UNIVERSITÉ DE MONTRÉAL

TEST DATA GENERATION FOR EXPOSING INTERFERENCE BUGS IN MULTI-THREADED SYSTEMS

NEELESH BHATTACHARYA
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION DU DIPLÔME DE
MAITRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
DÉCEMBRE 2012

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

TEST DATA GENERATION FOR EXPOSING INTERFERENCE BUGS IN MULTI-THREADED SYSTEMS

présenté par: BHATTACHARYA Neelesh
en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de:

Mme NICOLESCU Gabriela, Doct., présidente
M. ANTONIOL Giuliano, Ph.D., membre et directeur de recherche
M. GUÉHÉNEUC Yann-Gaël, Doct., membre et codirecteur de recherche
M. ADAMS Bram, Ph.D., membre

ACKNOWLEDGEMENTS

This research work would not have been possible without the guidance and the help of several individuals who in one way or another contributed and extended their valuable assistance in the preparation and completion of this study. First and foremost, my utmost gratitude goes to Prof Giuliano Antoniol, my Research Supervisor and Prof Yann Gaël Guéhéneuc, my Research Co-Supervisor. They have been inspirational in showing me the path during the hard times. They have been helped me overcome all the hurdles I faced during the completion this research work. I would also like to thank Prof Bram Adams and Prof Gabriela Nicolescu for agreeing to be a part of my Masters Jury.

My dear collegeaues Nasir, Aminata, Soumaya, Ferdaous, Etienne, Zephyrin, Wei, Laleh, Venera, Abdou and Francis provided me their all important support and encouragement throughout my stay in the lab.

The section would not be complete without mentioning the ever-helpful staff of the Informatics and Software Engineering Department, École Polytéchnique de Montreal for their help and support, whenever I needed. I would specially mention Madam Madeleine and Madam Louise for providing me timely administrative help.

Last but not the least, my family, my close friends and the one above all of us, the omnipresent God, for answering my prayers for giving me the strength to plod on despite my constitution wanting to give up and throw in the towel, thank you so much Dear Lord.

RÉSUMÉ

Tester les systèmes multi-thread est difficile en raison du comportement non-déterministe de l'environnement (ordonnanceurs, cache, interruptions) dans lequel ils s'exécutent. Comme ils ne peuvent contrôler l'environnement, les testeurs doivent recourir à des moyens indirects pour augmenter le nombre d'ordonnancements testés. Une autre source de non-déterminisme pour les systèmes multi-thread est l'accès à la mémoire partagée. Lors de leur exécution, les systèmes multi-thread peuvent générer des conditions de courses aux données ou d'interférence dues aux données partagées. La génération de jeux de test en utilisant des techniques basées sur les recherches locales a déjà fourni des solutions aux problèmes des tests de systèmes mono et multi-thread. Mais il n'y a pas encore eu de travaux abordant la question des bugs d'interférence dans les systèmes multi-thread en utilisant les recherches locales. Dans cette thèse, nous étudions la possibilité d'utiliser ces approches afin de maximiser la possibilité d'exposer des bugs d'interférence dans les systèmes multi-thread. Nous formulons notre hypothèse de recherche comme suit : les techniques basées sur les recherches locales peuvent être utilisées efficacement pour générer des jeux de test pour maximiser les conditions d'obtention de bugs d'interférence dans les systèmes multi-thread. Après étude de la littérature, nous avons découvert trois défis majeurs concernant l'utilisation des approches basées sur les recherches locales : C1 : Formuler le problème initial comme un problème de recherche locale, C2 : Développer une fonction de coût adaptée, et C3 : Trouver la meilleure recherche locale (la plus adaptée). Nous procédons d'abord à une étude préliminaire sur la façon dont ces défis peuvent être relevés dans les systèmes mono-thread. Nous pensons que nos résultats pourraient être ensuite applicables aux systèmes multi-thread. Pour notre première étude, nous abordons le problème de la génération des jeux de test pour lever des exceptions de type division par zéro dans un système mono-thread en utilisant des approches basées sur les recherches locales, tout en tenant compte de C1, C2 et C3. Nous constatons que les trois défis sont importants et peuvent être traitées, et que les approches basées sur les recherches locales sont nettement plus efficaces qu'une approche aléatoire lorsque le nombre de paramètres d'entrées croît. Nous traitons ensuite notre principal problème de génération de jeux de test afin de révéler des bugs d'interférence dans les systèmes multi-thread en utilisant des approches basées sur les recherches locales, tout en répondant aux trois mêmes défis. Nous avons constaté que même dans les systèmes multi-thread, il est important de traiter les trois défis et que les approches basées sur les recherches locales surpassent une approche aléatoire lorsque le nombre de paramètres d'entrée devient grand. Nous validons ainsi notre thèse. Cependant, d'autres études sont nécessaires afin de généraliser nos résultats.

ABSTRACT

Testing multi-threaded systems is difficult due to the non-deterministic behaviour of the environment (schedulers, cache, interrupts) in which the multi-threaded system runs. As one cannot control the environment, testers must resort to indirect means to increase the number of schedules tested. Another source of non-determinism in multi-threaded systems is the shared memory access. When executed, multi-threaded systems can experience one of many possible interleavings of memory accesses to shared data, resulting in data race or interference conditions being raised. Test data generation using search-based techniques has provided solutions to the problem of testing single and multi-threaded systems over the years. But to the best of our knowledge, there has been no work addressing the issue of interference bugs in multi-threaded systems using search-based approaches. In this thesis, we perform a feasibility study of using search-based approaches to maximize the possibility of exposing interference bug pattern in multi-threaded systems. We frame our thesis hypothesis as: Search-based techniques can be used effectively to generate test data to expose interference condition in multi-threaded systems. From the related work we found out three major challenges of using search-based approaches: C1: Formulating the original problem as a search problem, C2: Developing the right fitness function for the problem formulation, and C3: Finding the right (scalable) search-based approach using scalability analysis. Before studying multi-threaded systems, we perform a preliminary study on how these challenges can be addressed in single-threaded systems, as we feel that our findings might be applicable to the more complex multi-threaded systems. In our first study, we address the problem of generating test data for raising divide-by-zero exception in single-threaded systems using search-based approaches, while addressing C1, C2, and C3. We find that the three challenges are important and can be addressed. Further, we found that search-based approaches scale significantly better than random when the input search space grows from very small to huge. Based on the knowledge obtained from the study, we address our main problem of generating test data to expose interference bugs in multi-threaded systems using search-based approaches, while addressing the same challenges C1, C2, and C3. We found that even in multi-threaded systems, it is important to address the three challenges and that search-based approaches outperforms random when the input search-space becomes large. Thus we confirm our thesis. However, further studies are necessary to generalize.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABREVIATIONS

SUT	Software Under Test
PUT	Program Under Test
RESP	Reflexive Simulation Platform
SBSE	Search Based Software Engineering
EA	Evolutionary Algorithm
GA	Genetic Algorithm
HC	Hill Climbing
SA	Simulated Annealing
SD	Standard Deviation
CP-SST	Constraint Programming - Software Structural Testing
CSP	Constraint Satisfaction Problem
SSA	Static Single Assignment
CFG	Control Flow Graph
RND	Random
UUT	Unit Under Test
OS	Operating System
PE	Processing Element
FIFO	First In-First Out
SHC	Stochastic Hill Climbing

CHAPTER 1

Introduction

Multi-threading is the process in which multiple threads run concurrently and each thread defines a separate path of execution (McCarthy (1997)). Thus, multi-threading is a specialized form of multitasking. A thread is the smallest sequence of programming instructions that can be managed independently by an operating system (Butenhof (2010)). The popularity of multi-threading has come about due to several reasons: multi-threading increases the responsiveness of a process by letting a thread execute parts of the process when another thread is blocked or busy in a lengthy calculation; all threads of a process share its resources and thus it is more economical to create and context-switch threads. Multi-threading on a multiprocessor system increases concurrency because different parts of a multi-threaded process can be executed simultaneously on different processors.

Multi-threaded systems are difficult to maintain. One of the major risks in multi-threaded systems is related to shared memory access. A shared memory is memory that may be simultaneously accessed by multiple threads with an intent to provide communication among them. In shared-memory multi-threaded systems, the deterministic behavior is not inherent. When executed, such systems can experience one of many possible interleavings of memory accesses to shared data due to the scheduler being not synchronous or limited. An interleaving can result in an indeterministic behaviour leading to data race or interference conditions being raised. An interference condition in multi-threaded software system occurs when (1) two or more concurrent threads access a shared variable, (2) at least one access is a write, and (3) the threads use no explicit mechanism to prevent the access from being simultaneous.

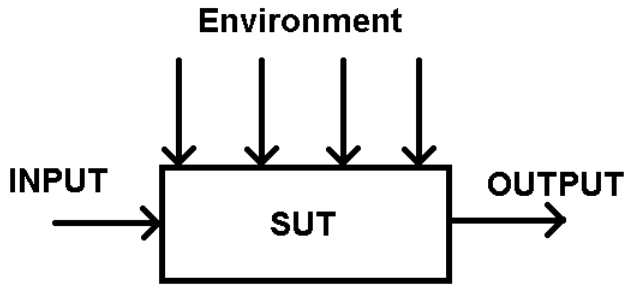

Figure 1.1 Multi-threading environment.

Testing multi-threaded software systems is a commonly adopted approach to expose the data-race and interference bugs. Since multi-threaded environment has non-determinism, it is important to test them so that their actual behaviour is controlled. Figure 1.1 shows a multi-threaded environment where a Software Under Test (SUT) is a multi-threaded system running in a multi-threaded environment. If the environment does not change, a given input to the SUT should produce the desired output. Although one has full control over the data inputs, one cannot control the environment on which the SUT is running. The environment consists of schedulers, cache, interrupts, and other programs running. A small change in any of the parameters, like temperature of the processor, could influence the timings of the thread execution and thus the scheduler. The scheduler, which is a part of the environment in which the system runs, determines the way available threads are assigned to each system. As the environment cannot be modified or controlled, the non-determinism due to the environment, inherent in multi-threaded systems, causes significant challenges by hindering the ability to create multi-threaded systems with repeatable results. The non-deterministic behaviour of the environment of the systems might lead to an erroneous output being produced for a given input and might lead to data-race or interference conditions when uncaught. Theoretically, most of the bugs would be detected by testing a software with all possible input data and environmental configurations, but this is practically not possible. Thus, researchers detect bugs by partitioning the input domain and testing the system with input data generated from each of the partitioned domains. However, the same test data might or might not detect a bug in multi-threaded software systems because of the non-deterministic nature of the environment in which the system is running. As a result, alternative approaches like model checking (symbolic execution Pasareanu et Visser (2009); King (1976)) could be used to find bugs in multi-threaded systems. However, they have their own limitations. Model checking approaches are not scalable and handles loops poorly (Pasareanu et Visser (2009); Saxena *et al.* (2009)).

Consequently, test data generation is a solution to detect interference bugs in multi-threaded systems. Test data generation in program testing is the process of identifying a set of test data that satisfies a given testing criterion (Korel (2003)). It tries to maximize the chances of exposing an error. On the other hand, a random approach might or might not work as it is a blind search. In a small search space, a random approach may suffice, but in a large search space, the probability of hitting the test target by mere chance reduces significantly. Thus, researchers developed test data generation approaches that are more reliable but costly and laborious as well. There is a trade-off between choosing a simple but inefficient vis-a-vis complex but efficient approach.

Automation of test data generation for multi-threaded software systems is complex be-

cause the interaction of the various threads at runtime can lead to various timings and data access problems like data race conditions and interference conditions. Testing a multi-threaded system becomes difficult as testing requires the simulation of the inputs from all the input domain partitions of the program. A new level of complexity is added to the program due to an exponential increase in the number of possible run-time scenarios. Over the years, researchers have tried using different input domain partitions (Vagoun (1996); Vagoun et Hevner (1997)) to try various inputs on different processors to find a bug. Successes have been few because of the lack of knowledge about which specific inputs expose bugs faster.

Search-based approaches have been effective in test data generation for software systems over the years. Search-based approaches use heuristics to find solutions to NP-hard or NP-complete problems at a reasonable computational cost Woeginger (2003); Mitchell et Ternovska (2005). They are not standalone algorithms in themselves, but rather strategies ready for adaptation to specific problems. For test data generation, the strategies involve the transformation of test criteria to objective functions. Test criteria are the benchmarks or standards against which test procedures and outcomes are compared. They are defined to guide the selection of subsets of the input domain. An objective function is an equation to be optimized given certain constraints and with variables that must be minimized or maximized. Objective functions compare solutions of the search with respect to the overall search goal. A fitness function is an objective function that is used to summarise, as a single figure of merit, how close a given solution is to achieving the set goal (McMinn (2011)). Using this information, the search is directed into potentially promising areas of the search space.

Nothing prevents search-based approaches from being applied both in single and multi-threaded systems, but they have not been applied to multi-threaded systems for exposing interference bugs yet. This thesis contributes to the test data generation for multi-threaded systems to maximize the chances of exposing interference bugs using search-based techniques. We want to analyse the effectiveness of search-based approaches in multi-threaded systems.

Therefore, our thesis is that:

Search-based techniques can be used effectively to generate test data to expose interference condition in multi-threaded systems.

In our work, we want to study the effectiveness of using search-based approaches for exposing interference condition in multi-threaded systems. From the literature study of search-based approaches (R. Saravanan et Vijayakumar (2003); Lim *et al.* (2006); Fang *et al.* (2002)), we found three major challenges of using search -based approaches:

1. C1: Formulating the original problem as a search problem.

2. C2: Developing the right fitness function for the problem formulation.

3. C3: Finding the right (scalable) search-based approach using scalability analysis.

Before going to the study of multi-threaded systems, we carry out a preliminary study to see the effect of various search-based approaches for exposing divide-by-zero exceptions in single-threaded systems. We seek to find how challenges C1, C2, and C3 can be addressed when search-based approaches are used for test data generation in single-threaded systems, because we feel that our findings from this study might be applicable to multi-threaded systems. We choose the problem of generating test data for raising divide-by-zero exception for single-threaded systems, because divide-by-zero exception is one of the most frequently occurring and common exception conditions (Leveson (1995); Lions (1996)). We follow the work by (Tracey *et al.* (2000)) and address the major challenges C1, C2, and C3. For addressing challenge C1, we formulate the problem of raising divide-by-zero exceptions as a branch coverage problem as done by (Tracey *et al.* (2000)). We improve the fitness function proposed by (Tracey *et al.* (2000)) to address challenge C2. We address challenge C3 by using various search-based techniques to observe their relative performance and effectiveness to raise divide-by-zero exceptions for single-threaded systems. We see which of the search-based approaches performs the best when the input search space domain (domain of the function to be optimized) is varied. We find that in a small search space, all the approaches performs equally, but varies when the search space increases. In a huge search space, the search-based approaches performs much better than a random approach. Scalability analysis shows which search-based approach performs the best among all.

Using the knowledge that we obtain from the first study regarding the importance of the three challenges C1, C2, and C3, we explore the same challenges in a much more complex situation, *i.e.,* multi-threaded systems. We want to see if the three challenges would still be important in a multi-threaded environment or we would face additional challenges. We choose the problem of generating test data for exposing interference bugs, as interference condition is most frequently occurring and important bug in multi-threaded systems (Park (1999); Software Quality Research Group, Ontario Institute of Technology (2010)). We address challenge C1 by formulating the problem into an equivalent optimization problem by injecting delays in the execution flow of each thread with the purpose of aligning in time different shared memory access events. We propose a novel fitness function supporting the problem formulation to solve the optimization problem to address challenge C2. For addressing challenge C3, we observe the relative performance of the search-based approaches on maximizing the possibility of exposing interference bugs and find the best approach when the input domain was varied. In the following subsections we explain the two studies, namely raising divide-

by-zero exception condition in single-threaded systems and exposing interference bug pattern in multi-threaded systems.

1.1 Raising Divide-by-zero Exception Condition in Single Threaded Systems

A fragment of code that contains a division statement involving a variable is prone to crash during execution when an input value leads the denominator to take the value 0. The system failure would result in an uncaught divide-by-zero exception. Such failures might lead to catastrophic results (financial, human loss), as happened with 1996 Arianne 5 incident during which an uncaught floating-point conversion exception led to the rocket self-destruction 40 seconds after launch. The exception handling code of a system is, in general, the least documented, tested, and understood part of the code (Leveson (1995)), because exceptions are expected to occur only rarely. Thus, it is important to solve the divide-by-zero exception problem to avoid system crash. The problem of raising divide-by-zero exception for single-threaded software systems was reformulated by (Tracey *et al.* (2000)) transforming a target system so that the problem of generating test data becomes equivalent to a problem of branch coverage. The possible divide-by-zero exception-prone statements were transformed into branches using if-else statements. Evolutionary testing was then applied to the transformed system to generate test data by traversing the branches and firing the exceptions. (Tracey *et al.* (2000)) proposed a fitness function that would guide the flow by showing how close a test input data is from reaching the test target (statement which is intended to be reached). The fitness function for firing divide-by-zero exceptions by branch coverage was solely guided by the *branch distance*. *Branch distance* does not provide the knowledge of how far an input data is from reaching the target and thus behaves like a random search. Further, (Tracey *et al.* (2000)) did not show how the various search-based approaches scale when the input search space is varied.

In our first study, we take up the problem of raising divide-by-zero exception and solve the limitations of (Tracey *et al.* (2000)). We meet the three challenges C1, C2, and C3 to solve the problem in the following manner: C1: We use the formulation proposed by (Tracey *et al.* (2000)) to translate the divide-by-zero exception raising problem into a search problem, C2: We frame an improved fitness function based on the problem formulation using *branch distance* and *approach level* to guide the search more effectively, and C3: We use various search-based approaches like Genetic Algorithm, Hill Climbing, Simulated Annealing, Random Search, and Constraint Programming to analyse their scalability when the input search space is varied.

From the study, we find that the three challenges C1, C2, and C3 can be addressed for

test data generation for single-threaded systems. Scalability analysis highlights the variation among the performance of the various search-based approaches while solving the given problem. In our case, Genetic Algorithm performed the best among all the search-based approaches. We suppose the challenges would be even more important for multi-threaded systems when the situation is much more complex. Our second study analyses the challenges in the context of multi-threaded systems.

1.2 Exposing Interference Bugs in Multi-threaded Systems

In multi-threaded systems, various threads interact with each other at run-time. When the interaction happens a key problem becomes the lack of control over which the schedule executes each time a system runs. Due to this lack of control, it becomes difficult to test the system for interference and data-race. Interference bugs are some of the most common bugs in concurrent systems and also the hardest to eradicate (Park (1999); Software Quality Research Group, Ontario Institute of Technology (2010)). Thus, it is important to expose interference bugs as early as possible. In our work, we verify how search-based approaches can be effectively used to maximize the chances of exposing interference bugs in multi-threaded systems. Existing works (Musuvathi *et al.* (2007)) try to solve the problem by checking all the possible thread schedules, which takes a lot of testing effort. For large software systems, it is almost infeasible to test all the possible thread schedules. From the related work, we find that there are three challenges of using search-based approaches: C1, C2, and C3. Because multi-threaded systems are much more complex than single-threaded systems, we want to meet the challenges on them to see if the challenges can be addressed like in our first study (with single-threaded systems). We want to see if we face additional challenges when handling multi-threaded systems, apart from C1, C2, and C3. Thus, we analyse the existing problem of exposing interference bug in multi-threaded software system from a three challenges perspective in the following manner: C1: We modify the problem formulation into a search problem by converting it into an equivalent optimization problem of maximizing the chances of exposing interference bugs in multi-threaded systems by the injection of delays in the execution flow of the threads. To the best of our knowledge, none of the previous work has solved the problem from a search-based perspective or approached the problem by manufacturing specific thread schedules and injecting delays. C2: We formulate a novel fitness function based on the problem formulation. C3: We analyse the scalability of various search-based techniques like Hill Climbing, Simulated Annealing, and Random Search to find the best approach.

We perform our empirical study by running the programs under test (PUTs) on a simu-

lation platform ReSP (Beltrame *et al.* (2009)), which gives us full control over all the components of the system for building a fully-parallel execution environment for multi-threaded systems. We model an environment in which all the common limitations of a physical hardware platform (*i.e.,* the number of processors, memory bandwidth, and so on) are not present and all the the operating system's latencies are set to zero. The PUTs are executed in this environment, exposing only the thread's inherent interactions.

1.3 Plan of the Thesis

The thesis is organized as follows: Chapter 2 describes the various background notions and previous works regarding search-based techniques and their use in test data generation for single and multi-threaded systems. Chapter 3 describes the approach and reports the results obtained for single-threaded system. Chapter 4 describes the approach and reports the results obtained for the multi-threaded systems. Finally, Chapter 5 concludes the thesis and outlines avenues for future work.

CHAPTER 2

Background

In our work, we have generated test data using various search-based approaches like Hill Climbing, Genetic Algorithm, Simulated Annealing to obtain results and thus confirm or not our thesis. In this chapter, we discuss the various approaches that we used, like search-based software engineering, test data generation, metaheuristic approaches and fitness functions. In this thesis, we propose an approach to generate test data for exposing interference bugs in multi-threaded software systems using search-based approaches. Thus, we focus our literature review on the work that has been carried out by researchers in the field of generating test data using search-based approaches in software systems and testing multi-threaded software systems to expose bugs. In this section we summarize the previous works and detail how our approach overcomes their limitations.

2.1 Search Based Software Engineering

Search-based software engineering (SBSE) seeks solution to software engineering problems by reformulating them as search problems and applying metaheuristic search techniques to these new search problems. Meta-heuristic search techniques are used to solve the problems where the solution has to be found in large search spaces. They explore the large search space iteratively and try to improve the current solution by using fitness functions. The final solution obtained by using a metaheuristic search technique may not be the optimal solution (as it is NP complete) but it is usually close to the optimal solution.

The domain of the use of metaheuristic approaches has ranged from software testing (Ferguson et Korel (1996); Jones *et al.* (1998); Mueller et Wegener (1998)), requirements analysis to software development, maintenance and restructuring (Mancoridis *et al.* (1998); Doval *et al.* (1998)). Metaheuristic approaches have gained immense popularity in the field of software testing. The techniques have been used in test data generation of software for many years and this use of a meta-heuristic optimizing search technique to automate or partially automate a testing task (McMinn (2011)) is known as search-based testing. Search-based testing relies on test data generation for software. Test data generation in program testing is the process of identifying a set of test data which satisfies a given testing criterion (Korel (2003)). Test data is generated based on test criteria and test coverage. Test criteria help the tester to organise the test process Spillner (1995). Test criteria are certain conditions

that should be met during testing. They should be chosen in accordance with the available test effort. Test coverage measures are defined as a ratio between the test cases required for satisfying the criteria and those of these which have been executed Spillner (1995). In search-based software engineering, solutions are found based on the framed fitness functions that guide the search. A fitness function is a particular type of objective function that is used to summarise, as a single figure of merit, how close a given design solution is to achieving the set aims. The role of the fitness function is to guide the search to good solutions from a potentially infinite search space, within a practical time limit (McMinn (2011)). The fitness function has two important parts, the *branch distance* and *approach level*. The *branch distance* is taken from the point at which execution diverged from the target structure for the individual. The branch distance is computed for the alternative branch according to the corresponding predicate condition (McMinn *et al.* (2009)). The *approach level* records how many conditional statements are left unencountered by the individual on the way to the target (McMinn *et al.* (2009)).

Following important points emphasizes the importance of the application of search-based approaches in software engineering:

1. Most search spaces involved in software engineering are large and thus the fitness functions should be such that it can be applied effectively in large search spaces.

2. Large search spaces may require thousands of fitness evaluations, that might in turn affect the overall complexity of the system. Thus maintaining a low complexity has always been a software engineering challenge.

3. Optimal solutions for software engineering problems are not evident, thus making the importance of search-based approaches more significant.

There are two main categories of metaheuristic search techniques, local search techniques and evolutionary algorithms. Local search techniques are used on problems that:

1. can be formulated as finding a solution maximizing/minimizing a criterion among a number of candidate solutions and

2. is possible to define a neighborhood relation on the search space.

Local search techniques move from one solution to the other in the search space till finding an optimal (maybe local optimal) solution or reaching the stopping criterion. A local search algorithm starts from a candidate solution and then iteratively moves to a neighbor solution. Typically, every candidate solution has more than one neighbor solution. The choice of

moving to which neighbor depends on the policy defined by the search algorithm. Moving to an improving neighborhood may be the accepted scheme in some algorithms. They may check which neighbor improves the solution more or may choose the first detected neighbor that improves the solution, without verifying the others. In some others moving toward a non improving solution can happen according to a probability function.

Evolutionary algorithms (EAs) are search methods that take their inspiration from biological evolution such as reproduction, mutation, recombination, and selection. Evolutionary algorithms are driven by the basic idea of natural selection based on the fitness of a population. Based on the fitness, some of the better candidates are chosen to seed the next generation by applying recombination and/or mutation on them. Recombination is an operator applied to two or more selected candidates and results in one or more new candidates. Mutation is applied to a candidate and results in a new candidate. Executing recombination and mutation leads to a set of new candidates that compete with the old ones for a place in the generation. The process iterates until a solution is found or previously set computational limit is reached.

Harman *et al.* (2004) described fitness function as an equation that guides the search to improve the solution. Fitness functions are generated according to the problem goal. In fact we translate the problem criteria to the fitness function. The definition of fitness function is not always straightforward. There is no exact approach or standard that shows how a fitness function should be defined for a problem. We are sometimes obliged to define and try several fitness functions to find an appropriate one. When the search algorithm generates a new solution, it evaluates the solution by calculating the fitness. The algorithm uses the calculated fitness either to verify if the solution has been improved and if it can be accepted as the current result (hill climbing) or to rank the solution against the other ones (genetic algorithm).

2.1.1 Genetic Algorithm

A genetic algorithm emulates biological evolution to solve optimization problems. It is formed by a set of individual elements (the population) and a set of biological inspired operators that can change these individuals. According to evolutionary theory only the individuals that are the more suited in the population are likely to survive and to generate off-springs, thus transmitting their biological heredity to new generations. In computing terms, genetic algorithms map numbers to each potential solution. Each solution becomes an individual in the population, and each string becomes a representation of an individual. The genetic algorithm manipulates the most promising strings in its search for an improved solution. The algorithm operates through a simple cycle:

- Creation of a population of strings
- Evaluation of each string
- Selection of the best strings
- Genetic manipulation to create a new population of strings

Figure 2.1 shows how these four stages interconnect. Each cycle produces a new generation of possible solutions (individuals) for a given problem. At the first stage, a population of possible solutions is created as a start point. Each individual in this population is encoded into a string (the chromosome) to be manipulated by the genetic operators. In the next stage, the individuals are evaluated, first the individual is created from its string description (its chromosome) and its performance in relation to the target response is evaluated. This determines how fit this individual is in relation to the others in the population. Based on each individual's fitness, a selection mechanism chooses the best pairs for the genetic manipulation process. The selection policy is responsible to assure the survival of the fittest individuals. The manipulation process applies the genetic operators to produce a new population of individuals, the offspring, by manipulating the genetic information possessed by the pairs chosen to reproduce. This information is stored in the strings (chromosomes) that describe the individuals. Two operators are used: Crossover and mutation. The offspring generated by this process take the place of the older population and the cycle is repeated until a desired level of fitness in attained or a determined number of cycles is reached.

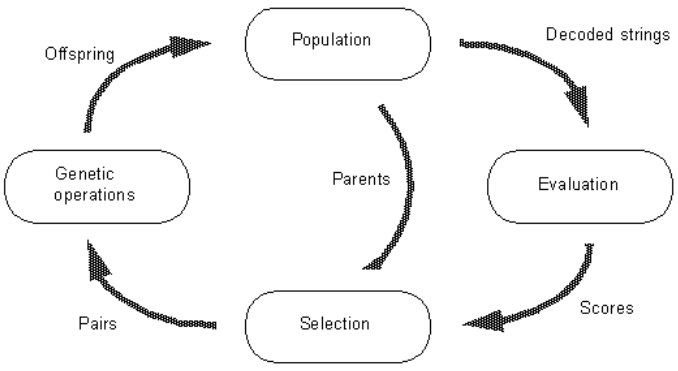

Figure 2.1 Genetic Algorithm

Test data generation using genetic algorithm starts by creating an initial population of n sets of test input data, chosen randomly from the input search space domain of the system

being tested. Each chromosome represents a test set; genes are values of the input variables. In an iterative process, the GA tries to improve the population from one generation to another using the fitness of each chromosome to perform reproduction, i.e., crossover and/or mutation. The GA creates a new generation with the l fittest test sets of the previous generation and the offspring obtained from cross-overs and mutations. It keeps the population size constant by retaining only the n best test sets in each new generation. It stops if either some test set satisfies the condition or the number of iterations reaches the maximum number already set. It uses crossover and mutation operations during the reproduction phase.

Crossover

Crossover relies on the production of two new chromosomes (offspring) by a specific combination of the two parents. If the offspring inherits the best characteristics from both of its parents, the chances of obtaining a fitter offspring is high. The crossover depends on the crossover probability (usually set by the user) and the crossover type. There can be several types of crossover: one-point, two point, uniform, to name a few.

One Point Crossover

In one-point crossover, after randomly choosing a split point and splitting the two parent chromosomes at this point, the new offsprings are created by swapping the tails of the two parent chromosomes. Figure 2.2 provides an example of how offsprings are created using one-point crossover.

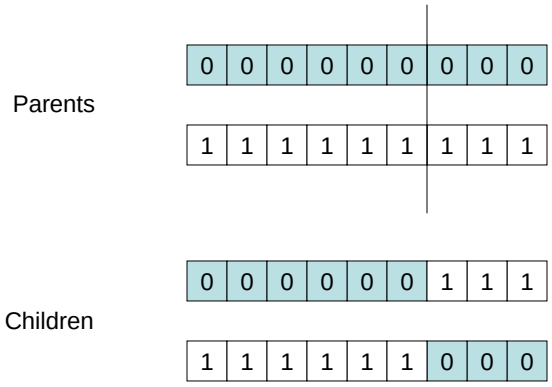

Figure 2.2 One-Point Crossover

Two Point Crossover

In two point crossover, after randomly selecting two split points and splitting the two parent chromosomes at these points, the new offsprings are generated by swapping the segments located between the two split points. Figure 2.3 provides an example of how the offsprings

are created using two point crossover.

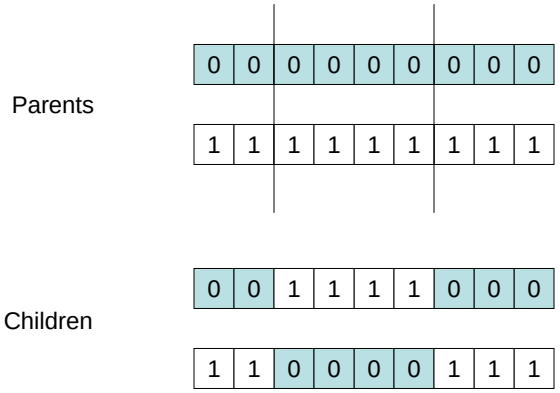

Figure 2.3 Two-Point Crossover

Uniform Crossover

Uniform crossover decides (with some probability, known as the mixing ratio) which parent will contribute each of the gene values in the offspring chromosomes. This allows the parent chromosomes to be mixed at the gene level rather than the segment level (as with one and two point crossover). Probabilistically it decides from which of the parents, the first child should inherit the gene; and thus automatically the second child gets the other parent's gene. For example, if the mixing ratio is 0.5, approximately half of the genes in the offspring will come from parent 1 and the other half will come from parent 2. Figure 2.4 shows how offsprings are created using uniform crossover.

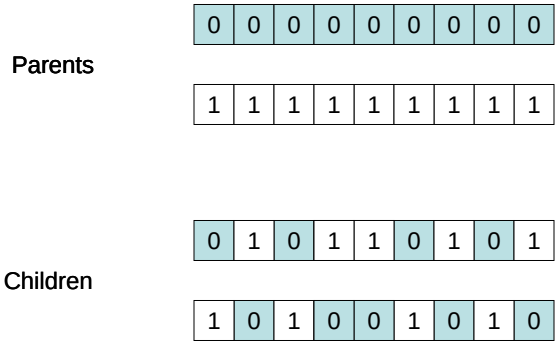

Figure 2.4 Uniform Crossover

Mutation

Mutation is another genetic operator that alters one or some genes value in a chromosome. This can result in generating a new chromosome. Mutation is important because it helps to maintain genetic diversity from one generation of a population by introducing new genetic structures in the population by randomly changing some of its building blocks, helping the algorithm escape local minima traps. Since the modification is totally random and thus not related to any previous genetic structures present in the population, it creates different structures related to other sections of the search space. Mutation probability should be low, lower than or equal to 0.05, because a high values probability drive the search close to a random search. There are different types of mutation such as bit-flip, boundary, uniform, non-uniform and character mutation.

In bit-flip mutation, the mutation operator simply inverts the value of the selected gene. In other words it replaces a 1 with a 0 and vice versa. In boundary mutation the value of the selected genes is replaced by the lower bound or upper bound defined for that gene. The choice of replacing with the upper or lower bound is done randomly. In uniform mutation, selected gene's value is replaced with a value that is selected randomly between the user defined lower and upper bound. For each gene, non-uniform mutation randomly modifies one or more bits. Mutation operator for character genes selects two genes randomly and exchanges their values.

Figure 2.5 shows an example of applying mutation operator.

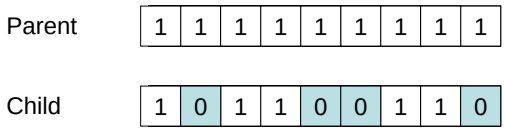

Figure 2.5 Mutation Operator

A schema of GA is shown in algorithm 2.1.1.

Pseudo-code of Genetic Algorithm

begin GA

$g := 0$ { generation counter }

Initialize population P(g)

Evaluate population P(g) // *Evaluate Population Fitness*

while (!Stopping Criterion Reached) // *Max Generations OR No Improvement*

Select Parents

Crossover Parents

Mutate Crossed Population

Evaluate Fitness
Create New P(g) // *Selecting new population from best parents and children*
end while
end GA

2.1.2 Hill Climbing

Hill climbing (HC) is the simplest, widely-used, and probably best-known search technique. HC is a local search method, where the search proceeds from an arbitrary solution to a problem, then attempts to find a better solution by incrementally changing a single element of the solution. If the change produces a better solution, an incremental change is made to the new solution, repeating until no further improvements can be found. Sometimes the technique gets stuck in local optima. In such a situation, the algorithm is restarted from a randomly chosen starting point.

Figure 2.6 explains the technique with an example. Hill Climbing starts with a randomly chosen solution and compares its fitness values with the immediate neighbours to swap its position to be better neighbourhood with the goal of reaching the global maximum, which is the best solution in the entire search space. Consider an initial solution found randomly is the current state 1 and its neighbours shown by the dotted region. Hill Climbing would tend to go uphill (indicated by the arrow) as the immediate neighbours are fitter solutions (better fitness values than itself). Thus the initial random solution would continue to swap its position with the fitter neighbours and would reach the top of the immediate hill to reach the local maximum, which is the best solution in the local neighbourhood. Once at the local maximum, the algorithm would terminate, as on either side of the neighbourhood the possible solutions are worse than itself (worse fitness values) and thus the technique would fail to attain the global maximum. In this situation, hill climbing solves by the problem by restarting the process with another randomly selected initial point. A relaunch allows the algorithm to restart but with a new randomly chosen value as the start point. Consider the new value to be 2 or 3 in Figure 2.6. Both 2 or 3 would continue moving up, due to the presence of fitter neighbours up the hill and would ultimately reach the global optimum and the process would terminate.

A schema of hill climbing with relaunches is shown in algorithm 2.1.2

Pseudo-code of Hill Climbing

loop do
currentNode = randomeStartNode();
loop do
improved = false;

```
        neighborsList = getNeighbors(currentNode) ;
        currEval = EVAL(curentNode);
        indice=0;
        loop do
            if (if(neighborsList.get(indice)) > currEval)
               currentNode = N[indice];
               currEval = EVAL(n);
               improved = true;
               endif
        while(!improved  indice< neighbors.size())
    while(improved);
    store(currentNode); // store best node of the current relaunch
while(!MaxRelaunches);
Return bestNode; //return the best node from all relauches
```

Hill Climbing is used to generate test data in software systems quite frequently. Given a test requirement and fitness function to drive the search, a test input data is randomly chosen and its fitness value is calculated. This value is then compared with the fitness values of the immediate neighbours. As long as their is continuous improvement, the initial value swaps its position with the better neighbours and reaches new positions in the search space. Once there is not further improvement and the process gets stuck in the local optimum, hill climbing is restarted with another randomly generated test input data and the process continues for several relaunches, until the global solution is found or the algorithm reaches the maximum number of iterations.

2.1.3 Simulated Annealing

SA like hill climbing, is a local search method and has a similar philosophy. It is method of solving the problem of global optimization. Like hill climbing, simulated annealing accepts the fitter neighbours and continues swapping its position. The difference of simulated annealing and hill climbing lies in the fact that simulated annealing can accept worst neighbours, depending on the temperature factor. By analogy with the physical process, the algorithm replaces the current solution with a probability depending on the difference in the fitness function of the initial and new solutions and a global option, conventionally known as temperature. It is assumed that the system can go into a state with a large value of the fitness function of the state with the lower the probability is greater, the higher the temperature and the smaller the increment of the fitness function. The temperature slowly decreases at each step of optimization, reducing the probability of a transition of the system to a state

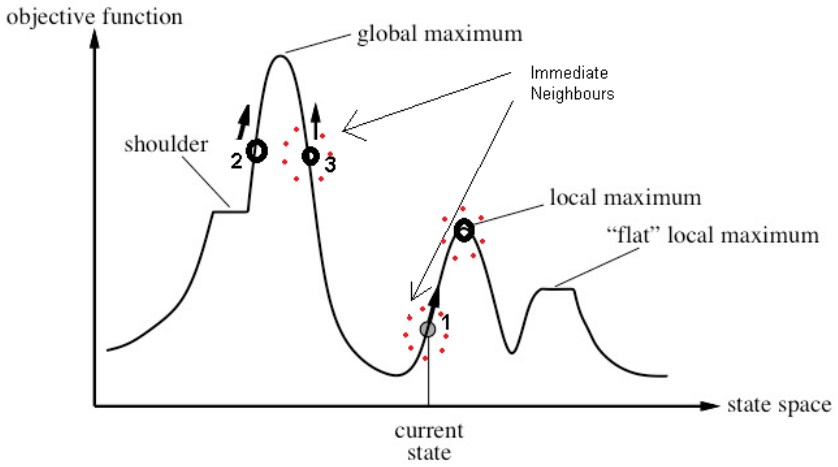

Figure 2.6 Hill Climbing Algorithm

with a large value of the fitness function. In the end, the system freezes in a state with a minimum temperature. The ability to move in the direction of increasing the value of the objective function at a minimum allows the system to search for move out of local minima, in contrast to the simple gradient descent.

Figure 2.7 explains the algorithm. If the initial randomly chosen point is 1 (local maximum) unlike hill climbing, simulated annealing does not need to relaunch with a new start point. Instead it starts accepting and swapping the position with the worse neighbours, depending on the acceptance probability (which depends on temperature). Thus the solution could accept worse solutions probabilistically and continue moving downhill on the left till it reaches the valley (shown by point 2). Once at point 2, solution can move up towards 3 as the immediate neighbours (uphill) are fitter neighbors and reach point 3, the global maximum.

A schema of Simulated Annealing is shown in 2.1.3

Pseudo-code of Simulated Annealing

randomly select currentNode

repeat

$j \leftarrow 0$

repeat

Reduce Temperature T // $T \leftarrow T_{max} \cdot e^{-jr}$

select $nextNode$ in the neighborhood of $currentNode$

if $fitness(nextNode) > fitness(currentNode)$

$currentNode \leftarrow nextNode$

else if random $[0, 1) < e^{\frac{fitness(nextNode) - fitness(currentNode)}{T}}$ // *Worse neighbour*

$currentNode \leftarrow nextNode$

until stopping criterion is reached // *Reaching maximum number of evaluations or reaching minimum temperature*

Simulated annealing has a 'cooling mechanism' that initially allows moves to less fit solutions if $p < m$, where p is a random number in the range $[0 \ldots 1]$ and m, acceptance probability, is a value that decays ('cools') at each iteration of the algorithm. The effect of 'cooling' on the simulation of annealing is that the probability of following an unfavourable move is reduced. This (initially) allows the search to move away from local optima in which the search might be trapped. As the simulation 'cools' the search becomes more and more like a simple hill climb. The choice of the parameters of SA are guided by two equations (Wright (2010)):

$$\begin{aligned} Final\ temperature &= Initial\ temperature \times \alpha^{Number\ of\ iterations} \\ Acceptance\ probability &= e^{\frac{-Normalized\ fitness\ function}{Final\ temperature}} \end{aligned}$$

where *Acceptance probability* if the probability that the algorithm decides to accept solution or not: if the current neighbour is "worse" than previous ones, it can still be accepted (unlike hill climbing) if the probability is greater than a randomly-selected value; α and $Number\ of\ iterations$ are chosen constants.

Test data generation by simulated annealing is done by starting with a randomly chosen test input data as the starting point. The fitness value of this point is calculated from the framed fitness function. The immediate neighbour fitness values are compared with the this fitness value and the start point continues moving towards the better neighbours until it gets stuck in the local optima. Once stuck, out of the worse neighbours at that position, depending on the acceptance probability (which depends on the temperature), the current solution is swapped with one of the worse neighbours and the process continues. If ultimately the algorithm gets stuck in the local optima, the process is restarted with a new randomly chosen test input data. The process stops when the global solution is found or the maximum number of iterations is reached.

2.1.4 Random Search

Random search belongs to the fields of Stochastic Optimization and Global Optimization. Random search is a direct search method as it does not require derivatives to search a continuous domain. It does not rely on using any heuristics to guide the search, unlike genetic algorithm, hill climbing and simulated annealing.

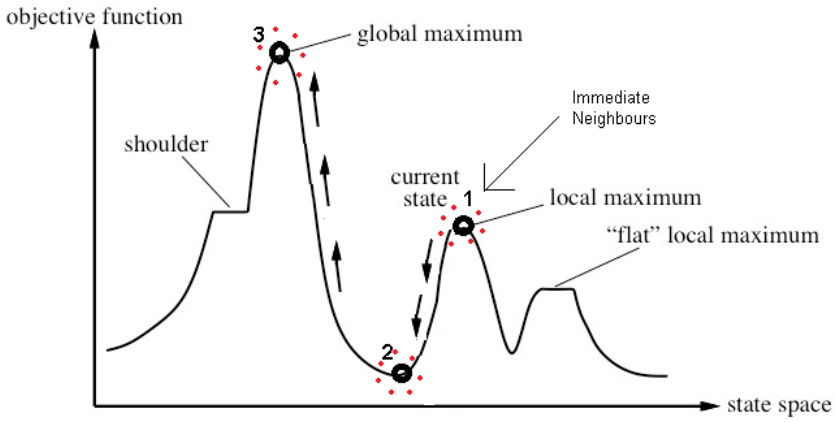

Figure 2.7 Simulated Annealing Algorithm

Test data generation by random search involves generating initial input data randomly to execute the transformed program and try to reach the desired test requirement. The search stops if either the generated test data fires the exception or the number of iterations reaches a maximum number that is set by the user.

2.2 Search Based Software Engineering for Test Data Generation

Over the years researchers have focussed on formulating new approaches for generating test data to satisfy specific code coverage criterion. Sinha *et al.* (2004) proposed an approach to reduce the complexity of a system in the presence of implicit control flow. Ryder *et al.* (2000) used the tool JESP to evaluate the frequency with which exception-handling constructs are used in Java systems. Their analysis found that exception-handling constructs were used in 16% of the methods that they examined. Chatterjee et Ryder (1999) proposed an approach for data flow testing and identified the definition-use associations arising along with the exceptional control flow paths and proposed a new def-use algorithm that could compute the above kinds of def-use relationships (besides other, traditional def-use relationships) in libraries written in a substantial subset of Java/C++. Jo *et al.* (2004) proposed an exception analysis approach for Java, both at the expression and method level, to overcome the dependence of JDK Java compiler on developers' declarations for checking against uncaught exceptions.

One of the most influential work on test data generation for software systems was carried out by (Tracey *et al.* (2000)). They proposed an approach to automatically generate test data for raising divide-by-zero exceptions by (1) transforming the statements containing

exceptions into a branch with guard conditions derived from the possible exception and the statement structure and (2) generating test data to traverse the added branch and thus fire the exception. They proved that the automated testing strategy can be effectively used to provide exception free safety-critical software systems. The objective was to simply check that the generated test data does in fact cause the desired exception. The results they obtained were effective enough to generate test data that could expose the exceptions and cover the exception handling code. But the fitness function they used was solely guided by the branch distance and thus the search was not well guided. Moreover the authors did not compare the effectiveness of their approach with other search-based approaches.

The advent of search-based approaches provided a good solution to test data generation. Local search was first used by (Miller et Spooner (1976)) with the goal of generating input data to cover particular paths in a system. They used read and write operations to perform reachability testing and concluded that when visualizing the cost effectiveness of a concurrent software system, reachability testing proves to be more cost effective than any other non-deterministic testing. Their approach suffered from execution time and space constraints.

Korel (1992) extended the work of (Miller et Spooner (1976)). In his work, the system was initially executed with some arbitrary input. If an undesired branch was taken, an objective function derived from the predicate of the desired branch was used to guide the search. The objective function value, referred to as branch distance (McMinn *et al.* (2009)), measures how close the predicate is to being true. Baresel (2000) proposed a normalization of the branch distance between in [0; 1] to better guide the search avoiding branch distance making approximation level useless. Tracey *et al.* (1998) applied simulated annealing and defined a more sophisticated objective function for relational predicates. The genetic algorithm was first used by (Xanthakis *et al.* (1992)) to generate input data satisfying the all branch predicate criterion.

Evolutionary based software testing was also used to generate test input data (Jones *et al.* (1996); Tracey *et al.* (2000); Wegener *et al.* (2001)). Evolutionary approaches are search algorithms, in particular genetic algorithm, that are tailored to automate and support testing activities. Jones *et al.* (1996) explored the possibility of using genetic algorithm for generating test sets automatically, provided that the test goal is precise and clear. They showed the use of genetic algorithms for testing a variety of programs of varying complexity and compared the performance of genetic algorithm with that of mutation testing. Wegener et al. Wegener *et al.* (2001) enhanced their previous work of Ada based programs Jones *et al.* (1996); Sthamer (1996) by automating the test case design for data and control flow oriented structural test methods. They improved the evolutionary testing by incorporating the approximation level based on the control graph of the program and seeding good quality

initial individuals for ensuring high quality test performance. A survey of evolutionary testing and related techniques was presented by (McMinn (2004)). He analysed the application of metaheuristic search techniques to software test data generation.

In general, search-based approaches face three major challenges:

- C1: Formulating the original problem as a search problem.
- C2: Developing the right fitness function for the problem formulation.
- C3: Finding the right (scalable) search-based approach using scalability analysis.

When the approaches were specifically used to generate test data for software systems, we found the same three challenges existed and they were partially dealt with, though not fully. Tracey *et al.* (2000) proposed a reformulation of the problem of test data generation for raising divide-by-zero exception. They also proposed a fitness function which was not well guided and thus behaved like random search. They did not provide a scalability analysis of the search-based approaches. The effectiveness of the approaches were not validated with input data varying from a very small to very large search space. In our work, we overcome the limitations by validating our approach with input data with varying search space and comparing the performance of our approach with multiple search-based approaches. We address the three challenges C1, C2, and C3 by building our approach on the work of (Tracey *et al.* (2000)). We reformulate the existing problem as (Tracey *et al.* (2000)) and improve the fitness function by using both *branch distance* and *approach level* to guide the search and compare the results with the work of (Tracey *et al.* (2000)). To the best of our knowledge, none of the previous work have formulated the fitness function in this manner to raise divide-by-zero exception in software systems.

2.3 Exposing Bugs in Multi-threaded Systems

In the second part of our work, generating and optimizing test data to fire interference bugs in multi-threaded software systems, we seek to maximize the possibility of exposing data-race and interference bugs. In this section, we summarize the work that has been carried out by researchers for handling and exposing various bug patterns in concurrent software systems.

Artho *et al.* (2003) provided a higher abstraction level for data races to detect inconsistent uses of shared variables and moved a step ahead with a new notion of high-level data races that dealt with accesses to set of fields that are related, introducing concepts like view and view inconsistency to provide a notation for the new properties.

Moving on to research carried out on bug patterns, (Hovemeyer et Pugh (2004)) used bug pattern detectors to find correctness and performance-related bugs in several Java programs

and found that there exists a significant class of easily detectable bugs. They were able to identify large number of bugs in real applications. Farchi *et al.* (2003) proposed a taxonomy for creating timing heuristics for the ConTest tool (Edelstein *et al.* (2003)), showing that it could be used to enhance the bug finding ability of the tool. A timing heuristic is an algorithm that sometimes forces context switches at concurrent events based on some decision function. The features of ConTest was enhanced by incorporating the new heuristics and this resulted in increasing the chances of exposing few concurrent bug patterns. This work was the first to introduce the sleep, losing notify, and dead-thread bug patterns. The authors opened the scope of future research in the area of understanding the relation of bug patterns and design patterns, which is still an open research question.

Eytani et Ur (2004) proposed a benchmark of programs containing multi-threaded bugs for developing testing tools. They asked undergraduates to create some buggy Java programs, and found that a number of these bugs cannot be uncovered by tools like ConTest (Edelstein *et al.* (2003)) and raceFinder (Ben-Asher *et al.* (2003)). Bradbury *et al.* (2006) proposed a set of mutation operators for concurrency programs used to mutate the portions of code responsible for concurrency to expose a large set of bugs, along with a list of fifteen common bug patterns in multi-threaded systems. Carver et Tai (1991) proposed repeatable deterministic testing for solving problems of non-deterministic execution behaviour in multi-threaded software systems. The idea of systematic generation of all thread schedules for concurrent system testing came with works on reachability testing (Hwang *et al.* (1995); Lei et Carver (2006)). Hwang *et al.* (1995) proposed a new testing methodology by combining non-deterministic and deterministic testing. The hybrid approach had distinct advantages over deterministic or non-deterministic testing, when executed separately.

Lei et Carver (2006) used reachability testing to generate synchronization sequences automatically and on-the-fly, without constructing any static models. Reachability testing derives test sequences on-the-fly as the testing process progresses, and can be used to systematically exercise all the behaviors of a system (Hwang *et al.* (1995)). On the fly describes activities that develop or occur dynamically rather than as the result of something that is statically predefined.

There have been several important and efficient tools developed by researchers over years related to exposing various concurrent bugs like data-race, deadlock, automicity violations, to name a few. The VeriSoft model checker (Godefroid (1997)) applied state exploration directly to executable programs, enumerating states rather than schedules. The tool consists of several concurrent processes executing arbitrary C code. ConTest (Edelstein *et al.* (2003)) is a lightweight testing tool that uses various heuristics to create scheduling variance by inserting random delays in a multi-threaded system. The tool can identify bugs earlier in the

testing process and the tests are executed automatically. An early notification of the bugs helps developers and testers take appropriate steps to fix the bug, which might be difficult if the bugs are caught at a later stage of the development cycle. On the flip side, the tool does not work in different environmental conditions. Thus, changing the operating system would reduce the efficiency of ConTest (Edelstein *et al.* (2003)).

CalFuzzer (Joshi *et al.* (2009)) uses analysis techniques to guide schedules toward potential concurrency errors, such as data races, deadlocks, and atomicity violations, while CTigger (Park *et al.* (2009)) exposes atomicity violation bugs in large systems.

One of the most influential tools developed for testing concurrent systems is CHESS (Musuvathi *et al.* (2007)). It overcomes most of the limitations of the tools developed before. What sets CHESS apart from its predecessors is its focus on detecting both safety and liveness violations in large multi-threaded systems. It relies on effective safety and liveness testing of such systems, which requires novel techniques for preemption bounding and fair scheduling. It allows a greater control over thread scheduling than the other tools and, thus, provides higher-coverage and guarantees better reproducibility. CHESS tries all the possible schedules to find a bug, whereas we create the schedules that maximizes the likelihood of exposing an interference bug, if it is present.

The previous works mostly relied on testing all the possible schedules of the multi-threaded software systems, which required lot of testing effort. Moreover, non-determinism in the environment of concurrent systems was not dealt with effectively, resulting in generating test data that might or might not find a bug. We overcome both the limitations in our work by formulating the existing problem as a search problem and addressing the three challenges C1, C2, and C3. We reduce the testing effort by not trying all the possible schedules for testing. We manufacture specific schedules by injecting delays in the execution of the threads. We use search-based approaches to optimize the delays and maximize the chances of exposing an interference bug pattern. Moreover, we take care of the non-determinism aspect by running our multi-threaded on ReSP (Beltrame *et al.* (2009)), a simulation platform so that we can tune the necessary parameters to avoid non-determinism in the concurrent environment.

CHAPTER 3

Raising Divide-By-Zero Exception via Branch Coverage

In this chapter, we focus on software systems which are not multi-threaded and propose an approach that generates test data to fire divide-by-zero exception in single-threaded software systems. As a pre-requisite for generating test data for exposing interference bugs in multi-threaded software systems using search-based approaches, in this chapter, we evaluate the importance of meeting the major challenges of using search-based approaches in single-threaded software systems. We formalise our approach based on the work of (Tracey *et al.* (2000)).

3.1 Challenges of Using Search Based Approaches in Single Threaded Systems

From the related work, we identified three major challenges of using search-based approaches in single-threaded software systems from the existing literature: C1: Formulating the original problem as a search problem, C2: Developing the right fitness function for the problem formulation, and C3: Finding the right (scalable) search-based approach using scalability analysis. Tracey *et al.* (2000) generated test data to raise divide-by-zero exception for single-threaded software systems by re-formulating the existing problem. The existing problem was that a test data leading to the denominator of a divide-by-zero prone statement to be zero would lead to system crash. Tracey *et al.* (2000) proposed a reformulation of the problem. Their approach resulted in the search to behave like a random search due to the usage of a fitness function that could not provide proper guidance to the search. Moreover the scalability of the various search-based approaches was not shown. Previous works by (Mattfeld (1999); Lanzelotte *et al.* (1993); Sivagurunathan et Purusothaman (2011)) have explored the scalability of search-based approaches and have shown the importance of varying the search space domain to analyse the effectiveness of these approaches in searching a value in the neighbourhood. Thus, in this chapter, we evaluate the importance of addressing the challenges on single-threaded software systems and overcoming the limitation of the approach proposed by (Tracey *et al.* (2000)).

3.2 Divide-By-Zero Exception

A divide-by-zero exception occurs in a piece of code when a division (at any statement of the code) involves a 0 at the denominator. In this case, a divide-by-zero exception is raised

at run-time and the program stops abruptly. Thus it is an unwanted condition because the program flow is terminated and the program cannot produce the desired output. It is one of the most well known and frequently encountered exception (Leveson (1995); Lions (1996)). Following is an example of a piece of Java code that contains a divide-by-zero prone statement in line 4. Once the flow reaches line 4, if the variable z takes a value of 1, the denominator becomes 0 and the exception will be raised immediately, resulting in terminating the program.

```
1    int z, x=0;
2    if (x<10)
3         if (x>3)
4              x = 1/(z-1);
5    else
6         x = 4;
```

3.3 Approach

Our approach seeks to overcome the limitation of the approach proposed by (Tracey *et al.* (2000)) by using a guard condition around the statement involving the divide-by-zero operation, to transform the problem into an equivalent branch coverage problem and reformulate the fitness function they proposed. We use both the *branch distance* and *approach level* to guide the search, unlike the approach proposed by (Tracey *et al.* (2000)). In this section, we explain our approach in detail by the means of a working example, in the context of the three major challenges mentioned before.

3.3.1 Challenge 1: Reformulation of the Problem

We follow the work of (Tracey *et al.* (2000)) to transform the problem of generating test input data to raise divide-by-zero exceptions into a branch coverage problem. This transformation essentially consists of wrapping divide-by-zero prone statements with a branch statement (an *if*), whose condition corresponds to the expression containing the possible division by zero. Consequently, satisfying the *if* through some branch coverage is equivalent to raising the divide-by-zero exception. For example, let us consider the following fragment of code:

```
1    int z, x=4;
2    if (Z>1 AND Z<=5)
3         return z;
4    else
```

```
5       return (x*4)/(z-1);
```

a divide-by-zero exception would be raised when z equals to 1 at line 5.

It is usually difficult to generate test data targeting a specific condition by obtaining appropriate variable values. We transform the code fragment above into a semantically-equivalent fragment in which the expression possibly leading to an exception becomes a condition. Then, it is sufficient to satisfy the new condition to obtain test input data raising the exception, as in the following fragment:

```
1     int z, x=4;
2     if (Z>1 AND Z<=5)
3         return z;
4     else
5.1       if (Z == 1)
5.2           print "Exception raised";
5.3       else
5.4           return (x*4)/(z-1);
```

where we transform the divide-by-zero prone statement at line 5 into the lines 5.1 to 5.4.

In general, such a transformation may not be trivial and different types of exceptions may require different types of transformations.

3.3.2 Challenge 2: Fitness Reformulation

We reformulate the fitness function based on the problem reformulation. As per the working example, to efficiently generate test data to expose the exception, it is not sufficient to reach line 5.1 using the *approach level* because between two test input data, both reaching line 5.1, we would prefer the data making the condition at line 5.1 true and thus reaching line 5.2. So we need to reformulate the fitness function.

Consequently, our fitness function is an additive function using both the *approach level* and the normalized *branch distance* (McMinn *et al.* (2009)), where the normalized branch distance is defined by Equation 3.1 and used in the fitness function defined by Equation 3.2.

$$Normalized\ branch\ distance = 1 - 1.001^{-branch\_distance} \tag{3.1}$$

$$Fitness\ function = Approach\ level + Normalized\ branch\ distance \tag{3.2}$$

3.3.3 Challenge 3: Scalability Analysis

We want to analyse the effectiveness of the search-based approaches when the input search space domain is varied from as small to large. The input to the program is the test input data. Thus for analysing and comparing the scalability of the approaches, we vary the test input search space domain from small to large to see their effect on the performance of the approaches. The domains have been varied from $[-100; +100]$ to $[-50,000; +50,000]$ for all the input variables. To enhance our search strategies and generalize our results, we also proposed three variants of hill climbing (for searching neighbours with good fitness values) and a novel approach for constraint programming, before comparing their respective performances in terms of the scalability of the various approaches. Often, to avoid local optima, the hill climbing algorithm is restarted multiple times from a random point (also called stochastic hill climbing). We have drawn inspiration from this idea and proposed three variants of hill climbing for searching neighbours with good fitness values.

Variant 1: The HC1 variant generates, for any input variable involved in the denominator of the exceptions raising statements, an immediate neighbour of the input data as the sum of the the current value of the variable with a randomly-generated value drawn from a Gaussian distribution with zero mean and an initial standard deviation (SD) of ten, which we choose after several trials to have neighbourhoods that are not too small or large.

If after a given number of moves in the neighbourhoods, the fitness values of the neighbours are always worse than the current fitness value, then we change the value of the SD to a larger value to expand the neighbourhood and give the algorithm an opportunity to get out of the, possible local optimum.

Variant 2: In the HC1 variant, the values of the SD may change at run-time. We observed that HC1 does not always improve the search and may lead to a slow search-space exploration. Thus, to avoid getting "trapped" in a specific region of the space, we define the HC2 variant that forces the search to take a jump away from unsuccessful neighbourhood in an attempt to move into a more favourable neighbourhood, similarly to HC with random restarts.

For a given SD value, if the search does not improve for a given number of iterations, instead of changing SD, we force a jump to another neighbourhood using a large value and HC reiterates its process. The "length" of a jump and the number of jumps depend on the search space. For example, a search space of $[-10,000; +10,000]$ would be likely covered with 40 jumps of lengths 500. As with the previous variant, this variant goes on until it reaches a maximum number of iterations or generates test data firing the targeted exception.

Variant 3: The HC3 variant is a combination of HC1 and HC2 variants. With HC3, we store the fitness values of the best neighbour of all previously-visited neighbourhoods before jumping to another neighbourhood. After having visited a given number of neighbourhoods as in HC2, HC3 returns to the "best" one, the neighbourhood with the best fitness value among all recorded values, and then increases the SD by 25 (making the SD 35), as in HC1, to visit more of this neighbourhood. As with the previous variants, this variant goes on until it reaches a maximum number of iterations or generates test data firing the targeted exception.

Constraint Programming

Constraint programming for software structural testing (CP-SST) is a generic technique for test data generation to reach a specific target or to satisfy a test coverage criteria, for proof post-condition, or for counter-example generation.

The main idea of CP-SST is to convert the program under test and the test target into a constraint solving problem (CSP) and to solve the resulting CSP to obtain test input data. The first step of CP-SST consists of transforming the program under test into the static single assignment (SSA) form. The second step consists of modelling the program control flow graph (CFG) as a preliminary CSP. CP-SST begins by generating the CFG that features an independent node for each parameter and global variable, each control statement, each block of statements, and each join point. CP-SST labels edges among nodes depending on the origin node: an edge outgoing from a statement node is labelled by 1, an edge outgoing from a condition node is labelled by 1 if the decision is positive and -1 if the decision is negative. Then, CP-SST generates the preliminary CSP by translating each node into a CSP variable whose domain is the set of labels of its outgoing edges, except for join nodes that take their domains from the joint nodes.

In the third step, CP-SST uses the preliminary CSP, the SSA form, and the relationships among nodes and their statements to create a new global CSP, which it then solves to generate test input data or return a failure if no solution can be reached.

3.4 Empirical Study

The *goal* of our empirical study is to compare our testability transformation against previous work and identify the best technique among the five presented in Section 3.3 using synthetic as well as real systems to generate integer test input data to fire divide-by-zero exceptions. The *quality focus* is the performance of the proposed hill climbing variants and other meta-heuristics and constraint programming techniques to raise divide-by-zero exceptions. The *perspective* includes researchers and software engineers working in search-based

Table 3.1 Details of the systems under test and tested units

Systems	Versions	Class Names	LOCs	Numbers of Exceptions	Bug Tracking Numbers
Tracey's code	N/A	F	13	1	N/A
Eclipse	2.0.1	GridCanvas	10	2	205772
Android	2.0	ProcessStats	41	3	Unavailable

software testing looking to generate test data for firing exceptions in the code. The *context* of our research includes three case studies: one synthetic program and two real software systems, namely, *Eclipse* and *Android*. Table 1 summarizes the three software systems and the selected methods/functions for testing and the corresponding class names having one, two, and three divide-by-zero exception statements, respectively.

We seek answers to four research questions:

RQ1: Based on the fitness function we use, which of the three proposed hill climbing variants is best suited to raise a divide-by-zero exception and what is the measure of its effectiveness?

RQ2: Which of all the meta-heuristic techniques is best suited to raise a divide-by-zero exception and what is the measure of its effectiveness? (Retaining the best-suited hill climbing variant from RQ1.)

RQ3: Which of Tracey's fitness function and the fitness function we used, is best suited to raise a divide-by-zero exception and what is the measure of its effectiveness? (Retaining the best-suited meta-heuristic from RQ2.)

RQ4: Which of the best-suited meta-heuristic technique and of the CP-SST is best suited to raise a divide-by-zero exception and what is the measure of its effectiveness? (Retaining the best-suited meta-heuristic from RQ2.)

3.4.1 Choice of the Comparison Measure

HC (in the three variants), SA, GA, and RND can be compared with one another using their numbers of fitness evaluations to fire some divide-by-zero exception. CP-SST is based on a completely different paradigm than the meta-heuristic techniques. Thus, CP-SST cannot be compared with the other techniques using the numbers of fitness evaluations and we use execution times of the different techniques for comparison. We consider the approach requiring less execution time to reach a target to be "better" to generate test data for firing divide-by-zero exceptions.

3.4.2 Choice of the Targeted Exceptions

We selected three methods in three classes of three different systems, for a total of six possible target exceptions. In this chapter, we report the results of our empirical study for the target exceptions that are chosen to lead to the worst performance for all the techniques, among the six possible targeted exceptions and called in the following units-under-test, UUT.

3.4.3 General Parameters of the Techniques

We chose different ranges of values for each input variable to analyse the performance of all the techniques to deal with values ranging from very small to very large. The domains have been varied from $[-100; +100]$ to $[-50,000; +50,000]$ for all the input variables. Reaching a success, raising the targeted divide-by-zero exception in the UUT, is the stopping criterion as well as a number of evaluations of 1000000. We repeated each computations 20 times to analyse the diversity in the observed values and conduct statistical tests. Table 3.2 details the values used.

3.4.4 Specific Parameters of the Techniques

We use the following parameters for the techniques (we do not report techniques which do not use any particular parameters):

Hill Climbing: For the first variant, we use a parameter *checkStagnation* to control the number of iterations before changing neighbourhood. We use 100, if no improvement occurs in a neighbourhood after 100 iterations, HC1 changes neighbourhood. We use 100 as SD because it led to better performance than other values.

In the second variant, we also use *checkStagnation* parameter. We use two other important parameters: *gaussianJumpLength* and *numberOfJumps* depicting the length of the "jump" and the number of jumps, respectively. We found the values of these two parameters by trial-and-error runs.

In the third variant, we also use *deviationValue*, the value of SD when the technique returns to the best neighbourhood. Table 3.3 depicts the parameter values.

Simulated Annealing: We base our choice of the initial temperature, α, and the number of iterations on several experiments in which we varied the initial temperature. Table 3.4 shows the values used.

Genetic Algorithm: We used single-point cross-over, bit-flip mutation, and binary-tournament selection. We choose the binary-tournament selection because its complexity is lower than that of any other selections and provides more population diversity to the cross-over operator than others (Zhang et Kim (2000)). Table 3.5 shows the various values.

3.5 Results

Figure 3.1, 3.2, and 3.3 reports the box plots of the number of fitness evaluation needed to raise a divide-by-zero exception for the Tracey exemplary code, Eclipse, and Android UTTs, respectively. We did not include results for RND as it performs always substantially worse then even the slowest HC1 variant.

We observe that HC3 is, in all cases, better then the other two hill climbing variants but in all cases is also performing substantially worse than SA and GA. Overall, Figure 3.1, 3.2, and 3.3 support the observation that GA is the most effective meta-heuristic technique as far as these UTTs are concerned.

Table 3.6, 3.7, and 3.8 reports the t-test values comparing the numbers of fitness evaluations needed by the different search techniques as well as the Cohen d effect size (Cohen (1988)).The effect size is defined as the difference between the means of two groups, divided by the pooled standard deviation of both groups. The effect size is considered small for $0.2 \leq d < 0.5$, medium for $0.5 \leq d < 0.8$, and large for $d \geq 0.8$ (Cohen (1988)).We chose the Cohen d effect size because it is appropriate for our variables (ratio scales) and given its different levels (small, medium, large) easy to interpret.

As expected from Figures 3.1, 3.2, and 3.3, the t-test and Cohen d effect size results support with very strong statistical evidence the superiority of HC3 over HC1 and HC2 as well as the superiority of GA over SA and HC3.

Box-plots as well as tables clearly support the superiority of GA over the other techniques. Overall, **we answer RQ1 by stating that HC3 performs better than HC1 and HC2 with a large effect size.** Furthermore, **we answer RQ2 by stating, with a large effect size also, that GA out-performs the other techniques.**

We compare our fitness function with that proposed by (Tracey *et al.* (2000)) by using two GA implementations, one using Tracey's fitness function and another using the fitness function explained before. We compare the two fitness functions in terms of the required numbers of fitness evaluations for all the UUTs in Figure 3.4 to reach the targeted exceptions. Figure 3.4 shows that our testability transformation allows the GA to reach the targeted exceptions in much less numbers of evaluations. Consequently, **we answer RQ3 by stating that, in comparison to Tracey's fitness function, our testability transformation**

Table 3.2 General Parameters

Input Domain	$[-100; +100]$ **-** $[-50,000; +50,000]$
Max # iterations	1000000
# Computations	20

Table 3.3 Hill Climbing Parameters

Variants	**CS**	**SD**
S1	100	-
S2	100	-
S3	100	35

dramatically improves the performance of a GA technique.

Finally, Table 3.9 reports average and standard deviations of the execution times for twenty experiments on the three UUTs for both GA and CP. For the given UUTs, it is clear that CP out-performs GA in term of execution times. This result may be due to the size of the UUTs, which are relatively small, and to the structure of the condition to satisfy. More evidence is needed to verify if the averages in Table 3.9 represent a general trend. Yet, on the selected UUTs, **we answer RQ4 by claiming that the CP-SST technique out-performs the best of the meta-heuristic techniques, GA.**

3.6 Study Discussions and Threats to Validity

3.6.1 Discussions

We presented the results of three UUTs to answer the four research questions. The other four UUTs from the same systems exhibit the same trends as the ones reported in this thesis, thus adding more evidence to our answers.

We also evaluated the performance of our testability transformation with respect to the one proposed by (Tracey *et al.* (2000)) in terms of required numbers of fitness evaluations. The results showed the importance of having both *approach level* and *branch distance* in the fitness function, as opposed to the one proposed by (Tracey *et al.* (2000)) which uses only the *approach level*.

3.6.2 Threats to the Validity

We now discuss the threats to the validity of our study.

Threats to *construct validity* concern the relationship between theory and observation. In our study, these threats can be due to the fact that one of the UUT is a synthetic code, even though previously-used to exemplify and study the divide-by-zero exception (Tracey *et al.* (2000)), and thus might represent real code. However, we extracted the two other UUTs

Table 3.4 Simulated Annealing Parameters

Params.	**Chosen Values**
Temper.	0.5-50 (20)
α	0.8-0.995 (0.99)
# Iter.	10-500 (100)

Table 3.5 Genetic Algorithm Parameters

Operators	**Type**	**Prob.**
Crossover	Single Point	0.9
Mutation	Bit Flip	0.09
Selection	Binary Tournament	-

from real-world systems (Eclipse and Android) and the method containing the divide-by-zero exceptions has been documented in the Eclipse issue tracking system. Finally, the code excerpt (Tracey *et al.* (2000)) as well as the Eclipse and Android studied methods contain multiple possible divide-by-zero statements and, in all cases, we focused on the statements leading to the worst performances, the most deeply-nested statements.

Threats to *internal validity* concern external factors that may affect an independent variable. We limited the bias of intrinsic randomness of our results by repeating each experiment 20 times and using proper statistics to compare the results. We have calibrated the HC (HC1, HC2, and HC3), SA, and GA settings using a trial-and-error procedure. We chose the values of the parameters of the techniques, such as *checkStagnation*, *gaussianJumpLength* and so on, after executing the techniques several times and evaluating their performance on a toy program. We also chose the cross-over and mutation operators by doing a small study on the same toy program: although we found evidence of the superiority of specific operators, it could happen that (1) studies on different systems would lead to a different choice of cross-over and mutation operators and (2) the obtained calibration may not be the most suitable for our subject systems.

Threats to *conclusion validity* involve the relationship between the treatment and the outcome. To overcome this threat, we inspected box-plots, performed t-tests, and evaluated the Cohen d effect sizes.

Threats to *external validity* involve the generalization of our results. We evaluated the testability transformation on UUTs from the work of (Tracey *et al.* (2000)) and two different Java systems. The sample size is small and, although for Eclipse code it corresponds to a documented bug, a larger evaluation is be highly desirable.

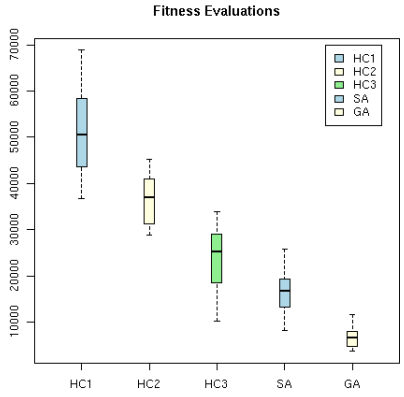

Figure 3.1 Comparison on Tracey (Tracey *et al.* (2000)) UUT of the different search techniques (input domain $[-50,000; +50,000]$)

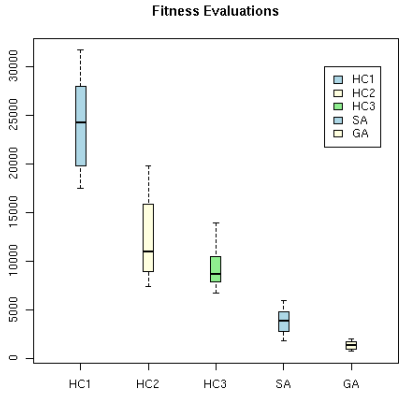

Figure 3.2 Comparison on Eclipse UUT of the different search techniques (input domain $[-50,000; +50,000]$)

Table 3.6 Results of t-test and Cohen d effect size for Tracey (Tracey *et al.* (2000)) UUT

Comparisons	p-values	Cohen d values
HC1-HC2	9.261e-10	2.55246
HC1-HC3	6.376e-16	5.951003
HC2-HC3	6.868e-08	2.428475
HC3-SA	7.049e-14	4.147889
HC3-GA	2.2e-16	8.223645
SA-GA	8.763e-15	6.254793

Table 3.7 Results of t-test and Cohen d effect size for Eclipse UUT

Comparisons	p-values	Cohen d values
HC1-HC2	3.245e-10	2.682195
HC1-HC3	7.167e-13	4.111778
HC2-HC3	0.003998	0.9912142
HC3-SA	2.387e-11	3.239916
HC3-GA	1.933e-13	5.258295
SA-GA	9.989e-09	2.694495

3.7 Conclusion

In this chapter, we presented an approach to address the problem of generating test data for raising divide-by-zero exceptions in single-threaded software systems. We evaluated and analysed three major challenges of using search-based approaches in single-threaded software systems: C1: Formulating the original problem of raising divide-by-zero exception as a search problem, C2: Developing the right fitness function for the problem formulation, and C3: Developing the right fitness function for the problem formulation. We reformulated the existing problem of test data generation raising divide-by-zero exceptions using a testability transformation to generate test input data to raise divide-by-zero exceptions in software systems. We reformulated the fitness function used by (Tracey *et al.* (2000)) by combining the *branch distance* and *approach level* which was used by hill climbing, simulated annealing, and genetic algorithm. We compared the performance of hill climbing, simulated

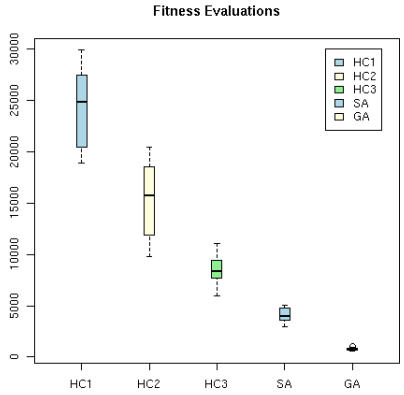

Figure 3.3 Comparison on Android UUT of the different search techniques (input domain $[-50,000; +50,000]$)

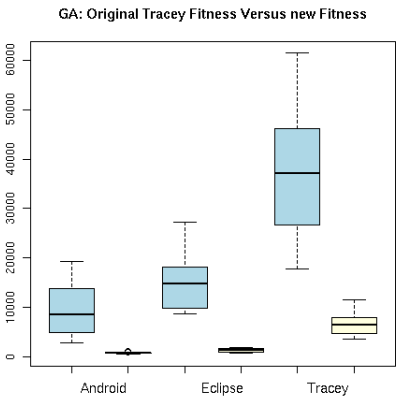

Figure 3.4 GA comparison against Tracey's original fitness (Tracey *et al.* (2000)) versus the fitness function we used (input domain $[-50,000; +50,000]$)

Table 3.8 Results of t-test and Cohen d effect size for Android UUT

Comparisons	p-values	Cohen d values
HC1-HC2	1.438e-06	1.894204
HC1-HC3	2.981e-12	3.345377
HC2-HC3	7.438e-08	2.12037
HC3-SA	0.0003169	1.266088
HC3-GA	1.283e-10	3.481401
SA-GA	4.531e-09	2.696728

Table 3.9 Comparison of GA against CP-SST in terms of average execution times (ms) and standard deviations for all UUTs

	Tracey's Code	Eclipse	Android
GA	8.067/1.439	2.129/1.149	1.926/1.177
CP	1.035/0.0135	0.01/0	0.01/0

annealing, genetic algorithm, random search, and constraint programming when using this fitness function varying the search space from very small to very large space. Further, we also proposed three hill climbing variants to improve basic hill climbing search. Finally, we chose the best meta-heuristic technique (genetic algorithm) and compared its performance with that of constraint programming in terms of execution time.

We validated our testability transformation and compared the search technique on three software units: one synthetic code fragment taken from (Tracey *et al.* (2000)) and two methods extracted from Eclipse and Android, respectively. While comparing the meta-heuristic techniques, genetic algorithm performed best in terms of the number of required fitness evaluations to reach the desired target for all the three units under test. Then, constraint programming out-performed the genetic algorithm in terms of execution times for all the three case studies.

We found that the three challenges are important and can be addressed for single-threaded

systems. With this knowledge of the three challenges, we move on to meeting these challenges in the more complex study of using search-based approaches in multi-threaded systems.

CHAPTER 4

Optimizing Thread Schedule Alignments to Expose Interference Bugs

In the previous chapter, we analysed the three major challenges of using search-based approaches in single-threaded software systems and explored the problem of generating test data to raise divide-by-zero exceptions. We showed how search-based approaches can be effectively used to fire divide-by-zero exception in single-threaded software systems. In this chapter, we meet the three challenges to explore the problem of exposing interference bugs in multi-threaded systems using search-based approaches and provide evidence to confirm or not our thesis hypothesis based on the effectiveness of the search-based approaches in solving the problem. We use the knowledge from the previous study and deal with the problem of generating test data to maximize the possibility of exposing interference bugs in multi-threaded systems using search-based approaches like stochastic hill climbing and simulated annealing.

Due to various interferences among threads, concurrent systems are problematic to develop and test, in spite of their many benefits with respect to performance. Concurrency is built around the notion of multi-threaded systems. A thread is defined as an execution context or a lightweight process having a single sequential flow of control within a program .A thread is typically created using a `start()` method and terminates once it finishes running. The state of a thread changes between being runnable or not runnable. Inserting delays in the execution of a thread is an efficient way of disrupting its normal behavior. The inserted delay shifts the execution of the subsequent statements by the time equal to the delay. If, by doing so, an event in one thread overlaps with another event in another thread, in the same time frame, there is a possibility of an interference bug.

Bradbury *et al.* (2006) defined fifteen possible bug patterns that could affect the normal behavior of threads and cause severe problems to concurrency. Out of them, we considered the interference bugs because they are the most commonly encountered bugs and can lead to undesired behavior, if uncaught (Park (1999); Software Quality Research Group, Ontario Institute of Technology (2010)). An interference bug occurs when (1) two or more concurrent threads access a shared variable, (2) at least one access is a write, and (3) the threads use no explicit mechanism to prevent the access from being simultaneous.

4.1 Interference Bug Sequence Diagrams

Figure 4.1 shows a sequence diagram with the normal behavior of a PUT, before the introduction of any delay. Figure 4.2 shows the sequence diagram describing the behavior of the same PUT after a delay has been injected, exposing an interference bug.

In Figures 4.1 and 4.2, a master thread creates child threads and shares some data with them. In the normal, expected behavior (without the bug), illustrated in Figure 4.1, each child thread accesses the shared data sequentially, so that every thread has the last version of the data when it reads it. When we inject a delay just before a child writes its modification to the shared data, as shown in Figure 4.2, another thread reads a wrong piece of data and may produce incorrect results.

The interference bug may also appear when two write events happen at the same time. To expose this bug, we intend to find out the maximum number of write pairs aligned in the same time frame in two threads. Thus, we frame the issue of identifying interference bugs as an optimization problem, which we solve using optimization algorithms.

4.2 Approach

Given a multi-threaded system, the interleaving of threads depends on the hardware (number of processors, memory architecture, etc.) and the operating system. There are as many schedules as there are environmental conditions, schedule strategies, and policies of the operating system when handling threads. Among these schedules, there could be a subset leading to the expression of one or more interferences. In general, the exhaustive enumeration of all possible schedules is infeasible, and in an ideal situation, all threads would run in parallel.

4.2.1 Challenge 1: Problem Reformulation

Any real hardware/software environment will deviate from the parallel execution environment described above. From the threads' point of view, any deviation will result in one or more delays inserted in their fully-parallel delay-free execution. Figure 4.4 summarizes the execution of four threads, where each thread performs four read and–or write accesses. For the sake of simplicity, let us assume that just one delay is inserted in a given thread. This delay will shift the thread's computation forward in time, and possibly cause some memory write access(es) to happen in close proximity, leading to the possibility of exposing an interference condition.

Concretely, at 102 ms the first thread writes into a memory location a value and, given the schedule, no interference happens. In other words, two threads do not attempt a write at the same time in the same memory location. However, if, for any reason, the thread schedule

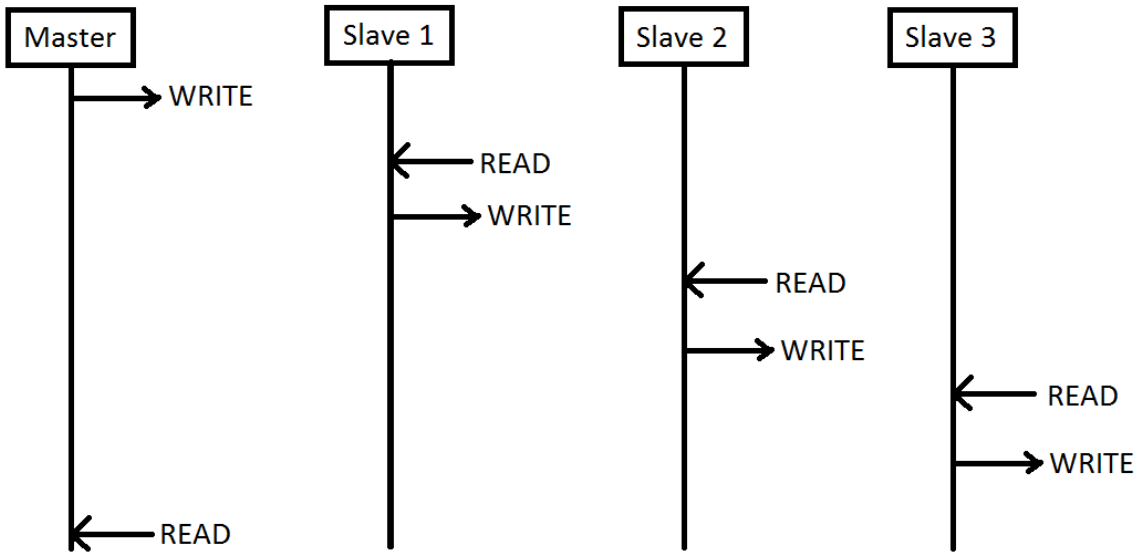

Figure 4.1 Behavior of a PUT.

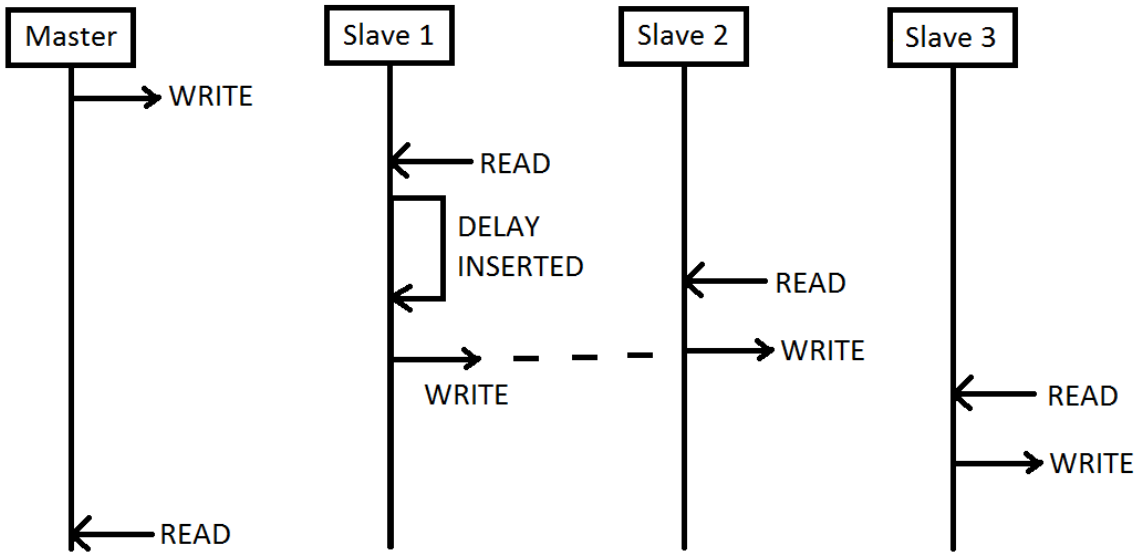

Figure 4.2 Behavior of the same PUT with an injected delay, exposing an interference bug.

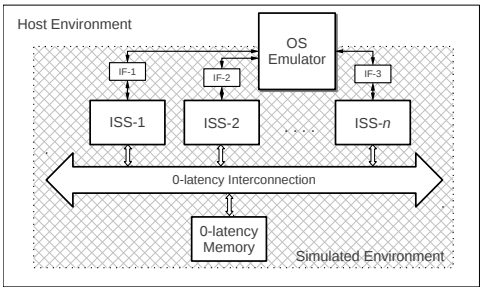

Figure 4.3 The virtual platform on which the PUT is mapped: each component except for the processors has zero latency, and OS calls are trapped and executed externally

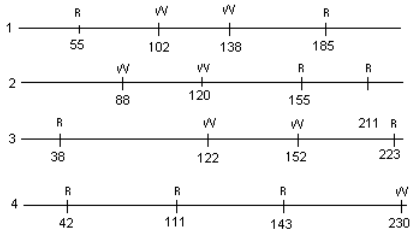

Figure 4.4 Concurrent Threads Example

is different and, for example, thread 2 is delayed by 2 ms after the first writing then a possible interference happens at 122 ms between threads 2 and 3.

The event schedule depends on the operating system, the computer workload, other concurrent systems being run, and the scheduler policy. Therefore, enhancing the possibility of exposing an interference bug via testing for any foreseeable schedule is a challenging problem that has been addressed in several ways, from search-based approaches (Briand *et al.* (2005, 2006)) to formal methods (Flanagan et Freund (2008); Tripakis *et al.* (2010)). The higher the number of available CPUs, the higher the number of scheduled threads and events, and the more difficult it is to manually verify that any two threads will not create an interference under any possible schedule. The example in figure 4.4 shows a larger system, with a higher number of threads and a longer execution time. Also in this case, the delays would be inserted in a similar manner between events (taking into account the longer execution time). In general, we believe that our approach would be able to increase the chances of exposing the interference conditions for systems of any size.

4.2.2 Challenge 2: Fitness Formulation

Let the PUT be composed on N threads. Let us assume that the i^{th} thread contains M_i events; let $t_{i,j}$ be, for the thread i, the time at which an event (a memory access, a function call, and so on) happens and let i^* be the thread subject to perturbation, the thread in which a delay Δ will be injected before an event p. Finally, let $a_{i,j}$ stands for the action performed at time $t_{i,j}$ by the thread i. Because ReSP allows to precisely track memory accesses as well as times, to simplify the formalization, let us further assume that $a_{i,j}$ equals to 1 to show that it is a "write" action to a given memory cell or 0 to show some other action. Then, our

objective is to maximize the number of possible interferences $N_{Interference}$:

$$N_{Interference} = \max_{\Delta,p,i*} \left\{ \sum_{i=1,i\neq i*}^{N} \sum_{j=1}^{M_i} \sum_{k=p}^{M_{i*}} \delta(a_{i*,k}, a_{i,j}) \delta(t_{i,j}, t_{i^*,k} + \Delta) \right\}$$

under the constraint $t_{i,j} \geq t_{i^*,k} + \Delta$, and where $\delta(x, y)$ is the Kronecker operator[1].

We want to maximize the numbers of alignments, two write events coinciding at the same time occurring in the same memory location, using Equation 4.1. Unfortunately, this equation leads to a staircase-like landscape as it result in a sum of 0 or 1. Any search strategy will have poor guidance with this fitness function and chances are that it will behave akin to a random search.

If we assume that a delay is inserted before each write event in all threads, then all threads events will be shifted. More precisely, if $\Delta_{i,j}$ is the delay inserted between the events $j-1$ and j of the thread i, all times after $t_{i,j}$ will be shifted. This shift leads to new time τ for the event $a_{i,j}$:

$$\tau_{i,j}(a_{i,j}) = t_{i,j} + \sum_{k=1}^{j} \Delta_{i,k} \tag{4.1}$$

Considering the difference between $\tau_{i_q,j_q}(a_{i_q,j_q})$ and $\tau_{i_r,j_r}(a_{i_r,j_r})$, when both a_{i_q,j_q} and a_{i_r,j_r} are write events to the same memory location, we rewrite Equation 4.1 as:

$$N_{Interference}(write) = \max_{\Delta_{1,1},\ldots,\Delta_{N,N_N}} \left\{ \sum_{i_r}^{N} \sum_{j_r}^{M_{j_r}} \sum_{i_q \neq i_r}^{N} \sum_{j_q}^{M_{j_q}} \frac{1}{1 + |\tau_{i_q,j_q}(a_{i_q,j_q}) - \tau_{i_r,j_r}(a_{i_r,j_r})|} \right\} \tag{4.2}$$

under the constraints $\tau_{i_q,j_q} \geq \tau_{i_r,j_r}$ and $a_{i_q,j_q} = a_{i_r,j_r} = write$ to the same memory location.

Equation 4.2 leads to a minimization problem:

$$N_{Interference}(write) = \min_{\Delta_{1,1},\ldots,\Delta_{N,N_N}} \left\{ \sum_{i_r}^{N} \sum_{j_r}^{M_{j_r}} \sum_{i_q \neq i_r}^{N} \sum_{j_q}^{M_{j_q}} (1 - \frac{1}{1 + |\tau_{i_q,j_q}(a_{i_q,j_q}) - \tau_{i_r,j_r}(a_{i_r,j_r})|}) \right\} \tag{4.3}$$

For both Equations 4.2 and 4.3, given a $\tau_{i_q,j_q}(a_{i_q,j_q})$, we may restrict the search to the

[1]The Kronecker operator is a function of two variables, usually integers, which is 1 if they are equal and 0 otherwise ($\delta(x, y) = 1$, if $x = y$; or 0 otherwise).

closest event in the other threads, typically: $\tau_{i_r,j_r}(a_{i_r,j_r}) \geq \tau_{i_q,j_q}(a_{i_q,j_q})$ & $\tau_{i_q,j_q}(a_{i_q,j_q}) \leq \tau_{i_q,j_s}(a_{i_q,j_s})$. Under this restriction, **Equation 4.3 is the fitness function used in the search algorithms to inject appropriate delays in threads to maximize the probability of exposing interference bugs (if any).** This equation also solves the staircase-like landscape problem because the fitness function is sum of real numbers, providing a smoother landscape.

4.2.3 Challenge 3: Scalability Analysis

We want to analyse the effectiveness of the search-based approaches when the input search space domain is varied from as small as 1 sec to as large as 1000 sec. The input to the program is the delay which will be injected before every thread event. Thus for analysing and comparing the scalability of the approaches, we vary the injected delays from small to large to see their effect on the performance of the approaches. Our parameter to compare the relative performances is the number fitness evaluations each approach requires for aligning a pair of write events.

4.2.4 ReSP: Parallel Execution Environment

Due to the non-deterministic behaviour of the multi-threaded environment controlling the thread execution becomes difficult. In our work, we control the environment by tuning the parameters we want to control using a platform ReSP. ReSP provides a deterministic parallel execution environment without external influences we use a virtual platform, namely ReSP (Beltrame *et al.* (2009)). ReSP is a virtual environment for modeling an ideal multi-processor system with as many processors as there are threads. ReSP is also a platform based on an event-driven simulator that can model any multi-processor architecture and that implements an emulation layer that allows the interception of any OS call. ReSP allows access to all execution details, including time of operations in milliseconds, accessed memory locations, thread identifiers, and so on.

To create our parallel execution environment, we model a system as a collection of Processing Elements (PEs), ARM cores in our specific case but any other processor architecture could be used, directly connected to a single shared memory, as shown in Figure 4.3. Therefore, our environment makes as many PEs available as there are execution threads in a PUT, each thread being mapped to a single PE. PEs have no cache memory, their interconnection is implemented by a 0-latency crossbar, and the memory responds instantaneously, thus each thread can run unimpeded by the sharing of hardware resources. This environment corresponds to an ideal situation in which the fastest possible execution is obtained.

We use ReSP to run our PUTs and calculate the threads' execution times for validating our results. We use ReSP's ability to trap any function call being executed on the PEs to route all OS-related activities outside the virtual environment. Thread-management calls and other OS functions (`sbrk`, file access, and so on) are handled by the host environment, without affecting the timing behaviour of the multi-threaded system. Thus, the PUT perceives that all OS functions are executed instantaneously without access to shared resources. Because all OS functions are trapped, there is no need for a real OS implementation to run the PUT in the virtual environment. This is to say the PUTs are run without any external interference. The component performing the routing of OS functions, referred to as the OS Emulator, takes care of processor initialization (registers, memory allocation, and so on) and implements a FIFO scheduler that assigns each thread to a free PE as soon as it is created.

The main difference between our virtual environment and a real computer system are cache effects, the limited number of cores, other applications running concurrently on the same hardware and other interactions that make scheduling non-deterministic and introduce extra times between events. In our environment, the full parallelism of the PUT is exposed and the only interactions left are inherent to the PUT itself.

4.2.5 Problem Modeling and Settings

As described above, the key concepts in our thread interference model are the times and types of thread events. To assess the feasibility of modeling thread interferences by mimicking an ideal execution environment, we are considering simple problem configurations. More complex cases will be considered in future works; for example, modeling different communication mechanisms, such as pipes, queues, or sockets. We do not explicitly model resource-locking mechanisms (semaphores, barriers, etc.) as they are used to protect data and would simply enforce a particular event ordering. Therefore, if data is properly protected, we would simply fail to align two particular events. At this stage, we are also not interested in exposing deadlock or starvation bugs.

From Equations 4.1 and 4.3, we can model the problem using the times and types of thread events. Once the occurrence write events has been timestamped using ReSP, we can model different schedules by shifting events forward in time. In practice, for a given initial schedule, all possible thread schedules are obtained by adding delays between thread events (using Equation 4.1).

Our fitness function 4.3 (what an expression of how fit is our schedule to help expose interference bugs) can be computed iterating over all thread write events. The fitness computation has therefore a quadratic cost over the total number of write events. However, for a given write event, only events occurring in times greater or equal to the current write are

of interest, thus making the fitness evaluation faster (though still quadratic in theory).

In general, it may be difficult or impossible to know the maximum delay that a given thread will experience. Once a thread is suspended, other threads of the same program (or other programs) will be executed. As our PUTs are executed in an ideal environment (no time sharing, preemption, priority, and so on), there is no need to model the scheduling policy and we can freely insert any delay at any location in the system to increase the chances of exposing the interference bug.

4.3 Empirical Study

The *goal* of our empirical study is to obtain the conceptual proof of the feasibility and the effectiveness of our search-based interference detection approach and validate our fitness function (see Equation 4.3).

As a metric for the quality of our model, we take the number of times we succeed in aligning two different write events to the same memory location. Such alignment increases the chances of exposing an interference bug, and can be used by developers to identify where data is not properly protected in the code.

To perform our conceptual proof, we have investigated the following two research questions:

RQ1: Can our approach be effectively and efficiently used on simple as well as real-world systems to maximize the probability of interferences between threads?

RQ2: How does the dimension of the search space impact the performance of three search algorithms: RND, SHC and SA?

The first research question aims at verifying that our fitness function guides the search appropriately, leading to convergence in an acceptable amount of time. The second research question concerns the choice of the search algorithm to maximize the probability of exposing interference bugs. The need for search algorithms is verified by comparing SHC and SA with a simple random search (RND): better performance from SHC and SA increases our confidence in the appropriateness of our fitness function.

One advantage of this approach is that it has no false positives, because it doesn't introduce any functional modification in the code: if a bug is exposed, the data are effectively unprotected. Nevertheless, we do not guarantee that a bug will be exposed even if present, as the approach simply increases the likelihood of showing interferences. The chances of exposure are increased because the manifestation of interference bug depends on the thread schedule and we, unlike other approaches, manufacture the schedules that maximize interference.

It might be argued that a data race detector does not need the timings of write alignments to be so accurate. But as we are dealing with a fully parallelized environment, we target specific instances and align the events with much more precision than what required by a data race detector (Artho *et al.* (2003)). Basically, we are pin-pointing the event times with accuracy.

Our approach can also be used in cases where it is enough to have two change the order of two events to verify the correctness of some code. Once the events are aligned, an arbitrarily small additional delay would change the order of two events, possibly exposing data protection issues. The correct use of locks or other data protection measures would prevent a change in the order of the events.

4.4 Results

To answer our two research questions, we implemented four small synthetic and three real-world multi-threaded systems with different numbers of threads and read/write events. Table 4.1 provides the details of these applications: their names, sizes in numbers of lines of code, number of threads, and sequences of read and write access into memory. The "Events" column shows the various events with a comma separating each thread. For example, the thread events in column 4 for row 3 (*Average of Numbers*) should be read as follows: Thread 1 has just one write event, thread 2 has a read, followed by a write event, and thread 3 has a read followed by a write event.

We believe that our results are independent of the execution system or architecture: our approach aligns write events among threads on a virtual system, and exposes data protection issues regardless of the final execution platform. Similarly, when applied to applications of any size, the interference conditions are exposed irrespective of the number of lines of code

Table 4.1 Application Details

PUTs	**LOCs**	**Nbr. of Threads**	**Events**
Matrix Multiplication (MM)	215	4	RWWR, WWWW, RRWW, WWRR
Count Shared Data (CSD)	160	4	WWW, RRW, RWRW, RW
Average of Numbers (AvN)	136	3	W, RW, RW
Area of Circle (AC)	237	5	RW, RWW, RRW, RWRW, RWWRW
CFFT	260	3	WR, WR, WRWRWR
CFFT6	535	3	WRWRWRWRWRWR, WR, WRWRWRWR
FFMPEG	290K	4	WRWRRR, WR, WR, WR

Table 4.2 Execution Times for Real-World Applications, in milliseconds

	CFFT		CFFT6		FFMPEG	
	16	**17**	**16**	**17**	**16**	**17**
SA	**3118**	**5224**	**27443**	**20416**	**1562**	**4672**
HC	**3578**	**4976**	**27328**	**21943**	**1378**	**5100**
RND	**113521**	**107586**	**342523**	**339951**	**59599**	**133345**

that the application may contain.

4.4.1 RQ1: Approach

RQ1 aims at verifying if our approach can effectively identify time events configurations, and thus schedules, leading to possible thread interferences. We experimented with RND, SHC, and SA. RND is used as a sanity check and to show that a search algorithm is effectively needed.

For the three search algorithms (RND, SHC and SA), we perform no more than 10^6 fitness evaluations. For the three algorithms, we draw the added delays $\Delta_{i,k}$ from a uniform-random distribution, between zero and a maximum admissible delay fixed to 10^7 ms (10^4 seconds).

We configured RND in such a way that we generated at most 10^7 random delays $\Delta_{i,k}$, uniformly distributed in the search space, and applied Equation 4.1 to compute the actual thread time events. We then evaluated each generated schedule, to check for interferences, using Equation 4.3.

We set the SHC restart value at 150 trials and we implemented a simple neighbor variable step visiting strategy. Our SHC uses RND to initialize its starting point, then it first attempts to move with a large step, two/three orders of magnitude smaller than the search space dimension, for 1000 times, finally it reduces the local search span by reducing the distance from its current solution to the next visited one. The step is drawn from a uniformly-distributed random variable. Thus, for a search space of 10^7, we first attempt to move with a maximum step of 10^4 for 20 times. If we fail to find a better solution, perhaps the optimum is close, and then we reduce the step to 10^3 for another 20 trials, and then to 500 and then to 50. Finally, if we do not improve in 150 move attempts (for the large search space), the search is discarded, a next starting solution is generated and the process restarted from the new initial solution. We selected the values and the heuristic encoded in the SHC via a process of trial and error. Intuitively, if the search space is large, we would like to sample distant regions but the step to reach a distant region is a function of the search space size, so we arbitrarily set the maximum step of two or three orders of magnitude smaller than the size of the search space.

We configured the SA algorithm similarly to the SHC algorithm, except for the cooling

factor and maximum temperature. In our experiments, we set $r = 0.95$ and T_{max} depending on the search space, 0.0001 for a search space of size 10^7 and 0.01 for smaller search spaces.

Regarding the times to compute the solutions with different algorithms, it is a well known fact that SHC and SA scale less than linearly with the design space. This fact can be proven by having a look at the results provided in table 4.2. Table 4.2 shows the execution times of various algorithms for computing the alignments 100 times only for the real-world applications with large search space (10^6 and 10^7 ms). It can be seen that despite the increasing size of the design space, SHC and SA converge to solutions in reasonable amounts of times, as compared to RND.

Figure 4.5 reports the relative performance over 100 experiments of RND, SHC, and SA for *Matrix Multiplication*, CFFT6 and FFMPEG for a search space of 10^7 ms. Each time that our approach has exposed a possible interference, we recorded the number of fitness evaluations and stopped the search. We obtained similar box-plots for the other applications, but do not show them here because they do not bring additional/contrasting insight in the behavior of our approach. The box-plots show that SHC outperforms RND and that SA and SHC perform similarly. However, SA performs marginally better than SHC The missing plots for CFFT6 in Figure 4.5 are due to the fact that RND did not find any solution in 10^6 iterations, even after 100 runs.

Overall, we can positively answer our first research question: **our approach can effectively and efficiently be used on simple program models to maximize the possibility of interferences between threads.**

4.4.2 RQ2: Search Strategies

RQ2 aims at investigating the performance of the different strategies for various search-space dimensions. Our intuition is that, if the search space is not large, then even a random algorithm could be used. We set the maximum expected delay to 1 sec (10^3 ms), 10 sec (10^4 ms) and 1000 sec (10^6 ms). As the search space is smaller than that in RQ1, we also reduced the number of attempts to improve a solution before a restart for both SHC and SA as a compromise between exploring a local area and moving to a different search region. We set this number to 50.

Figures 4.6, Figure 4.7 and Figure 4.8 report the box-plots comparing the relative performance of RND, SHC, and SA for the first three synthetic and two real-world applications. Figure 4.7 has been made more readable by removing a single outlier value of 28,697 for the number of fitness evaluations of *Count Shared Data* when using SHC.

As expected, when the search space is small, RND performs comparably to both SHC and SA, as shown in Figure 4.7. However, when the search space size increases, SHC and SA

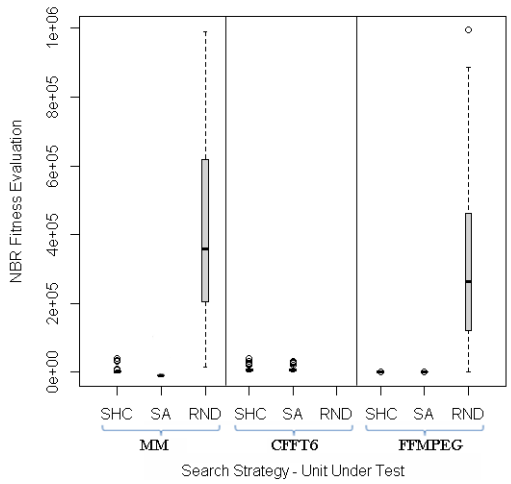

Figure 4.5 (RQ1) Algorithm comparison for a search space up to 10 Million sec delay

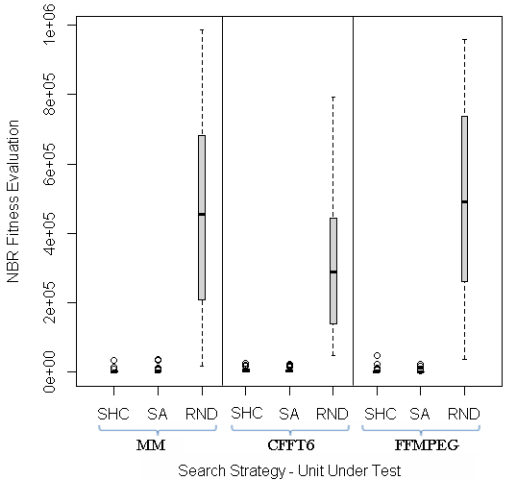

Figure 4.6 (RQ2) Algorithm comparison for a search space up to 1 Million sec delay

perform better than RND, as shown in Figure 4.8 and 4.6.

One might argue that in Figure 4.7 *Average of Numbers* shows instances where some outliers for which SHC reaches almost the maximum number of iterations to align the events, which does not seem to be the case with RND. The explanation is that in small search spaces RND can perform as well as any other optimization algorithm, sometimes even better. It is worth noting that even in a search space of 10^3 ms, there were instances where RND could not find a solution even within the maximum number of iterations (10^6 random solutions). SHC and SA were successful in exposing a possible bug each and every time (in some cases with higher number of iterations, which resulted in the outliers).

We also observe differences between the box-plots for the *Count Shared Data* and *Average of Numbers*. We explain this observation by the fact that *Count Shared Data* contains more write actions (see Table 4.1) than *Average of Numbers*. In other words, it is relatively easier to align two events in *Count Shared Data* than in *Average of Numbers*. Indeed, there are only three possible ways to create an interference in *Average of Numbers* while *Count Shared Data* has 17 different possibilities, a six-fold increase which is reflected into the results of Figure 4.7.

Once the search space size increases as in Figure 4.8, RND is outperformed by SHC and SA. In general, SA tends to perform better across all PUTs.

Overall, we can answer our second research question as follows: **the dimension of the search space impacts the performance of the three search algorithms. SA performs the best for the all PUTs and different delays.**

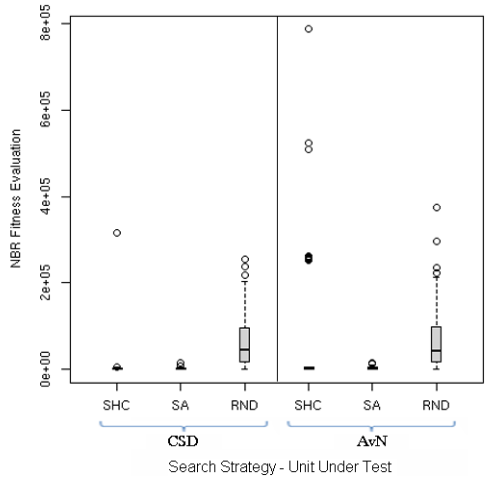

Figure 4.7 (RQ2) Algorithm comparison for a search space up to 1 sec delay

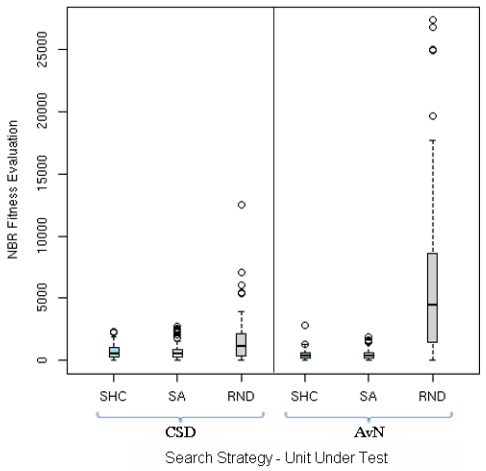

Figure 4.8 (RQ2) Algorithm comparison for a search space up to 10 sec delay

4.4.3 Threats to Validity

Our results support the conceptual proof of the feasibility and the effectiveness of our search-based interference detection approach. They also show that our fitness function (Equation 4.3) is appropriate, as well as the usefulness of a virtual environment to enhance the probability of exposing interference bugs.

Exposed interferences are somehow artificial in nature as they are computed with respect to an ideal parallel execution environment. In fact, the identified schedules may not be feasible at all. This is not an issue, as we are trying to discover unprotected data accesses, and even if a bug is found with an unrealistic schedule, nothing prevents from being triggered by a different, feasible schedule. Making sure that shared data is properly protected makes code safer and more robust.

Although encouraging, our results are limited in their validity as our sample size includes only four small artificial and three real-world systems. This is a threat to *construct validity* concerning the relationship between theory and observations. To overcome this threat, we plan to apply our approach on more number of real-world systems in future work.

A threat to *internal validity* concerns the fact that, among the four artificial systems used, we developed three of them. However, they were developed long before we started this work by one of the authors to test the ReSP environment. Thus, they cannot be biased towards exposing interference bugs.

A threat to *external validity* involves the generalization of our results. The number of

evaluated systems is small (a total of seven systems). Some of them are artificial, meant to be used for a proof of concept. Future work includes applying our approach to other large, real-world systems.

4.5 Conclusion

In this chapter, we proposed an approach to expose interference bugs. We analyse the problem of generating test data to maximize the possibility of exposing interference bugs as three challenges: C1: Formulating the original problem of exposing interference bug pattern as a search problem, C2: Developing the right fitness function for the problem formulation, and C3: Finding the right (scalable) search-based approach using scalability analysis. We modified the existing problem into a heuristic-based delay-injection problem to align write events in thread pairs. We then proposed a novel fitness function to maximize the number of write events aligned and thus maximize the possibility of exposing interference bug pattern. We varied the input search space domain from very small to huge for verifying the scalability of the search-based approaches. We used the ReSP platform to simulate the hardware on which a multi-threaded system runs and thus remove the inherent non-deterministic behaviour of the multi-threaded system environment. ReSP provided an ideal, fully-parallel virtual hardware environment, without the intervention of the operating system (in particular of the scheduler) to allow parallel execution of all threads. Using ReSP, we obtained thread execution times and then we mimicked the scheduler effect by injecting delays in the thread executions, as any scheduler will just reduce the level of parallelism with respect to a fully-parallel implementation. We identified the locations and durations of the delays using our novel fitness function and applying three different search algorithms. We applied algorithms, namely random search, stochastic hill-climbing algorithm, and simulated annealing algorithm, guided by the fitness function, to search the space of possible delays to insert in the executions of threads to expose interference bugs, two or more threads writing at the same time in a same memory location.

We report results of our fitness function and the three search algorithms on four small artificial and three real-world small/large multi-threaded systems. Our results suggest that, for small search spaces, even a simple random strategy may suffice, while in large search spaces (with delays in minutes) and thus millions of instructions interleaving two thread's events, a stochastic hill-climbing algorithm performs substantially better than random. We thus showed that our approach can expose interference bugs in these multi-threaded systems.

CHAPTER 5

CONCLUSION AND FUTURE WORK

Developing and testing concurrent code is much more difficult than a piece of sequential code. The prime reason is the non-deterministic behaviour of the environment in which the multi-threaded program runs. One cannot control the environment in which the program is running. As the environment cannot be modified or controlled, the tester must resort to indirect means to increase the number of schedules tested. The environment consists of schedulers, cache, interrupts, and other programs running. A small change in any of the parameters, like temperature of the processor could influence the timings and thus the scheduler. The scheduler, which is a part of the environment on which the program runs, determines the way available threads run each program. Another source of non-determinism in multi-threaded systems is the shared memory access. A shared memory is memory that may be simultaneously accessed by multiple threads with an intent to provide communication among them. In shared memory multi-threaded programs, the deterministic behavior is not inherent. When executed, such systems can experience one of many possible interleavings of memory accesses to shared data, due to the scheduler being not synchronous, which can result in an indeterministic behaviour leading to data race or interference conditions being raised. An interference condition in multi-threaded system occurs when (1) two or more concurrent threads access a shared variable, (2) at least one access is a write, and (3) the threads use no explicit mechanism to prevent the access from being simultaneous. Test data generation using search-based techniques have provided solutions to the problem of testing single-threaded systems over the years (Miller et Spooner (1976); McMinn *et al.* (2009); Eytani et Ur (2004)). But they have not been applied to multi-threaded systems for exposing bugs. Based on the previous work we formulate our thesis as:

Search-based approaches can be used effectively to generate test data to expose interference conditions in multi-threaded Systems.

We propose a novel formulation of exposing interference bugs in multi-threaded systems, using search-based techniques. Before exposing bugs in multi-threaded systems using search-based techniques, we intended to find the major challenges of using search-based approaches in single-threaded systems. From the related work, we identified three major challenges of using search-based approaches:

- C1: Formulating the original problem as a search problem.

- C2: Developing the right fitness function for the problem formulation.
- C3: Finding the right (scalable) search-based approach using scalability analysis.

We carry out two studies and address the three challenges on each on them to verify their effectiveness. As the first preliminary study, we analysed the three challenges in single-threaded systems to study the impact of using search-based approaches for generating test data to solve the problem of raising divide-by-zero exception conditions because we felt that our findings from this study might be applicable to the multi-threaded systems. We found that the three challenges C1, C2 and C3 are indeed important and can be addressed. Based on the knowledge about the importance of C1, C2, and C3 from the first part of our thesis, we moved on to propose a solution for a more complex problem of generating test data to expose interference bug pattern in multi-threaded systems using search-based approaches while addressing the three challenges. To verify if we can retain our claim that search-based approaches are as effective in exposing interference bugs for multi-threaded systems as they are for single-threaded systems, we proposed an approach to expose interference bugs using search-based approaches. We removed the inherent non-determinism behaviour of the multi-threaded environment using ReSP platform to simulate the hardware on which a multi-threaded programs run and tuning the necessary parameters to maintain a deterministic behaviour of the hardware. ReSP created an ideal, fully-parallel virtual hardware environment, without the intervention of the operating system (in particular of the scheduler) to allow parallel execution of all threads. We analysed the problem from the point of view of the three major challenges C1, C2 and C3. C1: We modified the problem formulation into a search problem by converting it into an equivalent optimization problem of maximizing the chances of exposing interference bug pattern in multi-threaded systems by the injection of delays in the execution flow of the threads. To the best of our knowledge, none of the previous work has solved the problem from a search-based perspective or approached the problem by manufacturing specific thread schedules and injecting delays. C2: We formulated a novel fitness function based on the problem formulation. C3: We analysed the scalability of various search-based techniques like Hill Climbing, Simulated Annealing, and Random Search to find the best approach by varying the input search space domain from very small to huge. We validated our approach on four small artificial and three real-world small/large multi-threaded programs. Our results suggested that, for small search spaces, even a simple random strategy may suffice, while in large search spaces (with delays in minutes) and thus millions of instructions interleaving two thread's events, a stochastic hill-climbing algorithm performs substantially better than random. Thus, also in the second study we found that search-based approaches perform much better than random. We thus confirm our thesis.

Based on the studies that we carried out involving single and multi-threaded systems, we can conclude that search-based approaches are as effective and efficient in exposing divide-by-zero exception in single-threaded systems as they are in exposing interference bugs in multi-threaded systems. However, our approach, both for single and multi-threaded systems, has been validated and verified on few systems. Thus, at this point of time, the approach cannot be concretely generalized beyond a certain extent. We definitely must apply our approaches to a greater number of software systems with varying range of complexities in terms of their functionalities, number of lines of code, number of threads, etc.

For the single-threaded systems, our approach is limited to generating data that could expose a divide-by-zero exception. In the future, we would like to extend this approach to other possible exception conditions like floating point and null pointer exceptions. Moreover, we would also like to generate test data to expose the exceptions when the piece of code contains complex data structures like arrays, hash tables etc. We would also like to integrate a chaining approach to better deal with data dependencies and study the testability transformations required to simplify and make it efficient to generate test input data to raise exceptions.

When validating our approach for multi-threaded systems, we would like to validate our approach when the multi-threaded systems would run on a real hardware platform, where the environment would be non-deterministic. In such a situation, the fitness function might be needed to be modified to take care of the non-determinism. We would like to extend our work by enriching our interference model to large and complex systems with more complex data structures, such as pipes, shared memories, or sockets. The extension would involve modeling other communication mechanisms than the sharing of a memory, such as pipes, queues, or sockets. We would also model resource-locking mechanisms, such as semaphores. Another scope of improvement in to model a similar mathematical models that would expose other bug patterns like deadlock using mutation operators, as proposed by (Bradbury *et al.* (2006)). Such a model might or might not follow the same philosophy of injecting delays and playing with them, so as to enhance the possibility of exposing the specific bug patterns.

REFERENCES

ARTHO, C., HAVELUND, K. et BIERE, A. (2003). High-level data races. *JOURNAL ON SOFTWARE TESTING, VERIFICATION RELIABILITY (STVR)*. 1–20.

BARESEL, A. (2000). *Automatisierung von strukturtests mit evolutionren algorithmen*. Diploma Thesis, Humboldt University, Berlin, Germany.

BELTRAME, G., FOSSATI, L. et SCIUTO, D. (2009). ReSP: a nonintrusive transaction-level reflective MPSoC simulation platform for design space exploration. *Computer-Aided Design of Integrated Circuits and Systems*, 28–40.

BEN-ASHER, Y., FARCHI, E. et EYTANI, Y. (2003). Heuristics for finding concurrent bugs. *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*. 288–297.

BRADBURY, J. S., CORDY, J. R. et DINGEL, J. (2006). Mutation operators for concurrent java (j2se 5.0). *Proceedings of the Second Workshop on Mutation Analysis*. IEEE Computer Society, Washington, DC, USA, MUTATION '06, 11–20.

BRIAND, L. C., LABICHE, Y. et SHOUSHA, M. (2005). Stress Testing Real-Time Systems with Genetic Algorithms. *Proceedings of the 2005 Conference on Genetic and Evolutionary Computation (GECCO '05)*. 1021–1028.

BRIAND, L. C., LABICHE, Y. et SHOUSHA, M. (2006). Using Genetic Algorithms for Early Schedulability Analysis and Stress Testing in Real-Time Systems. *Genetic Programming and Evolvable Machines*, 7, 145–170.

BUTENHOF, D. R. (2010). Threading programming guide. *Apple Inc.*

CARVER, R. H. et TAI, K.-C. (1991). Replay and testing for concurrent programs. *IEEE Softw.*, 8, 66–74.

CHATTERJEE, R. et RYDER, B. G. (1999). Data-flow-based testing of object-oriented libraries, 1–21.

COHEN, J. (1988). *Statistical power analysis for the behavioral sciences (2nd ed.)*. Lawrence Earlbaum Associates, Hillsdale, NJ.

DOVAL, D., MANCORIDIS, S. et MITCHELL, B. S. (1998). Automatic clustering of software systems using a genetic algorithm. *In Proceedings of Software Technology and Engineering Practice*. 73–91.

EDELSTEIN, O., FARCHI, E., GOLDIN, E., NIR, Y., RATSABY, G. et UR, S. (2003). Framework for testing multi-threaded java programs. *Concurrency and Computation: Practice and Experience*, 15, 485–499.

EYTANI, Y. et UR, S. (2004). Compiling a benchmark of documented multi-threaded bugs. *Parallel and Distributed Processing Symposium, International*, 17, 266–273.

FANG, H., KILANI, Y., LEE, J. H. M. et STUCKEY, P. J. (2002). Reducing search space in local search for constraint satisfaction. *Eighteenth national conference on Artificial intelligence*. American Association for Artificial Intelligence, Menlo Park, CA, USA, 28–33.

FARCHI, E., NIR, Y. et UR, S. (2003). Concurrent bug patterns and how to test them. *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*. 286–292.

FERGUSON, R. et KOREL, B. (1996). The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, 5, 63–86.

FLANAGAN, C. et FREUND, S. N. (2008). Atomizer: A dynamic atomicity checker for multithreaded programs. *Scientific Computer Program*, 71, 89–109.

GODEFROID, P. (1997). Model checking for programming languages using verisoft. *In Proceedings of the 24th ACM Symposium on Principles of Programming Languages*. 174–186.

HARMAN, M., HU, L., HIERONS, R., WEGENER, J., STHAMER, H., BARESEL, A. et ROPER, M. (2004). Testability transformation. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 30, 3–16.

HOVEMEYER, D. et PUGH, W. (2004). Finding bugs is easy. *ACM SIGPLAN Notices*. 132–136.

HWANG, G.-H., CHUNG TAI, K. et LU HUANG, T. (1995). Reachability testing: An approach to testing concurrent software. *International Journal of Software Engineering and Knowledge Engineering*, 5, 493–510.

JO, J.-W., CHANG, B.-M., YI, K. et CHOE, K.-M. (2004). An uncaught exception analysis for java. *J. Syst. Softw.*, 72, 59–69.

JONES, B., STHAMER, H. et EYRES, D. (1996). Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11, 299–306.

JONES, B. F., EYRES, D. E. et H. STHAMER, H. (1998). A strategy for using genetic algorithms to automate branch and fault-based testing. *The Computer Journal*, 41, 98–107.

JOSHI, P., NAIK, M., PARK, C.-S. et SEN, K. (2009). Calfuzzer: An extensible active testing framework for concurrent programs. *Proceedings of the 21st International Conference on Computer Aided Verification*. Springer-Verlag, Berlin, Heidelberg, CAV '09, 675–681.

KING, J. C. (1976). Symbolic execution and program testing. *Commun. ACM*, 19, 385–394.

KOREL, B. (1992). Dynamic method of software test data generation. *Softw. Test, Verif. Reliab*, 2, 203–213.

KOREL, B. (2003). Dynamic method for software test data generation. *Journal on Software Testing, Verification and Reliability*. vol. 2, 203–213.

LANZELOTTE, R. S. G., VALDURIEZ, P. et ZAïT, M. (1993). On the effectiveness of optimization search strategies for parallel execution spaces. *Proceedings of the 19th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, VLDB '93, 493–504.

LEI, Y. et CARVER, R. H. (2006). Reachability testing of concurrent programs. *IEEE Trans. Softw. Eng.*, 32, 382–403.

LEVESON, N. G. (1995). *Safeware: System Safety and Computers*.

LIM, A., LIN, J., RODRIGUES, B. et XIAO, F. (2006). Ant colony optimization with hill climbing for the bandwidth minimization problem. *Appl. Soft Comput.*, 6, 180–188.

LIONS (1996). Ariane 5: Flight 501 failure report.

MANCORIDIS, S., MITCHELL, B. S. et RORRES, C. (1998). Using automatic clustering to produce high-level system organizations of source code. *In Proc. 6th Intl. Workshop on Program Comprehension*. 45–53.

MATTFELD, D. C. (1999). Scalable search spaces for scheduling problems. *in Proceedings of GECCO99, ed.s W. Banzhaf et al*. Morgan Kaufmann, 1616–1621.

MCCARTHY, M. (1997). What is multi-threading? *Linux Journal*.

MCMINN, P. (2004). Search-based software test data generation: a survey. *Software Testing Verification and Reliability*, 14, 105–156.

MCMINN, P. (2011). Search-based software testing: Past, present and future. *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE Computer Society, Washington, DC, USA, ICSTW '11, 153–163.

MCMINN, P., BINKLEY, D. et HARMAN, M. (2009). Empirical evaluation of a nesting testability transformation for evolutionary testing. *ACM Trans. Softw. Eng. Methodol.*, 18, 1–26.

MILLER, W. et SPOONER, D. L. (1976). Automatic generation of floating-point test data. *IEEE Trans. Softw. Eng.*, 2, 223–226.

MITCHELL, D. G. et TERNOVSKA, E. (2005). A framework for representing and solving np search problems. *Proceedings of the 20th national conference on Artificial intelligence - Volume 1*. AAAI Press, AAAI'05, 430–435.

MUELLER, F. et WEGENER, J. (1998). A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems*. IEEE, 144–154.

MUSUVATHI, M., QADEER, S. et BALL, T. (2007). Chess: A systematic testing tool for concurrent software.

PARK, A. (1999). Multithreaded programming (pthreads tutorial). `http://randu.org/tutorials/threads/`.

PARK, S., LU, S. et ZHOU, Y. (2009). Ctrigger: exposing atomicity violation bugs from their hiding places. *SIGPLAN Not.*, 44, 25–36.

PASAREANU, C. S. et VISSER, W. (2009). A survey of new trends in symbolic execution for software testing and analysis. *Int. J. Softw. Tools Technol. Transf.*, 11, 339–353.

R. SARAVANAN, P. A. et VIJAYAKUMAR, K. (2003). Machining parameters optimisation for turning cylindrical stock into a continuous finished profile using genetic algorithm (ga) and simulated annealing (sa). *Proceedings of the International Journal of Advanced Manufacturing Technology*. vol. 21, 1–9.

RYDER, B. G., SMITH, D., KREMER, U., GORDON, M. et SHAH, N. (2000). A static study of java exceptions using jesp. *Proceedings of the 9th International Conference on Compiler Construction*. Springer-Verlag, London, UK, UK, CC '00, 67–81.

SAXENA, P., POOSANKAM, P., MCCAMANT, S. et SONG, D. (2009). Loop-extended symbolic execution on binary programs. *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, New York, NY, USA, ISSTA '09, 225–236.

SINHA, S., ORSO, A. et HARROLD, M. J. (2004). Automated support for development, maintenance, and testing in the presence of implicit control flow. *Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society, Washington, DC, USA, ICSE '04, 336–345.

SIVAGURUNATHAN, G. et PURUSOTHAMAN, T. (2011). Reduction of key search space of vigenere cipher using particle swarm optimization. *Proceedings of the Journal of Computer Science*. vol. 7, 1633–1638.

SOFTWARE QUALITY RESEARCH GROUP, ONTARIO INSTITUTE OF TECHNOLOGY (2010). Concurrency anti-pattern catalog for java. `http://faculty.uoit.ca/bradbury/concurr-catalog/`.

SPILLNER, A. (1995). Test criteria and coverage measures for software integration testing. *Software Quality Journal*, 4, 275–286.

STHAMER, H. H. (1996). The automatic generation of test data using genetic algorithms.

TRACEY, N., CLARK, J. et MCDERMID, J. (2000). Automated test-data generation for exception conditions. *Software - Practice and Experience*, 30, 61–79.

TRACEY, N., CLARK, J. A. et MANDER, K. (1998). Automated program flaw finding using simulated annealing. *ISSTA*. 73–81.

TRIPAKIS, S., STERGIOU, C. et LUBLINERMAN, R. (2010). Checking non-interference in spmd programs. *2nd USENIX Workshop on Hot Topics in Parallelism (HotPar 2010)*. 1–6.

VAGOUN, T. (1996). Input domain partitioning in software testing. *Proceedings of the 29th Hawaii International Conference on System Sciences Volume 2: Decision Support and Knowledge-Based Systems*. IEEE Computer Society, Washington, DC, USA, HICSS '96.

VAGOUN, T. et HEVNER, A. R. (1997). Feasible input domain partitioning in software testing: Rcs case study. *Ann. Software Eng.*, 4, 159–170.

WEGENER, J., BARESEL, A. et STHAMER, H. (2001). Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43, 841–854.

WOEGINGER, G. J. (2003). Combinatorial optimization - eureka, you shrink! Springer-Verlag New York, Inc., New York, NY, USA, chapitre Exact algorithms for NP-hard problems: a survey. 185–207.

WRIGHT, M. (2010). Automating parameter choice for simulated annealing. *Lancaster University Management School*, 32.

XANTHAKIS, S., ELLIS, C., SKOURLAS, C., GALL, A. L., KATSIKAS, S. et KARAPOULIOS, K. (1992). Application des algorithmes genetiques au test des logiciels. *5th Int. Conference on Software Engineering and its Applications*. 625–636.

ZHANG, B. T. et KIM, J. J. (2000). Comparison of selection methods for evolutionary optimization. *Evolutionary Optimization*, 2, 55–70.