2009-09-14

Guided Testing for Automatic Error Discovery in Concurrent Software

Neha Shyam Rungta
Brigham Young University - Provo

GUIDED TESTING FOR AUTOMATIC ERROR DISCOVERY IN CONCURRENT SOFTWARE

by

Neha S. Rungta

A dissertation submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

Brigham Young University

December 2009

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a dissertation submitted by

Neha S. Rungta

This dissertation has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date Eric G. Mercer, Chair

Date Michael D. Jones

Date Kevin D. Seppi

Date Mark J. Clement

Date Parris Egbert

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the dissertation of Neha S. Rungta in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

Eric G. Mercer
Chair, Graduate Committee

Accepted for the Department

Date

Kent E. Seamons
Graduate Coordinator

Accepted for the College

Date

Thomas W. Sederberg
Associate Dean, College of Physical and Mathematical Sciences

ABSTRACT

GUIDED TESTING FOR AUTOMATIC ERROR DISCOVERY IN CONCURRENT SOFTWARE

Neha S. Rungta

Department of Computer Science

Doctor of Philosophy

The quality and reliability of software systems, in terms of their functional correctness, critically relies on the effectiveness of the testing tools and techniques to detect errors in the system before deployment. A lack of testing tools for concurrent programs that systematically control thread scheduling choices has not allowed concurrent software development to keep abreast with hardware trends of multi-core and multi-processor technologies. This motivates a need for the development of systematic testing techniques that detect errors in concurrent programs.

The work in this dissertation presents a potentially scalable technique that can be used to detect concurrency errors in production code. The technique is a viable solution for software engineers and testers to detect errors in multi-threaded programs before deployment. We present a guided testing technique that combines static analysis techniques, systematic verification techniques, and heuristics to efficiently detect errors in concurrent programs. An abstraction-refinement technique lies at the heart of the guided test technique. The abstraction-refinement technique uses as input potential errors in the program generated by imprecise, but scalable, static analysis tools. The abstraction further leverages static analyses to generate a set of program locations relevant in verifying the reachability of the potential error. Program execution is guided along these points by ranking both thread and data non-determinism. The set of relevant locations is refined when program execution is unable to make progress. The dissertation also discusses various heuristics for effectively guiding program execution. We implemented the guided test technique to detect errors in Java programs. Guided test successfully detects errors caused by thread schedules and data input values in Java benchmarks and the JDK concurrent libraries for which other state of the art analysis and testing tools for concurrent programs are unable to find an error.

ACKNOWLEDGMENTS

My incredible journey of graduate school and its culmination was made possible by the support and encouragement of many people. I thank each and every one.

Eric Mercer has been an amazing mentor, guide, and friend throughout this journey. He has greatly impacted this work by constantly probing the boundaries of my abilities, challenging me to strive harder, and enthusiastically cheering my successes. His efforts have been critical in the development and evolution of my research, writing, and presentation skills.

Mike Jones helped to instill the importance of having a cohesive dissertation and also taught me the importance of having a strong narrative. Kevin Seppi, Mark Clement, and Parris Egbert have been exemplary committee members and provided invaluable feedback. Suggestions and feedback by Willem Visser, my Google Summer of Code Mentor, has greatly improved this work. While discussions with Peter Mehlitz on the Java Pathfinder model checker enabled me to make better design choices while constructing the guided test framework.

The financial support provided by my advisor Eric Mercer, the Computer Science Department at BYU, and Google Anita Borg Scholarship has made for a smoother ride. The administrative staff at the Computer Science Dept. have been invaluable while traversing through paper work and deciphering graduate policies. A special cheer for Mindy and Shannon.

My friends Bryant, Cheree, Dave, Devlin, Josh, Sabra, Sam, Sole, and Topher have cheered for me, listened to me rant, advised me, made ice-cream runs with me, and even proof read papers close to Apia Samoa midnight deadlines. Being part of their lives has made this journey fun and enjoyable.

My family Shyam, Kusum, Vandana, and Hemant Rungta made it possible for me to embark on such an extraordinary journey. It is their love, sacrifices, and unwavering support that has made this journey possible. They encouraged me to dream and provided the means to fulfill those dreams. I dedicate this work to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Problem: Testing Concurrent Software Systems

Computing systems are increasingly pervasive in all aspects of life. Examples of computational systems include electronic appliances, smart phones, desktop computers, computing clusters, and super-computers. Even devices that are not generally associated as computational systems, like access cards, contain a microchip. Much of the software running on various computational systems is created through a development process that are: requirements, specification, design, implementation, testing, deployment, and maintenance. Significant effort is expended in overcoming the various challenges associated with each phase of the software development cycle. The work in this dissertation focuses on the testing phase of the development cycle.

Software testing encompasses all activities that evaluate various behaviors of the program against the requirements of the system. Testing can be applied to functional correctness, quality assurance, and the validation of software programs. The biggest challenge in software testing results from the complexity of software programs. It is not feasible to fully explore all possible behaviors of software programs of even moderate complexity. This problem is further exacerbated in concurrent programs since the number of possible behaviors increases exponentially with the number of concurrent processes in a given system.

In recent years, there has been a paradigm shift in software from inherently sequential programs to highly concurrent and parallel programs. The ubiquity of multi-core processors is prompting the paradigm shift from sequential to concurrent programs to better utilize the computation power of the processors. Stress testing is the current industry standard for finding bugs in parallel applications; however, its inability to directly control scheduling choices renders it insufficient and ineffective for error discovery. The lack of tools and techniques to test parallel applications has significantly hindered mainstream developers.

In concurrent systems, threads or processes simultaneously execute different parts of the program and communicate with each other through shared variables and other limited resources. To maintain the integrity of shared resources, different contending threads acquire and release synchronization elements, such as locks. Incorrect usage of the synchronization elements, however, can lead to concurrency errors such as deadlocks and race conditions.

Deadlocks and race-conditions are the most common concurrency errors in multi-threaded programs. A *deadlock* state is when threads in the program cannot make any progress. At such a point, the system generally has to be reset in order to resume execution. A *race condition* is an unwanted program state caused by a specific order of operations performed by different threads on a shared resource. Unprotected accesses to shared resources often leads to race-conditions. Deadlocks and race-conditions are manifested under very specific execution orders of threads and/or certain input from the user or environment.

Two forms of non-determinism need to be accounted for in order to detect errors in concurrent programs. One form is scheduling where the operating system decides which thread can execute. The other form of non-deterministic choice is program input. Program behavior can change based on the schedule chosen by the operating system and input provided by the user or the environment in concurrent systems. Reasoning about behaviors in concurrent systems is harder for developers and testers, compared to sequential programs, because the number of behaviors in a concurrent system increases exponentially with the number of threads. In a traditional testing environment, developers and testers test a few interactions of the different threads and a subset of all possible data values to check if any deadlocks or race-conditions exist in the program. More importantly, in most cases developers cannot control thread schedules and check interesting behaviors of the system. Hence, when the system is in use and an untested sequence of events leads to an error, it can cause the system to either crash or lead to unwanted behaviors.

Although parallel programming and concurrent systems are well studied in academia and a few specialized problem domains, it is not a paradigm commonly known in mainstream programming. As a result, there are few, if any, tools available to programmers to help them test and analyze concurrent program for functional correctness. This motivates a need to find effective and scalable techniques for discovering concurrency errors before deployment.

1.2 Impact on Developers and the Cost of Software

Access to efficient testing tools is an important aspect in allowing developers to create reliable programs. Systems and application programmers developing concurrent networked applications, web servers, device drivers, and file systems need tools for testing the programs. We expect to see an increase in the number of application programmers writing concurrent applications to take advantage of the multi-core processors. With the future computing moving toward concurrent programs the need for tools to test concurrent software affects essentially all developers.

Testing programs accounts for a significant portion of the cost associated with the software development cycle. The cost rises even further when defects in the system are detected after its deployment. The effects are magnified when defects are detected in capital and safety-critical systems. Efficient testing tools reduce the cost of testing software and fixing defects in deployed software, aids engineers in the development process, and improves the overall reliability of the software.

1.3 Current State of the Art Testing Techniques

Various approaches such as static analysis, testing, model checking, and symbolic execution have been used to detect errors in concurrent software systems. We provide a brief overview of each of the techniques and their limitations.

Static analysis techniques abstract the actual execution environment of the program and reason about errors by performing a control and data flow analysis of the system [Artho and Biere, 2001, Engler and Ashcraft, 2003, Flanagan et al., 2002, Hovemeyer and Pugh, 2004, Sterling, 1993]. Most scalable static analysis techniques report warnings that *may* exist in the program. The programmer has to manually verify the feasibility of each warning by reasoning about thread schedules, input values, and branch conditions required to manifest the error in a real execution path in the program. When manually verifying the warnings, a large part of the effort is expended in checking warnings that do not correspond to real errors (false positives). Static analysis techniques often report a large number of false positives. In essence, manual verification of the static analysis warnings is tedious and, in general, not feasible for large programs.

The current industry standard for detecting errors in concurrent programs is dynamic analysis based testing techniques that entail running the program. In stress testing the software programs are executed under heavy loads in order to elicit an interleaving that leads to an error. For example, a networked application can be stress tested with twice or more number of incoming connections than those expected under normal circumstances. In another example, a large number of threads can be created in a shared memory system that operate on the same shared resource to stress test the program. The inability of stress testing to control scheduling choices renders it ineffective in detecting errors in concurrent programs. In order to overcome this problem, random testing uses random inputs or thread schedules to find errors in the program [Csallner and Smaragdakis, 2004, Dwyer et al., 2007, Sen, 2007]. Random testing controls the runtime environment to randomly sample from the possible thread schedules during program execution. Our empirical evidence, however, shows that random testing obtains a partial coverage of the behavior space of the program within a specified time bound, and is not effective in detecting subtle concurrency errors that are only manifested along certain execution paths [Rungta and Mercer, 2007a].

Symbolic execution has been extensively used for test-case generation and error detection in sequential programs for over twenty years. Symbolic execution substitutes concrete data values of the program with symbolic values representing arbitrary values. Seminal work by King [1976] uses symbolic execution to test programs over a large range of numbers. Even though the symbolic representation is useful in abstracting the data input values, the different thread schedules in a concurrent program are not represented in the symbolic data. For each thread schedule a separate set of constraints over the symbolic data values is constructed. Checking the satisfiability of the constraints (constraint solving) to determine the feasibility of each path in the program is extremely expensive for concurrent programs because the number of paths increase exponentially with the number of threads in the system. This high-cost of constraint solving has not allowed symbolic execution to be effectively used in concurrent programs.

Model checking is a precise, sound, and complete analysis technique that systematically explores all possible behavior of a concurrent software system. Traditional model checking techniques extract a model of a system under test and verify its properties [Ball and Rajamani, 2001, Havelund and Pressburger, 1998, Holzmann, 2003, Robby et al., 2003]. The process of model extraction can often be tedious and error-prone. Software model checking techniques, pioneered by tools such as VeriSoft and Java PathFinder, verify the software *as is* without model extraction; that is, the software is the model [Godefroid, 1997, Visser et al., 2000a]. Software model checking tools take control over the scheduling process to reveal subtle concurrency bugs in the system. In contrast to random testing techniques that control the runtime environment and randomly sample from the possible thread schedules, software model checking systematically explores all possible thread schedules. Additionally, software model checking tools also explore all possible data input values to find errors in the system caused by thread schedules, data input values, or a combination of both. The growing complexity, however, of concurrent systems leads to an exponential growth in the number of possible behaviors (state space). This state space explosion has prevented the use of software model checking in mainstream test frameworks.

In contrast to exhaustively searching the system, certain model checking techniques also use heuristics to guide the search quickly toward the error [Edelkamp and Mehler, 2003, Edelkamp et al., 2001b, Groce and Visser, 2002b, Rungta and Mercer, 2005, 2006, Yang and Dill, 1998]. Directed model checking uses heuristic values and path-cost to rank the states in order of interest in a priority queue. Directed model checking uses some information about the program or the property being verified to generate heuristic values. The information is either specified by the user or computed automatically. An efficient heuristic leads to errors quickly compared to other exhaustive search techniques. In large programs, however, the size of the search frontier grows very rapidly causing an explosion in the number of states saved in the priority queue which proves to be the bottleneck in the success of directed model checking. Some heuristics also require manual configuration and tuning of parameters to be effective. The manual configuration requires a significant knowledge of the program and the error being verified in order to be successful in error discovery.

In summary the existing solutions lack the ability to effectively detect errors in concurrent systems. The existing solutions only work for small systems and do not scale well. The main limitations of the current technologies that our solution addresses are as follows:

- A significant amount of user-intervention is required to manually validate warnings reported by static analysis tools.
- The reliance of testing and dynamic analysis tools on the ad-hoc sampling of thread schedules and data values does not allow them to detect errors as demonstrated by our empirical evidence [Rungta and Mercer, 2007a].
- Precise verification techniques such as model checking and symbolic execution attempt to build an exhaustive proof of correctness that demonstrates the program to be free of all possible errors. Such an exhaustive proof construction is infeasible for concurrent programs with even moderate complexity using the current state of art technology.

1.4 Automated Error Discovery in Concurrent Programs

In this work we have designed a fully-automated technique to efficiently detect errors in concurrent systems. The main focus of the solution is an effective strategy for bug-detection in concurrent programs rather than building a complete proof of correctness. Before presenting the solution we discuss a series of observations that were important in the design and implementation of the solution. These observations are based on existing research and also extensive empirical studies conducted by us.

1.4.1 Observations on Existing Research

To efficiently detect errors in concurrent programs it is imperative to take control over the scheduling decisions in a manner similar to software model checking tools.

It is, however, infeasible to construct an exhaustive proof of complete correctness (absence of all errors) in concurrent programs with even moderate complexity using current state of the art software model checking techniques. Our solution assumes that building such a proof of complete correctness is not feasible and instead tries to detect errors in the system.

Guided model checking directs the search into areas of the behavior space where errors are more likely to exist. It assumes the existence of an error and focuses the resources in trying to find the error. Guided model checking techniques that use heuristics based on the information about possible errors in the program can lead to the feasible errors faster than a systematic search that does not use such information. As part of our solution we have designed heuristics to rank thread and data non-determinism to efficiently detect concurrency errors.

Most static analysis techniques can scale to large programs since they abstract the runtime environment of the program and only analyze the source of the program. Furthermore, static analysis techniques can be used to detect possible errors in the program and extract additional information about the possible errors. Our solution uses such information to guide the program execution toward the possible errors and automatically detect feasible errors.

1.4.2 Observations on Our Empirical Studies

Randomization while guiding the program execution is an important element in overcoming the limitations of the default search order arising from heuristic ties. In Appendix A, our empirical study shows that randomization while guiding the program search, overall, decreases the number of states generated before error discovery when compared to a search with no randomization using the same heuristic [Rungta and Mercer, 2007b]. Our solution shuffles states in a priority queue or search stack with equivalent heuristic ties to overcome the limitations of the default search order.

It is important to evaluate the effectiveness of a technique in detecting errors on benchmarks that have been characterized with hard to find errors. In Appendix B our empirical study demonstrates that a hardness metric based on a stateful randomized depth-first search is a good baseline measure. Furthermore, it also shows how to convert easy models into hard models by pushing errors deeper in the system and manipulating the number of threads that actually manifest an error [Rungta and Mercer, 2007a]. We evaluate the effectiveness of our solution in error discovery on models with hard to find errors and compare it with other state of the art techniques.

1.4.3 Guided Test

To automatically detect errors in concurrent programs, we present a guided test technique that leverages information from static analysis techniques to systematically guide program execution toward possible error locations in the program. A class of heuristics have been designed to efficiently guide program execution and quickly detect feasible errors.

Guided test combines static analysis techniques, systematic verification techniques, and heuristics to automatically detect errors in concurrent programs. Imprecise static analysis techniques identify possible errors in the program and also generate a set of program locations relevant in determining the feasibility of a possible error in the program. Program execution is guided along the set of the relevant program locations to determine the feasibility of the error. A combination of heuristics and stochastic methods have been developed to efficiently guide the program execution. The heuristics in guided test rank, both, thread and data non-determinism to find errors in concurrent programs. The set of relevant program locations is refined by adding new locations when program execution is unable to make progress.

The guided test solution overcomes the limitations of existing work to efficiently detect errors in concurrent software systems. Unlike exhaustive software model checking and symbolic execution techniques, guided test assumes the existence of an error and does not attempt to construct a proof of correctness. It recognizes that such a proof construction is not feasible for programs with even moderate complexity and rather focuses on generating a witness for a possible error or static analysis warning. Guided test uses the information about possible errors in the program generated from scalable static analysis and uses the information to guide the program execution toward these possible errors. In contrast to static analysis techniques, however, the guided technique is sound in error discovery. No further manual investigation of the error is required. Finally, unlike traditional testing techniques, guided test is not reliant on an ad-hoc sampling of the thread schedules to detect the error. Instead, it systematically guides program execution toward parts of the behavior space that are more likely to contain errors.

1.5 Effectiveness of Guided Test

Guided test is a potentially scalable technique that can be used to detect concurrency errors in production code. Guided test is a viable solution for software engineers and testers to detect errors in multi-threaded programs before deployment. To evaluate the effectiveness of the guided test technique we use the algorithm to detect errors caused by thread schedules and data input values in Java benchmarks and the JDK concurrent libraries. We use the Java Pathfinder (JPF) model checker as the verification engine and as a platform to implement the guided test solution.

We evaluate the bug detection capabilities of various tools and techniques and compare them to the guided testing technique. We have compiled a set of Java benchmarks from various sources and our own efforts. For many of the Java examples we have created structurally equivalent C# programs. In each C# model corresponding to a Java program the same number of threads are created, similar data structures are instantiated, threads access the same data structures, and threads perform the same synchronization operations. Note that since Java and C# have very similar execution models we can recreate the same programs in both languages. In our multi-language benchmark suite we compare results from various tools: CalFuzzer, ConTest, CHESS, and Java Pathfinder.

We provide extensive results using Java Pathfinder for stateless random walk, randomized depth-first search, and guided testing. Using the data from our study that was conducted on benchmarks categorized as having *hard to find errors*, we demonstrate that iterative context-bounding and dynamic partial order reduction are not sufficient to render model checking for testing concurrent programs tractable. Iterative context-bounding limits the number of preemption points along a single path while partial order reduction techniques reduces the number of thread schedules that need to be explored in concurrent programs.

Exhaustive symbolic execution is unable to discover errors within a time bound of one hour in the benchmarks that use the JDK concurrent library in accordance with the documentation and contain both thread and data non-determinism. While the guided test only takes a few seconds to find the errors in the same models.

The extensive comparative analysis presented in this work demonstrates that guided testing techniques such as the one presented in this dissertation are essential to make software model checking tractable for testing concurrent programs. A prototype implementation of the algorithm is a `guidedsymbolic` extension that is now part of the JPF model checker. The extension can be obtained by downloading `JPF` from `http://javapathfinder.sourceforge.net`.

1.6 Summary Of The Contributions

We now present a detailed overview of the various research contributions of this work. Each contribution is an important element in automatically guiding program execution in order to detect errors in concurrent programs. In the rest of this section we discuss each of the research contributions, we also list each paper published for the corresponding research contribution.

1.6.1 Guided Program Execution using Abstraction-Refinement

- N. Rungta, E. G. Mercer, and W. Visser, “Efficient Testing of Concurrent Programs with Abstraction-Guided Symbolic Execution”, in *Proceedings of the SPIN Workshop on Software Model Checking*, Grenoble, France, June 2009.

Chapter 2 in this dissertation presents an abstraction-refinement technique that guides the program execution in order to determine the feasibility of a possible error. The possible error locations (target locations) in the program are generated using imprecise, but scalable, static analysis techniques. The technique further leverages static analysis techniques to generate an abstraction of the program that consists of program locations relevant in verifying the reachability of the potential error. The abstract system encodes the sequence of relevant program locations that determine the reachability of the target locations: call sites, conditional branch statements, and data definitions. The initial abstraction is generated along the sequential flow of the program and does not consider any inter-thread dependence.

The program execution is guided along the sequence of program locations in the abstract system using a variety of heuristics described in later chapters. The heuristics are used to rank, both, thread and data non-determinism. At points when the program execution is unable to follow the sequence of program location due to the value of certain global variables (e.g. in a certain conditional branch statement), the refinement process is invoked. The refinement adds the thread inter-dependence information by adding definitions of global variables of interest to the abstract system. The execution is then restarted along a new sequence of program locations in the abstract system.

1.6.2 Meta-Heuristic for Concurrent Programs

- N. Rungta and E. G. Mercer, "A Meta Heuristic for Effectively Detecting Concurrency Errors", in *Proceedings of Haifa Verification Conference (HVC)*, Haifa, Israel, November 2008.

Chapter 3 in this dissertation presents a meta heuristic that guides the program execution in a greedy depth-first manner along a sequence of program locations that are relevant in determining the feasibility of a possible error in the program. Using a greedy depth-first search allows us to overcome the memory bottlenecks of priority-queue based greedy best-first and A^* searches. The greedy depth-first search picks the best immediate successor of the current state and does not consider unexplored successors until it reaches the end of a path and needs to backtrack. The backtracking is stochastic and, also considers the rank assigned to the state by the meta heuristic.

The meta heuristic ranks the states based on the number of locations already observed from the sequence of program locations. States that have observed a greater number of locations from the sequence of relevant program locations are ranked as more interesting compared to the other states. In the case where multiple states have observed the same number of locations in the sequences, the meta heuristic uses a secondary heuristic to guide the search toward the next location in the sequence.

The empirical analysis in this work demonstrates that the meta-heuristic is extremely effective in localizing a feasible error, given the set of relevant locations. Furthermore, the results that the choice of the secondary heuristic has a dramatic effect on the time required for error discovery.

1.6.3 A Distance Heuristic for Programs with Polymorphism

- N. Rungta and E. G. Mercer, "Guided Model Checking for Programs with Polymorphism", in *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*, Savannah, Georgia, USA, January 2009.

Chapter 4 in this dissertation discusses a new distance estimate heuristic that efficiently computes a tighter lower-bound in programs with polymorphism when compared to the state of the art FSM distance heuristic. Distance heuristics guide the program execution toward a program location. The heuristic rank is an estimate of the number of transitions required by each thread to reach the desired program location. In Java programs that contain dynamic method invocation, the initial estimate is computed on a statically generated abstract model of the program that ignores all data values and only considers control flow. The estimates are dynamically refined when the targets of dynamic method invocations are resolved. All ties in heuristic values are randomly resolved.

In our empirical analysis the state of the art FSM distance heuristic is computationally infeasible for large programs with polymorphism while the new distance distance heuristic can quickly detect the errors. Another empirical analysis also demonstrates that the new distance heuristic is the most effective in error discovery as a secondary heuristic when used with the meta heuristic, compared to other state of the art heuristics.

Chapter 2

Guided Program Execution using Abstraction-Refinement

This chapter was published as:

N. Rungta, E. G. Mercer, and W. Visser, "Efficient Testing of Concurrent Programs with Abstraction-Guided Symbolic Execution", in *Proceedings of the SPIN Workshop on Software Model Checking*, Grenoble, France, June 2009.

Abstract

In this work we present an abstraction-guided symbolic execution technique that quickly detects errors in concurrent programs. The input to the technique is a set of target locations that represent a possible error in the program. We generate an abstract system from a backward slice for each target location. The backward slice contains program locations relevant in testing the reachability of the target locations. The backward slice only considers sequential execution and does not capture any inter-thread dependencies. A combination of heuristics are to guide a symbolic execution along locations in the abstract system in an effort to generate a corresponding feasible execution trace to the target locations. When the symbolic execution is unable to make progress, we refine the abstraction by adding locations to handle inter-thread dependencies. We demonstrate empirically that abstraction-guided symbolic execution generates feasible execution paths in the actual system to find concurrency errors in a *few seconds* where exhaustive symbolic execution fails to find the same errors in an hour.

2.1 Introduction

The current trend of multi-core and multi-processor computing is causing a paradigm shift from inherently sequential to highly concurrent and parallel applications. Certain thread interleavings, data input values, or combinations of both often cause errors in the system. Systematic verification techniques such as explicit state model checking and symbolic execution are extensively used to detect errors in such systems [Godefroid, 1997, Holzmann, 2003, King, 1976, Păsăreanu et al., 2008, Visser et al., 2003].

Explicit state model checking enumerates all possible thread schedules and input data values of a program in order to check for errors [Holzmann, 2003, Visser et al., 2003]. To partially mitigate the state space explosion from data input values, symbolic execution techniques substitute data input values with symbolic values [King, 1976, Păsăreanu et al., 2008, Tomb et al., 2007]. Explicit state model checking and symbolic execution techniques used in conjunction with exhaustive search techniques such as depth-first search are unable to detect errors in medium to large-sized concurrent programs because the number of behaviors caused by data and thread non-determinism is extremely large.

In this work we present an abstraction-guided symbolic execution technique that efficiently detects errors caused by a combination of thread schedules and data values in concurrent programs. The technique generates a set of key program locations relevant in testing the reachability of the target locations. The symbolic execution is then guided along these locations in an attempt to generate a feasible execution path to the error state. This allows the execution to focus in parts of the behavior space more likely to contain an error.

A set of target locations that represent a possible error in the program is provided as input to generate an abstract system. The input target locations are either generated from static analysis warnings, imprecise dynamic analysis techniques, or user-specified reachability properties. The abstract system is constructed with program locations contained in a static interprocedural backward slice for each target location and synchronization locations that lie along control paths to the target locations [Horwitz et al., 2004]. The static backward slice contains call sites, conditional branch statements, and data definitions that determine the reachability of a target location. The backward slice only considers sequential control flow execution and does not contain data values or inter-thread dependencies.

We systematically guide the symbolic execution toward locations in the abstract system in order to reach the target locations. A combination of heuristics are used to automatically pick thread identifiers and input data values at points of thread and data non-determinism respectively. We use the abstract system to guide the symbolic execution and do not verify or search the abstract system like most other abstraction refinement techniques [Ball and Rajamani, 2001, Henzinger et al., 2003]. At points when the program execution is unable to move further along a sequence of locations (e.g. due to the value of a global variable at a particular conditional statement), we refine the abstract system by adding program statements that re-define the global variables. The refinement step adds the inter-thread dependence information to the abstract system on a need-to basis. The contributions of this work are as follows:

1. An abstraction technique that uses static backward slicing along a sequential control flow execution of the program to generate relevant locations for checking the reachability of certain target locations.
2. A guided symbolic execution technique that generates a feasible execution trace corresponding to a sequence of locations in the abstract system.
3. A novel heuristic that uses the information in the abstract system to rank data non-determinism in symbolic execution.

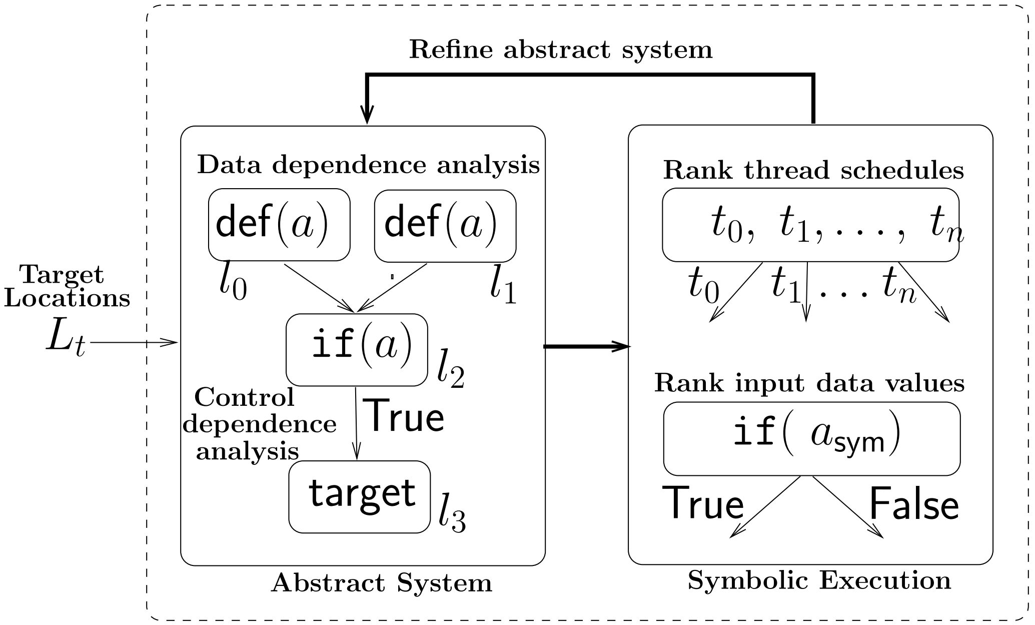


Figure 2.1: Overview of the abstraction-guided symbolic execution technique

4. A refinement heuristic to add inter-thread dependence information to the abstract system when the program execution is unable to make progress.

We demonstrate in an empirical analysis on benchmarked multi-threaded Java programs and the JDK 1.4 concurrent libraries that locations in the abstract system can be used to generate feasible execution paths to the target locations. We show that the abstraction guided-technique can find errors in multi-threaded Java programs in a *few seconds* where exhaustive symbolic execution is unable to find the errors within a time bound of an hour.

2.2 Overview

A high-level overview of the technique is shown in Figure 2.1.

Input: The input to the technique is a set of target locations, L_t, that represent a possible error in the program. The target locations can either be generated using a static analysis tool or a user-specified reachability property. The lockset analysis, for example, reports program locations where lock acquisitions by unique threads may lead to a deadlock [Engler and Ashcraft, 2003]. The lock acquisition locations generated by the lockset analysis are the input target locations for the technique.

Abstract System: An abstraction of the program is generated from backward slices of the input target locations and synchronization locations that lie along control paths to the target locations. Standard control and data dependence analyses are used to generate the backward slices. Location l_3 is a single target location in Figure 2.1. The possible execution of location l_3 is control dependent on the *true* branch of the conditional statement l_2. Two definitions of a *global variable* a at locations l_0 and l_1 reach the conditional statement l_2; hence, locations l_0, l_1, and l_2 are part of the abstract system. These locations are directly relevant in testing the reachability of l_3.

Abstraction-Guided Symbolic Execution: The symbolic execution is guided along a sequence of locations (an abstract trace: $\langle l_0, l_2, l_3 \rangle$) in the abstract system. The program execution is guided using heuristics to intelligently rank the successor states generated at points of thread and data non-determinism. The guidance strategy uses information that l_3 is control dependent on the *true* branch of location l_2 and in the ranking scheme prefers the successor representing the *true* branch of the conditional statement.

Refinement: When the symbolic execution cannot reach the desired target of a conditional branch statement containing a global variable we refine the abstract system by adding inter-thread dependence information. Suppose, we cannot generate the successor state for the *true* branch of the conditional statement while guiding along $\langle l_0, l_2, l_3 \rangle$ in Figure 2.1, then the refinement automatically adds another definition of a to the abstract trace resulting in $\langle l_1, l_0, l_2, l_3 \rangle$. The new abstract trace implicitly states that two different threads need to define the variable a at locations

```
Thread A{
...

public void run(Element elem){
  lock(elem)
    check(elem)
  unlock(elem)
}

public void check(Element elem)
  if elem.e > 9
    Throw Exception
}}
```

(a)

```
Thread B {
...

public void run(Element elem){
  int x   /* Input Variable */
  if x > 18
    lock(elem)
      elem.reset()
    unlock(elem)
}}
```

(b)

```
Object Element{
int e
...
public Element(){
  e := 1
}
public void reset(){
  e := 11
}}
```

(c)

Figure 2.2: An example of a multi-threaded program with two threads `A` and `B` that operate on a shared variable *elem* of type `Element`

l_1 and l_0. Note that there is no single control flow path that passes through both l_1 and l_0.

Output: When the guided symbolic execution technique discovers a feasible execution path we output the trace. The technique, however, cannot detect infeasible errors. In such cases it outputs a *"Don't know"* response.

2.3 Program Model and Semantics

To simplify the presentation of the guided symbolic execution we describe a simple programming model for multi-threaded and object-oriented systems. The restrictions, however, do not apply to the techniques presented in this work and the empirical analysis is conducted on Java programs. Our programs contain conditional branch statements, procedures, basic data types, complex data types supporting polymorphism, threads, exceptions, assertion statements, and an explicit locking mechanism. The threads are separate entities. The programs contain a finite number of threads with no dynamic thread creation. The threads communicate with each other through shared variables and use explicit locks to perform synchronization operations. The program can also seek input for data values from the environment.

In Figure 2.2 we present an example of such a multi-threaded program with two threads `A` and `B` that communicate with each other through a shared variable, *elem*, of type `Element`. Thread `A` essentially checks the value *elem.e* at line 9 in Figure 2.2(a) while thread `B` resets the value of *elem.e* in Figure 2.2(b) at line 7 by invoking the `reset` function shown in Figure 2.2(c). We use the simple example in Figure 2.2 through the rest of the paper to demonstrate how the guided symbolic execution technique works.

A multi-threaded program, $\mathcal{M}$, is a tuple $\langle\{\mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_{u-1}\}, V_c, D_{sym}\rangle$ where each $\mathcal{T}_i$ is a thread with a unique identifier $id \rightarrow \{0, 1, \ldots, u-1\}$ and a set of local variables; V_c is a finite set of concrete variables; and D_{sym} is a finite set of all input data variables in the system. An input data variable is essentially any variable that seeks a response from the environment.

A runtime environment implements an interleaving semantics over the threads in the program. The runtime environment operates on a program state s that contains: (1) valuations of the variables in V_c, (2) for each thread, $\mathcal{T}_i$, values of its local variables, runtime stack, and its current program location, (3) the symbolic representations and values of the variables in D_{sym}, and (4) a path constraint, ϕ, (a set of constraints) over the variables in D_{sym}. The runtime environment provides a set of functions to access certain information in a program state s:

- `getCurrentLoc`(s) returns the current program location of the most recently executed thread in state s.

- `getLoc`(s, i) returns the current program location of the thread with identifier i in state s.
- `getEnabledThreads`(s) returns a set of identifiers of the threads enabled in s. A thread is *enabled* if it is not blocked (not waiting to acquire a lock).

Given a program state, s, the runtime environment generates a set of successor states, $\{s_0, s_1, \ldots, s_n\}$ based on the following rules $\forall i \in \texttt{getEnabledThreads}(s) \wedge l := \texttt{getLoc}(s, i)$:

1. If l is a conditional branch with symbolic primitive data types in the branch predicate, P, the runtime environment can generate at most two possible successor states. It can assign values to variables in D_{sym} to satisfy the path constraint $\phi \wedge P$ for the target of the true branch or satisfy its negation $\phi \wedge \neg P$ for the target of the false branch.
2. If l accesses an uninitialized symbolic complex data structure o_{sym} of type T, then the runtime environment generates multiple possible successor states where o_{sym} is initialized to: (a) null, (b) references to new objects of type T and all its subtypes, and (c) existing references to objects of type T and all its subtypes [Khurshid et al., 2003].
3. If neither rule 1 or 2 are satisfied, then the runtime environment generates a single successor state obtained by executing l in thread $\mathcal{T}_i$.

In the initial program state, s_0, the current program location of each thread is initialized to its corresponding start location while the variables in D_{sym} are assigned a symbolic value $v_\perp$ that represents an uninitialized value.

A state s_n is reachable from the initial state s_0 if using the runtime environment we can find a non-zero sequence of states $\langle s_0, s_1, \ldots, s_n \rangle$ that leads from s_0 to s_n such that $\forall \langle s_i, s_{i+1} \rangle$, s_{i+1} is a successor of s_i for $0 \leq i \leq n-1$. Such a sequence of program states represents a feasible execution path through the system. The sequence of program states provides a set of concrete data values and a valid path constraint over the symbolic values. The reachable state space, S, can be generated using the runtime environment where $S := \{s \mid \exists \langle s_0, \ldots, s \rangle\}$.

A depth-first or breadth-first search can systematically generate and search the reachable state space using the runtime environment in an attempt to find the target state. In most programs, however, the reachable state space is so large that exhaustive search techniques are ineffective in finding the target state.

2.4 Abstraction

In this work we create an abstract system that contains program locations relevant in checking the reachability of the target locations. We then use the locations in the abstract system to guide the symbolic execution. The abstract system is constructed with program locations contained in a static interprocedural backward slice for each target location. The abstract system also contains synchronization locations that lie along control paths to the target locations. A backward slice of a program with respect to a program location l and a set of program variables V consists of all statements and predicates in the program that may affect the value of variables in V at l and the reachability of l.

2.4.1 Background definitions

Definition 1. *A control flow graph (CFG) of a procedure in a system is a directed graph* $G := \langle L, E \rangle$ *where* L *is a set of uniquely labeled program locations in the procedure while* $E \subseteq L \times L$ *is the set of edges that represents the possible flow of execution between the program locations. Each CFG has a start location* $l_{start} \in L$ *and an end location* $l_{end} \in L$.

Definition 2. *$\texttt{cfgPath}(l, l')$ describes a path in a CFG and returns true iff there exists a sequence of $q := \langle l, \ldots, l' \rangle$ such that $(l_i, l_{i+1}) \in E$ where $0 \leq i \leq \mathsf{length}(q) - 1$ and E is the set of edges in the CFG.*

We define the following functions to access information about the locations and edges in the CFGs:

- $\texttt{start}(l)$ returns true iff l is a start location.
- $\texttt{end}(l)$ returns true iff l is an end location.
- $\texttt{callSite}(l)$ returns true iff l is a call site that invokes a procedure.
- $\texttt{branch}(l)$ returns true iff l is a conditional branch statement.
- $\texttt{aquireLock}(l)$ returns true iff l acquires a lock.
- $\texttt{releaseLock}(l, l')$ returns true iff l releases a lock that is acquired at l'.
- $\texttt{cfgEdge}(l, l')$ returns true iff $\exists (l, l') \in \mathcal{E}$.
- $\texttt{callEdge}(l, l')$ returns true iff $\texttt{callSite}(l) \wedge \texttt{start}(l') \wedge$ l invokes l'.

Definition 3. *An interprocedural control flow graph (ICFG) for a system with p procedures is $\langle \mathcal{L}, \mathcal{E} \rangle$ where $\mathcal{L} := \bigcup_{0 \leq i \leq p} L_i$ and $\mathcal{E} := \bigcup_{0 \leq i \leq p} E_i$. Additional edges from a call site to the start location of the callee and from the end location of a procedure back to its caller are also added in the ICFG.*

Definition 4. *$\texttt{icfgPath}(l, l')$ describes a path in the ICFG and returns true iff there exists a sequence $q := \langle l, \ldots, l' \rangle$ such that $(l_i, l_{i+1}) \in \mathcal{E}$ where $0 \leq i \leq \mathsf{length}(q) - 1$*

Data and control dependences are an integral part in constructing the abstract system. The dependence analyses are defined along intraprocedural paths in the CFG. Data dependence is primarily based on *reaching definitions*; whereas control dependence is determined by whether the outcomes of branch predicates in conditional statements affect the reachability of certain locations. The definitions of the dependence analyses are as follows:

Definition 5. *$\texttt{postDom}(l, l')$ returns true iff for each path in a CFG between l and l_{end}, $q := \langle l, \ldots, l_{end} \rangle$, where l_{end} is an end location, and there exists an i such that $l_i = l'$ where $1 \leq i \leq \mathsf{length}(q) - 1$.*

Definition 6. *$\texttt{defines}(l, v)$ returns true iff the variable, v, is defined at program location l.*

Definition 7. *$\texttt{uses}(l, v)$ returns true iff the value of variable, v, is used at program location l.*

Definition 8. *$\texttt{reaches}(l, l')$ returns true iff there exists a path $q := \langle l, \ldots, l' \rangle$ such that $\texttt{icfgPath}(l, l') \wedge \texttt{defines}(l, v) \wedge \neg\texttt{defines}(l'', v) \wedge \texttt{uses}(l', v)$, for i, j where $l_i = l$, $l_j = l''$, and $i + 1 \leq j \leq \mathsf{length}(q)$.*

Definition 9. *$\texttt{controlD}(l, l')$ returns true iff there exists a path $q := \langle l, \ldots, l' \rangle$ such that $\texttt{cfgPath}(l, l') \wedge \texttt{branch}(l) \wedge \texttt{postDom}(l'', l') \wedge \neg\texttt{postDom}(l, l')$ for i, j where $l_i = l$, $l_j = l''$, and $i + 1 \leq j \leq \mathsf{length}(q)$.*

2.4.2 Abstract System

The abstract system is a directed graph $\mathcal{A} := \langle L_\alpha, E_\alpha \rangle$ where $L_\alpha \subseteq \mathcal{L}$ is the set of program locations while $E_\alpha \subseteq L_\alpha \times L_\alpha$ is the set of edges. The abstract system contains target locations; call sites, conditional branch statements, and data definitions in the backward slice of each target location; and all possible start locations of the program. It also contains synchronization operations that lie along control paths from the start of the program to the target locations.

To compute an interprocedural backward slice, a backwards reachability analysis can be performed on a system dependence graph [Horwitz et al., 2004]. Note that the backward slice only considers sequential execution and ignores all inter-thread dependencies. We describe the construction of the abstract system in the rest of this section.

The abstract system is constructed based on the set of input target locations L_t and the CFGs of the system $\langle \mathcal{L}, \mathcal{E} \rangle$. We initialize the set of abstract locations, L_α, with the set of target locations L_t and the set of all possible start locations of the program L_s. The set L_s contains the start location of each thread, $\mathcal{T}_i$, in the system. We initialize the set $L_\alpha := L_t \cup L_s$ and iteratively add locations, $l \in \mathcal{L}$, to L_α if one of the following four equations is satisfied. We continue to add locations until we reach a fixpoint, re-evaluating the four equations each time a location is added.

$$\begin{gathered} \exists l_t \in L_t, l_s \in L_s, l' \in \mathcal{L}, [\texttt{icfgPath}(l, l_t) \wedge \texttt{icfgPath}(l', l_t)] \wedge \\ [\texttt{icfgPath}(l_s, l) \wedge \texttt{icfgPath}(l_s, l')] \wedge [\texttt{callEdge}(l, l') \vee \texttt{callEdge}(l', l)] \end{gathered} \tag{2.1}$$

The call sites are added to L_α one at a time by satisfying Equation 2.1; the call sites are part of method sequences such that invoking a particular sequence leads from the start of the program to a procedure containing a target location. In addition to the call sites, start locations of the procedures invoked by the call sites are also added one at a time to the set of locations L_α when Equation 2.1 evaluates to true.

$$\begin{gathered} \exists l_\alpha \in L_\alpha,\ l' \in \mathcal{L},\ [\texttt{cfgPath}(l, l_\alpha) \wedge \texttt{cfgPath}(l', l_\alpha)] \wedge \\ \big\{ [\texttt{cfgEdge}(l, l') \wedge \texttt{controlD}(l, l') \wedge \texttt{controlD}(l, l_\alpha)] \vee \\ [\texttt{cfgEdge}(l', l) \wedge \texttt{controlD}(l', l) \wedge \texttt{controlD}(l', l_\alpha)] \big\} \end{gathered} \tag{2.2}$$

Conditional branch statements that determine the reachability of the locations that are already present in the abstract system are added to L_α whenever Equation 2.2 is satisfied. The branch statements that result from any nested control dependence are also added. Furthermore, the immediate target of a conditional branch statement is added when Equation 2.2 evaluates to true where the execution of the target depends on the same branch outcome as l_α. This allows the desired target of the branch to be encoded in the abstract trace.

$$\exists l_\alpha \in L_\alpha,\ \texttt{icfgPath}(l, l_\alpha) \wedge \texttt{defines}(l, v) \wedge \\ \texttt{isBranch}(l_\alpha) \wedge \texttt{uses}(l_\alpha, v) \wedge \texttt{reaches}(l, l_\alpha) \tag{2.3}$$

Locations that define variables used in branch predicates at the conditional statements in L_α are added if Equation 2.3 is satisfied. To compute the reaching definitions we conservatively compute the alias information based on the notion of *maybe an alias*. If two variables in a given procedure can be aliases of one another we assume they are aliases.

In order to add the synchronization locations we define the auxiliary functions $\texttt{acqLock}(l)$ that returns true iff l acquires a lock and $\texttt{relLock}(l, l')$ that returns true iff l releases a lock that is acquired at l'. For each $l_\alpha \in L_\alpha$ we update $L_\alpha := L_\alpha \cup l$ if Equation 2.4 is satisfied for l.

$$[\texttt{icfgPath}(l, l_\alpha)\ \wedge \texttt{acqLock}(l)] \vee [\texttt{icfgPath}(l_\alpha, l) \wedge \texttt{relLock}(l, l_\alpha)] \tag{2.4}$$

After the addition of the synchronization locations and locations from the backward slices we connect the different locations. Edges between the different locations in the abstract system are added based on the control flow of the program as defined by the ICFG. To map the execution order of the program locations in the abstract system to execution order in the ICFG we check the post-dominance relationship between the locations while adding the edges. An edge between any two locations l_α and l'_α in L_α is added to E_α if Equation 2.5 evaluates to true.

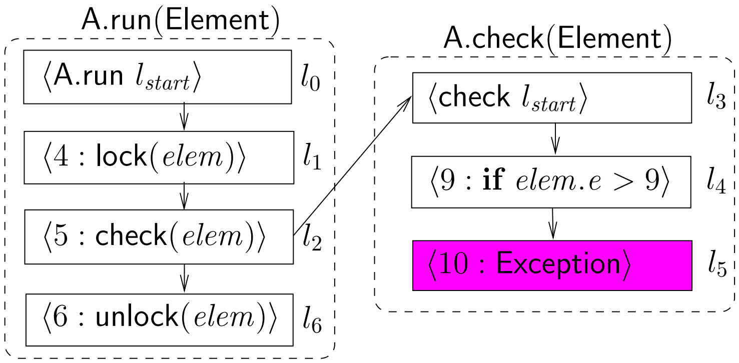


Figure 2.3: Initial abstract system.

$$\forall(l''_\alpha \in L_\alpha) \textit{ such that } \neg\texttt{postDom}(l_\alpha, l''_\alpha) \vee \neg\texttt{postDom}(l''_\alpha, l'_\alpha) \tag{2.5}$$

The abstract system for the example in Figure 2.2 where the target location is line 10 in the `check` method in Figure 2.2(a) is shown in Figure 2.3. Locations l_0 and α_0 in Figure 2.3 are the two start locations of the program. The target location, l_5, represents line 10 in Figure 2.2(a). Location l_2 is a call site that invokes start location l_3 that reaches target location l_5. The target location is control dependent on the conditional statement at line 9 in Figure 2.2(a); hence, l_4 is part of the abstract system in Figure 2.3. The locations l_1 and l_6 are the lock and unlock operations. The abstract system shows Thread `B` is not currently relevant in testing the reachability of location l_5.

2.4.3 Abstract Trace Set

The input to the guided symbolic execution is an abstract trace set. The abstract trace set contains sequences of locations generated on the abstract system, $\mathcal{A}$, from the start of the program to the various target locations in L_t. We refer to the sequences generated on the abstract system as *abstract traces* to distinguish them from the sequences generated on the CFGs. To construct the abstract trace set we first generate intermediate abstract trace sets, $\{P_0, P_1, \dots P_{t-1}\}$, that contain abstract traces between start locations of the program (L_s) and the input target locations (L_t); hence, $P_i := \{\pi | \pi$ satisfies Equation 2.6 and Equation 2.7 $\}$. We use the array indexing notation to reference elements in π, hence, $\pi[i]$ refers to the i^{th} element in π.

$$\exists l_0 \in L_s, l_t \in L_t \textit{ such that } \pi[0] == l_0 \wedge \pi[\mathsf{length}(\pi) - 1] == l_t \tag{2.6}$$

$$(\pi[i], \pi[i+1]) \in E_\alpha \wedge (i \neq j \implies \pi[i] \neq \pi[j]) \textit{ for } 0 \leq i, j \leq \mathsf{length}(\pi) - 1 \tag{2.7}$$

Equation 2.7 generates traces of finite length in the presence of cycles in the abstract system caused by loops, recursion, or cyclic dependencies in the program. Equation 2.7 ensures that each abstract trace generated does not contain any duplicate locations by not considering any back edges arising from cycles in the abstract system. We rely on the guidance strategy to drive the program execution through the cyclic dependencies toward the next interesting location in the abstract trace; hence, the cyclic dependencies are not encoded in the abstract traces that are generated from the abstract system.

Each intermediate abstract trace set, P_i, contains several abstract traces from the start of the program to a single target location $l_i \in L_t$. We generate a set of final abstract trace sets as:

$$\Pi_{\mathcal{A}} := \{\{\pi_0, \dots, \pi_{t-1}\} | \pi_0 \in P_0, \dots, \pi_{t-1} \in P_{t-1}\}$$

Each $\Pi_\alpha \in \Pi_\mathcal{A}$ contains a set of abstract traces. $\Pi_\alpha := \{\pi_{\alpha_0}, \pi_{\alpha_1}, \ldots, \pi_{\alpha_{t-1}}\}$ where each $\pi_{\alpha_i} \in \Pi_\alpha$ is an abstract trace leading from the start of the program to a unique $l_i \in L_t$. Since there exists an abstract trace in Π_α for each target location in L_t, $|\Pi_\alpha| == |L_t|$.

The input to the guided symbolic execution technique is $\Pi_\alpha \in \Pi_\mathcal{A}$. The different abstract trace sets in $\Pi_\mathcal{A}$ allow us to easily distribute checking the feasibility of individual abstract trace sets on a large number of computation nodes. Each execution is completely independent of another and as soon as we find a feasible execution path to the target locations we can simply terminate the other trials.

In the abstract system shown in Figure 2.3 there is only a single target location—line 10 in `check` procedure shown in Figure 2.2(a). Furthermore, the abstract system only contains one abstract trace leading from the start of the program to the target location. The abstract trace Π_α is a singleton set containing $\langle l_0, l_1, l_2, l_3, l_4, l_5 \rangle$.

2.5 Guided Symbolic Execution

We guide a symbolic program execution along an abstract trace set, $\Pi_\alpha := \{\pi_0, \pi_1, \ldots, \pi_{t-1}\}$, in order to construct a corresponding feasible execution path, $\Pi_s := \langle s_0, s_1, \ldots, s_n \rangle$. For an abstract trace set, the guided symbolic execution tries to generate a feasible execution path that contains program states where the program location of the most recently executed thread in the state matches a location in the abstract trace. The total number of locations in the abstract trace is $m := \sum_{\pi_i \in \Pi_\alpha} \texttt{length}(\pi_i)$ where the `length` function returns the number of locations in the abstract trace π_i. In our experience, the value of m is a lot smaller than n, $m << n$ where n is the length of the feasible execution trace corresponding to Π_α, because the abstract traces contain large control flow gaps between two locations in the abstract

```
 1: /* $backtrack := \emptyset,\ A_\alpha := \Pi_\alpha,\ s := s_0,\ trace := \langle s_0 \rangle$ */
procedure main()
 2: while $\langle s, \Pi_\alpha, trace \rangle \neq null$ do
 3:     $\langle s, \Pi_\alpha, trace \rangle :=$ guided_symbolic_execution($s, \Pi_\alpha, trace$)
 4:
procedure guided_symbolic_execution($s, \Pi_\alpha, trace$)
 5: while $\neg$(end_state($s$) or depth_bound($s$) or time_bound()) do
 6:     if goal_state($s$) then
 7:         print $trace$ exit
 8:     $\langle s', S_s \rangle :=$ get_ranked_successors($s,\ \Pi_\alpha$)
 9:     for each $s_{other} \in S_s$ do
10:         $backtrack := backtrack \cup \{\langle s_{other}, \Pi_\alpha, trace \circ s_{other} \rangle\}$
11:     if $\exists\ \pi_i \in \Pi_\alpha$, head($\pi_i$) == getCurrentLoc($s$) then
12:         $l_\alpha :=$ head($\pi_i$)   /* First element in the trace */
13:         $l'_\alpha :=$ head(tail($\pi_i$)) /* Second element in the trace */
14:         if branch($l_\alpha$) $\wedge$ ($l'_\alpha \neq$ getCurrentLoc($s'$)) then
15:             return $\langle s_0, A_\alpha :=$ refine_trace($A_\alpha, \pi_i$), $\langle s_0 \rangle \rangle$
16:         remove($\pi_i, l_\alpha$) /* This updates the $\pi_i$ reference in $\Pi_\alpha$ */
17:     $s := s',\ trace := trace \circ s'$
18: return $\langle s', \Pi_\alpha, trace \rangle \in backtrack$
```

Figure 2.4: Guided symbolic execution pseudocode.

trace. The intermediate program locations that are not part of the backward slice are also not included in the abstract system or the resulting abstract traces.

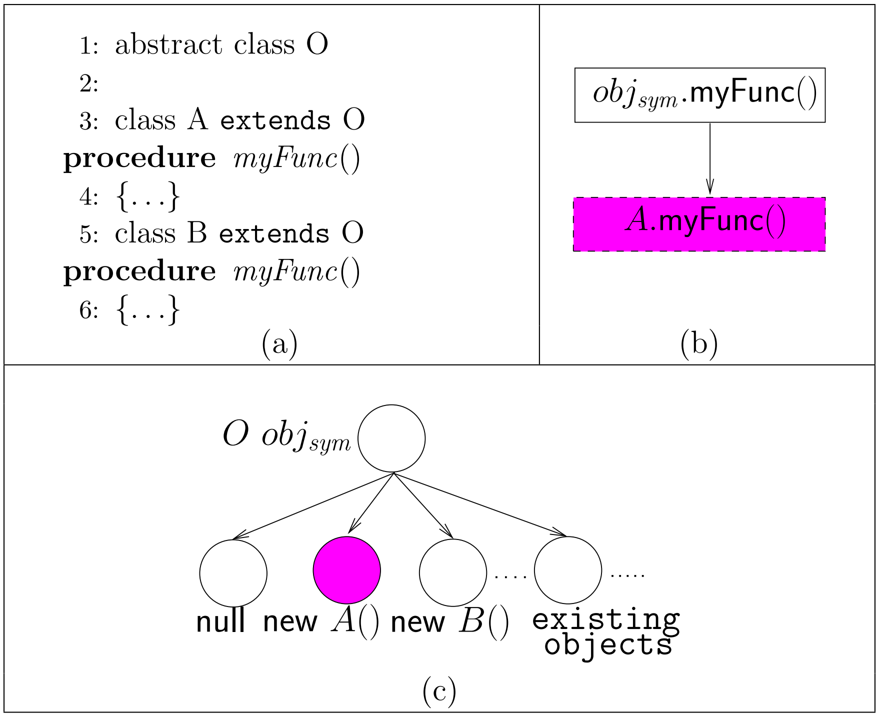
The pseudocode for the guided symbolic execution is presented in Figure 2.4. On line 1 we initialize the *backtrack* set as empty, store a copy of the input abstract trace set Π_α in A_α, set program state s to the initial program state s_0, and add s_0 to the feasible execution trace. On line 3, `main` invokes `guided_symbolic_execution` where the values of the elements in the tuple are $\langle s_0, \Pi_\alpha, \langle s_0 \rangle \rangle$. A time and depth bound are specified by the user as the termination criteria of the symbolic execution.

The guided symbolic program execution is a greedy depth-first search that picks the best immediate successor of the current state and does not consider unexplored successors until it reaches the end of a path and needs to backtrack. The search is executed along a path in the program until it reaches an end state (a state with no successors), a user-specified depth bound (line 5), a user-specified time bound (line 2), or the goal state (line 6). In the *goal state*, s, there exists a unique thread at each target location ($\forall l_i \in L_t, \exists j \in \texttt{getEnabledThreads}(s),\ \texttt{getLoc}(s, j) == l_i$). If the state s is the goal state (line 6) then the feasible execution *trace* is printed before exiting the search. In this scenario we are successfully able to find a corresponding execution trace that includes each location in the abstract trace set. The guided symbolic execution technique is guaranteed to terminate even if the goal state is not reachable because it is depth and time bounded.

States are assigned a heuristic rank in order to intelligently guide the program execution. The `get_ranked_successors` function returns a tuple $\langle s', S_s \rangle$ on line 8 in Figure 2.4 where s' is the best ranked successor of state s while all the other successors are in set S_s. Each $s_{other} \in S_s$ is added to the backtrack set with the abstract trace set and the feasible execution trace (lines 9 and 10). The feasible execution trace added to the backtrack set with s_{other} denotes a feasible execution path from s_0 to s_{other}. The best-ranked state s' is assigned as the current state and the feasible execution trace is updated by concatenating s' to it using the $\circ$ function (line 17). The $\circ$ function returns the concatenation of two input lists.

In order to match a location in the abstract trace set to a program state, the algorithm checks whether the program location of the most recently executed thread in state s matches the first location in an abstract trace, $\pi_i \in \Pi_\alpha$ (line 11). The `head` function returns the first element of the input abstract trace. The `tail` function returns the input abstract trace without its head. Location l_α is the first location in π_i while l'_α is the immediate successor of l_α. Location l_α is removed from the abstract trace (line 16) if refinement is not needed. Removing l_α updates π_i and in turn updates Π_α. The execution now attempts to match the location of the most recently executed thread in the current state toward the next location in π_i by directing the search.

The abstract trace set on line 15 is immediately refined when the program execution is unable to reach the desired target of a conditional branch statement



```
abstract class O

class A extends O
procedure myFunc()
{...}
class B extends O
procedure myFunc()
{...}
```

Figure 2.5: Ranking data non-determinism for complex data structures. (a) Classes A and B inherit from class O. (b) Locations in an abstract trace. (c) Different non-determinism choices for obj_{sym} of type O.

that contains a global variable in its predicate. The refinement is performed on the abstract trace set A_α (a copy of the original unmodified abstract trace set Π_α). After the refinement the search is restarted from the initial program state s_0 and the updated abstract trace set A_α. The details on the refinement process are given in Section 2.6.

The `get_ranked_successors`(s, Π_α) in Figure 2.4 takes as input a program state s and the abstract trace set Π_α. For each successor state s'_i of s we compute its heuristic value using a two-tier and data ranking scheme. The two-tier ranking scheme has been described in earlier works [Rungta and Mercer, 2008, 2009b, Chapter 3, Chapter 4]. In the first-tier rank program states along execution paths that correspond to more locations from the input abstract trace set are ranked better than others [Rungta and Mercer, 2008, Chapter 3]. The second-level rank is an estimate of the distance from the program state to the next program location in any of the abstract traces in Π_α [Rungta and Mercer, 2009b, Chapter 4].

The abstract trace contains conditional branch statements where the outcome of its branch predicate determines the reachability of the input target locations. When the path constraints for both outcomes of a conditional branch with primitive symbolic data types is satisfiable, the second-level heuristic uses information from the abstract trace to assign a better rank to the state at the desired outcome of the conditional branch.

New in this work, we use the information in the abstract trace to rank data non-determinism choices generated in the symbolic execution for complex input data structures. We rank s'_i at a point of complex data non-determinism for some object obj_{sym}. If there exists in an abstract trace in Π_α a call site l where obj_{sym} is the object that invokes the procedure containing the start location l', then we prefer successor states where obj_{sym} is initialized to objects of type $T := \texttt{getClass}(l')$. The `getClass` function returns the class containing the program location l'. The $h_3(s'_i) := 0$ if obj_{sym} points to an object of type T; otherwise, $h_3(s'_i) := 1$.

In Figure 2.5(a), two classes A and B inherit from the abstract base class O and implement the `myFunc` method. Figure 2.5(b) is an abstract trace where obj_{sym} is a symbolic object of type O that invokes the `myFunc` method in class A. Figure 2.5(c) shows the non-deterministic choices: (1) null, (2) new instance of class A or B, and (3) existing objects of type A and B to account for aliasing [Khurshid et al., 2003]. The information in Figure 2.5(b) indicates that the obj_{sym}.`myFunc` call needs to invoke the `myFunc` method in class A. This allows us to pick a state where the complex data structure is of type A. Information in the abstract trace about types of the objects required to reach target locations allows us to guide the symbolic execution to the target locations.

```
procedure refine_trace($A_\alpha, \pi_i$)
1: $l_{branch} := \texttt{head}(\pi_i)$
2: $L_v := \{l_v \mid \texttt{defines}(l_v, \mathsf{global\_vars}(l_{branch}))\}$
3: Update the $\mathcal{A}$ /* generate backward slices for $l_v \in V$ and synchronization
   locs(Section 2.4.2) */
4: $\pi_v := \mathsf{get\_abstract\_trace}(L_v)$
5: $\pi_{pre} := \langle l_0, \ldots, l_k \rangle$ such that $\exists \langle l_0 \ldots l_k \rangle \circ \pi_i \in A_\alpha$
6: if $\exists l_a \in \pi_{pre}, l_b \in \pi_v, \texttt{same\_lock}(l_a, l_b)$ then
7:     $\pi_v := \pi_v \circ l'_b$ where $\texttt{releaseLock}(l'_b, l_b)$
8: $\pi_{new} := \pi_v \circ \pi_{pre}$
9: $A_\alpha.\mathsf{replace\_trace}(\pi_{pre} \circ \pi_i, \pi_{new} \circ \pi_i)$
```

Figure 2.6: Refinement pseudocode.

2.6 Refinement

The refinement process is invoked when the symbolic execution cannot reach the target of the branch statement in an abstract trace, π_i. The branch predicate contains global variables that can be possibly redefined by other threads. The global variables can either be concrete or symbolic. In an effort to execute the needed branch condition a location that redefines a global variable in the branch predicate is added to the abstract trace. This allows us to account for inter-thread dependencies that affect the reachability of the target locations. We define some additional functions that are used to describe the refinement process.

- $\texttt{same\_lock}(l_a, l_b)$ returns true iff $\texttt{acqLock}(l_a) \wedge \texttt{acqLock}(l_b)$ such that l_a and l_b acquire the lock on the same object determined using a *may-alias* algorithm.
- $\texttt{get\_abstract\_trace}(L_v)$ returns an abstract trace from the start of a program to $l_v \in L_v$.
- $\Pi_\alpha.\texttt{replace\_trace}(\pi_i, \pi_j)$ substitutes π_i with π_j in Π_α.

The refinement process is shown in Figure 2.6. The first element of the abstract trace, π_i, is a branch statement as assumed on line 1 of Figure 2.6. To generate a set of program locations, L_v, on line 2 the `defines` function returns a set of program locations where global variables in the branch predicate of l_{branch} are redefined. The abstract system, $\mathcal{A}$, is updated from locations in backward slices that affect the reachability of $l_v \in L_v$ and additional synchronization locations. In essence, the process to generate locations and edges for the target location is repeated now with locations in $l_v \in L_v$. The `get_abstract_trace` returns an abstract trace in the abstract system from the start of the program to some location in L_v.

There can be many threads in the program that define a particular global variable of interest. We randomly pick a $l_v \in L_v$ and generate an abstract trace from the start of the program to l_v in $\mathcal{A}$. When there are multiple abstract traces to l_v then we, again, randomly pick an abstract trace. This refinement strategy is a heuristic that is forcing the symbolic execution to try and reach a program location where a global variable of interest is redefined and then *again* check whether the desired target location of l_{branch} is reachable.

The abstract trace set A_α is updated with a new abstract trace that contains additional locations leading to the definition of a variable used in the branch predicate. In Figure 2.6, $\pi_v := \langle l_0, \ldots, l_v \rangle$ is an abstract trace from the start of the program to location, l_v, that defines a variable in the branch predicate. The abstract trace π_{pre} is the prefix of the trace π_i in the original abstract trace set. The prefix denotes the sequence of locations from the start of the program up to, and not including, the conditional branch statement that cannot reach the desired target. This allows the refinement heuristic to add inter-thread dependence information that is ignored in the original abstraction.

In order to generate the replacement abstract trace we check the lock dependencies between π_{pre} and π_v. If π_{pre} and π_v acquire the lock on the same object (line 6), then we add the corresponding lock relinquish location to π_v (line 7). Adding the lock relinquish location ensures that if one thread acquires a lock to define a variable in the branch predicate, then after the definition another thread is not blocked trying

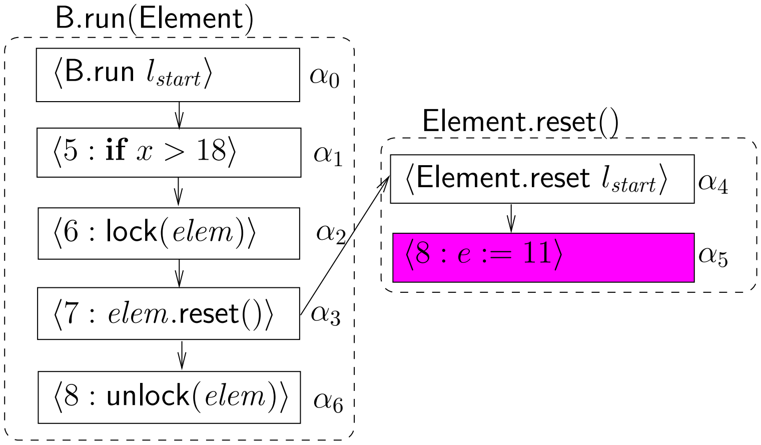


Figure 2.7: Additions to the abstract system after refinement

to acquire the same lock to reach the conditional statement. A new prefix, π_{new}, is essentially created by combining π_v and π_{pre}. This operation adds to the abstract trace the definition of a variable in the branch predicate before the conditional statement.

Finally we replace in the abstract trace set A_α the abstract trace corresponding to $\pi_{pre} \circ \pi_i$ with $\pi_{new} \circ \pi_i$ (line 9). The guided symbolic execution is now restarted from the initial program state s_0 and guided along the updated abstract trace set.

Suppose, $A_\alpha := \{\langle l_0, l_1, l_2, l_3, l_4, l_5\rangle\}$ and $\pi_i := \langle l_4, l_5\rangle$ for the example in Figure 2.2. In the runtime environment we have found a feasible execution trace that visits locations l_0 to l_3, but at the conditional branch l_4 the execution cannot reach the desired target location l_5. The refinement process shown in Figure 2.6 adds new locations and edges shown in Figure 2.7 to the abstract system in addition to the ones shown in Figure 2.3. In Figure 2.7 location α_5 defines the integer field, e, of the shared variable $elem$; $\pi_v := \langle \alpha_0, \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5\rangle$ such that the sequence leads from the start of the program to α_5 in Thread B. The prefix of π_i is $\pi_{pre} := \langle l_0, l_1, l_2, l_3\rangle$. Locations l_1 and α_2 in Figure 2.3 and Figure 2.7 respectively acquire the lock on the same object $elem$; hence, we add the lock release location to $\pi_v := \pi_v \circ \alpha_6$. Finally the guided symbolic execution is restarted from s_0 and $A_\alpha := \{\pi_v \circ \pi_{pre} \circ \pi_i\}$.

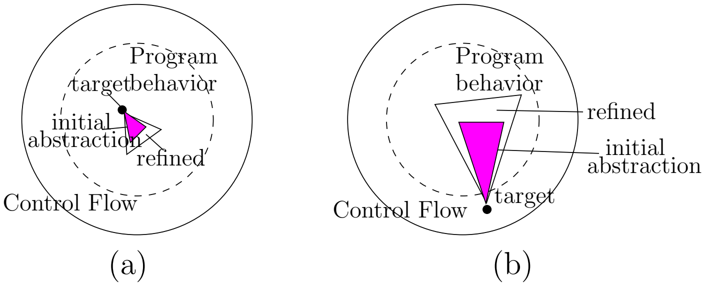


Figure 2.8: Abstraction and refinement in context of the program behavior and control flow. (a) Target is reachable. (b) Target is not reachable.

The refinement process can be invoked repeatedly for the same branch condition it is possible the same definition of the variable is added multiple times. Such a scenario allows us to handle the cases where the variable needs to be over a certain value in the branch predicate and its value is incremented by some variable or constant in the definition. The refinement strategy is in itself a heuristic. Developing and evaluating other precise refinement strategies is a work in progress.

In this illustrative example, we end up adding all the locations in Thread `B`; however, in our experience the set of locations added to the abstract set is smaller than the total number of locations in the program.

2.7 Discussion

The abstraction is an under-approximation of the control flow of the system, and the refinement adds more information as needed. In Figure 2.8(a) and (b) the outer-most circle represents the control flow of the program while the dashed circle represents the actual behaviors possible in the program. The control flow of the program is an over-approximation of all possible behaviors of the program. In Figure 2.8(a) and (b) the initial abstraction is represented by the shaded triangle. The initial abstraction generated from the backward slice is an under-approximation of the control flow of the system. It attempts to carve out parts of the control flow relevant in checking the feasibility of the target locations.

Consider the two possible cases. In the first case, the target location, initial abstraction, and the refined slice are all contained within the realm of the program behavior in Figure 2.8(a). When the symbolic execution is unable to reach the target location based on the information in the initial abstraction, the abstraction is refined (represented by triangle enclosing the shaded triangle). More information is added to the original under-approximation. Since the target is contained within the reachable part of the program behavior the guided symbolic execution is effective in discovering the error. In the second case shown in Figure 2.8(b), the target location is contained outside the program behavior. The approach in this work is unable to state its infeasibility since the refinement is a heuristic. In this case, the time bound is used as the termination criteria.

2.8 Experimental Results

We conduct experiments on machines with 8 GB of RAM and two Dual-core Intel Xeon EM64T processors (2.6 GHz). We use the symbolic extension of the Java PathFinder (JPF) v4.1 model checker with partial order reduction turned on [Păsăreanu et al., 2008]. The symbolic execution extension uses Choco Solver (`http:choco-solver.net`) to check the satisfiablity of the path constraints. JPF uses a modified JVM that operates on Java bytecode. This allows us to model the libraries as part of the program.

We present an empirical study on five multi-threaded Java programs. In Table 2.1, for each program we show the type of error, source lines of code (SLOC), total time taken in seconds to generate the set of abstract traces (Time), total number of abstract trace sets tested (Traces Sets), and total memory used (Memory). To pick the initial abstract trace sets we choose sets that contain traces with the smallest number of call sites leading from the start of the program to each target location.

	Error Type	SLOC	Time (secs)	Trace Sets	Memory MB
Reorder	Reachability	44	0.28	5	1.93 MB
Airline	Reachability	31	0.30	3	1.58 MB
VecDeadlock0	Deadlock	7267	1.21	5	38 MB
VecDeadlock1	Deadlock	7169	0.98	17	38 MB
VecRace	Race	7151	0.92	8	39 MB

Table 2.1: Information on models and abstract trace generation.

For the programs used in this empirical study, we were able to discover the goal state with the initial abstract trace sets.

The guided symbolic execution trials for the different abstract trace sets reported in Table 2.1 are launched in parallel on different computation nodes since each trial is completely independent of the other trials. When a feasible execution trace is generated along an abstract trace set, we terminate the other trials. We present the total number of states generated, total time taken, and total memory used in the trial that generates a feasible execution trace corresponding to the abstract trace set in Table 2.2. We also show the length of the initial trace ($\sum_{\pi_i \in \Pi_\alpha} |\pi_i|$) and total number of refinements performed on the abstract trace; Π_α is the input abstract trace set. The parameters with the program names indicate the thread configuration of a particular program. Each parameter represents the total number of symmetric threads in the system.

The `Reorder` and the `Airline` model are benchmarked examples and have user-defined reachability properties. These models do not contain any data non-determinism. The results of the models are used to demonstrate the effectiveness of the abstraction refinement technique in guiding a concrete program execution to an error state. The `Airline` model required a larger number of refinements because the reachability of the target location depends on the value of a global counter that is modified by different threads. The refinement adds the location where the global variable is defined at each iteration.

We created C# programs for the Reorder and Airline models to evaluate on CHESS, a stateless concurrency testing tool [Musuvathi and Qadeer, 2008, 2007]. The models with the thread configuration presented in this paper, CHESS was unable to find an error within a time bound of one hour. CHESS uses an iterative context-bound approach that bounds the number of preemptions along a certain path in order to reach the error faster. Note that the correct number of preemption points required to find the error was provided as input. Random walk and randomized depth-first search in JPF have also been shown ineffective to find errors in these models [Rungta and Mercer, 2007a, Appendix B]. A detailed empirical analysis is shown in [Rungta and Mercer, 2009a, Chapter 5].

VecDeadlock0, VecDeadlock1, and VecRace are examples that use the JDK 1.4 synchronized Vector library in accordance with the documentation. We use Jlint to automatically generate warnings on possible deadlocks and race-conditions in the synchronized Vector library [Artho and Biere, 2001]. Each model has two symbolic variables whose specific values in addition to certain thread schedules are required to manifest errors in the Vector library. Exhaustive symbolic execution using a depth-first search is unable to discover the errors in these models within a time bound of one hour. In the VecDeadlock0, the abstraction-guided symbolic execution only generates 1370 states and takes about 4.5 seconds to find the deadlock in the program. Similarly in the VecDeadlock1 and VecRace programs, the guided symbolic execution only generates a few thousand states before generating a concrete trace to the error. Using the information from the abstract trace set, the heuristic to rank the non-determinism of complex data structures allows us to achieve this dramatic improvement in error discovery over exhaustive symbolic execution.

Model	States	Time	Memory	Total trace	Total
		secs	MB	Length	Refinements
Reorder (9,1)	205	1.67	7MB	13	1
Reorder (10,1)	236	1.67	7MB	13	1
Airline (15,3)	1210	3.23	5MB	3	13
Airline (20,2)	3279	7.46	6MB	3	19
Airline (20,1)	3609	7.46	6MB	3	20
VecDealock0	1370	4.56	66MB	14	1
VecDeadlock1	2948	6.89	69MB	15	2
VecRace	3120	7.98	65MB	12	1

Table 2.2: Effort in error discovery and abstract trace statistics.

2.9 Related Work

Recent work by Tomb *et al.* uses symbolic execution to generate concrete paths to null pointer exceptions at an inter-procedural level in sequential programs [Tomb et al., 2007]. In contrast, concolic testing executes the program with random concrete values in conjunction with symbolic execution to collect the path constraints over input data values [Sen and Agha, 2007, Sen et al., 2005]. The cost of constraint solving in concolic testing to achieve full path coverage in a concurrent system is extremely high. The techniques presented in this work are complementary to concolic testing. The techniques can also be used to efficiently guide concolic testing.

Recent work shows that guiding the concrete program execution along a sequence of manually generated program locations relevant in verifying the feasibility of the target location dramatically lowers the time taken to reach the target location [Rungta and Mercer, 2008, Chapter 3]. The manual aspect of generating relevant program locations is tedious and sometimes intractable.

Race-directed random testing of concurrent programs uses the output of imprecise dynamic analysis tools and randomly drives threads to the input locations [Sen, 2008]. The work in [Rungta and Mercer, 2008, Chapter 3] shows that guiding the search through key locations relevant in determining the target locations yields significantly better error discovery rates. Dynamic analysis tools such as ConTest use heuristics to randomly add perturbations in the thread schedules [Eytani et al., 2007]. The results are similar to those obtained with just a stateless random search and it is not very effective in error discovery. Chess is a concurrency testing tool that systematically explores thread schedules in $C\#$ programs and supports iterative context bounding [Musuvathi and Qadeer, 2007].

Model checking is a formal approach for systematically exploring all possible behaviors of a concurrent software system [Ball and Rajamani, 2001, Godefroid, 1997, Holzmann, 2003, Visser et al., 2003]. The state space explosion problem renders it intractable in verifying medium to large-sized programs. Conservative abstractions are applied to high-level programming languages [Ball and Rajamani, 2001, Henzinger et al., 2003] in order to verify programs. The abstraction is iteratively refined if it generates an infeasible counter-example to an error state. Counter-example guided abstraction refinement techniques are successful in verifying sequential programs; however, they are not effective for testing concurrent programs.

Related works in hardware verification guide the simulation of the concrete model using an abstract model of boolean variables that represent the transition relation [Nanshi and Somenzi, 2006, Paula and Hu, 2007]; however, these works are limited to verifying circuit designs and boolean programs. The techniques cannot be extended to verify complex concurrent software systems. Another area of related work is the use of abstract databases and heuristics that are used to guide the searches in planning problems [Edelkamp, 2001].

2.10 Conclusions and Future Work

In this work we present an abstraction-guided symbolic execution technique that efficiently detects errors caused by thread schedules and data values in concurrent programs. Using backward slices for the input target locations the technique automatically generates an abstract system with relevant locations in checking the reachability of the target locations. The backward slices only consider sequential execution along the control flow of the program. The symbolic execution is guided along traces in the abstract system to generate a corresponding feasible execution path to the target locations. When the symbolic execution is unable to make progress we refine the abstraction by adding locations to handle inter-thread dependencies.

In the case when we are unable to discover a feasible execution path, we want to design a probabilistic measure to estimate the likelihood of the reachability of the target locations as future work. Another avenue of future work consists of studying more precise refinement techniques based on compositional symbolic execution [Anand et al., 2008].

Chapter 3

Meta-heuristic for Concurrent Programs

This chapter was published as:

N. Rungta and E. G. Mercer, "A Meta Heuristic for Effectively Detecting Concurrency Errors", in *Proceedings of Haifa Verification Conference (HVC)*, Haifa, Israel, November 2008.

Abstract

Mainstream programming is migrating to concurrent architectures to improve performance and facilitate more complex computation. The state of the art static analysis tools for detecting concurrency errors are imprecise, generate a large number of false error warnings, and require manual verification of each warning. In this paper we present a meta heuristic to help reduce the manual effort required in the verification of warnings generated by static analysis tools. We manually generate a small sequence of program locations that represent points of interest in checking the feasibility of a particular static analysis warning; then we use a meta heuristic to automatically control scheduling decisions in a model checker to guide the program along the input sequence to test the feasibility of the warning. The meta heuristic guides a greedy depth-first search based on a two-tier ranking system where the first tier considers the number of program locations already observed from the input sequence, and the second tier considers the perceived closeness to the next location in the input sequence. The error traces generated by this technique are real and require no further

manual verification. We show the effectiveness of our approach by detecting feasible concurrency errors in benchmarked concurrent programs and the JDK 1.4 concurrent libraries based on warnings generated by the Jlint static analysis tool.

..

3.1 Introduction

The ubiquity of multi-core Intel and AMD processors is prompting a shift in the programming paradigm from inherently sequential programs to concurrent programs to better utilize the computation power of the processors. Although parallel programming is well studied in academia, research, and a few specialized problem domains, it is not a paradigm commonly known in mainstream programming. As a result, there are few, if any, tools available to programmers to help them test and analyze concurrent programs for correctness.

Static analysis tools that analyze the source of the program for detecting concurrency errors are imprecise and incomplete [Artho and Biere, 2001, Engler and Ashcraft, 2003, Flanagan et al., 2002, Hovemeyer and Pugh, 2004]. Static analysis techniques are not always useful as they report warnings about errors that *may* exist in the program. The programmer has to manually verify the feasibility of the warning by reasoning about input values, thread schedules, and branch conditions required to manifest the error along a real execution path in the program. Such manual verification is not tractable in mainstream software development because of the complexity and the cost associated with such an activity.

Model checking in contrast to static analysis is a precise, sound, and complete analysis technique that reports only feasible errors [Holzmann, 2003, Visser et al., 2000a]. It accomplishes this by exhaustively enumerating all possible behaviors (state space) of the program to check for the presence and absence of errors; however, the growing complexity of concurrent systems leads to an exponential growth in the size of state space. This state space explosion has prevented the use of model checking in mainstream test frameworks.

Directed model checking focuses its efforts in searching parts of the state space where an error is more likely to exist in order to partially mitigate the state space explosion problem [Edelkamp et al., 2001b, Groce and Visser, 2002b, Rungta and

Mercer, 2005, 2006, 2009b]. Directed model checking uses heuristic values and path-cost to rank the states in order of interest in a priority queue. Directed model checking uses some information about the program or the property being verified to generate heuristic values. The information is either specified by the user or computed automatically. In this work we use the imprecise static analysis warnings to detect possible defects in the program and use a precise directed search with a meta heuristic to localize real errors.

The meta heuristic presented in this paper guides the program execution in a greedy depth-first manner along an input sequence of program locations. The input sequence is a small number of locations manually generated such that they are relevant in testing the feasibility of a static analysis warning or a reachability property. The meta heuristic ranks the states based on the portion of the input sequence already observed. States that have observed a greater number of locations from the input sequence are ranked as more *interesting* compared to other states. In the case where multiple states have observed the same number of locations in the sequence, the meta heuristic uses a secondary heuristic to guide the search toward the next location in the sequence. In essence, the meta heuristic automatically controls scheduling decisions to drive the program execution along the input sequence in a greedy depth-first manner. The greedy depth-first search picks the best-ranked immediate successor of the current state and does not consider unexplored successors until it reaches the end of a path and needs to backtrack.

In this work we do not consider any non-determinism arising due to data input and only consider the non-determinism arising from thread schedules. The error traces generated by the technique are real and require no further verification; however, if the technique does not find an error we cannot prove the absence of the error. The technique is sound in error detection but not complete.

To test the validity of our meta heuristic solution in aiding the process of automatically verifying deadlocks, race conditions, and reachability properties in multi-threaded programs, we present an empirical study conducted on several benchmarked concurrent Java programs and the JDK 1.4 concurrent libraries. We use the Java PathFinder model checker (an explicit state Java byte-code model checker) to conduct the empirical study [Visser et al., 2000a]. We show that the meta heuristic is extremely effective in localizing a feasible error when given a few key locations relevant to a corresponding static analysis warning. Furthermore, the results demonstrate that the choice of the secondary heuristic has a dramatic effect on the number of states generated, on average, before error discovery.

3.2 Meta heuristic

In this section we describe the input sequence to the meta heuristic, our greedy depth-first search, and the guidance strategy based on the meta heuristic.

3.2.1 Input Sequence

The input to our meta heuristic is the program, an environment that closes the program, and a sequence of locations that are relevant to checking the feasibility of the static analysis warning. The number and type of locations in the sequence can vary based on the static analysis warning being verified. For example, to test the occurrences of race-conditions, we can generate a sequence of program locations that represent a series of reads and writes on shared objects. Note that we do not manually specify which thread is required to be at a given location in the input sequence and rely on the meta heuristic to intelligently pick thread assignments.

We use the example in Figure 3.1 to demonstrate how we generate an input sequence to check the feasibility of a possible race condition from a static analysis warning. Figure 3.1 represents a portion of a program that uses the JDK 1.4

```
class raceCondition{
...
  public static void main(){
    AbstractList $l_1$ := new Vector();
    AbstractList $l_2$ := new Vector();
    AThread $t_0$ = new AThread($l_1, l_2$);
    AThread $t_1$ = new AThread($l_1, l_2$);
    $t_0$.start(); $t_1$.start();
    ...
  }
...
}
```

(a)

```
class AThread extends Thread{
  AbstractList $l_1$;
  AbstractList $l_2$;
  AThread(AbstractList $l_1$,
                AbstractList $l_2$){
    this.$l_1$ := $l_1$; this.$l_2$ := $l_2$;
  }
public void run(){
    ...
     if some_condition then
       $l_2$.add(some_object);
    ...
    $l_1$.equals($l_2$);
    ...
   }
}
```

(b)

```
class Vector extends
                 AbstractList{
...
  public synchronized boolean equals
                        (Object $o$){
    super.equals($o$);
  }
...
  public synchronized boolean add
                        (Object $o$){
    $modCnt++$;
    ensureCapacityHelper($cnt + 1$);
    elementData[$cnt++$] = $o$;
    return true;
  }
...
}
```

(c)

```
 class AbstractList
                implements List{
 public boolean equals(Object $o$){
   if $o == this$ then
     return true;
   if ¬($o$ instanceof List) then
     return false;
   ListIterator $e_1$ := ListIterator();
   ListIterator $e_2$ :=
               (List $o$).listIterator();
   while $e_1$.hasNext() and
                 $e_2$.hasNext() do
     Object $o_1$ := $e_1$.next();
     Object $o_2$ := $e_2$.next();
     if¬($o_1$ == null ? $o_2$ == null :
                 $o_1$.equals($o_2$)) then
        return false;
   return ¬($e_1$.hasNext() ||
                $e_2$.hasNext())
 }
}
```

(d)

Figure 3.1: Possible race-condition in the JDK 1.4 concurrent library.

concurrent public library. The raceCondition class in Figure 3.1(a) initializes two AbstractList data structures, l_1 and l_2, using the synchronized Vector sub-class implementation. Two threads of type AThread, t_0 and t_1, are initialized such that both threads can concurrently access and modify the data structures, l_1 and l_2. Finally, main invokes the run function of Figure 3.1(b) on the two threads. The threads go through a sequence of events, including operations on l_1 and l_2 in Figure 3.1(b). Specifically, an add operation is performed on list l_2 when a certain condition is satisfied; the add is then followed by an operation that checks whether l_1 equals l_2. The add operation in the Vector class, Figure 3.1(c), first acquires a lock on its own Vector instance and then adds the input element to the instance. The equals function in the same class, however, acquires the lock on its own instance and invokes the equals function of its parent class which is AbstractList shown in Figure 3.1(d).

The Jlint static analysis tool issues a series of warnings about potential concurrency errors in the concurrent JDK library when we analyze the program shown in Figure 3.1 [Artho and Biere, 2001]. The Jlint warnings for the equals function in the AbstractList class in Figure 3.1(d) are on the Iterator operations (lines $8 - 14$ and lines $18 - 19$). The warnings state that the Iterator operations are not synchronized. As the program uses a synchronized Vector sub-class of the AbstractList (in accordance with the specified usage documentation), the user may be tempted to believe that the warnings are spurious. Furthermore, people most often ignore the warnings in libraries since they assume the libraries to be error-free. To check the feasibility of the possible race condition reported by Jlint for the example in Figure 3.1 we need a thread iterating over the list, l_2, in the equals function of AbstractList while another thread calls the add function. A potential input sequence of locations to test the feasibility of the warning is as follows:

1. Get the ListIterator, e_2 at lines $9 - 10$ in Figure 3.1(d).
2. Check e_2 hasNext() at line 12 in Figure 3.1(d).

```
/* backtrack := ∅, visited := ∅ */
procedure gdf_search(⟨$s$, $locs$, $h_{val}$⟩)
visited := visited ∪ {$s$}
while $s \neq$ null do
  if error($s$) then
    report error statistics
    exit
  ⟨$s$, $locs$, $h_{val}$⟩ := choose_best_successor(⟨$s$, $locs$, $h_{val}$⟩)
  if $s ==$ null then
    ⟨$s$, $locs$, $h_{val}$⟩ := get_backtrack_state()
```

Figure 3.2: Pseudocode for the greedy depth-first search.

3. Add `some_object` to l_2 at line 11 in Figure 3.1(b).

4. Call e_2.`next`() at line 14 in Figure 3.1(d).

The same approach can be applied to generate input sequences for different warnings. Classic lockset analysis techniques detect potential deadlocks in multi-threaded programs caused due to cyclic lock dependencies [Engler and Ashcraft, 2003, Williams et al., 2005]. For example, it detects a cyclic dependency in the series of lock acquisitions $l_0(A) \rightarrow l_1(B)$ and $l_9(B) \rightarrow l_{18}(A)$, where A and B are the locks acquired at different program locations specified by l_n. To generate an input sequence that checks the feasibility of the possible deadlock we can generate a sequence of locations: $l_0 \rightarrow l_9 \rightarrow l_1 \rightarrow l_{18}$. A larger set of concurrency error patterns are described by Farchi *et. al* in [Farchi et al., 2003]. Understanding and recognizing the concurrent error patterns can be helpful in generating location sequences to detect particular errors.

In general, providing as much relevant information as possible in the sequence enables the meta heuristic to be more effective in defect detection; however, only 2–3 key locations were required to find errors in most of the models in our study. Any program location that we think affects the potential error can be added to the sequence. For example, if there is a data definition in the program that affects the variables in the predicate, `some_condition`, of the branch statement shown on line 10 in Figure 3.1(b), then we can add the program location of the data definition to the sequence. Similarly we can generate input sequences to check reachability properties such as NULL pointer exceptions and assertion violations in multi-threaded programs.

3.2.2 Greedy depth-first search

In this subsection we describe a greedy depth-first search that lends itself naturally in directing the search using the meta heuristic along a particular path (the input sequence of locations). The greedy depth-first search mimics a test-like paradigm for multi-threaded programs. The meta heuristic can be also used with bounded priority-queue based best-first searches with comparable results.

The pseudocode for the greedy depth-first search is presented in Figure 3.2. The input to gdf_search is a tuple with the initial state of the program (s), the sequence of locations ($locs$), and the initial secondary heuristic value (h_{val}). In a loop we guide the execution as long as the current state, s, has successors (lines $3-9$). At every state we check whether the state, s, satisfies the error condition (line 4). If an error is detected, then we report the error details and exit the search; otherwise, we continue to guide the search. The choose_best_successor function only considers the immediate successors of s and assigns to the current state the best-ranked successor of s (line 7). When the search reaches a state with no immediate successors, the technique requests a backtrack state as shown on lines $8-9$ in Figure 3.2. The details of choose_best_successor and get_backtrack_state are provided in Figure 3.4 and Figure 3.5 respectively.

Figure 3.3(a) demonstrates the greedy depth-first search using a simple example. The choose_best_successor function ranks c_0, c_1, and c_2 (enclosed in a dashed box) to choose the best successor of b_0 in Figure 3.3(a). The shaded state c_2 is ranked as the best successor of b_0. When the search reaches state d_2 that does not have any successors, the search backtracks to one of the unshaded states (e.g., b_1, b_2, c_0, c_1, d_0, or d_2). We bound the number of unshaded states (backtrack states) saved during the

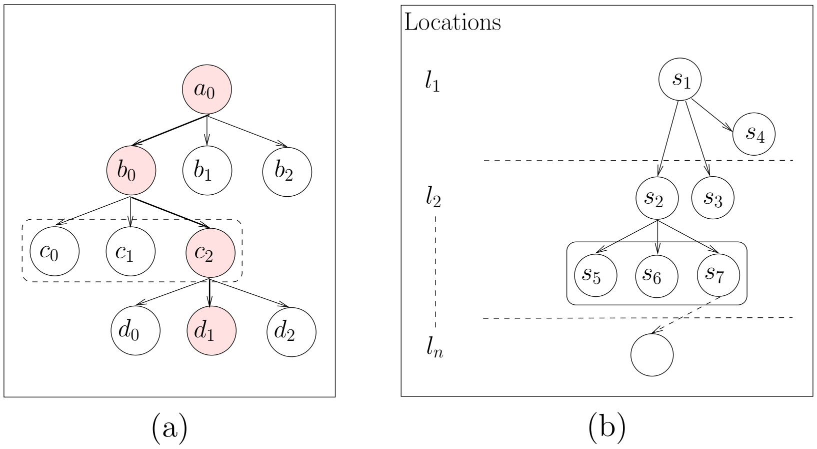


Figure 3.3: Guidance (a) Greedy depth-first search (b) Two-level ranking scheme

search. Bounding the backtrack states makes our technique incomplete; although, the bounding is not a limitation because obtaining a complete coverage of the programs we are considering is not possible.

3.2.3 Guidance Strategy

The meta heuristic uses a two-tier ranking scheme as the guidance strategy. The states are first assigned a rank based on the number of locations in the input sequence that have been encountered along the current execution path. The meta heuristic then uses a secondary heuristic to rank states that observed the same number of locations in the sequence. The secondary heuristic is essentially used to guide the search toward the next location in the input sequence.

In Figure 3.4 we present the pseudocode to choose the best successor of a given state. The input to the function is a tuple $\langle s, locs, h_{val} \rangle$ where s is a program state, *locs* is a sequence of locations, and h_{val} is the heuristic value of s generated by the secondary heuristic function. We evaluate each successor of s, s', and process s' if it is not found in the `visited` set (line $2-3$). To process s' we add it to the `visited` set (line 4), copy the sequence of locations *locs* into a new sequence of locations *locs*$'$ (line 5), and compute the secondary heuristic value for s' (line 6). If s' observes

```
1: /* $mStates := \emptyset$, $hStates := \emptyset$, $min\_h_{val} := \infty$ */
procedure choose_best_successor($\langle s, locs, h_{val}\rangle$)
2: for each $s' \in$ successors($s$) do
3:    if $\neg visited$.contains($s'$) then
4:       $visited$.add_state($s'$)
5:       $locs' := locs$ /* Make copy of locs */
6:       $h'_{val}$ = get_h_value($s'$)
7:       if $s'$.current_loc() == $locs$.top() then
8:          $mStates$ := next_state_to_explore($mStates$, $\langle s', locs'$.pop()$, h'_{val}\rangle$)
9:       else
10:         $hStates$ := next_state_to_explore($hStates$, $\langle s', locs, h'_{val}\rangle$)
11:      $backtrack$.add_state($\langle s', locs', h'_{val}\rangle$)
12: if $mStates \neq \emptyset$ then
13:    $\langle s, locs, h_{val}\rangle$ := get_random_element($mStates$)
14: else
15:    $\langle s, locs, h_{val}\rangle$ := get_random_element($hStates$)
16: $backtrack$.remove_state($\langle s, locs, h_{val}\rangle$)
17: bound_size($backtrack$)
18: return $\langle s, locs, h_{val}\rangle$

procedure next_state_to_explore($states$, $\langle s, locs, h_{val}\rangle$)
1: if $states == \emptyset$ or $h_{val} == min\_h_{val}$ then
2:    $states$.add_state($\langle s, locs, h_{val}\rangle$)
3: else if $h_{val} < min\_h_{val}$ then
4:    $states$.clear()
5:    $states$.add_state($\langle s, locs, h_{val}\rangle$)
6:    $min\_h_{val} := h_{val}$
7: return $states$
```

Figure 3.4: Two-tier ranking scheme for the meta heuristic.

an additional location from the sequence (line 7), then we update the mStates set (line 8); otherwise, we update the hStates set (line 10). An element from the $locs'$ is removed on line 8 to indicate s' has observed an additional location. We invoke the next_state_to_explore function with the mStates or the hStates set and the tuple containing s'. The best successor is picked from mStates if it is non-empty; else, it is picked from the hStates set. The algorithm prefers states in the mStates set because they have observed an additional location compared to their parent. All other successor states are added to the backtrack set (lines $12 - 18$).

The next_state_to_explore function in Figure 3.4 uses the secondary heuristic values (h_{val}) to add states to the mStates and hStates sets. Recall that the

```
procedure get_backtrack_state()
if $backtrack == \emptyset$ then
    return $\langle$null$, \infty, \infty\rangle$
else
    $x$ := pick_backtrack_meta_level()
    $b\_points$ := get_states($backtrack, x$)
    $b\_points$ := $b\_points \cap$ states_min_h_value($b\_points$)
    return get_random_element($b\_points$)
```

Figure 3.5: Stochastic backtracking technique.

`next_state_to_explore` is invoked with either the `mStates` set or `hStates` set which is mapped to the formal parameter `states`. When the `states` set is empty or the h_{val} is equal to the minimum heuristic value (min_h_{val}) then the algorithm simply adds the tuple with the successor state to the `states` set. If, however, the h_{val} is less than the minimum heuristic value then the algorithm clears the `states` set, adds the tuple with the successor state to `states`, and sets the value of min_h_{val} to h_{val}. Finally, the function returns the `states` set.

We use Figure 3.3(b) to demonstrate the two-tier ranking scheme. In Figure 3.3(b) the search is guided through locations l_1 to l_n. The dashed-lines separate the states based on the number of locations from the sequence they have observed along the path from the initial state. The states at the topmost level l_1 have encountered the first program location in the sequence while states at l_2 have observed the first two program locations from the sequence, so on and so forth. In Figure 3.3(b) we see that state s_1 has three successors: s_2, s_3, and s_4. The states s_2 and s_3 observe an additional location, l_2, from the sequence compared to their parent s_1. Suppose s_2 and s_3 have the same secondary heuristic value. We add the states s_2 and s_3 to the `mStates` set to denote that a location from the sequence is observed. Suppose, the secondary heuristic value of s_4 is greater than that of s_2 and s_3; then s_4 is not added to the `hStates` set.

After enumerating the successors of s_1, the mStates set is non-empty so we randomly choose between s_2 and s_3 (line 13 in Figure 3.4) and return the state as the best successor. When we evaluate successors of a state that do not encounter any additional location from the sequence, for example, the successors of s_2 in Figure 3.3(b) (enclosed by the box), the states are ranked simply based on their secondary heuristic values. The best successor is then picked from the hStates set. All states other than the best successor are added to the backtrack set. We bound the size of the backtrack set to mitigate the common problem in directed model checking where saving the frontier in a priority queue consumes all memory resources.

The get_backtrack_state function in Figure 3.5 picks a backtrack point when the guided test reaches the end of a path. Backtracking allows the meta heuristic to pick a different set of threads when it is unable to find an error along the initial sequence of thread schedules. As shown in Figure 3.5, if the backtrack set is empty, then the function returns null as the next state (lines $1-2$); otherwise, the function probabilistically picks a meta level, x, between 1 and n where n is the number of locations in the sequence. The states that have observed one program location from the sequence are at meta level one. We then get all the states at meta level x and return the state with the minimum secondary heuristic value among the states at that meta level. The stochastic element of picking backtrack points enables the search to avoid getting stuck in a local minima.

3.3 Empirical Study

The empirical study in this paper is designed to evaluate the effectiveness of the meta heuristic in detecting concurrency errors in multi-threaded Java programs.

3.3.1 Study Design

We conduct the experiments on machines with 8 GB of RAM and two Dual-core Intel Xeon EM64T processors (2.6 GHz). We run 100 trials of greedy depth-first search and randomized depth-first search. All the trials are bounded at one hour. We execute multiple trials of the greedy depth-first search since all ties in heuristic values are brokenly randomly and there is a stochastic element in picking backtrack points. An extensive study shows that randomly breaking ties in heuristic values helps in overcoming the limitations (and benefits) of default search order in directed search techniques [Rungta and Mercer, 2007b, Appendix A]. We pick the time bound and number of trials to be consistent with other recent empirical studies [Dwyer et al., 2006, 2007, Rungta and Mercer, 2007a, Appendix B]. Since each trial is completely independent of the other trials we use a super computing cluster of 618 nodes to distribute the trials on various nodes and quickly generate the results.[1] We use the Java Pathfinder (JPF) v4.0 Java byte-code model checker with partial order reduction turned on to run the experiments [Visser et al., 2000a]. In the greedy depth-first search trials we save at most 100,000 backtrack states.

We use six unique multi-threaded Java programs in this study to evaluate the effectiveness of the meta heuristic in checking whether the input sequence leads to an error. Three programs are from the benchmark suite of multi-threaded Java programs gathered from academia, IBM Research Lab in Haifa, and classical concurrency errors described in literature [Dwyer et al., 2006]. We pick these three artifacts from the benchmark suite because the threads in these programs can be systematically manipulated to create configurations of the model where randomized depth-first search is unable to find errors in the models [Rungta and Mercer, 2007a, Appendix B]. These models also exhibit different concurrency error patterns described by Farchi *et. al*

[1] We thank Mary and Ira Lou Fulton for their generous donations to the BYU Supercomputing laboratory.

in [Farchi et al., 2003]. The other three examples are programs that use the JDK 1.4 library in accordance with the documentation. Figure 3.1 is one such program that appears as `AbsList` in our results. We use Jlint on these models to automatically generate warnings on possible concurrency errors in the JDK 1.4 library and then manually generate the input sequences. The name, type of model, number of locations in the input sequence, and source lines of code (SLOC) for the models are as follows:

- **TwoStage**: Benchmark, Num of locs: 2, SLOC: 52
- **Reorder**: Benchmark, Num of locs: 2, SLOC: 44
- **Wronglock**: Benchmark, Num of locs: 3, SLOC: 38
- **AbsList**: Real, Num of locs: 6, Race-condition in the `AbstractList` class using the synchronized `Vector` sub-class Figure 3.1. SLOC: 7267
- **AryList**: Real, Num of locs: 6, Race-condition in the `ArrayList` class using the synchronized `List` implementation. SLOC: 7169
- **Deadlock**: Real, Num of locs: 6, Deadlock in the `Vector` and `Hashtable` classes due to a circular data dependency [Williams et al., 2005]. SLOC: 7151

3.3.2 Error Discovery

In Table 3.1 we compare the error densities of randomized depth-first search (`Random DFS`) to the meta heuristic using a greedy depth-first search. The error density which is a dependent variable in this study is defined as the probability of a technique finding an error in the program. To compute this probability we use the ratio of the number of error discovering trials over the total number of trials executed for a given model and technique. A technique that generates an error density of 1.00 is termed effective in error discovery while a technique that generates an error density of 0.00 is termed ineffective for error discovery.

Subject	Total Threads	Random DFS	Meta Heuristic		
			PFSM	Rand	Prefer Threads
TwoStage(7,1)	9	0.41	1.00	1.00	1.00
TwoStage(8,1)	10	0.04	1.00	1.00	1.00
TwoStage(10,1)	12	0.00	1.00	1.00	1.00
Reorder(9,1)	11	0.06	1.00	1.00	1.00
Reorder(10,1)	12	0.00	1.00	1.00	1.00
Wronglock(1,20)	22	0.28	1.00	1.00	1.00
AbsList(1,7)	9	0.01	1.00	0.37	0.00
AbsList(1,8)	10	0.00	1.00	0.08	0.00
Deadlock(1,9)	11	0.00	1.00	1.00	1.00
Deadlock(1,10)	12	0.00	1.00	1.00	1.00
AryList(1,5)	7	0.81	1.00	1.00	1.00
AryList(1,8)	10	0.00	1.00	1.00	0.01
AryList(1,9)	11	0.00	1.00	1.00	0.00
AryList(1,10)	12	0.00	1.00	1.00	0.00

Table 3.1: Error density of the models with different search techniques.

We test three different secondary heuristics which is an independent variable to study the effect of the underlying heuristic on the effectiveness of the meta heuristic: (1) The polymorphic distance heuristic (`PFSM`) computes the distance between a target program location and the current program location on the control flow representation of the program. The heuristic rank based on the distance estimate lends itself naturally to guiding the search toward the next location in the sequence [Rungta and Mercer, 2009b, Chapter 4]. (2) The random heuristic (`Rand`) always returns a random value as the heuristic estimate. It serves as a baseline measure to test the effectiveness of guiding along the input sequence in the absence of any secondary guidance. (3) The prefer-thread heuristic (`Prefer Threads`) assigns a low heuristic value to a set of user-specified threads [Groce and Visser, 2002b]. For example, if there are five total threads in a program then the user can specify to prefer the execution of certain threads over others when making scheduling choices.

The results in Table 3.1 indicate that the meta heuristic, overall, has a higher error discovery rate compared to randomized depth-first search. In the `TwoStage`

Subject	PFSM Heuristic			Random Heuristic			Prefer-thread Heuristic		
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
TwoStage(7,1)	209	213	217	40851	130839	409156	414187	2206109	4813016
TwoStage(8,1)	246	250	255	49682	217637	502762	609085	4436444	10025314
TwoStage(10,1)	329	333	340	52794	314590	827830	2635251	6690008	8771151
Wronglock(1,10)	804	3526	12542	73	7082	22418	560	120305	675987
Wronglock(1,20)	2445	21391	175708	67	24479	242418	1900	3827020	15112994
Reorder(5,1)	106	109	112	1803	5597	10408	259	977	2402
Reorder(8,1)	193	197	202	17474	36332	65733	523	3110	13536
Reorder(10,1)	266	271	277	28748	67958	110335	771	5136	16492
AryList(1,10)	1764	14044	55241	3652	15972	63206	-	-	-
AbsList(1,10)	1382	1382	1382	10497302	10497302	10497302	-	-	-

Table 3.2: Comparison of the heuristics when used with the meta heuristic.

example the error density drops from 0.41 to 0.00 when going from the configuration of `TwoStage(7,1)` to the `TwoStage(10,1)` configuration. A similar pattern is observed in the `Reorder` model where the error density goes from 0.06 to 0.0; in the `AryList` model the error density drops from a respectable 0.81 to 0.00. For all these models, the meta heuristic using the polymorphic distance heuristic finds an error in every single trial as indicated by the error density of 1.00. In some cases, even when we use the random heuristic as the secondary heuristic, the greedy depth-first search outperforms the randomized depth-first search.

The `AbsList`, `AryList`, and `Deadlock` models represent real errors in the JDK 1.4 concurrent library. The `AbsList` model contains the portion of code shown in Figure 3.1. In addition to the locations shown in Section 2.1 we manually add other data definition locations that are relevant in reaching the locations shown in Section 2.1. We use the meta heuristic to successfully generate a concrete error trace for the possible race condition reported by Jlint. The counter-example shows that the race-condition is caused because the `equals` method in Figure 3.1(c) never acquires a lock on the input parameter. This missing lock allows another thread to modify the list (by adding an object on line 11 in Figure 3.1(b)) while the thread is iterating over the list in the `equals` method. To our knowledge, this is the first report of the particular race condition in the JDK 1.4 library. It can be argued that the application using the library is incorrect and changing the comparison on line 13 of Figure 3.1(b) to l_2.`equals`(l_1) can fix the error; however, we term it as a bug in the library because the usage of the library is in accordance with the documentation.

Table 3.2 reports the minimum, average, and maximum number of states generated in the error discovering trials of the meta heuristic using the three secondary heuristics. The entries in Table 3.2 marked "-" indicate that the technique was unable to find an error in 100 independent greedy depth-first search trials that are time-bounded at one hour. In the `TwoStage`, `Reorder`, `AryList`, `AbsList` subjects, the minimum, average, and maximum states generated by the PFSM heuristic is perceptibly less than the random and prefer-thread heuristics. Consider the `Twostage(7,1)` model where, on average, the PFSM heuristic only generates 213 states while the random heuristic and prefer-thread heuristic generate $130,839$ and $2,206,109$ states respectively, on average, before error discovery. In the `AbsList(1,10)` model the PFSM heuristic finds the error every time by exploring a mere 1382 states. In contrast, from a total of 100 trials with the random heuristic only a single trial finds the error after exploring over a million states, while the prefer-thread heuristic is unable to find the error in the 100 trials. `Wronglock` is the only model where the minimum number of states generated by the random heuristic is less than the PFSM heuristic. This example shows that it is possible for the random heuristic to get just lucky in certain models. The results in Table 3.2 demonstrate that a better underlying secondary heuristic helps the meta heuristic generate fewer states before error discovery. The trends observed in Table 3.2 are also observed in total time taken before error discovery, total memory used, and length of counter-example.

3.3.3 Effect of the Sequence Length

We vary the number of key locations in the input sequence provided to the meta heuristic to study the effect of the number of locations on the performance of the meta

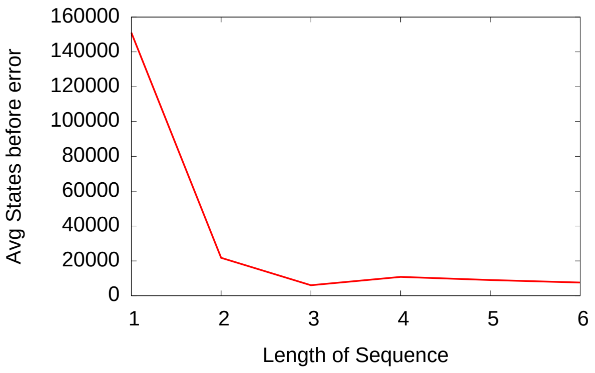


Figure 3.6: Effect of varying the number of locations in the sequence in the AryLst(1,10) program to verify the race condition in the JDK1.4 concurrent library.

heuristic. In Figure 3.6 we plot the average number of states generated (the dependent variable) before error discovery while varying sequence lengths in the `AryLst` model. In Figure 3.6 there is a sharp drop in the number of states when we increase the number of key locations from one to two. A smaller decrease in the average number of states is observed between sequence lengths two and three. We observe the effects of diminishing returns after three key locations and the number of states does not vary much. In general, for the models presented in this study, only 2–3 key locations are required for the meta heuristic to be effective. In the possible race condition shown in Figure 3.1 (`AbsList` model), however, we needed to specify a minimum of five key program locations in the input sequence for the meta heuristic to find a corresponding concrete error trace. Recall that the `AbsList` model represents the race-condition in the `AbstractList` class while using the synchronized `Vector` sub-class in the JDK 1.4 library.

3.4 Related Work

Static analysis techniques ignore the actual execution environment of the program and reason about errors by simply analyzing the source code of the program. ESC/Java relies heavily on program annotations to find deadlocks and race-conditions in the programs [Flanagan et al., 2002]. Annotating existing code is cumbersome and time consuming. RacerX does a top-down inter-procedural analysis starting from the root of the program [Engler and Ashcraft, 2003]. Similarly, the work by Williams *et al.* does a static deadlock detection in Java libraries [Williams et al., 2005]. FindBugs and Jlint look for suspicious patterns in Java programs [Artho and Biere, 2001, Hovemeyer and Pugh, 2004]. Error warnings reported by static analysis tools have to be manually verified which is difficult and sometimes not possible. The output of such techniques, however, serve as ideal input for the meta heuristic presented in this paper. Furthermore, dynamic analysis techniques can also be used to generate warnings about potential errors in the programs [Havelund, 2000, Shacham et al., 2007].

Model checking is a formal approach for systematically exploring the behavior of a concurrent software system to verify whether the system satisfies the user specified properties [Holzmann, 2003, Visser et al., 2000a]. In contrast to exhaustively searching the system, directed model checking uses heuristics to guide the search quickly toward the error [Edelkamp and Mehler, 2003, Edelkamp et al., 2001b, Groce and Visser, 2002b, Rungta and Mercer, 2005, 2006, 2009b]. Property-based heuristics and structural heuristics consider the property being verified and structure of the program respectively to compute a heuristic rank [Edelkamp et al., 2001b, Groce and Visser, 2002b]. Distance estimate heuristics rank the states based on the distance to a possible error location [Edelkamp and Mehler, 2003, Rungta and Mercer, 2005, 2006, 2009b]. As seen in the results, the PFSM distance heuristic is very effective in guiding the search toward a particular location; however, its success is dramatically improved in combination with the meta heuristic.

The trail directed model checking by Edelkamp *et. al* uses a concrete counter-example generated by a depth-first search as input to its guidance strategy [Edelkamp et al., 2001a]. It uses information from the original counter-example (trail) in order to generate an optimal counter-example. The goal in this work, however, is to achieve error discovery in models where exhaustive search techniques are unable to find an error. The deterministic execution technique used to test concurrent Java monitors is related to the technique presented in this paper [Harvey and Strooper, 2001]. The deterministic execution approach, however, requires a significant manual effort with the tester required to provide data values to execute different branch conditions, thread schedules, and sequence of methods.

Similar ideas of guiding the program execution using information from some abstraction of the system have been explored in hardware verification with considerable success [Nanshi and Somenzi, 2006, Paula and Hu, 2007]. An interesting avenue of future work would be to study the reasons for the success (in concretizing abstract traces by guiding program execution) that we observe in such disparate domains with very different abstraction and guidance strategies.

3.5 Conclusions and Future Work

This paper presents a meta heuristic that automatically verifies the presence of errors in real multi-threaded Java programs based on static analysis warnings. We provide the meta heuristic a sequence of locations and it automatically controls scheduling decisions to direct the execution of the program using a two-tier ranking scheme in a greedy-depth first manner. The study presented in this paper shows that the meta heuristic is effective in error discovery in subjects where randomized depth-first search fails to find an error. Using the meta heuristic we discovered real concurrency errors in the JDK 1.4 library. In future work we want to take the output of a static analysis tool and automatically generate the input sequence using control and data dependence analyses. Also we would like to extend the technique to handle non-determinism arising due to data values.

Chapter 4

A Distance Heuristic for Programs with Polymorphism

This chapter was published as:

N. Rungta and E. G. Mercer, "Guided Model Checking for Programs with Polymorphism", in *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*, Savannah, Georgia, USA, January 2009.

Abstract

Exhaustive model checking search techniques are ineffective for error discovery in large and complex multi-threaded software systems. Distance estimate heuristics guide the concrete execution of the program toward a possible error location. The estimate is a lower-bound computed on a statically generated abstract model of the program that ignores all data values and only considers control flow. In this paper we describe a new distance estimate heuristic that efficiently computes a tighter lower-bound in programs with polymorphism when compared to the state of the art distance heuristic. We statically generate conservative distance estimates and refine the estimates when the targets of dynamic method invocations are resolved. In our empirical analysis the state of the art approach is computationally infeasible for large programs with polymorphism while our new distance heuristic can quickly detect the errors.

4.1 Introduction

The ubiquity of multi-core processors is creating a paradigm shift from inherently sequential to highly concurrent and parallel systems. The lack of scalable verification techniques to detect concurrency errors is proving to be a hindrance for programmers developing concurrent programs. The trend toward parallelism and concurrency motivates a need to develop effective and scalable error detection techniques for concurrent programs.

Model checking techniques exhaustively enumerate all possible behaviors of the system to verify the presence as well as the absence of errors in programs [Ball and Rajamani, 2001, Henzinger et al., 2003, Holzmann, 2003, Robby et al., 2003, Visser et al., 2003]. The systematic exploration of all possible behaviors enables model checking to find subtle concurrency errors that are often missed by ad-hoc testing techniques. The exhaustive nature of model checking leads to a huge state space explosion making it intractable in verifying practical applications.

Guided model checking tries to overcome the state space explosion problem by focusing the search in parts of the program that are more likely to contain an error [Edelkamp and Mehler, 2003, Edelkamp et al., 2001b, Groce and Visser, 2002b, Rungta and Mercer, 2005, 2006]. Guided model checking techniques use heuristic functions to rank states in order of interest in an attempt to quickly generate a counterexample. States are ordered in a priority queue or a search stack based on their heuristic rank and path cost such that the states estimated to lead to an error state are explored before others.

Distance estimate heuristics try to compute a reasonable lower-bound on the number of computation steps required to reach a target location from the current location [Edelkamp and Mehler, 2003, Rungta and Mercer, 2005, 2006]. The distance estimates are used to guide the concrete program execution toward the target locations. The target locations are either provided by the user or generated using static analysis techniques. The estimates are computed on a statically generated abstract model that ignores all data values and only considers the control flow of the program. In the absence of data values to compute an accurate distance estimate between two arbitrary program locations is undecidable in general.

The FSM distance heuristic is the state of the art distance heuristic for programs with polymorphism [Edelkamp and Mehler, 2003]. The FSM distance heuristic ignores all calling context information and is unable to compute a reasonable lower-bound. Furthermore, the complexity of the FSM distance heuristic is cubic in the total number of instructions in the program. This complexity renders the FSM distance heuristic intractable for computing distance estimates in medium to large sized programs. Note that this is after we perform a rapid type analysis that uses the information about instantiated classes to create a reduced set of executable methods in programs with polymorphism [Bacon and Sweeney, 1996].

In this work we present a distance heuristic estimate that computes a tighter lower-bound on the distance estimates in polymorphic programs compared to the FSM distance heuristic. The polymorphic distance heuristic first performs an inter-procedural static analysis to compute an initial lower-bound on distance estimates; second, during the model checking we refine the distance estimates on demand when the targets of dynamic method invocations are resolved. The complexity of the new approach is cubic in the number of instructions in the method with the largest number of instructions. Note that this is significantly less than the complexity of the FSM distance heuristic.

We present an empirical analysis to demonstrate the effectiveness of the polymorphic distance heuristic in error discovery in a set of benchmarked multi-threaded programs where exhaustive and randomized search techniques are unable to find an error. We compare the polymorphic distance heuristic to the FSM distance heuristic and a random heuristic that assigns a random value as the rank of a state. Using the polymorphic distance heuristic we are able to detect real errors in the JDK 1.4 concurrent library. We also demonstrate that it significantly outperforms the FSM distance heuristic and the random heuristic in the number of states, time taken, and total memory used before error discovery.

4.2 Background

Distance estimate heuristics compute a heuristic value based on the distance to a target state, t, from a current state, s. The state s contains a set of unique thread identifiers, a program location and stack for each thread, and a heap. A transition relation generates a set of successor states for s, $\{s'_0, s'_1, \ldots, s'_n\}$, where the transition to each successor $s \rightarrow s'_i$ represents a possible change in state from s. Iteratively applying the transition relation to each state allows us to build the entire reachable behavior space of the program. A path, π, is a sequence of transitions, $s \rightarrow s' \rightarrow s'' \rightarrow s''' \ldots$, that represents a feasible execution path in the behavior space.

Definition 10. *The distance, d, between state, s, and target state, t, is the number of computation steps in an execution path from s to t: $d(s,t) := |\pi|$ where $\pi := s \rightarrow s' \rightarrow s'' \rightarrow \ldots \rightarrow t$.*

Computing accurate distance estimates between two states requires us to first build the reachable state space of a program that essentially entails solving the original problem a priori. In order to overcome this problem, distance estimate heuristics use a heuristic function, h, that approximates the distance between s and t on a statically generated abstract transition graph of the program. In the abstract system a state simply contains a unique program location identifier. In other words, an abstract state represents a single program instruction. The control flow of the program is the transition relation used to generate the abstract transition system. The abstract transition system ignores all data values and is an over-approximation of the original system. The program location of the recently executed thread in state s is used to map the concrete state to an abstract state. The heuristic function, h, estimates the distance between s and t by computing the distance between their respective abstract counterparts in the abstract transition system. Different distance heuristics compute the heuristic values on the abstract transition graph with varying degrees of calling context information.

The FSM distance heuristic is the state of the art heuristic for computing distance estimates in programs with polymorphism [Edelkamp and Mehler, 2003]. The FSM distance heuristic performs an interprocedural control flow analysis to statically compute the lower-bound on the distance between two arbitrary instructions in the program. The FSM distance heuristic is unable to compute a reasonable lower-bound because it ignores all calling context information and simply minimizes across the different methods. A detailed example is shown in [Rungta and Mercer, 2005]. This problem is further exacerbated in programs with polymorphism because it minimizes the distance estimates across all implementing sub-type targets of a dynamic method invocation.

The e-FCA heuristic computes full calling context-aware distance estimates in non-recursive C programs with resolved function pointers using a combination of static and dynamic information [Rungta and Mercer, 2006]. The e-FCA improves on previous distance estimates based on the FSM heuristic function and the EFSM heuristic function [Edelkamp and Mehler, 2003, Rungta and Mercer, 2005]. The FSM distance heuristic does not consider any calling context while the EFSM distance heuristic only considers partial context information. A comparative empirical study in [Rungta and Mercer, 2006] demonstrates that computing a tighter lower-bound by adding more calling context information enables us to more efficiently find an error.

The e-FCA distance estimate is computed based on the statically generated abstract model that only contains control flow information with following rules: at a given program location, we can either (a) reach the return statement of the current function and return to its caller without encountering the target location; or (b) reach the target location without executing the return statement of the current function (the target location can be reached in the forward direction). In cases where the target location cannot be reached in the forward direction, the e-FCA looks up the return point of the current function on the dynamic runtime stack of the recently executed thread in the concrete state. If the target location is reachable in the forward direction from the return point then the final estimate is the summation of the cost of moving to the return point and the distance in the forward direction from the return point to the target location. Otherwise, the algorithm keeps unrolling the stack until it reaches the `main` function.

The e-FCA lower-bounds all distance estimates by computing the shortest paths through various branching and looping constructs of a program. This allows the heuristic to be admissible and consistent.

Definition 11. *An admissible heuristic h is a function that guarantees a lower bound on the distance from every state, s, to the target state, t: $h(s,t) \leq d(s,t)$.*

Definition 12. *A consistent heuristic h is a function that guarantees for every state s and each successor of s' of s the estimated distance from s to t is less than or equal to the distance between s and s' plus the estimated distance from s' to t: $h(s,t) \leq d(s,s') + h(s',t)$*

In an A^* search, [Russell et al., 1995], the e-FCA generates minimal length counter-examples. The e-FCA heuristic is, however, not designed to compute distance estimates in the presence of dynamic method invocations whose targets cannot be statically resolved using a type analysis.

4.3 Motivation

```
 1: class AbstractList implements List{
 2: ...
 3: public boolean equals(Object $o$){
 4:  if $o == this$ then
 5:    return $true$;
 6:  if $\neg$($o$ instanceof List) then
 7:    return $false$;
 8:  ListIterator $e_1$ := ListIterator();
 9:  ListIterator $e_2$ := (List $o$).listIterator();
10:  while $e_1$.hasNext() and $e_2$.hasNext() do
11:    Object $o_1$ := $e_1$.next();
12:    Object $o_2$ := $e_2$.next();
13:    if$\neg$($o_1$ == null ? $o_2$ == null : $o_1$.equals($o_2$))then
14:      return $false$;
15:  return $\neg$($e_1$.hasNext() || $e_2$.hasNext())
16: }
17: ...
18: }
```

Figure 4.1: The `equals` function in the `AbstractList` implementation of the JDK 1.4 library which uses polymorphism.

It is important to compute accurate distance estimates in the presence of polymorphism because there is an increasing use of object oriented languages like Java and C# which inherently encourage the use of polymorphism. More importantly, Java and C# are being used to develop concurrent applications because they natively support concurrency. The inability of the FSM distance heuristic to compute estimates in the presence of polymorphism makes it ineffective for guiding program execution in Java and C# programs.

The example shown in Figure 4.1 is the `equals` function of the `AbstractList` class in the JDK 1.4 concurrent library. In order to compute the distance estimate from the start to the end of the `equals` method we have to evaluate the cost of moving through the method calls in Figure 4.1. The list iterator operations (lines 8-12 and 15) and the call to `equals` on the objects from both lists (line 13) are dynamic method invocations whose targets cannot be determined statically. A very small portion of the call graph with the class hierarchy for the method in Figure 4.1 is shown in Figure 4.2. The call graph shows that even for a single method call, there

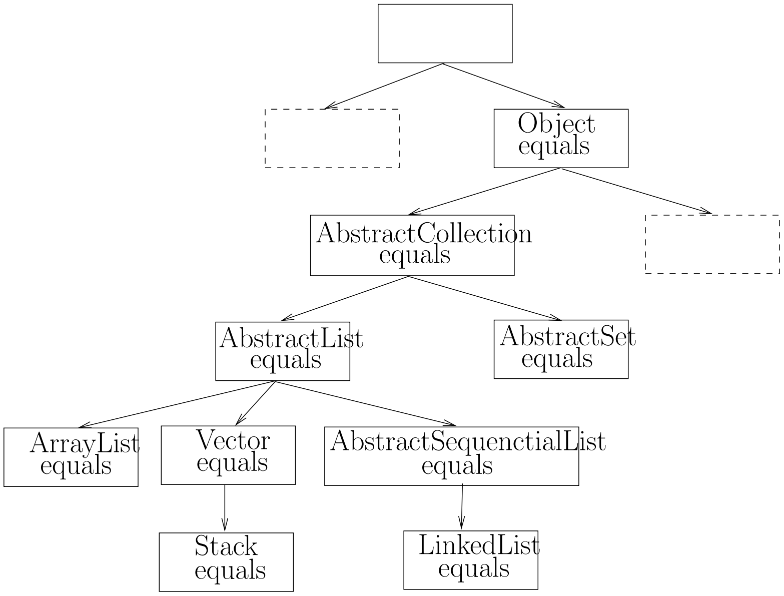


Figure 4.2: A partial call graph for the `equals` function in the `AbstractList` implementation.

may be a large number of possible functions that we need to evaluate for computing heuristic estimates.

Simply minimizing the distance estimates across all the methods implemented by the sub-classes for a particular polymorphic method call is not computationally feasible. Our tests show that even for medium-sized programs such an analysis does not complete within a time bound of one hour. The heuristic estimates computed using such a brute force approach also tend to be inaccurate because at every program location it simply computes a lower-bound across all implementations. For programs with a large number of types the inaccurate estimates degenerate essentially into random estimates.

4.4 Polymorphic Distance Heuristic

In this section we present a new polymorphic distance estimate (PFSM) that performs an interprocedural static analysis to conservatively compute distance estimates with partial context information for targets of dynamic method invocations that are not statically resolved with a type analysis (unresolved polymorphic methods). It then dynamically computes the distance estimates on demand when the type of polymorphic methods are resolved during model checking. In other words, without completely analyzing all subtypes it lower-bounds distance estimates and computes the estimate on demand as type information is discovered at runtime.

```
procedure polymorphic_distance_heuristic($main$)
/* $N$ is set of nodes, $E$ is the set of edges, $n_{start}$ is the start node, and $n_{end}$ is the end node in the CFG */
$\langle N, E, n_{start}, n_{end}\rangle$ := get_CFG($main$)
compute_estimates($\langle N, E, n_{start}, n_{end}\rangle$)

procedure compute_estimates($\langle N, E, n_{start}, n_{end}\rangle$)
/* Entries along the diagonal are 0 while others are $\infty$ */
$L : |N| \times |N| \to \mathbb{N} \cup \{\infty\}$
$L$ := analyze_function($n_{start}, L, \emptyset$)
$L$ := compute_all_pairs_shortest_distance($L$)
$Explored$.add($\langle N, E, n_{start}, n_{end}\rangle, L$)

procedure analyze_function($n, L, Visited$)
if is_call_site($n$) then
    if has_resolved_type($n$) then
        $\langle N', E', n'_{start}, n'_{end}\rangle$ := get_target_CFG($n$)
        $d_{succ}$ := get_distance_to_end($\langle N', E', n'_{start}, n'_{end}\rangle, n$)
    else
        $d_{succ} := 2$ /* Conservative estimate */
else
    $d_{succ} := 1$ /* Instructions other than call sites */
for each $n' \in$ succ($n$) and $n' \notin Visited$ do
    $L(n, n') := d_{succ}$;  $Visited := Visited \cup \{n'\}$
    $L$ := analyze_function($n', L, Visited$)
return $L$

procedure get_distance_to_end($\langle N, E, n_{start}, n_{end}\rangle, n_c$)
if $n_c \in N$ then
    return 2 /* Recursive call */
if $\neg Explored$.contains($\langle N, E, n_{start}, n_{end}\rangle$) then
    compute_estimates($\langle N, E, n_{start}, n_{end}\rangle$)
$L$ := $Explored$.get_element($\langle N, E, n_{start}, n_{end}\rangle$)
return $L(n_{start}, n_{end})$
```

Figure 4.3: Pseudocode for computing distance estimates statically.

4.4.1 Static analysis phase

An abstract model of the program is created to compute initial distance estimates. The model ignores all data values of the program and focuses only on control flow. The abstract model combines a control flow graph for each procedure in the program and a call graph that represents the call hierarchy of various procedures. The control flow graphs and the call graph denote the control flow of the program at an intra and inter procedural level respectively. The distance estimates between instructions in a procedure are computed as a lower-bound in the presence of branching and iterative constructs.

We algorithmically construct the abstract model and compute the distances between instructions in a method. The algorithm uses a reverse invocation order to estimate the cost of moving through method calls if the type of the callee can be statically determined. The analysis, however, does not step into methods whose type cannot be statically resolved after a rapid type analysis. In such cases a conservative estimate of two (one to call the function and another for the return edge) is assigned as the cost of moving through the corresponding call site to its immediate successor in the analysis. The conservative estimates are superseded by the distance estimates dynamically computed on demand as the type of methods is resolved during the model checking run.

The pseudo-code for the static analysis phase of computing the distance estimate values is presented in Figure 4.3. The tuple, $\langle N, E, n_{start}, n_{end} \rangle$, is a control flow graph (CFG) where N is a set of abstract nodes labeled with unique program location identifiers, $E \subseteq N \times N$ is the set of edges, $n_{start} \in N$ is the start node, and n_{end} is the end node in the CFG. The variable, L, is a matrix of values that holds the distance estimates between instructions in each method. The *Explored* variable is a map used to memoize the distance matrices for the different CFGs so that each method in the program is evaluated only once. The *Visited* set is used to detect cycles in the control flow of a particular method. The function is_call_site takes as input a node in the CFG and returns true if the node represents a call site in the program. The has_resolved_type function takes as input a node that is a call site and returns true if the type of the target method (callee) is statically resolved after the rapid type analysis; the function get_target_CFG returns the CFG of the target method given a call site. Finally, the succ function returns a set of the immediate successors of a node n in the CFG, $\mathsf{succ}(n) = \{n' \in N | (n, n') \in E\}$.

The polymorphic_distance_heuristic function is invoked by the `main` method to statically compute distance estimates as shown in Figure 4.3 (lines 1-4). The function invokes the compute_estimates function with the CFG of the `main` method (lines 2-3). The compute_estimates function initializes a distance matrix $L : |N| \times |N|$ where the entries along the diagonal are set to zero while all other entries are set to ∞ (line 6). Next, on line 7 of Figure 4.3, the analyze_function is called with the start node of the CFG, n_{start}, and the corresponding distance matrix, L, to initialize the edge costs between the nodes in the CFG.

The analyze_function uses a depth-first search traversal (lines 19-21) to update edge costs in L. For all nodes that are not call sites, the distance between the node and its immediate successor, d_{succ}, is set to one (line 18). When we encounter a call site during the traversal whose target method cannot be statically resolved then we conservatively set the cost of moving from the call site to its immediate successor node as two (lines 15-16). In essence, we do not evaluate any methods whose type cannot be statically resolved. If the type of the method can be resolved statically we update the cost between the call site and its successor by computing the distance estimate of moving through the target method (lines 12-14). After all the edge costs are updated in the distance matrix, L, the matrix is returned (line 22). At this point the analysis resumes on line 8 of the compute_estimates function where an all-pairs shortest path analysis is performed on the distance matrix. Finally the matrix is added to the *Explored* map with its corresponding CFG (lines 8-9).

To compute the cost of moving through the target method of a call site, we invoke the get_distance_to_end function with the CFG of the target method and its corresponding call site (line 14). A simple check of whether the call site is also part of the target CFG reveals a recursive method call, in which case a conservative estimate of two is returned. In non-recursive method calls, if the target method is not found in the *Explored* set, then we step into the target method by calling the compute_estimates function with the CFG of the target method (line 27). When the execution flow returns on line 28 of Figure 4.3, we get the corresponding distance matrix, L, for the target method. The shortest distance from the start node to the end node in the distance matrix is returned as the cost of moving from the call site to its immediate successor on line 14 of the analyze_function.

4.4.2 Guided Search

The heuristic computed on the abstract model is used to intelligently rank the concrete states generated at points of thread non-determinism during model checking. A concrete state, s, contains a set of unique thread identifiers, a program location and stack for each thread and a heap. For each successor, s', of s the PFSM distance estimate from the current program location of the recently executed thread in s' to the specified target location is assigned as the heuristic rank of the s'. Intuitively, the PFSM heuristic drives certain threads toward the target locations specified by the user or generated using static analysis techniques.

4.4.3 Dynamic heuristic computation

The abstract model consisting of control flow graphs and the call graph of the program is refined when type information is discovered during model checking. As the

```
procedure get_forward_distance_estimate($curLoc$, $targetLoc$)
$\langle N, E, n_{start}, n_{end} \rangle$ := get_function_containing($curLoc$)
if $\neg Explored$.contains($\langle N, E, n_{start}, n_{end} \rangle$) then
    compute_estimates($\langle N, E, n_{start}, n_{end} \rangle$)
if $targetLoc \in N$ then
    return get_distance($curLoc$, $targetLoc$)
return get_estimate(get_CFG_node($curLoc$), get_CFG_node($targetLoc$))

procedure get_estimate($n, n_t$)
$hVal := \infty$
for each $n' \in$ call_sites(get_function_containing($n$)) do
    if not_related($n', n_t$) then
        continue
    $d :=$ get_distance($n, n'$)
    if $d < hVal$ then
        $hVal :=$ min(compute_dynamic_estimate($n', n_t, d, hVal$), $hVal$)
return $hVal$

procedure compute_dynamic_estimate($n_c, n_t, d, hVal$)
if $n_t \in$ target_CFG_nodes($n_c$) then
    $hVal' := d +$ get_distance_from_start_to_node($n_t$) $+ 1$
    return $hVal'$
else
    /* $CGR \subseteq X_c \times X_c$ where $X_c$ is the set of all call sites in the program. */
    for each $n'_c \in CGR(n_c)$ do
        if not_related($n'_c, n_t$) then
            continue
        $d' := d +$ get_distance_from_start_to_node($n'_c$) $+ 1$
        if $d' < hVal$ then
            $hVal :=$ min(compute_dynamic_estimate($n'_c, n_t, d', hVal$), $hVal$)
    return $hVal$

```

Figure 4.4: Pseudocode for computing the distance heuristic during runtime

refinement step we compute the distance estimates between the instructions in the control flow graph of a procedure whose alias information is discovered during the model checking run. The final heuristic value is computed on the abstract model along a sequence of call sites across the different control flow graphs from the current location to the target location.

We algorithmically show the heuristic computation in the dynamic analysis phase of the PFSM heuristic. The algorithm traverses the call graph in a depth-first manner to implicitly construct call traces between the current location and the target location in the forward direction. It uses the type information in the state generated during model checking to compute the distance estimates on demand along a particular call trace by using correct alias information to resolve types. The algorithm uses a branch and bound technique to restrict the number of call traces that need to be evaluated. The algorithm minimizes the distance estimate among all the call traces that lead from the current location to the target location. If the target location is not reachable in the forward direction then we look up the return point of the current function in the runtime stack extracted from the state generated during model checking as described in [Rungta and Mercer, 2005]. Next, if the target location is reachable from the return point we return the sum of the cost of moving to the end of the current function plus the distance estimate from the return point to the target location as the heuristic estimate; otherwise we keep unrolling the stack and repeat the above process.

The pseudocode for the dynamic phase of the algorithm is shown in Figure 4.4. The get_forward_distance_estimate function in Figure 4.4 takes as input the current program location of the most recently executed thread in the concrete state (*curLoc*) and the target location (*targetLoc*) to compute the distance estimate between them in the forward direction. The get_function_containing returns the CFG which contains the current program location (line 1). If the CFG containing the current location has not been previously analyzed (line 2) then we know that the type of a polymorphic method is now resolved. At this point we can compute the distance estimates between the instructions in the method (line 3) by calling the compute_estimates function in Figure 4.3. Next, if the target node is contained within the same CFG as the current node (line 4) then the algorithm returns the value obtained from the get_distance function. The get_distance function returns the shortest distance between the two nodes in the same CFG. Note that the distance between the two nodes in the CFG is computed using partial context information because the algorithm conservatively assigns the distance between a call site for an unresolved polymorphic type to its immediate successor as two in the CFG.

The get_estimate and compute_dynamic_estimate functions traverse the nodes in the call graph implicitly constructing the call traces from the current location to the target location in the forward direction to compute the heuristic value, $hVal$. The function uses a branch and bound algorithm in an attempt to restrict the number of call traces that need to be evaluated for computing $hVal$. The function call_sites (line 9) generates the set of nodes that represent call sites in the input CFG while the not_related (lines 10 and 23) function returns true if there does not exist a path between the input nodes, n' and n_t (line 10), in the forward direction on the call graph. The get_estimate and compute_dynamic_estimate functions compute the distances along call traces using a depth-first traversal of the call graph (lines 9-14 and 22-27 respectively) such that the target node is reachable in the forward direction along the call trace. Note that we detect loops in the call trace in our implementation and backtrack appropriately. The get_estimate function constructs the first part of the call trace. It gets the distance from the current location to a call site within its own method that leads to the target node (lines 10-12). The get_estimate function then calls compute_dynamic_estimate (line 14) to compute the distance estimate on the rest of the call trace.

The compute_dynamic_estimate function computes the distances through the different call sites in a call trace. It uses a call graph relation, $CGR \subseteq X_c \times X_c$, where X_c is the set of the call sites in the entire program, to build a path through the different call sites in the program (lines 22-27) to a target location. Intuitively, a call graph relation describes the edges between different nodes in a call graph. The algorithm maintains a running summary of the distance estimates between the call sites (line 25). The get_distance_from_start_to_node function takes as input a node (which in this case is a call site) and gets the CFG that contains the input node. If the *Explored* set contains the CFG then the get_distance_from_start_to_node function returns the shortest distance from the start node of the CFG, n_{start}, to the call site; otherwise it returns a conservative estimate of two. This essentially computes the distance estimates between different call sites in the call trace. When the algorithm reaches a call site whose callee CFG contains the target node, the function returns the summation of the distances along the path in the call trace up to the target node as the heuristic value (lines 17-19). The heuristic value is computed as a lower-bound and is propagated along the different call paths to prune other call traces when the value along a path becomes greater than the current heuristic value.

Theorem 1. *The PFSM heuristic computes a lower-bound on the distance estimate, if there exists one or more sequences of call points, $\langle c_0, c_1, \ldots, c_k \rangle$, in the call graph through k methods, in the presence of unresolved polymorphic methods, that represent a path between the current location, l, and the target location, t, $d_{min}(l, t) := d_{min}(l, c_0) + \sum_{i=1 to \ k-1} d_{min}(\mathsf{start}(i), c_i) + d_{min}(\mathsf{start}(k), t)$, between l and t and minimizes across all call sequences.*

Proof. Assume that the algorithm does not lower-bound the distance estimate along a particular sequence of call points. There are two possible cases when computing the distance along a sequence of call points. (1) The type of the method containing a call point is resolved—either statically or dynamically. Here the algorithm performs an all-pairs shortest path analysis on the CFG of the method. The analysis returns values between the nodes that are a lower-bound on the actual distance in the presence of branching and looping constructs. (2) The type of the method containing the call point is not resolved. The algorithm assigns a lower-bound of two to account for moving from the start of the method to a call site and then moving to next call site. The summation of all the values as the algorithm moves along a particular call sequence is a lower-bound on the distance estimate between l and t which contradicts our assumption. □

Corollary 1. *The PFSM heuristic estimate is consistent.*

Proof. The proof follows the one described for the FSM distance heuristic in [Edelkamp and Mehler, 2003]. There are two possible cases: (1) The shortest path from s to t contains s'. Suppose the length of the shortest path from s to t is l then, by definition, $h(s,t) = h(s',t) - d(s,s')$ which satisfies $h(s,t) \leq h(s',t) + d(s,s')$. (2) The shortest path from s to t *does not* contain s'. Consider the path, $\pi = s \rightarrow s' \rightarrow \ldots \rightarrow t$, where $|\pi| \geq l + d(s,s')$ and l is the shortest path between s and t. Furthermore, $\pi' = s \rightarrow \ldots \rightarrow t$, which implies $|\pi'| \geq l$ and $h(s') \geq l$. Hence we have $h(s,t) \leq h(s',t) + d(s,s')$. □

Corollary 2. *The PFSM heuristic is admissible.*

Proof. By definition, a consistent heuristic is also admissible [Russell et al., 1995]. From Corollary 1 we know that the PFSM heuristic is admissible.

Theorem 2. *The complexity of computing the PFSM distance heuristic in the forward direction is $O(\sum_{i=0}^{rm} |N_i|^3 + |N_c + E_c|)$ where rm is the number of methods with resolved types, N_i is set of nodes representing instructions in method i, N_c is the set of nodes in the call graph, and E_c is the set of edges in the call graph.*

Proof. The complexity of computing the PFSM distance heuristic in the forward direction is $O(\sum_{i=0}^{rm} |N_i|^3)$ for performing an all-pairs shortest path analysis on every method whose type has either been statically (line 8 in Figure 4.3) or dynamically resolved. Note that when the type of a method is dynamically resolved, line 3 in Figure 4.4 calls the function compute_estimates in Figure 4.3 and then performs the all-pairs shortest path analysis on line 8 in Figure 4.3. The complexity of the PFSM distance heuristic is also linear in the number of nodes and edges in the call graph as it computes a lower-bound on the distance estimates across the different call sequences between the current and target location (lines 9-14 and 22-27 in Figure 4.4). In the worst case, the algorithm explores the entire call-graph in a depth-first manner;

in general, however, propagating the lower-bound across the different call sequences is successful in pruning a large number of call sequences that do not need to be explored. □

In contrast, the FSM distance heuristic, minimizes over all possible implementing sub-types of a particular method and has a complexity of $O(X^3)$, where X is the number of total instructions in the program $X = \sum_{1 \leq i \leq m} |N_i|$, and m is the total number of methods in the program; however, X is very large for most programs of interest. The PFSM heuristic is more computationally effective even though both heuristics belong to the same complexity class.

As the model checking run progresses, PFSM distance heuristic estimate, h_p, is a tighter lower-bound compared to the FSM distance heuristic estimate, h_f, such that $h_f \leq h_p$. The FSM distance heuristic is a context-insensitive algorithm and under-approximates distance values by ignoring all calling context. As the type of one or more methods are resolved during the model checking run the PFSM distance heuristic computes distances along different methods based on correct alias information. The PFSM distance heuristic uses the context information on the runtime stack of the state in a manner similar to the e-FCA [Rungta and Mercer, 2006]. A more detailed example demonstrating the effects of calling context is shown in [Rungta and Mercer, 2006].

4.4.4 Example of heuristic computation

We use the example in Figure 4.5 to demonstrate how the heuristic values are computed. The class `X` in Figure 4.5(a) is an abstract class with three methods: `aa`, `test`, and `bb`. The classes `Y` (Figure 4.5(b)) and `Z` (Figure 4.5(c)) inherit from the `X` class. In Figure 4.5(a), the input to the `test` method is an object, x, of type `X`. On lines 7 and 8, methods `bb` and `aa` are invoked, respectively, on the current instance of `X` and the input parameter x. Statically we can determine that the call

```
1:  public abstract class X{
2:
3:    public abstract void aa();
4:
5:    public void test(X x){
6:      i := 0;
7:      this.bb();
8:      x.aa();
9:     }
10:
11:   public void bb(){
12:     this.val := 10;
13:     this.otherVal := 11
14:   }
15: }
```

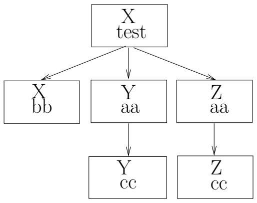
(a)

```
1:  class Y extends X{
2:
3:    public void aa(){
4:      this.cc();
5:     }
6:
7:    public void cc(){
8:     if(...) then
9:        throw RuntimeException()
10:    }
11: }
```

(b)

```
1:  class Z extends X{
2:
3:    public void aa(){
4:      this.cc();
5:     }
6:
7:    public void cc(){
8:    /* Local Instruction */
9:     }
10: }
```

(c)



(d)

Figure 4.5: An example program and its corresponding call graph to demonstrate the heuristic computation.(a) An abstract class, X, with an abstract method and implementations for two functions. (b) The Y class that inherits from the X class. (c) The Z class that inherits from the X class. (d) The call graph for the functions in X, Y, and Z.

on line 7 of the test method invokes the bb method on lines $11 - 15$ in Figure 4.5(a); however, aa is a dynamically invoked method and the target of the call on line 8 of Figure 4.5(a) depends on the type of x. The overall calling structure of the program is shown in Figure 4.5(d). The test method in X can call the aa method in either the Y or Z class. The aa method then calls the cc method in its respective class. In the example shown in Figure 4.5, the goal is to drive the program execution to line 9 in the cc method of the Y class in Figure 4.5(b). Recall that in medium to large programs evaluating all possible implementing subtypes is intractable.

Suppose for the program shown in Figure 4.5, a main method calls test with different instances of X objects. During the static analysis phase, when we reach the test function in Figure 4.5(a), the analysis accounts for the cost of moving through the *this*.bb method call on line 7 in Figure 4.5(a); however, the analysis cannot statically resolve the type of x; thus, the static analysis does not evaluate either implementation of aa in the Y or Z class and assigns a conservative estimate of two to account for the cost of moving from line 8 to the end of the test method. At the end of the static analysis, the *Explored* set only contains the test and bb methods.

Let us consider two cases in the dynamic computation of the heuristic. In the first case, suppose the current location of the program is at line 6 in Figure 4.5(a) and we want to compute a distance estimate to line 9 in Figure 4.5(b). We first get all the call sites that are reachable from the current location such that there exists a path from the call site to the target location on the call graph and the call sites are in the same CFG as the current location. The only call site that satisfies the condition in Figure 4.5 is x.aa(). We then call the get_estimate() function in Figure 4.4 with the corresponding call site. The x.aa() call site can call the aa function in either the X class or the Y class. This maps to two entries in the call graph relation: Y.aa() → Y.cc() and Z.aa() → Z.cc(); however, the target location can only be reached from Y.cc() based on the calling hierarchy shown in Figure 4.5(d). Since the distance estimates in the aa method of the Y class have not been computed on the CFG (as the method does not currently exist in the *Explored* set), a conservative cost of two is added along the call trace when moving from x.aa() to Y.cc(). Similarly, a conservative estimate of two is added for the cost of moving to the cc method in the Y class and reaching the target at line 9 because the cc method does not exist in the *Explored* set. A final heuristic estimate of four is returned for the example.

In another example that demonstrates the dynamic computation of the heuristic, suppose the current location of the program is at line 4 in Figure 4.5(b). The location implicitly resolves the type of the aa method in the Y class because the model checking search is at the method. At this point we run the static analysis algorithm (shown in Figure 4.3) on the aa method in Figure 4.5(b). Note that since the target of this.cc() is dynamically resolved the static analysis technique computes the cost of moving through the cc method at line 4 in Figure 4.5(b). After refining the distance estimates, we return to the dynamic heuristic computation in Figure 4.4. The analysis computes the distance estimates on the call trace Y.aa() → Y.cc() based on the shortest distances in the CFGs of the methods aa and cc in the Y class. The distance estimates in a CFG lower-bounds all values across the iterative and looping constructs.

4.5 Results

The experiments are conducted on machines with 8 GB of RAM and two Dual-core Intel Xeon EM64T processors (2.6 GHz). We run 100 trials of guided search with various heuristics. Note that we break all heuristic ties randomly that enables us to overcome the benefits and limitations of a default search order in guided search [Rungta and Mercer, 2007b, Appendix A]. All the trials are time bounded at one hour. This is consistent with other empirical studies [Dwyer et al., 2006, Rungta and Mercer, 2007a,b, Appendix A, Appendix B]. Since each trial is completely independent of the other trials we use a super computing cluster of 618 nodes to distribute the trials on different nodes.[1] Even though the algorithm does not require parallel computation, using the super computing cluster allows us to quickly generate results. We use the Java Pathfinder (JPF) v4.1 model checker with partial order reduction turned on to conduct the experiments described in the paper. JPF model checks Java byte-code using a modified virtual machine.

[1] We thank Mary and Ira Lou Fulton for their generous donations to the BYU Supercomputing laboratory.

The input to the guided search is the model and possible error locations. The possible error locations are be derived by user-specified reachability properties, can be generated by static analysis tools, or generated from dynamic analysis tools [Artho and Biere, 2001, Havelund, 2000, Hovemeyer and Pugh, 2004]. For example, static analysis tools report program locations where lock acquisitions by unique threads *may* lead to a deadlock. These tools, however, cannot state the feasibility of the deadlock. We use the technique described in [Rungta and Mercer, 2008, Chapter 3] to generate a sequence of program locations that are relevant to checking the reachability of the possible error locations.

We use five unique multi-threaded Java programs in this study to evaluate the effectiveness of the PFSM heuristic. Three programs are from the benchmark suite of multi-threaded Java programs gathered from academia, IBM, and classical concurrency errors described in literature [Dwyer et al., 2006]. We pick these three artifacts from the benchmark suite because the threads in these programs can be systematically manipulated to create configurations of the model where randomized depth-first search is unable to find errors in the models [Rungta and Mercer, 2007a, Appendix B]. These models also exhibit different concurrency error patterns described by Farchi *et. al* in [Farchi et al., 2003]. The `AbsList` and the `AryList` are programs that use the JDK 1.4 library in accordance with the documentation. We use Jlint on the `AbsList` and `AryList` models to automatically generate warnings on possible concurrency errors and then manually generate the input sequences as described in [Rungta and Mercer, 2008, Chapter 3]. The name, type of model, number of locations in the input sequence, and source lines of code (SLOC) for the models are as follows:

- **TwoStage**: Benchmark, Num of locs: 2, Null Pointer Exception, SLOC: 52
- **Reorder**: Benchmark, Num of locs: 2, Null Pointer Exception, SLOC: 44

- **Wronglock**: Benchmark, Num of locs: 3, Deadlock due to inconsistent locking. SLOC: 38
- **AbsList**: Real, Num of locs: 6, Race-condition in the AbstractList class using the synchronized Vector sub-class. SLOC: 7267
- **AryList**: Real, Num of locs: 6, Race-condition in the ArrayList class using the synchronized List implementation. SLOC: 7169

Exhaustive search techniques like randomized depth-first search either struggle or fail to find an error in the models used in the empirical study. A more detailed comparison with a randomized depth-first search is shown in [Rungta and Mercer, 2008, Chapter 3].

We use a greedy depth-first search to guide the search. The greedy depth-first search picks the best-ranked immediate successor of the current state and does not consider unexplored successors until it reaches the end of a path and needs to backtrack. We observe comparable results with a traditional greedy best-first search with a bounded queue. We use the distance heuristic to guide the search through each of the input locations generated using the technique in [Rungta and Mercer, 2008, Chapter 3] to mimic a test-like paradigm. The effects of varying the length of the sequence on the performance of the heuristic are also reported in [Rungta and Mercer, 2008, Chapter 3].

Only a portion of the frontier states are saved as backtrack points which turns the complete search into a partial search; however, our aim is to find a counter-example efficiently rather than to do an exhaustive proof or find an optimal counter-example. It is important to note that in medium to large programs, it is intractable to generate optimal counter-examples using an A^* search because it exhausts the memory resources very quickly.

In Table 4.1 we specifically compare the performance of the PFSM heuristic with the FSM and random heuristic while guiding the program execution through a small sequence of locations. The entries in Table 4.1 with "-" in the FSM heuristic columns indicate that the static analysis did not finish within the time bound of one hour.

The performance of the PFSM heuristic is dramatically better than the random and FSM heuristic. In the `Twostage(7,1)` model the PFSM heuristic generates a mere 213 states, on average, before error discovery while the random and FSM heuristic generate $109,259$ and $30,193$ states respectively, on average, in the error discovering trials. A similar improvement for the PFSM heuristic is noticed in the total time taken and memory used. In the `TwoStage(7,1)` model the PFSM only takes 0.42 seconds on average for error discovery, in contrast, the random heuristic takes 40.14 seconds while the FSM heuristic takes 39.11 seconds. In some models such as `Reorder(5,1)` and `Wronglock(1,10)` where the magnitude of states generated is small, the memory usage of the random heuristic is lower than the PFSM heuristic because it does not incur the additional heuristic computation cost. Note that the variance in the results using the FSM and PFSM heuristics is caused because we break all ties in heuristic values randomly.

4.6 Discussion

Recent work and our experience in testing and verifying multi-threaded programs show that only a small number of perturbations to certain global or shared variables are required to find a particular error in the multi-threaded system. The key, however, lies in discovering and driving the program execution through these perturbations to elicit the error. Recent work uses the output of static analysis warnings to generate a sequence of interesting programs relevant for verifying the feasibility a particular static analysis warning. The sequence is small with large gaps between each location. We rely on the distance estimate heuristic presented in this paper to guide the program execution toward the locations in the sequence. In essence, the distance heuristic drives certain threads toward specific program locations without manual intervention that is required in the other heuristics such as the prefer-thread heuristic [Groce and Visser, 2002b]. This allows us to scale to realistic benchmarks and discover errors after exploring only a few hundred states.

The heuristic lower-bounds the values across loops and recursive function calls. If the loops and recursive function calls operate solely on local variables the dynamic partial order reduction allows us to process a series of instructions as a single transaction; however, if they operate on global variables then the distance heuristic is sufficient to drive a particular thread through a loop until it exits the loop and moves toward the location of interest.

4.7 Conclusions and Future Work

In this work we present a distance heuristic function that computes estimates in programs with unresolved polymorphic methods. The PFSM heuristic performs an interprocedural static analysis to conservatively compute distances estimates and, then, dynamically computes the distance estimates on demand after the types of polymorphic methods are resolved at transaction boundaries during model checking. The empirical analysis shows that the PFSM heuristic outperforms the FSM distance heuristic that ignores the calling context information and the baseline random heuristic. In future work we want to study and evaluate the trade-off between the accuracy in the heuristic estimate and the performance in the heuristic computation in how it affects the effectiveness of the guided search. For example, to further improve the accuracy of the distance estimate we can propagate the types—extracted from the state—along the program, as far as possible. A def-use analysis could be used to detect how far we can propagate the values in the program.

EXPLORED STATES

Subject	Random Heuristic			FSM Heuristic			PFSM Heuristic		
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
Twostage(7,1)	15249	109259	409156	3279	30193	178653	209	213	217
Twostage(8,1)	23025	204790	603629	5956	46259	281132	246	251	255
Twostage(10,1)	36056	364859	1216340	14232	156697	1302040	329	335	340
Wronglock(1,10)	58	7064	49100	75	196	2362	367	3781	15923
Reorder(5,1)	1803	6006	12529	912	2562	5765	106	109	112
Reorder(8,1)	10155	34193	98683	5422	24022	96681	193	197	202
Reorder(10,1)	24890	80160	343429	6785	65506	149916	266	272	277
AryList(1,10)	3652	15972	63206	-	-	-	846	5216	50904
AbsList(1,10)	10497302	10497302	10497302	-	-	-	982	982	982

TIME IN SECONDS

Subject	Random Heuristic			FSM Heuristic			PFSM Heuristic		
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
Twostage(7,1)	4.52	40.14	124.17	33.49	39.11	65.93	0.37	0.42	2.57
Twostage(8,1)	6.70	76.24	184.82	34.65	41.87	83.45	0.39	0.41	0.49
Twostage(10,1)	10.89	132.08	318.93	36.35	59.90	242.78	0.43	0.46	0.52
Wronglock(1,10)	0.22	2.85	12.46	10.25	10.70	12.49	0.48	1.66	4.24
Reorder(5,1)	1.19	2.34	4.08	12.78	13.37	14.41	0.28	0.31	0.67
Reorder(8,1)	3.59	9.70	34.72	13.90	17.84	34.02	0.34	0.39	0.54
Reorder(10,1)	6.81	25.62	97.33	14.31	26.30	41.99	0.37	0.41	0.45
AryList(1,10)	2.12	7.95	26.11	-	-	-	12.21	13.60	22.56
AbsList(1,10)	2585.79	2585.79	2585.79	-	-	-	4.85	4.92	5.92

MEMORY IN MB

Subject	Random Heuristic			FSM Heuristic			PFSM Heuristic		
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
Twostage(7,1)	219	972	2090	922	1325	2219	160	182	203
Twostage(8,1)	352	1415	2541	961	1462	2411	163	178	203
Twostage(10,1)	508	2038	3902	1033	1886	3227	163	181	203
Wronglock(1,10)	18	117	374	204	434	693	77	114	187
Reorder(5,1)	50	89	166	185	348	590	160	179	203
Reorder(8,1)	159	387	848	214	571	1168	160	173	203
Reorder(10,1)	279	851	1856	362	1032	1453	163	179	203
AryList(1,10)	136	275	572	-	-	-	280	318	391
AbsList(1,10)	6154	6154	6154	-	-	-	165	193	256

Table 4.1: Comparing the performance of various heuristics.

Chapter 5

An Extensive Comparative Empirical Analysis

This chapter was published as:

N. Rungta and E. G. Mercer, “Clash of the Titans: Tools and Techniques for Hunting Bugs in Concurrent Programs”, in *Proceedings of Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD VII)*, Chicago, US, July 2009.

Abstract

To evaluate the effectiveness of guided test in detecting bugs compared to other state of the art tools and techniques we create a benchmark suite of concurrent programs for Java and C# programs. We have compiled a set of Java benchmarks from various sources and our own efforts. For many of the Java examples we have created structurally equivalent C# programs. All the benchmarks are available for download.

In our multi-language benchmark suite we compare results from the following tools: CalFuzzer, ConTest, CHESS, and Java Pathfinder. We provide extensive results for the Java Pathfinder using stateless random walk, randomized depth-first search, and guided search using abstraction refinement. Based on this data, we argue that iterative context-bounding and dynamic partial order reduction are not sufficient to render model checking for testing concurrent programs tractable and secondary

techniques such as guidance strategies are required. As part of this work, we have also created a wiki to publish benchmark details and data from the specified tools on those benchmarks to a broader research community.

5.1 Introduction

The last several years have seen growth in both multi-core processors and a desire to write concurrent programs to take advantage of these multi-core processors. The growth, however, has not been matched by any improvement in our ability to test, analyze, and debug concurrent programs. For example, despite the proliferation of concurrent programs, developers are largely unaware as to how these programs should be tested (or even written for that matter), and as a result often employ stress testing which is known to be very ineffective in detecting concurrency errors.

To be fair, the growth in concurrent programming has been matched by research into methods such as model checking to test, analyze, and debug such programs; although, the research has yet to be widely adopted or even shown to be practically applicable to mainstream programming. This assertion we believe to be true of static analysis, test generation, model checking, or any other such technique. We believe there are several reasons for such a slow uptake such as

i) No technique has been shown effective in a general setting. Of the myriad of algorithms and tools, there has yet to be a single approach (or even a combination of approaches) that is demonstrated clearly to be effective.

ii) No clear comparison of competing approaches. It is very difficult to even begin to identify and select a possible superior approach because, as is common in academics and especially model checking, every tool uses a different input language, produces a different output format, and tests on a unique set of benchmarks often distinct from other researchers (and not available too). How is one to compare without re-implementing every published technique that is seemingly useful?

iii) We have yet to discover the right technology (or combination of technologies). The research community may still be in search of the needed technology to manage, maintain, and develop the emerging concurrent world.

iv) There is no money in it. Perhaps it is just a matter of capitalistic forces and no group or individual has seen the right technology to produce sufficient revenue to justify and overcome the development costs of producing a useful and effective tool.

Regardless of the reason for the slow uptake of these emerging technologies to support concurrent programming, we as researchers are obliged to be more scientific in our quest to tackle the problem of test, analysis, and debug of such programs. We ought to have a common set of problems for which we produce results using our various techniques, and the problems in the set are sufficient to rationally compare competing technologies. Such a commonality benefits the researcher, the business person, and the developer as decisions can now be made from a common reference point.

As one of many first steps (by ourselves and other groups) in the direction of commonality, this paper presents our modest efforts to produce a benchmark suite of concurrent programs for multiple programming languages with results from several tools. Although our focus in the benchmark suite is detecting shared memory concurrency errors in the *testing* mindset, such a suite can be used for analysis and debug as well. Our benchmark suite builds on the Dwyer FSE 2006 benchmark suite, [Dwyer et al., 2006], and the IBM benchmark suite, [Eytani and Ur, 2004, Eytani et al., 2007], by adding new models including C# versions of many of the models. We also follow the pattern of the DiVinE tool from the Paradise labs, [Pelanek, 2007], in that we publish the results for the benchmarks; only we also include results from other tools. Unlike other efforts, we also make the raw data available for mining, and we have put everything in a public repository where other researchers can contribute as appropriate. To demonstrate the value of such a common reference point, we present a small empirical study on the data we have thus far collected that motivates guidance strategies and randomization to improve error discovery in dynamic software model checking. The main contributions of this paper are

i) **Multi-language Benchmarks**: we have taken our set of Java benchmarks compiled from academia, IBM, and our own efforts and created many equivalent C# versions. All of the benchmarks are available to download. Such a multi-language collection of benchmarks is important to understanding and evaluating different technologies for detecting concurrency errors.

ii) **Multi-tool Results**: for select models in our multi-language benchmark suite we have results for `CalFuzzer`, `ConTest`, `CHESS`, and `Java PathFinder`. For `Java Pathfinder` we provide extensive results for stateless random walk, randomized depth-first search, and guided search using abstraction refinement. We are working on results for Inspect which is a dynamic model checker for C programs using pthreads. Such a repository of raw data facilitates more rigorous data analysis for future technologies and the needs of other researchers in the area.

iii) **On-line Resource**: we have created a wiki to publish benchmark details and data from various tools on those benchmarks to a broader research community. Such an on-line publication encourages researchers to compare emergent technologies for detecting concurrency errors to current state of the art. It also identifies the strengths and weaknesses of such technologies.

iv) **Empirical Support of Randomization and Guidance**: using the data from our study we provide a modest empirical analysis showing the merits of randomization and guidance in improving error discovery in dynamic model checking. We further argue that techniques such as iterative context bounding and dynamic partial order reduction are not sufficient to render model checking tractable and secondary techniques such as guidance strategies are required if model checking is ever to be practical in mainstream development. Further results to support this claim are found on the on-line resource.

5.2 Benchmarks

A set of 45 unique Java programs has been collected from various sources [Dwyer et al., 2006, Eytani and Ur, 2004, Rungta and Mercer, 2008, Sen, 2008, Visser et al., 2000a]. The benchmarks encompass a wide variety of Java programs and concurrency errors. Program derived from concurrency literature, small to medium sized realistic programs obtained from `sourceforge`, models developed at IBM to support their analyses research, and programs designed to exhibit patterns of concurrency bugs usually found in real-world programs. The programs have been parameterized in order to control the number of threads in the program. This allows us to study the effectiveness of the error discovery tool or technique as the number of threads increase in the program.

One of the contributions of this work is that we have created a set of C# programs structurally corresponding to many of the Java programs. We have C# programs for 12 unique Java models. In each C# model corresponding to a Java program the same number of threads are created, similar data structures are instantiated, threads access the same data structures, and threads perform the same synchronization operations. Note that since Java and C# have very similar execution models we can recreate the same programs in both languages. The methods responsible for generating the various threads in the Java programs are used to create unit tests in the C# programs. The unit tests start the appropriate threads needed to execute the concurrent program. The C# programs that are created essentially have the same functionality and behavior as the original Java programs. For these models we also have multi-tool results.

The Java and their corresponding C# programs are available from a public repository. Furthermore, the data generated from the tests is also available. The details of how to obtain the benchmarks and data are available on a wiki location at: `http://vv.cs.byu.edu`

5.3 Multi-tool Results

In this section we present an overview of the various tools and techniques that are evaluated on a set of concurrent programs. The CHESS concurrency testing tool is used to check multi-threaded C# programs while all other techniques and tools are used to check multi-threaded Java programs. The random walk, randomized depth-first search (DFS), and abstraction guided refinement techniques are implemented and evaluated in the Java Pathfinder (JPF) model checker [Visser et al., 2000a] with dynamic partial order reduction turned on [Flanagan and Godefroid, 2005]. The JPF model checker is a modified Java virtual machine that provides the ability to systematically explore all possible thread schedules in concurrent programs. ConTest and CalFuzzer are dynamic analysis tools that rely on instrumentation to control the scheduling of concurrent programs.

Random Walk: Random walk is a stateless search technique [Haslum, 1999, Stoller, 2002]. Starting from the initial state of the program random walk randomly picks one of the possible successors of the current state to explore a sequence of states in the transition graph. The search continues until it either reaches a state with no successors (end state), an error state, or some user-specified depth-bound. Due to its stateless nature it does not store any information on previously visited states.

Randomized Depth-first Search: A randomized depth-first search (DFS) is a stateful search technique [Dwyer et al., 2006, 2007]. Similar to random walk, a randomized DFS explores a sequence of states starting from some initial state. At each state it randomly picks one successor to explore in a depth-first manner. In order to obtain better coverage of the transition graph, it maintains a set of *visited* states to track states that have been explored.

CHESS with iterative context-bounding: CHESS is a concurrency testing tool for C# programs [Musuvathi and Qadeer, 2008, 2007]. It systematically explores the various thread schedules *deterministically*. CHESS requires an idempotent unit-test that creates the requisite threads to test a piece of concurrent program. In an idempotent test, the number of threads running at the end of the test needs to be the same as the number of threads running before the start of the test. Furthermore all allocated resources are freed and the global state is reset. CHESS executes the unit test and explores a different schedule at iteration of the test. CHESS is also a stateless search technique that does not track any information on the states.

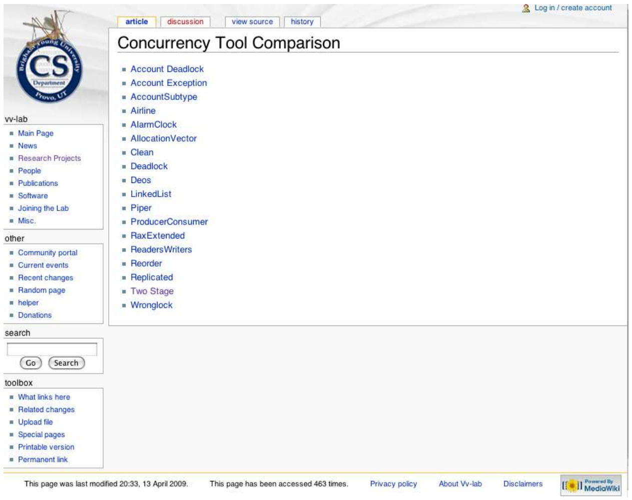
The iterative context-bounding approach bounds the number of preemptions along a certain path in order to reach the error faster [Musuvathi and Qadeer, 2007]. To further restrict the number of preemptions CHESS only considers preemption points simply at synchronization operations. In programs that contain data races, however, CHESS provides a knob for turning on preemptions at accesses to shared volatile variables.

Guided Search with Abstraction Refinement: The guided search using abstraction refinement attempts to verify the reachability of a set of target locations that represent a possible error in the program [Rungta et al., 2009, Chapter 2]. The input locations are either generated from imprecise static analysis warnings or user-specified reachability properties. An abstract system contains call sites, conditional statements, data definitions, and synchronization points in the program that lie along control paths from the start of the program to the target locations. The program execution is then guided toward the locations in the abstract system in order to reach the target locations. A combination of heuristics are used to automatically pick thread identifiers [Rungta and Mercer, 2008, 2009b, Chapter 3, Chapter 4]. At points in the execution when the program execution cannot be guided further along a sequence of locations (e.g. a particular conditional statement) the abstract system is refined by adding program statements that re-define the variables of interest.

Contest Contest is a concurrency testing tool for Java programs [Eytani et al., 2007]. It attempts to insert noise (randomness) at various synchronization operations while dynamically running the program. It uses a variety of heuristics to drive the schedules. It closely resembles random walk in both behavior and technique.

CalFuzzer: There are two parts to the tool. The RaceFuzzer uses an existing hybrid dynamic race detection to compute a pair of program states that could potentially result in a race condition [Sen, 2008]. RaceFuzzer randomly selects a thread schedule until a thread reaches one program location that is part of the potential data race pair. At this point the execution of the thread is paused at that program location, while the other thread schedules are randomly picked in order to drive another thread to the second program location that is part of the possible data race. When two threads reach the program location part of the data race and access the same memory location such that at least one of the accesses is a write operation then a real data race is reported. Similarly the DeadlockFuzzer uses the Goodlocks algorithm and locket set analysis, [Engler and Ashcraft, 2003, Havelund, 2000], to detect potential deadlocks in the program and randomly drives the threads toward the program locations similar to the abstraction guided search in JPF. CalFuzzer, however, uses a naive guidance strategy. It randomly picks thread schedules until the thread reaches a point of interest.

In a manner similar to CHESS, CalFuzzer is stateless. Also it only performs thread switches before synchronization operations and input program locations that represent the potential data race in the program. If the reachability of the program locations, however, depends on another data race in the program, the current implementation of CalFuzzer is unable to detect the error. To overcome this limitation



Concurrency Tool Comparison

- Account Deadlock
- Account Exception
- AccountSubtype
- Airline
- AlarmClock
- AllocationVector
- Clean
- Deadlock
- Deos
- LinkedList
- Piper
- ProducerConsumer
- RaxExtended
- ReadersWriters
- Reorder
- Replicated
- Two Stage
- Wronglock

This page was last modified 20:33, 13 April 2009. This page has been accessed 463 times.

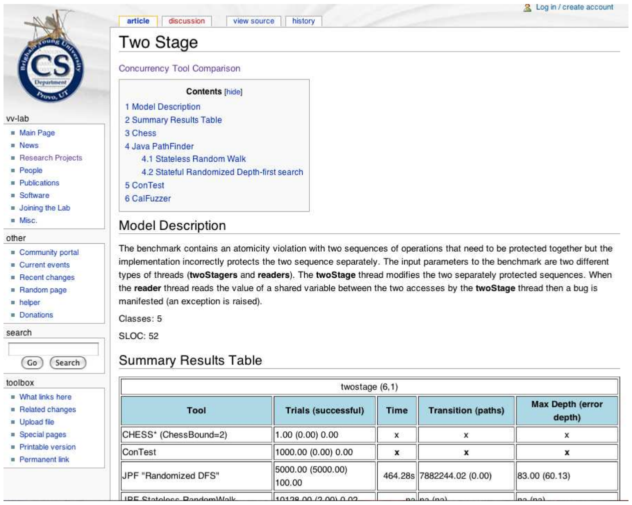
Figure 5.1: *Concurrency Tool Comparison* wiki containing benchmark details with multi-tool summary results and tool specific results: top level page showing the available models.

the tool has to insert preemption points at *all data races* in the program. The performance of CalFuzzer would most likely degrade considerably, if preemption points were added at all data races, for two reasons: (1) the overhead in the runtime of the analyses used to detect all data races and (2) the increase in the size of the transition graph resulting from the larger number of preemption points in the program.

5.4 On-line Resource

We have created a *concurrency tool comparison* wiki to publish details on each available benchmark including summary tables for results from each of the tools for which the benchmark has been run with tool specific tables showing more complete output. The wiki is located at: `http://vv.cs.byu.edu`

Figure 5.1 and Figure 5.2 show screen shots from the wiki. Figure 5.1 is the main screen listing benchmarks in the repository. Future work displays the main page as a table providing summary data on the benchmarks including location count, class count, thread count as a function of parameters, error type, and a notion of hardness. Hardness in our work is based on randomized DFS and is a ratio of error



Two Stage

Concurrency Tool Comparison

Contents [hide]
1 Model Description
2 Summary Results Table
3 Chess
4 Java PathFinder
4.1 Stateless Random Walk
4.2 Stateful Randomized Depth-first search
5 ConTest
6 CalFuzzer

Model Description

The benchmark contains an atomicity violation with two sequences of operations that need to be protected together but the implementation incorrectly protects the two sequence separately. The input parameters to the benchmark are two different types of threads (**twoStagers** and **readers**). The **twoStage** thread modifies the two separately protected sequences. When the **reader** thread reads the value of a shared variable between the two accesses by the **twoStage** thread then a bug is manifested (an exception is raised).

Classes: 5

SLOC: 52

Summary Results Table

twostage (6,1)				
Tool	**Trials (successful)**	**Time**	**Transition (paths)**	**Max Depth (error depth)**
CHESS* (ChessBound=2)	1.00 (0.00) 0.00	x	x	x
ConTest	1000.00 (0.00) 0.00	x	x	x
JPF "Randomized DFS"	5000.00 (5000.00) 100.00	464.28s	7882244.02 (0.00)	83.00 (60.13)
JPF Stateless RandomWalk	10128.00 (2.00) 0.02	na	na (na)	na (na)

Figure 5.2: *Concurrency Tool Comparison* wiki containing benchmark details with multi-tool summary results and tool specific results: an example of a model page.

finding randomized DFS trials divided by a total number of trials. The idea is that randomized DFS, at a minimum, should fail to find an error most of the time for a model to be considered *hard*.

Figure 5.2 is an example of the information found on a specific benchmark page. Each benchmark page includes a description with location count, class count, parameters, the number of threads as a function of parameters, and the type of error in the benchmark. It also includes different languages expressing the model. For the shown example, it exists in both Java and C# currently.

After the description comes a complete summary table displaying data from all the tools run on the benchmark. The summary data currently lists the tool, the trials run including successful trials in ()'s with the ratio of successful trials to total trials next to the ()'s (for deterministic tools the number of trials is 1), time in seconds, transition count with paths explored in ()'s, and max depth explored with error depth in ()'s. We are attempting to select summary statistics that would be applicable to most model checkers or systematic search tools; although, as noted in the introduction, there is no general consensus on what should be output from any given tool.

5.5 Empirical Study

We present a summary of the interesting results by evaluating the various tools and techniques on the set of benchmarks. In this empirical study we present the results on the multi-tools on three unique programs. A more extensive comparison across a larger set of benchmarks is provided on the wiki. We vary model, the independent variable, in our study to evaluate the effectiveness of the error discovery by the different techniques. For a particular model we also vary the number of threads and the kind of threads created by each model.

Time Bound: The randomized DFS trials and the CHESS trials are time bounded at one hour. This time bound was picked to be consistent with other studies [Dwyer et al., 2006, 2007, Rungta and Mercer, 2007a, Appendix B].

Number of Trials: Over the course of the last three years, we have conducted extensive studies [Rungta and Mercer, 2007a,b, Appendix B, Appendix A] on a set of benchmarks collected from [Dwyer et al., 2006]. To recreate the results from the parallel randomized stateful search (PRSS) technique, [Dwyer et al., 2007], we ran upto 5000 independent trials of randomized DFS each time bounded at one hour. To compare the random walk results reported in [Dwyer et al., 2006] for certain benchmarks we have run upto 10,000 independent random walk trials. For those benchmarks, we present the results based on the data that has already been collected. We have executed between 100 to 1000 trials for the other tools. We believe that this number is sufficient to compare the effectiveness of the different non-deterministic techniques. Increasing the number of trials for these tools is part of ongoing work.

Hardness: We measure the semantic measure of hardness in error discovery as described in [Rungta and Mercer, 2007a, Appendix B] in order to evaluate the error discovery abilities of the different tools and techniques. For a given benchmark and technique, the hardness is the ratio of the total number of trials executed over the total number of error discovering trials. Consider, for example, if 100 trials are executed and 50 trials are successful in finding an error then the hardness is reported as 0.50 or 50%. Non-deterministic algorithms like randomized DFS and random walk provide a range of values between 0 (hard) and 1 (easy), however, for deterministic algorithms the hardness is either 0 (hard) or 1 (easy).

In this study we compare the error discovery capabilities of stateful as well as stateless search techniques. In order to measure the effectiveness of error discovery we measure the number of transitions that were executed before error discovery, the maximum depth of the search, and the depth at which the error was found. Note that some tools such as ConTest and CalFuzzer do not output values for the measures. The number of transitions executed are reported to better compare stateful and stateless techniques.

In Table 5.1, Table 5.2, and, Table 5.3 we present the results for the `reorder`, `twostage`, and `airline` benchmarks respectively. The column labeled `Tool` indicates the tool used to evaluate the benchmark. ConTest is run with the default values where it randomly picks one of its heuristics for improving error discovery. The parameters of CHESS indicate whether volatile variables are monitored (V=true) and the preemption bound (B=num). Recall that preemptions at volatile variables are turned off by default. The randomized DFS, random walk, and abstraction guided search (Abs. Guided) are executed in JPF with dynamic partial order reduction turned on. The *na* entries in the table indicates that the particular output is *not available* for the particular tool. The `x` legend indicates that none of the trials found an error and, hence, there is no data to report.

Tool	Trials (Successful)	Error Dis. Rate	Time	Transition	Max Depth (Error Depth)
Reorder(2,1) ThreadNum=4					
CalFuzzer	100 (12)	12%	1.376s	na	na(na)
CHESS* (V=true, B=2)	1 (1)	100%	0.44s	2600	26 (na)
ConTest	1000 (23)	2.30%	na	na	na(na)
Randomized DFS	100 (100)	100%	0.61s	560.58	28.33 (21.24)
Random Walk	1000 (2)	0.20%	0.28s	27.50	27.50 (27.50)
Abs. Guided	1 (1)	100 %	1.67s	44	14 (14)
Reorder(1,5) ThreadNum=7					
CalFuzzer	100 (24)	0.24%	1.658s	na	na(na)
CHESS* (V=true, B=2)	1 (0)	0%	x	x	x
ConTest	1000 (285)	0.28%	na	na	na(na)
Randomized DFS	100 (100)	100%	16.61s	238450.80	52.94 (25.52)
Random Walk	10128 (438)	4.32%	0.35s	32.41	32.41 (32.41)
Abs. Guided	1 (1)	100%	2.67s	29	12 (12)
Reorder(4,1), ThreadNum=6					
CalFuzzer	100 (12)	0.12%	1.45s	na	na(na)
CHESS* (V=true, B=2)	1 (1)	100	140.53s	1200000	40
ConTest	1000 (3)	0.3%	na	na	na(na)
”Randomized DFS”	100 (100)	100%	7.35s	33895.58	46.61 (34.29)
Random Walk	1000 (0)	0%	x	x	x
Abs. Guided	1 (1)	100%	1.67s	80	18 (18)
Reorder(9,1) ThreadNum=11					
CalFuzzer	100 (10)	0.10%	1.49s	na	na(na)
CHESS* (V=true, B=2)	1 (0)	0%	x	x	x
ConTest	1000 (10)	0.01%	na	na	na(na)
Randomized DFS	5000 (593)	11.86%	2497.98s	38704112.56	81.67 (59.64)
Random Walk	10124 (0)	0%	x	x	x
”Abs. Guided”	1 (1)	100%	1.67s	205	28 (28)
Reorder(10,1) ThreadNum=12					
CalFuzzer	100 (9)	0.09%	1.49s	na	na (na)
CHESS* (V=true, B=2)	1 (0)	0%	x	x	x
ConTest	1000 (0)	0%	x	x	x
Randomized DFS	5000 (4)	0.08%	2414.82s	38022689.25	89.75 (65.25)
Random Walk	10127 (0)	0%	x	x	x
Abs. Guided	1 (1)	100%	1.67s	236	30 (30)

Table 5.1: Comparing the error discovery of different techniques on the `reorder` benchmark.

5.5.1 Reorder

The benchmark contains a data race. There is a check whether the data race causes an inconsistency in the data values. When such an inconsistency is discovered an exception is raised. The benchmark contains two kinds of threads: `setter` and `getter` that cause a data race in the program. The `getter` thread also checks for the inconsistency in the data values caused by the data race. As we increase the number of `setter` threads while keeping the `getter` thread constant, the semantic measure of the hardness in error discovery increases (i.e., get harder) as shown in [Rungta and Mercer, 2007a, Appendix B].

CHESS is effective for error discovery in the `reorder` model with a small number of threads as shown in Table 5.1. The data race in the model can be manifested with a preemption bound of two (B=2). We also monitor volatiles variables (V=true). As the number of threads increases even with a preemption bound of two, CHESS progressively takes more time to find the error. In Table 5.1 we notice that CHESS is unable to find an error in `reorder(9,1)` and `reorder(10,1)` in a time bound of one hour.

The error discovering capability of randomized DFS search degrades when the number of threads are increased. In the `reorder(10,1)` model only 8 trials out of 5000 were able to find an error and, on average, took 2414.82 seconds to find an error as shown in Table 5.1. Random walk and ConTest are only able to find an error in `reorder(2,1)` and `reorder(1,5)` models.

CalFuzzer and the abstraction guided search are most effective in finding errors as the number of threads increase in the `reorder` model. CalFuzzer is more effective than randomized depth-first search because it only considers preemptions points at the pair of program locations that are reported to be part of a data race. In the `reorder` model there are no synchronization operations, hence, for CalFuzzer there only exist two points of preemption. The abstraction guided search is able to find the error quickly even when the model checker considers preemption points at all shared variable accesses.

5.5.2 TwoStage

The benchmark contains an atomicity violation. The program has sequences of operations that need to be protected together but the implementation incorrectly protects the two sequences separately. The input parameters to the benchmark are two different types of threads (`twoStagers` and `readers`). The `twoStage` thread modifies the two separately protected sequences. When the `reader` thread reads the value of a shared variable between the two write accesses by the `twoStage` thread then a bug is manifested (an exception is raised).

CalFuzzer is unable to find a set of input location using their initial dynamic analysis. In the current configuration of the tool we are unable to manually specify the input location to CalFuzzer. The input target location is the program location where the exception is raised. This is to check whether the exception is ever raised. The ConTest tool is unable to find the bug in 1000 trials for any configuration of the model. Random walk is also not very effective in error discovery. The error discovery rate for `twostage(1,1)` is only 4.25%.

CHESS is effective in error discovery for the `twostage(1,1)` and `twostage(1,2)` models where there are a small number of threads. The `twostage(1,1)` has 3 total threads while `twostage(1,2)` has 4 threads. As, however, the number of threads in the benchmark increase, CHESS is less effective in error discovery. Note that in `twostage`(2,2) the average time taken by randomized DFS to find the error is only 2.41 seconds while CHESS takes 17.44 seconds to find the error.

The abstraction guided search does not degrade as we increase the number of threads in the program. So while it takes more time to detect the error in the benchmark when there is a small number of threads (such as `twostage(1,1)`) compared

Tool	Trials (Successful)	Error Dis. Rate	Time	Transition	Max Depth (Error Depth)
TwoStage(1,1), ThreadNum=3					
CHESS* (B=2)	1 (1)	100%	0.94s	180	20 (na)
ConTest	1000 (0)	0%	x	x	x
Randomized DFS	100 (100)	100%	0.61s	88.04	20.18 (18.51)
Random Walk	10127 (430)	4.25%	0.02s	0.94	0.94 (0.94)
Abs. Guided	1 (1)	100%	3.53s	30	11 (11)
TwoStage(1,2), ThreadNum=4					
CHESS* (B=2)	1 (1)	100%	0.52s	5800	29 (na)
ConTest	1000 (0)	0%	x	x	x
Randomized DFS	100 (100)	100%	1.07s	1966.10	38.46 (30.08)
Random Walk	1000 (10)	1%	0.30s	30.20	30.20 (30.20)
Abs. Guided	1 (1)	100%	3.53s	46	13 (13)
TwoStage(2,2), ThreadNum=5					
CHESS* (B=2)	1 (1)	100%	17.44s	228000	38 (na)
ConTest	1000 (0)	0%	x	x	x
Randomized DFS	100 (100)	100%	2.41s	10223.50	45.72 (36.07)
Abs. Guided	1 (1)	100%	3.53s	46	13 (13)
TwoStage(1,5), ThreadNum=7					
CHESS* (B=2)	1 (0)	0%	x	x	x
ConTest	1000 (0)	0%	x	x	x
Randomized DFS	100 (100)	100%	319.01s	3341784.11	73.96 (38.09)
Abs. Guided	1 (1)	100%	2.53s	30	11 (11)
TwoStage(8,1), ThreadNum=10					
CHESS* (B=2)	1 (0)	0%	x	x	x
ConTest	1000 (0)	0%	x	x	x
Randomized DFS	4999 (126)	2.52%	2155.76s	32969192.09	105.46 (73.40)
Random Walk	10200 (0)	0%	x	x	x
Abs. Guided	1 (1)	100%	2.53s	184	25 (25)

Table 5.2: Comparing the error discovery of different techniques on the `twostage` benchmark.

to the other techniques, the time stays consistent even as we increase the number of threads. In benchmarks where increasing the number of threads does not change the conditions that cause the error, the abstraction guided search is consistent in its performance showing little change in running time.

5.5.3 Airline

The benchmark contains a data race. As the number of threads are created the value of a global variable is updated. An incorrect assumption on atomicity allows more threads to be created leading to a data inconsistency. The two parameters to the `airline` model are: `ticketsIssued` and `cushion`. The minimum depth of the error is pushed deeper in the execution trace when we increase the value of the `cushion` and keep the total number of threads, `ticketsIssued`, constant.

The reachability of the error state in the `airline` model depends on the value of a global counter that is modified by different threads. Based on the input parameters of the model, a different number of preemptions is required to elicit the error in the airline model. In Table 5.3 for the `airline(20,1)` model, 18 preemptions are required to elicit the error. Even after setting the correct number of preemptions, CHESS is unable to discover the error in a time bound of one hour. The need to specify the correct preemption bound is another limitation of the iterative context-bounding approach.

Recall that CalFuzzer only inserts preemption points at synchronization points and the input pair of program locations that represent a potential data race in the program. In the `airline` model, CalFuzzer is unable to find the error in the benchmark because a specific number of preemptions at unprotected global variables are required to elicit the bug.

The abstraction guided search technique can successfully find the error in the model. In the refinement process the location of setting the global variable is

Tool	Trials (Successful)	Error Dis. Rate	Time	Transition	Max Depth (Error Depth)
Airline(4,1), ThreadNum=5					
CalFuzzer	100 (0)	0%	x	x	x
CHESS* (V=true, B=2)	1 (1)	100%	231.240s	5,200,425	114 (na)
CHESS* (V=true, B=3)	1 (1)	100%	1982.522s	44,916,000	114 (na)
Randomized DFS	200 (200)	100%	0.81s	5776.74	14.28 (13.23)
Random Walk	1011 (98)	9.69%	0.25s	22.85	22.85 (22.85)
Abs. Guided	1 (1)	100%	3.46s	61	17 (17)
Airline(4,2), ThreadNum=5					
CalFuzzer	100 (0)	0%	x	x	x
CHESS* (V=true, B=2)	1 (1)	100%	67.533s	1,482,000	114 (na)
Randomized DFS	200 (200)	100%	0.25s	655.86	13.75 (12.59)
Random Walk	1021 (433)	42.41%	0.19s	19.05	19.05 (19.05)
Abs. Guided	1 (1)	100%	3.46s	46	14 (14)
Airline(5,1), ThreadNum=7					
CalFuzzer	100 (0)	0%	x	x	x
CHESS* (V=true, B=2)	1 (0)	0%	x	x	x
CHESS* (V=true, B=3)	1 (0)	0%	x	x	x
Randomized DFS	200 (200)	100%	58.19s	670929.49	25.27 (22.17)
Random Walk	1021 (111)	10.87%	0.39s	28.75	28.75 (28.75)
Abs. Guided	1 (1)	100%	3.46s	98	21 (21)
Airline(20,8), ThreadNum=21					
CalFuzzer	100 (0)	0%	x	x	x
CHESS* (V=true, B=2)	1 (0)	0%	x	x	x
Randomized DFS	5000 (2507)	50.14%	285.85s	4111966.14	121.07 (118.62)
Random Walk	10123 (696)	6.88%	0.09s	12.98	12.98 (12.98)
Abs. Guided	1 (1)	100%	5.46s	2680	60 (60)
Airline(20,3), ThreadNum=21					
CalFuzzer	100 (0)	0%	x	x	x
CHESS* (V=true, B=2)	1 (0)	0%	x	x	x
Randomized DFS	4992 (24)	0.48%	375.78s	5905884.67	121.75 (119.75)
Random Walk	11132 (19)	0.17%	0.44s	110.63	110.63 (110.63)
Abs. Guided	1 (1)	100%	7.46s	3210	75 (75)
Airline(20,1), ThreadNum=21					
CalFuzzer	100 (0)	0%	x	x	x
CHESS* (V=true, B=18)	1 (0)	0%	x	x	x
Randomized DFS	4996 (0)	0%	x	x	x
Random Walk	11125 (0)	0.00%	x	x	x
Abs. Guided	1 (1)	100%	7.46s	3609	95 (95)

Table 5.3: Comparing the error discovery of different techniques on the `airline` benchmark.

iteratively (and automatically) added. This enables the abstraction guided search to find the error even when a specific sequence of preemptions are required. For example, `airline(20,1)` requires the most number of refinements, hence, its total time for error discovery is 7.46 seconds while the model `airline(4,1)` that requires fewer refinements takes only 3.46 seconds. The results are encouraging since the other techniques struggle to find an error in the model within significant constraints of time and memory.

5.5.4 Discussion

The performance of CHESS is hindered due to its deterministic nature. Even in the presence of iterative context-bounding CHESS is severely limited by the benefits and limitations of default search order. With a systematic randomization implemented within CHESS, we believe it can outperform randomized DFS in models that require only one or two preemptions along an execution path to reach an error location. In large programs, however, where the errors exists along very few paths in the transition graph, iterative-context bounding is not likely to pick the schedule that leads to an error state (even with randomization).

The performance of ConTest is comparable to that of random walk. A more systematic random search such as randomized depth-first search is in general more effective for error discovery within the same constraints of time and memory. At a certain point, however, simple randomized search techniques fail to find an error. The data from our empirical study demonstrate that iterative context-bounding and dynamic partial order reduction are not sufficient to find errors in concurrent programs. There is a need for a more sophisticated guidance such as the guided search with abstraction refinement that consistently performs well across the different benchmarks in this study.

5.6 Related Work

Various efforts have been made to compile a set of benchmarks for concurrent programs [Eytani and Ur, 2004, Eytani et al., 2007]. The programs collected exhibit a variety of concurrency errors. The benchmark suite has multi-threaded programs with documented bugs. The annotations about bugs in the program also helps evaluate imprecise static and dynamic analysis technique in determining whether the warning on a possible error is a false positive. This work attempts to take the idea of a benchmark suite one step further by having multi-language programs and results from different tools on the programs.

The BEEM—BEnchmarks for Explicit Model Checkers is a benchmark set that contains 50 parameterized models along with with their correctness properties [Pelanek, 2007]. For a given model, an instance generator can generate models in the DiVinE specification language (DVE) as well as Promela, the input language for the SPIN model checker [Holzmann, 2003]. Our work here attempts to provide the same resource for programs targeted to software model checking rather than pure model checking.

The work on defining a semantic hardness measure for concurrent programs using randomized depth-first search has allowed us to test new techniques on models where exhaustive and randomized techniques fail to find an error [Rungta and Mercer, 2008, 2007a, Rungta et al., 2009, Appendix B, Chapter 3, Chapter 2]. Our experience shows that the hardness measures generated by a randomized depth-first search with partial order reduction turned on provides a good threshold that needs to be overcome by a new technique or tool in order for that technique or tool to be considered effective in error discovery.

5.7 Conclusion

We present in this work a modest attempt to bring commonality and a sure reference point to research for the test, analysis, and debug of concurrent programs. Our contribution is in the form multi-language benchmarks, multi-tool results, an online resource for broader impact, and a empirical study showing the merits of various techniques. All of the models are open to other researchers, and we hope other are able to contribute to the work. An example of the effectiveness of such a common reference for research is in our empirical study. Using the data from our study show that techniques such as iterative context bounding and dynamic partial order reduction are not sufficient to render model checking tractable and secondary techniques such as guidance strategies are required if model checking is ever to be practical in mainstream development.

We intend to continue to publish benchmarks and data on the wiki. Immediate future work is to automate table construction from the raw data using various scripts. We are also working on a summary table showing all the available benchmarks as the opening main page rather than the simple list of benchmarks. We hope to continue to produce C# models as appropriate. The next target tool is Inspect requiring C pthread models. We are also looking to begin exploring and publishing results from automated test tools like `jCUTE` [Sen and Agha, 2007]. `jCUTE` is a concolic testing framework for concurrent programs using a dynamic partial order reduction.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The guided test technique presented in this dissertation automatically detects errors in concurrent programs. It is a systematic test technique that uses information derived automatically from the source of the programs. Imprecise static analysis techniques identify possible errors in the program, and other program locations relevant in determining the feasibility of a possible error in the program. Program execution is guided along the relevant program locations to determine the feasibility of an error. A combination of heuristics and stochastic methods have been developed to efficiently guide program execution. The heuristics used in guided test rank, both, thread and data non-determinism to find errors in concurrent programs. The set of relevant program locations is refined by adding new program locations when the execution is unable to make progress.

The guided test solution is effective in error discovery in programs where other existing techniques fail. To evaluate the effectiveness of the guided test technique we used the algorithm to detect errors caused by thread schedules and data input values in Java programs. The algorithm was able to efficiently detect errors in Java benchmarks and the JDK concurrent libraries where other state of the art analysis and testing tools for concurrent programs are unable to find an error.

To further compare the effectiveness of the guided test solution with CHESS, a stateless concurrency tool for C# programs, we created C# programs corresponding to the Java benchmarks. The C# programs are structurally similar to the Java programs. In each corresponding model the same number of threads are created, similar data structures are created, the threads perform the same accesses on the data structures, and the threads perform the same synchronization operations. Guided test can quickly detect the bugs within a few seconds while CHESS is unable to find an error within a time bound of one hour.

By effectively automating the process of bug detection in concurrent programs, we can significantly improve the efficiency of software testing and verification in general. The broader impact of this result is seen first and foremost in software being more reliable and correct. Reliability is not the only impact, however, as improving efficiency in verification reduces overall cost in software development which hopes to save the end-user money in licensing. Also, as designers are able to use tools that can efficiently test concurrent programs, they are more likely to be aggressive in the manner in which they use concurrency to improve performance. The end result is that consumers are going to receive a more reliable product at a lower cost while the developers are provided with better verification and testing solutions.

6.2 Future Work

An efficient implementation of the technique that can be used out of the box by a non-model checking expert on production code represents a significant software engineering challenge. The immediate next step is to test a larger set of benchmarks and identify the areas where the algorithm or the implementation can be further improved.

The current algorithm is restricted to detecting errors in programs with explicit synchronization operations. One of the software engineering challenges is to extend the algorithm such that it is applicable to a varied set of synchronization constructs. For example, a large number of concurrent applications use wait and notify operations, atomic references, and lock-free synchronization operations; in future work we want to effectively find errors in such programs.

We believe that guided test approach can allow us to state the infeasibility of certain static analysis warnings, in other words, automatically identify certain false positives generated by static analysis tools. In order to automatically identify false positives we would develop a modular partial order-reduction to prune all interleavings from dependencies that are not relevant in testing the feasibility of a given *static analysis warning* or *reachability property*. Developing the property-specific partial order-reduction technique might allow us to state functional correctness of concurrent programs with respect to a certain static analysis warning or reachability property. This will provide the guided testing technique the ability to guarantee the absence of certain errors.

Using the modular partial order reduction, if we detect a relevant location as unreachable and through transitivity the error as infeasible, we can efficiently detect the absence of certain errors. We can hope to accomplish this result due to the modular nature of the abstraction refinement process presented in this dissertation that focuses verification of certain static analysis warnings and possible errors.

In case we are unable to conclusively state the feasibility of the error, a coverage measure is critical in quantifying the test effort expended in the verification process. Currently there are no standardized coverage metrics for concurrent programs. The existing coverage measures such a branch-coverage and predicate coverage are insufficient for measuring test effort in concurrent programs or making any claims about the functional correctness of the program. There is a need to design and measure a metric for concurrent programs in identifying gaps in coverage in terms of thread schedules with respect to a particular error in the program. In general we recognize that it is not possible to obtain complete coverage in a concurrent system, however, we need a measure that quantifies the verification for test effort to identify coverage gaps.

Appendices

Appendix A

Randomization in Guided Execution

This empirical study was published as:

N. Rungta and E. G. Mercer, "Generating Counter-examples through Randomized Guided Search", in *Proceedings of the SPIN Workshop on Model Checking of Software*, Berlin, Germany, pages 39-57, July 2007

Abstract

Computational resources are increasing rapidly with the explosion of multi-core processors readily available from major vendors. Model checking needs to harness these resources to help make it more effective in practical verification. Directed model checking uses heuristics in a guided search to rank states in order of interest. Randomizing guided search makes it possible to harness computation nodes by running independent searches in parallel in a effort to discover counter-examples to correctness. Initial attempts at adding randomization to guided search have achieved very limited success. In this work, we present a new low-cost randomized guided search technique that shuffles states in the priority queue with equivalent heuristic ties. We show in an empirical study that randomized guided search, overall, decreases the number of states generated before error discovery when compared to a guided search using the same heuristic. To further evaluate the performance gains of randomized guided search using a particular heuristic, we compare it with randomized depth-first

search. Randomized depth-first search shuffles transitions and generally improves error discovery over the default transition order implemented by the model checker. In the context of evaluating randomized guided search, a randomized depth-first search provides a lower bound for establishing performance gains in directed model checking. In the empirical study, we show that with the correct heuristic, randomized guided search outperforms randomized depth-first search both in effectively finding counter-examples and generating shorter counter-examples.

A.1 Introduction

The current trend in micro-processor design is to group multiple processors into a single silicon die and package. For example, dual-core processors are quickly becoming mainstream, and quad-core packages are readily available from most vendors. CEO Paul Otellini, at a recent Intel development forum, displayed an 80 core prototype chip capable of terabyte per second data exchange and pledged production runs in the next five years [Krazit, 2006]. The trend is clearly to put more processors on a single die rather than to increase clock speed and computation in a single processor. This is leading to an explosion in computational resources.

The question for the model checking community given the growth in multicore processors, as well as parallel and distributed systems, is how can we harness this computation power? At the heart of explicit state model checking is an exhaustive proof to show the absence of a specific behavior. The proof literally enumerates, in a largely brute-force manner, the entire behavior space of the system being verified [Clarke et al., 1986]. The complexity of the systems, however, limits practical application of model checking in both time and space. Aggregating the available computation resources to solve the model checking problem can help to improve the situation.

Parallel and distributed model checking has shown some limited promise in utilizing large amounts of computation resources [Barnat et al., 2001, Brim et al., 2006, Holzmann, 2006, Inggs and Barringer, 2006, Jabbar and Edelkamp, 2006, Stern and Dill, 1997]. The focus of the community is to find ways to harness several computation nodes to cooperatively construct the exhaustive proof. These approaches generally look appealing in low node counts but are less efficient as more computation nodes are added [Jones et al., 2003]. Seminal work goes so far as to prove that depth-first search itself is inherently sequential and does not lend itself to parallel computation [Reif, 1985]. This may explain the lack of scaling in current approaches and possibly suggest that we need a fundamentally different algorithm for model checking that is less sequential and more amenable to parallelization.

As a counterpoint, it is possible to parallelize model checking by moving away from an exhaustive proof and instead focus on counter-example generation. In other words, run several independent experiments with some degree of randomization on individual computation nodes to find a counter-example to the proof. This is in contrast to several computation nodes cooperatively constructing an exhaustive proof. The shift in focus from exhaustive proof to counter-example generation began in the directed model checking community, and it opens new avenues for distributed model checking.

Early researchers of parallel and distributed model checking explored the concept of random walk for counter-example generation with modest success [Haslum, 1999, Jones and Sorber, 2005, Sivaraj and Gopalakrishnan, 2003]. Random walk has inherently low memory requirements, and the work distributes these random walk based searches over many computation nodes in hopes of discovering a counter-example. The effectiveness of random walk in terms of coverage is critically dependent on the structure of the model [Barnat et al., 2006, Holzmann, 1997, Pelanek et al., 2005]. Empirical studies show that random walk is not very useful for error discovery in the models where it achieves poor coverage. This creates a need for effective randomized searches which better harness the computation resources.

Recent work studying default search order in model checker performance contributes a key insight to randomization of a regular depth-first search [Dwyer et al., 2006]. Controlling for default search order in depth-first search by randomly choosing transitions to explore (randomized DFS) dramatically improves counter-example generation [Dwyer et al., 2007]. Independent randomized DFS searches easily distribute to any number of computation nodes, however, like any search method, randomized DFS breaks down in certain models [Rungta and Mercer, 2007c]. The issue in randomized DFS is that it blindly moves through the behavior space even when there is information readily available about the structure of the model and the property being invalidated that can improve the search.

Directed model checking uses heuristics to rank interest in states and guide the search of the behavior space to efficiently generate counter-examples [Edelkamp and Jabar, 2006, Edelkamp et al., 2001a,b, Groce and Visser, 2002a, Pasareanu et al., 2003, Rungta and Mercer, 2006, Seppi et al., 2006, Yang and Dill, 1998]. The heuristics generally consider either the model structure or the property being validated to rank the states. A guided search then orders the states in a priority queue based on the path cost and heuristic ranking where states estimated to lead more quickly to a counter-example are explored before other states. Guided search is effective in counter-example generation and often succeeds where depth-first search fails. More importantly, the length of the counter-examples generated by guided search algorithms are often shorter than those generated by depth-first search. This simplifies the developer's task of understanding the counter-example.

Guided search also benefits from randomization, and like depth-first search, once randomized, it can be run independently in parallel (randomized GDS[1]). Preliminary work in randomized GDS chooses randomly from the first n-best entries of the priority queue when selecting the next state to explore [Jones and Mercer, 2004]. The effectiveness of the randomization is not clear from the empirical study. In some instances, the randomization helps; while in other instances, the randomization hurts. The control, n, in [Jones and Mercer, 2004] only ranges over a limited set of values between two and five, and the algorithm also does not distinguish between states in the priority queue with different heuristic values. In Java PathFinder v4.0 (JPF), it is also possible to execute a randomized GDS by randomizing the transition order in generating successors before adding them to the priority queue. This random-

[1] We use randomized GDS to refer generally to any algorithm that adds randomization into guided search, and we will clearly indicate how the search is randomized in the context in which it appears.

ization, however, has very limited impact on the actual default search order in the guided search. Clearly, there are several open questions in randomized GDS left to be explored.

This paper presents a new randomized GDS algorithm that completely shuffles states in the priority queue with equal heuristic rankings. We show that full randomization of the guided search improves the effectiveness of the search over default search order in an empirical study. The empirical study uses characterized benchmarks from [Dwyer et al., 2006, Rungta and Mercer, 2007c] and published heuristics for the JPF, [Visser et al., 2000b], and Estes, [Mercer and Jones, 2005], model checkers. This paper also presents a second empirical study on the new randomized GDS algorithm in context of randomized DFS using the previously mentioned models and heuristics. The second study highlights the role of the heuristic in performance. When the heuristic is correctly matched to the models and properties, the new randomized GDS algorithm outperforms randomized DFS in both the effectiveness of the search in finding counter-examples and the length of the counter-examples. When the heuristic is not correctly matched to the models or properties, randomized DFS is more effective in error discovery which demonstrates a need to develop better heuristics for those classes of models and properties.

The algorithm and empirical studies in this paper underscore a need to develop methods that match heuristics to models and the properties being disproved. This work and other work such as [Jones and Mercer, 2004] and [Dwyer et al., 2007] also revisit a new way to view randomization, model checking, and search techniques. It motivates a need to study and understand how to best use randomization in model checking and parallelization for counter-example generation. Research in this area is especially timely given the rapid increase in computational resources, and more importantly, the ever increasing need for practical model checking in system design.

A.2 Background

It is important to control for default search order when evaluating model checking algorithms because implementation details in the model checker itself affect performance to a larger degree than previously supposed [Dwyer et al., 2006]. For example, in a simple depth-first search, the state at the top of a search stack may have several enabled transitions that move the current state to the next state of computation. The choices arise from non-determinism in the model, where the non-determinism is usually a result of scheduling decisions or input locations. The principle observation in [Dwyer et al., 2006] is that controlling for the default order in which a model checker selects transitions during depth-first search dramatically affects the outcome of counter-example generation. The work in [Dwyer et al., 2006] proposes a randomized DFS that controls for default transition order by shuffling transitions enabled at each state. Follow-on work in [Dwyer et al., 2007] shows that randomized DFS is effective in counter-example generation across their benchmark set[2]. In the words of [Dwyer et al., 2006], "*[T]hese findings tell a strong cautionary tale*", because default search order significantly affects performance of the techniques being evaluated in comparison studies. This is especially critical for directed model checking which relies on comparison studies to establish performance gains.

Directed model checking uses a guided search rather than depth-first or breadth-first search to find counter-examples for the property being verified. The fundamental assumption is that an error does exist in the model, and the goal is to find the error before exhausting computation resources. The work in this paper focuses on a greedy best-first search; although, the ideas are equally applicable to other best-first search techniques that make no guarantee on the optimality of the counter-example. In other words, the results of an A^* search are not significantly affected by our approach. A

[2] There are other default orders in model checkers that are yet to be controlled as evidenced in [Rungta and Mercer, 2007c], where different versions of JPF yield different results in randomized DFS.

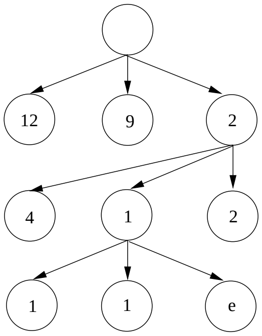


Figure A.1: An illustration of greedy best-first search that chooses the state nearest to the goal state to expand in the search based on a heuristic function.

greedy best-first search is illustrated in Figure A.1. The top state in Figure A.1 is the initial state. At each iteration of the search, a state is removed from a priority queue, its successors are generated, ranked by a heuristic function, and inserted into the priority queue. For example, the initial state in Figure A.1 has three successors which are ranked 12, 9, and 2. These states are inserted into the priority queue. The next iteration of the search removes the state with rank 2 from the priority queue and repeats the process. The heuristic function estimates the nearness of a state to an actual goal state. The goal state in our example is marked with the 'e' character. The goal state in directed model checking is an error state from which we build a counter-example to the specified property. A good heuristic for a greedy best-first search often converges quickly to an error state, and the length of the counter-example is near minimal.

Directed model checking critically relies on empirical studies to show performance gains over depth-first search, and like depth-first search, must control for default search order. For example, consider a priority search queue that contains over 100,000 states and a heuristic function that assigns an integer value between one and six to each state. Invariably, there are many thousand states with equivalent heuristic values. The order in which they are explored is largely controlled by the order in which they are generated by the model checker and ordered in the priority queue. During a guided search, some function compares the heuristic value of a newly generated state to the heuristic values of existing states in the queue before inserting the new state in the queue based on its ranking. Most often, this function uses a predetermined ordering to sort states that have the same heuristic value. For example, when comparing a newly generated state, s_1, with a heuristic value, x, to an existing state in the priority queue, s_2, with a heuristic value, x, the state ordering function always inserts state s_1 after s_2 in the priority queue. The order in which states s_1 and s_2 are explored can potentially affect the total number of states generated before error discovery—a fact disregarded by the ordering function. The lesson from [Dwyer et al., 2006] is that these default choices in the model checker need to be controlled. This gives rise to randomized GDS which in the context of this paper refers to a greedy best-first search with some randomization to control for default order.

There are several ways to implement randomized GDS, and each controls for default order in the priority queue to a certain extent. For example, [Jones and Mercer, 2004] randomly chooses between the n-best entries in the priority queue, and JPF v4.0 allows the transition order to be shuffled during state generation. The former method shows some potential while the later method is not effective in randomization. This paper presents a new algorithm for randomized GDS that controls for all heuristic ties in the priority queue. We show that with the correct heuristic function, our new algorithm for randomized GDS outperforms not only the greedy best-first search using default ordering but randomized DFS as well. This is especially true in models that are hard—that is, models where randomized DFS is not successful.

A.3 Randomized GDS

Current techniques for randomization of guided search are not effective in exploiting the full potential of the randomization. For example, as mentioned previously, the approach presented in [Jones and Mercer, 2004] limits the randomization to the n-best entries in the priority queue, where n is specified by the user. As another example, JPF allows for randomization in its searches. To understand its approach, we need to first look at its priority queue implementation; specifically, the `DefaultComparator` class. The class uses state identifiers and hash values to resolve heuristic ties between states in the priority queue. The state identifiers and hash values map to the same states in every single run of a guided search and deterministically resolve the heuristic ties. Turning on the `randomize_choices` option in JPF successfully modifies the order in which successors, for a particular state, are added to the priority queue because the successors are now assigned different state identifiers every time we execute a guided search trial. This randomized GDS approach causes only a small amount of variance in the number of states generated before error discovery when compared to the guided search since the randomization is limited to the successors of a given state. Our studies show that the limited amount of randomization is not effective in significantly changing the default search order.

To fully exploit the potential of randomization in directed model checking we define a randomized GDS algorithm that randomly shuffles states with equivalent heuristic ranking in the priority queue. The pseudo-code for this algorithm is presented in Figure A.2. The algorithm is *embarrassingly parallel* [Foster, 1995]. Several trials of the new randomized GDS algorithm can be launched in parallel on different computation nodes since each randomized GDS trial is completely independent of the other trials. There is no communication overhead between the trials which allows the algorithm to scale up to an arbitrary number of computation nodes.

In the randomized GDS algorithm, we associate a random value with each state generated during model checking in addition to its heuristic value. The tuple $\langle s_i, h_i, r_i \rangle$ in Figure A.2 is an element stored in the priority queue where s_i is the state, h_i is the heuristic ranking of s_i, and r_i is the random value associated with s_i. The randomized GDS algorithm employs a new comparator function, compare_vals,

```
/* $N$ is the set of computation nodes */
procedure randomized_guided_search_init($N$)
for each $i \in N$ do
    execute(randomized_guided_search(), $i$)
wait_for_all_nodes_to_terminate_execution()
gather_results($1...N$)
return

/* Add initial element $\langle s_0, h_0, r_0 \rangle$ to PriorityQueue PQ */
/* Add $s_0$ to the Visited set */
procedure randomized_guided_search()
while $PQ \neq \emptyset$ do
    $\langle s_i, h_i, r_i \rangle := PQ$.dequeue()
    for each $s' \in$ successors($s_i$) do
        if error($s'$) then
            return Error Statistics
        if $s' \notin$ Visited then
            $Visited := Visited \cup \{s'\}$
            $PQ$.enqueue($\langle s'$, heuristic($s'$), rand_val()$\rangle$)
return No Errors Found

/* PriorityQueue PQ uses compare_vals to order states */
procedure compare_vals($\langle s_1, h_1, r_1 \rangle, \langle s_2, h_2, r_2 \rangle$)
if $h_1 > h_2$ then
    return true
else if $h_1 < h_2$ then
    return false
else
    if $r_1 > r_2$ then
        return true
    else
        return false
```

Figure A.2: Pseudo-code for randomized GDS that shuffles states with the same heuristic values using a secondary key from a random number generator.

that is also shown in Figure A.2 and uses the random values as a secondary key to sort states with the same heuristic rankings. The approach enables us to effectively randomize the order of states with same heuristic values across different states and search levels. The new randomized GDS algorithm has a low cost of randomization because maintaining the random value is the only additional cost it incurs when compared to a regular guided search.

We present two empirical studies that compare randomized GDS to default order guided search. The first study is in JPF v4.0 uses Java benchmarks and the second study is in Estes uses C benchmarks. JPF contains a suite of structural heuristics, [Groce and Visser, 2002a], that exploit thread properties in Java programs and also has a heuristic for finding feasible abstract counter-examples [Groce and Visser, 2002a, Pasareanu et al., 2003]. The Java models used in this study are small to medium sized programs that contain concurrency errors. These models have been collected from different sources: original papers presenting the heuristics [Groce and Visser, 2002a], concurrency literature [Eytani et al., 2007], research describing Java specific errors [Farchi et al., 2003], and the IBM benchmark suite [Eytani and Ur, 2004]. Additionally, these models are characterized to a certain degree having been used recently in two extensive benchmarking studies [Dwyer et al., 2006, Rungta and Mercer, 2007c].

Our empirical study is conducted on a super-computing cluster with 618 nodes. We conduct a single experiment of executing 100 trials of our randomized GDS algorithm in parallel for each subject in the study. The choice of 100 trials is arbitrary, but we believe its size is sufficient to indicate general trends in performance. The randomized GDS trials and the guided search are allocated 7GB RAM, and the execution time is bounded at 1 hour. The 1 hour is again arbitrary but together with 100 trials constitutes an upper bound of 100 hours of computation for each model—a significant amount of resources.

Table A.1 is a comparison between the default order guided search and our new randomized GDS algorithm in JPF. We present results for four different heuristics in JPF: choose-free heuristic, most-blocked heuristic, interleaving heuristic, and the prefer-thread heuristic. Based on the description of the heuristics in [Groce and Visser, 2002a] and our knowledge of the models, we pick heuristics that are most likely to work well for a given model. We present, in Table A.1, the number of states generated for a default order guided search (`GDS`). The values in Table A.1 with the form, x^*, indicate that the search generated x number of states before running out of either time or memory. For the new randomized GDS algorithm (`Randomized-GDS`),

Table A.1: Comparing the performance of default order guided search (GDS) and randomized guided search (Randomized-GDS) using the heuristics in JPF and published benchmarks.

Model	GDS	Randomized-GDS				
		PED	Minimum	Mean	Maximum	95% CI
ChooseFree Heuristic						
Deos(abstracted)	16	1.00	11	40	423	14
RwNoExcpChk(2,100,1)	372,826	1.00	769	6,419	20,865	739
MostBlocked Heuristic						
Clean(1,1,12)	188	1.00	33	377	993	59
Piper(2,2,2)	16,437	1.00	240	1,338	3,909	171
Piper(2,4,4)	$2,478,360^*$	0.87	138,916	1,229,530	2,274,249	116,015
Interleaving Heuristic						
Raxextended(4,3)	$1,225,743^*$	1.00	404	20,774	670,813	14,480
PreferThreads Heuristic						
Accountsubtype(2,2)	$2,225,914^*$	1.00	30,726	193,313	642,193	94
Producerconsumer(1,10,4)	$1,783,620^*$	0.93	2,774	145,466	742,693	36,519
Producerconsumer(1,12,4)	$1,781,899^*$	0.90	13,830	238,092	960,610	52,981
Producerconsumer(1,16,4)	$1,781,530^*$	0.49	7,280	257,131	889,248	67,850
Producerconsumer(1,8,4)	$1,835,216^*$	1.00	1,148	156,428	925,537	38,689
Producerconsumer(2,2,4)	$2,591,457^*$	1.00	10,902	109,394	313,929	13,602
Producerconsumer(2,4,4)	$2,016,936^*$	1.00	2,592	213,491	1,122,008	45,523
Producerconsumer(2,8,4)	$1,721,824^*$	0.68	21,055	434,401	1,098,461	77,976
Reorder(1,1)	144	1.00	40	98	163	6
Reorder(1,5)	545	1.00	36	14,864	64,447	4,312
Reorder(10,1)	1,727,521	0.00	-	-	-	-
Reorder(5,1)	15,207	1.00	393	10,850	30,790	1,473
Reorder(8,1)	274,125	0.80	10,789	714,454	2,624,613	120,013
Reorder(9,1)	691,264	0.32	324,035	861,445	1,412,937	110,618
Twostage(1,1)	218	1.00	53	134	246	9
Twostage(2,5)	24,187	0.96	218	361,571	1,681,177	97,480
Twostage(5,2)	322,593	0.96	5,419	417,841	2,170,752	95,440
Twostage(6,1)	716,413	0.94	31,346	486,830	1,626,718	76,994
Twostage(7,1)	$2,354,460^*$	0.36	81,218	867,382	1,411,624	120,191
Twostage(8,1)	$2,119,657^*$	0.05	178,476	755,151	1,259,085	514,492
Wronglock(1,1)	156	1.00	37	67	122	4
Wronglock(1,10)	7,391	1.00	94	98,616	1,805,704	58,614
Wronglock(1,20)	7,391	0.78	97	562	2328	99
Wronglock(10,1)	$2,330,993^*$	1.00	795	4,848	26,070	834
Wronglock(20,1)	$2,056,532^*$	1.00	3,176	32,484	163,642	6,282

in Table A.1, we present the following statistics: path error density (PED), minimum (Minimum) and maximum (Maximum) number of states generated in a single error discovering randomized GDS trial among all the trials, mean (Mean) number of states generated in all the error discovering randomized GDS trials, and the 95% confidence interval (95% CI) for the mean number of states. The path error density is the ratio of the number of error discovering randomized GDS trials to the total number of trials executed.

The results in Table A.1 show that the new randomized GDS algorithm, overall, improves the error discovery for a given heuristic over default search order. In the AccountSubtype(2,2) model, the default order guided search does not find an error even after exploring over 2.22 million states. In contrast, all 100 trials of the new randomized GDS algorithm find an error and explore only $193,313$ states—on average—before error discovery. Furthermore, the maximum number of states generated—642,193—by a single randomized GDS run of the new algorithm is also dramatically lower than the number of states generated by the default order guided search. Similar behavior is observed in all the ProducerConsumer models, and some TwoStage, Piper, and Wronglock models. In certain models, the mean number of states generated by the new randomized GDS algorithm is more than the states generated by the default order guided search, as seen in the Deos(abstracted) and Reorder(1,5) models; however, even in these models, the minimum number of states generated by the new randomized GDS algorithm is less than the number of states generated by the default order guided search.

Table A.2 presents the results of running our new randomized GDS algorithm on different distance heuristic functions implemented in the Estes model checker [Mercer and Jones, 2005]. We evaluate three specific distance heuristic functions in Table A.2: FSM [Edelkamp and Mehler, 2003], EFSM [Rungta and Mercer, 2005], and e-FCA [Rungta and Mercer, 2006]. The only change in the setup for evaluating heuris-

Table A.2: Comparing the performance of default order guided search (GDS) and randomized guided search (Randomized-GDS) using the Estes model checker.

Model	GDS	Randomized-GDS				
		PED	Minimum	Mean	Maximum	95% CI
FSM Heuristic						
Barbershop(5)	132,376	1.00	13,917	59,496	154,473	5,948
Barbershop(9)	492,166	0.59	61,732	785,698	2,003,928	118,996
Barbershop(11)	$1,292,835^*$	0.15	381,808	813,644	1,247,461	157,172
e-fca Heuristic						
Barbershop(5)	814	1.00	921	1,012	1,308	13
Barbershop(9)	1,070	1.00	1,543	1,692	1,918	18
Barbershop(11)	1,196	1.00	1,939	2,243	2,671	27
Barbershop(20)	1,767	1.00	5,099	6,319	8,439	131
Barbershop(25)	2,086	1.00	7,654	9,873	12,657	233
EFSM Heuristic						
Barbershop(5)	21,706	1.00	4,950	19,849	67,875	1,853
Barbershop(9)	17,537	0.65	94,357	816,848	1,999,595	129,344
Barbershop(11)	30,256	0.06	293,893	701,278	1,181,985	412,829

tics in Estes from the study in JPF is that the randomized GDS trials and guided search using default search order are allocated 2 GB of RAM. The performance of the FSM distance heuristic function improves with the new randomized GDS algorithm as seen in Table A.2. In the `Barbershop(11)` model, the default order guided search does not find an error in over 1.2 million states while the new randomized GDS algorithm explores only $813,644$ states—on average—in 15 error discovering trials.

It is interesting to note that for some models, the default order guided search outperforms the new randomized GDS algorithm using the EFSM and e-FCA distance heuristics. For example, in the `Barbershop(20)` model, 1767 states are generated with guided search while the minimum number of states generated by the randomized GDS algorithm is 5099. The examples where default order guided search outperforms the new randomized GDS algorithm support the hypothesis presented in [Dwyer et al., 2006] that certain reported performance gains of directed model checking techniques can potentially be an artifact of the default order implemented by the model checker rather than the technique itself.

This empirical study shows—on average—that the new randomized GDS algorithm is a better search technique than a default order guided search with no randomization. As a side note, we omit the results on the n-best algorithm in [Jones and Mercer, 2004] and JPF's random choice generator because they are not competitive with the new randomized GDS algorithm. For the remainder of this paper, we use randomized GDS to refer to our new randomized GDS algorithm. The next section shows in another empirical study that with the correct heuristic, randomized GDS performs well in the models where randomized DFS is unable to find an error. We refer to these models as *hard* [Rungta and Mercer, 2007c].

A.4 Evaluation

Randomized DFS serves as a good standard for comparison when we evaluate the performance gains of randomized GDS [Rungta and Mercer, 2007c]. Randomized GDS and randomized DFS both effectively control for the default search of the model checker implementation which makes them well-suited for comparison. Also, when evaluating the performance of a new heuristic, it is sometimes hard to find another heuristic that is designed to work on the same class of programs or properties. Randomized DFS serves as an ideal comparison technique to evaluate the performance of such heuristics. It also provides a tighter lower bound on performance than say a metric based on stateless random walk, [Rungta and Mercer, 2007c], and is a significant bar to overcome when showing performance gains in stateful techniques such as randomized GDS.

We design an empirical study to compare the performance of existing heuristics, using randomized GDS, to randomized DFS implemented by JPF. Like the previous study, we run 100 trials of randomized GDS for each model and an equal number of randomized DFS trials. We bound the execution time at 1 hour for each trial. In our initial experiments, the size of the frontier, states in the priority queue, increases rapidly in randomized GDS trials which causes the searches to run out of memory in JPF before reaching the specified time bound. To overcome this issue, we bound the size of the queue in JPF at 100,000 states. This allows randomized GDS trials to successfully run for an hour in JPF without exhausting the available memory. Bounding the size of the queue turns the complete search into a partial search; however, guided search aims to find a counter-example efficiently rather than to do an exhaustive proof. An earlier study, [Groce and Visser, 2002a], and our experiments show that bounding the size of the queue does not affect, in general, the number of randomized GDS trials that discover an error. The system configuration used to conduct this empirical study is the same as described in the previous section.

We record and normalize values of five different metrics in the randomized GDS and randomized DFS trials to study the performance gains of randomized GDS over randomized DFS. We measure the path error density, number of states generated, time taken before error discovery, length of the counter-example, and total memory utilized for each of the search trials. Recall that the path error density is the ratio of the error discovering trials over the total number of trials executed. We measure the minimum, mean, and maximum values for all metrics, except path error density, generated during the error discovering trials since the randomization generates different results in each trial. The minimum, mean, and maximum values generated by the search trials are normalized between 0.00 and 1.00 for each metric. Here is an explanation of the normalization process for states generated: the smallest number of states generated among the trials of both search techniques, for a given model, is mapped to the value of 1.00; similarly, the largest number of states generated among the trials is mapped to the value of 0.00. All other values for states generated, in the given model, are normalized between these two values. The values are normalized to

Table A.3: Comparing the average values generated in error discovering trials of randomized guided search (RGDS), using the Prefer-Thread heuristic, and randomized DFS (DFS).

	PED		States		Time		Trace		Memory	
	DFS	RGDS	DFS	RGDS	DFS	RGDS	DFS	RGDS	DFS	RGDS
Accountsubtype(1,1)	1.00	1.00	0.98	0.58	0.58	0.68	0.37	0.45	0.62	0.60
Accountsubtype(2,2)	1.00	1.00	1.00	0.59	0.99	0.60	0.42	0.36	0.99	0.37
Wronglock(10,1)	1.00	1.00	1.00	0.79	0.89	0.70	0.34	0.65	0.98	0.78
Wronglock(1,1)	1.00	1.00	0.89	0.52	0.55	0.94	0.70	0.49	0.58	0.56
Wronglock(1,10)	1.00	0.97	0.47	0.98	0.45	0.98	0.57	0.53	0.90	0.93
Twostage(1,1)	1.00	1.00	0.83	0.48	0.66	0.83	0.39	0.54	0.40	0.67
Twostage(2,5)	1.00	0.96	0.52	0.91	0.54	0.94	0.44	0.59	0.39	0.78
Twostage(6,1)	1.00	0.98	0.60	0.87	0.62	0.92	0.31	0.64	0.87	0.63
Reorder(5,1)	1.00	1.00	0.34	0.72	0.34	0.83	0.45	0.75	0.44	0.79
Reorder(8,1)	1.00	0.89	0.36	0.84	0.40	0.92	0.41	0.72	0.89	0.61
ProdCons(1,16,4)	0.67	0.87	1.00	0.88	0.99	0.85	0.55	0.72	1.00	0.67
Twostage(7,1)	0.41	0.73	0.42	0.76	0.42	0.89	0.17	0.58	0.97	0.53
Wronglock(1,20)	0.28	0.81	1.00	0.99	1.00	0.99	0.50	0.62	1.00	0.99
Reorder(9,1)	0.06	0.57	0.31	0.75	0.16	0.87	0.10	0.74	0.99	0.48
Twostage(8,1)	0.04	0.57	0.70	0.70	0.40	0.74	0.01	0.50	0.99	0.43
Reorder(10,1)	0.00	0.34	0.00	0.63	0.00	0.70	0.00	0.51	0.00	0.38

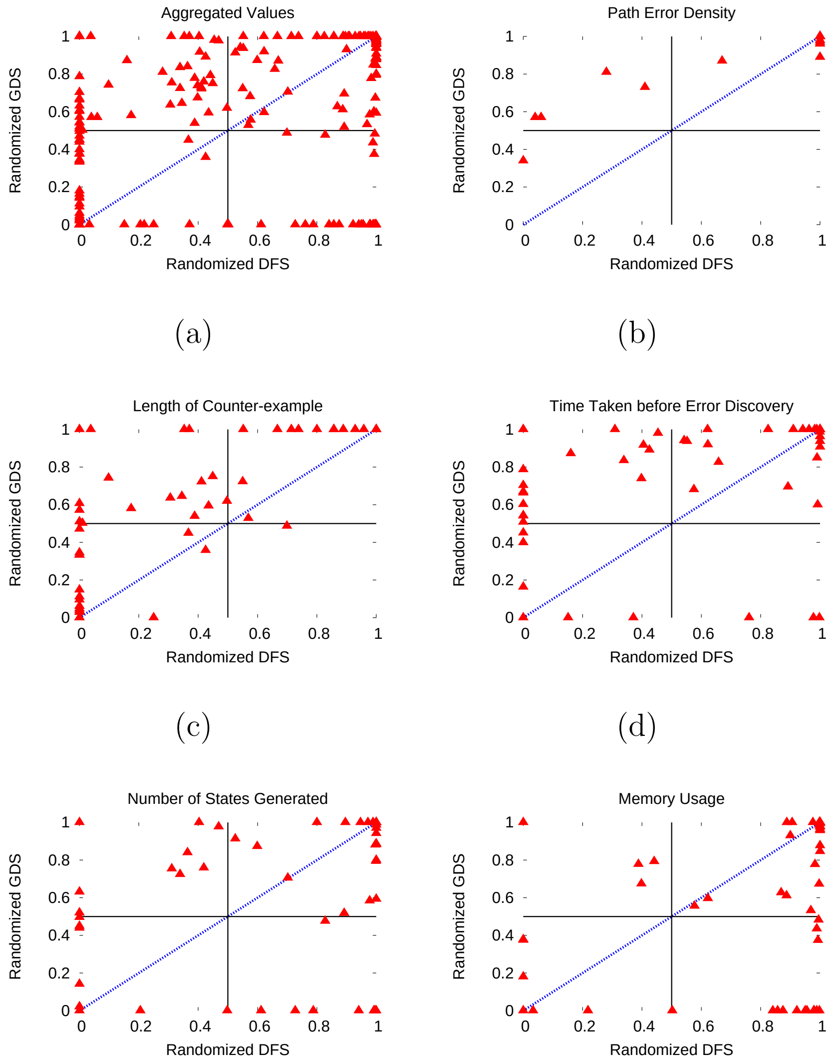
the maximum or minimum values since these represent the extremes in the observed performance across several trials. The normalization process is conducted separately for each metric in a model. Intuitively, values close to 1.00 indicate good performance for a given metric while values close to 0.00 indicate the opposite. The normalization technique helps us in better understanding and visualizing the performance of the heuristic in different models because it puts all metrics on the same scale and graph across both search techniques.

The prefer-thread heuristic, using randomized GDS, performs well in the models shown in Table A.3. Please note that this table omits the data for the minimum and maximum values across our several metrics. Table A.3 only presents the average values that have been normalized. The values given in Table A.3 are as follows: path error density (`PED`), number of states (`States`), time taken (`Time`), length of counterexample (`Trace`), and memory utilized (`Memory`) measured in error discovering trials of randomized GDS and randomized DFS. In a large number of models, the path error density is the same, 1.00, for both randomized DFS and randomized GDS. In models where randomized DFS has a path error density of 1.00, finding an error is not difficult, and the results on these models do not convey much information on the effectiveness of the heuristic.

To overcome some of the weakness in the benchmarks, our study uses hard models generated in [Rungta and Mercer, 2007c] to evaluate the true effectiveness of the heuristic, which are the last six entries in Table A.3. For example, in the `Wronglock(1,20)` model, the measured path error density of randomized DFS is 0.28 while the path error density of the randomized GDS is dramatically higher at 0.81. The average values for states, time, and memory are close to 1.00 for both search techniques in the `Wronglock(1,20)` model; however, the average length of the counter-example for randomized GDS is smaller than the average length of the counter-example recorded from the randomized DFS trials. In understanding the length of a counter-example, values closer to 1.00 depict a shorter counter-example while values close to 0.00 indicate a longer counter-example. There are other models like `Reorder(9,1)`, `Twostage(8,1)`, and `Reorder(10,1)` where randomized GDS improves over randomized DFS.

The high path error density of randomized GDS in models where randomized DFS struggles to find an error makes a compelling argument for the use of the heuristic in the given models. The results in Table A.3 show that randomized GDS, using the prefer-thread heuristic, successfully overcomes the lower bound on the performance set by randomized DFS in the given models.

In Figure A.3 we visualize the comparative performance of randomized DFS and randomized GDS using the prefer-thread heuristic for the models shown in Table A.3. The minimum, mean, and maximum values for all the different metrics and models are aggregated in Figure A.3(a). The different edges along the graph show which search technique generates the best and worst boundary values. The points in the graph along the axis where $x = 0$ show all the worst values that are contributed



(a)

(b)

(c)

(d)

(e)

(f)

Figure A.3: Visualizing the normalized minimum, mean, and maximum values of different metrics comparing randomized GDS, using the Prefer-Threads heuristic, to randomized DFS. (a) An aggregation of all values for the different metrics. (b) Values comparing path error density. (c) Values comparing length of counter-example. (d) Values comparing time taken before error discovery. (e) Values comparing number of states generated. (f) Values comparing memory usage.

by randomized DFS for the measured metrics while the points along the axis where $y = 0$ show all the worst values generated by randomized GDS. Similarly, points along $x = 1$ represent the best values contributed by randomized DFS while points along $y = 1$ represent the best values contributed by randomized GDS. The points above the dashed diagonal line in Figure A.3(a) show the values of the metrics where randomized GDS improves over randomized DFS. In general, there is a high density of points above the diagonal that show for the given set of models, it is more effective to use randomized GDS, with the prefer-thread heuristic, over randomized DFS. There is also a high density of points in the upper right corner of the graph. These points represent the values where both randomized GDS and randomized DFS perform well and do not help us in evaluating the true effectiveness of the search and heuristic over randomized DFS. We now look at each of the metrics separately to understand the areas in which randomized GDS scores over randomized DFS.

There are three metrics where randomized GDS clearly outperforms randomized DFS in the benchmark suite using the prefer-thread heuristic. These three metrics are the path error density, length of the counter-example, and time taken before error discovery as shown in Figure A.3(b), (c), and (d) respectively. The points in the upper right corner of the graph in Figure A.3(b) show that in all trials, both search techniques are equally successful in finding the error; however, points that are above the dashed diagonal line show that a larger number of randomized GDS trials find an error in models where only a small number of randomized DFS trials find an error. The high path error density of randomized GDS is a very compelling measure that depicts the improvement of randomized GDS over randomized DFS. Randomized GDS also performs extremely well in generating shorter counter-examples. The high density of points above the diagonal in Figure A.3(c) indicates that randomized GDS has dramatically shorter counter-examples compared to randomized DFS across all the models in test. Similarly, the distribution of points in Figure A.3(d)

Table A.4: Comparison of results using the Most-Blocked Heuristic with a randomized guided search (RGDS) to results from randomized DFS (DFS).

	PED		States		Time		Trace		Memory	
	DFS	RGDS	DFS	RGDS	DFS	RGDS	DFS	RGDS	DFS	RGDS
Clean(1,1,12)	1.00	1.00	0.09	0.59	0.52	0.87	0.34	0.25	0.42	0.65
Piper(2,4,4)	1.00	1.00	0.96	0.65	0.96	0.63	0.60	0.85	0.94	0.25
Piper(2,8,4)	0.96	0.00	0.92	0.00	0.92	0.00	0.52	0.00	0.47	0.00
Clean(10,10,1)	0.96	0.00	0.95	0.00	0.96	0.00	0.37	0.00	0.85	0.00
Piper(2,16,8)	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table A.5: Comparison of results using the Interleaving Heuristic with a randomized guided search (RGDS) to results from randomized DFS (DFS).

	PED		States		Time		Trace		Memory	
	DFS	RGDS	DFS	RGDS	DFS	RGDS	DFS	RGDS	DFS	RGDS
Airline(6,1)	1.00	1.00	0.75	0.99	0.74	0.99	0.22	0.62	0.53	0.90
Airline(6,2)	1.00	1.00	0.96	1.00	0.95	1.00	0.25	0.60	0.89	0.97
Raxextended(4,3)	1.00	1.00	0.96	0.99	0.96	1.00	0.67	0.99	0.87	0.96
Airline(20,4)	0.03	0.00	0.55	0.00	0.59	0.00	0.47	0.00	0.39	0.00
Airline(20,3)	0.01	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00
Airline(20,2)	0.01	0.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00	0.00

indicates that randomized GDS takes less time to find an error when compared to randomized DFS.

In Figure A.3(e), it is hard to discern which search technique performs better in generating fewer number of states before error discovery; however, the randomized DFS clearly outperforms randomized GDS in the amount of memory utilized as shown in Figure A.3(f). Randomized GDS maintains the frontier of states that need to be explored. The increasing frontier size, however, has a dramatic impact on the memory usage. The unbounded priority queue in JPF causes a serious explosion in memory usage while executing the randomized GDS. In fact, as mentioned earlier, we restrict the size of the priority queue to only 100,000 states so that 7 GB of RAM is not exhausted before reaching the specified time bound. Overall, across the different metrics, randomized GDS using the prefer-thread heuristic improves performance over randomized DFS by effectively finding counter-examples and generating shorter counter-examples.

Table A.6: Comparison of results using the Choose-Free Heuristic with a randomized guided search (GDS) to results from randomized DFS (DFS).

	PED		States		Time		Trace		Memory	
	DFS	RGDS	DFS	RGDS	DFS	RGDS	DFS	RGDS	DFS	RGDS
Deos(true)	1.00	1.00	0.72	0.97	0.56	0.96	0.36	0.95	0.60	0.92
Replicated(5,2)	0.97	0.00	0.81	0.00	0.87	0.00	0.57	0.00	0.88	0.00
RWNoExpChk	0.77	1.00	0.97	0.72	0.72	0.55	0.75	0.99	0.94	0.69

We present results for the most-blocked, interleaving, and choose-free heuristics in Table A.4, Table A.5, and Table A.6 respectively. These heuristics do not perform well on the class of models for which they are designed, and the comparison with randomized DFS makes these heuristics even less appealing in our benchmarks. For example, the randomized DFS path error density for `Piper(2,8,4)` model is 0.96 while the path error density of randomized GDS using the most-blocked heuristic as seen in Table A.4 is 0.00. Similar behavior is seen for the model `Clean(10,10,1)`. The choose-free, most-blocked, and interleaving heuristics do not overcome the randomized DFS lower bound and are not effective in generating counter-examples for models in the tables. The sub-par performance of these heuristics argues a greater need to identify models where they are effective.

The results in this section indicate that given the correct heuristic for a set of models, randomized GDS is effective in finding errors where randomized DFS struggles. It is also important to note that better error discovery, shorter counter-examples, and reduced error discovery time in randomized GDS comes at the cost of increased memory usage due to the large search frontier.

A.5 Conclusions and Future Work

This paper presents a new randomized GDS algorithm that completely shuffles states in the priority queue with equal heuristic rankings. The algorithm is easily implemented, efficient, and has low overhead in terms of memory and time. We show that full randomization of the guided search improves the effectiveness of the search over the regular guided search. To evaluate the performance of randomized GDS using a particular heuristic, we compare it with randomized DFS because randomized DFS creates a lower bound for establishing performance gains in directed model checking. Also, when the heuristic is correctly matched to the models and properties, the new randomized GDS algorithm outperforms randomized DFS in both the effectiveness of the search in finding counter-examples and the length of the counter-examples. The approach is timely given the recent explosion in computation resources and is easily distributed to several computation nodes to improve the likelihood of error discovery.

There is a need to explore other avenues for combining randomization and directed model checking. For example, can we use randomization to balance exploring new parts of the behavior space and use heuristics to exploit the information available about the model? Also, as we develop heuristics appropriate for use in a randomized GDS algorithm, there is a need to understand the intended problem domain for the heuristic. In other words, we need to characterize heuristics in terms of the models for which they are expected to be effective. Without this characterization, it is not obvious which heuristic best fits a given property and model. There also a need to define language and metrics to characterize heuristics for their intended problem domains. An interesting avenue of research is to use something similar to the *“Patterns”* categorization for specifications [Dwyer et al., 1998].

Appendix B

Designing Benchmarks to Evaluate the Effectiveness of Error Discovery Techniques

This empirical study was published as:

N. Rungta and E. G. Mercer, "Hardness for Explicit State Software Model Checking Benchmarks", in *Proceedings of the 5th IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, London, UK, pages 247-256, September 2007

Abstract

Directed model checking algorithms focus computation resources in the error-prone areas of concurrent systems. The algorithms depend on some empirical analysis to report their performance gains. Recent work characterizes the hardness of models used in the analysis as an estimated number of paths in the model that contain an error. This hardness metric is computed using a stateless random walk. We show that this is not a good hardness metric because models labeled hard with a stateless random walk metric have easily discoverable errors with a stateful randomized search. We present an analysis which shows that a hardness metric based on a stateful randomized search is a tighter bound for hardness in models used to benchmark explicit state directed model checking techniques. Furthermore, we convert easy models into hard models as measured by our new metric by pushing the errors deeper in the system and manipulating the number of threads that actually manifest an error.

B.1 Introduction

Model checking is a formal approach for systematically exploring the behavior of a concurrent software system to verify whether the system satisfies the user specified properties [Havelund and Pressburger, 1998, Robby et al., 2003]. A model of a concurrent software system is a transition graph that consists of states and transitions. Each state in the transition graph is a snapshot of the program condition which consists of the values of all variables at a specific program location; while the transitions in the graph are rules that represent the change in the program condition from one state to another.

Exhaustive search techniques such as breadth-first search (BFS) or depth-first search (DFS) are commonly used to explore all the states in the transition graph. Starting from an initial state, the search technique computes the enabled transitions at each state to generate and explore the possible successors in either a breadth-first or depth-first manner.

A path or state in the transition graph that violates a user specified property is known as an error in the model. Since model checking considers all possible thread interactions, it enables us to find subtle concurrency errors deep in the execution trace. These errors are hard to detect in a traditional validation technique based on test vector generation because scheduling decisions cannot be controlled by the input vectors.

The primary challenge in model checking is managing the size of the transition graph. The increase in the size of the transition graph is also known as the state space explosion. For large software systems, the computation resources are exhausted before a search finishes exploring the transition graph. Directed model checking is one solution to the state space explosion problem. It aims to guide the search to parts of the transition graph where errors are more likely to exist [Edelkamp et al., 2001b, Groce and Visser, 2002a, Rungta and Mercer, 2005, 2006, Tan et al., 2004].

It assumes an error exists in the software, and the goal is to find the error before it runs out of computational resources (time or memory).

Directed model checking techniques use heuristics to rank the states in order of interest with states estimated to be near errors explored before the other states. The performance of a given heuristic estimate is compared to existing heuristic functions, or a DFS, with an empirical analysis. A reduction in the number of states generated and a decrease in the total time taken before error discovery are two commonly used metrics to measure performance gains of a directed model checking technique.

The reliance of directed model checking algorithms on an empirical analysis to assess and validate the performance gains of a given technique motivates a need to characterize the quality of models used in such an analysis. The set of models used to benchmark directed model checking should at least be computationally expensive for simple variants of DFS or BFS techniques. In other words, if the baseline model checking algorithm easily solves a benchmark for directed model checking, then the benchmark is not successful or useful in delineating performance. There is a need to classify and characterize benchmarks for directed model checking to control for this situation. We believe that understanding the benchmarks improves understanding in the algorithm.

The work in [Dwyer et al., 2006] is the pioneering research in characterizing the hardness of benchmarks for directed model checking. The benchmarks are guaranteed to contain an error and the goal is to rank the benchmarks in terms of effort, time and memory, required for error discovery. The work in [Dwyer et al., 2006] presents the traditional syntactic metrics for hardness in a set of Java benchmarks such as thread count, class count, location count, etc., and it then defines a semantic hardness metric as a lower bound on the estimated number of paths in the model that contain errors. The lower bound is computed by conducting a large number of stateless random walks on the model. Follow on work in [Dwyer et al., 2007] shows that only 5 to 20 non-deterministic DFS trials are required to guarantee that one DFS trial successfully discovers the error. The DFS trial results are reported on a set of seven models that are classified as hard by the semantic metric for directed model checking in [Dwyer et al., 2006]. The results in [Dwyer et al., 2007] contradict the intuition that a hard model used for benchmarking directed model checking needs to minimally challenge basic search techniques and indicates that the reported lower bound on the hardness for these models in [Dwyer et al., 2006] is not sufficient.

To provide a semantic metric with a tighter bound on the estimated number of errors in a benchmark for explicit state directed model checking, we define a new hardness metric that is computed by conducting a large number of non-deterministic DFS trials. To test the effectiveness of this new hardness metric, we conduct an analysis on a set of 36 models that have not been previously analyzed with non-deterministic DFS trials. In a large subset of the 36 models that have a low estimated number of errors as computed by random walk, all non-deterministic DFS trials conducted are successful in finding an error. The large performance gap between a random walk and non-deterministic DFS indicates that a hardness metric based on non-deterministic DFS trials is a better starting baseline measure of hardness than the one computed using random walk.

To aid researchers in designing hard benchmarks, we identify certain factors that control the hardness in models as measured by the new semantic hardness metric. Traditionally, the total number of threads is a syntactic measure of hardness used when evaluating directed model checking approaches. Our analysis indicates, however, that arbitrarily increasing the total number of threads in the model does not necessarily challenge the new hardness measure. In fact we create two versions of several models with the same number of total threads that have diametrically opposite hardness values as measured by the new metric. As such, we show that the type of threads that cause errors to be discovered and the depth at which errors occur in

```
procedure Random_Walk_Init($s_0$)
    $s := s_0,\ depth := 0$
    Random_Walk($s, depth$)

procedure Random_Walk($s, depth$)
    $X_{succ} :=$ get_successors($s$)
    while ($X_{succ} \neq \emptyset$ or $depth \leq depth\_bound$) do
        $s :=$ get_random_element($X_{succ}$)
        $depth := depth + 1$
        if error($s$) then
            print “1 Error Found”
            return
        $X_{succ} :=$ get_successors($s$)
    print “No Errors on this Path”
```

(a)

```
procedure Random_DFS_Init($s_0$)
    $Visited := \{s_0\}$
    Random_DFS($s_0, Visited$)

procedure Random_DFS($s, Visited$)
    if (within_time_bound()) then
        if error($s$) then
            print “1 Error Found”
            exit
        $X_{succ} :=$ get_successors($s$)
        randomize_elements($X_{succ}$)
        for each $s' \in X_{succ}$ do
            if $s' \notin Visited$ then
                $Visited := Visited \cup \{s'\}$
                Random_DFS($s', Visited$)
    else
        print “Out of Time”
```

(b)

Figure B.1: Pseudo-code for randomized search techniques (a) True random walk with no backtracking (b) DFS with a randomized transition order

the transition graph are two controlling factors that affect the hardness measure. We present evidence for these factors in making seemingly easy models into hard models by systematically varying these factors in the models.

The main contributions of this paper are: (1) Defining non-deterministic DFS (randomized DFS) as a tighter bound on the hardness of a model when compared to random walk, (2) Showing correlation between error discovering threads and depth of errors with the hardness of models, and (3) Characterizing a set of existing benchmarks as well as creating hard benchmarks based on the new metric.

B.2 Background and Motivation

Recent work in [Dwyer et al., 2006] defines *path error density* as a conservative probability estimate on the number of paths in a model that contain an error. This estimate is a lower bound on the total number of paths that actually contain an error in the model. To compute the path error density, a large number of independent random walks are conducted on the model. The probability estimate is the ratio of random walks that find an error to the total number of random walks. This estimate is assigned as the path error density of the model. The path error density of an easy model is close to one if a large number of random walks find an error. This demonstrates that there is a high probability of finding an error along an arbitrary path in the program which makes the model extremely easy in terms of error discovery. Conversely, the path error density of a hard model is close to zero if only a few random walks are successful in finding an error. The work in [Dwyer et al., 2006] also reports syntactic metrics, like lines of code and thread count, on the models used in their study. The study shows that syntactic metrics are not able to predict path error density values. A model that looks syntactically hard may actually be semantically easy. This syntax-semantic gap creates a need for a semantic metric, like path error density, to classify benchmarks for directed model checking until we better understand the relationship between syntax and error discovery.

Random walk is a stateless search technique that does not store information on states that are already explored. In Figure B.1(a), we present the pseudo-code for a pure random walk with no backtracking. Starting from an initial start state (s_0), a random walk explores a sequence of states in the transition graph expanding a random successor at each state in the path (lines $4-7$ and 11). If the random walk encounters an error, it reports the error (lines $8-10$); however, when the random walk reaches a node with no successors or a depth greater than the specified depth bound (line 5), it simply terminates the search (line 12).

The path error density does not provide a tight bound on the estimated number of paths in a model that contain an error due to the inherent limitations of random walk. New research in [Pelanek et al., 2005] shows that the total coverage obtained by a pure random walk is largely dependent on the structure of the graph. It also shows that coverage of the random walk increases logarithmically with the number of computation steps; thus, during the initial phase of the random walk, a large number of new states are visited, but with time, the number of newly visited states decreases rapidly. Experimental analysis indicates that the coverage achieved by pure random walk ranges between 100% to 1% for transition graphs commonly used in model checking [Barnat et al., 2006, Dill, 1996, Holzmann, 1997]. In models where pure random walk achieves medium to low coverage, the path error density measure does not accurately reflect the effort required in finding an error in the model because the coverage is so sporadic.

The work in [Dwyer et al., 2007] shows that a parallel randomized state-space search (PRSS) is very effective in finding errors for models of [Dwyer et al., 2006] with relatively low path error densities. Intuitively, PRSS runs independent randomized DFS trials in parallel to discover an error. A randomized DFS is simply a variant of the rudimentary DFS that randomizes the order of its successors in the search. The PRSS approach computes the required number of nodes such that with every node running a randomized DFS trial in parallel at least one node finds the error in the model.

In Figure B.1(b), we present the pseudo-code for a randomized DFS. It explores a sequence of states starting from the start state (s_0), where at each state it generates a set of all possible successors, randomizes their order, and picks one successor to explore in a depth-first manner (lines $8 - 13$). When the search encounters a node with no successors, it backtracks to the next node in the list of randomized successors. A randomized DFS is a stateful search technique that maintains a set of visited states to track every explored state (lines $10 - 11$). The randomized DFS terminates when an error state is encountered (lines $4 - 7$) or reaches the specified time bound (lines 4 and $14 - 15$). Note that the algorithm for the randomized DFS presented in Figure B.1(b) assumes that the model contains an error. Memory resources limit the amount of time a randomized DFS trial can run. Unlike a random walk, memory in a randomized

DFS trial is exhausted if it is run long enough. For seven subjects presented in [Dwyer et al., 2006] with a relatively low path error density, the PRSS requires only between 5 to 20 nodes to guarantee error discovery in at least one randomized DFS trial [Dwyer et al., 2007]. This is counterintuitive since the models labeled hard seem so easy.

A hard model should at least challenge a randomized DFS which is a basic search technique used in model checking tools. It is counterintuitive for a small number of parallel randomized DFS trials in the PRSS approach to consistently discover errors in supposedly hard models. This contradiction motivates a need for defining a better notion of hardness in models for benchmarking path analyses techniques and model checking algorithms. We especially need this metric to characterize and classify benchmarks for comparative studies in explicit state directed model checking.

B.3 Error Density Measure

The path error density based on a stateless random walk underestimates the hardness of models for benchmarking stateful directed model checking algorithms. Specifically, it tends to label models hard even when the error discovery is trivial with a stateful randomized DFS. A hard model used for benchmarking directed model checking algorithms needs to at least be computationally expensive in terms of time and space for a stateful randomized DFS. To demonstrate the utility of having a stateful hardness measure, we re-run the PRSS analysis for 36 models in [Dwyer et al., 2006]; however, instead of computing the number of nodes required to run randomized DFS in parallel to guarantee at least one node finds an error, we record the number of randomized DFS trials that encounter an error.

Based on randomized DFS trials, we define a new hardness metric, the observed randomized-DFS (R-DFS) error density which is the ratio of the randomized DFS trials that find an error to the total number of randomized DFS trials conducted. Since the observed R-DFS error density is based on a stateful search, it provides a

Table B.1: Comparing path error density and randomized DFS

Subject			Randomized DFS trials			
Name(Thread Num)	Params	Path error	observed R-DFS	Number of States		
		density[Dwyer et al., 2006]	error density	Minimum	Average	Maximum
Account-NoDeadlkCk(6)	none	0.549	1.00	182	27,928	1,089,171
Account-NoExcpCk(6)	none	0.077	0.48	405	1,749,259	13,151,326
AccountSubtype(10)	8,1	0.152	0.34	250	248,714	3,245,340
Airline(21)	20,8	0.069	0.49	101	571,214	6,479,374
Airline(7)	6,2	0.030	1.00	40	226,846	5,112,586
Airline(7)	6,1	0.003	1.00	50	1,618,915	6,401,539
Airline(21)	20,2	0.000	0.01	5,249	5,249	5,249
Alarm Clock(4)	9	0.093	1.00	28	112	288
Alarm Clock(4)	4	0.083	1.00	41	111	147
AllocateVector(3)	8,20,1	0.441	0.99	33	198,206	4,623,001
AllocateVector(3)	2,20,4	0.294	1.00	34	5,646	143,866
AllocateVector(3)	2,20,1	0.084	1.00	34	4,832	7,773
AllocateVector(3)	2,100,1	0.017	1.00	34	28,406	40,248
Clean(21)	10,10,1	0.289	0.96	206	283,357	5,497,056
Clean(3)	1,1,12	0.033	1.00	12	907	987
Deadlock(3)	1	0.450	1.00	12	17	33
Deadlock(3)	2	0.379	1.00	5	6	8
Deos(4)	abstracted	0.190	1.00	7	747	2,638
LinkedlistSync(5)	4,100	0.000	1.00	9,324	10,014	12,351
Piper(17)	2,8,4	0.083	0.96	146	621,340	7,921,766
Piper(9)	2,4,4	0.029	1.00	1,611	189,872	1,288,076
ProducerConsumer(11)	2,8,4	0.967	1.00	127	261	12,334
ProducerConsumer(7)	2,4,4	0.956	1.00	97	116	372
ProducerConsumer(5)	2,2,4	0.768	1.00	93	112	210
RaxExtended(6)	2,3	0.128	1.00	25	1,783	19,502
Reorder(3)	1,1	0.030	1.00	16	55	80
Reorder(7)	1,5	0.043	1.00	15	49,151	65,490
ReplicatedWorkers(9)	8,2,0,10,.001	0.948	0.97	1,739	1,801	1,866
RWNoExcepChk(5)	2,2,100	0.769	0.80	52	533	2,031
TwoStage(3)	1,1	0.043	1.00	20	57	127
TwoStage(8)	2,5	0.028	1.00	30	1,759,759	3,702,115
TwoStage(5)	2,2	0.022	1.00	34	3,301	8,638
WrongLock(12)	10,1	0.478	1.00	61	94	167
WrongLock(12)	1,10	0.200	1.00	25	1,574,058	2,966,459
WrongLock(3)	1,1	0.068	1.00	13	25	43

tighter bound on the hardness of models for benchmarking explicit state directed model checking algorithms compared to path error density which is computed using random walk. The underlying assumption is that randomized DFS always achieves better coverage of a transition graph compared to a random walk. We do not consider comparisons with BFS because variants of BFS often have prohibitively large frontier sizes that render BFS techniques ineffective for error discovery in the benchmark set.

B.3.1 Experiment Design

To compare path error density and observed R-DFS error density we conduct random walk and randomized DFS trials on a cluster of 618 nodes. Every node in the cluster has 8 GB of RAM and two Dual-core Intel Xeon EM64T processors (2.6 GHz). The execution time for a single randomized DFS trial is bounded at one hour. We pick the time bound to be consistent with the other recent studies [Dwyer et al., 2006, 2007]. Later in this section we also study the affects of changing the time bound. The programs in this study are compiled using Java 1.5 and verified by the Java PathFinder (JPF) v4.0 model checker with partial order reduction turned on [Visser et al., 2003].

For each model in test we conduct 100 randomized DFS trials to compute its semantic hardness. We experimented with different number of trials to pick an upper bound on the required number of trials for predicting the semantic hardness. For the models in our test suite we found that 100 trials are sufficient to compute the semantic hardness. To compute the path error density, we execute 10,000 trials of random walk, where one trial is a single random walk execution or single path in the program. The original study of [Dwyer et al., 2006] uses between 1000 and 10,000 random walk trials to estimate the path error density of the model.

Independent Variable

We vary models, the independent variable, in our study to test whether randomized DFS provides a tighter bound on hardness of benchmarks used in explicit state directed model checking. We conduct the study on a set of 36 models used in the benchmarking analysis of [Dwyer et al., 2006] and have not been previously analyzed with a randomized DFS. The set of benchmarks encompasses a wide variety of Java programs with concurrency errors. The test suite includes programs derived from concurrency literature, small to medium-sized realistic programs, models designed to exhibit Java-specific errors described in [Farchi et al., 2003], and models developed at IBM to support their analyses research [Eytani and Ur, 2004]. Many models have been made parameterizable to control the number of threads for studying their effect on the path error density.

Dependent Variables and Measures

The dependent variables in this study are the path error density and the observed R-DFS error density values. We compute the values of the path error density rather than report the values in [Dwyer et al., 2006][1]. We also compute the observed R-DFS error density which is the ratio of randomized DFS trials that find an error over the total number of randomized DFS trials executed. On the scale of hardness, an observed R-DFS error density of 1.00 indicates an extremely easy model while an observed R-DFS error density of 0.00 indicates a very hard model. Note that this scale is consistent with the path error density hardness scale of [Dwyer et al., 2006] where probabilities close to one indicate easy models whereas probabilities close to zero indicate hard models. We measure the number of states generated during the randomized DFS trials to gain a better understanding on the effort required for error discovery by the randomized DFS trials in terms of time and memory resources.

B.3.2 Results

The results of the study are presented in Table B.1 where the first column indicates the name of the subject, and the maximum number of threads created in the subject is indicated in the parenthesis (*Name(Thread Num)*). The second column specifies the input parameters (*Params*) used by the subject (see [Dwyer et al., 2006] for parameter details and other syntactic metrics such as thread count, class count, location count, etc.). In the section of Table B.1 labeled *randomized DFS trials*, we present four

[1] The values in [Dwyer et al., 2006] are computed on JPF3.1.2 while we do our analysis on JPF4.0.

statistics: the *observed R-DFS error density*, the *minimum* and *maximum* number of states generated in a single trial of randomized DFS among the error discovering trials, and the *average* number of states generated across all randomized DFS trials that find an error.

The analysis in Table B.1 shows that for a large number of models that have near zero path error densities with random walk, almost all of the randomized DFS trials find an error. For example, the model `Clean` with parameters (1,1,12) has a path error density of 0.033 while its observed R-DFS error density is 1.00. The parameterized versions of the `TwoStage` and `Reorder` models have a path error density of less than 0.050 but have an observed R-DFS error density of 1.00. In 26 examples presented in Table B.1 out of the total 36 subjects, all 100 trials of randomized DFS find an error. Furthermore, in some models with a low path error density and high observed R-DFS error density, the minimum and average number of states generated in the randomized trials is very small. This indicates that the computation cost in terms of memory for error discovery in these models is very low. Fourteen models with an observed R-DFS error density of 1.00 generate less than 1000 average states before error discovery. In fact, some models like `TwoStage` with parameters (1,1) and `ProducerConsumer` with parameters (2,4,4) generate a maximum of only 127 states and 372 states respectively in a single randomized DFS trial out of the 100 trials. The small state counts further show for these models that a stateful search technique is effective in finding an error when compared to random walk.

The fact that most models have a hardness of 1.00 under the observed R-DFS error density metric shows that the set of models used in [Dwyer et al., 2006] severely lacks in diversity when evaluating directed model checking approaches. It also indicates that the more varied distribution of hardness values computed by the path error density in [Dwyer et al., 2006] is not representative of the amount of effort required to find errors in these models with stateful search methods.

The examples in Table B.1 that appear hard in terms of the observed R-DFS error density are interesting to study in order to identify factors that cause a low observed R-DFS error density in the models. For example, the Accountsubtype model with parameters (8,1) is a moderately hard model with an observed R-DFS error density of 0.34, and the average number of states generated before error discovery is significant. Further examination of the Accountsubtype model may assist in identifying the factors affecting the low observed R-DFS error density. There are two other parameterized subjects that have a low observed R-DFS error density: Airline with parameters (20,2) and Piper with parameters (2,16,8). These are interesting subjects because other parameterized versions of these models have a high observed R-DFS error density. For example, Piper with parameters (2,4,4) and Airline with parameters (6,1) have an observed R-DFS error density of 1.00.

B.3.3 Effect of the Time Bound

The observed R-DFS error density measure in Table B.1 is dependent on the time bound of 1 hour set for the randomized DFS trials. We test the effect of the time bound on the observed R-DFS error density by running randomized DFS trials on a set of hard models using different time bounds. In the next section we show how to create the hard models. The independent variable in this study is the time bound while the dependent variable is the observed R-DFS error density. We expect the observed R-DFS error density to increase with the time bound. In Table B.2 we present results of the study.

In certain models, the observed R-DFS error density steadily increases with time while in others, it is not clear how the observed R-DFS error density changes. In the TwoStage model with parameters (7,1), the observed R-DFS error density increases from 0.41 to 0.93. This still shows that Twostage(7,1) is a moderately hard model for stateful search techniques because it takes an upper bound of 300

Table B.2: Increasing Time Bound

Subject		observed R-DFS error density		
Name (Thread Num)	Params	1 hour	2 hours	3 hours
Airline(21)	20,2	0.01	0.00	0.00
Reorder(11)	9,1	0.06	0.45	0.37
TwoStage(9)	7,1	0.41	0.69	0.93
TwoStage(10)	8,1	0.04	0.03	0.07
TwoStage(12)	10,1	0.00	0.00	0.00
Wronglock(22)	1,20	0.18	0.20	0.20

computation hours—a significant amount of resources—to obtain an observed R-DFS error density of 0.93 in the model. In essence, the time bound allows researchers to set their own threshold of hardness. In general, we expect a decrease in time bound makes a model progressively harder and vice-versa.

In the following section, we use models defined as hard in terms of the observed R-DFS error density measure to identify the factors that contribute toward hardness other than the time bound. In other words, given a *fixed time bound*, how do we make an easy model hard? We show that the number of threads that manifest an error in the model and the depth of the transition graph at which errors occur assist in making hard models. We also use these factors to convert some easy models into hard models in a given time bound.

B.4 Making Models Hard

Making hard models for benchmarking directed model checking algorithms requires an understanding of the factors that make a model hard in terms of the observed R-DFS error density measure. From prior studies, [Dwyer et al., 2006], and our own empirical analysis on different models, we show there is a causal relationship between the observed R-DFS error density and the depth of errors in the transition graph as well as the thread count in the model. To aid in our discussion on the effects of the depth of errors on the observed R-DFS error density, we present in Table B.3 several members of our study group from Table B.1 with their minimum error depth (*Min*), maximum error depth (*Max*), average error depth (*Avg*), Standard deviation in the error depth (*Std Dev*), the 95% confidence interval (*95% CI*), and the maximum depth observed in the model (*Depth*) as measured by several trials of randomized DFS. We selected these models from the larger set because we are able to make these models hard by controlling the depth of the errors and the threads that produce the errors. We are still working on other models from the study.

A closer analysis indicates that for a large number of the models, error depth is closely tied to the thread count of the model which means it is not always possible to strictly separate the depth of errors and number of threads while making a model hard. Furthermore, simply increasing the number of threads arbitrarily in a model does not always result in hard models. Sometimes it is important to systematically vary the number of threads that manifest an error in the model. This also makes it hard to define a syntactic metric to classify benchmarks for directed model checking.

A summary of the models we make hard is presented in Table B.4. For each model we make hard (*Subject*), we show the parameters (*Params*), the factor we used to make models hard (*Hardness Factor*), the lowest observed R-DFS error density seen among the parameterized versions of the model (*observed R-DFS error density*), and the parameter values used to obtain the lowest observed R-DFS error density (*P. val*). In Table B.5 we present evidence on the importance of the error depths and number of threads that manifest an error by making models hard using these factors. For each subject (*Name*) with its corresponding parameters (*Params*) in Table B.5 we conduct an experiment of a 1000 randomized DFS trials time-bounded at 1 hour. We present the *observed R-DFS error density*, the minimum (*Min*), maximum (*Max*), and average (*Average*) number of states generated, and the minimum, maximum and average depth of errors observed during the randomized DFS trials.

Table B.3: Error depth statistics

Name(Thread Num)	Params	Min	Max	Avg	Std Dev	95 % CI	Depth	observed R-DFS error density
Piper(9)	2,4,4	63	89	75	3.70	0.102	119	1.00
AllocateVector(3)	2,100,1	28	102	71	21.8	0.622	119	.948
RaxExtended(6)	2,3	16	701	179	143	3.981	712	1.00
WrongLock(12)	1,10	15	64	27	6.84	0.189	83	1.00
Wronglock(12)	10,1	61	98	85	4.49	0.124	98	1.00
ProducerConsumer(11)	2,8,4	110	158	126	4.76	0.137	173	.926
AccountSubtype(10)	8,1	239	274	255	6.37	0.249	278	.500
Airline(7)	6,1	31	40	35	1.30	0.036	41	1.00
Airline(7)	6,2	24	40	34	2.15	0.059	41	1.00
Airline(21)	20,2	114	124	119	1.36	0.144	124	.068
Airline(21)	20,8	87	127	113	7.00	0.195	127	.986

Table B.4: Summary of Models made hard

Subject	(Params) : Making models Hard
Piper	(**#seatRequests, #producers and #consumers, bufferSize**) : Errors are pushed deeper in the transition graph when we increase the **bufferSize** and keep the number of threads, **#producers and #consumers**, constant.
Airline	(**#ticketsIssued, cushion**) : The minimum depth of the error is pushed deeper in the execution trace when we increase the value of **cushion** and keep the total possible number of threads, **#ticketsIssued**, constant.
Accountsubtype	(**#correctAccounts, #incorrectAccounts**) : We increase the number of threads that create **#correctAccounts** and keep the number of threads that create **#incorrectAccounts** constant because only threads that create **#incorrectAccounts** cause an error condition.
Wronglock	(**#dataLockers, #classLockers**) : We increase the **#classLockers** while keeping the **#dataLockers** constant; dataLockers check for the data inconsistency created by classLockers.
ProducerConsumer	(**#producers, #consumers, #items**) : We increase **#consumers** and keep **#producers** constant because the error condition, deadlocked consumer threads, is detected after the correctly running consumer threads complete execution.
Reorder	(**#setters, #checkers**) : We increase the **#setters** and keep **#checkers** constant; setter threads create the error while checker threads manifest the error.
TwoStage	(**#twoStagers, #readers**) : We increase the **#twoStagers** and keep **#readers** constant; twoStager threads cause the error while reader threads manifest the error.

The hardness in the `Piper` and `Airline` model can be controlled by varying the depth of the error for a specific thread configuration. The `Piper` model with parameters (2,4,4) has a moderately-deep distribution of errors as seen in Figure B.2 and the minimum depth of an error is fairly high as seen in Table B.3. The depth of errors in the `Piper` model can be controlled by increasing the size of the global buffer as shown in Table B.4. This is because a larger buffer requires more execution steps in the transition graph to fill the buffer. The `Piper` model with parameters (2,8,5) with 17 threads has an observed R-DFS error density of 0.930 as shown in Table B.5.

Table B.5: Making models hard as measured by the observed R-DFS error density

Subject		observed R-DFS	States			Error Depth Statistics		
Name(Thread Num)	Params	error density	Min	Max	Average	Min	Max	Average
Piper(17)	2,8,5	.930	166,680	9,124,665	3,993,474	133	165	147
Piper(17)	2,8,6	.010	1,597,902	6,897,906	3,670,129	139	148	144
Piper(17)	2,8,7	0.00	-	-	-	-	-	-
Airline(21)	20,7	.938	293	5,671,899	85,169	94	125	116.0
Airline(21)	20,6	.836	336	7,799,950	142,970	101	126	117.7
Airline(21)	20,5	.669	353	6,707,146	237,961	107	125	118.5
Airline(21)	20,4	.441	372	7,490,825	546,131	107	124	119.0
Airline(21)	20,3	.228	379	6,048,042	448,035	108	123	119.5
Airline(21)	20,1	.007	464	5,671,267	1,649,334	119	122	120.0
Accountsubtype(11)	9,1	.316	1,270	7,983,855	1,078,276	269	300	281
Accountsubtype(12)	10,1	.171	1,908	7,694,424	1,499,804	295	326	306
Accountsubtype(13)	11,1	.085	1,826	6,992,772	1,270,414	322	350	332
Accountsubtype(22)	20,1	0.00	-	-	-	-	-	-
Accountsubtype(11)	8,2	.995	1,084	6,595,167	156,152	267	315	292
Accountsubtype(17)	8,8	1.00	2,308	1,218,295	5,131	475	584	560
Wronglock(22)	1,20	.186	50	182	98	18	51	30
ProducerConsumer(10)	1,8,4	.373	232	6,680,332	1,218,856	80	127	95
ProducerConsumer(12)	1,10,4	.232	263	4,160,717	60,268	92	125	104
ProducerConsumer(14)	1,12,4	.198	269	605	395	102	134	113
ProducerConsumer(18)	1,16,4	.178	353	711	455	122	152	133
TwoStage(8)	6,1	1.00	61,572	4,028,179	2,335,507	51	61	58
TwoStage(9)	7,1	.160	1,007,210	9,650,374	6,259,728	62	69	65
TwoStage(10)	8,1	.002	7,484,947	8,308,465	7,896,706	72	73	72
TwoStage(12)	10,1	0.00	-	-	-	-	-	-
Reorder(7)	5,1	1.00	8,083	34,671	27,280	30	35	34
Reorder(10)	8,1	1.00	550,927	4,326,254	3,415,442	40	45	49
Reorder(11)	9,1	.017	3,475,434	7,470,672	5,615,955	52	55	54
Reorder(12)	10,1	0.00	-	-	-	-	-	-

If we keep the same number of producers and consumers but increase the buffer size from 5 to 7, the corresponding observed R-DFS error density rapidly drops to 0.00. The minimum depth of the errors also dramatically increases as shown in Table B.5. A strong dependence on the depth of errors in the `Piper` model allows us to create hard versions of the `Piper` model in terms of the observed R-DFS error density. If we fix the number of producers and consumers to any arbitrary value we are always able to push the error depth by increasing the buffer size. This effectively makes the model hard as measured by the observed R-DFS error density. The `Airline` model is made hard in the same fashion as described in Table B.4. By controlling the value of the **cushion** parameter the `Airline` models with 21 threads get progressively harder as shown in Table B.5.

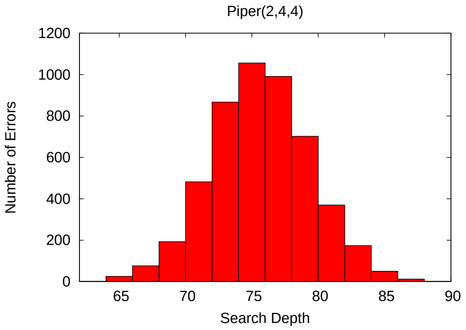


Figure B.2: Frequency of errors at various search depths

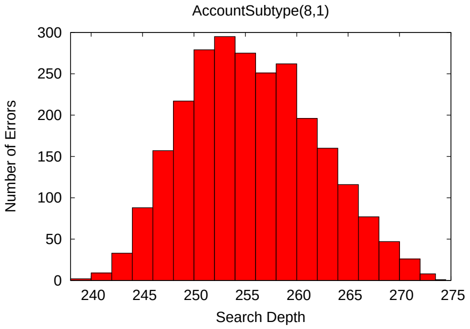


Figure B.3: Frequency of errors at various search depths

In other models a more complicated thread count manipulation is required to decrease the observed R-DFS error density of the model. The Accountsubtype is a classic example of how simply increasing the size of the transition graph arbitrarily does not decrease the observed R-DFS error density of the model.

In the Accountsubtype model we control the number of threads that create an error in the model to create hard versions of the model in terms of the observed R-DFS error density. The Accountsubtype model has fairly deep errors; however, simply increasing the minimum depth of the errors by arbitrarily changing the number of threads does not always decrease the observed R-DFS error density of the model. The Accountsubtype model with parameters (8,1) creates a total of 10 threads and has an observed R-DFS error density of 0.500 as shown in Table B.1. Table B.3 shows that the (8,1) thread configuration has an average error depth of 255 and the maximum depth of the transition graph is 278. Moreover, a large number of the errors are clustered around the mean as demonstrated by Figure B.3. We can push the errors deeper in the transition graph by increasing the number of threads by one, and we get two possible input parameter configurations with 11 threads: (9,1) and (8,2). The minimum depth of errors for the (9,1) and (8,2) configurations are 269 and 267 respectively as shown in Table B.5, while the minimum depth of errors for the (8,1) configuration shown in Table B.3 is 239. Although, the depth of errors for the (9,1) and (8,2) configurations are deeper than the (8,1) configuration, the observed R-DFS error density of Accountsubtype model with parameters (9,1) is 0.316 which is dramatically lower than 0.995 for the (8,2) configuration. Simply increasing the thread count may push the errors deep in the transition graph but may not necessarily challenge the observed R-DFS error density metric. This is why syntactic metrics are hard to define to predict hardness. Until we understand the syntactic relation to the error, we need to rely on semantic definitions for benchmarks to use in comparative studies for directed model checking.

In the Accountsubtype model, we control a specific type of thread count to make models hard in terms of the observed R-DFS error density. The Accountsubtype model takes two parameters as input where the first parameter is the number of threads that create error-free accounts while the second parameter is the number of threads that create error-causing accounts. The error-causing accounts update a local data value of its neighboring thread with an unprotected access to the data causing a data inconsistency. As we increase the number of threads that create error-free accounts from 9 to 20 while fixing the number of threads that create error-causing accounts at 1, the observed R-DFS error density value drops from 0.316 to 0.00 as shown in Table B.5. In contrast, when we fix the number of threads that create error-free accounts at 8 while increasing the number of threads that create error-causing accounts from 1 to 8, the observed R-DFS error density value dramatically increases from 0.50 for parameters (8,1) seen in Table B.1 to an observed R-DFS error density of 1.00 for parameters (8,8) seen in Table B.5.

We make the other models in Table B.4 hard according to our semantic metric by either pushing the error deeper into the execution or by controlling the specific type of thread that manifests an error. All of the models in Table B.4 challenge a time-bounded randomized DFS and are suitable for studying directed model checking algorithms. This is the value of the semantic metric. The performance gains occur where the basic search strategy is not successful in the benchmark.

B.5 Other considerations

The observed R-DFS error density measure is dependent on the time bound of the randomized DFS trials. It is fair to ask, as we increase the depth of errors or the number of threads in a model, does an increase in the corresponding time bound of the randomized DFS trials result in a better observed R-DFS error density estimate? To test this hypothesis we re-run some of our experiments for models with relatively low observed R-DFS error density after doubling the randomized DFS execution time bound to 2 hours. In these experiments we did not see any perceptible rise in the observed R-DFS error density of the model. However, to find a definitive relation between the time bound of the randomized DFS and the observed R-DFS error density, all the randomized DFS trials should be run without a time bound and have memory be the only factor limiting the duration of the randomized DFS trial. Obtaining computation resources to perform such a expensive study is very challenging. Regardless, when showing performance in directed model checking relative to randomized DFS, we can compare time-bound to time-bound to give meaning to performance gains.

Memory is the other important factor that affects the observed R-DFS error density of the model. The `BoundedBuffer` model with parameters (3,6,6,1) presented in [Dwyer et al., 2007] requires 14 GB of memory before the PRSS approach can guarantee error discovery in 20 nodes. In our experiments with the same model run with lower memory constraints, the randomized DFS trials run out of memory in a few minutes. Given sufficient memory, a randomized DFS trial always discovers an error but there is a realistic constraint on the amount of memory available. At some point there is a tradeoff on whether we should spend additional computation resources of time and memory on a randomized DFS or expend resources in developing efficient model checking algorithms. Regardless, any directed model checking study should discuss the computation limits in articulating performance gains.

As seen in the previous section, varying the depth of errors and the number of threads manifesting an error do not always allow us to increase the hardness of models. For example, pushing the errors deeper in the `Piper` model resulted in a decrease of the observed R-DFS error density while for the `Accounsubtype` model, increasing the depth of errors did not give the same results. Since making the model hard is so dependent on the model, the depth of errors and number of threads that manifest an error can be used as a starting point for making models hard in terms of the observed R-DFS error density measure. We are still trying to identify additional factors that can potentially affect the performance of the randomized DFS.

B.6 Related Work

In recent years tremendous progress has been made in the field of software model checking [Ball and Rajamani, 2001, Clarke et al., 2004, Henzinger et al., 2003, Holzmann, 1997]. Java Pathfinder model checks the actual Java bytecode using a Java virtual machine [Visser et al., 2003]. Similar approaches use simulators and debuggers for other machine architectures [Leven et al., 2004, Mercer and Jones, 2005]. These approaches retain a high-fidelity model of the target execution platform while retaining a low-level control of scheduling decisions. There is a growing interest in developing tools and models for benchmarking different model checking approaches used to verify multi-threaded programs [Eytani and Ur, 2004, Eytani et al., 2007, Farchi et al., 2003]. Recent work [Dwyer et al., 2006] makes a good first attempt in trying to evaluate the hardness of models used for benchmarking directed model checking by using random walk to estimate the number of paths in a model that contain an error. It is the first time random walk is used to evaluate the quality of directed model checking benchmarks. Other researchers have often used variants of random walk as an error discovery mechanism with limited success [Haslum, 1999, Jones and Sorber, 2005, Pelanek et al., 2005, Sivaraj and Gopalakrishnan, 2003].

Randomization techniques have been used in tandem with different model checking approaches by various researchers. Stoller uses randomized scheduling to find thread interactions that lead to an error in Java programs [Stoller, 2002], while Jones and Mercer randomize a decentralized parallel guided search to disperse the search in different parts of the transition graph [Jones and Mercer, 2004]. The work in [Dwyer et al., 2006] shows that the default search order used by an algorithm in a model significantly affects the results for error discovery in empirical analysis. The analysis in [Dwyer et al., 2006] demonstrates that by simply randomizing the default search order, the same algorithm may perform worse than other algorithms. The PRSS approach in [Dwyer et al., 2007] overcomes the limitations of the default search order by using a depth-first search that randomizes the order of successors.

B.7 Conclusions and Future Work

Characterized and classified experimental benchmarks for directed model checking are critical to understand the performance in explicit state directed model checking. Currently, we do not have the syntactic metrics for this classification and characterization. As such, this paper defines the observed R-DFS error density as a semantic metric suitable for directed model checking empirical studies. The observed R-DFS error density is based on a rudimentary search technique and provides a lower bound on the number of errors in a model. Our analysis in this paper of the most comprehensive benchmark set of Java programs for explicit state directed model checking shows the set to be lacking in diversity and hardness. We study the few Java models that have a low observed R-DFS error density to understand the factors that contribute toward making them hard. Our analysis of the hard models seems to indicate that a model can be made hard by pushing errors deep in the transition graph and manipulating the thread count of specific threads reducing the number of errors. We use these factors to systematically lower the observed R-DFS error density of several easy models.

In a follow-on work, [Rungta and Mercer, 2007b], we test the effectiveness of heuristics in JPF, [Groce and Visser, 2002a], on models defined as hard in this paper. The study in [Rungta and Mercer, 2007b] shows that the most-blocked, interleaving and choose-free heuristics are not effective in error discovery on hard models. Note that we test the performance of these heuristics only on the class of subjects for which they are designed. The prefer-thread heuristic consumes more resources in terms of time and memory, as the models get harder, to find errors effectively in a certain class of subjects. The empirical evidence of [Rungta and Mercer, 2007b] shows that the observed R-DFS error density measure of hardness provides a good starting point in defining the quality of the models for evaluating directed model checking techniques.

In future work, we want to identify additional factors that affect the observed R-DFS error density of a model and tie those factors to syntactic constructs in the model. Some interesting factors to study are the depths of the transition graph where the randomized DFS spends a large portion of its search time and the structure of the transition graph derived from the branching factor.

Bibliography

S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *LNCS*, pages 367–381. Springer, 2008. ISBN 978-3-540-78799-0.

Cyrille Artho and Armin Biere. Applying static analysis to large-scale, multi-threaded java programs. In *Proc. ASWEC*, page 68, Washington, DC, USA, 2001. IEEE Computer Society.

David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. pages 324–341, New York, NY, USA, 1996. ACM. ISBN 0-89791-788-X. doi: http://doi.acm.org/10.1145/236337.236371.

T. Ball and S. Rajamani. The SLAM toolkit. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. CAV*, volume 2102 of *LNCS*, pages 260–264, Paris, France, July 2001. Springer-Verlag.

J. Barnat, L. Brim, and J. St. Distributed LTL model checking in SPIN. In *Proceedings of the 8th International SPIN workshop on Model Checking of Software*, pages 200–216. Springer-Verlag New York, Inc., 2001. ISBN 3-540-42124-6.

J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, and P. Šimeček. DiVinE – A Tool for Distributed Verification (Tool Paper). In *Computer Aided Verification*, volume 4144/2006 of *LNCS*, pages 278–281. Springer Berlin / Heidelberg, 2006.

L. Brim, I. Cerna, P. Moravec, and J. Simsa. How to order vertices for distributed LTL model-checking based on accepting predcessors. *Electronic Notes in Theoretical Computer Science*, 135(2):3–18, February 2006.

E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176, Barcelona, Spain, April 2004. Springer. ISBN 3-540-21299-X.

E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2), 1986.

Christoph Csallner and Yannis Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software - Practice and Experience*, 34(11):1025–1050, 2004. ISSN 0038-0644. doi: http://dx.doi.org/10.1002/spe.602.

D. L. Dill. The Murϕ verification system. In *CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification*, pages 390–393, London, UK, 1996. Springer-Verlag. ISBN 3-540-61474-5.

M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property specification patterns for finite-state verification. In Mark Ardis, editor, *Proceedings of the 2nd Workshop on Formal Methods in Software Practice (FMSP-98)*, pages 7–15, New York, 1998. ACM Press. URL `citeseer.ist.psu.edu/article/dwyer98property.html`.

M. B. Dwyer, S. Person, and S. Elbaum. Controlling factors in evaluating path-sensitive error detection techniques. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 92–104, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-468-5. doi: http://doi.acm.org/10.1145/1181775.1181787.

M. B. Dwyer, S. Elbaum, S. Person, and R. Purandare. Parallel randomized state-space search. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 3–12, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2828-7. doi: http://dx.doi.org/10.1109/ICSE.2007.62.

S. Edelkamp and S. Jabar. Large-scale directed model checking LTL. In A. Valmari, editor, *13th International Workshop on Model Checking of Software (SPIN'06)*, volume 3925 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2006.

S. Edelkamp and T. Mehler. Byte code distance heuristics and trail direction for model checking Java programs. In *Proc. MoChArt*, pages 69–76, 2003.

S. Edelkamp, A. L. Lafuente, and S. Leue. Trail-directed model checking. In S. D. Stoller and W. Visser, editors, *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier Science Publishers, 2001a.

S. Edelkamp, A. L. Lafuente, and S. Leue. Directed explicit model checking with HSF-SPIN. In *Proceedings of the 7th International SPIN Workshop*, number 2057 in Lecture Notes in Computer Science. Springer-Verlag, 2001b.

Stefan Edelkamp. Planning with pattern databases. In *Proc. European Conference on Planning*, pages 13–24, 2001.

Dawson Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proc. SOSP '03*, pages 237–252, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-757-5. doi: http://doi.acm.org/10.1145/945445.945468.

Y. Eytani and S. Ur. Compiling a benchmark of documented multi-threaded bugs. In *Proceedings of the Workshop on Parallel and Distributed Systems: Testing and Debugging*, page 266a, Los Alamitos, CA, USA, 2004. IEEE Computer Society. ISBN 0-7695-2132-0. doi: http://doi.ieeecomputersociety.org/10.1109/IPDPS.2004.1303339.

Y. Eytani, K. Havelund, S. D. Stoller, and S. Ur. Towards a framework and a benchmark for testing tools for multi-threaded programs: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(3):267–279, 2007. ISSN 1532-0626. doi: http://dx.doi.org/10.1002/cpe.v19:3.

E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 286.2, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1926-1.

Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proc. POPL*, pages 110–121, New York, NY, USA, 2005. ACM. ISBN 1-58113-830-X. doi: http://doi.acm.org/10.1145/1040305.1040315.

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proc. PLDI*, pages 234–245, New York, NY, USA, 2002. ACM. ISBN 1-58113-463-0. doi: http://doi.acm.org/10.1145/512529.512558.

Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0201575949.

P. Godefroid. Model checking for programming languages using Verisoft. In *Proc. of POPL*, pages 174–186, New York, NY, USA, 1997. ACM. ISBN 0-89791-853-3. doi: http://doi.acm.org/10.1145/263699.263717.

A. Groce and W. Visser. Model checking Java programs using structural heuristics. In *International Symposium on Software Testing and Analysis*, pages 12–21, July 2002a. URL `citeseer.nj.nec.com/groce02model.html`.

A. Groce and W. Visser. Model checking Java programs using structural heuristics. In *Proc. ISSTA*, pages 12–21, 2002b.

Craig Harvey and Paul Strooper. Testing Java monitors through deterministic execution. page 61, Washington, DC, USA, 2001. IEEE Computer Society.

P. Haslum. Model checking by random walk. In *Proceedings of ECSEL Workshop*, 1999. URL `http://citeseer.ist.psu.edu/haslum99model.html`.

K. Havelund. Using runtime analysis to guide model checking of java programs. In *Proc. SPIN Workshop*, pages 245–264, London, UK, 2000. Springer-Verlag. ISBN 3-540-41030-9.

K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder, 1998. URL `citeseer.ist.psu.edu/havelund98model.html`.

T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with Blast. In T. Ball and S.K. Rajamani, editors, *Proc. SPIN Workshop*, volume 2648 of *LNCS*, pages 235–239, Portland, OR, May 2003.

G. J. Holzmann. The design of a distributed model checking algorithm SPIN. FMCAD 2006 Invited Presentation, November 2006.

G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley, 2003.

G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.588521.

Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *SIGPLAN Not.*, 39(4):229–243, 2004. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/989393.989419.

David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12): 92–106, 2004. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1052883.1052895.

C. P. Inggs and H. Barringer. CTL* model checking on a shared-memory architecture. *Formal Methods in System Design*, 29(2):135–155, 2006. ISSN 0925-9856. doi: http://dx.doi.org/10.1007/s10703-006-0008-z.

S. Jabbar and S. Edelkamp. Parallel external directed model checker with linear I/O. In E. A. Emerson and K. S. Namjoshi, editors, *Seventh International Conference on Verification, Model Checking and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 237–251, 2006.

M. Jones, E. Mercer, T. Bao, R. Kumar, and P. Lamborn. Benchmarking explicit state parallel model checkers. In *2nd International Workshop on Parallel and Distributed Methods in Verification*, 2003.

M. D. Jones and E. Mercer. Explicit state model checking with Hopper. In *International SPIN Workshop on Software Model Checking (SPIN'04)*, number 2989 in LNCS, pages 146–150, Barcelona, Spain, March 2004. Springer.

M. D. Jones and J. Sorber. Parallel search for LTL violations. *Software Tools for Technology Transfer*, 7(1):31–42, 2005.

S. Khurshid, C.S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. *Proc. TACAS*, pages 553–568, 2003.

James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/360248.360252.

T. Krazit. Intel pledges 80 cores in five years. CNET News.com, September 2006. URL `http://news.com.com/2100-1006_3-6119618.html`.

Peter Leven, Tilman Mehler, and Stefan Edelkamp. Directed error detection in C++ with the assembly-level model checker StEAM. In *Proceedings of 11th International SPIN Workshop, Barcelona, Spain*, volume 2989 of *Lecture Notes in Computer Science*, pages 39–56. Springer, 2004. ISBN 3-540-21314-7.

E. G. Mercer and M. Jones. Model checking machine code with the GNU debugger. In *12th International SPIN Workshop*, volume 3639 of *Lecture Notes in Computer Science*, pages 251–265, San Francisco, USA, August 2005. Springer.

M. Musuvathi and S. Qadeer. Fair stateless model checking. In *Proc. of PLDI*, pages 362–371, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: http://doi.acm.org/10.1145/1375581.1375625.

M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. *SIGPLAN Not.*, 42(6):446–455, 2007. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1273442.1250785.

Kuntal Nanshi and Fabio Somenzi. Guiding simulation with increasingly refined abstract traces. In *Proc. DAC*, pages 737–742, New York, NY, USA, 2006. ACM. ISBN 1-59593-381-6. doi: http://doi.acm.org/10.1145/1146909.1147097.

C. Pasareanu, M. Dwyer, and W. Visser. Finding feasible abstract counter-examples. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 5(1):34–48, November 2003.

F. M. De Paula and A. J. Hu. An effective guidance strategy for abstraction-guided simulation. In *Proc. DAC '07*, pages 63–68, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-627-1. doi: http://doi.acm.org/10.1145/1278480.1278498.

R. Pelanek. BEEM: Benchmarks for explicit model checkers. *Lecture Notes in Computer Science*, 4595:263, 2007.

R. Pelanek, T. Hanzl, I. Cerna, and L. Brim. Enhancing random walk state space exploration. In *FMICS '05: Proceedings of the 10th International Workshop on Formal methods for industrial critical systems*, pages 98–105, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-148-1. doi: http://doi.acm.org/10.1145/1081180.1081193.

C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *Proc. ISSTA*, pages 15–26, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-050-0. doi: http://doi.acm.org/10.1145/1390630.1390635.

J. H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, 1985.

Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. *ACM SIGSOFT Software Engineering Notes*, 28(5):267–276, September 2003.

N. Rungta and E. G. Mercer. A context-sensitive structural heuristic for guided search model checking. In *Proc. ASE*, pages 410–413, Long Beach, California, USA, November 2005.

N. Rungta and E. G. Mercer. An improved distance heuristic function for directed software model checking. In *Proc. FMCAD*, pages 60–67, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2707-8. doi: http://dx.doi.org/10.1109/FMCAD.2006.5.

N. Rungta and E. G. Mercer. A meta heuristic for effectively detecting concurrency errors. In *Haifa Verification Conference*, Haifa, Israel, 2008.

N. Rungta and E. G. Mercer. Clash of the titans: Tools and techniques for hunting bugs in concurrent programs. In *To Appear in Proceedings of Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD VII)*, Chicago, US, 2009a.

N. Rungta and E. G. Mercer. Guided model checking for programs with polymorphism. In *Proc. PEPM*, pages 21–30, New York, NY, USA, 2009b. ACM. ISBN 978-1-60558-327-3. doi: http://doi.acm.org/10.1145/1480945.1480950.

N. Rungta and E. G. Mercer. Hardness for explicit state software model checking benchmarks. In *Proc. SEFM*, pages 247–256, London, U.K, September 2007a.

N. Rungta and E. G. Mercer. Generating counter-examples through randomized guided search. In *Proceedings of the 14th International SPIN Workshop on Model Checking of Software*, pages 39–57, Berlin, Germany, July 2007b. Springer–Verlag.

N. Rungta and E. G. Mercer. Hardness for explicit state software model checking benchmarks. Technical Report SMC-BYU-0107, Brigham Young University, Department of Computer Science, 2007c.

N. Rungta, E. G. Mercer, and W. Visser. Efficient testing of concurrent programs with abstraction-guided symbolic execution. In *Proceedings of the 16th International SPIN Workshop on Model Checking of Software*, pages 174–191, Grenoble, France, June 2009. Springer–Verlag.

S.J. Russell, P. Norvig, J.F. Canny, J. Malik, and D.D. Edwards. *Artificial intelligence: a modern approach.* Prentice Hall Englewood Cliffs, NJ, 1995.

K. Sen. Race directed random testing of concurrent programs. *SIGPLAN Not.*, 43(6): 11–21, 2008. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1379022.1375584.

Koushik Sen. Effective random testing of concurrent programs. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 323–332, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-882-4. doi: http://doi.acm.org/10.1145/1321631.1321679.

Koushik Sen and Gul Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In *Proc. HVC*, volume 4383 of *LNCS*, pages 166–182. Springer, 2007. ISBN 978-3-540-70888-9.

Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Proc. ESEC/FSE-13*, pages 263–272, New York, NY, USA, 2005. ACM. ISBN 1-59593-014-0. doi: http://doi.acm.org/10.1145/1081706.1081750.

K. Seppi, M. Jones, and P. Lamborn. Guided model checking with a bayesian metaheuristic. *Fundamenta Informaticae*, 70(1-2):111–126, 2006.

O. Shacham, M. Sagiv, and A. Schuster. Scaling model checking of dataraces using dynamic information. *J. Parallel Distrib. Comput.*, 67(5):536–550, 2007. ISSN 0743-7315. doi: http://dx.doi.org/10.1016/j.jpdc.2007.01.006.

H. Sivaraj and G. Gopalakrishnan. Random walk based heuristic algorithms for distributed memory model checking. In *Proceedings of Workshop on Parallel and Distributed Model Checking*, 2003. URL `www1.elsevier.com/gej-ng/31/29/23/141/47/28/89.1.006.pdf`.

N. Sterling. Warlock— a static data race analysis tool. In *USENIX Technical Conference Proceedings*, pages 97–106, 1993.

U. Stern and D. L. Dill. Parallelizing the Murϕ verifier. In O. Grumburg, editor, *Computer-Aided Verification (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 256–267, Haifa, Israel, June 1997. Springer-Verlag.

Scott D. Stoller. Testing concurrent Java programs using randomized scheduling. In *Proc. Second Workshop on Runtime Verification (RV)*, volume 70(4) of *Electronic Notes in Theoretical Computer Science*. Elsevier, July 2002.

J. Tan, G. S. Avrunin, L. A. Clarke, S. Zilberstein, and S. Leue. Heuristic-guided counterexample search in flavers. In *SIGSOFT '04/FSE-12: Proceedings of the*

12th ACM SIGSOFT twelfth International symposium on Foundations of software engineering, pages 201–210, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-855-5. doi: http://doi.acm.org/10.1145/1029894.1029922.

Aaron Tomb, Guillaume Brat, and Willem Visser. Variably interprocedural program analysis for runtime error detection. In *Proc. ISSTA*, pages 97–107, New York, NY, USA, 2007. ACM Press. ISBN 978-1-59593-734-6. doi: http://doi.acm.org/10.1145/1273463.1273478.

W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. ASE*, Grenoble, France, September 2000a.

W. Visser, K. Havelund, G. Brat, and S. Park. Java PathFinder: Second generation of a Java model checker. In G. Gopalakrishnan, editor, *Proceedings of the Workshop on Advances in Verification (WAVE'00)*, July 2000b.

W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

Amy Williams, William Thies, and Michael D. Ernst. Static deadlock detection for Java libraries. In *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, pages 602–629, Glasgow, Scotland, July 27–29, 2005.

C. H. Yang and D. L. Dill. Validation with guided search of the state space. In *35th Design Automation Conference (DAC98)*, pages 599–604, 1998. URL `http://citeseer.nj.nec.com/yang98validation.html`.