs, these algorithms flounder because they have to 'get lucky' to target the

[8]No information is given on the counts of positive and negative tests.

[9]An optimization used by both GenProg and RSRepair.

[10]Both GenProg and RSRepair declare success after fixing one bug, regardless of how many there are in the program.

areas at and around the bugs. Once they do, the change made could cause multiple positive tests to fail. These members lose fitness and are preferentially selected out of the population.

GenProg's goal is to fix all kinds of bugs. Casting a wide net constrains how it can attack the problem. In this sense the use of fitness based selection might harm mutational diversity and its ability to search the space of program fixes to find a solution.

4.3.2 CORE's Search Strategy

CORE is similar to GenProg and RSRepair in that it often finds fixes for deadlocks and data races within the first generation. This suggests that the difficulty of the repair problem is to a great extent defined by the constraints of the approach. They determine what is fixable and the search space of the problem. If the constraints are reasonable, then the bug fixing process could be 'easy'. When the search space is defined, the algorithmic implementation determines reachability and efficiency.

This thesis attempted to tackle both aspects of the problem as the CORE framework was developed. In terms of constraints, more analyses were added to find the classes, methods and variables used concurrently. Efficiency was improved by reducing the amount of work that the framework had to do to a minimum by removing incorrect mutants and curbing the number of uncompilable programs, amongst others.

4.4 Synchronizing Run()

In the evaluation that follows, 4 programs were fixed by synchronizing the $run()$ method. This has the undesirable side-effect of turning the parallel program into a serial one. As this defeats the purpose of parallelizing the program in the first

Figure 4.6: In the Account program, the *transfer* method can lose updates because of a lack of synchronization on the *ac* variable.

```
synchronized void transfer(Account ac, double mn){
  amount-=mn;
  ac.amount+=mn;
}
```

Figure 4.7: In the Account program, synchronizing ac in the *transfer* method fixes the data race.

```
synchronized void transfer(Account ac, double mn){
  amount-=mn;
  synchronized(ac) {
    ac.amount+=mn;
  }
}
```

place we investigated why this occurred and what additional steps could be taken to find fixes that maintain parallelism. CORE applies syntactic changes to the source by modifying *synchronize*() statements. It is not designed to apply template-like transformations. In this way, synchronizing *run*() is a perfectly valid mutation. As a first step, a configurable toggle was added to exclude *run*() from being synchronized. When this was done, CORE couldn't find fixes for the Account, Airline and Lottery programs[11].

CORE relies on its tools to find the classes, methods and variables used concurrently. Primitive types are removed from the variables list, leaving legal synchronizable objects. Account is an example of a program in which ConTest and Chord find the proper method to synchronize, $transfer$, but not the variable, *ac* (Figure 4.6). After adding the ability to find the variables declared in function definitions in CORE-

[11]One of the fixes found for PingPong and Deadlock was to synchronize *run*(). They are still fixable when *run*() was excluded, as other methods can be synchronized.

Figure 4.8: In the Airline program, all of the parallel code is in the *run()* method. All of the variables used in *run()* are primitive types.

```
public void run() {
  Num_Of_Seats_Sold++;

  if (Num_Of_Seats_Sold > Maximum_Capacity)
    StopSales = true;
}
```

IMC, CORE could fix Account again without synchronizing run() (Figure 4.7) (See Section 7.2).

For Airline, the parallel code is all contained within *run* (Figure 4.8). It uses only primitive variables, while *run* itself has no parameters. The analysis step correctly reports that no (synchronizable) variables are found and that only the *run* method is used concurrently. CORE finds the only fix possible - synchronizing *run*.

In Lottery three methods race on the *randomNumber* variable (Figure 4.9). The analysis tools find synchronizable variables, but not all of the synchronizable methods. They find only that the *generate* method is used concurrently. Without the *present* and *record* methods, all of the lines racing on *randomNumber* cannot be synchronized to fix the data race. None of the three methods receive arguments, so searching for them is of no help. With incomplete information, CORE finds the only fix it can - synchronizing *run*. This problem is in part a limitation of the tools used. Different analysis tools (static, dynamic, ...) could find the other two methods. Even then CORE would still have some difficulty because the lines in the three methods must synchronize on the same lock. Currently CORE randomly selects a lock for each added synchronize statement.

One lesson learned from this is that well structured programs are more likely to be fixed by the CORE framework. Programs that properly delegate functionality to

methods and encapsulate data as arguments to these methods are more likely to be fixable. An example of this is the Account program. Counter-examples include the Airline and Lottery programs. Airline uses only primitive types and is run from the constructor. In Lottery the functions don't accept any arguments. Programs that use non-primitive variables are also more likely to be fixed. Airline is an counter-example of a program that only uses primitive types.

4.5 Evaluating CORE

There is a large amount of scholarly work on finding and suppressing concurrent bugs. One would think it is easy to assemble a suite of Java programs with data races or deadlocks to test against CORE. In reality it is difficult. Even existing benchmarks from the concurrent bug finding/suppressing/fixing literature aren't of much help.

Java Grande[12] for example, is a suite of benchmarks "measuring and comparing alternative Java execution environments in ways which are important to Grande applications." Of interest was the multi-threaded benchmark. It contains parallel Java programs moldyn, monte-carlo and raytracer. These have been studied extensively in the parallel bug detection literature [39, 42, 66, 68, 73]. However, good for bug detection doesn't equate to good for bug fixing. The Java Grande approach was to take a single threaded program and run it on many threads. Predictably there are bugs. One such bug occurs when different threads access the same statically declared object without locking. Java Grande programs fall under the 'gross misunderstanding of concurrency' category as the programs were not written with parallelization as a goal. CORE is not designed to fix them. In other candidate programs the data races detected are benign. As there is nothing wrong with the program, there is no test

[12] http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html

Table 4.8: The set of programs in the benchmark used to evaluate CORE.

Program	**Lines of Code**	**Bug Type**	**Can Fix?**
Account	165	Data race	Yes
Account sub-type	209	Data race, deadlock	Yes
Accounts	75	Data race	Yes
Airline	931	Data race	Yes
Bubblesort2	104	Data race	Yes
Buffer	319	Data race	No
Deadlock	109	Deadlock	Yes
Linked list	243	Data race	Yes
Lottery	157	Data race	Yes
Pingpong	143	Data race	Yes
Readers Writers	170	Data race	Yes
Cache4j	2706	Data race	No
String buffer	1278	Data race	No
Travelling sales	702	Data race	No

case to demonstrate the non-existent problem and nothing to fix.

In order to evaluate CORE, a number of programs were selected from different sources: eight programs from the IBM Concurrency Benchmark [26], 2 from pjbench[13] (Cache4j and TSP), a benchmark created by the Program Analysis Group at Georgia Tech[14] and one (StringBuffer) from the CalFuzzer [42] benchmark. Programs were also downloaded from the Software-Artifact Infrastructure Repository [20]. For the IBM benchmark we chose 6 programs containing bugs CORE can fix and 2 CORE cannot as a sanity check. Their properties are summarized in Tables 4.8, 4.9 and 4.10.

CORE is designed to be flexible. Table 4.11 describes the configuration options and values used in our evaluation. Parameters were not optimized by project in the benchmark. Standard values used in the literature were identified and incorporated.

[13]http://code.google.com/p/pjbench/, retrieved April 2013
[14]http://pag.gatech.edu/software, retrieved April 2013

Table 4.9: Class, method and variable counts of the benchmark programs.

Program	**# Classes**	**# Methods**	**# Variables**
Account	3	8	9
Account sub-type	6	19	16
Accounts	2	5	9
Airline	1	3	8
Bubblesort2	2	3	4
Buffer	5	22	47
Deadlock	2	10	8
Linked list	5	22	26
Lottery	2	8	8
Pingpong	4	13	9
Readers writers	6	17	16
Cache4j	20	46	133
String buffer	1	20	4
Travelling sales	4	17	35

In the following subsections each program and their concurrent bug are described. Source listings of the bugs and the fixes found by CORE are deferred to Appendix 1.

Account

The Account program simulates transactions at a bank. Transactions are run on threads representing customer bank accounts. These customer threads contain a data race between the $transfer$, *depsite* (sic) and *withdraw* methods, all of which are called from the threads *run* method. During execution, account thread i invokes the $transfer$ method of account i + 1. On the next iteration of the loop, thread i + 1 invokes *depsite* and *withdraw*. Because of a lack of synchronization, the $transfer$ can be overwritten by the *depsite* or *withdrawal* method calls. In plainer language, \$99 is deducted from sender i, but isn't received by the receiver $i + 1$ as it is overwritten by the deposit of \$300 or the withdrawal of \$100.

CORE consistently fixed Account by synchronizing the *run* method. After adding

Table 4.10: Concurrent properties of the benchmark programs.

Program	**# Critical Regions**	**% of Statements in Critical Regions**	**# Threads**
Account	3	2.4	10
Account sub-type	6	3.8	11
Accounts	2	5.3	10 - 110
Airline	0	0	100
Bubblesort2	2	17.3	1000
Buffer	5	12.5	24
Deadlock	4	25.7	15
Linked list	2	3	2
Lottery	4	9.6	333
Pingpong	5	11.2	120
Readers writers	16	9	4
Cache4j	31	3.7	4
String buffer	24	16	2
Travelling sales	6	25	2

the ability to scan function definitions for synchronizable variables and excluding *run*, CORE was able to fix Account by synchronizing lines in the $transfer$ method.

Account Sub-Type

Account Sub-Type [23] is a modification of the Account program. In it an abstract class represents a basic account object. Two child types were created: *business* and *personal* accounts. In the $Business.transfer$ method the transfer was properly synchronized while $Personal.transfer$ was not. As with Account above, the lack of synchronization on $Personal.transfer$ caused a data race between the $transfer$, *deposite* and *withdraw* methods. CORE fixed it the same way it fixed Account - by synchronizing within the $Personal.transfer$ method.

This wasn't the end of the story though. After fixing the data race from $Personal.transfer$, the JPF model checker found a deadlock where ConTest didn't.

Table 4.11: The set of parameters that CORE uses along with their descriptions and values.

Parameter	**Description**	**Value**
# Runs	How many times each program is run	30
Search depth	How deep the model checker searches	50
ConTest Runs	Test suite executions per gen. per member	15
Validation Mult.	Multiplier on ConTest runs when validating potentially correct programs	10
Timeout Mult.	Time multiplier on program before timeout	10
Generations	Maximum number of generations of the GA¬C	30
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

JPF reported that the $account.transfer$ method calls for personal and business accounts in $Manager.run$ deadlocked. Account N calls $transfer$. First it locks itself then it attempts to lock account N+1. At the same time account N+1 calls $transfer$ to lock itself and then lock object N+2 and so on. JPF detects a deadlock because each account can be in $transfer$ leading to a circular deadlock.

The only way to fix this bug is to serialize access to $transfer$. Both calls to $transfer$ in run must be synchronized on a global lock, and depending on the mutations, the same global lock. This is a hard fix for CORE to find. Synchronizing run (if allowed in the configuration file) also fixes the bug - at the cost of parallelism.

Accounts

Accounts simulates a bank by performing transactions on many threads. It maintains an overall sum of the amount of funds transferred in a globally declared $Bank\_Total$ variable. Account threads have unrestricted access to the global balance leading to data races. These threads access the global sum through the *service* method. In *run* an account repeatedly adds a random amount to a balance and the global running sum. Multiple threads race on $Bank\_Total$ causing the global sum to disagree with the sum of individual account balances. CORE finds multiple fixes. The optimal one is to synchronize the $Bank\_Total$ line in the *Service* method. A less optimal fix is to synchronize both lines in the *Service* method or synchronize the method itself.

Airline

Airline is a threaded ticket sales simulator for airlines. It uses a class level variable to keep track of the number of seats sold on an aeroplane. Sales stop when the number of tickets sold is equal to the aircraft capacity. This programs has a race on the $Num\_Of\_Seats\_Sold$ and *StopSales* variables between the main body of code and the *run* methods of the ticket sellers. One or more seats can be sold past the capacity of the aeroplane on different threads after *StopSales* is set to *true*.

Initially Airline was classified as unfixable by ARC. After reviewing it again for this thesis it was clear that it should be fixable by ARC-OPT. Sadly, ARC-OPT couldn't fix airline either, but both CORE-MC and CORE-IMC could. The cause of ARCs failure to fix airline was traced to the fact that all of the concurrent variables discovered by the analyses were of primitive types. Java doesn't allow synchronization on primitive types, so the mutants generated by both ARCs didn't compile. Both ARC and ARC-OPT generated 173 ASAT (Add Synchronization Around sTatement) mutants using primitive types as the locking variables, 15 ASM (Add Synchronization

around a Method) mutants, again using primitives for locking variables and 3 ASIM (Add Synchronization In Method header) mutants. Of the 191 mutants available, only 2 of the ASIM mutants created compilable programs[15]. ARC and ARC-OPT both failed to fix Airline for two linked reasons. First, 189 of the 191 mutants didn't compile. Second, there was a bug in both ARCs that caused them not to consider the ASIM mutants. If ARC or ARC-OPT could never chose an ASIM mutant, they could never create a compilable program and never find a fix. This is important because one of the ASIM mutants did fix the bug.

During the development of the COREs, this ASIM selection bug was fixed. Now both CORE and CORE-INC can fix Airline. One of the features added to the framework was the elimination of primitive types from the list of synchronizable variables - which was all of them. No ASAT or ASM mutants were generated- only ASIM. As described above, the only possible fix was to synchronize the *run* method.

Bubblesort2

Bubblesort2 parallellizes the bubblesort algorithm. It contains a data race on the globally declared array variable *array*. The algorithm creates new threads to perform the swapping in the sort. All of these swapping calls go through the *swpArray* method. This call only has synchronization within the object but not between objects. Multiple objects can be in *swpArray*, simultaneously making changes to the global array and causing data races. CORE fixes the data race by locking array accesses on the *array* variable.

[15]One ASIM mutant attempted to synchronize the constructor, so only 2 of them compiled.

Deadlock

The working physicist deadlock is implemented in the deadlock program in terms of file copying from A to B and B to A simultaneously. The *write* method of each thread attempts to lock the source file, then the destination file in turn. If two threads each lock their source file the program deadlocks. CORE's fix is to synchronize access to the write method. Other fixes CORE found include synchronizing access to *write* in *run* and synchronizing all of *run*. Note that to fix this program it must be serialized.

Lottery

As described in the previous section, because the analysis only determines that the *generate* function is used concurrently, while missing the *present* and *record* methods, the only fix CORE can find is to serialize access to the conflicting methods in *run*().

Pingpong

In Pingpong there is a class level variable called *pingPongPlayer* accessed by all threads. Every thread calls the *ping* method that in turn calls *pingPong*. In it the *pingPongPlayer* variable is set to null for 50 milliseconds. A different thread trying to call *pingPongPlayer.getI*() during this 50 milliseconds generates a *NullPointerException*. CORE fixed this program by synchronizing any method in the chain of calls - *pingPong*, *ping*, or *run*.

Linked List

Linked List is a concurrent implementation of a linked list. It has a data race in the *insert* method. The last line of code, *p._current._next* = ... can be raced on, causing random linking within the list. The fix is to extend the *synchronized*(*this*) block

down one line to properly synchronize the method. Other fixes CORE found include synchronizing more or all of the lines in the method.

Readers-Writers

Readers-Writers is a concurrent implementation of readers and writers operating on a common pool. It has a data race between the readers and writers. Sometimes a reader can be active when a writer is writing. When this occurs, the $beforeRead$ method throws a $java.lang.IllegalMonitorStateException$). The fix is to synchronize the $beforeRead$ method.

Buffer

Buffer has multiple readers consuming from and writers writing to a buffer. It contains a *notify* vs *notify all* bug. If a buffer is full (resp. empty) and a writer (reader) notifies another writer (reader), nothing happens and the program deadlocks. If $notifyall$ had been called instead, a reader thread (writer thread) would activate to consume some content from the full buffer (write some content to the empty buffer). CORE wasn't designed to fix notify vs notifyall bugs so it cannot fix this program.

Cache4j

Cache4j[16] is an in-memory cache for Java objects. Different bug detection research papers have different things to say about Cache4j. In one [65] it has a benign atomicity violation. In another [73] a race over the *sleep* field in CacheCleaner.java was found leading to an uncaught exception. If *sleep* is set to *true* and followed by a context switch to another thread, an uncaught $InterruptedException$ is thrown causing the second thread to crash.

[16]Cache4j, http://cache4j.sourceforge.net/

CORE is unable to fix Cache4j.

StringBuffer

StringBuffer is a modifiable string object in the java.lang package. It contains a data race on the *count* variable in all methods that use it. It occurs very rarely – on the order of 1 in 6000 ConTest runs[17]. When it occurs StringBuffer crashes and throws a *StringIndexOutOfBoundsException*. As CORE uses ConTest, it cannot demonstrate the bug with any regularity causing it to erroneously report the original buggy program as 'fixed'.

Model checking in CORE-MC and CORE-IMC cannot fix the race either. It can improve the search depth at which the model checker finds the bug though. For example, the search depth was improved from 22 to 38 over the course of a run. This suggests the bug will occur less frequently. There are two reasons to be sceptical of this result. First, the data race already occurs rarely. Improving this from 1 in 6000 to 1 in 10000 (say) only makes the bug harder to find. Second, even removing synchronization from functions can improve the search depth! For StringBuffer there isn't a clear correlation between higher search depth and the StringBuffer class being more correct.

The underlying problem in StringBuffer is a complete lack of synchronization at the statement level. Only methods are synchronized. Two different methods can race on any shared variable. This complete lack of statement level synchronization is a gross misunderstanding of concurrency that CORE cannot fix.

[17]The author has seen this exception occur only once.

Travelling Salesperson (TSP)

Once again different papers had different things to say about this implementation of the travelling salesperson algorithm. One detection method [66] stated that the data races in TSP were benign. Another found both benign data races and malignant ones that "... involved updates that could be lost, leading to incorrect results" [85]. There was no specific information on where the bug was or how frequently it occurred when found.

TSP is similar to StringBuffer in that the data race appears infrequently. Where most programs are noised by contest 15 times, it was necessary to noise TSP over 1000 times for every member for every generation for ConTest to expose the bug with any regularity. On top of the rarity of the data race, TSP times out on average once every 300 runs. Timeouts appear about 3× more commonly than the data race, when the timeout is set to 20× the average running time of the program. (The default timeout value is 4 times the average running time of the program.) This results in every potential fix being declared incorrect because a timeout is considered incorrect. In practice ARC and ARC-OPT run until the large number of ConTest runs causes the framework to crash. These timeouts could hide a correct program. Sadly CORE does no better. Java Path Finder is unable to model-check TSP. Overall, the CORE framework cannot fix TSP.

Figure 4.9: In the Lottery program the methods *generate*, *present* and *record* race on the class level variable *randomNumber*.

```
public void run () {
  int i = 0;
  while (i != numOfUsers) {
    generate ();
    for (i = 0; i < numOfUsers; i ++) {
      if (history [i] == randomNumber) break;
    }
  }
  present ();
  record ();
}

protected synchronized void generate () {
  generated [userNumber] = randomNumber = (long) (Math.random
     ()
    * Math.pow (10, MAX_DIGITS));
}

protected synchronized void present () {
  System.out.print ("user " + userNumber + " assigned " +
    (presented [userNumber] = randomNumber) + ".");
}

protected synchronized void record () {
  history [userNumber] = randomNumber;
}
```

Chapter 5

ARC and ARC-OPT

5.1 Introduction

In this chapter we examine the development of ARC and ARC-OPT in Section 5.2. What differentiates them most strongly is the addition of static analysis and the software optimizations added to ARC-OPT described in Section 5.3. Both versions of the framework are evaluated in Section 5.4.

5.2 ARC and ARC-OPT

ARC was successful enough to be written up as a paper [43] accepted by MUSEPAT'13, the *International Conference on Multicore Software Engineering, Performance, and Tools*. That paper was the basis of the previous chapter. Once this thesis was decided on, the first step was to generalize ARC into the CORE framework. Numerous optimizations and bug fixes were incorporated - enough to separate MUSEPAT'13 ARC from what came after, ARC-OPT. These optimizations are described in detail in Section 5.3.

Optimizations and bug-fixes alone are not enough to distinguish ARC from ARC-OPT. What makes the distinction clear is the addition of the *static analysis* of the program to be fixed to the CORE framework and thus ARC-OPT, CORE-MC and all future versions. Static analysis is the automated analysis of the source code. The source is analysed to determine the classes, methods and variables used concurrently. With this information, the search for fixes for deadlocks and data races is more focused on the lines of code used concurrently. Note that the static analysis allows the framework to better target the code used concurrently, but not the code specific to the bugs. Attempting to target the execution path containing the bugs requires some form of dynamic analysis - which CORE doesn't do. The static analysis is described in greater detail in Section 5.3.1.

All of these changes had a large impact on the performance of ARC-OPT. ARC took on average 34 minutes to fix the programs in Table 5.3. ARC-OPT fixes the same programs in an average of 13 minutes.

5.2.1 Summary

Recall that ARC was described in detail in Chapter 4. This section provides a summary of it.

ARC is a framework for fixing data races and deadlocks in concurrent Java programs. It can only fix known bugs demonstratable by test cases. An analysis performed by ConTest determines the classes and variables used concurrently. The search space of potential fixes is narrowed as ARC focuses only on those lines of code containing variables identified by ConTest.

ARC uses a genetic algorithm without crossover (GA¬C) to perform the search. ARC instruments the program with the ConTest thread noising tool and runs it a set number of times, recording the number of successes, data races and deadlocks. Every

member of the GA¬C population is repeatedly noised and is given a higher score for every ConTest run that doesn't demonstrate a deadlock or data race.

Changes to the population are introduced by mutation. All mutants are exhaustively generated for a member of the population for all applicable mutant operators. Once this is completed a mutant is chosen to be applied to the member of the population. First, a type of mutation and a member of that group is selected, for example the 14^{th} ASAT (Add Synchronization Around a sTatement operator) mutant. This mutant is copied into the project. If the program doesn't compile, a different mutant is selected. Once a compilable program is generated it is repeatedly noised by ConTest to expose and exercise different interleavings. This mutant program scores higher for each run that doesn't expose a deadlock or data race. If a program passes all runs (15) for example, the program is executed by ConTest a larger number of times ($10 \times 15 = 150$ runs) to give us confidence the fix is a good one. If this larger number of runs succeeds, ARC copies the fixed program to the output directory and ends.

This process of mutation followed by evaluation continues until a correct program is found or the GA¬C runs out of generations. The GA¬C is a stochastic or guided random search. Every time the framework is invoked, the search will give different results. CORE isn't guaranteed to succeed if a fix is possible and it isn't guaranteed to produce the optimum fix. When a fix is found, the documentation inserted into the source can help guide a programmer to a better fix, if one is possible.

5.2.2 Limitations

We assume the competent programmer hypothesis - competent programmers strive to create correct programs. Bugs are manageable deviations from correctness. By assuming competence, we reason that the fix is a reasonable distance away in the

search space. Expressed another way, the distance from 'here' (the bug) to 'there' (the fix) is manageable enough that CORE has a chance of finding the path connecting them. Gross misunderstandings of concurrency (such as completely forgetting to add it) is not fixable by CORE.

CORE is designed to fix deadlocks and data races. It does so by manipulating *synchronize(...)* statements. These manipulations include adding, removing, growing and shrinking synchronization blocks, changing the locking variable within one and swapping variables within two nested synchronize blocks. Only bugs fixable by these operations can be improved upon. Other kinds of bugs, like notify vs notify all, cannot be fixed because the operators needed to insert, remove and manipulate notify events don't exist.

Like other search-based approaches, CORE cannot fix unknown bugs. A test case must exist to demonstrate the data race or deadlock in question. Bugs must also be consistently demonstratable. A class like StringBuffer isn't fixable when the data race shows up once in roughly 6,000 ConTest runs[1].

5.3 Software Engineering Optimizations

Numerous optimizations were added to the CORE framework during the development of ARC-OPT. Most are based on experience gained using the original ARC.

A very simple optimization was developed from the observation that all members of the GA¬C population have the same mutants in the first generation. Each member mutates the base program. Effort is saved by mutating the program once for the first member of the population and then copying the mutants to the other N-1 members. CORE uses a base number of ConTest runs (usually 15) to determine if a program

[1]CORE breaks for some unknown reason when the number of back to back ConTest runs goes over roughly 1,000.

still has data races or deadlocks. If no bugs are found, the base number of runs is multiplied by a security factor (usually 10). ConTest is run this many more times ($15 \times 10 = 150$) to see if the proposed fix is a good one. ARC used this large number of runs (150) to determine the timeout value for the program. In practice, this many runs was found to be unnecessary. In ARC-OPT (and all future versions), the number of runs to determine the timeout was lowered to a flat 20 runs.

ARC exhaustively generated all mutants for every member of the GA¬C population before it created and evaluated mutant projects. If the Xth member of the population fixes all bugs, the generation of mutants for members X+1, ... was unnecessary. ARC-OPT improved on this by generating all mutants for a member, then immediately creating and evaluating its mutated project. In ARC the bug-free verification step ran for the full number of ConTest runs (say, 150). Even if a bug was found on the first run the next 149 runs were still executed. An obvious optimization for ARC-OPT was to stop as soon as the first ConTest run failed.

It is possible that ARC will generate the same mutant project more than once over the course of a bug fixing run. Evaluating a repeat mutant is unnecessary if we record the identity of the mutant along with the results of the evaluation. When a particular mutant project is encountered again, the evaluation is copied into the new project. ARC-OPT adds this capability to CORE by recording a hash of the archived source and associating it with the testing results. ARC could only handle projects where the source code was in a single directory. This was an oversight resulting from testing only the IBM benchmark programs. ARC-OPT lifted this restriction by properly generating and using mutants for all source files within a source tree.

5.3.1 Static Analysis

Static analysis of the source code was the biggest feature added to the CORE framework during the development of ARC-OPT. Chord[2] was selected due to its familiarity and the ease of extracting the list of concurrently used *classes*, *methods* and *variables* it generates. In data race finding mode it determines the classes and variables used concurrently and the classes and methods. Chord is also capable of finding deadlocks. The files generated, though, can be hundreds of megabytes in size and are not even loadable. More accurate targeting is achieved by ARC-OPT when data race classes, methods and variables are combined with the *class* and *variable* information generated by ConTest. In the future, additional analysis tools (static, dynamic, ...) could be added to the framework.

The list of concurrent variables is used for two purposes: first, for synchronizing on statements containing these variables, and second, for use as synchronization variables. For example, if myInt and myClass are variables used concurrently, the ASAT mutants generated are,

synchronized(myClass) { ... myClass ... }

synchronized(myClass) { ... myInt ... }

synchronized(myInt) { ... myClass ... }

synchronized(myInt) { ... myInt ... }

Synchronization on primitive types (ints, floats, ...) isn't allowed in Java, so any mutants created from them won't compile. ARC-OPT adds functionality to remove primitive types from the list of variables used for synchronization (3^{rd} and 4^{th} entries in the list) but keeps primitive types in the list of variables to be targeted for synchronization (2^{nd} entry in list.). It is possible that both the static analysis and

[2]http://pag.gatech.edu/chord/

Table 5.1: Number of *classes*, *methods* and *variables* targeted by ARC-OPT after static analysis. The *variables* column is the number of non-primitive variables found.

Program	**Static Worked**	**Classes**	**Methods**	**Variables**	**Primitive Variables Eliminated**
Account	Yes	2	2	2	3
Accounts	Yes	2	1	2	4
Bubblesort2	No	1	3	1	3
Deadlock	No	1	10	4	0
Lottery	Yes	2	1	3	3
Pingpong	Yes	3	1	2	2
Airline	Yes	1	1	5	3
Buffer	Yes	2	2	2	9
Travelling sales	No	1	17	4	16
String buffer	No	1	20	1	3
Cache4j	No	3	46	122	11

ConTest may fail to find concurrent classes, methods or variables. ARC-OPT must be able to deal with all of these situations. When both fail the only variable ARC-OPT can synchronize on is 'this'. In practice ConTest never failed. If it doesn't find any concurrent classes or variables, something else has gone wrong. TXL operators that add synchronization had to be split up to account for these cases. For example, ASAT (add synchronization around a statement) was refactored into ASAT_CMV (classes, methods and variables used concurrently are known), ASAT_CV (classes and variables are known), ASAT_MV (methods and variables are known) and ASAT (no classes, methods or variables known, so the framework can only synchronize on 'this').

Compare Table 4.9 listing the number of classes, methods and variables in the test programs with Table 5.1 showing the number considered by ARC-OPT after the static analysis was run. Consider the Account program. It has 3 classes, 8 methods and 9 variables. After analysis by ConTest and Chord, 2 classes, 2 methods and 5

Table 5.2: Summary of the results of running the test programs through ARC-OPT 30 times.

Program	Avg. Time	Fix Gen.	Min. Time	Max. Time
Account	2m 55s	1	2m 2s	5m 20s
Account sub-type	7m 45s	1.4	2m 14s	23m 40s
Accounts	9m 1s	1	5m 43s	15m 22s
Airline	7m 43s	2	7m 6s	8m 24s
Bubblesort2	8m 58s	1	7m 6s	13m 52s
Deadlock	10m 54s	1.2	5m 5s	18m 53s
Linked list	3m 40s	1	2m 7s	5m 59s
Lottery	36m 47s	3.3	4m 39s	139m 16s
Pingpong	6m 32s	1	5m 13s	10m 11s
Reader-writer	2m 28s	1	2m 6s	5m 12s

variables are found to be used concurrently. After identifying 3 variables as primitive types, there are 5 variables targetable for synchronization (both primitive and non-primitive) and 2 to synchronize on (non-primitives only.).

5.4 Evaluating ARC

Initially ARC was tested on a subset of programs from the IBM benchmark, (Account, Accounts, Airline, Bubblesort2, Deadlock, Lottery and Pingpong). During the development of this thesis, additional programs were added to the test suite (Account sub-type, Linked list, Reader-writer, Buffer, Travelling sales, String buffer and Cache4j) to test ARC-OPT, CORE and CORE-IMC. ARC was able to fix all of the programs in the first set, except for Airline. A bug prevented it from selecting the proper mutant (Section 4.5). This bug was fixed for ARC-OPT and future versions.

Each program in Table 4.8 was run through ARC and ARC-OPT 30 times using the parameters described in Table 4.11. Results are summarized in Table 5.2. ARC-OPT was able to fix all 10 fixable programs and wasn't able to fix the 4 non-

Table 5.3: Comparison of ARC and ARC-OPTs performances.

	ARC		CORE	
Program	**Time**	**Fix Gen.**	**Time**	**Fix Gen.**
Account	8m 8s	5	2m 55s	1
Accounts	44m	1	9m 1s	1
Airline	12m 47s	2	7m 43s	2
Bubblesort2	100m 20s	2.2	8m 58s	1
Deadlock	2m 12s	1	10m 54s	1
Lottery	38m	2.4	36m 47s	3.6
Pingpong	12m 32s	1	6m 32s	1

fixable ones (Buffer, Travelling sales, String buffer and Cache4j). For the repairable programs, the time taken to find a fix ranged from about 2 minutes to 36 minutes on average. The average fix time dropped from 34 minutes with ARC to 13 with ARC-OPT, an improvement of 61%. The most time consuming aspect of ARC-OPT is the numerous ConTest executions. The second is the waiting necessary to determine the difference between a successful execution and a timeout. The *Timeout Multiplier* in Table 4.11 allows ARC and ARC-OPT to wait up to 4 times the running time of the program for it to complete.

All fixes were found in the first or second generation. The static analysis by Chord and the dynamic analysis by ConTest significantly constrained the state space. For example, the Account program contains 3 classes, approximately 9 methods and 6 variables. After the analysis, these are reduced to 2 classes, 3 methods and 3 variables. A population of 30 may exceed the number of mutations available causing the search space to be exhaustively covered. If the correct program is 1 or 2 mutation steps from the incorrect one, it should be found quickly.

Chapter 6

CORE-MC

6.1 Introduction

In this chapter we examine the development of CORE with model checking (CORE-MC) in Section 6.2. Software engineering optimizations are described in Section 6.3. CORE-MC is evaluated in Section 6.4. Variable selection is studied in Section 6.5.

6.2 CORE-MC

CORE-MC is the second contribution of this thesis and the third instantiation of the CORE framework (after ARC and ARC-OPT). It augments the uncertainty of noising with the certainty of model checking (Section 2.9). Model checking is used to determine if a proposed fix truly eliminates the deadlocks and data races by exhaustively searching the state space of the program. Exhaustive model-checking of mutants generated by CORE-MC provides certainty about results: a data race exists; a deadlock exists; there are no data races and no deadlocks. Java Pathfinder (JPF)[1] (Section 2.9.1) was selected as the model checker used by CORE-MC and

[1]Java Pathfinder, http://babelfish.arc.nasa.gov/trac/jpf

CORE-IMC.

Early in the development process we found that model checking by itself didn't work well. It took too long - hours sometimes, or crashed after running out of memory due to the state space explosion problem. After some thought, a hybrid approach was developed where both model checking and noising were used to test every proposed fix. First, the program was model checked. If unsuccessful CORE-MC fell back on noising with ConTest. Model checking can be very time consuming. As this thesis has the requirement that CORE operate in a reasonable amount of time, it was necessary to cap the model checking to a given search depth (say, 50 steps in the search space) and a given search time (say, 30 seconds). CORE-MC fell back on noising when the model checking failed and when the model checking reported the mutant was bug free.

Model checking extends the reach of CORE-MC. StringBuffer is an example of a program CORE couldn't fix because ConTest noising couldn't expose its data race with any regularity. Java Path Finder finds the data race within 1 second[2].

6.2.1 Adding JPF to CORE

CORE-MC was written in Python 2.7.X, JPF in Java. In CORE-MC, JPF is run by a regular Java class. For the sake of argument we call it *JRunJPF*. JRunJPF accepts configuration values for JPF, runs JPF and massages the results for later consumption. Py4j[3] is used as a bridge between CORE-MC/Python and JRunJPF/Java. It starts, manages and shuts down JRunJPF. Py4j automatically converts many data types between the two languages - allowing for easy configuration and querying of results.

[2]CORE-MC still isn't able to fix StringBuffer due to its complete lack of statement level locking.
[3]py4j, http://py4j.sourceforge.net/

Figure 6.1: Java allows locking on objects that haven't been created yet. JPF flags this as an error.

```
object myObject;
synchronized(myObject) {
   ...
   }
myObject = new Object();
...
```

Adding JPF to CORE-MC wasn't an easy task. There were a number of hurdles to jump and details to iron out. Most noticeable was the way JPF reported problems like deadlocks, data races and exceptions. Sometimes they are reported in the error text, otherwise they are found in the exception text. It was necessary to concatenate both texts together when searching for results.

Java allows one to use an object that hasn't been instantiated yet as a synchronization lock (Figure 6.1). JPF flags this as an error. In all future generations, JPF continues to flag this program as incorrect even if CORE-MC has fixed all of the deadlocks and data races in it. In effect any member of the population with this problem is 'knocked out' and doesn't contribute any more to the fixing process. During testing for example, 2 - 3 members of the population were 'knocked out' this way every generation for the Lottery test program. For a population of 30, the search stalls after 10 - 15 generations. CORE-MC exhausts all remaining generations futilely searching for a fix that is obscured by this synchronization-before-creation problem. To avoid this it was necessary to search for the specific error and when encountered, to reset the member to the previous generation.

One of the advantages of model checking is the wealth of information available. For deadlocks one can retrieve the list of classes involved. For data races, classes and variables are retrieved. These can be added to the existing list generated by

other forms of analysis (ConTest, Chord, ...) for better targeting of concurrently used classes, methods and variables to better constrain the search space.

6.3 Software Engineering Optimizations

In section 5.3 the addition of static analysis to ARC-OPT was described. One observes quickly that the list of concurrently used variables found by Chord never changes. A useful optimization added to CORE-MC is to save the results of the analysis to file so it doesn't have to be re-run every time CORE-MC is invoked. This optimization was back-ported to ARC-OPT to make comparisons between them more consistent.

In practice the CORE framework generates a large number of files. The ASAT (Add Synchronization Around sTatements) mutator is the largest contributor. It generates all mutants synchronizing individual lines of code. In a typical run the number of mutants generated is

$$\begin{aligned}&\#\text{ non-primitive concurrent variables} \times \#\text{ lines where concurrent variables are used}\\ &\qquad\qquad \times \#\text{ generations} \times \text{population size}\end{aligned}$$

If ASAT generates on average 200 mutants for each of the 30 members of the population over 30 generations, up to 18,000 mutants are generated. This consumes a lot of disk space - especially when one has to save nearly 800 runs for analysis! Generating this many files caused CORE to completely fill a hard disk on more than one occasion. An obvious optimization is to delete mutants that are no longer needed. CORE-MC deletes mutants that are 2 or more generations old. We must keep the previous generation of mutants for those occasions when a member needs to be rolled back to the previous generation. Only when CORE-MC completes its analysis are

Table 6.1: Summary of the results of running the test programs through CORE-MC 30 times.

Program	**Avg. Time**	**Fix Gen.**	**Min. Time**	**Max. Time**
Account	8m 8s	1.3	2m 35s	17m 3s
Account sub-type	5m 48s	1.6	2m 4s	12m 13s
Accounts	10m 29s	1.1	6m 10s	22m 11s
Airline	13m 25s	2	11m 50s	15m 24s
Bubblesort2	9m 39s	1	6m 57s	17m
Deadlock	6m 6s	1	2m 33s	12m 35s
Linked list	2m 56s	1.1	2m 7s	6m 17s
Lottery	9m 36s	1.5	2m 58s	34m 14s
Pingpong	5m 52s	1	5m 25s	6m 35s
Reader-writer	2m 37s	1	2m	4m 24s

the N^{th} and $N-1^{th}$ generations of mutants deleted. Note that only the mutants are deleted. Every other file is maintained.

Deleting mutants this way introduced new problems. Standard Python 2.7.X file deletion libraries were used. They would sometimes run for minutes at a time when deleting files. This introduced an unacceptable amount of uncertainty into the running time of CORE-MC. A specialized library, *SendToTrash*[4], was used to bypass Python's libraries and send the files directly to the trash/recycle bin. This didn't eliminate the overhead of file deletion, but deferred it to a time of the user's choice.

6.4 Evaluation

Results of testing CORE-MC on the test suite are in Table 6.1. CORE-MC fixes the 10 programs ARC-OPT can fix and cannot fix the 4 unfixable programs (Buffer, String buffer, Cache4j and Travelling sales). When compared to ARC-OPT, CORE-MC appears to do very well (Table 6.2). It is worse in 3 cases, about the same for 4 and better for 3 of the test programs. On average CORE-MC is 36% faster than

[4]SendToTrash 1.3, https://pypi.python.org/pypi/Send2Trash.

Table 6.2: Comparison of the average time required to find fixes for the test programs for ARC-OPT and CORE-MC.

	ARC-OPT		CORE-MC	
Program	**Time**	**Fix Gen.**	**Time**	**Fix Gen.**
Account	2m 55s	1	8m 8s	1
Account sub-type	7m 45s	1.4	5m 48s	1.6
Accounts	9m 1s	1	10m 29s	1.1
Airline	7m 43s	2	13m 25s	2
Bubblesort2	8m 58s	1	9m 39s	1
Deadlock	10m 54s	1.2	6m 6s	1
Linked list	3m 40s	1.1	2m 56s	1
Lottery	36m 47s	3.6	9m 36s	1.5
Pingpong	6m 32s	1	5m 52s	1
Reader-writer	2m 28s	1	2m 37s	1

Table 6.3: Variable study: JPF search time. Values studied were (90s, 60s, 30s, 20s, 10s). CORE-MC used a default search time of 30 seconds.

Program	Best Search Time	Time
Account	90	3m 45s
Accounts	30	6m 51s
Bubblesort2	30	8m 11s
Deadlock	10	6m 49s
Lottery	10	8m 45s
Pingpong	20	4m 8s

ARC-OPT.

6.5 CORE-MC Variable Study

All of the evaluations of ARC-OPT and CORE-MC were made using a common set of parameters. An open question is whether the parameters are good ones. Is one global set sufficient or does each test program have its own optimal set? Default values include a population of 30 for the GA¬C, 30 generations for the GA¬C, JPF

Table 6.4: Variable study: GA¬C population size. Values studied were (50, 30, 20, 10, 5). CORE-MC used a default population of 30.

Program	Best Population	Time
Account	50	8m 14s
Accounts	20	8m 1s
Bubblesort2	50	8m 40s
Deadlock	30	4m 4s
Lottery	5	9m 13s
Pingpong	30	4m 4s

Table 6.5: Variable study: JPF search depth. Values studied were (200, 150, 100, 50, 25). CORE-MC used a default search depth of 50.

Program	Best Search Depth	Time
Account	150	4m 30s
Accounts	25	8m 11s
Bubblesort2	25	8m 31s
Deadlock	200	4m 52s
Lottery	150	3m 45s
Pingpong	200	4m 12s

search depth of 50 and JPF timeout of 30 seconds.

This section describes a study undertaken to determine how good the default parameter choices for the GA¬C and JPF were by examining a range of values for each parameter[5]. Parameters examined were the GA¬C generations, GA¬C population size, JPF search time and JPF search depth. For example we looked at search times of 90s, 60s, 30s, 20s and 10s. Results on the JPF search time are described in Table 6.3, GA¬C population size in Table 6.4, JPF search depth in Table 6.5 and GA¬C generations in Table 6.6. It is interesting to observe that each parameter has a range of optimal values, except for GA¬C generations. Any value equal to or greater than 5 is 'optimal' because CORE-MC ends when a fix is found and always ended in

[5]When this test was performed we still thought the airline program was unfixable, so it was excluded.

Table 6.6: Variable study: GA¬C generations. Values studied were (30, 20, 10, 5, 3). CORE-MC used a default 30 generations.

Program	Best Generations	Time
Account	3	4m 13s
Accounts	3	6m 28s
Bubblesort2	5	8m 56s
Deadlock	3	4m 50s
Lottery	3	3m 36s
Pingpong	3	4m 4s

Table 6.7: Comparison of optimized variables vs non-optimized for CORE-MC.

	CORE-MC		**CORE-MC Optimized**	
Program	**Time**	**Fix Gen.**	**Time**	**Fix Gen.**
Account	7m 6s	1.1	8m 20s	1.1
Accounts	9m 31s	1	9m 26s	1.2
Bubblesort2	12m 25s	1	10m 27s	1
Deadlock	5m 31s	1	5m 3s	1
Lottery	34m 49s	3.3	4m 18s	3
Pingpong	7m 8s	1	5m 38s	1

under 5 generations.

Once all of the optimal values were collected, each program was run through CORE-MC using these values. Results are summarized in Table 6.7. Optimized values are worse in one case, the same for one case and better for the rest. Lottery is the only program dramatically affected by the optimized variables - dropping from 34 min to 4 min. On average, the optimized values are 13% better than the default selection. This indicates that the default selection was good but not optimal. Optimal is of course program dependent.

Chapter 7

CORE-IMC

7.1 Introduction

In this chapter we examine the development of CORE-IMC (Incremental Model Checking). We begin by discussing the software engineering optimizations added to CORE-IMC as described in Section 7.2. These are based on experience gained using CORE-MC. Further constraining the search space by searching function headers for in-scope lock variables is described in Section 7.3. Incremental model checking is introduced in Section 7.4. A test and a study were undertaken to give us confidence that incremental model checking works for CORE. They are described in Section 7.5. A full evaluation of incremental model checking is undertaken in section 7.6.

7.2 Software Engineering Optimizations

As described earlier (Section 4.4), one of the mutations CORE makes is the synchronization of the run() method. This mutation may fix a program but has the undesirable side-effect of serializing it. An option was added to the project configura-

Figure 7.1: In CORE-INC, mutations were created that contained nested synchronization on the same variable.

```
void transfer(Account ac, double mn){
  synchronized (ac) {
    synchronized (ac) {
      synchronized (ac) {
        amount-=mn;
        ac.amount+=mn;
      }
    }
  }
}
```

tion file to disallow any mutations from synchronizing the *run*() method. Even when synchronizing run is allowed, a check is added to see if that was the fix actually found. If it was, an alert is written to the log file explaining that the evolved fix synchronizes run() and describes the configuration option to disallow it in future runs.

Examination of the mutants generated revealed that CORE was creating mutants that contained nested synchronization on the same variable (Figure 7.1). A check was added to prevent this. Note that nested synchronization on different variables is still allowed. Finally, some programs cannot be model checked successfully in the time allotted. CORE was modified to disable the model checker if it timed out more than 10 times in a run. From that point onward the program under test would only be noised.

7.3 Scanning Function Headers

CORE was unable to fix three programs when synchronizing run() was disallowed. Further investigation determined that CORE didn't have the ability to find variables

passed as arguments to functions. CORE extracts the classes, methods and variables used concurrently that are found by ConTest, Chord and JPF. The list of variables is incomplete in the sense that a method may be found to be used concurrently, but no in-scope variables for that method were found on which it could synchronize. Either no mutants be generated for this method or any mutants created for it will fail to compile as the synchronization variables were out of scope. A code scanning step was added that looked for variables in the function declarations that were found to be used concurrently and added them to the internal list of variables used concurrently. As usual, primitive types were filtered out.

Note that the framework doesn't associate variables used concurrently to the methods in which they were found. In some cases this isn't possible - such as when ConTest returns the list of classes and variables it found were used concurrently. There is no associated method. In other cases a variable associated with one method could be used by another method. This increases the chance of finding a fix by a small amount. When CORE attempts to compile a candidate program, any mutants synchronizing on out of scope variables will be rejected.

7.4 CORE-IMC

As we learned in Chapter 6, model checking is time and resource intensive. CORE-MC is purely mutation driven. It only manipulates *synchronize*(...) statements. Changes from one generation to the next are usually small. Model checking would benefit greatly if it could re-use the results of the previous model-checking run from generation N-1 during the current run at generation N. *Incremental model-checking* techniques [48, 81] (Section 2.9.3) add this capability. After fully model checking a proposed fix in generation 1 of the GA¬C, only incremental model checking should be

Table 7.1: Number of full model checking (MC) and incremental model checking runs per strategy for population N and generations G.

Strategy	**Full MC Runs**	**Incremental MC Runs**
Model check each member of population in generation 1, incremental for the rest	N	(N - 1) × G
MC original program once, incremental for the rest	1	N × G

required to check the changes introduced by each mutation in each future generation. We believe this augmented approach will be faster than using model checking alone.

Using this strategy the number of full model checking runs is equal to the size of the population N. The number of incremental checks is equal to the number of generations G, minus the first generation, times the population size: $(G - 1) \times N$. CORE-IMC currently uses this approach. It may be possible to do better by model checking the buggy program once, generating the first generation for the evolutionary strategy, each containing a random mutation, then incrementally model checking each mutated program against the un-mutated. One full model checking run is performed and $G \times N$ incremental runs are performed. Table 7.1 summarizes this.

7.5 Does Incremental Model Checking Actually Generate Any Saving?

Incremental model-checking adds the overhead of loading, managing and saving the list of states explored by the model checker to the model checking process. It works well when the changes from one run to the next are small. The question becomes then, are changes to synchronized statements small? A test program and a study were

Table 7.2: Hand-seeded mutations for the Accounts program, for the proof of concept incremental run.

Generation	Mutant
0	None (base program)
1	ASAT on a for-loop in Bank.java
2	ASM in Account.java
3	ASAT on a different for-loop in Bank.java

Table 7.3: Results of the proof of concept incremental run on the Accounts program. Savings is calculated from the 2^{nd} and 4^{th}columns.

Gen.	Build (Non-Incr.)		Reuse (Incr.)		Savings?
	Time (s)	States ($\times 10^3$)	Time (s)	States ($\times 10^3$)	
0	9	27.8	9	27.8	No
1	9	28.5	9	28.5	No
2	25	119	17	92	Yes (32%)
3	22	110	16	83.3	Yes (27%)

undertaken to determine if incremental model checking would realize any benefits in practice.

7.5.1 Accounts Test

A proof of concept incremental model-checking test was created by mutating the Accounts program. The base Accounts program was used for generation 0. Three mutations were hand-added to the program for generations 1, 2 and 3[1]. Mutations are cumulative. Table 7.2 describes the mutations added. Table 7.3 shows how regular model-checking in the second and third columns compares to incremental model-checking in the fourth and fifth columns.

During the 0^{th} and 1^{st} generations there were no savings from incremental model

[1]All mutations were added before the model checking was invoked. No attempt was made to optimize this proof of concept run.

checking. Overhead from managing the incremental list is small, as the non-incremental and incremental times are approximately the same. This changes in the third and fourth generations. We see improvements of 32% and 27% respectively in the time required and states explored for incremental model-checking. This gives us some confidence that incremental model-checking works by reducing the amount of time required to model check this program.

7.5.2 Population 2 Study

The results of the previous section are encouraging but they don't tell us anything about how CORE-IMC will perform. For an initial test with CORE we created an easy test that the incremental aspect should be able to pass. If CORE fails this test, then we have strong reason to doubt that incremental model checking will be beneficial to CORE.

The test is biased and easy because we reduced the population of the GA¬C to 2. This low population is unusual for evolutionary strategies and will hamper their ability to find a solution quickly. To find a solution the GA¬C will run for more generations and give the incremental aspect of CORE-IMC a chance to take hold and demonstrate whether or not it offers any savings. For this study, ARC-OPT, CORE-MC and CORE-IMC were run with the standard settings and a population of two. Results are recorded in Tables 7.4, 7.5 and 7.6.

From the tables we see that CORE-MC performed better than ARC-OPT on all programs. For the population of 2, ARC-OPT averaged 16 minutes to find a fix while CORE-MC averaged 10 min, 35 sec. CORE-IMC has mixed results compared to CORE-MC. CORE-IMC is faster in 3 cases, slower in 2 and about the same for the sixth. Overall though, CORE-IMC comes out slightly ahead of CORE-MC at an average run time of 10 min, 2 sec compared to CORE-MC at 10 min, 35 sec.

Table 7.4: Results from ARC-OPT for the population 2 study.

Program	Successes	Time	Min Time	Max Time	Fix Gen.
Account	19/20	9m 45s	5m 3s	32m 29s	7.6
Accounts	18/20	35m 4s	8m 4s	186m 26s	5.9
Bubblesort2	20/20	12m 17s	8m 28s	23m 46s	3.5
Deadlock	18/20	16m 21s	4m 30s	93m 12s	4.4
Lottery	5/20	14m 47s	4m 53s	26m 50s	14.2
Pingpong	20/20	7m 38s	5m 17s	20m 13s	2.8

Table 7.5: Results from CORE-MC for the population 2 study.

Program	Successes	Time	Min Time	Max Time	Fix Gen.
Account	13/20	8m 42s	2m 15s	18m 13s	11.2
Accounts	18/20	13m 19s	6m 7s	81m 5s	4.2
Bubblesort2	20/20	11m 31s	6m 59s	32m 9s	4.1
Deadlock	20/20	13m 14s	2m 6s	140m 12s	4.5
Lottery	14/20	10m 49s	2m 58s	42m 23s	6.6
Pingpong	12/20	5m 56s	5m 21s	9m 10s	2.6

CORE-MC is on average 33% faster than ARC-OPT and CORE-IMC is on average 5% faster than CORE-MC[2]. For this scenario, model checking is much better than noising. Incremental model-checking also sees gains, but they are smaller than with the decision to use a model checker. It is also interesting to observe that not all runs found fixes for the deadlocks and data races. When the population was 30, the CORE framework always found a fix if one was available. A population of 2 often flounders. The small population cannot adequately cover the test programs search spaces.

From these two studies we have some confidence that incremental model checking will offer benefits over model checking alone.

Table 7.6: Results from CORE-IMC for the population 2 study.

Program	Successes	Time	Min Time	Max Time	Fix Gen.
Account	12/20	6m 39s	2m 23s	12m 38s	8.3
Accounts	15/20	12m 11s	6m 23s	43m 52s	4.3
Bubblesort2	20/20	13m 17s	6m 55s	34m 59s	5.3
Deadlock	16/20	10m 42s	2m 6s	95m 51s	4.3
Lottery	10/20	11m 36s	3m 10s	29m 26s	9.7
Pingpong	16/20	5m 46s	5m 22s	10m 5s	1.5

Table 7.7: Summary of the results of running the programs through CORE-IMC 30 times.

Program	Avg. Time	Fix Gen.	Min. Time	Max. Time
Account	3m 29s	1	2m 39s	4m 48s
Account sub-type	5m 31s	1.7	3m 56s	7m 46s
Accounts	10m 32s	1.2	6m 23s	18m 58s
Airline	11m 22s	2	9m 42s	12m 34s
Bubblesort2	10m 27s	1	7m 23s	16m 45s
Deadlock	6m 8s	1	2m 34s	11m 53s
Linked list	5m 4s	1.7	2m 5s	22m 36s
Lottery	9m 30s	1.5	4m 22s	22m 21s
Pingpong	6m 12s	1	5m 44s	7m 19s
Reader-writer	2m 41s	1	2m 1s	3m 44s

7.6 Evaluation

Results of testing CORE-IMC on the test suite are in Table 7.7. CORE-IMC can fix the 10 programs ARC-OPT and CORE-MC can fix and cannot fix the 4 unfixable programs (Buffer, Stringbuffer, Cache4j and Travelling sales.) When compared to CORE-MC, CORE-IMC is faster than CORE-MC in 3 cases, about the same in 4 cases and is slower than CORE-MC for the other 3. (Table 7.8). Overall CORE-IMC runs on average 10% faster than CORE-MC. This smaller improvement is expected as incremental model checking only applies to JPF and not to the thread noising.

[2]Only successful runs are considered in the population 2 study.

Table 7.8: Comparison of CORE-IMC and CORE-MC.

	CORE-MC		CORE-IMC	
Program	**Time**	**Fix Gen.**	**Time**	**Fix Gen.**
Account	8m 8s	1.3	3m 29s	1
Account sub-type	5m 48s	1.6	5m 31s	1.7
Accounts	10m 29s	1.1	10m 32s	1.2
Airline	13m 25s	2	11m 22s	2
Bubblesort2	9m 39s	1	10m 27s	1
Deadlock	6m 6s	1	6m 8s	1
Linked list	3m 1s	1.1	5m 4s	1.7
Lottery	9m 36s	1.5	9m 30s	1.5
Pingpong	5m 52s	1	6m 12s	1
Reader-writer	2m 37s	1	2m 41s	1

Chapter 8

Conclusions and Future Work

8.1 Introduction

We summarize the work and its contributions in Section 8.2. Next, future work is explored in Section 8.3. Finally in Section 8.4 we end with the ways in which the framework may be generalized for use in other areas.

8.2 Conclusions

Fixing bugs in programs is an expensive and time consuming process. Significant progress has been made on fixing bugs in single threaded programs. GenProg is an example of the state of the art. It assumes the fix is in the program already and that moving it to the execution path will fix the problem. Both of these decisions strongly constrain the search space of solutions and allow GenProg to fix bugs in real programs.

A great deal of work has been done on finding data races and deadlocks and suppressing them. Automatic repair of parallel programs is an under studied area.

This thesis contributes to it by developing the CORE framework. It fixes data races and deadlocks in concurrent Java programs. Contributions from this thesis come in two forms, constraining the search space and employing an efficient algorithm for navigating said search space.

In the CORE framework the search space is constrained by using different tools' abilities to find the classes, methods and variables used concurrently. They are;

- Using ConTest to find classes and variables in ARC
- Using Chord to find the classes, methods and variables in ARC-OPT
- Using JPF to find the classes and methods in CORE-MC
- Scanning function headers to find in-scope lockable variables in CORE-IMC

The final lists of classes, methods and variables are used to guide the search though the state space of potential mutations. These mutations add, remove, grow, shrink and change variables in synchronize(...) statements.

Software engineering improvements to the framework from one program to the next are too numerous to mention. They can be found in sections 5.3, 6.3 and 7.2. At each step the framework was evaluated and compared to the prevous step.

In the first phase ARC was heavily optimized to produce ARC-OPT (Chapter 5). Evaluation of the results was very positive, as Table 8.1 indicates. The average fix time dropped from 34 minutes to 13, an improvement of 61%.

The second phase integrated a model checker into the framework (CORE-MC, Chapter 6). Noising has its limitations. It is simply unable to expose some rarely occurring data races and deadlocks. We believed that model checking would be competitive with or faster than noising. In practice we found that noising and model checking had to be used together. Model checking doesn't work on all programs and

Table 8.1: Comparison of ARC's and ARC-OPT's performances on the test suite.

	ARC		ARC-OPT	
Program	**Time**	**Fix Gen.**	**Time**	**Fix Gen.**
Account	8m 8s	5	2m 55s	1
Accounts	44m	1	9m 1s	1
Airline	-	-	7m 43s	2
Bubblesort2	100m 20s	2.2	8m 58s	1
Deadlock	2m 12s	1	10m 54s	1
Lottery	38m	2.4	36m 47s	3.6
Pingpong	12m 32s	1	6m 32s	1

Table 8.2: Comparison of ARC-OPT and CORE-MC on the test suite.

	ARC-OPT		CORE-MC	
Program	**Time**	**Fix Gen.**	**Time**	**Fix Gen.**
Account	2m 55s	1	8m 8s	1
Account sub-type	7m 45s	1.4	5m 48s	1.6
Accounts	9m 1s	1	10m 29s	1.1
Airline	7m 43s	2	13m 25s	2
Bubblesort2	8m 58s	1	9m 39s	1
Deadlock	10m 54s	1.2	6m 6s	1
Linked list	3m 40s	1	2m 56s	1
Lottery	36m 47s	3.6	9m 36s	1.5
Pingpong	6m 32s	1	5m 52s	1
Reader-writer	2m 28s	1	2m 37s	1

because we want CORE-MC to return results in a reasonable time, we had to limit the search time and search depth. When model checking didn't work or ran out of time, CORE-MC fell back on noising. Comparing this hybrid approach to the noising only approach (Table 8.2), gives mixed results. The dramatic improvement to Lottery resulted in CORE-MC being, on average, 36% faster than ARC-OPT.

During this phase, a study of the default configuration variables used for the heuristic search (number of generations, model checking time, search depth, ...) was undertaken to determine how close to optimal the chosen values were. We learned

Table 8.3: Comparison of CORE-IMC and CORE-MC on the test suite.

	CORE-MC		CORE-IMC	
Program	**Time**	**Fix Gen.**	**Time**	**Fix Gen.**
Account	8m 8s	1.3	3m 29s	1
Account sub-type	5m 48s	1.6		
Accounts	10m 29s	1.1	10m 32s	1.2
Airline	13m 25s	2	11m 22s	2
Bubblesort2	9m 39s	1	10m 27s	1
Deadlock	6m 6s	1	6m 8s	1
Linked list	3m 1s	1.1	5m 4s	1.7
Lottery	9m 36s	1.5		
Pingpong	5m 52s	1	6m 12s	1
Reader-writer	2m 37s	1	2m 41s	1

they were within 13% of optimum and that as expected, optimal values were program dependent.

In the third phase, incremental modelling is added to CORE-MC to produce CORE-IMC (Incremental model checking). Incremental model checking attempts to speed up model checking runs by using the results of previous model checking runs. Any program state evaluated in a previous run and recorded to file doesn't need to be computed again - saving efford. On average CORE-IMC was 10% faster than CORE-MC. Table 8.3 evaluates CORE-IMC and compares it to CORE-MC.

8.3 Future Work

CORE is full of possibilities for future work and research. The framework is highly adaptable and can grow in many directions. Some of these are explored in the next two subsections.

8.3.1 Software Engineering

1. No attempt was made to optimize the configuration values of the tools used, ConTest, Chord and JPF. Different choices should affect the time required for the framework to find solutions. (Undergraduate)

2. Record the testing results of each mutation to a database, so that when a mutation is seen again the results can be retrieved quickly. (Undergraduate)

3. Add crossover and selection to the GA¬C. (Masters)

4. Update the Python code from version 2.7.X to the most recent release[1]. (Undergraduate)

5. Integrate the framework into an existing software development tool chain. (Masters)

8.3.2 Theoretical

1. Augmenting the existing tools (ConTest, Chord and JPF) with new ones could better constrain the search space or provide more information to the heuristic search. (Masters)

2. Adding a dynamic analysis tool to CORE would give it the ability to target the portions of code affected by the bug. (Masters)

3. Augment the mutation operators with additional operators to help repair data races and deadlocks. (Undergraduate) Study their effectiveness. (PhD)

4. CORE's operator weighting scheme needs to be improved upon and evaluated. (Undergraduate)

[1]Python 3.3.5 is the latest version available as of June, 2014.

5. CORE's optimization phase needs to be improved upon and evaluated. (Masters)

6. Create metrics to measure the complexity of concurrent bugs. Lines of code don't work. (PhD)

7. Since random searching is a strong competitor to heuristic searching in automatic bug repair,

   (a) compare random mutant generation and selection against weighted generation and selection. (Masters)

   (b) compare random operator type selection against weighted selection. (Undergraduate)

   (c) study and describe why random search works so well for automatic bug repair. (Masters/PhD)

8. Some programs fail to model check because they use code (like Integer.parseInt) that JPF doesn't like. Develop a system whereby JPF can be modified to work on as wide a range of Java code as possible[2]. (Masters)

8.4 Generalizations

CORE was customized to repair deadlocks and data races in concurrent Java programs. There are numerous ways the framework could be generalized and used in other areas.

1. Modify CORE to work with other programming languages.

[2]For example, by using existing modules, http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/start and http://babelfish.arc.nasa.gov/trac/jpf/wiki/summer-projects/start.

   (a) New analysis tools (Noisers, model checkers, ...) may be needed for this. (Undergraduate)

   (b) New TXL operators would need to be written. (Masters)

2. Add mutation operators for different kinds of software bugs. (Masters)

3. Modify CORE to fix bugs in sequential programs. (PhD)

4. Augment CORE with additional heuristic search techniques like ant, simulated annealing or particle swarm. (Masters)

At its heart CORE is a mutation engine. Mutations perturb existing values in an attempt to find better solutions. CORE can be adapted to fix problems in other parts of the search-based software engineering field. For many of these tasks, the code in the target program must be altered or moved. TXL is very good at doing so in a programming language agnostic way. It has already been integrated into CORE. Generalizations include;

1. Program refactoring. (Masters)

2. Test suite optimization, selection or refactoring using the ConMan operators and other tools. (Masters)

3. Security testing. (PhD)

4. Input mutation for testing (eg, security testing) (Masters/PhD)

5. Metric optimization through source refactoring. (Masters)

6. Performance optimization. (Masters)

Appendices

Appendix A

Source Listings for Test Programs

A.1 Source Listings

In the following sections the source code is shown for the data race or deadlock along with the fix found by the CORE framework for each of the test programs.

A.1.1 Account

In Account there is a data race between the $transfer$, *depsite* (sic) and *withdraw* methods in the *run* method. When run the i^{th} account thread invokes the $transfer$ method of the $(i+1)^{th}$ account (Figure A.1). On the next iteration of the loop the $(i+1)^{th}$ account thread invokes *depsite* and *withdraw*. As the *ac* class in $transfer$ isn't synchronized, the transfer can be overwritten by the *depsite* or *withdrawal* method calls. In plainer language, \$99 is deducted from sender i, but isn't received by the receiver $i+1$ as it is overwritten by the deposit of \$300 or the withdrawal of \$100.

CORE can fix Account two different ways. If synchronizing $run()$ is allowed, this fixes the data race - but at the cost of serializing the program. Otherwise, CORE

Figure A.1: In the Account program, the *Transfer* method can lose updates because of a lack of synchronization on the *ac* variable.

```
synchronized void transfer(Account ac, double mn){
  amount-=mn;
  ac.amount+=mn;
}
```

Figure A.2: In the Account program, synchronizing ac in the *Transfer* method fixes the data race.

```
synchronized void transfer(Account ac, double mn){
  amount-=mn;
  synchronized(ac) {
    ac.amount+=mn;
  }
}
```

synchronizes *ac* in the $transfer$ method (Figure A.2).

A.1.2 Account Sub-Type

Account Sub-Type [23] is a modification of the Account program. In it an abstract class represents a basic account type. Two child types were created: *business* and *personal* accounts. In the $Business.transfer$ method the transfer was properly synchronized, while in the $Personal.transfer$ method it was not (Figure A.3). As with Account above, the lack of synchronization on $Personal.transfer$ caused a data race between the $transfer$, *deposite* and *withdraw* methods. CORE fixed it the same way it fixed Account - by synchronizing within the transfer method.

This wasn't the end of the story though. After fixing the data race from $Personal.transfer$, the JPF model checker found a deadlock where ConTest didn't. JPF reported that the $account.transfer$ method calls for personal and business ac-

Figure A.3: In Account Sub-Type, the *PersonalAccount.Transfer* method lost updates because of a lack of synchronization on the *ac* variable. Once that was fixed, the program deadlocked on the *transfer* method calls in *run*.

```
public synchronized void transfer(Account ac, int mn){
    amount-=mn;
    ac.amount+=mn;
}

public void run(){
    ...
    account.transfer(bank.getAccount(nextNumber),10);
    account.deposit(10);
    account.withdraw(20);
    account.deposit(10);
    account.transfer(bank.getAccount(nextNumber),10);
    account.withdraw(100);
  }
```

counts in *Manager.run* deadlocked. Account N calls *transfer*. First it locks itself then it attempts to lock account N+1. At the same time account N+1 calls *transfer* to lock itself and then lock object N+2 and so on. JPF detects a deadlock because each account can be in *transfer* leading to a circular deadlock.

The only way to fix this bug is to serialize access to *transfer*. Both calls to *transfer* in *run* must be synchronized on a global lock, and depending on the mutations, the same global lock. This is a hard fix for CORE to find. Synchronizing run (if allowed in the configuration file) also fixes the bug - at the cost of parallelism.

A.1.3 Accounts

Accounts has a very straightforward data race in the *service* method (Figure A.5). In *run* the account repeatedly adds a random amount to a balance and the global running sum. Everything works correctly if the sum of balances is equal to the

Figure A.4: In Account Sub-Type, CORE fixed the data race in *PersonalAccount.Transfer* but had trouble fixing the deadlock in *run*().

```
public synchronized void transfer(Account ac, int mn){
    amount-=mn;
    synchronized (ac) {
        ac.amount+=mn;
    }

public void run(){
    ...
    synchronized (bank) {
        account.transfer(bank.getAccount(nextNumber),10);
    }
    account.deposit(10);
    account.withdraw(20);
    account.deposit(10);
    synchronized (bank) {
        account.transfer(bank.getAccount(nextNumber),10);
    }
    account.withdraw(100);
  }
}
```

Bank_Total global sum. The work is done in the static *Service* method. Multiple threads race on *Bank_Total* in *Service* causing the global sum to disagree with the sum of individual account balances.

The optimal fix is to synchronize the *Bank_Total* line in the *Service* method (Figure A.6). A less optimal fix is to synchronize both lines or the method itself.

A.1.4 Airline

Airline is a threaded ticket sales simulator for airlines. It uses a class level variable to keep track of the number of seats sold on an aeroplane. Sales stop when the number of tickets sold is equal to the aircraft's capacity. This programs has a race on the

Figure A.5: In the Accounts program, threads race on the *Service* method.

```
public void run(){
  int loop = 100;
  for(int i = 0; i< loop; i++){
    this.Action();
    ...
  }
}

public void Action(){
  int sum = random.nextInt() % MAX_SUM;
  Bank.Service(Account_Id,sum);
}

public static void Service(int id,int sum){
  accounts[id].Balance += sum;
  Bank_Total += sum;
}
```

Num_Of_Seats_Sold and *StopSales* variables between the main body of code and the *run* methods of the ticket sellers. One or more seats can be sold past the capacity of the aeroplane on different threads after *StopSales* is set to *true*.

Initially Airline was classified as unfixable by ARC. After reviewing it again for this thesis it was clear that it should be fixable by ARC-OPT. Sadly, ARC-OPT couldn't fix airline either, but both CORE-MC and CORE-IMC could. Upon analysis, the cause of ARC-OPT's failure to fix airline was traced to the fact that all of the concurrent variables discovered by the analyses were of primitive types. Java doesn't allow synchronization on primitive types, so the mutants generated by both ARCs didn't compile. Both ARC and ARC-OPT generated 173 ASAT (Add Synchronization Around sTatement) mutants using primitive types as the locking variables, 15 ASM (Add Synchronization around a Method) again using primitives for locking

Figure A.6: CORE's fix for the Accounts program.

```
public static void Service(int id,int sum){
  accounts[id].Balance += sum;
  synchronized(accounts) {
    Bank_Total += sum;
  }
}
```

variables and 3 ASIM (Add Synchronization In Method header) mutants. Of the 191 mutants available, only 2 of the ASIM mutants created compilable programs[1]. ARC and ARC-OPT both failed to fix Airline for two linked reasons. First, 189 of the 191 mutants didn't compile. Second, there was a bug in both ARC and ARC-OPT that caused them not to consider the ASIM mutants. If ARC or ARC-OPT never chose an ASIM mutant, they would never create a compilable program and never find a fix. This is important because one of the ASIM mutants did fix the bug.

During the development of the CORE-MC and CORE-IMC, this ASIM selection bug was fixed. Now both CORE and CORE-IMC can fix Airline. One of the features added to the framework was the elimination of primitive types from the list of synchronizable variables. As discussed, all discovered variables were primitive. No ASAT or ASM mutants were generated - only ASIM. With no guidance, ARC fell back on method synchronization and found the only fix it could, synchronizing run().

A.1.5 Bubblesort2

Bubblesort2 parallellizes the bubblesort algorithm (Figure A.9). It creates a data race on the globally declared array variable *array*. Every thread uses the *swpArray* function to modify the array. This call only has synchronization within the object but

[1]One ASIM mutant attempted to synchronize the constructor, so only 2 of them compiled.

Figure A.7: In the Airline program, there is a race on the *StopSales* variable between the main body of code and the *run* method.

```
...
for (int i = 0; i < Num_of_tickets_issued; i ++) {
  threadArr [i] = new Thread (this);
  if (StopSales) {
    Num_Of_Seats_Sold --;
    break;
  }
  threadArr [i].start ();
}
...

public void run () {
  Num_Of_Seats_Sold ++;
  if (Num_Of_Seats_Sold > Maximum_Capacity)
    StopSales = true;
}
```

not between objects. Multiple objects of the class can be in *swpArray* simultaneously making changes to the global array. This causes data races.

CORE fixes the data race by locking array accesses on the *array* variable (Figure A.10). Ideally CORE would synchronize the lines in *swpArray*. This isn't possible because there are no non-primitive variables in scope on which to lock. CORE finds the only fix it can, synchronizing *run*. Once again the program is unavoidably serialized.

A.1.6 Deadlock

Our working physicists' deadlock is implemented in the Deadlock program in terms of file copying from a to b and b to a simultaneously (Figure A.11). The *write* method attempts to lock the source file, then the destination file in turn. If two threads each

Figure A.8: CORE fix for the Airline program.

```
public synchronized void run () {
  Num_Of_Seats_Sold ++;
  if (Num_Of_Seats_Sold > Maximum_Capacity)
    StopSales = true;
}
```

lock their source file, the program deadlocks.

CORE's fix is to synchronize access to the *write* method (Figure A.12). Other fixes found by CORE include synchronizing the whole run method and synchronizing the *write* method in *run*.

A.1.7 Lottery

In Lottery three methods race on the *randomNumber* variable (Figure A.13). The different analysis tools find synchronizable variables, but not all of the synchronizable methods. They find only that the *generate* method is used concurrently. Without the *present* and *record* methods, all of the lines racing on *randomNumber* cannot be synchronized to fix the data race. None of the three methods receive arguments, so searching for them is of no help.

With incomplete information, CORE finds the only fix it can - synchronizing *run* (Figure A.14). This problem is in part a limitation of the tools used. Different analysis tools might find the other two methods. Even then CORE would still have some difficulty because the lines in the three methods must synchronize on the same lock. Currently CORE randomly selects a lock for each added synchronize statement.

Figure A.9: In Bubblesort2 the threads can race on the *array* variable in the *run* and *swpArray* methods.

```
public void run()
{
  int i;
  for(i=0;i<fin;i++)
  {
    if(array[i]>array[i+1]) {
      swpArray(i);
    }
    if(i==0) {
      NewThread ntt=new NewThread(fin-1);
      ntt.start();
    }
  }
}

private static synchronized void swpArray(int i){
  int temp;
  temp=array[i+1];
  array[i+1]=array[i];
  array[i]=temp;
}
```

A.1.8 Pingpong

In Pingpong there is a class level variable called *pingPongPlayer* accessed by all threads. These threads call the *ping* method that in turn calls *pingPong*. In i*pingPong* the *pingPongPlayer* variable is set to null for 50 milliseconds. A different thread trying to call *pingPongPlayer.getI*() during this 50 milliseconds generates a *NullPointerException*.

CORE fixed PingPong by synchronizing any method in the chain of calls - *run*, *ping* (Figure A.16) or *pingPong*.

Figure A.10: CORE fix for the Bubblesort2 program.

```
public void run () {
  int i;
  synchronized (array) {
    for (i = 0; i < fin; i ++) {
      if (array [i] > array [i + 1]) {
        swpArray(i);
      }
      if(i==0) {
        NewThread ntt=new NewThread(fin-1);
        ntt.start();
      }
    }
  }
}
```

A.1.9 Linked List

Linked list is a concurrent implementation of a linked list. It has a data race in the *insert* method. The last line of code, $p._c urrent._n ext$ = ... can be raced on, causing random linking within the list (Figure A.17). The fix is to extend the *synchronized*(*this*) block down one line to properly synchronize the method (Figure A.18). Other fixes found by CORE include synchronizing more or all of the lines in the method.

A.1.10 Readers-Writers

Readers-Writers is a concurrent implementation of readers and writers operating on a common pool. It has a data race between the readers and writers. Sometimes a reader can be active when a writer is writing. When this occurs, the $beforeRead$ method throws a $java.lang.IllegalMonitorStateException$ (Figure A.19). The fix is to synchronize the $beforeRead$ method (Figure A.20).

A.1.11 Buffer

Buffer has multiple readers consuming from and writers writing to a buffer. It contains a *notify vs notify all* bug. If a buffer is full (or empty) and a writer (reader) notifies another writer (reader), nothing happens and the program deadlocks. If $notifyall$ had been called instead, a reader thread (writer thread) would activate to consume some content from the full buffer (write some content to the empty buffer). CORE doesn't have any mutation operators to change notify statements to notifyall statements, so it cannot fix this program.

A.1.12 StringBuffer

StringBuffer contains a data race on the *count* variable between the *append* methods and the *delete* method (Figure A.22). It occurs very rarely – on the order of 1 in 6000 ConTest runs[2]. When it occurs StringBuffer crashes and throws a $StringIndexOutOfBoundsException$. As ARC and ARC-OPT use ConTest, they cannot demonstrate the bug with any regularity, causing CORE to erroneously report the original buggy program as 'fixed'.

CORE cannot fix the race either. It can improve the search depth at which the model checker finds the bug though. For example, the search depth was improved from 22 to 38 over the course of a run. This suggests the bug will occur less frequently. There are two reasons to be sceptical. First, the data race already occurs rarely. Improving this from 1 in 6000 to 1 in 10000 (say) only makes the bug harder to find. Second, these improvements were from synchronizing other methods like $ToString$. Even removing synchronization from methods can improve the search depth! For StringBuffer there isn't a clear correlation between higher search depth

[2]The author has seen this exception occur only once.

and the program being more correct.

CORE attempted to fix the $append(StringBufferSB)$ method. By itself this isn't enough. The underlying problem in StringBuffer is a complete lack of synchronization at the statement level. Only methods are synchronized. Two different methods can race on any shared variable. This complete lack of synchronization is a gross misunderstanding of concurrency that CORE cannot fix.

A.1.13 Cache4j

Different bug detection research papers have different things to say about Cache4j. In one [65], it has a benign atomicity violation. In another [73], a race over the *sleep* field in CacheCleaner.java was found, leading to an uncaught exception (Figure A.23). If *sleep* is set to *true* by a thread in the left part of Figure A.23, followed by a context switch (before this thread enters the *try* block) to a thread at the right part of Figure A.23, an uncaught $InterruptedException$ is thrown, causing the second thread to crash. To fix this bug CORE has to synchronize the code in the left part of Figure A.23.

A.1.14 Travelling Salesperson (TSP)

Once again different papers have different things to say about this implementation of the Travelling Salesperson algorithm. One detection method [66] states that the data races in TSP are benign. Another found both benign data races and malignant ones that "... involved updates that could be lost, leading to incorrect results" [85]. There was no specific information on where the bug is or how frequently it occurs when found.

TSP is similar to StringBuffer in that the data race appears infrequently. While

most programs are noised by ConTest 15 times, it is necessary to noise TSP over 1000 times for every member for every generation, in order for ConTest to expose the bug with any regularity. On top of the rarity of the data race, Cache4j times out on average once every 300 runs. Timeouts appear about 3× more commonly than the data race, when the timeout is set to 20× the average running time of the program. This results in every potential fix being declared incorrect because a timeout is considered incorrect. In practice, CORE runs until the large number of ConTest runs causes the framework to crash. CORE is unable to fix the Travelling Salesperson program.

Figure A.11: The Deadlock program simulates the working physicist problem by locking files. Each thread locks one file and then deadlocks while trying to lock the other file.

```
public void run()
{
  String n=Thread.currentThread().getName();
  int num= Integer.parseInt(n);
  if (num%2==0)
    write(a,b);
  else
    write(b,a);
}

public void write(Object from, Object to)
{
  ...
  synchronized(from) {
    ...
    if (hash.contains(to) ){
      System.out.println("deadlock on "+to);
      ...
    }
    else {
      synchronized(to) {
        // here the copying is being done.
        hash.remove(from);
      }
    }
  }
}
```

Figure A.12: CORE's fix for the Deadlock program.

```
public void run () {
  String n = Thread.currentThread ().getName ();
  int num = Integer.parseInt (n);
  synchronized (b) {
    if (num % 2 == 0) write (a, b);
      else write (b, a);
  }
}
```

Figure A.13: In Lottery the methods *generate*, *present* and *record* race on the class level variable *randomNumber*.

```
public synchronized void run () {
  int i = 0;
  while (i != numOfUsers) {
    generate ();
    for (i = 0; i < numOfUsers; i ++) {
      if (history [i] == randomNumber) break;
    }
  }
  present ();
  record ();
}

protected synchronized void generate () {
  generated [userNumber] = randomNumber = (long) (Math.random
      () *
    Math.pow (10, MAX_DIGITS));
}

protected synchronized void present () {
  System.out.print ("user " + userNumber + " assigned " +
    (presented [userNumber] = randomNumber) + ".");
}

protected synchronized void record () {
  history [userNumber] = randomNumber;
}
```

Figure A.14: CORE's fix for the Lottery program.

```
public synchronized void run () {
  int i = 0;
  synchronized (generated) {
    while (i != numOfUsers) {
      generate ();
      for (i = 0; i < numOfUsers; i ++) {
        if (history [i] == randomNumber) break;
      }
    }
    present ();
    record ();
  }
}
```

Figure A.15: All threads call the *pingPong* method containing the class level variable *pingPongPlayer*. Calling *get* while it is null generates a *NullPointer Exception*.

```
public void pingPong() {
  try {
    this.pingPongPlayer.getI();
    PingPong newPlayer;
    newPlayer = this.pingPongPlayer;
    this.pingPongPlayer = null;
    long time = System.currentTimeMillis();
    while ((System.currentTimeMillis() - time) < 50) ;
    this.pingPongPlayer = newPlayer;
  } catch ... { ... }
}

public void run () {
  this.ping ();
}

public void ping () {
  bg.pingPong ();
}
```

Figure A.16: CORE's fix for the Pingpong program.

```
public void ping () {
  synchronized (bg) {
    bg.pingPong ();
  }
}
```

Figure A.17: In the concurrent linked list implementation, a race occurs within the *insert* method.

```
public void insert (Object x, MyLinkedListItr p) {
   if (p != null && p._current != null) {
     MyListNode tmp;
     synchronized (this) {
        tmp = new MyListNode (x, p._current._next);
     }
     p._current._next = tmp;
   }
}
```

Figure A.18: CORE fixes the data race in the *insert* method by synchronizing it.

```
public void insert (Object x, MyLinkedListItr p) {
  if (p != null && p._current != null) {
    MyListNode tmp;
    synchronized (this) {
      tmp = new MyListNode (x, p._current._next);
      p._current._next = tmp;
    }
  }
}
```

Figure A.19: In the Readers-Writers program, a data race occurs where a reader is active when a writer is writing. This can cause a *java.lang.IllegalMonitorStateException* to be thrown from within the *beforeRead* method.

```
protected void beforeRead () {
  try {
     ++ waitingReaders;
      while (! allowReader ()) {
       try {
         wait ();
       } catch (InterruptedException ie) {
         -- waitingReaders;
       }
    }
    -- waitingReaders;
  } catch (Exception e) {
    RWVSNDriver.goodRun = false;
  }
}
```

Figure A.20: CORE fixed the exception and race by synchronizing the *beforeRead* method.

```
protected synchronized void beforeRead () {
  try {
    ++ waitingReaders;
    while (! allowReader ()) {
      try {
        wait ();
      } catch (InterruptedException ie) {
        -- waitingReaders;
      }
    }
    -- waitingReaders;
  } catch (Exception e) {
    RWVSNDriver.goodRun = false;
  }
}
```

Figure A.21: The *enq* method in Buffer has a *notifyvsnotifyall* bug.

```
public synchronized void enq(Object newObj) ... {
  ...
  if (_consoleOut) {
    printBuffer();
  }
  ...
  this.notify();
}
```

Figure A.22: In StringBuffer, the *append* and *delete* methods can interfere and cause a data race on the *count* variable. In general StringBuffer is missing statment level locking.

```
public synchronized StringBuffer append(StringBuffer sb) {
  ...
  sb.getChars(0, len, value, count);
  count = newcount;
  return this;
}

public synchronized StringBuffer delete (int start, int end)
  {
  ...
  if (len > 0) {
    if (shared)
      copy();
  System.arraycopy(...);
  count -= len;
  }
  return this;
}
```

Figure A.23: In cache4j, two threads can interfere and cause a crash on the *sleep* variable.

```
_sleep = true;
try {
  sleep(_cleanInterval);
} catch (Throwable t){
} finally {
  _sleep = false;
}

synchronized(this) {
  if(_sleep) {
    interrupt();
  }
}
```

Bibliography

[1] Abrial, J. *The B Book: Assigning Programs to Meaning*. Cambridge University Press, 1996.

[2] Acree, A. T., Budd, T. A., DeMillo, R. A., Lipton, R. J. and Sayward, F. G. Mutation analysis. Tech. rep., GIT-ICS-79/08, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, Sep. 1979.

[3] Anand, S., Păsăreanu, C., and Visser, W. JPF-SE: A symbolic execution extension to Java PathFinder. *Tools and Algorithms for the Construction and Analysis of Systems* (2007), 134–138.

[4] Arcuri, A. On the Automation of Fixing Software Bugs. In *Proc. of International Conference on Software Engineering (ICSE'08)* (2008), IEEE, pp. 1003–1006.

[5] Arcuri, A. Evolutionary repair of faulty software. *Applied Software Computing 11*, 4 (2011), 3494–3514.

[6] Arcuri, A., and Fraser, G. On parameter tuning in search based software engineering. *Search Based Software Engineering* (2011), 33—-47.

[7] Arcuri, A., and Yao, X. A novel co-evolutionary approach to automatic software bug fixing. In *Proc. of the IEEE Congress on Evolutionary Computation (CEC'08)* (2008), IEEE, pp. 162–168.

[8] Attie, P., and Saklawi, J. Model and program repair via sat solving. *Arxiv preprint arXiv:0710.3332 19*, 6 (2007), 29.

[9] Baker, P., Harman, M., Steinhofel, K., and Skaliotis, A. Search-based approaches to the component selection and prioritization problem. In *Proc. of the IEEE International Conference on Software Maintenance (ICSM'06)* (2006), IEEE Computer Society, pp. 176–185.

[10] Beasley, D., Bull, D., and Martin, R. An overview of genetic algorithms: Part 1 fundamentals. *University Computing 15*, 2 (1993), 58–69.

[11] Bradbury, J., Kelk, D., and Green, M. Effectively using search-based software engineering techniques within model checking and its applications. In *Proc. of the 1st International Workshop on Combining Modelling and Search-Based Software Engineering (CMSBSE 2013)* (May 2013), pp. 67–70.

[12] Buccafurri, F., Eiter, T., Gottlob, G., and Leone, N. Combining abduction and model checking techniques for repair of concurrent programs. *Periodica Polytechnica, Electrical Engineering 42*, 1 (1998), 91–101.

[13] Busetti, F. Genetic algorithms overview. *Teknik Teblig* (2002), 1–13.

[14] Chew, L., and Lie, D. Kivati: fast detection and prevention of atomicity violations. In *Proc. of the 5th European Conference on Computer Systems (EuroSys'10)* (2010), ACM Press, pp. 307–320.

[15] ClearSy. *Atelier-B Proof Obligation Reference Manual Ver. 3.7*. ClearSy, 2009.

[16] Conway, C., Namjoshi, K., Dams, D., and Edwards, S. Incremental algorithms for inter-procedural analysis of safety properties. In *Proc. of the 17th International Conference on Computer Aided Verification (CAV'05)* (2005), Springer, pp. 387–400.

[17] Cordy, J., Halpern, C., and Promislow, E. Txl: A rapid prototyping system for programming language dialects. In *Proc. of the International Conference on Computer Languages (ICCL'88)* (1988), IEEE, pp. 280–285.

[18] d'Amorim, M., Lauterburg, S., and Marinov, D. Delta execution for efficient state-space exploration of object-oriented programs. *Software Engineering, IEEE Transactions on 34*, 5 (2008), 597–613.

[19] DeAmorim, M., Sobeih, A., and Marinov, D. *Optimized Execution of Deterministic Blocks in Java PathFinder*, vol. 4260. Springer, 2006, pp. 549–567.

[20] Do, H., Elbaum, S., and Rothermel, G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering 10*, 4 (2005), 405–435.

[21] Dwyer, M., Elbaum, S., Person, S., and Purandare, R. Parallel randomized state-space search. In *Proc. of the 29th International Conference on Software Engineering (ICSE'07)* (2007), IEEE, pp. 3–12.

[22] Dwyer, M., Hatcliff, J., Robby, R., Pasareanu, C., and Visser, W. Formal software analysis emerging trends in software model checking. In *Proc. of the Conference on Future of Software Engineering (FOSE'07)* (2007), vol. 19, IEEE, pp. 120–136.

[23] Dwyer, M. B., Person, S., and Elbaum, S. Controlling factors in evaluating path-sensitive error detection techniques. In *Proce. of the 14th International Symposium on Foundations of Software Engineering (FSE'06)* (2006), ACM, pp. 92–104.

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

[25] EI-Fakih, K., Yevtushenko, N., and Bochmann, G. FSM-based incremental conformance testing methods. *Software Engineering, IEEE Transactions on 30*, 7 (July 2004), 425–436.

[26] Eytani, Y., Havelund, K., Stoller, S. D., and Ur, S. Toward a Framework and Benchmark for Testing Tools for Multi-Threaded Programs. *Concurrency and Computation: Practice & Experience 19*, 3 (2006), 267–279.

[27] Ferreira, M., and Gomez-Pulido, J. Detecting protocol errors using particle swarm optimization with Java PathFinder. In *Proc. of the High Performance Computing & Simulation Conference (HPCS'08)* (2008), ACM Press, pp. 319––325.

[28] Fonseca, C., and Fleming, P. An overview of evolutionary algorithms in multiobjective optimization. *Evolutionary Computation 3*, 1 (1995), 1–16.

[29] Forrest, S., Nguyen, T., Weimer, W., and Le Goues, C. A Genetic Programming Approach to Automated Software Repair. In *Proc. of Genetic And Evolutionary Computation Conference (GECCO'09)* (2009), pp. 947–954.

[30] Gligoric, M., Gvero, T., Lauterburg, S., Marinov, D., and Khurshid, S. Optimizing generation of object graphs in Java PathFinder. In *Proc.*

of the International Conference on Software Testing Verification and Validation (ICST'09) (2009), IEEE, pp. 51–60.

[31] Hari, K., and Srinidhi, V. Deterministic dynamic deadlock detection and recovery. *ACM Trans. Program. Lang. Syst.* (2012), 44.

[32] Harman, M. The current state and future of search based software engineering. In *Proc. of the Conference on the Future of Software Engineering (FOSE'07)* (2007), IEEE Computer Society, pp. 342–357.

[33] Harman, M. Search based software engineering for program comprehension. In *Proc. of the Conference on Program Comprehension (ICPC'07)* (2007), IEEE, pp. 3–13.

[34] Harman, M. Automated Patching Techniques: The Fix is in: Technical Perspective. *Communications of the ACM 53*, 5 (May 2010).

[35] Harman, M. Why the Virtual Nature of Software Makes it Ideal for Search Based Optimization. In *Proc. of the 13th International Conference on Fundamental Approaches to Software Engineering (FASE'10)* (2010), Springer, pp. 1–12.

[36] Havelund, K. *Using Runtime Analysis to Guide Model Checking of Java Programs*, vol. 1885. Springer-Verlag, 2000, pp. 245–264.

[37] Henzinger, T., Jhala, R., Majumdar, R., and Sanvido, M. Extreme model checking. In *Proc. of the International Symposium on Verification - Theory and Practice, 2003* (2003), Springer, pp. 332–358.

[38] Hovemeyer, D., and Pugh, W. Finding Bugs is Easy. *ACM SIGPLAN Notices 39*, 12 (Dec. 2004), 92–106.

[39] Huang, J., and Zhang, C. Persuasive prediction of concurrency access anomalies. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'11)* (2011), ACM Press, pp. 144–154.

[40] Jin, G., Song, L., Zhang, W., Lu, S., and Liblit, B. Automated atomicity-violation fixing. In *Proc. of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)* (2011), ACM Press, pp. 389–400.

[41] Jin, G., Zhang, W., Deng, D., Liblit, B., and Lu, S. Automated concurrency-bug fixing. In *Proc. of the Symposium on Operating System Design and Implementation (OSDI'12)* (2012).

[42] Joshi, P., Naik, M., Park, C., and Sen, K. CalFuzzer: An Extensible Active Testing Framework for Concurrent Programs. In *Proc. of the 21st International Conference on Computer Aided Verification (CAV'09)* (2009), Springer, pp. 675–681.

[43] Kelk, D., Jalbert, K., and Bradbury, J. *Automatically Repairing Concurrency Bugs with ARC.* In *Proc. of the 1st International Conference on Multicore Software Engineering, Performance, and Tools (MUSEPAT 2013)* (2013), pp. 73–84.

[44] Koza, J., and Poli, R. Genetic programming. *Search Methodologies* (2005), 127–164.

[45] Krena, B., Letko, Z., Tzoref, R., Ur, S., and Vojnar, T. Healing data races on-the-fly. In *Proc. of the 2007 ACM workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD'07)* (2007), ACM Press, pp. 54–64.

[46] Křena, B., Letko, Z., Vojnar, T., and Ur, S. A Platform for Search-based Testing of Concurrent Software. In *Proc. of the 8th Workshop on Parallel and Distributed Systems Testing, Analysis, and Debugging (PADTAD'10)* (2010), ACM Press, pp. 48–58.

[47] Lakhotia, K., McMinn, P., and Harman, M. Automated test data generation for coverage: Haven't we solved this problem yet? In *Proc. of the Testing: Academic and Industrial Conference-Practice and Research Techniques (TAIC PART'09)* (2009), IEEE, pp. 95–104.

[48] Lauterburg, S., Sobeih, A., Marinov, D., and Viswanathan, M. Incremental state-space exploration for programs with dynamically allocated data. In *Proc. of the 13th International Conference on Software Engineering (ICSE'08)* (2008), ACM Press, pp. 291–300.

[49] Le Goues, C., Dewey-vogt, M., Forrest, S., and Weimer, W. A systematic study of automated program repair: Fixing 55 out of 105 bugs for 8 dollars each. In *Proc of the 34th International Conference on Software Engineering (ICSE'12)* (2012), IEEE, pp. 3–13.

[50] Le Goues, C., Nguyen, T., Forrest, S., and Weimer, W. GenProg: A generic method for automated software repair. *Software Engineering, IEEE Transactions on 38*, 99 (2012), 54–72.

[51] Le Goues, C., Weimer, W., and Forrest, S. Representations and operators for improving evolutionary software repair. In *Proc. of the 14th International Conference on Genetic and Evolutionary Computation Conference (GECCO'12)* (2012), ACM Press, pp. 959–966.

[52] Lenguajes, D., Gomez-Pulido, J., Ferreira, M., and Alba, E. Finding deadlocks in large concurrent Java programs using genetic algorithms. In *Proc. of the 10th Annual Conference on Genetic and Evolutionary Computation (GECCO08)* (2008), ACM Press, pp. 1735–1742.

[53] Letko, Z., Vojnar, T., and Křena, B. AtomRace: Data race and atomicity violation detector and healer. In *Proc. of the 6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD'08)* (2008), ACM Press, p. 7.

[54] Lin, Y., and Kulkarni, S. Automatic repair for multi-threaded program with deadlock/livelock using maximum satisfiability, 2013.

[55] Liu, P., Tripp, O., and Zhang, C. Grail: context-aware fixing of concurrency bugs. In *Proc. of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FOSE 2014)* (2014), ACM, pp. 318–329.

[56] Liu, P., and Zhang, C. Axis: Automatically fixing atomicity violations through solving control constraints. In *Proc. of the International Conference on Software Engineering (ICSE'12)* (2012), IEEE Press, pp. 299–309.

[57] Liu, P., Zhang, C., Wang, Y., and Kelly, T. A unified approach to eliminating concurrency bugs via control synthesis, 2013.

[58] Lucia, B., Ceze, L., and Strauss, K. Colorsafe: Architectural support for debugging and dynamically avoiding multi-variable atomicity violations. In *ACM SIGARCH Computer Architecture News* (2010), vol. 38, ACM Press, pp. 222–233.

[59] Lucia, B., Devietti, J., Strauss, K., and Ceze, L. Atom-Aid: Detecting and surviving atomicity violations. In *Proc. of the 35th International Symposium on Computer Architecture (ISCA'08)* (2008), IEEE, pp. 277–288.

[60] Musuvathi, M., Qadeer, S., and Ball, T. CHESS: A systematic testing tool for concurrent software. *Redmond: Microsoft Research Technical Report, MSRTR-2007 149*, MSR-TR-2007-149 (2007), 16.

[61] Naik, M., and Aiken, A. Conditional Must not Aliasing for Static Race Detection. In *Proc. of the 34th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)* (Jan. 2007), vol. 42, ACM Press, pp. 327–338.

[62] Naik, M., Park, C., Sen, K., and Gay, D. Effective static deadlock detection. In *Proc. of the IEEE 31st International Conference on Software Engineering (ICSE'09)* (2009), IEEE, pp. 386–396.

[63] Nguyen, T., Weimer, W., Le Goues, C., and Forrest, S. Using Execution Paths to Evolve Software Patches. In *Proc. of 2nd International Workshop on Search-Based Software Testing (SBST 2009)* (2009), pp. 152–153.

[64] Nir-Buchbinder, Y., and Ur, S. Contest listeners: A concurrency-oriented infrastructure for java test and heal tools. In *Proc. of the 4th International Workshop on Software Quality Assurance (SOQUA'07)* (2007), ACM Press, pp. 9–16.

[65] Park, C.-S., and Sen, K. Randomized active atomicity violation detection in concurrent programs. In *Proc. of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'08)* (2008), ACM Press, pp. 135–145.

[66] Park, S., Vuduc, R. W., and Harrold, M. J. Falcon: Fault localization in concurrent programs. In *Proc. of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)* (2010), ACM Press, pp. 245–254.

[67] Pradel, M., and Gross, T. R. Fully automatic and precise detection of thread safety violations. In *ACM SIGPLAN Notices* (2012), vol. 47, ACM, pp. 521–530.

[68] Prvulovic, M. Cord: Cost-effective (and nearly overhead-free) order-recording and data race detection. In *Proc. of the 12th International Symposium on High-Performance Computer Architecture (HPCA'06)* (2006), IEEE, pp. 232–243.

[69] Qi, Y., Mao, X., Lei, Y., Dai, Z., and Wang, C. Does genetic programming work well on automated program repair? In *Proc. of the 5th International Conference on Computational and Information Sciences (ICCIS'13)* (2013), IEEE, pp. 1875–1878.

[70] Rungta, N., and Mercer, E. A meta heuristic for effectively detecting concurrency errors. In *Proc. of the 4th International Haifa Verification Conference (HVC'08)* (2008), Springer, pp. 23–37.

[71] Schneider, S. *The B-Method: An Introduction.* Palgrave, New York, 2001.

[72] Schulte, E., Fry, Z., Fast, E., Forrest, S., and Weimer, W. Software mutational robustness: Bridging the gap between mutation testing and evolutionary biology. Tech. rep., University of Virginia, Stanford and the University of New Mexico, Albuquerque, USA, 2012.

[73] Sen, K. Race directed random testing of concurrent programs. In *ACM SIGPLAN Notices* (2008), vol. 43, ACM Press, pp. 11–21.

[74] Smirnov, A., and Chiueh, T.-c. Dira: Automatic detection, identification and repair of control-hijacking attacks. In *In NDSS* (2005).

[75] Smolin, L. The fate of black hole singularities and the parameters of the standard models of particle physics and cosmology. *arXiv preprint gr-qc/9404011* (1994).

[76] Smolin, L. Scientific alternatives to the anthropic principle. *Universe or Multiverse* (2007), 323–366.

[77] Smolin, L. A perspective on the landscape problem. *Foundations of Physics 43*, 1 (2013), 21–45.

[78] Sobeih, A., and Lauterburg, S. Incremental state space exploration in J-Sim. Tech. rep., Department of Computer Science, University of Illinois at Urbana Champaign, 2007.

[79] Sokolsky, O., and Smolka, S. Incremental model checking in the modal mu-calculus. In *Computer Aided Verification (CAV'94)*, D. Dill, Ed., vol. 818 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1994, pp. 351–363.

[80] Staunton, J., and Clark, J. *Searching for Safety Violations Using Estimation of Distribution Algorithms*. IEEE, 2010, pp. 212–221.

[81] Staunton, J., and Clark, J. Applications of model reuse when using estimation of distribution algorithms to test concurrent software. *Search Based Software Engineering* (2011), 97–111.

[82] Ujma, M., and Shafiei, N. JPF-Concurrent: An extension of Java PathFinder for java.util.concurrent. *arXiv preprint arXiv:1205.0042* (2012).

[83] Veeraraghavan, K., Chen, P. M., Flinn, J., and Narayanasamy, S. Detecting and surviving data races using complementary schedules. In *Proc. of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)* (2011), ACM Press, pp. 369–384.

[84] Visser, W., Havelund, K., Brat, G., Park, S., and Lerda, F. Model checking programs. In *Proc. of the 15th IEEE International Conference on Automated Software Engineering (ASE 2000)* (2000), vol. 10, IEEE Computer Society, pp. 3–11.

[85] von Praun, C., and Gross, T. R. Object race detection. *ACM SIGPLAN Notices 36*, 11 (2001), 70–82.

[86] Wang, Y., Kelly, T., Kudlur, M., Lafortune, S., and Mahlke, S. A. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *OSDI* (2008), vol. 8, pp. 281–294.

[87] Weimer, W., Forrest, S., Le Goues, C., and Nguyen, T. Automatic program repair with evolutionary computation. *Communications of the ACM 53*, 5 (2010), 109–116.

[88] Weimer, W., Nguyen, T., Le Goues, C., and Forrest, S. Automatically finding patches using genetic programming. In *Proc. of the 31st International Conference on Software Engineering (ICSE'09)* (2009), IEEE Computer Society, pp. 364–374.

[89] White, D., Arcuri, A., and Clark, J. Evolutionary improvement of programs. *IEEE Transactions on Evolutionary Computation 15*, 4 (2011), 1–24.

[90] Whitley, D. An overview of evolutionary algorithms: Practical issues and common pitfalls. *Information and Software Technology 43*, 14 (2001), 817–831.

[91] Wilkerson, J., and Tauritz, D. Coevolutionary Automated Software Correction. In *Proc. of the Genetic And Evolutionary Computation Conference (GECCO'10)* (2010), pp. 1391–1392.

[92] Wu, J., Cui, H., and Yang, J. Bypassing races in live applications with execution filters. In *OSDI* (2010), pp. 135–149.