USING ESTIMATION OF DISTRIBUTION ALGORITHMS TO DETECT CONCURRENT FAULTS

jan staunton

Submitted for the degree of Doctor of Philosophy

Department of Computer Science
University of York

May 2012

Jan Staunton: *Using Estimation of Distribution Algorithms to Detect Concurrent Faults*, PhD Thesis, May 2012

ABSTRACT

With processors tending toward more processing cores instead of higher clock speeds, developers are increasingly forced to grapple with concurrent paradigms to maximally exploit new CPU designs. Embracing concurrent paradigms entails the potential risk of encountering concurrent software faults. Concurrent faults result from unforeseen timings of interactions between concurrent components, as opposed to traditional software faults that arise from functional failures. As a result, concurrent faults have a higher probability of surviving the software development process, potentially causing a catastrophic failure of high cost.

As the complexity of software and hardware systems increases they become increasingly difficult to test. One measure of complexity is the number of potential execution paths a system can follow, with a higher complexity attributed to a greater number of paths. In concurrent software, the number of execution paths in a system typically increases exponentially as the number of concurrent components increase. Testing complex concurrent software is therefore difficult, with state of the art static and dynamic analysis techniques yielding only false positives or exhausted resources. This problem is likely only to be exacerbated given the trends highlighted above.

Stochastic metaheuristic search techniques can often triumph where deterministic or analytical techniques fail. Methods such as Genetic Algorithms and Ant Colony Optimisation have shown great strength on hard problems, including testing concurrent software. Metaheuristic techniques often trade a perfect solution for good enough solutions, and merely accurately detecting a concurrent fault is better than allowing a fault to survive to a production system. Whilst metaheuristic techniques have had some success in this domain, the state of the art still struggles for a high success rate in some circumstances. There are a few metaheuristic search techniques that have yet to be tried in this area, and this thesis presents a study on one such technique.

This thesis presents a literature review, detailing the state of the art in detecting concurrent faults in software and hardware systems. Following a review of metaheuristic techniques applied to finding concurrent faults, I set out a hypothesis asserting that a particular subclass of metaheuristic techniques, Estimation of Distribution Algorithms, are effective in detecting and debugging concurrent faults. To investigate the hypothesis, I first make an algorithmic proposal based on a particular EDA to search the state space of concurrent systems. I then demonstrate through

experimentation the ability of the algorithm to detect faults and to optimise the quality of faults found in systems derived from industrial scenarios. I also outline methods of using features unique to EDAs to scale to large systems. Finally, I complete the thesis with a conclusion examining the hypothesis with respect to the evidence collected from empirical work, highlighting the novel aspects of the thesis and outlining future paths of research.

CONTENTS

I INTRODUCTION 1
1 MOTIVATION 3
1.1 Multi-processor Systems 3
1.2 Utilising Multiple Cores 3
1.3 Concurrent Software Verification 4
1.4 Metaheuristic Search Techniques 5
1.5 Motivation for research 5
1.6 Research goals 5
1.7 Structure of this report 6
2 HYPOTHESES 7
2.1 Introduction 7
2.2 Hypothesis and Research Strands 7
2.3 Finding Faults in Mainstream Language Code 7
2.4 Finding and Optimising Wide Ranges of Faults in Complex Systems 8
2.5 Scaling to Large Systems 9
2.6 Summary 9

II LITERATURE REVIEW 11
3 CONCURRENT SOFTWARE VERIFICATION 13
3.1 Introduction to Concurrent Software 13
3.1.1 Scheduling 14
3.1.2 Communication 15
3.2 Prominent faults in concurrent software 17
3.2.1 Deadlock 17
3.2.2 Starvation 22
3.2.3 Livelock 22
3.2.4 Data races 23
3.3 Concurrent software verification 24
3.3.1 Coverage metrics 25
3.4 Static verification techniques 26
3.4.1 Lockset analysis 26
3.4.2 Data-flow analysis 26
3.4.3 Summary of static verification techniques 27
3.5 Dynamic verification techniques 27
3.5.1 Schedule altering techniques 27
3.5.2 Runtime monitoring 28
3.5.3 Summary of dynamic verification techniques 28
3.6 Formal methods 29
3.6.1 Specification/Modelling 29
3.6.2 Annotation-based theorem proving 30
3.6.3 Model checking 30

3.6.4 Summary of formal methods 31
3.7 Summary 31
4 MODEL CHECKING 33
4.1 Introduction 33
4.2 Building the Model 34
4.2.1 Transition Systems 35
4.2.2 Paths 37
4.2.3 Reachable States 37
4.2.4 Expressing Models 38
4.3 Specification of Properties 38
4.3.1 Temporal Logics 39
4.4 Verification 40
4.4.1 Checking Invariant Properties 40
4.4.2 Checking Safety Properties 41
4.4.3 Checking Liveness Properties 41
4.4.4 Complete Model Checking Mechanisms 42
4.4.5 On-the-fly Model Checking 42
4.4.6 Guided Model Checking 42
4.4.7 Strengths and Limitations of Model Checking 43
4.4.8 Metaheuristic Model Checking 44
4.5 Summary 44
5 METAHEURISTIC SEARCH 45
5.1 Introduction 45
5.1.1 Metaheuristic Search 46
5.2 Local Search 46
5.2.1 Convergence and Optima 47
5.2.2 Landscapes 48
5.2.3 Fitness Function 49
5.2.4 Simulated Annealing 49
5.3 Other Local Search Mechanisms 50
5.4 Population-based Search 50
5.4.1 Evolutionary Algorithms 51
5.4.2 Ant colony optimisation 53
5.5 Estimation of Distribution Algorithms 54
5.5.1 Example EDA 54
5.5.2 Types of EDA 56
5.6 Summary 57
6 METAHEURISTIC SEARCH OF TRANSITION SYSTEMS 59
6.1 Genetic Algorithms 60
6.1.1 Solution Encoding 60
6.1.2 Fitness Functions 61
6.2 Particle Swarm Optimisation 62
6.3 Ant Colony Optimisation 62
6.3.1 Liveness Properties 63
6.4 Evaluation of Metaheuristic Model Checking Work So Far 64

6.5 Summary 65
6.5.1 Limitations of State Of The Art 65
6.5.2 Potential Routes Forward 66
6.5.3 Summary 66
III ALGORITHMIC PROPOSAL 67
7 ALGORITHMIC PROPOSAL 69
7.1 Model and Solution Space 69
7.2 Learning the Model 72
7.3 Model Sampling 73
7.4 Fitness Function 74
7.5 Other Parameters and Features 76
7.6 Novelty of Algorithm 77
7.7 Implementation 78
7.7.1 N-gram Implementation 78
7.7.2 Interaction with a Model Checker 78
7.8 Computational Complexity 79
7.9 Summary 79
IV EXPERIMENTATION 81
8 FINDING DEADLOCK IN MAINSTREAM LANGUAGE CODE 83
8.1 Introduction 83
8.2 Experimentation 83
8.2.1 Fitness Function 84
8.2.2 Parameters 85
8.2.3 Results and Discussion 85
8.3 Summary 89
9 FINDING AND OPTIMISING COUNTEREXAMPLES IN INDUSTRIAL CODE 91
9.1 Introduction 91
9.2 Experimentation 91
9.2.1 Implementation 91
9.2.2 Connected Component Classification 92
9.2.3 Fitness function 93
9.3 Experiments in Finding and Optimising Counterexamples 94
9.3.1 Example Promela Models 94
9.3.2 Parameters of the EDA 95
9.3.3 Experiments 96
9.3.4 Discussion of Results 113
9.4 Summary 115
10 SCALING TO LARGE SYSTEMS 117
10.1 Introduction 117
10.2 Model Reuse 117
10.2.1 Reuse during Debugging 118
10.2.2 Reuse during Refinement 118
10.2.3 Reuse when tackling Problem Families 118

10.3 Experimentation with Problem Families 119
10.3.1 Sample Models 119
10.3.2 Heuristics 120
10.3.3 Parameters 121
10.3.4 Smaller Instances 122
10.3.5 Larger Instances 122
10.4 Summary 126

V CONCLUSION 127

11 CONCLUSION 129
11.1 Introduction 129
11.2 Hypotheses 129
11.2.1 Finding Faults in Mainstream Language Code 129
11.2.2 Finding and Optimising Wide Ranges of Faults in Complex Systems 130
11.2.3 Scaling to Large Systems 131
11.2.4 Over-arching hypothesis 131
11.3 Novel Contributions 132
11.4 Limitations of the Research 133
11.5 Potential Algorithm Refinements 134
11.5.1 Improve the context in which a decision is made 135
11.5.2 Augmenting the fitness function 136
11.6 Potential Avenues of Future Research 137
11.7 Summary 137

BIBLIOGRAPHY 139

LIST OF FIGURES

Figure 1 Round robin schedule for processes A, B and C 14
Figure 2 Mutual exclusion implementation using monitors (Java style) 16
Figure 3 Resource allocation graph for resources 1 through 5 and processes 1 through 5, loosely based on [Silberschatz et al., 2004] 18
Figure 4 Wait-for graph for processes 1 through 5, loosely based on [Silberschatz et al., 2004] 18
Figure 5 Wait-for cycle depiction 20
Figure 6 Dining Philosophers problem 21
Figure 7 Naive dining philosopher behaviour 21
Figure 8 Data race example code 23
Figure 9 Nasty interleaving 23
Figure 10 Traffic light transition system, based on the traffic light example found in [Baier and Katoen, 2008] 36
Figure 11 Concurrent Dining Philosophers system with two Philosophers 36
Figure 12 Example landscapes 49
Figure 13 Algorithm of a vanilla EA 52
Figure 14 Algorithm of a vanilla EDA 55
Figure 15 UMDA modelling Process 55
Figure 16 Numbered choices in a transition graph, from [Alba et al., 2008] 61
Figure 17 Real-Numbered Vector encoding of a path in a graph, from [Alba et al., 2008] 61
Figure 18 Fitness function for deadlock from [Alba et al., 2008] 62
Figure 19 A typical trace from JPF 71
Figure 20 Illustration of the N-gram learning process 72
Figure 21 Scaling results for dining philosophers 88

LIST OF TABLES

Table 1 Results table from experiments 86
Table 2 Table showing models used in experiments 95
Table 3 Results table from finding and optimising empirical work. 98
Table 4 Results table from finding and optimising empirical work. 106
Table 5 Problem families tested in our experiments 119
Table 6 Measurements from the initial runs 123
Table 7 Measurements from the model reuse runs on the Dining Philosophers 128 system 124
Table 8 Measurements from the model reuse runs on the Leader 10 system 124
Table 9 Measurements from the model reuse runs on the GIOP 20 system 125

ACKNOWLEDGMENTS

I would first like to thank my supervisor John A. Clark for his patience and support throughout my studies, and for letting me run amok with wild ideas for three years. Many thanks to Simon Poulding for his friendship, advice and encouragement. It was a pleasure collaborating with him on our award winning work. Thanks to all members of the SEBASE project at York throughout my spell as a Research Student, with specific thanks to David White for his advice and support. I will forever remember my time as part of SEBASE with fondness and I hope the team can forgive me for my regular 1430 sleepies.

Thank you to my wonderful house and flat mates over my time at York, particularly James Williams, Chris Poskitt, Peter Booth, Dog, Csaba, and Ed Milner. Particular thanks go to Jimmy who helped with proof reading, although thanks will be withdrawn if he traumatises me with his best man speech. (Edit: Post wedding minor corrections, he did very well!)

I would like to thank the EPSRC for their financial support, on grant EP/D050618/1, without which none of this would have been possible. Thank you to the wider SEBASE team for encouraging such an interesting field and allowing me to join their flock. Thanks also to the creators of the various tools I have used during my thesis, the makers of ECJ, JPF, HSF-SPIN (particular thanks to Alberto Lluch Lafuente for his help via Skype!).

I would like to thank my family; I hope they will be proud of me for being the first in the family to achieve a doctorate. The unshakeable encouragement and support of my sister and parents has helped me through a long period of studying at York: 8 years in all. And last, but by no means least, I would like to thank my beautiful fiancée Danusia for her love and support. (Edit: Post wedding minor corrections, now wife!)

DECLARATION

I declare that all the work in this thesis is my own, except where attributed to another author. Many of the ideas and figures in this thesis have appeared in the following publications:

1. Jan Staunton and John A. Clark, Searching for Safety Violations Using Estimation of Distribution Algorithms, in *Software Testing Verification and Validation Workshop, IEEE International Conference on Software Testing, Verification, and Validation* (ICST 2010).

   This paper won best Student Paper Prize for the workshop, and constitutes the content of Chapter 8.

2. Jan Staunton and John A. Clark, Finding Short Counterexamples in Promela Models using Estimation of Distribution Algorithms, in *Proceedings of the 13th annual conference on Genetic and Evolutionary Computation* (GECCO 2011).

   This paper was nominated for the best paper prize in the Search-Based Software Engineering track, and constitutes the content of Chapter 9.

3. Jan Staunton and John A. Clark, Applications of Model Reuse when using Estimation of Distribution Algorithms to Test Concurrent Software, in *Search Based Software Engineering - Lecture Notes in Computer Science* (SSBSE 2011).

   This paper constitutes the content of Chapter 10.

Dedicated to my wife Danusia, and my family.

Part I

INTRODUCTION

1 MOTIVATION

1.1 MULTI-PROCESSOR SYSTEMS

For many years, Moore's law provided a fairly accurate measure of the increase in processing power one can fabricate on a single chip over time. When purchasing a new processor during this time, one could expect faster completion of sequential tasks over an older processor. Recently, this trend has been invalidated due to the physical limitations of modern day chip manufacturing processes [Koch, 2005]. In order to continue delivering more performance per processor over time, CPU manufacturers have adopted *multi-core* architectures, processors consisting of two or more individual processing units called execution *cores*. Multi-core processors have also provided a convenient way of delivering more performance with minimal additional energy usage [Koch, 2005].

The trend of focusing on adding execution cores to chips is showing no sign of slowing down. At the time of writing, it is common to find two-core (dual-core) processors in most modern machines, and in some instances four-core (quad-core) processors in high end laptops and desktops. Intel, a major chip manufacturer, have plans to commercially market processors with 80 processing cores [Intel Corp., 2007].

1.2 UTILISING MULTIPLE CORES

Modern operating systems can use multiple processing cores to run many separate processes concurrently. In many modern operating systems, upward of 100 individual processes can be running at any time. The utilisation of multi-core systems by an operating system can result in a more responsive system overall when running multiple independent tasks. However, for any given task to see the benefit of parallel processing, application developers must explicitly divide the work required across the cores.

Most modern programming languages support the use of multiple cores. Parallel processing can be expressed in a variety of ways, but most use the notion of a sub-process. Languages can express this explicitly in terms of multiple threads of control or separate tasks within a program. Developers are now faced with challenges above and beyond traditional single-threaded software development. Processes within a concurrent software

program can execute at different rates and few software programming languages provide the facilities to express the rate at which processes execute. This phenomenon requires explicit methods of coordination to manage access to shared resources, and allow communication between sub-processes.

1.3 CONCURRENT SOFTWARE VERIFICATION

As with any language constructs, programming errors can be made when using the co-ordination primitives to mediate access to shared resources. There are a number of examples of errors that concurrent software programmers make, the most notorious being to allow the potential for *deadlock* and *race conditions*. Detecting such errors in concurrent systems is a hard problem, and verifying that a system if free from errors is often intractable. The main reason for this is the nondeterminism of execution of a concurrent software program. Nondeterminism arises from the varying rates at which each process in a concurrent software program execute. The rate at which processes execute is determined at run time by a scheduling mechanism. Each execution of the program can yield a different *interleaving* of actions or commands performed by each process within the program.

Concurrent faults typically manifest through a subtle interleaving of actions or commands performed by concurrent processes, and are typically not sensitive to program inputs. Faults may only be present on certain executions of the program, and those said executions may occur with low probability during normal running of the system. Race conditions are particularly notorious, as execution can continue in the presence of race conditions with functionally incorrect results. Static analysis techniques can highlight the potential for concurrent faults through examination of the source code, but produce spurious errors to the point that trust is lost in such techniques. Dynamic techniques can take a long time to reveal an error, if at all, and suffer from repeatability problems.

Another approach to concurrent software verification is to examine all possible executions of a concurrent program in order to discover errors. This requires examining the state space of a program. This can be viewed as a directed graph, where the nodes are the states of the program. The state of the concurrent software system consists of the composition of the states of each process within the system. Edges between states represent the progression of a single process with in the program, yielding a potentially new state. The main problem with this approach is that the state space of a concurrent program can be huge, potentially large enough to exhaust the computational resources available to any particular developer.

1.4 METAHEURISTIC SEARCH TECHNIQUES

It has been shown that software engineering tasks, particularly software verification, can be couched as an optimisation problem [McMinn, 2004; Clark et al., 2003]. Using this formulation of the problem, metaheuristic search techniques can be applied. Metaheuristic search techniques are optimisation strategies that improve candidate solutions iteratively using a fitness function as a guide, and are typically stochastic in nature. Metaheuristic search techniques have been used to solve hard optimisation problems, sometimes scaling better than analytical approaches. Some example techniques include simulated annealing, evolutionary algorithms, ant colony optimisation and particle swarm optimisation.

The large state space of a concurrent program presents itself as a target for search mechanisms. Search mechanisms can focus the exploration of the state space to areas that are more likely to contain concurrent faults, discovering faults efficiently with respect to computational resources. They also provide the potential to scale to large systems which may be of interest to industrial practitioners. Of particular interest to the author is the potential for the application of a search mechanism to yield a general concurrent fault finding tool that can be applied over a wide range of target languages and systems.

1.5 MOTIVATION FOR RESEARCH

The use of metaheuristic search for the verification of concurrent software is in the preliminary stages, with the bulk of the work residing in the proof of concept stage. This work is detailed in Chapter 6 of this report. There remain a number of metaheuristic search techniques yet to be tried on this problem. There is the potential for different representations of the solution space to be explored. Little investigation has been carried out as to the scalability of these techniques, as most of the state of the art has focussed on proving novel concepts. Little is known as to what constitutes a difficult concurrent bug for a particular search technique to detect, and whether metaheuristic techniques are indeed at all necessary for this domain.

1.6 RESEARCH GOALS

The aim of this research is to develop a new automatic method for detecting faults in a concurrent system, making use of a previously untested metaheuristic search technique based on Estimation of Distribution Algorithms. Metaheuristic search techniques

are not complete mechanisms, so the aim will be to provide a fault finding tool rather than a complete verification tool.

Thorough investigation as to the scalability of the new approach will be carried out, using rigorous experimental methods. Comparisons will be made between the new mechanisms and previous work reported in the field, including traditional non-heuristic methods. Work will be carried out investigating the application of the new mechanism to finding faults other than deadlock. These can include finding violations of liveness properties, as well as assertion violations.

1.7 STRUCTURE OF THIS REPORT

The structure of this thesis is as follows. The following chapter, Chapter 2, contains the hypotheses investigated in this report. Part ii consists of the literature survey detailing the difficulty of finding faults in concurrent systems, and previous efforts in this domain. Part iii details an algorithmic proposal for finding faults in concurrent systems based upon Estimation of Distribution Algorithms. Part iv presents empirical work giving evidence for and against the hypothesis investigated in this thesis. And finally, Part v presents conclusions and discusses the overarching hypothesis of this report in light of the evidence presented in the experimental chapters.

2 HYPOTHESES

2.1 INTRODUCTION

In this chapter, I will set out the hypotheses that will be addressed in this thesis. An overarching hypothesis is given, followed by more narrow hypotheses that divide the overarching hypothesis into strands of research. Each strand is then treated individually in separate chapters, with a summary at the end. The hypotheses given assume that an approach based on Estimation of Distribution Algorithms (EDAs) to finding concurrent faults will be developed as part of the investigation.

2.2 HYPOTHESIS AND RESEARCH STRANDS

The overarching hypothesis addressed in this thesis as as follows:

Estimation of Distribution Algorithms are an effective mechanism for detecting and debugging concurrent faults.

In order to investigate this hypothesis, I will report on three strands of research. The respective strands will:

1. Establish the ability of EDAs to find concurrent faults in programs described by mainstream industrial languages.
2. Investigate the ability of EDAs to find a wide range of error types in complex industrially derived systems.
3. Demonstrate the ability of EDAs to scale to large problem sizes, using features unique to EDAs.

I will now detail the above strands of research and their respective sub-hypotheses.

2.3 FINDING FAULTS IN MAINSTREAM LANGUAGE CODE

Hypothesis: *Estimation of Distribution Algorithms are an effective mechanism for detecting faults in systems described by mainstream languages.*

For an algorithmic proposal to be applicable to real-world systems, it must be established that the algorithm can examine systems described by mainstream industrial languages. These

languages include Java and C++/C. The goal is to detect concurrent faults accurately whilst being computationally efficient. If this ability can be established, then the integration of EDAs into mainstream programming tools, such as popular integrated development environments like Eclipse and Visual Studio, becomes possible. Consequently, the algorithmic proposal can become a tool in the arsenal of a concurrent software developer, running in the background alongside other static analyses and dynamic checks.

The author believes that, along with other metaheuristic techniques such as Genetic Algorithms and Ant Colony Optimisation, EDAs will be able to detect concurrent faults in mainstream languages where other techniques fail. The author also believes that EDAs have the potential to outperform other metaheuristic techniques in certain situations, achieving an equivalent or higher detection rate whilst using equivalent or fewer resources. This strand will establish a proof of concept implementation which will be extended to provide evidence for the hypotheses in subsequent strands.

2.4 FINDING AND OPTIMISING WIDE RANGES OF FAULTS IN COMPLEX SYSTEMS

Hypothesis: *Estimation of Distribution Algorithms can find not only faults in complex industrial systems, but also optimise them.*

For any algorithmic proposal to gain acceptance by practitioners developing systems to be deployed in real world scenarios, the proposed algorithm must be able to find a wide variety of fault types in large and complex systems. Error types can be divided into two categories, safety or invariant properties, and liveness properties (these categories are explained in the literature review). If the proposed EDA-based algorithm can be shown to detect faults efficiently in both categories, then the proposal can be useful to practitioners in practical scenarios.

In addition to establishing that the proposed EDA-based algorithm can find a wide variety of error types, this strand will look at finding those faults in systems derived from real world industrial scenarios. The systems examined will be complex in nature, to simulate application in practical settings. The quality of the information relating to the faults found will also be scrutinised. The author believes that an EDA-based approach will be able to find higher quality information regarding a fault, either through yielding a large quantity of useful information, or eliminating useless information.

2.5 scaling to large systems

Hypothesis: *Estimation of Distribution Algorithms can scale to find faults in large complex systems.*

The author believes that features unique to EDAs can aid in performance when detecting faults in large complex systems. By exploiting features unique to EDAs, an advantage can be gained over previous work in the problem domain, metaheuristic or otherwise. By demonstrating the ability of an algorithmic proposal to scale to large systems, EDA-based concurrent fault finders gain credibility in potentially being used in industrial settings as part of a practitioner's tool set.

2.6 summary

The three strands of research outlined above are addressed in Chapters 8, 9 and 10 respectively. In the next Part of this thesis, I will conduct a literature survey detailing previous effort in the area of detecting concurrent faults.

Part II

LITERATURE REVIEW

3 CONCURRENT SOFTWARE VERIFICATION

3.1 INTRODUCTION TO CONCURRENT SOFTWARE

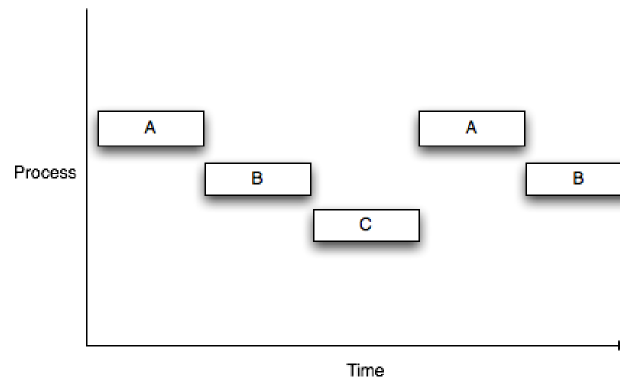
Traditional sequential software programs consist of a sequence of atomic actions applied to some input data. An atomic action is indivisible, i.e. cannot be broken down into smaller steps. For a given input, a sequential program typically exhibits a particular behaviour, called an execution. The behaviour can be described by a sequence of states. A state of the software program consists of the current values in a memory and a program counter that represents the next action to be executed. The program progresses by executing the next action represented by the current program counter on the current state. This yields a new state that potentially includes an incremented program counter. The program whilst executing is sometimes called a process.

A sequential software program consists of a single process. Repeated executions of a sequential program with a particular input will yield the same behaviour each time, unless there is some stochastic element to the program[1]. In mainstream programming languages such as C and Java a process is implemented as a thread, and a sequential software program consists of a single thread. A thread encapsulates the program counter and any implicit state information required for process execution, such as values in CPU registers.

Concurrent software programs consist of n processes logically progressing at the same time, where $n > 1$. Each of the n processes is a sequential process as described above. The state of a concurrent program is made up of the cross product of the states of the n processes. Concurrent software programs can be useful when decomposing a problem. For example, when constructing a reactive software program responding to events in an environment, dedicating a single process to handle a particular event is a popular method of implementation. Using this model, the process can respond to a particular event independently from the other processes in the software, yielding a quick reaction. In this type of system, a number of processes are co-operating to solve a problem. A concurrent software program can be thought of as a concurrent system, and the terms shall be used interchangeably throughout this document.

1 Pseudo-random number generators are fundamentally deterministic, and the input to the random number generator can be considered as input to the program.

Figure 1: Round robin schedule for processes A, B and C



3.1.1 *Scheduling*

The rate at which each of the n processes progresses is dependent on the implementation of a process. In a software system, n processes typically run on one or more *processors*. The prominent view of n processes progressing is that of the actions of each of the n processes are being interleaved on a single processor according to some scheduling policy, the simplest schedule being a round robin schedule (Figure 1). The round robin scheduler switches between each of the n processes, giving each process an equal amount of time steps to execute actions before switching to another process. The act of switching between processes is known as a *context switch* [Burns and Wellings, 2001]. More advanced scheduling policies exists, such as pre-emptive scheduling [Silberschatz et al., 2004], which allow for a process to pre-empt or interrupt the scheduler and be scheduled immediately, without waiting for the previously determined next turn of the process. This enables a concurrent system to quickly react to a particular event, rather than waiting on a predetermined schedule.

In an environment where the execution time of a single processor is divided between n different processes, the n processes do not actually progress at the same time. A single process is active at any one time, and a scheduler switches between them. This pseudo-concurrent [Magee and Kramer, 2006] environment allows the appearance of processes progressing at the same time. When there are two or more processors executing a concurrent software program, the processes that make up said concurrent software program can be divided between the processors. In an environment where there are say 4 processors, up to 4 processes can be physically progressing at the same time. This can be termed as real concurrent execution, as opposed to pseudo-concurrent execution [Magee and Kramer, 2006]. In industrial

systems, it is typical for the number of processes to exceed the number of available processors to execute them. Therefore, some pseudo-concurrent execution can and likely will take place even in real concurrent hardware. In general, viewing real concurrent execution as pseudo-concurrent execution yields few practical limitations [Magee and Kramer, 2006] and a pseudo-concurrent view of concurrent execution is used in a wide variety of analysis techniques (discussed later).

In general, the scheduling of each process in a concurrent software program is hidden from the programmer by the programming language. The scheduling policy is typically implemented by a runtime schedule manager with a particular scheduling policy, and is seldom explicitly referenced by the programming language. Examples of such languages are Ada and Java, both of which schedule processes at runtime using a runtime schedule manager. As a consequence, software programmers constructing a concurrent program for either of these languages must assume that any of the n processes in the program can be active at any time, and that any currently executing process can be "swapped out" by the scheduler in favour of any other process at any time. The scheduling of each of the n processes is viewed as nondeterministic, and the execution of a concurrent software program can also be viewed as nondeterministic. There are some exceptions to this model, such as coroutines and cyclic executives [Burns and Wellings, 2001], which allow explicit scheduling of processes as part of the software program specification.

The execution of concurrent software programs has been established as nondeterministic. A consequence of this is that a concurrent program executed two or more times with the same input may yield different results. For example, consider a concurrent software program P with two threads, A and B. Thread A outputs the numbers 1 to 26 sequentially in ascending order, whilst thread B sequentially outputs the letters a through z. A particular execution of P may yield the output trace $1a2b3c4d5e..25y26z$. However, on a subsequent execution, the threads may be scheduled differently, potentially yielding the trace $123a456bcd7e...$. The nondeterminism introduced in concurrent software, and concurrent systems in general, is a major hurdle in the verification of such systems [Clarke et al., 2000].

3.1.2 *Communication*

The simplest concurrent software program consists of 2 processes progressing independently, i.e. each individual process can progress without dependence on the progress of any other process. An example of such a process is program P described above [2].

2 This is not strictly true, as both processes are sharing an output resource/facility.

Figure 2: Mutual exclusion implementation using monitors (Java style)

```
1 //s is a shared memory location
2 synchronize(s) { //lock s
3 //Do something with s
4 s++;
5 } //s is now unlocked
6 //Do some other work...
```

In general, a concurrent software program is more complex if the behaviour of any process within the software program is dependent on progress of another. Dependencies between a process A and a process B are caused by *communication* between A and B. Communication can happen implicitly through sharing of resources, or explicitly through coordination primitives. In order for process A and B to communicate safely and successfully, the two processes must explicitly coordinate their actions.

Most mainstream programming languages support a shared-memory model of communication between threads/processes. This involves the various processes reading and writing to shared memory locations in main memory, sharing a global state. In order to control access to shared memory locations, locking mechanisms are typically used. Examples of locking mechanisms include semaphores, mutexes and monitors [Silberschatz et al., 2004; Burns and Wellings, 2001]. Locking mechanisms can be used to ensure mutually exclusive access to a shared memory location, or to control the order in which processes receive access. An example of this process is described in Figure 2, showing how processes can coordinate mutually exclusive access to a shared memory location. Each thread must explicitly lock s before manipulating it, ensuring that no two processes read and write inconsistent values to s in shared memory. Any process that fails to lock s before manipulation may cause a data race (discussed later in this chapter). Mainstream languages such as Java and C# both implement shared-memory model coordination mechanisms.

Message-passing is another communication mechanism available in certain mainstream programming languages. Message-passing between processes can occur in a number of ways. The first is asynchronous message passing, where the sending process sends the message and progresses onward immediately after sending. The message is buffered somehow until the receiving process is ready to receive. The second is synchronous message passing, where both the sending process and receiving process must explicitly synchronise before a message can be exchanged. This mechanism does not require a buffer and is sometimes known as rendezvous [Burns and Wellings, 2001].

Message-passing highly constrains how processes can communicate, and therefore makes it easier to predict the behaviour of a system. However, the constraints can make simple communication (such as that allowed by shared memory models) difficult and can be computationally expensive at runtime. [Burns and Wellings, 2001] gives a good overview of the various message passing schemes and the languages that implement them, with Ada being a prominent example.

3.2 PROMINENT FAULTS IN CONCURRENT SOFTWARE

[Beizer, 1990] describes a taxonomy of bugs that can surface in sequential software programs, along with techniques for finding those bugs. Concurrent software programs can exhibit the bugs outlined in [Beizer, 1990], and additional bugs that manifest themselves as a consequence of communication between processes. Bugs may not exhibit their behaviour on every execution due to the nondeterministic nature of concurrent software, and indeed may only manifest themselves during a subtle interleaving of process actions. [Farchi et al., 2003] outline the fact that concurrent bug patterns usually manifest in the form of an unforeseen interleaving of processes that causes unexpected or undesirable behaviour.

In this section, I shall outline some of the prominent concurrent faults that occur in real concurrent programs along with examples.

3.2.1 *Deadlock*

Perhaps the most infamous concurrent fault in the literature is that of *deadlock*, also called *deadly embrace* [Dijkstra, 1968]. A concurrent software program is in deadlock when none of the processes that constitute the program can progress. The typical causes of this particular bug relates to the locking of resources. The locking of resources R by a set of processes P can be represented as a resource allocation graph G. There are two types of nodes in a resource allocation graph, one for processes and one for resources. An arc in the graph G from process P_i to resource R_j represents the fact that a process in P_i wants to acquire resource R_j. An arc from resource R_j to process P_i represents R_j having been locked by process P_i. A particular graph G represents the resources locked and requested at a particular time instance. If a cycle exists in G, the processes involved in the locking of the resources involved in that cycle are said to be deadlocked. This is illustrated in figure 3.

This graph can reduced to a *wait-for* graph, where each node is a process, and an arc from process P_i to P_j represents that P_i is

Figure 3: Resource allocation graph for resources 1 through 5 and processes 1 through 5, loosely based on [Silberschatz et al., 2004]

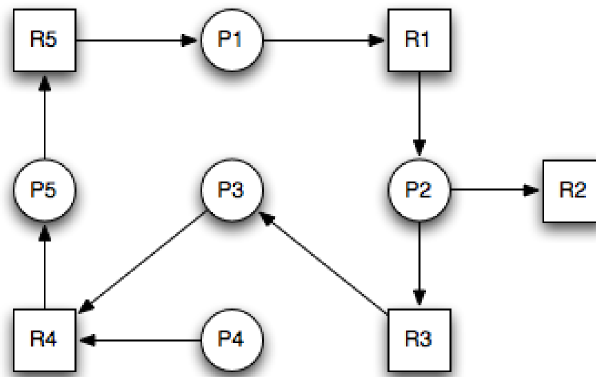
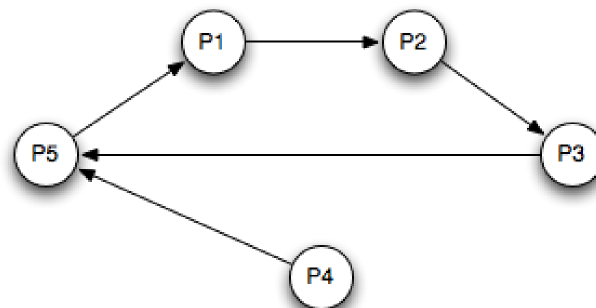


Figure 4: Wait-for graph for processes 1 through 5, loosely based on [Silberschatz et al., 2004]



waiting for a resource that P_j currently holds. A wait-for graph that is equivalent to the resource allocation graph in figure 3 is depicted in figure 4.

One of the simplest cases of deadlock is as follows. Concurrent software program P is made up of two processes, A and B. A and B require access to resources R and S, the access to both is governed by the locking mechanism described above. Process A accesses the resources in R and S in that order, and process B accesses them in the order S and then R. When each process has acquired both resources, the resources are used and then released. Each process locks, uses, and unlocks the two resources *ad infinitum*.

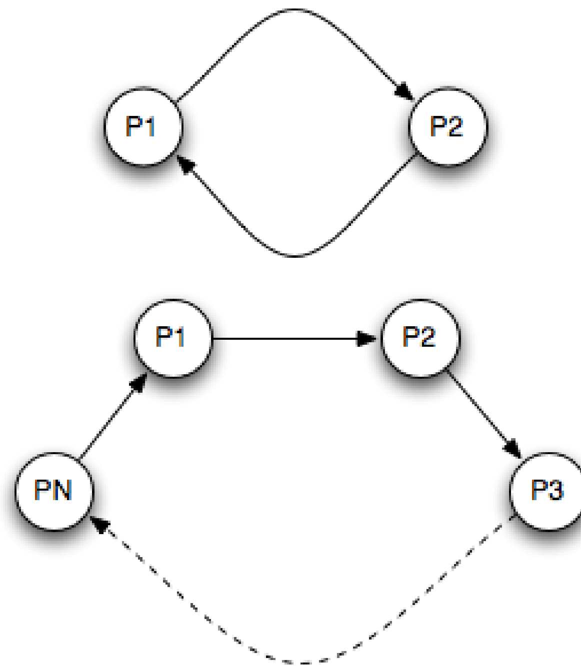
It is possible for this particular software program to enter a deadlocked state. If process A locks resource R and then process B locks resource S, then both processes are waiting for a lock on the other resource which is held by the other process. Therefore, neither process can proceed as they are waiting on a lock that will never be unlocked. This situation could have been overcome by enforcing that rule that all processes must lock resources in the same order, i.e. R then S. This deadlocking situation can be generalised to a circular chain (wait-for cycle) of n processes, where for all n processes, the ith $(0 \leqslant i \leqslant n)$ process is waiting on a resource held by process $i+1$, modulo the number of processes n. Figure 5 shows a lock graph illustrating a 2 party deadlock and an n party deadlock.

[Coffman and Elphick, 1971] describe four conditions that must be present for deadlock to arise. The first (1) is that tasks must acquire resources for exclusive use, known as the *mutual exclusion* condition. Second (2), tasks can hold resources they require whilst waiting for the acquisition of additional resources, known as the *wait for* condition. Third (3), tasks that are holding resources cannot be forced to release them in favour of a another task by some resource manager, known as the *no pre-emption* condition. And fourth (4), a circular chain of tasks exist such that each task has locked resources that the next task in the chain requires, known as the *circular wait* condition. In order to avoid deadlock when constructing a concurrent system, one of these four conditions must be eliminated.

3.2.1.1 *Dining philosophers problem*

The Dining philosophers problem [Dijkstra, 1968] is likely the most famous example of a potential deadlock situation. The situation is as follows. A set of *n* philosophers share a circular dining table (Figure 6), in this case 5. Each philosopher alternates indefinitely between eating spaghetti and thinking. In order to eat, a philosopher requires two forks. Unfortunately, the philosophers can only afford a fork each (5 forks in this case). Prior to the

Figure 5: Wait-for cycle depiction



eating arrangement, a fork is placed in between each pair of philosophers and they agree that they will only use the fork to the immediate left or right of them.

The resource in contention in this situation are the forks. In order for a philosopher to do some processing (eat), a philosopher must acquire a lock on both the fork to her/his left and right. Depending on how each of the philosophers behave, there is the potential for deadlock. For instance, 5 naive philosophers would implement behaviour described in figure 7.

There is a potential deadlock situation here if every philosopher follows this behaviour. It is easy to imagine an interleaving of each of these processes where philosopher 1 picks up the fork to the left, then philosopher two acquires the fork to the left and so on. If all the philosophers have acquired the left fork, then they are all waiting to acquire the right fork, a resource which has already been acquired by another philosopher. None of the philosophers can progress, and the philosophers are said to be in deadlock.

A different policy shared by each philosopher can be established such that one of the four conditions for deadlock is nullified. An example is that acquisition of both forks by any particular philosopher is atomic, eliminating the wait for condition (2). In addition, the philosophers could time out if they have waited for a lock for too long, eliminating the no pre-emption condition (3).

Figure 6: Dining Philosophers problem

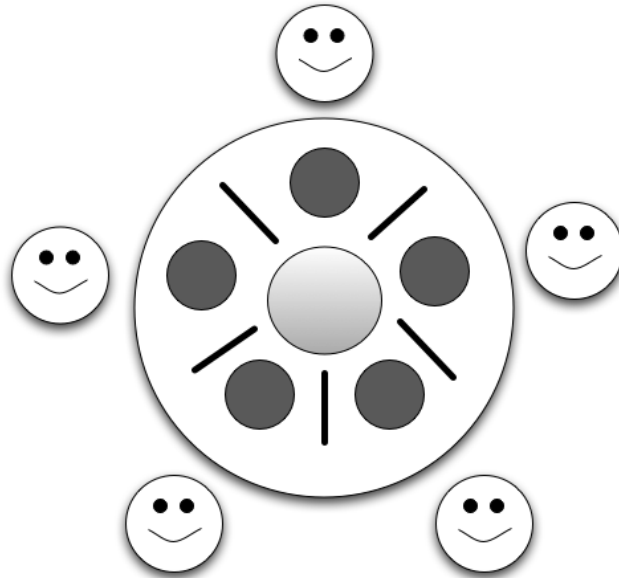


Figure 7: Naive dining philosopher behaviour

```
1 while (hungry) {
2         acquire_left_fork();
3         acquire_right_fork();
4         eat();
5         release_left_fork();
6         release_right_fork();
7 }
```

The dining philosophers problem is important because it shows the potential for deadlock even when processes do not require the majority of the resources in a system [Coffman and Elphick, 1971]. The dining philosophers is a demonstration of the most extreme example of the fourth requirement for deadlock.

3.2.2 *Starvation*

A phenomenon related to deadlock is that of *indefinite blocking*, or the more popular term *starvation*. Starvation is when a process cannot progress due to one factor or another. Starvation can occur in a number of ways, but I feel the behaviour is best expressed as an example. Consider the dining philosophers problem, with a co-ordination strategy that involves the use of a single *token*. Whichever philosopher currently holds the token can pick up the required forks and eat. Once a philosopher has eaten, the token is placed in the middle of the table for any philosopher to pick up. The token becomes the critical resource in the system, and is managed by an arbiter.

In this co-ordination system, the token dictates which philosopher can eat making the arbitration of the token a crucial component. A "fair" arbitration process could pass the token around in a round robin fashion. However, a "cruel" arbiter could favour some philosophers over others, starving (quite literally) the discriminated philosophers. A philosopher can become indefinitely blocked if they are never given the token under this regime.

3.2.3 *Livelock*

Livelock is similar to deadlock in that the processes that make up a software system cannot make meaningful progress. In a deadlocked system, processes cannot make any progress whatsoever due to indefinitely waiting on some resource. During livelock, processes may be progressing in the strictest sense, but not actually doing anything useful. For example, imagine a situation where two processes are in deadlock, but there is a external deadlock detection agent that recognises the situation, and signals the two processes to release their respective locks and try again. It is potentially the case that the two processes could again become deadlocked, and then forced to releases their locks for second time and so on. In this situation, the two processes are still progressing, but not doing anything useful. Whilst the symptom of deadlock is easy to detect, testing generally for livelock is a more difficult challenge, as it is difficult to define in the general case.

Figure 8: Data race example code

```
int tmp_var = x;
tmp_var = tmp_var + 50;
x = tmp_var;
```

Figure 9: Nasty interleaving

```
Global: int x = 0;
Thread one: int tmp_var = x;
Thread two: int tmp_var = x;
Thread two: tmp_var = tmp_var + 50;
Thread one: tmp_var = tmp_var + 50;
Thread one: x = tmp_var;
Thread two: x = tmp_var;
```

3.2.4 *Data races*

Race conditions occur when two more more processes access a shared memory object without proper co-ordination. This results in the processes interfering with each other, leading to functionally incorrect/inconsistent results. Race conditions are particularly notorious as a concurrent software program can potentially continue processing in the face of race conditions, whereas deadlock has an obvious "show stopping" behaviour.

A simple example of a data race is as follows. A concurrent Java program is made up of two threads/processes that access the shared/global integer variable x, initially set to 0. Both threads apply the operations outlined in figure 8, each with a unique thread-local $tmp_{v}ar$.

The intuitive view of this operation is that both threads will increment x by 50, yielding $x == 100$. However, it is possible for the interleaving outlined in figure 9 to occur. The result in x after this interleaving is 50. This phenomena is known as interference, as the execution of each thread has interfered with the execution of the other.

In order to avoid interference, the threads must negotiate *mutual exclusion* over the *critical section* of each thread to ensure the atomicity of the update of x. A critical section is a segment of code that manipulates a shared memory object. In this case all three statements in figure 8 constitute the critical section of each process. Critical sections are typically related to some shared memory object. For any shared memory object, at most one process can execute a related critical section. Negotiation of critical sections can be implemented using some of the synchronisation mechanisms highlighted on page 15.

Data race errors can be more subtle than the example described above. For example, some assignment operations in some programming languages are non-atomic [Farchi et al., 2003]. For instance, Java guarantees atomic assignment to all primitive types except for `long` and `double` [Gosling et al., 2005]. An assignment to a 8 byte variable, such as a *long* for instance, may divide the operation into multiple stages, but appear as a single operation in the program code (e.g. long x = 100). This may lead a programmer to believe that there is no interference over the `long x`, but an erroneous interleaving could exist. Another example that is common to a lot of imperative languages is an operation equivalent to `x++`. Again, this looks like a single operation, but can be broken down at the assembly instruction phase into multiple stages. [Farchi et al., 2003] describe this as the difference between a programmer model of the execution of the program (the source code level) and the actual execution model of the program (the assembly level).

3.3 CONCURRENT SOFTWARE VERIFICATION

The verification of sequential software programs has the goal of verifying whether a software program accurately implements a specification. The methods for achieving this goal can be divided into formal and informal methods. Formal methods use complete rigourous mathematical reasoning to prove that a software program meets some specification. Formal methods include deductive theorem proving, modelling and model checking.

Informal methods use non-complete techniques to gain assurance that a software program meets some specification. A good reference on the testing of sequential software are the complimentary books [Beizer, 1990] and [Beizer, 1984]. The combination of the two works provides an excellent description of the world of sequential software testing.

The nondeterminism of concurrent software programs was introduced above, showing how two executions of a concurrent software program with the same input can yield different behaviours. The nondeterminism introduced by concurrent processes makes concurrent software more difficult to test than sequential software. For instance, [Beizer, 1990] describe a great number of testing methods that rely on deterministic input/output relationships defined by a sequential software function/program. This deterministic input/output relationship is not held by concurrent software programs due to the nondeterministic nature of the scheduler. Due to the nondeterministic nature of concurrent software, a bug may only manifest itself on a particular interleaving of processes. This is evident in the dining philosophers

problem, where deadlock occurs only on particular interleavings of philosophers locking forks.

Below is a survey of various techniques for verifying or testing concurrent software. Both formal and informal methods are included, along with a brief section on coverage metrics specific to testing concurrent software.

3.3.1 *Coverage metrics*

Software coverage metrics, in single-threaded software testing, define a measure of completeness for a given set of test cases on some program [Beizer, 1990]. Some examples of coverage metrics are statement coverage, branch coverage, and predicate coverage. Data-flow based metrics can also be used, such as the define-use metric [Beizer, 1990]. The notion of *100% path coverage* defines a process that exercises a program over all possible executions of that program, subsuming all other coverage metrics. [Beizer, 1990] states that this may not be achievable, or impractical to achieve.

100% path coverage of a multi-threaded program requires that all possible interleavings over all possible inputs are exercised. The set of all possible executions of a concurrent program can be extremely large, typically increasing exponentially with the number of processes. Rather than aiming for 100% path coverage, a programmer may want to target testing at crucial areas that satisfy coverage metrics for concurrent software. Coverage metrics from single-threaded software testing can be used, but may not be useful in revealing bugs that rely on a particular interleaving of threads/processes.

[Bron et al., 2005] define *synchronisation coverage* as a possible metric for revealing illusive bugs in multi-threaded software. The metric measures how well blocking statements, such as `synchronise` and `wait` in Java, are exercised in a set of given test cases. To achieve complete synchronisation coverage over a particular blocking statement B, all processes that can execute B must be blocked by B, and must cause another process to be blocked in one or more test cases.

[Lu et al., 2007] abstract slightly from the work of [Bron et al., 2005] and others to form a hierarchical set of criteria relating to coverage of possible interleavings, that follow a subsumption relation similar to the coverage metrics outlined in [Beizer, 1990]. The metrics target interleavings of accesses to shared variables only, disregarding interleavings that involve processing local to each process.

3.4 STATIC VERIFICATION TECHNIQUES

Static analysis techniques examine the structure and source code of a program without actually executing the program [Beizer, 1990]. Static analysis techniques can potentially reveal errors using fewer computational resources than dynamic techniques that require the software to be executed. The scalability of static analysis techniques is usually sensitive to the size of the source code of a program, rather than the size of the path or state space.

Static analysis techniques typically operate on similar structures to those used during the compilation of programs. Example structures include control flow graphs and data flow graphs. The structures used can be considered as an abstract interpretation [Cousot and Cousot, 1977] of the program. Because static analysis techniques operate on an abstract interpretation of a program (i.e. an approximation), results from such techniques can be inaccurate, potentially reporting false positives [Beizer, 1990]. What follows is an overview of some static analysis techniques for finding concurrent bugs, and a summary of the techniques.

3.4.1 *Lockset analysis*

Lockset analysis techniques analyse the usage of locks in the source code of a program. The term Lockset Analysis was coined in a work on the dynamic analysis of concurrent programs [Savage et al., 1997], but the general mechanism has been applied in some static analysis work. The basic Lockset analysis algorithm is extremely simple and assumes that nothing is known about which locks protect which shared memory objects. The technique monitors what locks are held by a thread accessing a particular shared memory object.

The RacerX [Engler and Ashcraft, 2003] static analysis tool utilises lockset monitoring to find race conditions and deadlocks in large programs, and the authors have targetted operating systems such as Linux and FreeBSD in their work. The primary focus of the effort in [Engler and Ashcraft, 2003] is to reduce the amount of spurious errors, given the size of the programs targeted. The Jlint tool [Artho and Biere, 2001; Artho and Havelund, 2003] also employs lock monitoring in order to detect potential deadlocks and data races.

3.4.2 *Data-flow analysis*

Various data-flow analysis techniques can be used to aid in the static analysis of concurrent software. Data-flow analysis in concurrent software is a tricky endeavour, as the value of a variable can be affected by the actions of multiple threads. Various data-

flow analysis techniques for single-threaded software [Beizer, 1990] have been extended for concurrent software.

Program slicing [Tip, 1994; Weiser, 1981], for instance, has been extended to concurrent languages such as Java [Zhao, 1999], and it seems that the ideas can be applied to other languages. For a particular line of source code l, program slicing identifies all the lines of code and variables that can affect the outcome of l. Program slicing, and data-flow analysis in general, can be useful in identifying the potential for data races, as statements that effect shared variables can be identified.

3.4.3 *Summary of static verification techniques*

Static analysis techniques for concurrent software allow for a quick and cheap test of a software program for potential faults at run time. Work in the field [Engler and Ashcraft, 2003; Artho and Biere, 2001; Artho and Havelund, 2003] has noted short runtimes for incredibly large pieces of software. However, effort is required to reduce the amount of spurious errors reported to the user, as too many errors can result in a debugging headache for users. This problem can make the techniques useless when faced with large software programs/systems. Static analysis techniques can point out potential deadlocks and data races within a system, but other concurrent issues such as starvation and livelock are hard to detect statically.

3.5 dynamic verification techniques

Dynamic verification techniques analyse programs at run time, executing a program artefact and monitoring execution in order to detect errors. It has been discussed previously that the execution of concurrent programs is nondeterministic, and that concurrent bugs may become apparent in some executions and not others. The dynamic verification techniques described below aim to increase the likelihood of a concurrent bug appearing during a test, or use real execution data to highlight potential problems.

3.5.1 *Schedule altering techniques*

The set of possible thread interleavings or executions for a particular concurrent program can be extremely large. The subset of the possible interleavings that exhibit concurrent bugs can be extremely small. There exist a number of heuristic techniques that attempt to force the execution of a concurrent program at run time onto a particular interleaving that may reveal a concurrent

bug. The techniques leave the other functional properties (i.e. those not related to deadlock and data races) intact.

A number of tools exist for the Java programming language that attempt this kind of procedure. The ConTest [Edelstein et al., 2003] tool from IBM creates additional test cases by inserting schedule altering behaviours into suspicious parts of existing Java test cases. The schedule altering behaviours are as simple as `sleep(t)` and `yield()` statements, trying to force unforeseen thread schedules. raceFinder [Ben-Asher et al., 2003] can be viewed as an extension to ConTest, with a particular focus on data races. raceFinder uses coverage metrics, such as synchronisation coverage, to aid a heuristic in choosing which threads are scheduled next in particular parts of the code. [Stoller, 2002] describe a technique that inserts random context switches at selected points in source code, a technique [Eytani et al., 2007] refer to as *noise making*.

A few unit testing frameworks for concurrent software components exist that allow programmers to explicitly check certain interleavings of events in their software. [Pugh and Ayewah, 2007] describe the *MultithreadedTestCase* framework that provide a set of generic tools that do precisely that, allowing test threads to wait for certain conditions to become true before proceeding, such as a value being updated in another thread. The tests are manually generated in this particular framework, but there is potential for some test cases to generated automatically and inexpensively. Additionally, the unit tests generated can be used as part of a regression suite [Beizer, 1990].

3.5.2 *Runtime monitoring*

Runtime monitoring techniques typically instrument concurrent programs in order to analyse behaviour. The Atomizer tool [Flanagan and Freund, 2008], for instance, uses instrumentation to perform runtime lockset analysis, leading to greater accuracy over a static approach. Atomizer tests for atomicity violations. Atomicity requirements can be specified by user annotations of source code. The ConTest tool also uses runtime monitoring to guide the placement of context switching commands, measuring various coverage metrics at runtime. Static analysis of instrumentation logs can be performed in order to determine the precise cause of an error. This process is sometimes referred to as post-mortem analysis.

3.5.3 *Summary of dynamic verification techniques*

Dynamic analysis techniques have an advantage over static techniques in that they examine genuine executions of the software

program. Most concurrent bugs, including starvation and livelock, can be detected using proper instrumentation and logging facilities. Tools such as ConTest and raceFinder have shown that additional effective testing can be generated inexpensively as part of an existing testing framework. Concrete errors traces are provided by dynamic techniques, as the bugs found can have the events leading up to them logged and reported.

Dynamic verification techniques have a few drawbacks. The major one is that execution of the program can be very expensive. Due to the nondeterminism in concurrent software, bugs may still not show up even during heavy stress testing, and may only show up in a low probability catastrophic event. The unit testing techniques are limited to the tests that the user specifies, and may miss crucial interleavings when using substandard unit test suites.

3.6 formal methods

Formal verification methods attempt to prove that a system correctly implements some specification. One such method is the use of deductive mechanisms to prove that a system meets a formal specification. Descriptions of systems are manipulated in order to show the equivalence of a specification and some implementation. In general, this process requires a high degree of manual human effort, but can be partially automated. As well as refinement to implementation, the behaviour of models of systems (or indeed the system artefact itself) can be examined in order to prove particular properties. Various methods of specification/modelling and analysis exist, and some are described briefly below.

3.6.1 *Specification/Modelling*

One method of expressing the behaviour of a concurrent system and verifying properties of that system is through the use of a process algebra [Roscoe et al., 1998]. Process algebras, also known as process calculi, express the communication and synchronisation that occurs between a set of processes. Rules for manipulation of process descriptions are packaged with a particular process algebra, allowing transformation of systems and proofs of equivalence. This allows for formal verification of the behaviour of a system, showing that systems implement a particular specification in all circumstances. Examples of process algebras are CSP and CCS [Milner, 1982; Hoare and Hoare, 1985; Roscoe et al., 1998].

Process algebras typically do not model the internal state of each process that makes up a concurrent system. Languages exist that express communication between processes as well as their

internal state. An example language here is the original version of CSP, from which the programming language occam takes inspiration [Jones and Goldsmith, 1988; Roscoe et al., 1998]. Languages that allow the modelling and manipulation of state take the formal process "closer" to real running code. Although Java is in the strictly technical sense a language that can describe/model systems, it lacks the tools for formal manipulation that allow for transformation and proof.

3.6.2 *Annotation-based theorem proving*

Theorem proving is a technique used to prove whether some implementation meets a specification. The implementation in this case is the program source code. The specification can be in the form of a full formal document, or annotations in program source code. Tools that implement this sort of functionality exist for Ada in the form of an extended SPARK Ravenscar analyser [Amey and Dobbing, 2003] that allows annotations over concurrent tasks. Extended static checking systems [Detlefs, 1996], a blanket term for static analysers that take additional annotations as input, exist for a variety of other languages such as Java [Flanagan et al., 2002] and Modula-3 [Leino and Nelson, 1998].

In order to obtain a specification, a potentially large amount of manual human effort is required. The annotations aid the programmer in further specifying the design, and the tools automatically check if the current implementation can invalidate that design.

3.6.3 *Model checking*

When verifying various properties or behaviours of a model, an automated approach exists that examines all the possible behaviours of a model or specification. The state space of a model is extracted from the model and exhaustively checked for conformance with a specification given by a human. This process is known as *model checking* [Clarke et al., 2000; Baier and Katoen, 2008]. If a violation is found, a concrete trace of events leading up the error can be reported.

Efficient algorithms exist to check a variety of properties, including deadlock and race conditions. Using these techniques, all possible interleavings of process actions can be examined explicitly without execution of the program. This eliminates some of the downfalls of traditional static and dynamic analysis. As noted earlier, however, checking all possible paths in a concurrent program can be intractable as large state spaces can exhaust available memory. Heuristic methods can be used to partially check a model, focusing on areas of the state space more likely to contain

an error. The heuristic methods available can be thought of as a testing mechanism for a model, but unlike dynamic testing techniques, have explicit control of the scheduling of the processes of a system. State spaces can be extracted from program source code, such as Java and C# and checked in a similar fashion.

3.6.4 *Summary of formal methods*

Whilst formal methods offer the prospect of proving that a concurrent system always functions correctly in all possible paths, there are a number of drawbacks. The deductive mechanisms described above require a great deal of human effort, something that may not be possible in industrial settings. There is also the potential misalignment between a formal proof of an abstract version of a system, and the refined/implemented version of the system. Model checking goes some way to alleviating the human effort problem, by automating almost all of the proving process beyond the specification of properties. However, checking all possible paths of a typical industrial program can be intractable, limiting the practical applications this technique.

3.7 SUMMARY

This chapter has outlined the basics of concurrent programming, and the prominent issues when dealing with concurrent software verification. The problem of finding concurrent bugs has been established as difficult due to the nondeterminism of a software program process schedule. Various methods for testing concurrent software have been discussed, and a few address the interleaving issue explicitly.

All of the techniques described have a number of drawbacks that limit their applicability to industrial systems. The static analysis techniques are all computationally efficient, but yield a large number of false positives and are not complete. Dynamic techniques eliminate false positives, but can be expensive to run (taking a large number of runs to find errors) and can suffer from repeatability problems. Whilst formal methods can prove that systems function correctly in all paths, often the human effort required is too much, or the computational resources too few.

A potential way forward in this area is to try to eliminate the failings of the various techniques. It will be difficult to reduce the human and computational costs in the formal methods space when proving programs correct. However, attacking the downfalls that incomplete methods have may prove fruitful. One potential avenue of research would be to tune the parameters of both the static and dynamic methods. However, I see more potential in the heuristic model checking arena. Heuristic model

checking mechanisms use a heuristic to narrow down the search of a state space, and aim for error detection rather than a complete proof of correctness. Heuristic model checking allows a practitioner to use the rich theoretical and specification framework provided by model checking, whilst potentially avoiding the high computational cost of exhaustive verification.

Metaheuristic search techniques have shown promise in a wide variety of areas for finding solutions to problems in large solution spaces. Searching the potentially vast state space of concurrent programs seems like a viable target for metaheurstic search algorithms. In the following chapters, I will give a brief introduction to model checking, metaheuristic search and research that combines the two. I will outline gaps in the research, and then describe how I have addressed those gaps.

4 MODEL CHECKING

4.1 INTRODUCTION

Model checking is an automated technique used for the verification of finite-state concurrent systems [Clarke et al., 2000]. Model checking is a formal method that can be used to gain assurance that a model of a system exhibits specified properties. To this effect, a model checker is employed to systematically check all possible behaviours of a model of a system to ensure that it meets a specification. Model checking can be used in addition to or as an alternative to other methods of verifying a system.

The main concept of the method is that a model of the system is constructed, usually in the form of a finite-state automaton. The states in this automaton correspond to the possible states of the system, and the transitions between those states are actions that effect some change in the system being modelled. Paths through the automata correspond to possible behaviours of the system, sometimes referred to as *properties* [Baier and Katoen, 2008]. Specifications are written using a language that can express sets of paths in the automaton. Efficient algorithms are then employed to ensure that the automaton does in fact model the specification.

The method was originally developed by two teams working independently [Clarke and Emerson, 1981; Queille and Sifakis, 1982] and the term 'Model checking' was coined by Clarke and Emerson [Clarke et al., 2000]. Model checking was used successfully to verify protocols and hardware systems, and has since evolved into a useful technique for verifying a broad range of system classes. Model checking has discovered subtle non-trivial errors in real software and hardware systems and is gaining increasing acceptance in industrial settings [Baier and Katoen, 2008]. For example, [Clarke et al., 2000] cites the success of [Clarke et al., 1995] in finding previously unknown errors in the IEEE Futurebus+ cache coherency protocol.

The model checking process can be broken down into three distinct sub-processes, each of which is explained in some detail below. The three sub-processes are the construction of a model, the expression of the specification, and the verification or model checking process. The majority of this section is based upon the material found in the excellent model checking reference [Baier and Katoen, 2008], a modern reference with specific coverage of software model checking. Parts are also derived from the

prominent textbook in the field "Model Checking" by Clarke et al [Clarke et al., 2000]. Both texts are fine references for the major concepts of model checking, with [Clarke et al., 2000] cited in all related works.

4.2 BUILDING THE MODEL

The first stage of model checking is to construct a model of the system to be checked. This stage consists of expressing the system in a language from which a state-based model can be derived [Baier and Katoen, 2008]. A state-based model of the system is a digraph in which nodes are global unique states of the system, and edges are the events that transition between states. There are numerous ways to describe the graph structure. [Clarke et al., 2000] refer to a *Kripke structure*, a simple structure capturing a set of states, transitions between those states and an encoding of information about those states. [Baier and Katoen, 2008] and [Magee and Kramer, 2006] refer to *transition systems* that are similar to Kripke structures, but expressed slightly differently. The author prefers to use the transition system approach which appears more intuitive.

There are numerous languages for specifying the behaviour of a model from which a state-based model can derived. The language used is dependant on which model checking tool one is using. The SPIN model checker [Holzmann, 1997, 2004; Baier and Katoen, 2008] for example uses the language Promela (short for process metalanguage) to specify a model of a system. The Java PathFinder (JPF) model checker [Visser et al., 2003] currently allows model checking on Java bytecode directly, automatically generating a transition system over Java bytecode. In this example, the language for expressing the model is any language that can be compiled to Java bytecode, such as Java or Python. In the initial version [Havelund, 1999], JPF translated Java programs into another language, Promela, which can then be checked using the SPIN model checker. Model checking tools exist for a variety of mainstream industry languages, including the .NET platform [Aan de Brugh et al., 2009] and C/C++ [Clarke et al., 2004].

The generation of the transition system is typically done during the verification phase (discussed later), once a model has been expressed in an appropriate language and a specification has been obtained. However, I will discuss the concept of a transition system here in order to give the specification section some context.

4.2.1 *Transition Systems*

To model the execution of a system, a transition system [Baier and Katoen, 2008] is constructed. A transition system is made up of states and transitions between those states. A state encodes some information about the system [Baier and Katoen, 2008]. For example, one may encode a binary variable to encode the current phase of a traffic light in a traffic light system. A state only need encode information relevant to the property one is verifying. For instance, the current state of the weather may be irrelevant in verifying properties of a traffic light system. The removal of unnecessary information relevant to the specification being checked is a form of *abstraction* [Clarke et al., 2000; Baier and Katoen, 2008].

Definition 1. Transition system, definition from [Baier and Katoen, 2008].

A *transition system* TS is a tuple $(S, Act, \longrightarrow, I, AP, L)$ where

- S is a set of states.
- Act is a set of actions.
- $\longrightarrow \subseteq S \times Act \times S$ is a transition relation.
- $I \subseteq S$ is a set of initial states.
- AP is a set of atomic propositions. and
- $L : S \rightarrow 2^{AP}$ is a labelling function.

A transition, or edge in the system, takes the system from one state to another. The transition relation $\longrightarrow$ describes how the system behaves over time, linking a state s to a set of successor states $P \subseteq S$ and associating an action with each possible transition. A transition from a state s to s' is written $s \xrightarrow{\alpha} s'$, where $\alpha \in Act$. A state s can be a successor state of itself.

For example, figure 10 shows the transition system of a model traffic light. In this transition system, $s_0, s_1 \in S$ and $a_0, a_1 \in Act$. The transition relation is made up of $s_0 \xrightarrow{a_1} s_1 \in \longrightarrow$ and $s_1 \xrightarrow{a_0} s_0 \in \longrightarrow$. For the purposes of this example, s_0 is the initial state $(I = s_0)$. A model is deterministic if there is one initial state, and only one transition is possible from all $s \in S$. By this rule, the traffic light transition system in figure 10 is deterministic. Figure 11 shows a nondeterministic transition system that models a simple Dining Philosopher system. Act in this case consists of "Pick-up left fork", "Pick-up right fork" and "Put down both forks". The transition relation derived from this system is large, and is summarised by the transition system in Figure 11.

Figure 10: Traffic light transition system, based on the traffic light example found in [Baier and Katoen, 2008]

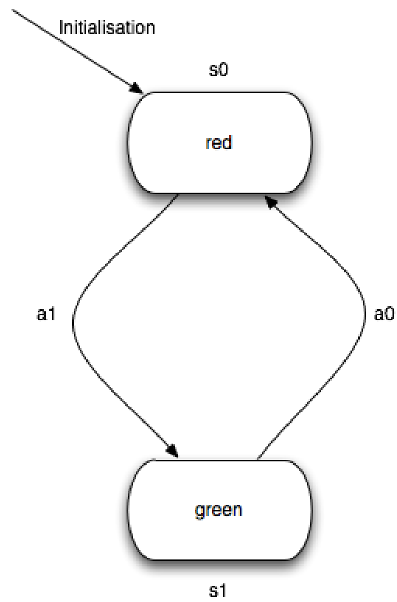
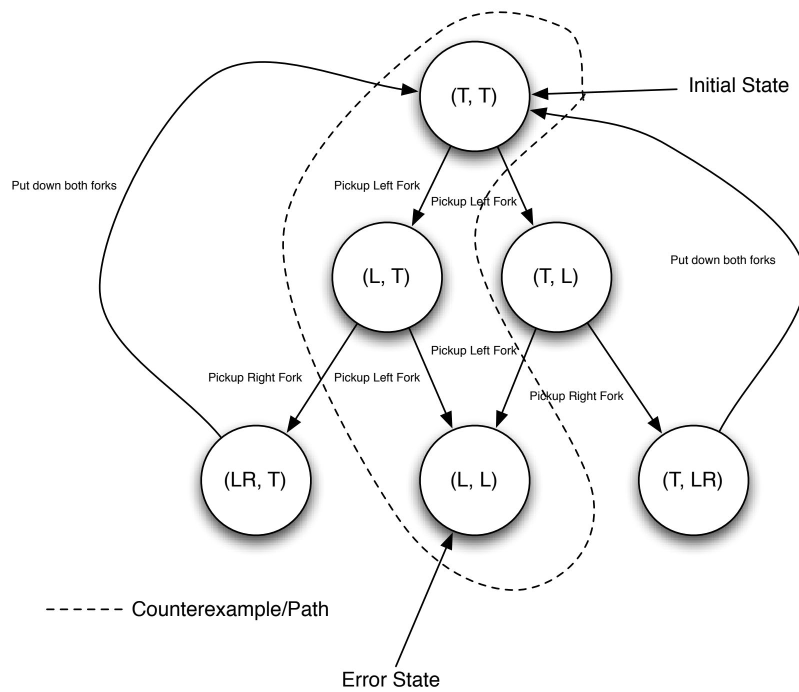


Figure 11: Concurrent Dining Philosophers system with two Philosophers



AP is a set of atomic propositions used to encode information about states in the model. The labelling function L maps a state $s \in S$ to the set of atomic propositions that are true in s. For example, when modelling a traffic light system, the propositions $AP = \{red, green\}$ are adequate for expressing all the possible states. L maps states to sets of properties in AP. In the traffic light model, $s_0 \rightarrow red \in L$ and $s_1 \rightarrow green \in L$. In the Dining Philosophers model, the L label means a philosopher has picked up the left fork, R means the right fork has been locked, and T means the philosopher is thinking. Each state is made up of the conjunction of two philosophers states.

4.2.2 *Paths*

A *path fragment* ρ through a transition system from states s_0 to s_n is an finite alternating sequence of states and actions starting with state s_0 and ending with s_n. A path fragment can be either finite or infinite. Infinite paths are common in systems that do not terminate, such as the traffic light example above.

$$\rho = s_0 a_1 s_1 a_2 ... a_n s_n \text{ such that } s_i \xrightarrow{\alpha_{i+1}} s_{i+1} \text{ for all } 0 \leqslant i < n$$

If a path fragment ρ contains an initial state, then ρ is said to be *initial* [Baier and Katoen, 2008]. If ρ ends with a terminal state or is an infinite path, then ρ is said to be *maximal* [Baier and Katoen, 2008]. A *path* of a modelled system refers to a path fragment that starts in an initial state, and ends in either a terminal state or an infinite path [Baier and Katoen, 2008]. In other words, a path is a path fragment that is both initial and maximal. A path represents a possible execution of the system, and the set of all paths in a transition system is sometimes known as the *execution space*. In a deterministic transition system, there is only one path through the transition system. In a nondeterministic transition system, there are two or more paths through the transition system

A model can have a set of terminating states T i.e. states from which no transition is possible. Path fragments can be expressed solely in terms of either the states visited (either explicitly or via the L mapping) or the transitions executed. For example, $s_0 s_1$ and $a_1 a_2$ are both examples of path fragments, where $s_i \in S$ and $a_i \in AP$. In the traffic light example, $s_0 a_1 s_1$ is a path fragment that is initial and $(s_0 s_1)^\omega$ is an execution and therefore both initial and maximal. The notation ρ^ω denotes a infinite path that consists of ρ repeated infinitely.

4.2.3 *Reachable States*

A state s is said to be reachable if there is an initial path that includes s. [Baier and Katoen, 2008] defines *Reach(TS)* as all the

states that are reachable from all the initial states in a transition system TS. In the traffic light example, *Reach(TS)* $= S$, that is, all states in the traffic light system are reachable from the initial state via the relation $\longrightarrow$. $\mathtt{Reach}(s)$ where $s \in S$ can also be defined as all the states reachable from state s via the transition relation.

4.2.4 *Expressing Models*

A model can be expressed in a number of ways. For instance, process calculi such as CSP [Hoare and Hoare, 1985; Roscoe et al., 1998] can be used to express a model from which a transition system can be extracted. Promela [Holzmann, 1997] is a prominent model specification language, used as the input language for the model checker SPIN [Holzmann, 1997]. JPF [Visser et al., 2003] has shown that the Java programming language can be used to specify models, automatically extracting transition systems from Java bytecode. Model checking tools also exist for the .NET platform [Aan de Brugh et al., 2009] and C/C++ [Clarke et al., 2004].

4.3 SPECIFICATION OF PROPERTIES

Once a model of the system has been constructed, or extracted from mainstream industry languages, the properties of the model to be verified must be expressed. Specifications are commonly formalised in a temporal logic [Baier and Katoen, 2008; Clarke et al., 2000], a class of languages that can describe paths (sequences of states) in a model. The use of temporal logic in automated model checking was introduced by [Clarke et al., 2000; Clarke and Emerson, 1981; Emerson, 1981]. When temporal logic is used to reason about the system model, the logic can refer to either the states along a path, the actions along a path, or both. A specification consists of the paths, or properties, a transition system should exhibit. A specification language expresses sets of paths or properties.

Properties can have one of two forms. The first, *safety properties*, specify conditions that must be satisfied in all states or paths of a model. An example of a safety property is an invariant on all states and paths in a model. For instance, in the sequential traffic light model, we may wish to check that both lights never enter the green state in any part of the model. The other form is *liveness properties*, which are used to specify properties that must be satisfied eventually by a model. An example of a liveness property in the sequential traffic light model is a "state where one light is red and one light is green must be reachable".

[Baier and Katoen, 2008] uses the following perception to define the two. Safety properties generally state that "a bad event *b*

must never happen", and liveness properties generally state that "a good event g must happen infinitely often". For an event e to happen infinitely often, it must be the case that event e does not *not happen for an infinite amount of time* [Clarke et al., 2000; Baier and Katoen, 2008]. An event e in this context is a particular path fragment ρ, i.e. a particular sequence of states are visited, or a particular sequence of transitions are executed. Safety properties are violated by finite paths to states that violate the given safety property. Liveness properties, however, are violated by the presence of infinite paths that do not satisfy the given liveness property [Baier and Katoen, 2008].

4.3.1 *Temporal Logics*

The prevalent temporal logics used in model checking extend traditional proposition logic with temporal modalities [Clarke et al., 2000]. Temporal logics allow the expression of the order of events, which is useful when describing properties in a state-space model. There are a number of different temporal logics available, coupled with efficient algorithms for processing them with respect to a model [Clarke et al., 2000; Baier and Katoen, 2008].

Most temporal logics allow the expression of the following two modalities [Baier and Katoen, 2008; Clarke et al., 2000]. The first, $\Diamond$, means eventually in the future from the current time. An example usage of this operator is $\Diamond a$, meaning that the proposition a will eventually be satisfied by a state reachable from the "current" state. The second, $\Box$, means always in the future from now onwards. $\Box a$ expresses that a holds true in every state reachable from the current state, including the current state.

The types of temporal logics available to a practitioner depend on the model checking tool in use. Most model checking tools allow for the checking of invariant properties as the algorithms for checking invariant properties are simple. More sophisticated and established tools, such as the SPIN model checker, allow the user to specify properties in a temporal logic known as Linear Temporal Logic (LTL) [Clarke et al., 2000]. Alternative temporal logics exist, including Computation Tree Logic (CTL) [Clarke and Emerson, 1981; Queille and Sifakis, 1982], Extended Computation Tree Logic (CTL*) [Baier and Katoen, 2008].

For a full description of how temporal logics can be used to express the behaviour of concurrent systems, I refer you to both [Baier and Katoen, 2008] and [Clarke et al., 2000].

4.4 VERIFICATION

Once a model has been constructed and the specification realised, the chosen model checking tool performs a search on the model for violations of the specification. The problem is to verify that $TS \models \Phi$, that is all states and paths in the transition system TS satisfy the specification Φ [Clarke et al., 2000; Baier and Katoen, 2008]. Traditional model checking algorithms focus on completeness and correctness using exhaustive algorithms thus guaranteeing that a model satisfies a given property.

The verification process is largely automatic and has one of three outcomes [Clarke et al., 2000]. The tool can return *yes*, indicating that the specification is satisfied by the model. The tool can also return *no*, indicating that the model does not satisfy the property. When the tool returns *no*, it can also return a *counterexample*, a path in the model that violates the property.

The last result a model checking tool can return is an error indicating that the tool ran out of resources whilst traversing the model. This is typically the result of the underlying transition system being too large to fit into memory. To check safety properties, an exhaustive search of the model is required to be sure that no path violates the property. For liveness properties, it is sufficient to search the model for a single path that satisfies the property.

If a counterexample is returned, this can then be used to debug either the specification or the model, depending on where the error is situated. Whilst one can be assured that the modelled system satisfies the specification, it is still possible for modelling and specification errors. Once the error has been corrected the verification stage is executed again, leading to an iterative process of verification and debugging until a fixed point is reached. If a resource is exhausted, the model must be refined in order to fit the full model into memory. This typically involves removing irrelevant information, either explicitly or by abstraction.

The algorithm used for the checking the model against a specification depends on what type of property is being checked, and how it is expressed. The complexity of the algorithm used generally increases with the expressiveness of the language used to specify the property.

4.4.1 *Checking Invariant Properties*

A invariant is a property of the form $\Box\Phi$ [Baier and Katoen, 2008]. A property of this form describes that all states that are reachable from the initial state(s) of the transition system must satisfy the condition Φ. Φ is a formula over the atomic propositions in AP, and can only refer to states, not paths or path fragments [Baier

and Katoen, 2008]. A property of this form requires an exhaustive traversal of all reachable states in a transition system. Exhaustive traversal of a finite transition system is typically performed using a breadth or depth-first search (BFS or DFS respectively), and each state s visited is checked for $s \models \Phi$ [Baier and Katoen, 2008]. If a state is found where $s \not\models \Phi$, then the algorithm returns false, indicating that $TS \not\models \Phi$. The algorithm can also return a counterexample, a path fragment through the transition system that begins in an initial state and ends in the state that violated Φ. If the algorithm exhaustively checks all states and finds no violation of Φ, then the algorithm returns true, indicating that $TS \models \Phi$.

4.4.2 *Checking Safety Properties*

Invariant properties are a subset of safety properties. Verifying safety properties that place requirements on paths in a transition system requires a more complex procedure. Rather than searching for a reachable state that violates the given safety property, one must search the transition system for a finite path from the initial state that violates the safety property. A path is this form is known as a *bad prefix* [Baier and Katoen, 2008].

The goal of checking a model for violations of a safety property Φ_{safe} is to exhaustively search the model for a path in a transition system that is in the set of all bad prefixes defined by the negation of the given safety property [Baier and Katoen, 2008]. If a path $\pi \in BadPrefixesTS$ is found, then π is returned along with false, indicating that transition system $TS \not\models \Phi_{safe}$. If no path can be found that is in the set of bad prefixes after an exhaustive search, then true is returned indicating that $TS \models \Phi_{safe}$. There is also a preference on finding shorter bad prefixes, as a short path is easier to debug. The exhaustive checking of a safety property can also be performed using a DFS or BFS [Baier and Katoen, 2008].

4.4.3 *Checking Liveness Properties*

Liveness properties express that "something good will happen". In order for a transition system to satisfy a liveness property Φ, the transition system must be checked for the absence of a path that does not satisfy Φ. This is equivalent to finding a finite path fragment that satisfies $\neg\Phi$ and then finding a cycle in the transition system that includes the state that satisfies $\neg\Phi$ [Baier and Katoen, 2008]. Checking a liveness property can be achieved by repeatedly executing a DFS to find a path that satisfies $\neg\Phi$, and then executing another DFS from that state to discover a cycle.

4.4.4 *Complete Model Checking Mechanisms*

Complete, or global, model checking mechanisms refer to the verification methods that exhaustively traverse a transition system to ensure conformance with a given specification. Different mechanisms are required depending on the particular property being verified. [Clarke et al., 2000; Merz, 2001; Baier and Katoen, 2008] excellent resources for these types of mechanisms. For the sake of brevity, they are not included here as they require in-depth explanations.

4.4.5 *On-the-fly Model Checking*

The verification stage can happen off-line or on-the-fly (OTF). When performing off-line verification, the entire transition system is generated and then a graph traversal algorithm is applied to the generated system. This is known as *global model checking* by [Baier and Katoen, 2008]. On-the-fly model checking, on the other hand, generates the model as part of the graph traversal algorithm, effectively combining the transition system generation stage and the traversal stage. OTF model checking has the potential advantage of revealing counterexamples without having to generate and store the entire transition system. This approach is taken by the Java PathFinder tool, as well as other software model checkers.

4.4.6 *Guided Model Checking*

When performing the verification stage of the model checking process, it may be preferable to discover errors in the model as quickly as possible. Discovering errors quickly can help reduce the verification and debugging cycle described earlier. When using on-the-fly model checking, finding the error sooner rather than later results in less of the model being generated before an error is found. This may well be the only option when it comes to checking large systems that are impossible to fit into memory.

Exhaustive DFS as described above traverses the state-space blindly, expanding the next transition in a system in a fixed order or random fashion. An improvement can be made by attempting to choose transitions that will more likely lead to violations of the specification. This is the essence of guided model checking [Yang and Dill, 1998], also referred to in the literature as directed model checking [Edelkamp et al., 2001a]. Guided model checkers use heuristic information [Russell et al., 1995], gained from the current state, path and domain specific information, in order to expand parts of the state-space that are more likely to contain errors [Yang and Dill, 1998]. The guided model check-

ing algorithms are referred to as the class of best-first search algorithms in [Russell et al., 1995]. A heuristic in this circumstance ranks the possible successor states to expand based on how "close" they are to a state that violates a property.

A number of algorithms can use heuristics to guide the traversal of the state-space toward errors. The most basic is greedy best-first search [Russell et al., 1995], which expands the state ranked highest by the heuristic first. This is similar to DFS, but the depth-first expansion is chosen according to the heuristic. More sophisticated mechanisms exist such as A* search [Russell et al., 1995], which aims to return the shortest path to an error state.

4.4.7 *Strengths and Limitations of Model Checking*

Model Checking has a number of advantages over techniques discussed in Chapter 3 when testing/verifying concurrent software. Once a system has been described and a specification sourced, error detection or complete proof can be achieved through automatic methods given enough resources. If a counterexample is found, the information gained is very useful in debugging as the precise circumstances of an error are returned. Specifications can be rich, and can even express temporal aspects through the use of the temporal logics highlighted earlier. The technique is quite general, being able to operate in circumstances where a state space can be extracted from some description of a system, including mainstream languages such as Java.

Model checking does have some limitations. Whilst one can be sure that a model satisfies some specification after an exhaustive check, there is still the issue of whether the model/specification accurately reflect the intentions of practitioners. This issue is alleviated to some respect when extracting models automatically from mainstream languages and checking for generic problems like deadlock. Model checking has some issues with dense data types [Baier and Katoen, 2008], such as real values, which can limit the applicability of the technique in practical circumstances. Some difficulty arises when systems comprising of components described by many different languages, as the model checking apparatus typically does not support this kind of analysis.

The most prominent issue with model checking is what is known as the state-space explosion problem [Baier and Katoen, 2008; Clarke et al., 2000]. It is typically the case that as the size of the description of a system increases, or indeed the number of concurrent components of a system increases, then the size of the state space to be checked increases exponentially. This problem makes some systems impossible to check exhaustively, and

can cause issues for guided and metaheuristic model checking algorithms.

4.4.8 *Metaheuristic Model Checking*

Along with guided model checking techniques, metaheuristic model checking attacks the state-space explosion issue by focusing the search of the state space more likely to reveal an error. By using the model checking framework, comprising of explicit state space exploration and a rich specification language, a metaheuristic search technique can be used to test concurrent systems for a variety of faults. These faults include anything that can be specified in a temporal logic, including deadlock. Since model checkers for mainstream languages such as Java and the .NET platform exist, practitioners can potentially exploit the benefits of model checking in practical/industrial scenarios. However, the same limitations of model checking apply.

A detailed overview of metaheuristic search algorithms is given in Chapter 5. A critical overview of metaheuristic methods applied to model checking is given in Chapter 6.

4.5 SUMMARY

In this chapter, I have described how model checking techniques can be used to verify properties of concurrent systems. I have described how exhaustive techniques verify a system conforms to a specification, and how the state-space explosion problem limits the applicability of an exhaustive proof. When faced with this scenario, I have described how guided and metaheuristic model checking techniques can be used to show the presence of an error rather than the absence of one. This approach is effectively software testing, exploring parts of the program state space that are more likely to contain an error. In this scenario, model checking tools are exploited for their ability to examine the state space explicitly, as well as being able to check potentially complex specifications, rather than their exhaustive proof capabilities.

In the following two chapters, I will give a detailed overview of metaheuristic search techniques, and show how they have been applied to the problem of searching transition systems. I will describe the state of the art, and highlights gaps where I hope to bring novelty to the field.

5 METAHEURISTIC SEARCH

5.1 INTRODUCTION

A solution space is the set of all solutions to some problem. Some members of the solution space may be a "better" solution, by some measure of *fitness* or *evaluation*, than other members. A solution space may be large in size, so large in fact that enumeration of the entire solution space is unfeasible. In situations where enumeration of the solution space is unfeasible, one can sample the solution space in the hopes of finding or getting close to the best solution.

An example problem is the $\mathrm{MaxOnes}(v)$ problem, also known as OneMax [Eiben and Smith, 2003] or the bit counting problem [Chen et al., 2002]. $\mathrm{MaxOnes}(v)$ takes a vector v where each vector element v_i is either 0 or 1 and returns $\sum_{i=0}^{n} v_i$ where n is the length of vector v. For example, $\mathrm{MaxOnes}(100) = 1$ and $\mathrm{MaxOnes}(111) = 3$. The input vector can be referred to as a string of bits, or a bit string. In this problem, the optimal solution is a vector w such that all vector elements w_i are of value 1, i.e. the value returned by $\mathrm{MaxOnes}(w)$ is the length of vector w. The $\mathrm{MaxOnes}$ problem has a solution space of size 2^n where n is the length of the input vector, which becomes very large for large n. $\mathrm{MaxOnes}$ in this case is the measure of fitness, sometimes referred to as the *fitness function* [Eiben and Smith, 2003], *objective function* [Russell et al., 1995] or *evaluation function* [Eiben and Smith, 2003].

One possible method of sampling the solution space is repeated uniform random sampling, storing the best solution found so far. In the $\mathrm{MaxOnes}$ example, this would amount to repeatedly generating a vector w, where each vector element w_i is either 0 or 1 with equal probability. $\mathrm{MaxOnes}(w)$ is evaluated, and if the result is better than the result for the best vector so far b, then $b := w$. No information from the history of solutions generated is used. If this method were to run for an infinite amount of time, this mechanism is guaranteed to stumble upon the fittest solution. However, in the majority of cases, one does not have an infinite amount of time, and would like a solution in some finite time frame. In this case, a more refined sampling strategy is required.

5.1.1 *Metaheuristic Search*

Metaheuristic search techniques represent one possible implementation of such a strategy. Metaheuristic search techniques utilise a heuristic to help solve the problem of finding a solution in some solution space [Russell et al., 1995]. A heuristic provides information on how far a particular solution is from some desired optimum [Russell et al., 1995]. Metaheuristic search techniques can potentially use information from the history of a search, i.e. the solutions that have already been checked. Metaheuristic search techniques aim to sample the solution space effectively in order to find a solution with less computational cost than explicit enumeration or random sampling.

The metaheuristic search techniques reviewed in this chapter take two major inputs. The first is some representation of the solution space. This is an encoding of the solution space that can be processed by machines and is typically easily manipulated. For example, the $MaxOnes$ problem solution space is a bit string of length n, and an encoding of this solution space can also be a bit string of length n. Another possible encoding is to use a single integer i which encodes the number of 1s in the bit string. The second input is the evaluation function described above. Metaheuristic search techniques use this information, and potentially more, to effectively sample the encoded solution space.

5.2 LOCAL SEARCH

Local search techniques are a class of metaheuristic search techniques. Local search mechanisms store and operate on a single point x in the search space [Reeves, 1993]. x represents the *current solution* of the current *iteration* [Russell et al., 1995]. The point x is a particular instance of an encoding [Eiben and Smith, 2003] of the solution space. A local search algorithm transitions through a series of time steps, or iterations. At each time step, a non-empty set of candidate solutions C is generated. The members of C are some function of the current solution x. Then, according to a selection policy, x is assigned to some $y \in C \cup (x)$. This can be described as *accepting* a candidate solution in $C \cup (x)$. The local search algorithm stores the best solution b found so far, updating b at each time step if a better solution is found. Initially, x is set to some candidate solution in the solution space. x can be seeded with a best known solution, or a randomly selected solution. The algorithm terminates when some termination criterion are met, such as the optimum solution being found or after some amount of iterations.

The generation of the candidate solution set C amounts to a sampling of the *neighbourhood* of the current solution x [Reeves,

1993]. The neighbourhood of the current solution x is a function of x and some manipulation operation. The neighbourhood is the set of all possible results from applying the manipulation function to the current candidate solution. Therefore, the candidate solution set C is a subset of the neighbourhood of x. In the $MaxOnes$ example, a possible manipulation function could be as simple as flipping a randomly selected bit x_i in the current solution vector x. An example of a bad manipulation operator for the $MaxOnes$ problem is one that swaps values of a randomly selected bit x_i with another randomly selected bit x_j. This manipulation operator has no effect on result of the fitness function, since the number of 1s in vector x has not changed.

There are many variations on local search algorithms. A local search mechanism is typically defined by how the candidate set C is generated, and what selection mechanism is used. One of the simple examples is the random walk algorithm. The candidate set C at each time step only has one member y. The member is obtained by sampling the neighbourhood as defined above. The selection policy of the random walk algorithm is to always choose the generated candidate solution y, even if y is less fit than current solution x. Given an infinite amount of time, random walk will find the optimum solution [Russell et al., 1995].

A possible improvement to the random walk algorithm is to alter the selection policy to only accept the candidate solution y if the fitness of y is greater than the fitness of x according to the fitness function. With this selection policy, the current solution x can only either stay the same or improve over time. This particular variant of local search is known as hill climbing [Russell et al., 1995; Reeves, 1993].

5.2.1 *Convergence and Optima*

Hill climbing highlights one of the potential pitfalls of local search and search in general. Hill climbing accepts better candidate solutions only, i.e. the algorithm only *moves* to better parts of the solution space. A *move* in local search is when the current solution x is changed. If the algorithm only moves to better solutions in the solution space, then the algorithm risks getting stuck in a *local optima*, sometimes called *local maxima* [Russell et al., 1995].

A solution space can have one or more global optima, i.e. best solutions in the solution space. An interesting (i.e. non-trivial) solution space will have one or more local optima. A local optimum is defined as a point in the search space that is not a global optima and whose neighbourhood consists of less fit solutions only [Russell et al., 1995]. Once a hill climbing search has moved to a point in the search space that is locally optimal, the search will not progress to any better solutions. The hill

climbing algorithm is trapped in the local optimum. When a search algorithm cannot make any progress, i.e. it is not possible for the algorithm to find a better solution than the current point, then the algorithm is said to have *converged* [Reeves, 1993]. Ideally, one would like for a search algorithm to converge on a global optimum.

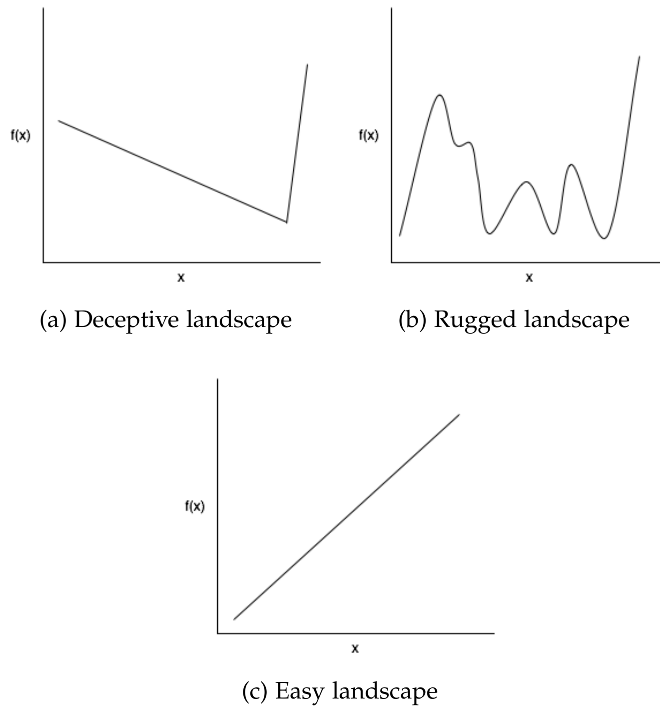
5.2.2 *Landscapes*

The fitness values of members in a solution space can be plotted against the aspects, or parameters, that constitute the solution. A visualisation of the solution space of this kind is known in the literature as a *fitness landscape* [Langdon and Poli, 2002; Wright, 1932]. For example, the fitness of $MaxOnes$ can be plotted on a 2D graph, where the x-axis represents the number of bits set to 1, and the y-axis represents the respective fitness of x. This landscape is depicted in figure 12c. The landscapes resemble mountainous regions of land, in which the height of a particular point is analogous with a solutions fitness [Langdon and Poli, 2002]. Local search algorithms can be seen as sending an agent to wander a fitness landscape with visibility restricted to the neighbourhood, with the mission of finding the highest peak in the landscape. This leads to the analogies of walking and hill climbing.

Characterisations can be made of a fitness landscape, and analysis of a landscape can aid in the choosing of search technique and the setting of any parameters. Examination of a fitness landscape can yield clues as to how hard a problem is for a particular search technique [Langdon and Poli, 2002]. The $MaxOnes$ landscape is an example of a fairly trivial landscape. From any point in the solution space, except for the global optimum, the only possible way of improving the solution is to increase the tally of 1s in the solution. However, interesting solution spaces rarely exhibit this property.

Interesting solution spaces are likely to be *deceptive* in nature [Whitley, 1991]. An example of a fitness landscape of this kind is depicted in figure 12a. In this landscape, there is one local optima and one global optima. The local optima is "far away" in terms of features from the global optima but is of high fitness. The density of fitter solutions is skewed toward the local optima, which will cause problems for many search techniques.

Rugged landscapes are landscapes that contain many local optima. Figure 12b depicts a rugged landscape, and figure 12c shows a landscape that is not classed as rugged. When searching over a landscape that has many local optima, the chance of converging on a local optimum increases over a landscape that contains fewer local optimum.



(a) Deceptive landscape (b) Rugged landscape

(c) Easy landscape

Figure 12: Example landscapes

5.2.3 *Fitness Function*

From the description of fitness landscapes above, one can see that the difficulty of a problem for a search technique is defined by the fitness function. The choice of fitness function for use in a metaheuristic search technique can be crucial. A simple example to demonstrate this is the use of Gray coding when operating over a binary string. When one uses a typical binary encoding scheme for integers, it is not always possible to achieve a similar phenotype (integer) by making a small change to the genotype (binary string). However, when using a Gray coding scheme, it is always possible to reach a similar integer (the preceding or successive integer) by changing a single bit in the bit string. This yields a higher likely hood of similar phenotypes resulting from mutations in the genotype.

5.2.4 *Simulated Annealing*

In order to increase the chances of converging on the global optimum, a more sophisticated search mechanism is needed. Either the neighbourhood sampling mechanism or the selection policy must be improved in order to avoid getting trapped in local optima. If one were to focus solely on the selection policy, the policy must allow the possibility of accepting moves that

aid in escaping from local optima. One such selection policy forms part of the implementation of a technique called *simulated annealing* [Kirkpatric et al., 1983]. Simulated annealing is based on observations of the metallurgic process of annealing. The process of annealing strengthens a metal by heating it up, and allowing it to cool slowly. Cooling the metal slowly allows for the structures within the metal to form a stronger overall structure, as opposed to quick cooling which is more likely to result in the overall structure being in a weaker state [Russell et al., 1995].

A similar idea is employed in simulated annealing. The selection policy employs the notion of a temperature. The temperature represents the probability of accepting a candidate solution if it is a move to a less fit solution in the search/solution space. The higher the temperature, the higher the probability that a worsening move is accepted. Also, the more sacrificial the move is, the less likely it is to be accepted. The temperature is gradually reduced according to a cooling schedule over the course of the algorithm until it reaches some minimum level. As the temperature approaches the minimum level, the algorithm degrades into hill climbing as there is zero probability of the algorithm accepting a worsening move. At this time, it is assumed a good portion of the search space has been sampled, and that a good solution has been found. If this is not the case, the temperature is raised and the algorithm continues to sample the solution space.

5.3 OTHER LOCAL SEARCH MECHANISMS

Many variations on traditional local search algorithms can be devised in order to more effectively sample the solution space. For instance, Tabu search [Glover, 1986; Reeves, 1993] exploits information from the history of moves gathered during the course of a search. Tabu search maintains a tabu list that contains recently visited areas or states, which the algorithm is prohibited from returning to. Restricted parts of the search space are gradually forgotten over a number of iterations. The selection policy of tabu search denies any move that is on the tabu list, but can be overridden by aspiration criteria [Reeves, 1993]. For example, one does not wish to deny a move to a solution that is the best so far even if the move is in the tabu list.

5.4 POPULATION-BASED SEARCH

The above sections discuss algorithms that operate on a single point in the solution space. In this section, metaheuristic search techniques that operate on a set of points in the solution space, commonly referred to as the *population*, shall be discussed. By operating on multiple points in the search space, one can exploit

the additional information provided by storing multiple points in the solution space in the hope of converging on global optima using less computational resources. In the fitness landscape analogy, this is equivalent to having multiple searching agents on the landscape, co-ordinating in order to efficiently head toward the global peak [Langdon and Poli, 2002].

5.4.1 *Evolutionary Algorithms*

Evolutionary algorithms (EAs) are one class of stochastic population-based metaheuristic search techniques. EAs attempt to exploit the theory of Darwinian evolution [Darwin, 1860] and natural selection in order to *evolve* a population of solutions over a series of generations. Simulated *selection pressure* and reproductive mechanisms are employed to guide the population toward a set of solutions to some problem. The reproductive mechanisms are used by EAs as a way of exploiting the additional information in a population of solutions to help guide the search [Eiben and Smith, 2003].

Evolutionary algorithms operate on a population of solutions. Initially, the population can be randomly generated or seeded with previously best known solutions. EAs step through a number of iterations, and the population at iteration i is referred to as generation i. Each generation of solutions is evaluated according to a fitness function, and those values are linked to each solution. The evaluation by fitness function is analogous to a creatures ability in the natural world to obtain resources and survive [Eiben and Smith, 2003].

To generate the successor generation $i+1$, solutions are selected from generation i. This selection process is typically biased toward the fitter individuals in the population based on their fitness function value, mimicking competition for survival (selection pressure) in the Darwinian view of the natural world. The selected individuals from generation i are described as seeding generation $i+1$ [Eiben and Smith, 2003]. An EA then generates new candidate solutions by combining the selected solutions using reproductive operators, sometimes known as recombination operators, analogous to sexual reproduction in the natural world. The hope is that the “children” of the recombination process will exhibit the features that made the selected “parents” evaluate highly in terms of fitness. Parent solutions are recombined enough times to yield an equally sized successor generation. Mutation in the natural world can also be modelled, and is typically applied after the recombination stage. A mutation operator is applied to members of generation $i+1$ typically with low and independent probability. A mutation operator takes a single solution and outputs a similar but different solution.

Figure 13: Algorithm of a vanilla EA

```
1      //Initial population with random or seeded solutions
2      INITIALISE(population);
3      //While the termination condition is not satisfied
4      while (NOT(TERMINATION_CONDITION)) {
5              //Evaluate all solutions in population
6              EVALUATE(population);
7              //Select parents from previous generation
8              parents = SELECT(population);
9              //Produce children from parents using
                   recombination operators
10             new_pop = RECOMBINE(parents);
11             //Apply low probability mutation to children
12             MUTATE(new_pop);
13             //New solutions become new generation
14             population = new_pop;
15     }
```

This process of selection, recombination and mutation is applied repetitively. The use of artificial selection pressure and recombination operators act as a guide to increasing the average fitness of the population [Eiben and Smith, 2003]. The recombination and mutation operators aid in maintaining diversity within a population, in order to increase the chance of novel solutions being discovered [Eiben and Smith, 2003]. The process is repeated until some termination criterion is satisfied. The termination criterion can be that an optimal solution is found, or that a certain number of solutions have been evaluated. The process is depicted in figure 13 showing an algorithmic description.

Variations can exist on the pseudo-code above. For instance, when constructing the next generation, one can use a *steady-state* approach where a finite number of solutions in the previous generation are replaced by newly constructed solutions. This is opposed to the *generational* approach above, where the entire population is replaced by a newly constructed set of solutions. Despite some of the variations, the underlying concept of selection and recombination is common throughout all instances of EAs.

5.4.1.1 *Prominent Examples of EAs*

One of the most of prominent instances of an EA is the Genetic Algorithm (GA). Widely attributed to [Holland, 1975], GAs are the archetypical implementation of an EA for bit string solution spaces, or indeed vectors of most data types. The MaxOnes problem outlined above has a bit string solution space, meaning that popular GA methods and implementations can be applied.

Vector recombination and mutation operators are defined and have been well studied for various problem types [Goldberg, 1989]. GAs have been shown empirically to be effective in many problem domains [Goldberg, 1989], and theoretical work exists attempting to explain why GAs perform well [Holland, 1975]. In the next chapter, I will describe how GAs have been applied to the transition system search problem.

Other instances of EAs include Genetic Programming [Koza, 1992], that selectively samples the solution space of computer programs in order to find solutions to programming problems, and Grammatical Evolution [Ryan et al., 1998] which searches the space of solutions that conform to particular grammars.

5.4.2 *Ant colony optimisation*

Ant colony optimisation (ACO) is a stochastic population-based metaheuristic search technique that exploits observations on the path finding ability of ant colonies when foraging for resources [Dorigo and Di Caro, 1999]. ACO operates on solution spaces represented as a graph structure. The nodes of this graph structure are components of a solution. The edges of the graph are possible connections between each component, and each link has an associated cost. There can also be a set of constraints over the nodes and edges in the graph. A solution s in this structure is represented by a sequence of states linked by connections that satisfies any given constraints. s has a cost, defined by the cost of all the links between the states in the sequence s.

ACO simulates a set of agents (Ants) that co-operate to discover a solution. ACO discovers a sequence of states in the graph structure defined above that has minimal cost. Each agent traverses the structure in an effort to discover a minimal costing path through the graph. The ant starts in a particular component, and traverses edges that are feasible with respect to the constraints. The ant does this repeatedly, incrementally building a sequence of states representing a solution. A choice of multiple edges is decided by a probabilistic function, which is a function of the problem constraints, the fitness function and an implicit communication medium between ants.

The ant agents implicitly communicate via the use of pheromones. As an ant traverses an edge, the ant deposits pheromones upon that edge. The more pheromone on an edge, the more likely an ant is to choose to traverse that edge. Pheromone deposits evaporate over time, evaporating more quickly on edges with higher cost. Over time, given enough ant agents navigating the graph structure, short paths to goal states (or good solutions) are reinforced. The implicit self-organisation of the ant agents via pheromone communication classes ACO as a swarm intelligence

optimisation algorithm, where multiple agents cooperate to solve a problem.

A detailed description of ACO can be found in [Dorigo and Di Caro, 1999]. ACO has been found to be effective when solving a variety of problems, including routing/traffic problems [Sim and Sun, 2003], scheduling problems [Merkle et al., 2002] and data mining [Parpinelli et al., 2002]. ACO has shown strength in problems where the goal state changes over time, such as dynamic traffic management [Sim and Sun, 2003].

Another notable swarm intelligence inspired algorithm exists, namely Particle Swarm Optimisation [Kennedy and Eberhart, 1995] (PSO), which draws inspiration from the intelligence movement of flocks of birds and schools of fish. PSO is designed to optimise vectors of real numbers, but can be adapted to optimise discrete problems such as those suited to ACO.

5.5 ESTIMATION OF DISTRIBUTION ALGORITHMS

Estimation of distribution algorithms (EDAs) [Mühlenbein and Paaß, 1996] operate on a probabilistic model of the solution space, whilst maintaining a population of solutions at each iteration. A model of the solution space is constructed from the better solutions in an initial random sample of the solution space. This model is then used to generate a new set of candidate solutions, and the better solutions from the new set are used to improve the model. This process is repeated until some termination criterion has been reached. EDAs are also known as probabilistic model-building genetic algorithms (PMBGAs) [Pelikan et al., 2002]. EDAs can be thought of as a subset of Evolutionary Algorithms, where the recombination and mutation step are implemented by a model building and model sampling step.

5.5.1 *Example EDA*

To describe the function of an EDA, I shall describe one of the simplest implementations. This algorithm is known as the Univariate Marginal Distribution Algorithm (UMDA) Mühlenbein and Paaß [1996]. The algorithm samples solution spaces described by strings of bits and roughly follows the generic EDA algorithm described in Figure 14.

The algorithm holds an initial model of solutions in the solution space. This model is represented by a vector m of probabilities m_i that is a fixed length. Initially, nothing is known about the solution space, so each probability is set to 0.5, and these values are updated as the algorithm progresses. The algorithm builds an initial set of candidate solutions that are bit strings the same length as m. For each bit i in each candidate solution c, a random

Figure 14: Algorithm of a vanilla EDA

```
1        //Initial model with no bias toward any solution
2        INITIALISE(model);
3        //Initial sample of the solution space
4        SAMPLE(sample);
5        //While the termination condition is not satisfied
6        while (NOT(TERMINATION_CONDITION)) {
7                //Evaluate all solutions in the sample
8                EVALUATE(sample);
9                //Select best candidates from sample
10               best = SELECT(sample);
11               //Update model based on best candidates
12               UPDATE(model,best);
13               //Sample solution space with model
14               SAMPLE(sample, model);
15       }
```

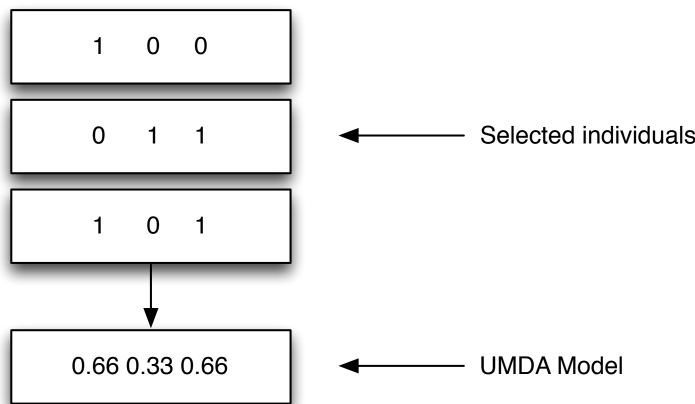


Figure 15: UMDA modelling process

real-valued number r between 0 and 1 is generated. If $r < m_i$ then c_i is set to 1, $r \geqslant m_i$ then c_i is set to 0. Each c_i in the model represents the probability of the ith bit of a candidate solution being a 1.

Once a set of candidate solutions are generated, the candidates are evaluated and the best n candidates are selected from the candidate solution set using standard EA selection operators. Then, each element m_i of the model m is updated by taking the average of all the s_is, where s is a member of the selected best n individuals. For example, the candidate solutions 100, 011 and 101 would result in the model 0.66..., 0.33..., 0.66.... This process is illustrated in Figure 15.

5.5.2 *Types of EDA*

The various forms of EDAs differ from each other in terms of the complexity of the model that they construct [Pelikan et al., 2002]. [Pelikan et al., 2002] gives a good survey of the various PMBGAs and the model constructions they employ. The simplest algorithms assume that genes are independent of one another. The algorithm described above, UMDA, is an example of an EDA that assumes that there are no interactions between the bit variables. Other example algorithms that use a gene independent model are the population-based incremental learning (PBIL) algorithm [Baluja, 1994] and the compact GA (cGA) [Harik et al., 1998].

More complex EDAs exist that assume interactions between genes. Graduating from simple approaches such as UMDA, slightly more complex algorithms assume that simple relationships exist between genes. An example algorithm of this kind is the mutual information-maximising input clustering (MIMIC) algorithm [de Bonet et al., 1997], that assumes a chain distribution between neighbouring genes [Pelikan et al., 2002]. MIMIC assumes that a gene directly influences its immediate neighbour. At the most complex end of the EDA spectrum, the Bayesian Optimisation Algorithm (BOA) [Pelikan et al., 2000] can build arbitrarily complex models using little domain specific information. The model takes the form of a bayesian network between the variables in the solution space. The more complex the model, the more expensive the the algorithm is in terms of computation costs [Pelikan et al., 2002]. There are also a variety of EDAs for modelling real numbered vectors. Additionally, EDA equivalents of Genetic Programming have been developer with some success [Pelikan et al., 2002].

5.5.2.1 *Advantages and Disadvantages of an EDA*

EDAs have been shown to outperform GAs and other evolutionary algorithms on a variety of difficult optimisation problems [Pelikan et al., 2002], as well as being more robust when faced with the deceptive problems described earlier in this chapter. This suggests that a greater success can potentially be achieved using EDAs instead of GAs or ACO for searching transition systems. Any reduction in the number of states explored during the search of a transition system is desirable, as expanding the state space through on-the-fly methods tends to be the most expensive aspect of model checking.

The primary disadvantage of using EDAs is that the model building step for the more complex algorithms, such as BOA, tends to be computationally expensive. For problems with a large number of variables (i.e. a large bit string per solution) the cost of

building a Bayesian network model of the best solutions can be prohibitively high. GAs and other EAs tend to have computationally cheap recombination operators and therefore do not suffer from this problem. I believe, however, that the expense incurred by using EDAs to search will be overwhelmed by the expense of exploring the state space of a transition system

In addition to this, the majority of EDA algorithms operate on bit string solution spaces. Whilst some work has been done applying the model building meme to other solution spaces, such as PIPE Salustowicz and Schmidhuber [1997], it can require a great deal of effort to define probabilistic modelling techniques for richer variable types.

5.6 summary

In this chapter I have introduced metaheuristic search techniques, and have highlighted prominent algorithms within the field. In the next chapter, I will survey the state of the art of applying the techniques discussed in this chapter to the problem of searching transition systems/state spaces. I will then highlight a gap in the research, and outline an algorithm proposal to fill this gap.

6 METAHEURISTIC SEARCH OF TRANSITION SYSTEMS

Making use of model checking frameworks allows us as researchers to check for a wide variety of fault types, assuming that your fault can be expressed in the specification language of a given model checking tool. The property "a system must never deadlock", for example, can be expressed as an invariant that says that in all states of a concurrent transition system, at least one subsystem must be able to progress. As stated earlier, an invariant can only refer to state properties in AP. When safety is of primary concern, the invariant property must be checked by using a systematic traversal of the state-space in order to find states that violate the invariant, and returning a counterexample for debugging purposes. When searching for safety errors, a counterexample consists of a finite path to the error state, known as a bad prefix. Searching for liveness errors in a transition system is a more complicated task than searching for safety violations, as an infinite path that violates a property must be discovered. This involves finding a path that violates the property, and then a cycle to the state at the end of that path.

In order to focus the traversal on states which are more likely to violate the invariant, a guided complete model checking algorithm with a suitable heuristic can be used to find erroneous states sooner without having to expand the entire state-space. Despite initially focusing on areas of the state-space that may violate the property, guided model checking algorithms may run into resource constraints if the transition system of a model is too large to fit into memory. In this situation, it may be desirable to sample the state-space intelligently in order to gain assurance of the correctness of the software. This arguably amounts to a form of testing [Beizer, 1984], showing that an erroneous state exists and providing a counterexample for debugging purposes.

One method of doing this is to use a non-systematic, yet resource efficient algorithm to efficiently sample the state-space. Stochastic metaheuristic search techniques are one possible avenue of interest. Search techniques operate on a solution space. A solution in this case is a path to the state that violated the invariant property. One possible solution space is the execution space, or path space, of the transition system, which can be thought of as the set of all possible paths through a system. The potential is that one can use fewer resources in order to obtain a precise ordering of events that leads to the deadlock of a system. What

follows is a survey of metaheuristic techniques that have been applied in this domain.

6.1 GENETIC ALGORITHMS

[Godefroid and Khurshid, 2004] and [Alba et al., 2008] have both applied GAs to finding paths to invariant violations in finite state transition systems. Both works use GAs to search over the set of all possible executions of a transition system. Both works highlight the huge reduction in computational resources required to detect faults using the GA-based approach. [Godefroid and Khurshid, 2004] implemented their approach on top of the VeriSoft model checker, that builds abstract models over arbitrary programming languages like C and C++. [Alba et al., 2008] implemented their approach on the Java PathFinder model checker that builds abstract models over Java bytecode.

6.1.1 *Solution Encoding*

To encode a path in the transition system, [Godefroid and Khurshid, 2004] use a bit string representation. [Alba et al., 2008] encode a path using a vector of floating point numbers. Both methods constitute the genotype of the respective approaches. [Godefroid and Khurshid, 2004] highlight some of the core issues when encoding paths for use in search algorithms. The first issue is that the number of transitions from each state cannot be known a priori, leading to the requirement of a solution encoding that can handle arbitrary numbers of transitions from any state. The second issue is that the encoding will have to handle paths of arbitrary length as, in general, the length of a path to an invariant violation cannot be known a priori.

Due to the different genotype encoding methods, both works use different methods for mapping the genotype to a phenotype. The phenotype in this case is a path in the model constructed by the model checkers used by the respective implementations. [Godefroid and Khurshid, 2004] use a bit string that is interpreted dynamically to handle arbitrary numbers of transitions at any state. The path length problem is dealt with by searching up to a fixed path length d. If a path with length n less than d is encountered, then there are spare bits after the nth bit. The path is said to have an effective length of n. Standard bit string mutation and crossover operators are applied to positions up to the effective length n.

[Alba et al., 2008] use a vector of real numbers v in the range $[0..1)$ to encode a path in the transition system. When making a choice in the transition system using the ith element v_i, v_i is simply multiplied by the number of transitions available at the

Figure 16: Numbered choices in a transition graph, from [Alba et al., 2008]

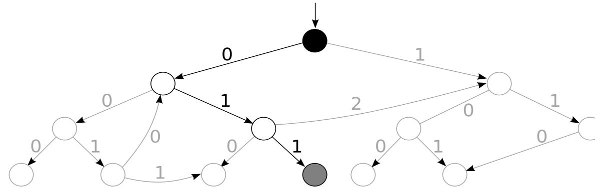
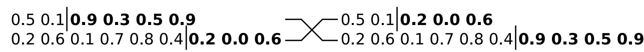


Figure 17: Real-Numbered Vector encoding of a path in a graph, from [Alba et al., 2008]



state in question. In order to handle paths of arbitrary length, the chromosome is permitted to vary in length according to specialised crossover and mutation operators. These processes are both illustrated in Figures 16 and 17. [Alba et al., 2008] outline a *memory operator* (MO) mechanism that exploits the history of the GA search. The MO uses the observation that the earlier transitions of a path through the state-space tend to "lock" on certain values. The MO stores the fitter prefixes in order to save on memory later in the search, allowing longer paths to be explored. This approach appears analogous to the memory saving techniques employed when using ACO [Alba and Chicano, 2007, 2008] which is discussed later in this chapter.

6.1.2 *Fitness Functions*

Both works describe fitness functions for detecting deadlock in a transition system. [Godefroid and Khurshid, 2004] use an interesting heuristic that sums the number of possible transitions from all states along the candidate solution path. The assumption here is that the number of possible transitions will decrease over the course of a path, finally leading to deadlock. It is not clear from the paper as to whether the length of the path is taken into account, but I assume that shorter paths with fewer possible transitions are favoured. [Alba et al., 2008] use a somewhat basic fitness function in their work, and it is described by the equation in figure 18. In this equation, $deadlock$ is set to 1 when the final state is in deadlock, $numblocked$ is the number of blocked

Figure 18: Fitness function for deadlock from [Alba et al., 2008]
$$f(x) = deadlock + numblocked + \frac{1}{1+pathlen}$$

threads in the final state of the path and $pathlen$ is the length of the path in terms of the number of transitions.

[Godefroid and Khurshid, 2004] also briefly describe some elements of a fitness function for finding arbitrary invariant specification violations, however the description is very brief. Both works highlight that some work on the fitness function will be needed in order to check for other invariant properties, but the principals of the solution space encoding can remain the same. Whilst the fitness functions discussed by both works are adequate for finding invariant violations in a large class of systems, for greater efficiency improvements must be made in this area. Some potential improvements will be discussed later in this report.

6.2 PARTICLE SWARM OPTIMISATION

Using the vector of real number solution encoding described above, [Ferreira et al., 2008] have also applied the Particle Swarm Optimisation (PSO) algorithm to finding counterexamples in transition systems. To this end, a population of particles fly around an n-dimensional hyperspace, where n is the length of the floating point vector. Each particle is a solution in their approach. Each of the n dimensions represents one floating point value in a vector solution, and has a boundary of $[0..1)$. The position of each solution particle p is updated at every iteration according to a simple rule, which is a function of the nearest neighbours of p as well as the best solution in the population. The hope is that aspects of "swarm intelligence" [Kennedy and Eberhart, 1995] are exploited in order to optimise solutions until a goal is found. The solution length is fixed apriori, so some knowledge of the depth of the error must be known by a practitioner. This can lead to some wasted effort if the discrepancy between the actual error depth and the solution length is large.

6.3 ANT COLONY OPTIMISATION

[Alba and Chicano, 2007] apply ant colony optimisation to finding safety errors in PROMELA models using the HSF-SPIN framework. Ant colony optimisation aims at finding low cost paths through graphs that represent a solution. In this instance, the ant agents are tasked with co-operatively discovering short paths to safety violations in a transition system. The problem of finding counterexamples in transition systems is naturally analogous to ACO, as both are searching graph structures. The cost of solu-

tions in this case is simply the length of the counterexample with respect to the number of transitions.

[Alba and Chicano, 2007] describe a mechanism for dealing with large transition systems known as the *missionary technique*, as part of the algorithm ACO for huge graphs (*ACOhg*). Once an ant has reached a predefined depth, the best paths that the ants have discovered are stored. A new ACO *stage* then begins, forgetting any previous pheromone trails that have been built. In the new stage, the ants start in the last states of the stored best paths, continuing the search. This technique is used to avoid storing large amounts of data to represent pheromone trails, as pheromone data is potentially required for every edge/transition in the system. The number of edges/transitions in a system can be unknown a priori, and can be intractably large.

[Alba and Chicano, 2007] report excellent results using this mechanism on a variety of benchmark problems. The authors compare the approach against guided model checking techniques, such as A* and best-first search, as well as the GA of [Godefroid and Khurshid, 2004]. The affinity of ACO's target domain and the problem of searching a transition system is evident in this work, producing quite remarkable results. ACOhg outperforms the traditional mechanisms in terms of the rate at which it found errors, as well as the counterexample length. Shorter counterexamples are preferred. It would have been interesting to see a comparison with the GA from [Alba et al., 2008], although the evidence suggests that ACOhg would vastly outperform the GA. The approach has also been shown to work well with partial order reduction techniques [Chicano and Alba, 2008b].

6.3.1 *Liveness Properties*

[Alba and Chicano, 2008] have taken the ACO-based approach to model checking and extended it to support the checking of liveness properties. The authors have dubbed the algorithm *ACOhg-live*. ACOhg-live operates on the product automaton approach to model checking [Vardf and Wolper, 1986; Baier and Katoen, 2008]. In this approach, the negation of the liveness property is transformed into a Büchi automaton using an automated process. A Büchi automaton differs from traditional automaton in the way it accepts strings. The language which a Büchi automaton B accepts is made up of all the strings that visit an accepting state in B infinitely often. A product automaton of the negated liveness property automaton and the transition system is created, with both automatons transitioning synchronously. The end result is a product automaton that can be searched in the same fashion as when searching for deadlock.

Ant agents are then employed to search over the product automaton. The ant agents co-operatively find the shortest path to an accepting state in the product automaton. Once an accepting state is found, ACOhg-live allows a finite amount of time for the ant agents to shorten the path as far as possible. After this finite amount of time, a secondary stage commences. This stage aims to find a cycle in the graph from the found accepting state back to itself. If none can be found in some time period, the entire process is repeated from the initial stage. The process uses techniques to avoid unnecessary repetition of work, and implements a PROMELA mechanism that allows the process to ignore irrelevant parts of the automaton.

6.4 EVALUATION OF METAHEURISTIC MODEL CHECKING WORK SO FAR

Using all of the approaches described above, is has been reported that each technique manages to effectively narrow the search space in order to discover deadlock errors. Each approach outperforms an exhaustive search of the transition system in terms of the error detection rate (the probability of finding an error) as well as computational resource usage. In addition to this, [Alba and Chicano, 2008] reports excellent results when searching for liveness violations in PROMELA models. The authors report that the method obtains shorter counterexamples for all the models tested, and in some cases uses much less memory and CPU time.

Recently as of the time of writing, [Chicano et al., 2011] have performed a comparison of the techniques described above, along with Simulated Annealing (SA), when searching for deadlock errors in Java programs using the Java PathFinder tool. The authors examine four different systems, Dining Philosophers (Loop and no-loop), Stable Marriage Problem , the Global Inter-Operability Protocol (GIOP) and a communication protocol called GARP. The authors included depth-first search (DFS), breadth-first search (BFS), A* search, beam search (BS) and random search (RS).

The comparison shows that all the metaheuristic mechanisms have a higher or equivalent error detection rate than that of DFS, BFS and A*. However, the comparison shows that all of the algorithms (including traditional as well as RS) fail on the Dining Philosopher model as it is scaled up to large sizes. Somewhat surprisingly, the comparisons also show Random Search achieving an equivalent or higher hit rates than every other algorithm in the comparison, suggesting that RS is the most robust (in terms of error detection) option of those compared.

[Chicano et al., 2011] also show the abilities of the algorithms compared to optimise the errors found. An optimised error is an

error with the shortest path to said error. Shorter errors are preferred as superfluous information related to the error is removed. With regards to finding optimal errors, all of the metaheuristic approaches are able to optimise errors (when they can be detected) whilst traditional techniques (including A*) cannot. Random Search, whilst being robust in the error detection rate, tends to find poor quality paths to errors in almost all cases.

6.5 summary

6.5.1 *Limitations of State Of The Art*

The state of the art in the area of applying metaheuristic search to searching transition systems shows some promise with regards to error detection rates and the optimisation of path lengths. However, there appears to be a trade off between robustness (with regards to error detection) and optimisation capability in the algorithms available at the time of writing. All of the traditional, guided and metaheuristic search mechanisms appear to fall short on the robustness criterion, whilst random search has high robustness but lacks an ability to perform optimisation.

It seems as if the GA, ACO, PSO and SA approaches are too conservative in their exploration of the state space, whilst random search is broader but not as broad as the likes of A* and other traditional mechanisms. With respect to GAs, I believe this is due to the way the way the paths are encoded and the variable length crossover mechanism. It is plausible that too much effort is spent on learning the appropriate length of the vector before any optimisation takes place. Additionally, the mapping of floats to choices may also cause problems, as changes at the beginning of the path can have huge consequences for future transitions. The can potentially lead to the loss of partial solutions toward the end of the vector in values at the beginning of the vector are changed. Since PSO and SA are also using this encoding, this problem may be limiting both of those algorithms. Additionally, it is has not been proven that GAs, PSO and SA can find liveness errors, whilst ACO has been proven effective at this task.

ACO seems to have a problem with memory usage when applied to search transition systems, and this has been addressed with the ACO "huge graph" (ACOhg) variant of the algorithm. However, I believe this locking in of solutions may be having a detrimental affect upon the search process, leading to a more conservative search. This is evident in the results of [Chicano et al., 2011], showing that random search has a equivalent or higher hit rate on all of the benchmark problems tested.

6.5.2 *Potential Routes Forward*

The majority of the popular metaheuristic optimisation algorithms have been applied by others to the problem of finding counterexamples in large state spaces. There is one notable exception, however, and that is the use of Estimation of Distribution Algorithms (EDAs). Using the real number vector representation of [Alba et al., 2008], one could quite easily apply some of the real number vector EDAs described in [Pelikan et al., 2002] to the problem. However, I believe that the real number vector encoding has problems due to later vector elements being highly dependant on earlier ones. Consequently, small changes earlier in the genotype can result in disproportionate fitness changes, and a potential loss of partial solutions later on in the vector. Due to this, I believe an entirely new approach must be developed that steers clear of the real number vector approach, whilst attacking the limitations of the algorithms discussed above.

6.5.3 *Summary*

In this chapter I have highlighted and evaluated the current state of the art in applying metaheuristic search algorithms to the problem of finding counterexamples in large transition systems. It is my belief that a new encoding and approach to searching transition systems must be proposed and evaluated. The new approach must be as robust as random search, whilst maintaining the optimisation capability of established metaheuristic techniques. The algorithm must be agnostic to the depth of the error within the transition system, as it seems that error depth limits the efficacy of both GAs and PSO. The new approach must be memory efficient, in order to avoid the memory saving tricks employed in both ACOhg and the memory operator in the GA based approach. The new technique must also maintain the run time and error detection rate advantages over traditional transition system search techniques that other metaheuristic search techniques seem to have.

To this end, in the next chapter I will describe in detail an EDA-based approach to searching transition systems based on work in [Poli and McPhee, 2008]. In the chapters following that description, I will present empirical work that demonstrates the potential of the new approach when used to find faults in transition systems, to optimise the length of those faults and to scale to large systems.

Part III

ALGORITHMIC PROPOSAL

7 ALGORITHMIC PROPOSAL

The algorithmic proposal described below aims to address the apparent flaws in the current state of the art of metaheuristic search of transition systems. The main problem seems to stem from the arbitrary length of paths to errors that can crop up in test suites and real systems. Whilst ACO can handle paths of arbitrary lengths, it has a real problem with memory usage and hence the need for memory optimisations in the form of ACOhg [Chicano and Alba, 2008a]. Whilst GAs, PSO and SA annealing all have low memory usage, measures are required in order to adjust the solution representation on the fly to account for arbitrary path lengths. I believe the algorithm proposed below lies somewhere in the middle of this trade off, having reasonable memory usage and the ability to represent arbitrarily long paths. With this improvement, the hope of achieving the robustness of random search whilst being able to optimise solutions could be achieved.

What follows is a detailed description of the proposed algorithm, with justifications for various choices at each stage of the design.

7.1 MODEL AND SOLUTION SPACE

The solution space of paths in a transition system requires a novel modelling and encoding mechanism to address the problems outlined above. There are a number of choices available when encoding and modelling paths in a transition system. When making this choice, a number of factors must be considered. One of the major factors when choosing a modelling and solution representation is the ability to represent paths of arbitrary length. In some situations, knowing the length of a counterexample may be infeasible. The majority of EDA algorithms can effectively deal only with representations of a fixed or known size, and are focused on optimisation bit strings. I also want to avoid mapping real number vectors to choices used in the GA, PSO and SA algorithms discussed in the previous chapter. EDAs for optimising real numbered vectors are few in number, and empirical evaluation of such techniques on a wide variety of problems do not exist. Additionally, the loss of partial solutions discussed in the previous chapter would also be problematic when using EDAs to optimise real numbered vectors.

In order to encode paths in a transition system, a simple string representation is used. A path in a transition system can be viewed as a sequence of actions causing transitions between states. The alphabet of the string representation used in this work is the set of actions that can be executed in the transition system. The choice of action set is problem dependent and can be at any level of abstraction. In this work, we retrieve the alphabet from Java Path Finder APIs [Visser et al., 2003], where each alphabet member represents a choice of action in the transition system. Each alphabet member consists of Java file names, line numbers and byte code instructions.

Examples of the typical alphabet members from the Java Path Finder, a prominent Java software model checker, are shown in Figure 19. All alphabet members are of the same format, apart from a few special instructions. In the alphabet member on line 6 of Figure 19, <examples/DiningPhil.java:38> refers to the filename and line number currently being executed. < synchronized (left) {> is the contents of line 38 in DiningPhil.java and <monitorenter> is the instruction being executed. Note that the alphabet is thread independent in an effort to scale well with symmetrical problems, e.g. dining philosopher systems. A typical path in a system is shown in Figure 19. In the path, there are a few special instructions at the start of the path that do not follow the format described. These instructions are special Java instructions relating to the creation of the initial objects and are at the beginning of every path in JPF.

When searching over other forms of models, such as Promela models or C code, a suitable alphabet must be chosen. The rationale for the alphabet used in this work is as follows. The alphabet must be fine grained enough to include some reference to the bytecode instruction or atomic action being executed. Simply referring to the line number in a Java or high level source code file may ignore a lot of crucial detail, as many bytecode or atomic instructions can be executed when executing a high level statement. However, referring to atomic instructions alone is not enough as it is highly likely a particular atomic instruction is executed as a part of many high-level statements in a given program. Good results were obtained using a combination of the high-level statement being executed as well as the atomic instruction during small-scale empirical evaluation.

To model strings in the solution space, a variant of N-gram GP is used [Poli and McPhee, 2008]. An n-gram is a subsequence of length n from a longer sequence. N-gram GP learns the joint probabilities of fit string subsequences of length n. In this work, the n-grams represent recent histories of n actions, and the distributions associated with these N-grams are sampled to determine the best action to choose next. N-gram GP has the advantage of

```
1  <null transition>
2  <[synthetic] [clinit]<clinit>><[synthetic] [clinit]<clinit>><invokeclinit
       >
3  <java/lang/ThreadGroup.java:859><(java/lang/ThreadGroup.java:859)><
       monitorexit>
4  <examples/DiningPhil.java:38><    synchronized (left) {><runstart>
5  <examples/DiningPhil.java:38><    synchronized (left) {><getfield>
6  <examples/DiningPhil.java:38><    synchronized (left) {><monitorenter>
7  <examples/DiningPhil.java:33><    start();><invokevirtual>
8  <java/lang/ThreadGroup.java:844><(java/lang/ThreadGroup.java:844)><
       monitorenter>
9  <examples/DiningPhil.java:39><      synchronized (right) {><getfield>
10 <examples/DiningPhil.java:39><      synchronized (right) {><monitorenter>
11 <examples/DiningPhil.java:41><      }><monitorexit>
12 <java/lang/ThreadGroup.java:859><(java/lang/ThreadGroup.java:859)><
       monitorexit>
13 <examples/DiningPhil.java:38><    synchronized (left) {><runstart>
14 <examples/DiningPhil.java:42><    }><monitorexit>
15 <examples/DiningPhil.java:33><    start();><invokevirtual>
16 <examples/DiningPhil.java:38><    synchronized (left) {><getfield>
17 <examples/DiningPhil.java:41><      }><monitorexit>
18 <examples/DiningPhil.java:42><    }><monitorexit>
```

Figure 19: A typical trace/string/path from JPF on the Dining Philosopher problem with 2 philosophers. This trace does not include a deadlocked state. Note the lack of references to specific threads.

not being limited to a fixed size representation. One can sample from an N-gram GP model until some stopping criterion is met. This will discussed in the model sampling section.

7.2 LEARNING THE MODEL

The model learning step used in this work is a simple frequency count of actions after each unique n-gram. A set of fitter paths or strings S is selected from the current population. Then, a count of frequencies is performed on all the unique n-grams over all the strings in S. The frequency count is then normalised to obtain a probability distribution. An illustration of this process can be found in Figure 20, where the As and Bs maps onto alphabet members like those shown in Figure 19. In addition to learning all the n-gram distributions, the distributions for (n-1)-grams and (n-2)-grams and so on are also learned down to 1-grams.

Figure 20: Illustration of the N-gram learning process (2-grams in this case). A frequency count is performed for each unique N-gram in the selected set of strings. The boxes represent a basic sliding window algorithm, with frequency counts display on the right.

When learning the model, there is a choice between building an entirely new model from the selected individuals or updating the existing model using the selected individuals. In this work, the entirely new model approach was taken as it proved more effective when evaluated empirically. After each generation, the model is destroyed and a new one constructed using the procedure described above. This eliminates any kind of "update rate" parameters which may have to be tuned to match problem characteristics.

There is a small issue with regard to the substrings at the start of each string/path. At the beginning of each path, there are substrings that are less than the required n-gram length n. This is dealt with in the same way as the work in [Poli and McPhee, 2008]. A distribution is learned for all of the n-grams up to length n at the start of the strings only. From then onwards, the method outlined above is used.

The model learned using this method can be described as a strategy for traversing the transition system. Given a recent history of actions of length n, the n-gram model can be queried to obtain a distribution that can be used to choose the best next action. If the alphabet is thread agnostic then this model has the potential of describing concise solutions for highly symmetrical problems. The model is also agnostic to the type of property being verified.

7.3 MODEL SAMPLING

To sample the model learned above, paths are generated using the learned model as a guide for choosing actions in a state. To generate an individual/path in the transition system, the sampling method initially starts with an empty string p representing the history of actions and the initial state i of the transition system. Then, an algorithm called $MakeAChoice()$ is executed, which takes a state s (initially i) and the history of actions p as arguments and returns an action/transition a that is possible from s. The transition a is then chosen in the model checker, yielding a new state i'. The action a is appended to p to accurately reflect the history of actions on the current path. $MakeAChoice()$ is then called with i' and p as arguments to yield the next state. This process is repeated until either a state that has previously been encountered on this path, or an end state is reached.

The $MakeAChoice()$ algorithm can be broken down into a number of stages. The first stage is *distribution acquisition*, in which the model is queried for a relevant distribution given a recent history of actions. From the input string p, the n most recent actions are obtained from the end of p. This n-gram r is then used to query the model to obtain a distribution. If no distribution is known for r, r' is created using the most recent $n-1$ actions in p and the model is queried again. This shortening of the n-gram is repeated until a distribution is found, or until the n-gram length is 0. If no distribution can be found for the n-gram, or any shorter n-gram, then a "blank" distribution is obtained. If a blank distribution is returned from the model, this indicates that the n-gram was not observed during the model learning phase of the EDA.

Once a distribution is obtained, the *distribution sampling* stage is executed. $MakeAChoice()$ takes a state s as an argument. From s, a set of transitions T are available. The goal of the distribution sampling stage is to choose one of the transitions available from s. Each transition in T represents the progress of a component within the transition system, in this case a thread of a multithreaded program. Each transition in T has an associated action that caused that transition. Therefore, there is a set of actions A

possible from state s. A number of transitions may be caused by concurrent components taking the same action. For instance, a group of threads may be competing to obtain a lock on an object. Thus in some instances, the cardinality of the set of possible actions A may be less than the cardinality of T.

In this proposal, the distributions in the model are distributions over the possible actions from a state rather than the transitions. The rationale behind this is to aid in the scalability of the approach on problems with symmetrical descriptions, such as the dining philosophers coordination problem, whilst gracefully degrading for non-symmetrical problems. If n threads (where $n > 1$) are poised to take an action, and that action is chosen, then one of the n threads is chosen at random.

When the set of actions A are the same actions as those represented by the obtained distribution, then a choice is made according to that distribution. However, when A is not perfectly described by the obtained distribution, a few special cases which must be handled during the $MakeAChoice()$ procedure. If these special cases are ignored, then paths that do not exist within the transition system could be generated. The first is when there are actions in the distribution that are not in A. When this occurs, the excess actions in the distribution are culled and the distribution is normalised. Then an action is selected according to the distribution.

The second case is when one or more actions in A are not in the distribution. In this algorithm, a low probability of selection is given to those actions that are not in the distribution. The actions in A that are in the distribution are given their respective probabilities, whilst those that are not in the distribution are given half the lowest probability. Then the distribution is normalised and sampled to yield a choice of action. An alternative to this can be to set the probabilities of actions that are not in the distribution to some arbitrarily low value.

A third case also exists. If no distribution can be found for a recent history, a blank distribution is returned and a uniformly random choice is made between the actions in A. In addition, to generate the initial population, the model could be seeded with a blank distribution for every n-gram. With this approach, all individuals in the initial population would be generated at random.

7.4 FITNESS FUNCTION

An EDA samples a solution space in a meaningful way in order to find good solutions, using a fitness function as a guide. Solutions are selected from the population, with a bias toward the fitter individuals. The selected solutions are then used to learn the

n-gram model. In this proposal, truncation selection is used as this seems a popular choice in the EDA community [Pelikan et al., 2002]. Truncation selection simply selects the top n or $n\%$ of the population.

Metaheuristic search algorithms can be viewed as function optimisers, aiming to minimise or maximise a particular objective function. The solution space outlined earlier is the set of all possible paths in the transition system of the code under test. Good solutions in this solution space are those that end with an error and are short. Solutions in the solution space that are "closer" to leading to an error state must rank higher than those that are "far away".

Determining how close a path is to a path ending in an error state is an open and challenging question. For instance, the sequence of actions and states leading up to a deadlocked state in a commercial software system may be very different from a deadlock in a poor dining philosophers coordination policy. Defining a general deadlock finding fitness function is likely impossible, with each problem potentially requiring a custom fitness function with domain specific knowledge to be effective. A possible avenue of research is to automatically derive helpful metrics for this purpose using analysis of the system under test.

Algorithm 1 Fitness function used to rank individuals.

Require: A, B are Individuals;
  if $A.error_found \neq B.error_found$ **then**
    return IndividualWithErrorFound(A,B);
  else if $A.error_found$ **and** $B.error_found$ **then**
    return IndividualWithShortestPath(A,B);
  else
    return IndividualWithLowestStateMetricSummation(A,B);
  end if

Algorithm 2 State metric summation algorithm.

Require: I is an Individual;
  aggregateState = 0;
  for all $States\ s \in I.Path$ **do**
    aggregateState += s.SingleStateMetricFromModelCheckingTool;
  end for

One problem that must be addressed is the fact that path based heuristics do not exist in model checking tools. Every heuristic available in mainstream model checking tools gives heuristic information about a single state only, estimating how close that state is to some desired goal. Determining the fitness of an entire path, as opposed to a state, requires a different approach. One could potentially choose a representative state

for the path, either a random state in the path or the end state, and use that as a measure of the path's overall fitness. However, previous work [Alba and Troya, 1996; Godefroid and Khurshid, 2004; Alba et al., 2008] suggests that combining the individual state heuristic along the path can be effective. In this proposal, we take a similar approach by simply summing the individual state heuristic values along the path. It is assumed that the lower the overall summation value, the closer the path is to an error or goal state. This algorithm is outlined in Figure 2.

The summation algorithm in Figure 2 is then used by a fitness function to rank paths described in Figure 1. The function is a path comparison function, comparing two paths and returning the more desirable path. The fitness function is structured simply, and the functions called in the algorithm perform the actions indicated by their respective names. *IndividualWithShortestPath(A,B)*, for instance, returns the individual with the shortest path. The fitness function will always favour a path that contains an error state to one that does not. If both paths contain an error state, then the shortest path is favoured. If neither path contains an error state, then the path that has the lowest metric summation is chosen. This is a generic fitness function for ranking paths in a variety of situations, either detecting or optimising errors for example. If the goal is to simply detect errors and stop, then the shortest path and error found metrics do not get exercised. However, if the goal is to find and optimise errors, then the other two metrics will be exercised once errors are found. Algorithm 2 also has an implicit path shortening effect, as the summation of a shorter path will likely have a lower metric than that of a longer one.

7.5 OTHER PARAMETERS AND FEATURES

A common feature of EDA implementations is a mutation operator, similar to that used in genetic algorithms. The purpose of such an operator is to introduce new genetic material into a population. In this proposal, mutation occurs when choices are made. When making a choice during the path generation phase of the algorithm, with probability m, a uniformly random choice is made from the available actions. If m is set to 1.0, all choices are made at random yielding a random path search mechanism. The probability of mutation in the majority of implementations of EDAs or GAs is typically set to a low value as a high value can be disruptive to the search process.

A common feature in Evolutionary Algorithms is the notion of *elitism*, and is sometimes implemented in EDA work. Elitism is the practice of copying the best n individuals from one generation to the next without mutation or change, whilst the rest of the

population are generated in the usual way. In this proposal, elitism was implemented and seemed to give better results than an EDA without elitism in small-scale experimentation.

7.6 NOVELTY OF ALGORITHM

N-gram GP in vanilla form [Poli and McPhee, 2008] is aimed at generating and searching strings that are then interpreted as a program. It is the author's belief that this is the first time N-gram GP, or indeed n-gram sequence modelling, has been applied in the context of searching state spaces or transition systems. This proposal constitutes the first time anything labelled as an EDA has been applied to the problem of searching transition systems or state spaces, therefore the first time an EDA has been applied in a model checking context. I believe that the algorithm potentially has applications outside of searching transition systems, particularly in circumstances where a labelled state space is being searched.

The model sampling phase of the algorithm constitutes the main difference between N-gram GP [Poli and McPhee, 2008] and this proposal. Vanilla N-gram GP is not restricted by an underlying model in the choices that can be made at each stage. Any alphabet member can follow any other during the course of model sampling. This can be pitched as a state space search, where each state consists of the action sequence or program generated so far, and all actions are possible from every state.

The approach addresses directly some of the shortcomings of other metaheuristic approaches to this problem. Firstly, the arbitrary path length limitation of a floating point solution space encoding, used by the GA, PSO and SA algorithms highlighted in the previous chapter, has been addressed. The proposed algorithm can generate paths of arbitrary length, requiring no prior knowledge of the potentially solution depth in the state space. Secondly, I believe that the memory usage concerns of ACO have been addressed. Rather than store pheromone values for a potentially huge number of arcs in the state space, a more limited number of distributions are stored for the action space of the model. Whilst the action space of a transition system could be as large the number of edges, in the benchmark scenarios examined by previous work it is always the case that the action space is much smaller than the number of edges. This potentially delivers a memory usage reduction for the algorithm, avoiding the necessity of memory saving measures such as those in [Alba and Chicano, 2008].

Overall, this proposal transforms elements of a formal method into a form of software testing. I feel it best to think of the algorithm as focussed stress testing, akin to tools like ConTest

[Edelstein et al., 2003] but with the precise counterexample and repeatability that model checking offers. The proposal has the potential to drastically reduce the time to detect an error in a concurrent system, whilst eliminating false positives.

7.7 IMPLEMENTATION

7.7.1 *N-gram Implementation*

Due to the recent development of n-gram GP, at the time of writing there is no generic framework for producing n-gram GP like algorithms. In order to experiment with the above proposal, I had to implement a bespoke framework. I briefly evaluated a variety of EA frameworks. Framework performance was not an issue, as the majority of the computational effort will be made expanding states in model checking tooling. Extensibility was my primary concern, and since I had experience with a particular framework in the past, ECJ, I decided to capitalise on time invested. Additionally, for the following chapter, the model checking tool used during experimentation is also written in Java, allowing for easy interoperability.

ECJ has a number of useful features (generic genetic operators, multithreaded evaluation, sane parameter system etc), but has no generic EDA implementation. ECJ is highly equipped for solution interaction, for example the combination of (typically) two solutions to create new solutions. EDAs on the other hand focus on a global view of the population, selecting a large number of individuals and creating a model. This renders a large chunk of ECJ features somewhat useless. However, by creating a new "breeder" within ECJ, I managed to salvage other the above mentioned ECJ features. The breeder implemented a basic truncation selection and a basic bean counting algorithm to construct the n-gram model.

7.7.2 *Interaction with a Model Checker*

The custom ECJ implementation controls a model checker to produce a population of paths. The custom ECJ implementation does this by choosing actions repeatedly, starting in an initial state, until a valid path is produced. Each model checker used in the experimental chapters of this thesis offer the same basic feature. Given a model/system and a history of actions, a model checker will return a set of options from which to choose from. For instance, the Java PathFinder (JPF) model checker takes a Java program, and allows an outside mechanism to control which thread progresses next. Given the set of choices, the implementa-

tion of the proposed EDA has to present a strategy for choosing an option.

Injecting a strategy for choosing into the JPF model checker was easy enough, as I had access to JPF source code so everything could be done in process. However, for the later chapters a C-based model checker was used. After discounting Java's native API due to awkwardness, I decided on an out of process approach, passing an encoded strategy from Java to C via the file system. Then, in the HSF-SPIN model checker, a lightweight path expansion process used the strategy to expand a path, and then return the sequence to ECJ. HSF-SPIN allows for searching of models described in PROMELA, a popular language/toolkit in the model checking community. Implementation of the model checking expansion phase was abstracted, yielding a clean(ish) interface of passing strategies to other toolkits. The abstraction amounted to "Given this strategy, return me a sequence of alphabet members/actions/strings".

A large amount of time (6 months+) was spent making ECJ interact with the JPF model checker, due to having to understand the inner workings of JPF and long execution times. Half way through my PhD programme, I started using the enormous computation power of the SEBASE (now NSC) grid, which sped up development on the HSF-SPIN interaction. This environment allowed for checking of larger PROMELA models, experiments which feature in later chapters.

7.8 computational complexity

Empirical comparison of various metaheuristic algorithms on model checking algorithms are difficult due to the variability of implementations. For instance, it is difficult to discount the unnecessary memory usage of unused parts of ECJ.

7.9 summary

In this chapter, I have outlined an algorithmic proposal based upon Estimation of Distribution Algorithms to find error states in transition systems. In the following chapters, I will show empirically the potential of this algorithm to discover multiple error types in a wide variety of transition systems. Additionally, I will show how the models can be reused in order to save effort in the software development cycle, something I believe is unique to EDAs and the proposed algorithm above.

Part IV

EXPERIMENTATION

8 FINDING DEADLOCK IN MAINSTREAM LANGUAGE CODE

8.1 introduction

In this chapter, I will demonstrate the potential for the EDA-based algorithm proposed in the previous chapter to find concurrent faults in systems described by mainstream languages. Specifically, this chapter will focus on the task of finding deadlock in Java programs. As established in Chapter 3, deadlock in shared-memory concurrent systems can be subtle, and can avoid detection when faced with strenuous stress testing. However, the proposed algorithm will potentially focus much of the stress testing effort on areas of the system's state space that are more like to contain an error.

By targeting a mainstream language such as Java, integration of the proposed algorithm into Integrated Development Environments (IDEs) such as Eclipse become a possibility. One can imagine dedicating a percentage of our envisioned many core computers [Intel Corp., 2007] in the future to searching the state space of concurrent software. This could happen in the background, and alert developers to problems in the same way that static analysis can reveal dead code and compilation errors. Model checkers exist for a wide variety of mainstream languages, including the .NET platform [Aan de Brugh et al., 2009] and C/C++ [Clarke et al., 2004]. Tools like these can be used for integration into their respective popular IDEs.

The work in this chapter is based on [Staunton and Clark, 2010].

8.2 experimentation

To experiment with the proposed algorithm, the implementation described in Chapter 7 was used with the Java PathFinder model checker. An implementation of the Dining Philosophers problem is used to test the proposed algorithm. In this experimentation, a naive coordination strategy is used for benchmark. The benchmark, known in the JPF community as *DiningPhil*, implements the following policy. All philosophers first pick up the fork to the left, then the fork to right, eat, and then release the right and left fork in that order. The philosophers then terminate execution. The implementation used in these experiments is the Java class included with JPF. The size of the state space of this particular

Dining Philosopher implementation grows exponentially with the number of philosophers. [Alba et al., 2008] report state space sizes of 2094 and 120544 for 3 and 4 philosophers. Experiments were carried out on varying sizes of the DiningPhil problem, from 4 to 40 philosophers.

8.2.1 *Fitness Function*

The fitness function used for this set of experiments follows the generic fitness function outlined in the algorithmic proposal chapter. The algorithm described in 3 is identical to the 1 in the previous chapter, apart from we are trying to maximise our cumulative state metric. The heuristic information for each state is gathered from heuristics available within Java PathFinder. Specifically, the "blocked threads" metric from JPF was used, which returns the number of threads blocked in a given state.

Algorithm 3 Fitness function used to rank individuals. Individuals that are "closer" to deadlock are favoured.

```
Require: A, B are Individuals;
  return IndividualWithHighestBlockMetric(A,B);
```

Algorithm 4 Blocked threads metric algorithm.

```
Require: I is an Individual;
  aggregateBlocked = 0;
  for all States s ∈ I.Path do
    aggregateBlocked += s.NumberOfBlockedThreads;
  end for
```

The fitness function described in Algorithm 3 depends on deadlock being defined as all threads in the program being blocked, rather than a subset. The assumption is that states with a higher number of blocked threads are closer to deadlock than those in which all threads can progress. By summing the blocked thread heuristic for each state along a path, a measure of the overall "blockiness" for the path is produced. The assumption is that the more high blocking states there are in a path, the closer that path is to leading to a deadlock state. This is an approach taken by previous work [Godefroid and Khurshid, 2004; Alba et al., 2008]. Algorithm 3 is similar to that outlined in Algorithm 1. However, in this work only error detection is considered, so parts of the generic algorithm that are irrelevant have been removed.

8.2.2 *Parameters*

The parameters chosen for this set of experiments are the result of small-scale empirical evaluation and are kept constant for all experiments. An n-gram length of 3 was used, meaning models for 3-grams, 2-grams and 1-grams are constructed from each generation. The population size for each generation was set to 150. This means that 150 paths are sampled from the model to build each generation. The mutation parameter for these experiments is set to 0.001, meaning that on average 1 in 1000 transition choices are made randomly, disregarding the model. The elitism parameter was set to 1, meaning that the top individual from the population is copied to the next generation. In order to build the model from which the next generation is sampled, truncation selection selects the top 20% of individuals from the population. This means that the top 30 individuals from the current population are used to build the EDA model. The algorithm terminates when a solution is found. Initially, the model is a blank model meaning that all the paths evaluated during the first generation are completely random.

8.2.3 *Results and Discussion*

Comparisons of the EDA with other techniques are also shown. The other techniques are depth-first search, breadth-first search and random search. Depth-first search examines the "deepest" states before any others, whereas breadth-first search favours the "shallowest" states. The random search algorithm behaves precisely like the proposed technique in this paper, however all choices are made at random. The algorithm, starting from the initial state, chooses transitions at random until a previously found state is encountered, or an end state is reached. The algorithm repeats this process until a deadlock state is found. The random search process is equivalent to the EDA with the mutation parameter set to 1.0. Since the EDA is a pseudo-random probabilistic process, the results reported are statistics of 50 independent executions. Each run was performed using a unique seed for the pseudo-random number generator. The results for the random search algorithm are also statistics of 50 independent runs, each with a unique seed for the pseudo-random number generator.

The results from the experiments are shown in Table 1. Each of the algorithms were run on the Dining Philosopher problem using gradually increasing numbers of philosophers. Since DFS and BFS are deterministic algorithms, the result of only one execution is shown. Statistically significant differences from the EDA, with a confidence level of 0.05, are shown using $(+)$ to indicate a significant result, and $(-)$ is indicate otherwise. When

Table 1: Results for each of the algorithms

Problem Size		DFS	BFS	RS	EDA
4	Errors / runs	1/1	1/1	50/50	50/50
	Med. Time (s)	(+)**0.491**	(+)3.971	(-)7.246	7.811
	Min. Time (s)	**0.491**	3.971	3.453	4.175
	Max. Time (s)	**0.491**	3.971	18.165	11.933
	Med. Max. Mem. Usage (MB)	(-)482	(+)**80**	(+)951	951
	Med. States Visited	(-)**312**	(-)13029	(+)7,738	5,697
	Med. Paths Evaluated	-	-	(+)221	**163**
	Med. Generations	-	-	1	1
8	Errors / runs	1/1	0/1	0/50	50/50
	Med. Time (s)	(-)25.191	-	-	**22.512**
	Min. Time (s)	25.191	-	-	**11.428**
	Max. Time (s)	**25.191**	-	-	40.859
	Med. Max. Mem. Usage (MB)	(-)**496**	-	-	1,199
	Med. States Visited	(+)238,264	-	-	**58,146**
	Med. Paths Evaluated	-	-	-	822
	Med. Generations	-	-	-	5
12	Errors / runs	1/1	0/1	0/50	50/50
	Med. Time (s)	(+)16375.392 (4h32m55s)	-	-	**49.936**
	Min. Time (s)	16375.392 (4h32m55s)	-	-	**24.420**
	Max. Time (s)	16375.392 (4h32m55s)	-	-	**107.376**
	Med. Max. Mem. Usage (MB)	(-)3233	-	-	**2,130**
	Med. States Visited	(+)150841824	-	-	**152,443**
	Med. Paths Evaluated	-	-	-	1,448
	Med. Generations	-	-	-	9
16	Errors / runs	0/1	0/1	0/50	50/50
	Med. Time (s)	-	-	-	102.709
	Min. Time (s)	-	-	-	38.498
	Max. Time (s)	-	-	-	299.005
	Med. Max. Mem. Usage (MB)	-	-	-	2,895
	Med. States Visited	-	-	-	299,370
	Med. Paths Evaluated	-	-	-	2,128
	Med. Generations	-	-	-	14

comparing the 50 independent executions of the EDA against the single result from a deterministic algorithm, the Wilcoxon signed-rank test is used. When comparing the EDA against the random search approach, both using a sample of 50 independent executions, a Mann-Whitney rank sum test is used. Both tests are valid under general conditions, for instance the assumption of normality is not required. These tests are the non-parametric equivalents to the one and two-sided t-tests.

The "Errors / runs" row of the results table shows the number of runs which found an error. The time statistics are wall clock times measured with millisecond accuracy. Memory statistics are collected from the measurements of JPF. The number of paths evaluated results are only relevant to the random search and the EDA, since DFS and BFS operate on a state by state basis.

When testing the algorithms against the 4 philosopher problem, the DFS algorithm is not only the fastest algorithm in terms of time, but also the most efficient in terms of the number of states visited as well as memory usage. Also worth noting is the equivalent results of the random search algorithm and the EDA approach. One can see from the median number of generations

required for the EDA to find an error that the median is 1. This indicates that the EDA approach in some cases was finding errors without performing any learning, showing that a random search is enough for such a small problem. However, the median number of states examined before finding an error is significantly less for the EDA as opposed to random search. This is explained by when both the EDA and RS fail to find an error in the initial random stage, the EDA is influenced by a generation of model building, and is therefore quicker to find an error in subsequent generations. This reduces the median number of states needed to find an error for the EDA, as opposed to random search which has no model to search by. An odd result was the memory usage reported for the BFS algorithm. BFS typically exhibits high memory usage, and the low memory usage reported by JPF for 4 philosophers may be due to a bug.

When the problem size is increased to 8 philosophers, the results start to favour the EDA. Both BFS and random search fail to find errors in the 8 philosopher problem. BFS ran out of memory (even when given an upper limit of 30GB), and random search examined 30000 paths before terminating. Whilst the time taken to find an error is equivalent between the EDA and DFS, the number of states examined before an error is found is significantly less when using the EDA. Also worth noting is that the median number of generations required to find the error was 5, indicating that the EDA in some cases required several rounds of model learning before an error could be found.

This trend continues when analysing problems with increasing numbers of states. For the 12 philosopher problem, DFS requires an enormous amount of time to find an error. The average time required for the EDA to find an error is significantly less. The most striking difference however is the number of states examined before finding an error. Whilst the DFS results have jumped from 2.4×10^5 states for 8 philosophers to 1.5×10^8 for 12, the EDA numbers have increased more gradually. The EDA, on average, examined 1.5×10^5 states before finding an error in the 12 philosopher problem. This indicates that the EDA is focusing the search more effectively on areas of the state space that are more likely to contain deadlocked states.

For the 16 philosopher problem, the EDA is the only technique that finds deadlock when given reasonable resources. DFS ran out of memory after 24 hours of searching, using all of a 30GB memory limit. The EDA, however, found an error with an average time of roughly two minutes, using less memory.

An often cited advantage of EDAs, as opposed to standard evolutionary techniques such as GAs, is the insight into the target problem the model itself can yield. Whilst the model learned by an EDA is not itself a solution to the problem, the model can

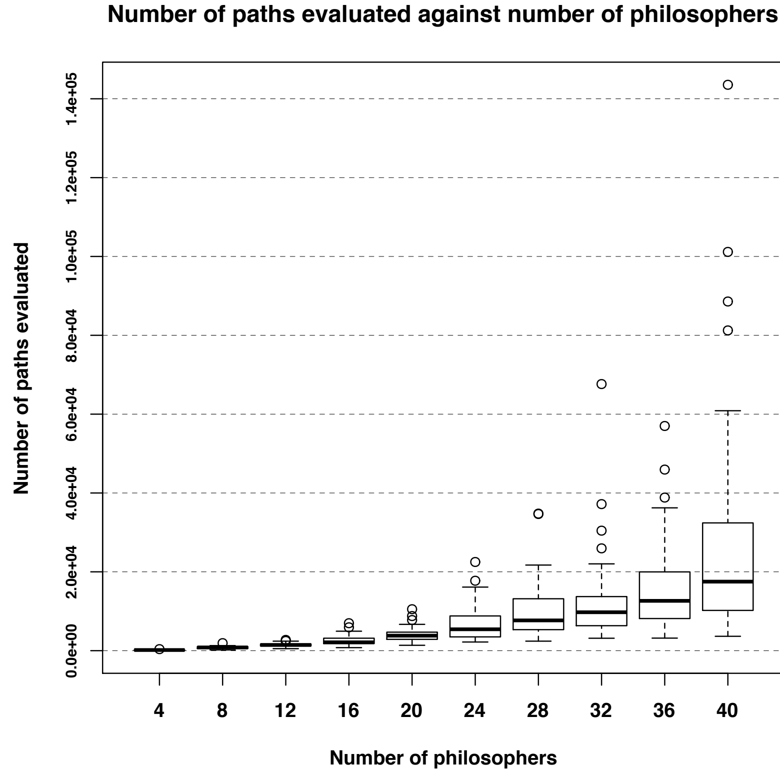


Figure 21: Boxplots showing the number of paths evaluated by the EDA before finding an error against the number of philosophers.

reveal important insights and provide helpful debugging information. The output of a typical evolutionary algorithm run is a bag of solutions and nothing more. The model learned by the EDA model checking algorithm proposed in this paper could indeed provide insight into the target problem. The model structure can highlight interesting interleaving of actions between threads that lead to increased blocking, or whatever attribute the fitness function is trying to optimise.

One interesting phenomenon that was noted from the experiments is that the models from which the error paths were sampled are all similar. The model from which the solution for 8 philosophers was sampled is similar in structure and distributions to the model for 32 philosophers. This indicates that the EDA is learning a strategy for finding errors in the DiningPhil family of problems. The model learned from solving the 4 philosopher problem can easily be sampled from to aid in solving the 32 and higher philosopher problems. One can exploit this phenomena to exert less computational effort by using models learned from smaller variants of problems to effectively search much larger problems. This kind of approach is often used when verifying hardware designs [Clarke et al., 2000].

Figure 21 shows the results of the EDA applied to increasingly larger DiningPhil problems. The parameters for these runs are the same as those described earlier. The boxplots show the outliers, lower, median and upper quartiles for a sample of 50 runs. The number of paths evaluated before finding an error are plotted against the number of philosophers in the target problem, ranging from 4 to 40 philosophers. The figure shows an upward trend in the number of paths evaluated before an error is found for increasing numbers of philosophers. The trend, however, is impressive given that the state space of the problem is increasing at an enormous rate.

Showing results from larger numbers of philosophers is unfortunately difficult due to the high memory usage of JPF on such large problems. The majority of the memory usage of the EDA can be attributed to JPF, as the EDA is lightweight in terms of both memory and computational effort. With a more efficient model checker or with refinements to JPF, I believe there is no reason why larger instances of the dining philosophers problem can not be solved. For instance, I believe that the implementation for this proof on concept could use JPF more efficiently, allowing for checking on larger systems. However, due to time constraints this was not possible.

8.3 SUMMARY

In this chapter, I have shown how the proposed algorithm can be used to detect deadlock in systems described by mainstream programming languages. The approach was demonstrated using the Java PathFinder model checker, a popular tool that can analyse compiled Java programs. The proposed algorithm was shown to outperform common model checking algorithms depth-first search and breadth-first search, as well as a random search procedure on large instances of the Dining Philosopher problem. For larger instances of the Dining Philosophers problem (16 philosophers and above), the EDA was the only algorithm to find deadlocked states using reasonable resources. The EDA-based approach produced results using a competitive amount of resources for smaller instances that are trivially solved using random search and deterministic techniques. This means that there is little additional cost to running the EDA as opposed to other techniques, whilst retaining the advantages shown in the experimentation.

In this chapter, only the dining philosophers system is used to demonstrate the potential of the proposed technique. Whilst the algorithm in the proposed form seems effective on the Dining Philosophers problem, the simple recent history algorithm may not be as effective on a wider range of systems. However, I believe

that the algorithm can be extended in certain circumstances to potentially handle more difficult problems. As well as making transition choices based upon a recent history, other contextual information can be used. For instance, using the current length of the path during the sampling stage of the could help make a more informed decision about which action to choose. A different n-gram model could be used for choices at depth 0-20, 21-40 etc. N-gram models could be based upon the current active classes in a given state, allowing for history models to be different depending on which types of threads are interacting.

In this work, only the Dining Philosophers example is used to prove the potential of the algorithm. It is safe to say that the Dining Philosopher problem is not representative in common industrial concurrent systems, and can be considered as a theoretical exercise. In the next chapter, I will show how the algorithm can be used not only on examples derived from industrial code, but also to find other errors types such as assertion violations and liveness properties.

9 FINDING AND OPTIMISING COUNTEREXAMPLES IN INDUSTRIAL CODE

9.1 INTRODUCTION

In the previous chapter I have shown how the algorithmic proposal can be used to find deadlock in Java programs, proving the potential for finding deadlock efficiently in mainstream language code where other techniques fail. The example used was the Dining Philosophers problem, which most would argue is not representative of a typical concurrent system found in industrial settings. In this chapter, I demonstrate how the proposed algorithm can be used not only on more complex system descriptions, but also to find error types other than deadlock. The complex system descriptions examined in this chapter are taken from models included with the SPIN/HSF-SPIN distribution [Edelkamp et al., 2001a], and are derived from industrial settings by manual translation into a format more amenable to checking. Analysing arbitrary systems, such as an open source Java program, is outside the scope of this chapter.

As well as demonstrating the ability of the proposed algorithm to find multiple error types on more complicated systems, I establish the ability of the EDA to find high quality solutions to the model checking problem. The quality of a solution, or counterexample, is measured solely in terms of the length of the path to an error. Shorter counterexamples are favoured over longer ones. A developer trying to debug an error is only interested in the actions that are necessary to cause the error. A long path to an error, as opposed to a short path to the same error, contains superfluous information with regards to what causes an error. The ability of metaheuristic algorithms to optimise the quality of solutions found to the model checking problem has been established, so to compete the same must be able to be said about the proposed algorithm.

The work in this chapter is based on [Staunton and Clark, 2011b].

9.2 EXPERIMENTATION

9.2.1 *Implementation*

In order to demonstrate the capabilities described above, this work uses a well established model checker both in the hardware

and software model checking community called SPIN. Specifically, we use the HSF-SPIN [Edelkamp et al., 2004] variant, a framework which allows heuristic algorithms to be implemented within the SPIN environment. SPIN takes system descriptions written in the PROMELA language and can check the system against a wide variety of specification types, including absence of deadlock, assertion violations and properties specified using Linear Temporal Logic (LTL). The experimentation here again uses the implementation described in Chapter 7, using ECJ to drive the HSF-SPIN model checker.

LTL is a rich language for describing properties of paths over time, including both deadlock and assertion violations. In order to check for violations of a specification expressed in LTL, automata-based model checking is a typically implemented by a model checker. Most model checkers (including SPIN) create (at least conceptually) a Büchi automaton that is equivalent to the negation of the LTL formulae. A Büchi automaton differs from regular finite automaton only in the acceptance condition. Whereas regular finite automata accept strings that end in an accepting state, Büchi automata require an infinite string that visits an accepting state *infinitely often*. The negation of the LTL formulae is used because typically the LTL formulae describe the correct behaviour of the system, therefore the negation of the formulae describe behaviours that should never happen. The created Büchi automaton accepts paths which violate the LTL specification. In the SPIN model checker, the Büchi automaton is expressed using a *never claim* and can be automatically created from the negation of LTL formulae.

9.2.2 *Connected Component Classification*

HSF-SPIN implements a particular method that can reduce computational effort whilst model checking, and that method is exploited in this work. The detection and classification of strongly connected components within a negated LTL Büchi automata equivalent can help detect irrelevant paths in the product automata. A subset S of nodes in a graph are strongly connected if for all nodes v and u in S, there is a path from v to u, and a path from u to v. A strongly connected component (SCC) is a strongly connected set of nodes that is not linked to any other by a cycle. The SCCs of a never claim can be computed in linear time, and a library for doing so is provided in HSF-SPIN. The library can also be used to classify certain types of SCC.

[Edelkamp et al., 2004] describes a number of useful types of SCC. The first is the N-SCC, an SCC with no accepting cycles. Second, the P-SCC (partial SCC) an SCC with at least once cycle that does not contain an accepting state, and one cycle that does

contain an accepting state. Finally, the F-SCC, in which all cycles contain an accepting state. For liveness specifications that contain N-SCCs and F-SCCs, once a cycle is found and it is part of an F-SCC, then an error has been discovered due to all cycles in an F-SCC being accepting. Accepting states found in N-SCCs can be ignored because no accepting cycles exist in N-SCCs, potentially saving some computational effort. Classifying and exploiting SCCs seems to result in a higher hit rate in the ACOhg algorithm [Alba and Chicano, 2008], and the same exploitation occurs in the implementation used in this work.

9.2.3 *Fitness function*

The fitness function used in this work differs slightly from the function used in the previous chapter. The pseudo-code of the ranking function can be found in Algorithm 5. The fitness functions compares two individuals, returning the "fitter" individual. The fitness function firstly prefers paths that contain errors to paths that do not contain an error. If both paths contain an error, then the shorter path is preferred. If neither path contains an error, then the decision falls back to the HSF-SPIN metric. The HSF-SPIN metric is described in Algorithm 6.

Algorithm 5 Fitness function used to rank individuals. Individuals that are "closer" to violating a property are favoured.

Require: A, B are Individuals;
  if $A.error_found \neq B.error_found$ **then**
    return IndividualWithErrorFound(A,B);
  else if $A.error_found$ **and** $B.error_found$ **then**
    return IndividualWithShortestPath(A,B);
  else
    return IndividualWithLowestHSFSPINMetric(A,B);
  end if

Algorithm 6 HSF-SPIN heuristic metric algorithm.

Require: I is an Individual;
  aggregateMetric = 0;
  for all $States\ s \in I.Path$ **do**
    aggregateMetric += s.HSFSPINMetric;
  end for
  return aggregateMetric/LengthOfPath;

The algorithm described in Algorithm 6 aggregates a heuristic value calculated by HSF-SPIN for all the states of a path, and then averages that value over the path length. This calculation gives a heuristic value for the entire path which the EDA

seeks to minimise. The choice of average over summation of the per-state heuristic was decided on after brief experimentation yielded positive results by using an average. The heuristics used in this work are described in the experimental section. The fitness function described above favours shorter counterexamples to longer ones. Because of this, if the algorithm is allowed to execute further once an error is found, there is the potential for shorter counterexamples to be found.

9.3 EXPERIMENTS IN FINDING AND OPTIMISING COUNTEREXAMPLES

In this section, we present the results of experiments in finding and optimising counterexamples in Promela models using the EDA-based approach. We compare the EDA-based approach to traditional deterministic algorithms included in the HSF-SPIN package. Rather than just running the EDA until an error is found, we allow the EDA to run for 200 generations allowing the EDA to potentially optimise the length of the counterexample. The implementation allows us to not store previously visited states in memory, keeping memory usage to a fairly constant level of around 300MB or less depending on the Promela model. This measurement includes the Java and HSF-SPIN process. To demonstrate the capabilities of the EDA-based approach, we have selected a number of models that violate safety, assertion and liveness properties.

9.3.1 *Example Promela Models*

The HSF-SPIN software distribution includes a number of different models and systems that violate a variety of different kinds of properties. In this work, we use the models listed in Table 2 to demonstrate the efficacy of the EDA-based approach. The table shows the name of the model, the maximum number of processes created, the number of Lines of Code in the model and the property the model violates.

The selected models exhibit a range of property violations which include safety and liveness properties. The first two (*phil-loop* and *phil-noloop*) are Promela implementations of the Dining Philosophers coordination problem. In both of these examples, each philosopher effectively locks/picks up the left fork, then then right fork, then releases them in that order. The "locking" is implemented by reading and writing to channels that represent the forks. The first model performs this behaviour in an infinite loop, whilst the second model terminates each philosopher after executing the fork retrieval. The *deadlock-giop* model is an implementation of the CORBA General Inter-Orb Protocol with a

Table 2: Models and the respective properties violated.

Model	Processes	LoC	Property
phil-loop*n*	$n+1$	34	Deadlock
phil-noloop*n*	$n+1$	35	Deadlock
deadlock-giop*n*	$n+6$	717	Deadlock
pots	3	453	Deadlock
leader*n*	$n+1$	117	Assertion
alter	2	64	$\Box(p \to \Diamond q) \land \Box(r \to \Diamond s)$
elev*n*	$n+3$	191	$\Box(p \to \Diamond q)$
ltlgiop*n*	$n+6$	740	$\Box(p \to \Diamond q)$
sgc	20	1001	$\Diamond p$

faulty timeout phase that can lead to deadlock. The *pots* model is an implementation of the Plain Old Telephony System which exhibits a deadlock. *leader* implements an election algorithm for nodes in a unidirectional ring configuration and violates a consistency assertion. *alter* is an implementation of the alternating bit protocol. The *elevn* models an elevator that services *n* floors in a building. The *ltlgiop* model is the same implementation as the *deadlock-giop* model but includes an LTL property which the model violates. Finally, the *sgc* simulates the operator protocol of a power planet. The *giop* series of models have been derived from an industrial standard, and the *phil*, *elev* and *giop* models are stated as having very large state spaces [Alba and Chicano, 2008].

9.3.2 *Parameters of the EDA*

The parameters used in this experiment were chosen through small-scale experimentation on the selection of Promela models, using the parameters from [Staunton and Clark, 2010] as a guide. An n-gram length of 3 was used, meaning models for 3-grams, 2-grams and 1-grams are constructed from each generation. The population size for each generation was set to 150. This means that 150 paths are sampled from the model to build each generation. The mutation parameter for these experiments is set to 0.001, meaning that on average 1 in 1000 transition choices are made randomly, disregarding the model. The elitism parameter was set to 1, meaning that the top individual from the population is copied to the next generation. In order to build the model from which the next generation is sampled, truncation selection selects the top 20% of individuals from the population. This means that the top 30 individuals from the current population are used to build the EDA/N-gram model. All individuals in the population are replaced at each generation with individuals sampled from the model. The algorithm terminates once it reaches 200 genera-

tions, allowing for the potential optimisation of counterexamples. Initially, the model is a blank model meaning that all the paths evaluated during the first generation are completely random.

9.3.3 *Experiments*

In this section, we present results from experimentation that compares the EDA-based approach against traditional deterministic approaches for finding short property violations. When searching for deadlock, we use the active processes heuristic provided by HSF-SPIN and compare against the A* algorithm. The active processes heuristic simply returns the number of active processes/enabled transitions in a particular state. If the heuristic returns 0 and the state is not a valid terminal state, then the state is a deadlocked state. When searching for the assertion violation in the leader model, we use the formula-based heuristic and A*. The formula-based heuristic is described in [Edelkamp et al., 2001b] and returns an estimate on the number of transitions from a given state s to a violation based upon satisfaction of subformulae in a specified property. And finally, we use the endstate of claim distance heuristic and the improved nested depth-first (INDFS) search when searching for LTL violations. The endstate of claim distance heuristic estimates the distance a given state is from an accepting state in the product Büchi automaton. In this scenario, INDFS uses the heuristic to determine the order in which new states are expanded during the depth-first expansion. The same heuristic is used for the EDA and for the respective deterministic approaches, with the deterministic approaches using the heuristic on individual states rather than paths. In order to gain statistically sound results, we run the EDA-based approach 100 times (with the exception of *ltlgiop20* due to time constraints) as the algorithm is probabilistic.

Table 3 shows the results from this experiment, along with some statistical comparison information. The performance with respect to a selection of measures of each algorithm (apart from the EDA) is compared with the performance of the EDA. These measures include the length of the error found, the generation in which the error was found and the number of states expanded before the error is found. These measurements are shown for the first error found by the EDA, and the best error found. In some cases, the number of states expanded may be lower than the path length to the error. This is due to path length being the number of action choices made, and more than one state can be expanded as the result of an action choice. The states measurement is provided by the HSF-SPIN model checking tool, whilst the path length is the number of actions that are chosen by the EDA algorithm to generate the path. For each measurement, the mean is given in

the left column whilst the median resides on the right. Lower and upper quartiles are given below each statistic, lower on the left, upper on the right. We opted to not show wall clocks times as the number of states expanded is a more realistic measure of the algorithms performance. The vast majority of the CPU time and memory used during the course of a run is spent expanding states, as opposed to building models/evaluating statistics. Most run times, of both the EDA and the deterministic approach, range from a 4 milliseconds to a maximum of 4 hours. Statistical comparisons are indicated with plus and minus symbols, plus being significant and minus not significant. In order to compare the EDA-based algorithm against the deterministic variants, we use the Wilcoxon signed-rank test with a significance level of $\alpha = 0.05$. We have also performed comparisons with random search.

Table 3: Results for the EDA and the respective deterministic algorithms.

Statistic	EDA		Deterministic
pots			
Errors/Runs	100/100		1/1
First error:			
Length	68.08	69	67(+)
	67	69	
Generation	19.34	9	–
	1	24.75	
States	44,292.95	22,926.5	7060(+)
	5047.5	55791.75	
Best error:			
Length	68.64	69	67(+)
	69	69	
Generation	42.15	24	–
	7.75	60.25	
States	92,286.66	58,298	7060(+)
	20,998.25	135,721.5	
phil-noloop64			
Errors/Runs	100/100		1/1
First error:			
Length	258	258	258(−)
	258	258	
Generation	6.35	6	–
	5	7	
States	324,209.42	333,886.5	258(+)
	258,062.25	381,206	
Best error:			
Length	258	258	258(−)
	258	258	
Generation	6.35	6	–
	5	7	
States	324,209.42	333,886.5	258(+)
	258,062.25	381,206	

Continued on next page...

Continued from previous page...

Statistic	EDA		Deterministic
phil-noloop128			
Errors/Runs	100/100		1/1
First error:			
Length	514	514	514(−)
	514	514	
Generation	18.93	18	–
	15	22	
States	1,859,306.63	1,762,271.5	514(+)
	1,520,985.75	2,138,105	
Best error:			
Length	514	514	514(−)
	514	514	
Generation	18.93	18	–
	15	22	
States	1,859,306.63	1,762,271.5	514(+)
	1,520,985.75	2,138,105	
phil-loop64			
Errors/Runs	100/100		1/1
First error:			
Length	611.4	594	258(+)
	517	687	
Generation	0	0	–
	0	0	
States	1,732.25	1,414.5	258(+)
	760.25	2547	
Best error:			
Length	268.84	258	258(−)
	258	258	
Generation	29.21	9	–
	8	12	
States	826,669.98	312,698.5	258(+)
	279,739.5	384,795	

Continued on next page...

Continued from previous page...

Statistic	EDA		Deterministic
phil-loop128			
Errors/Runs	100/100		1/1
First error:			
Length	1,294.12 1166	1,272 1431	514(+)
Generation	0 0	0 0	–
States	2,766.09 1,188	2,276.5 3,497.75	514(+)
Best error:			
Length	735.32 665	754 834	514(+)
Generation	68.62 2	5.5 147.25	–
States	4,267,972.45 302,913.75	708,306 8,760,617.25	514(+)
deadlock-giop20			
Errors/Runs	100/100		0/1
First error:			
Length	197.4 196	199 215	Failed
Generation	0 0	0 0	–
States	140.44 139	142 156	Failed
Best error:			
Length	141 141	141 141	Failed
Generation	0.48 0	0 1	–
States	17,553.87 6,212	18,917 25,854.25	Failed

Continued on next page...

Continued from previous page...

Statistic	EDA		Deterministic
deadlock-giop40			
Errors/Runs	100/100		0/1
First error:			
Length	283.29	279	Failed
	276	297	
Generation	0	0	–
	0	0	
States	190.46	182	Failed
	179	199	
Best error:			
Length	221	221	Failed
	221	221	
Generation	0.39	0	–
	0	1	
States	19,836.35	17,704	Failed
	5,905.75	30,796	
leader5			
Errors/Runs	100/100		1/1
First error:			
Length	69.17	69	58(+)
	65.75	73	
Generation	0	0	–
	0	0	
States	48.17	48	5,108(+)
	44.75	52	
Best error:			
Length	55	55	58(+)
	55	55	
Generation	9.52	7	–
	4	11	
States	69,607.76	53,748.5	5,108(+)
	29,533.25	79,322.25	

Continued on next page...

Continued from previous page...

Statistic	EDA		Deterministic
leader10			
Errors/Runs	100/100		1/1
First error:			
Length	124.24	125	88(+)
	119	130	
Generation	0	0	–
	0	0	
States	83.24	84	4,876,999(+)
	78	89	
Best error:			
Length	75.65	76	88(+)
	75	76	
Generation	86.86	75	–
	35.75	135.25	
States	926,073.91	800,486.5	4,876,999(+)
	386,996.25	1,440,948.55	
alter			
Errors/Runs	100/100		1/1
First error:			
Length	26.32	24	64(+)
	16	36	
Generation	0	0	–
	0	0	
States	15.31	14	32(+)
	10	20	
Best error:			
Length	8	8	64(+)
	8	8	
Generation	0	0	–
	0	0	
States	202.36	170	32(+)
	62.5	312.75	

Continued on next page...

Continued from previous page...

Statistic	EDA		Deterministic
elev20			
Errors/Runs	100/100		1/1
First error:			
Length	767.72	529	1,159(+)
	491.5	1,015.5	
Generation	0.01	0	–
	0	0	
States	4,127.04	2,978	558(+)
	1,300	5,785.5	
Best error:			
Length	378.88	381	1,159(+)
	375	381	
Generation	70.78	53	–
	25.75	105.5	
States	1,698,200.08	1,266,285	558(+)
	604,450	2,567,113.75	
elev40			
Errors/Runs	100/100		1/1
First error:			
Length	1,196.82	723	2,039(+)
	678.5	1,209.5	
Generation	0.03	0	–
	0	0	
States	8,104.37	5,494	978(+)
	2,659.25	11,691.75	
Best error:			
Length	578.98	581	2,039(+)
	575	581	
Generation	55.99	39	–
	20	78.25	
States	2,254,850.07	1,603,325	978(+)
	789,908	3,186,499.25	

Continued on next page...

Continued from previous page...

Statistic	EDA		Deterministic
ltlgiop10			
Errors/Runs	100/100		0/1
First error:			
Length	209.66	184	Failed
	160	244.75	
Generation	0	0	–
	0	0	
States	3,153.72	707.5	Failed
	366.75	5,552.5	
Best error:			
Length	73.45	74	Failed
	70	76	
Generation	111.54	113	–
	67.75	149	
States	7,291,250.65	7,297,643	Failed
	4,757,607.75	10,092,245	
ltlgiop20			
Errors/Runs	100/100		0/1
First error:			
Length	348.9	338	Failed
	260.75	436.5	
Generation	0	0	–
	0	0	
States	57,244.90	50,511	Failed
	15,577.75	85,825.5	
Best error:			
Length	96.58	96	Failed
	92	98	
Generation	105.75	117	–
	58.5	159.25	
States	44,940,817.34	49,606,760	Failed
	24,707,603.75	67,728,321.25	

Continued on next page...

Continued from previous page...

Statistic	EDA		Deterministic
sgc			
Errors/Runs	100/100		1/1
First error:			
Length	19.54	18	46(+)
	18	22	
Generation	0	0	–
	0	0	
States	4.4	4	11(+)
	4	5	
Best error:			
Length	18	18	46(+)
	18	18	
Generation	0	0	–
	0	0	
States	6.28	4	11(+)
	4	9	

Table 4: Results for the EDA and Random Search.

Statistic	EDA		Random Search	
pots				
Errors/Runs	100/100		100/100	
First error:				
Length (-)	68.08	69	68.12	69
	67	69	67	69
Generation	19.34	9	–	–
	1	24.75		
States (-)	44,292.95	22,926.5	22,284.23	16,306
	5047.5	55791.75	5,589.5	31,081.75
Best error:				
Length (+)	68.64	69	68.32	69
	69	69	67	69
Generation	42.15	24	–	–
	7.75	60.25	–	–
States (+)	92,286.66	58,298	45,845.55	29,342
	20,998.25	135,721.5	12,270.25	63,006.5
phil-noloop64				
Errors/Runs	100/100		0/100	
First error:				
Length (+)	258	258	Failed	
	258	258		
Generation	6.35	6	–	–
	5	7		
States (+)	324,209.42	333,886.5	Failed	
	258,062.25	381,206		
Best error:				
Length (+)	258	258	Failed	
	258	258		
Generation	6.35	6	–	–
	5	7		
States (+)	324,209.42	333,886.5	Failed	
	258,062.25	381,206		

Continued on next page...

Continued from previous page...

Statistic	EDA		Random Search	
phil-noloop*128*				
Errors/Runs	100/100		0/100	
First error:				
Length (+)	514	514	Failed	
	514	514		
Generation	18.93	18	–	–
	15	22		
States (+)	1,859,306.63	1,762,271.5	Failed	
	1,520,985.75	2,138,105		
Best error:				
Length (+)	514	514	Failed	
	514	514		
Generation	18.93	18	–	–
	15	22		
States (+)	1,859,306.63	1,762,271.5		
	1,520,985.75	2,138,105	Failed	
phil-loop*64*				
Errors/Runs	100/100		100/100	
First error:				
Length (-)	611.4	594	615.72	588
	517	687	534	682
Generation	0	0	–	–
	0	0		
States (-)	1,732.25	1,414.5	1,506.09	1,055.5
	760.25	2547	599.5	2,121.5
Best error:				
Length (+)	268.84	258	319.16	318
	258	258	313	326
Generation	29.21	9	–	–
	8	12		
States (+)	826,669.98	312,698.5	5,827,332.94	5,042,235
	279,739.5	384,795	2,465,755.75	9,407,784

Continued on next page...

Continued from previous page...

Statistic	EDA		Random Search	
phil-loop128				
Errors/Runs	100/100		100/100	
First error:				
Length (-)	1,294.12 1166	1,272 1431	1,272.12 1133	1,278 1391
Generation	0 0	0 0	—	—
States (-)	2,766.09 1,188	2,276.5 3,497.75	3,737.85 1,676.5	2,469 4,280.25
Best error:				
Length (-)	735.32 665	754 834	763.76 746	772 786
Generation	68.62 2	5.5 147.25	—	—
States (+)	4,267,972.45 302,913.75	708,306 8,760,617.25	12,079,792.36 4,085,110	12,584,058.5 18,707,884.75
deadlock-giop20				
Errors/Runs	100/100		100/100	
First error:				
Length (-)	197.4 196	199 215	203.59 196	199 218
Generation	0 0	0 0	—	—
States (-)	140.44 139	142 156	145.98 139	142 159
Best error:				
Length (-)	141 141	141 141	141 141	141 141
Generation	0.48 0	0 1	—	—
States (-)	17,553.87 6,212	18,917 25,854.25	22,896.67 5,583.5	15,647 31,655

Continued on next page...

Continued from previous page...

Statistic	EDA		Random Search	
deadlock-giop40				
Errors/Runs	100/100		100/100	
First error:				
Length (-)	283.29	279	286.79	279
	276	297	276	298
Generation	0	0	—	—
	0	0		
States (-)	190.46	182	190.73	182
	179	199	179	199
Best error:				
Length (-)	221	221	221	221
	221	221	221	221
Generation	0.39	0	—	—
	0	1		
States (+)	19,836.35	17,704	30,456.97	20,623.5
	5,905.75	30,796	10,137.5	39,338.75
leader5				
Errors/Runs	100/100		100/100	
First error:				
Length (-)	69.17	69	69.59	69
	65.75	73	66	73
Generation	0	0	—	—
	0	0		
States (-)	48.17	48	48.59	48
	44.75	52	45	52
Best error:				
Length (+)	55	55	55.12	55
	55	55	55	55
Generation	9.52	7	—	—
	4	11		
States (+)	69,607.76	53,748.5	422,239.46	298,308
	29,533.25	79,322.25	98,509.75	673,580.5

Continued on next page...

Continued from previous page...

Statistic	EDA		Random Search	
	leader10			
Errors/Runs	100/100		100/100	
First error:				
Length (-)	124.24 119	125 130	124.91 118.75	126 131
Generation	0 0	0 0	–	–
States (-)	83.24 78	84 89	83.91 77.75	85 90
Best error:				
Length (+)	75.65 75	76 76	91.12 90	91.5 93
Generation	86.86 35.75	75 135.25	–	–
States (-)	926,073.91 386,996.25	800,486.5 1,440,948.55	1,022,467.47 444,148.25	903,307 1,582,721.75
	alter			
Errors/Runs	100/100		100/100	
First error:				
Length (-)	26.32 16	24 36	28.46 18	30 36
Generation	0 0	0 0	–	–
States (-)	15.31 10	14 20	16.41 11	17 20
Best error:				
Length (-)	8 8	8 8	8 8	8 8
Generation	0 0	0 0	–	–
States (-)	202.36 62.5	170 312.75	190.52 55	125 230.45

Continued on next page...

Continued from previous page...

Statistic	EDA		Random Search	
elev20				
Errors/Runs	100/100		100/100	
First error:				
Length (-)	767.72	529	723.78	526
	491.5	1,015.5	461	708.5
Generation	0.01	0	—	—
	0	0		
States (-)	4,127.04	2,978	4,046.71	3,223
	1,300	5,785.5	1,257	5,546
Best error:				
Length (+)	378.88	381	388.76	389
	375	381	389	389
Generation	70.78	53	—	—
	25.75	105.5		
States (+)	1,698,200.08	1,266,285	883,742.55	680,025.5
	604,450	2,567,113.75	401,169.25	1,181,546
elev40				
Errors/Runs	100/100		100/100	
First error:				
Length (-)	1,196.82	723	1,159.6	723
	678.5	1,209.5	664	917
Generation	0.03	0	—	—
	0	0		
States (-)	8,104.37	5,494	7,182.44	4,811
	2,659.25	11,691.75	2,335.5	9,248
Best error:				
Length (+)	578.98	581	588.76	589
	575	581	589	589
Generation	55.99	39	—	—
	20	78.25		
States (+)	2,254,850.07	1,603,325	1,599,969.86	1,204,717
	789,908	3,186,499.25	481,500.75	2,413,688

Continued on next page...

Continued from previous page...

Statistic	EDA		Random Search	
		ltlgiop10		
Errors/Runs	100/100		100/100	
First error:				
Length (-)	209.66	184	217.08	188
	160	244.75	159.75	249
Generation	0	0	–	–
	0	0		
States (+)	3,153.72	707.5	2,742.45	839.5
	366.75	5,552.5	405.25	5641
Best error:				
Length (+)	73.45	74	114.64	114
	70	76	114	116
Generation	111.54	113	–	–
	67.75	149		
States (+)	7,291,250.65	7,297,643	2,714,784.96	2,648,932
	4,757,607.75	10,092,245	1,426,870.5	4,067,802
		ltlgiop20		
Errors/Runs	100/100		100/100	
First error:				
Length (-)	348.9	338	330.44	301.5
	260.75	436.5	224.75	407.25
Generation	0	0	–	–
	0	0		
States (-)	57,244.90	50,511	53,843.89	43,084
	15,577.75	85,825.5	10,389.5	76,062
Best error:				
Length (+)	96.58	96	175.77	176
	92	98	174	178
Generation	105.75	117	–	–
	58.5	159.25		
States (+)	44,940,817.34	49,606,760	27,733,336.12	22,856,985
	24,707,603.75	67,728,321.25	11,932,572.25	40,329,617.5

Continued on next page...

Continued from previous page...

Statistic	EDA		Random Search	
sgc				
Errors/Runs	100/100		100/100	
First error:				
Length (-)	19.54 18	18 22	20.38 18	18 22
Generation	0 0	0 0	—	—
States (-)	4.4 4	4 5	4.64 4	4 5
Best error:				
Length (-)	18 18	18 18	18 18	18 18
Generation	0 0	0 0	—	—
States (-)	6.28 4	4 9	7.25 4	4 9

9.3.4 *Discussion of Results*

Initial observations from the results table reveal that the EDA is the only algorithm to achieve a 100% hit rate on all of the sample models. The deterministic approaches fail to find an error on the *deadlock-giop* and the *ltlgiop* examples, either hitting a 64GB memory limit or going over a time limit of 24 hours. Random search also failed to discover an error on the *phil-noloop* model family. These models are particularly large, and the results show the ability of the EDA to focus the search on promising areas of the state space revealing errors by expanding fewer states, and consequently using less memory and CPU time. In the *phil-noloop* model, there is only one path to get to the error (always take left), and this explains the similarity between first and best paths found by both the EDA and the deterministic algorithms. The 100% hit rate on all the test cases shows that the EDA-based approach is a very promising algorithm for discovering counterexamples in large state spaces, especially with respect to robustness and sensitivity to the state space explosion problem. The ACOhg algorithm in [Chicano and Alba, 2008a; Alba and Chicano, 2008] shows a less than 100% hit rate on the *ltlgiop* and philosophers models, suggesting that the EDA-based approach may have an advantage on some of the larger models.

In the majority of test cases the EDA found a statistically significantly shorter counterexample than the deterministic approach, with the exception of the *pots* model and the larger *phil-loop* model. However, the difference in the length of the best paths found is a matter of 1 or 2 states on the *pots* model. The larger difference on the *phil-loop128* model is likely explained by the way the heuristic is constructed. By using the average of the heuristic values of states in a path as the heuristic value for a path in the EDA, the EDA may favour longer paths in some instances. For example, a longer path of low heuristic values may be favoured over a short path with high heuristic values. This result shows that the EDA can be sensitive to the heuristic used, and that some models may require a carefully thought out heuristic in order for the EDA to be effective. This effect is only evident in the *phil-loop** models, and in the majority of cases does not seem to hinder the EDA On the *phil-loop** models, the EDA is beating random search by either finding a shorter counter example, or expanding fewer states on average. The ability to optimise counterexamples makes the EDA an appealing approach, as shorter counterexamples remove superfluous information enabling a software tester/model checking practitioner to focus on the underlying cause of an issue. To optimise the counterexample, however, more states must be expanded in order to learn a model that reflects shorter counterexamples. This is shown in the results table on the majority of the models with the exception of the *leader* model, where the EDA managed to find a shorter error by expanding far fewer states and therefore using less memory and CPU time.

In some cases, the number of states required to find the first error in an EDA run is either less than or statistically insignificantly greater than or equal to the results from the deterministic algorithms. In the cases where the EDA does expand more states to find the first error, mere milliseconds are added to the EDA search time with respect to the deterministic search time. However, this is not the case on the *phil-noloop* and *pots* models, where the states required to find the first error is far greater than the deterministic algorithm. However in the vast majority of cases, the EDA found the first error by expanding a reasonable amount of states. The results shown in this experiment show that the EDA is capable of finding errors in a short space of time, even in models (the *giop* based models) that are deemed very large by previous work [Alba and Chicano, 2008].

In the majority of the tests, one can observe that the number of generations required to find the first error is on average zero. This indicates that the EDA algorithm found the first error using random search alone due to the EDA starting with a blank model. When given a blank model, all transitions are chosen at random. It is the case that random search is also able to find an error

with a 100% hit rate in all of the test cases with the exception of the *phil-noloop* model. However in all but three cases (the *elevn*, *alter* and *sgc* models), the EDA was able to shorten the error statistically significantly when compared to the best error found by random search. The fact that random search is able to find an error with a 100% hit rate in the majority of the tests, as well being statistically equivalent to the EDA when faced with the *elev*, *alter* and *sgc* models suggests that a comparison with random search is necessary when evaluating probabilistic algorithms in the model checking domain. It also suggests that as part of a benchmark suite of tests for model checking algorithms, relevant models that defeat random search must be found. We can state, however, that the EDA-based approach achieves a 100% hit rate, whilst being able to more effectively optimise counterexamples in the majority of our selected benchmark tests when compared with random search and the most prominent deterministic algorithms. Due to the fact that the first error is often found in the first generation, the HSF-SPIN aspect of the fitness function is seldom exercised. Therefore, the path shortening pressure governs the selection process in the majority of the scenarios. This explains the negligible difference I observed in preliminary testing between the averaging and summation based fitness functions.

On some of the tests, the EDA finds the best error in generation 0 100% of the time. The models in which this occurs are the *sgc* and *alter* models. The best error found in these systems is the shortest possible error that can be found. This indicates that the model is trivial for any sensible algorithm, since the shortest possible error can be found using random search without any guidance/learning. These trivial models are likely not good candidates for future model checking benchmarks and we suggest that the use of these models should be avoided.

9.4 SUMMARY

In this chapter, I have shown how the proposed algorithm can not only find counterexamples to a variety of property types in systems derived from industrial code, but can also optimise the quality of said errors. I have compared the algorithm against prominent deterministic approaches using similar heuristics, showing that the EDA can find shorter counterexamples in the majority of the test cases. In two cases, we have shown that the EDA can find errors where traditional approaches fail due to exhaustion of resources, showing that the EDA can effectively focus the effort of the search. The optimisation of counterexamples is a key feature that model checking practitioners could find useful. Current prominent mechanisms, namely the improved nested depth-first search algorithm, do not provide this ability. Although

they can be extended to potentially do so, it is not clear how well an extended nested depth-first search algorithm would scale on large state spaces. The EDA-based approach shows promise with respect to large state spaces, with the results of the experiment showing robustness when tested with a variety of models. With respect to the state of the art in the domain, the EDA-based algorithm has a higher success rate than ACO on a number of models, including the *giop* model and the Dining Philosopher models.

In addition to the work of the previous chapter showing the ability to find deadlock in concurrent systems, I have shown that the EDA can find violations of assertions and liveness properties, a key step in demonstrating the efficacy of an EDA-based approach. When proposing the algorithm, I was initially concerned that the EDA can only be effective on highly symmetric toy problems such as the Dining Philosophers problem. However, the results show that the EDA can find errors in large asymmetric systems that are derived from industrial scenarios. The results suggest that an EDA-based approach to searching a state space can be used to find an error of any kind provided a suitable heuristic can be defined. In addition to this, the EDA could potentially be used as a counterexample optimiser as a supplement to another algorithm known to be effective on a particular model. For instance, one could use a traditional algorithm to find an error in the *phil-loop* series of models, and then use the EDA-based algorithm with a state-distance heuristic to find a shorter counterexample.

An often cited potential feature of EDAs is the ability to gain information or insight by examining the models created at each generation during the run. A typical Evolutionary Algorithm produces a sequence of bags of solution to a problem, whilst an EDA additionally produces a sequence of models. The analysis or reuse of this modelling information may yield savings in computational effort when used for model checking, giving the algorithmic proposal of this thesis a unique advantage over existing model checking mechanisms. In the next chapter, I set out to explore this aspect of the algorithmic proposal and how reuse of model information can save effort when verifying large industrially relevant systems.

10 SCALING TO LARGE SYSTEMS

10.1 INTRODUCTION

In the previous two chapters, I have established the proposed algorithm as a credible contender in the space of algorithms that detect counterexamples in large transition systems. As well as having a robust detection rate when compared to all other algorithms, the EDA-based approach can optimise the quality of the solutions found. By finding shorter counterexamples, practitioners of model checking or concurrent software developers can obtain concise information about an error leading to more efficient debugging.

An often cited advantage of EDAs over other metaheuristic approaches is the potential for the model information created during the run of an EDA to be used to improve the performance of the EDA with respect to some criteria [Pelikan et al., 2002; Očenášek, 2002]. Given that the EDA-based model checking algorithm I have proposed in this thesis is the only EDA-based approach available to the model checking community, this capability could give the EDA a unique advantage over all other algorithms in the model checking domain.

In this chapter, I will show how the proposed algorithm and the use of modelling information can be used to save large amounts of computational effort in practical software engineering scenarios, and show empirically the potential in on of those scenarios. The work in this chapter is based on [Staunton and Clark, 2011a].

10.2 MODEL REUSE

A typical GA run will produce a sequence of bags of solutions to a target problem. In the case of detecting concurrent faults, this will amount to sets of paths within the transition system which may or may not lead to an error. Whilst it is possible to use some of these solutions to seed future runs, even a small change to the target problem could destroy the applicability of these solutions. Similarly, Ant Colony Optimisation constructs pheromone trails that may lose relevance once the target problem has been changed. The EDA-based model checking technique, on the other hand, produces a strategy for navigating a transition system in addition to structures that represent solutions. We believe that with some simple steps the models can be used to reduce computational effort on future instances of similar problems. Outlined below

are a number of potential practical scenarios for model reuse that could greatly reduce computational effort.

10.2.1 *Reuse during Debugging*

The first proposed scenario for model reuse is during the testing/debugging phases of the development life cycle. It is plausible that during any execution of the EDA a constructed model can represent a strategy for finding not just a single error but multiple errors, as well as "problem areas" of the state space. If a single error has been found in an execution e on the ith revision of the system, e could be halted in order to fix the bug. This would create the $i+1$th revision. Once the error has been corrected, execution can continue using the model constructed in the last generation of e, labelled m. It is also plausible that erroneous actions from the ith revision of the system in m can be mapped to the corrected actions in the revised system, allowing for the EDA to focus on these areas initially. This may potentially eliminate computation if errors still exist in the area of the state space where the initial error was found. Equally, if a practitioner is confident that the error has been corrected, the area of the state space can be added to a tabu list, allowing the EDA to focus effort elsewhere.

10.2.2 *Reuse during Refinement*

Another potential scenario for reusing models is during refinement. During implementation, versions of the system are refined to meet various ends, including performance improvements and bug fixing. Refinements may increase the size of the transition system enormously. In order to combat this, it is plausible that the EDA can be executed on the more abstract version of the system/software in order to determine potential problem areas that rank highly with the fitness function. The models constructed during these initial explorations can then potentially be used to explore future refinements of the system. If refining the system increases the state space size significantly, then significant computational effort could be spared. There is also the potential to map actions from the abstract system to actions in the refined system, potentially increasing the saving of computational effort.

10.2.3 *Reuse when tackling Problem Families*

In my published work [Staunton and Clark, 2010, 2011b] and in earlier chapters I have speculated that the EDA-based technique is learning effective strategies for navigating the state spaces of

problem families, rather than just the problem instance itself. A problem family is simply a system which can be scaled up and down respective to some parameter, typically the number of processes/threads in the system/software. This opens up the possibility of models learned whilst tackling small instances of a problem family can be used to save effort whilst finding errors in larger instances. Finding the same error in a number of instances of a problem family can provide additional debugging information, potentially shortening the debugging life cycle. Problem families arise frequently in practical situations [Clarke et al., 2000], in both hardware and software systems.

In this work, we provide empirical evidence showing how computational effort can be saved when detecting errors in problem families. Extra information can be gained from detecting faults in varying instances of a problem family, and this can save time and ultimately lower costs when building and debugging practical systems.

10.3 EXPERIMENTATION WITH PROBLEM FAMILIES

10.3.1 *Sample Models*

In order to demonstrate the ability for model reuse, we aim to show that the EDA can learn structures in three problem families. The three test cases used in this work are listed in Table 5. The test cases are diverse in the system description as well as the property under test. The first is a non-looping implementation of the Dining Philosophers system.

Table 5: Models and the respective properties violated.

Model	Processes	LoC	Property
Dining Philosopher No Loop$n$	$n+1$	35	Deadlock
Leader n	$n+1$	117	Assertion
GIOP n	$n+6$	740	$\Box(p \rightarrow \Diamond q)$

For the second test case, a leader election system is modelled by the Leader model. In this model, n processes must agree on a common leader process in a unidirectional ring configuration. The faulty protocol used to establish a leader allows members of the ring to disagree on a leader. An assertion represents this specification, and the EDA algorithm will be aiming to find violations of this assertion. The final test case implements the CORBA Global Inter-ORB protocol (GIOP) with n clients using a single server configuration. The system violates a property specified in LTL. This model is particularly large, and is reported as difficult to find errors in for large n (large n being $n >= 10$) [Alba and Chicano, 2008; Staunton and Clark, 2011b]

In this set of examples, we have a deadlock, an assertion and an LTL violation in order to show the applicability of the technique to errors other than deadlock. Whilst the Dining Philosophers problem is a well studied toy problem, the GIOP model is derived from an industrial source, adding credibility to using this approach in industrial scenarios. Finally, whilst the Dining Philosophers model is symmetrical in nature (all the processes have the same description), the GIOP model is asymmetric, adding further weight to the empirical argument that the technique could be effective in industrial scenarios.

10.3.2 *Heuristics*

The fitness function detailed in Algorithm 7 is used to rank solutions and makes use of heuristic information implemented in the HSF-SPIN framework [Edelkamp et al., 2001a]. The heuristics implemented in HSF-SPIN give information about a single state only. Algorithm 8 combines the information from the individual states in a path to give a heuristic measure for the entire path. This is done by simply summing the heuristic information of all the states along the path. The algorithm aims to minimise this cumulative total, whilst favouring shorter paths and paths with errors. The same heuristic is used on all of the runs in this work.

Algorithm 7 Fitness function used to rank individuals. Individuals that are "closer" to violating a property are favoured.

Require: A, B are Individuals;
  if $A.error_found \neq B.error_found$ **then**
    return IndividualWithErrorFound(A,B);
  else if $A.error_found$ **and** $B.error_found$ **then**
    return IndividualWithShortestPath(A,B);
  else
    return IndividualWithLowestHSFSPINMetric(A,B);
  end if

Algorithm 8 HSF-SPIN heuristic metric algorithm.

Require: I is an Individual;
  aggregateMetric = 0;
  for all $States\ s \in I.Path$ **do**
    aggregateMetric += s.HSFSPINMetric;
  end for
  return aggregateMetric;

HSF-SPIN implements a variety of heuristics which can be used on various types of properties. In Algorithm 6, the *s.HSFSPINMetric* is calculated using the following heuristics. We use the active pro-

cesses heuristic for finding deadlock in the Dining Philosophers model [Edelkamp et al., 2001b,a]. The active processes heuristic, when given a state, returns the number of processes that can progress in that state. When looking for assertion violations in the Leader model, the formula-based heuristic is used [Edelkamp et al., 2001b,a]. The formula-based heuristic estimates how close a state is to violating a formula by examining the satisfaction of constituent sub-formulae in that state. And finally, when searching for LTL formulae violations, we use the HSF-SPIN distance-to-endstate heuristic [Edelkamp et al., 2001b,a]. The distance-to-endstate heuristic estimates how many transitions a state is away from the end state of a product Büchi automaton, a structure used when verifying LTL properties and implemented in HSF-SPIN. Put simply, the distance-to-endstate heuristic estimates how far a state is away from violating the LTL specification.

10.3.3 *Parameters*

The parameters for all of the executions are derived from small scale empirical work, as well as experimental results from our previous publications [Staunton and Clark, 2010, 2011b]. We expect that these parameters may work well on a wide variety of problems, but some problems may need extra tuning. An n-gram length of 3 was used, meaning models for 3-grams, 2-grams and 1-grams are constructed from each generation. The model is completely destroyed and rebuilt from the selected individuals at each generation. This also means that the reused model is essentially a seed model for the runs on larger instances. The population size for each generation was set to 150. This means that 150 paths are sampled from the model to build each generation. The mutation parameter for these experiments is set to 0.001, meaning that on average 1 in 1000 transition choices are made randomly, disregarding the model. The elitism parameter was set to 1, meaning that the top individual from the population is copied to the next generation. In order to build the model from which the next generation is sampled, truncation selection selects the top 20% of individuals from the population. This means that the top 30 individuals from the current population are used to build the EDA/N-gram model. All individuals in the population are replaced at each generation with individuals sampled from the model. The algorithm terminates once it reaches 200 generations, allowing for the potential optimisation of counterexamples. Initially, the model is a blank model meaning that all the paths evaluated during the first generation are completely random.

10.3.4 *Smaller Instances*

In order to learn strategies that can be used on any instance of a particular problem family, we ran the EDA algorithm on a small instance of each problem family. For the Dining Philosophers problem family, a small instance is a system with 32 philosophers. For the Leader model, we use a unidirectional ring with 3 members. And finally, for the GIOP model, we use a single server 2 client configuration. The smaller instance numbers seemed at the time to represent realistic values for debugging the respective systems. Additionally, for the philosophers model, a number was chosen that couldn't be discovered trivially by random search. For each model, we allow the algorithm to run for a fixed number of generations, allowing execution to continue if an error is found in order to optimise the model and find shorter counterexamples. The model constructed from the final generation of a single execution of the EDA is the model used in the subsequent executions on the larger instances. The model is simply serialised out to a file to be used as input to a future run. At this stage, there is the possibility of inspecting the model in order to make improvements. In this work however, the model is used verbatim in the execution on the larger model. Models from various runs can potentially be archived for use in future work. Some measurements from these initial runs can be found in Table 6. We have proven empirically in earlier papers [Staunton and Clark, 2010, 2011b] that the EDA is capable of consistently finding good strategies in the time scales shown in Table 6. The numbers below the First Error header are numbers relating to the first error found during the execution. The best error table shows the numbers related to the shortest error found.

10.3.5 *Larger Instances*

The larger instances of the problem families consist of the following. For the Dining Philosopher problem family, we used a 128 philosopher system. For the Leader system, we used a unidirectional ring with 10 voters. Unfortunately it is not possible to scale this model further due to implementation limitations on the part of the system, not the EDA. And finally, for the GIOP system, an instance with a single server and 20 clients is used. The sizes of both the Dining Philosopher system and the GIOP system were chosen due to the availability of measurements on those systems without model reuse. We are confident that the technique will scale beyond these numbers, but due to time constraints we could not explore larger instances.

Table 6: Measurements from the initial runs

Measurement	Dining Philosophers	Leader	GIOP
First error:			
Generations	3	0	0
Path Length	34	35	59
States	73,058	35	729
Time	27.45s	0.3s	0.3s
Best error:			
Generations	3	0	17
Path Length	34	32	21
States	73,058	2,080	80,478
Time	27.45s	0.63s	3m8s
Total for run:			
Generations	50	200	200
States	1,150,400	1,040,495	931,691
Time	13m30s	19m47s	37m33s

The statistics shown in Tables 7, 8 and 9 are taken from 100 executions on the Dining Philosopher, Leader and GIOP systems respectively. Each of the 100 runs used the single model constructed in the initial run stage described in Section 10.3.4. Any statistics in the "n/m" format are stating the "median/mean". In order to compare total amounts of computation, the "With Model Reuse" column in the tables includes the computation up to the best error found in the initial runs. We argue that this is a fair definition of the computation involved in building a model initially because practitioners are likely to limit the number of generations to find a good enough error, especially if the EDA-based technique is used regularly during a development life cycle. The "Without Initial Run" column shows the numbers of the reuse run only, without the computation of the strategy on the smaller instance. Statistical comparisons with the results obtained without model reuse are indicated with plus (significant difference) and minus (insignificant difference) symbols. In order to compare the model reuse runs against the non-reuse runs, we use the Wilcoxon rank-sum test with a significance level of $\alpha = 0.05$.

The results in Table 7 show statistics for the Dining Philosopher problem family. In the Dining Philosopher system, there is a single error. The error can be reached in multiple ways but is always at the same depth/path length. This explains the similarity

Table 7: Measurements from the model reuse runs on the Dining Philosophers 128 system

Measurement	Without Model Reuse	With Model Reuse	Without Initial Run
First error:			
Generations	19/19.4(+)	**3/3**	**0/0**
Path Length	**130/130(-)**	**130/130**	**130/130**
States	1,831,394/1,898,568.21(+)	**73,831/74,281.1**	**773/1,223.1**
Time	47m24s/1h14m32s(+)	**29.572s/30.057s**	**2.122s/2.606s**
Best error:			
Generations	19/19.4(+)	**3/3**	**0/0**
Path Length	**130/130(-)**	**130/130**	**130/130**
States	1,831,394/1,898,568.21(+)	**73,831/74,281.1**	**773/1,223.1**
Time	47m24s/1h14m32s(+)	**29.572s/30.057s**	**2.122s/2.606s**

Table 8: Measurements from the model reuse runs on the Leader 10 system

Measurement	Without Model Reuse	With Model Reuse	Without Initial Run
First error:			
Generations	**0/0(-)**	**0/0**	**0/0**
Path Length	84/82.75(+)	**71/71.21**	**71/71.21**
States	**84/82.75(+)**	2,151/2,151.21	**71/71.21**
Time	**0.239s/0.622s(+)**	1.127s/1.606s	0.497s/0.976s
Best error:			
Generations	17/20.26(-)	**15/19.23**	**15/19.23**
Path Length	**36/35.45(-)**	36/35.47	36/35.47
States	193,616/225,050.01(-)	**163,429/209,150.82**	**161,349/207,070.82**
Time	22m51s/25m57s(+)	**4m7s/5m19s**	**4m6s/5m18s**

between the first and best results. From the numbers achieved, it is clear that model reuse can have a huge impact on the amount of computational effort required to find errors in the larger instance. The mean time to discover an error is reduced by over 99%. This means that rather than wait an hour for additional information regarding the error, information can be obtained in a mere 30 seconds, potentially reducing time spent in the debugging cycle substantially in this case. We expected a large gain on the Dining Philosopher family as it is a symmetrical problem. The strategy to finding an error in the Dining Philosopher is trivial, "Always choose the action that is Pickup the Left Fork".

The results in Table 8 show statistics for the Leader election problem family. In this problem family, the results are less impressive than that of the Dining Philosopher family. We attribute this to the fact that the EDA can find a short counterexample with little computation, often in the first generation before any strategy building has taken place. This suggests that the model is trivial and does not require mechanisms to reduce computational effort. However, we still obtain a significant speed increase in

Table 9: Measurements from the model reuse runs on the GIOP 20 system

Measurement	Without Model Reuse	With Model Reuse	Without Initial Run
First error:			
Generations	**0/0.01(+)**	**17/17**	**0/0**
Path Length	132/150.09(+)	**61/73.37**	**61/73.37**
States	**40,421/60,681.01(+)**	90,773/98,194.14	**10,295/17,716.14**
Time	**1m26s/2m1s(+)**	3m28s/3m46s	**19.56s/38.017s**
Best error:			
Generations	30/28.71(+)	**20/28.21**	**3/11.21**
Path Length	31/31.21+)	**26/25.6**	**26/25.6**
States	13,068,139/12,337,306(+)	**1,495,644/4,942,260.07**	**1,415,166/4,861,782.07**
Time	6h47m16s/8h13m24s(+)	**57m34s/3h12m14s**	**54m26s/3h9m6s**

terms of time spent searching the transition system. We attribute this to the EDA exploring a narrower area of the state space on the larger instance due to the initial strategy constructed from the smaller instance. This may avoid expanding useless parts of the search space, resulting in a reduction in CPU and memory usage.

The most impressive results are listed in Table 9 for the GIOP problem family. We expected poorer results on this model due to the description of the system being asymmetric. However, not only is a 62% reduction of mean time in finding a best error achieved (86% reduction in the median time), the quality of the solutions discovered are also improved. The improvement in the path length of the solutions found allow a practitioner to instantly assess the properties of the error. In this instance, the paths are of a similar length meaning it is highly likely that only a subset of the processes in the system are required to cause the error. If all 20 clients were involved, you can expect a substantial increase in the path length over the 2 client model. The Dining Philosopher system, for instance, requires that all processes perform actions to cause a deadlock, and this is reflected by the increase in path length from the 32 philosopher system to the 128 philosopher system. Model reuse and the ability of the EDA to find and optimise counterexamples efficiently [Staunton and Clark, 2010, 2011b] could make the EDA-based technique a valuable tool for practitioners, as useful information such as this could be revealed along with other insights. Furthermore, the practitioner could gain this information with zero effort, as there is the potential for this approach to be automated.

The strategy for navigating to an error state in the Dining Philosophers example is trivial, "Always choose the action Pickup the Left Fork", and the algorithmic proposal of this thesis seems to be discovering this strategy efficiently with regards to computational resources. Reusing this discovered strategy on the larger instance

reduces the computational effort required to discover an error in the larger system, suggesting that the strategy constructed is effective and concise. It is plausible that the algorithm is also doing the same for the other two systems under test in this empirical work, due to the fact that effort is saved when examining the larger instances of the respective systems. The fact that the strategies are reusable even as the underlying system changes lends weight to the plausibility of the other scenarios outlined as part of this chapter. For instance, if we can discover a strategy for finding errors in a given system, one can then use that strategy as part of regression testing in future iterations of the system, potentially saving large amounts of time and effort.

10.4 SUMMARY

To summarise, presented above is an approach for saving computational and manual effort when building and debugging concurrent systems using the EDA-based technique proposed in this thesis. This is achieved by reusing information, specifically information from the models constructed, from an earlier execution to aid the search in a future execution. The analysis and reuse of modelling information learned by EDAs is an often cited advantage [Pelikan et al., 2002], and this advantage has been demonstrated in a practical scenario. Using this new approach, it is possible to save computational effort when analysing problem families, and other scenarios have been described where effort could potentially be saved. Our results show that information can been gained using an insignificant amount of additional computational resources. This information can yield insights that can save time in the debugging phase, which could ultimately lower development costs. The scenario tested in this paper could potentially be automated, meaning no manual effort would be required to gain additional information. At the time of writing, to the best of my knowledge, this is the first application of EDA model analysis/reuse in the SBSE domain.

I believe that there is ample scope for further work in this area. The scenarios for model reuse described and tested in this work are likely a subset of what is possible. There may well be other scenarios in which this work could be beneficial, and not just in the concurrent software testing domain. N-gram GP is essentially a sequence modelling algorithm, and approaches like this could be used wherever the solution space can be represented as a sequence. This could include problems that can be couched as graph search. We feel that the algorithm proposed in this thesis can be applied to stress testing. In this application domain, the EDA could be used to learn problematic sequences that cause the performance of systems to degrade or indeed completely fail.

Part V

CONCLUSION

11 CONCLUSION

11.1 INTRODUCTION

This thesis collates empirical work and an algorithmic proposal that helps determine the validity of a hypothesis, originally set out in Chapter 2. The hypothesis stated that Estimation of Distribution Algorithms are an effective mechanism for detecting and debugging concurrent faults, and comprises are three sub-hypotheses that lend weight to the overarching hypothesis. For each sub-hypothesis, I have included a chapter of related empirical work and drawn conclusions from that empirical work. The empirical work has shown that the algorithm proposed as part of this thesis can detect a range of concurrent fault types in a variety of systems, and also provide additional useful information with regards to debugging and potentially maintaining the system going forward.

This chapter will summarise the discussion on each sub-hypothesis, briefly outlining the empirical work of this thesis and how it reflects upon the overarching hypothesis. Following this, I will outline the novel contributions of the thesis, discussing how I have potentially improved upon the state of the art. The limitations of the research will then be outlined, followed by a brief overview of how the algorithm may be refined and applied in other problem domains.

11.2 HYPOTHESES

11.2.1 *Finding Faults in Mainstream Language Code*

Hypothesis: *Estimation of Distribution Algorithms are an effective mechanism for detecting faults in systems described by mainstream languages.*

The empirical work in Chapter 8 addressed this sub-hypothesis by applying the algorithmic proposal of this thesis to an example problem written in Java, a language widely-used to describe industrial systems. Using a Dining Philosophers system with varying numbers of concurrent philosophers, the EDA achieved a 100% success rate in detecting deadlock, starting from scratch with a blank model. The experiments performed in Chapter 8 indicated that the algorithm scaled well as the number philosophers in the system increased (See Figure 21).

Whilst other metaheuristic mechanisms can also detect concurrent faults in Java programs, the EDA is the first to achieve a 100% success rate in detecting deadlock. Unfortunately, in the case of analysis of Java programs, there are few statistics that can be directly compared between approaches, apart from the detection rate. In Chicano et al. [2011], experiments show only beam search achieving a 100% success rate on this particular variant of the Dining Philosopher problem with large numbers of philosophers (16+). Other prominent metaheuristic search techniques, such as Ant Colony Optimisation and Genetic Algorithms, start to fail as the number of philosophers increases beyond 12.

With regards to the sub-hypothesis, the empirical work in Chapter 8 shows that indeed EDAs can join other metaheuristic techniques in claiming the ability to detect faults in concurrent software written Java, and likely other mainstream languages such as C++ and the .NET family. With the high success rate on during experimentation, this lends weight to the validity of the sub-hypothesis.

11.2.2 *Finding and Optimising Wide Ranges of Faults in Complex Systems*

Hypothesis: *Estimation of Distribution Algorithms can find not only faults in complex industrial systems, but also optimise them.*

The experimental work in Chapter 9 addressed this sub-hypothesis and involved analysing transition systems that included those derived from industrial settings. In addition to detecting deadlock, the EDA was shown to be able to detect violations of properties expressed in Linear Temporal Logic. The algorithm was configured to continue searching for shorter paths to errors once an error is detected, and the fitness function included a selection pressure that favoured shorter paths. The results from the experimentation in Chapter 9 showed that on the systems under test, EDAs can find and optimise a variety of fault types, and do so with a higher success rate than state of the art metaheuristic mechanisms. This evidence lends weight to the statement that EDAs can find and optimise faults found in complex industrial scenarios. The shorter paths to errors found by the EDA are of a higher quality to a practitioner looking to debug an issue in a particular system. Superfluous action sequences that are not required to manifest an error are filtered out, allowing a practitioner to focus on solving the fundamental issue.

11.2.3 *Scaling to Large Systems*

Hypothesis: *Estimation of Distribution Algorithms can scale to find faults in large complex systems.*

This sub-hypothesis is addressed by the empirical work in Chapter 10, and involved exploiting a feature unique to EDAs which enables scaling to large systems. The chapter outlined a number of scenarios in which model reuse, the reuse of strategies constructed by the algorithmic proposal of this thesis, can be exploited to scale to larger systems, as well as reduce computational effort in other software development activities. The empirical work in Chapter 10 showed how reusing strategies discovered when analysing smaller instances of a problem can be used to gain additional information about the fundamental nature of the problem in larger instances. This could aid in the debugging process, and provided a proof of concept that highlights the potential of model reuse.

The empirical work in Chapter 10 increases confidence in the validity of this sub-hypothesis. Using a feature unique to EDAs, the results show that EDAs can scale beyond that which has been demonstrated for other model checking techniques, heuristic or otherwise. The mechanism used to achieve this scaling has a number of potentially useful side effects which are outlined in Chapter 10.

11.2.4 *Over-arching hypothesis*

The overarching hypothesis addressed by this thesis is as follows:

Estimation of Distribution Algorithms are an effective mechanism for detecting and debugging concurrent faults.

The sub-hypotheses and the related empirical work summarised above increase confidence in the validity of this hypothesis. The work set out in this thesis increases the ability of a practitioner to detect the precise circumstances of a concurrent fault, and allows for a more effective debugging cycle due to the higher quality of solutions. The algorithmic proposal improves upon the state of the art in concurrent fault detection in comparable circumstances, and shows the potential to be useful in a range of practical scenarios. Additionally, this thesis has shown that features unique to EDAs, the building of strategies as well as solutions, can yield practical benefits in scalability as well as fault comprehension, as outlined in Chapter 10. I suspect that there is a wealth of opportunity with regards to this aspect in future research, and some of these potential avenues of research are outlined later in this chapter.

11.3 NOVEL CONTRIBUTIONS

As part of this thesis, I have made the following novel contributions to the state of the art:

- Proposed the first EDA-based algorithm for searching labelled transition systems, applying the technique to model checking problems.

  Previous work has focussed on approaches such as Genetic Algorithms and Ant Colony Optimisation, which have shown some weakness when compared with Random Search as demonstrated in Chapter 9. The proposal in this thesis has been shown to scale well to prominent benchmark problems in the model checking domain, without the need for special memory reduction measures. The algorithm is designed in such a way that it can search any labelled transition system, not just those generated by concurrent systems. There is the potential for the proposal to be applied to any problems that can be encoded as a "finding the goal-state in a labelled transition system".

- Shown that EDAs can not only find errors in concurrent systems, but optimise those solutions to achieve higher quality results.

  As well as detecting concurrent faults, the algorithmic proposal in this thesis has been shown to improve the quality of any errors detected. The empirical work in this thesis has shown that using a fitness function that favours higher quality solutions and by allowing the algorithm to continue execution after errors have been detected can yield higher quality solutions. The quality of a solution in the model checking domain is determined by the length of the path to a fault, the shorter the better. In practical concurrent fault finding scenarios, this can translate to a shorter debugging cycle by removing superfluous information from found faults, allowing a user to more quickly determine the cause of a problem.

- Outlined an information reuse strategy that can significantly reduce the effort required for gaining additional information, as well as improving the quality of solutions.

  As part of the published work in this thesis, several practical scenarios have been outlined in which huge effort savings can be obtained by reusing model information constructed and output during the course of the execution of the algorithm. The model constructed as part of the execution of the algorithm encodes a strategy for navigating a

transition system. This model can be output as part of the search process alongside any solutions found. The strategy can then be re-used in future executions of the algorithm, whereas solutions alone may be difficult to reuse. The scenarios in which this may be useful include regression testing, tabu search-like elimination of previously found errors, and searching for faults in problem families. The empirical work in this thesis examined the latter, showing potentially huge reductions in this scenario.

- Shown that Random Search is effective on a wide variety of problems.

  Evaluation of the performance of Random Search on the suite of benchmark problems available to the model checking community has been sparse until recently [Chicano et al., 2011]. Work undertaken as part of this thesis has highlighted the strength of Random Search in this problem domain, with Random Search competing well with all algorithms considered state-of-the-art. As part of this work, I have shown that Random Search can perform adequately on all but a few benchmark problems, suggesting that either more difficult problems should be sought by the community, or that Random Search may be good enough for many situations (assuming that the benchmark suite is representative of common scenarios). At the very least, the author recommends that Random Search should be used as a comparison algorithm when performing future work in this area.

11.4 Limitations of the research

Despite making inroads on a major obstacle to model checking large systems, the state space explosion problem, the approach investigated as part of this thesis is still bound by capabilities of the underlying model checker being used. All model checkers are subject to the limitations detailed in literature such as [Clarke et al., 2000] and [Baier and Katoen, 2008], and in some cases model checking tools are limited in the features of a language they can simulate. In these cases, systems are either impossible to check, or require simplifications to proceed. Unfortunately, this can affect the analysis of a wide variety of practical systems. For instance, checking a system that interacts with an agent over a network connection can lead to a huge number of additional states, and may require a simplifying “mock” agent to simulate. This mock implementation may miss key elements that may cause faults in the system under test, hindering the checking process. Due to the practical limitations of model checking, I cannot claim

that my algorithm can test an arbitrary concurrent program, in contrast to some static analysis techniques that can [Engler and Ashcraft, 2003].

As well as limitations in the types of system that the proposed algorithm can analyse, the types of faults that can be detected are also limited. Model checking tools typically have limitations on the types of properties that can be verified. For instance, whilst the Java PathFinder (JPF) model checker has a general deadlock finding capability, along with deadlock hinting heuristic measures, it does not provide a general "unwanted interference" property verifier with associated heuristics. In some cases, a manual definition of a concurrent fault is described and this is typically expressed using a language such as Linear Temporal Logic (LTL), which has limited support in practical model checking tools such as JPF. As a result of these limitations, the empirical work analysing Java software has focussed on finding deadlock only. For more complex properties, such as those specified using LTL, a more esoteric model checker SPIN must be used. The limitations on JPF (and other industrial language model checkers such as the C Bounded Model Checker (CMBC)) with respect to verifying more complex properties may be overcome in the future, and the approach outlined in this thesis could likely be applied to the revised toolsets.

Along with other work in the metaheuristic model checking domain, no effort has been made as part of this thesis to optimise fitness/objective functions when paired with various metaheuristic algorithms. All work to date in this domain has placed a objective value on a path through a system by summarising the objective values of the individual states that comprise that path. The function that achieves this summary is typically just a simple sum of individual values, or an appropriate averaging function. Whilst these summation mechanisms may prove effective on simple systems such as the Dining Philosophers problem, they may fall short on large practical systems. It is my suspicion that more complex path ranking mechanisms will be required in order for metaheuristic approaches to be effective when analysing more complex problems.

11.5 POTENTIAL ALGORITHM REFINEMENTS

In this section, I will provide a brief overview of potential refinements to the algorithm that could serve the approach well in the scenarios examined in this thesis, as well as others.

The EDA-based approach outlined in this thesis uses a simple mechanism to make the decision as to which action to choose next from a given state s when navigating a transition system. The choice is made using a particular context: Given the previous

n actions leading up to n, and a n-gram model of fit paths from a previous generation, which action should be executed next? The empirical work as part of this thesis has shown that this simple decision process can be effective on some systems, however improvement may be necessary to attack other systems.

Whilst any metaheuristic model checking technique would be augmented by any improvements to underlying tools such as JPF or SPIN, these kinds of improvements are outside the scope of this thesis. The algorithm is largely defined by how high ranking paths in a system under test are modelled, as this defines the learning process the mechanism used to generated further paths. Any changes to the modelling mechanism will require complimentary changes to the sampling mechanism. Outlined below are just a few ways in which the algorithm could be augmented.

11.5.1 *Improve the context in which a decision is made*

In addition to using the previous n actions and the n-gram model of fit paths, additional information could be used in choosing the next action to execute from a given state. As it stands, the proposed algorithm can use n-grams found in later parts of a path to inform decisions at earlier points during path generation. Whilst this may be beneficial in some circumstances (e.g. always go left in Dining Philosophers), in other situations it could potentially hinder the decision making process. For example, an action that always occurs at the beginning of every possible sequence, and could potentially occur later on, may have undue influence on the decision making process. In these cases, additional information can be used to further differentiate choices. Below are a number of ways the model could be augmented, along with a brief rationale.

- Differentiate by action depth

  The depth at which an action occurs in a sequence is defined by the action's position in that sequence. An action a is deeper than action b if a occurs later in a given sequence, with an action having a depth of zero if it occurs at the beginning of a sequence. Using the depth information of actions and n-gram could further enhance the decision making process, and tackle the issue highlighted above. For example, a simple augmentation may be to have different n-gram models for decisions at depth 0-20, 21-40 and so on, segmenting the model based on depth. This would limit the influence of n-grams found in deeper parts of paths on decisions made at shallower points, and vice-versa. Whether limiting the influence of n-grams in this way would

be beneficial to the search process is unclear and requires further investigation.

- Software Module or Class Name

  Using the current software module or class to inform the decision may be one way of aiding the choosing of actions. For instance, two classes in a Java program could contain the same action hoping to acquire a particular lock. The current n-gram modelling mechanism as described by this thesis will see both actions as the same. If class information is taken into account, the model sampling process could further differentiate between available actions. Constructing n-gram models specifically for each class, or pairs of interacting classes, may be one way of using this information. Implementation of this will be language specific, and could be automated in languages with obvious modularity (such as Java or C++ class information). Where obvious modularity is not available, modules in the description of the system could be described manually.

- Using domain specific knowledge

  During design and implementation of a system, particular parts of a system may be noted as "danger points" where concurrent faults are more likely to occur. It may be beneficial to tag these particular areas with annotations in the system description, which in turn is used by the EDA-based approach to aid in decision making. A simple form of this may be to tag particular actions with an "always choose" annotation, or conversely "never choose", with the EDA would taking appropriate action when these annotations are encountered. Other strategies are undoubtedly possible, and some augmentations may be problem specific.

11.5.2 *Augmenting the fitness function*

As mentioned above, the empirical work in this thesis uses a simple amalgamation of individual state fitnesses to produce a fitness for an overall path. Whilst this appears to be effective on the systems analysed as part of this thesis, improvements may be required for tackling more complex systems. When using a simple summation of individual state fitness values to obtain an overall path value, all states along the path potentially have the same influence as any other. It may be desirable in some cases to limit the influence of particular states, or accentuate the influence of others, whilst using the same function to amalgamate the individual state values. For instance, one could use domain specific knowledge to annotate certain classes or methods thought to be

suspicious. The states that touch any annotations could then be given extra weight, or any states that do not involve annotated code could be ignore completely during the amalgamation process. Additional weighting strategies could be derived from any of the scenarios outlined in Section 11.5.1. For example, extra weight could be given to states based on depth if an error is suspected to be at a particular depth.

11.6 potential avenues of future research

The algorithm detailed in this thesis searches over labelled transition systems for a goal state, using a heuristic to guide the search. The empirical work in this thesis has focussed on searching labelled transition systems that are generated by concurrent system specifications. However, I see no reason why this approach could not be applied to other problems that can be couched in this way. For instance, any problems that have previously been attacked by Ant Colony Optimisation may potentially be a target for the EDA, and the EDA may even compete when the problem evolves over time. Examples of problems successfully tackled by ACO include traffic routing and task scheduling.

The EDA may be useful in stress testing scenarios, such as testing a server component of a system. A stress test may involve making a high number of requests to a web server and the requests are one of a set number of types. The n-gram modelling mechanism can be used to generate such sequences, and generated sequences can be ranked with respect to some kind of desirable or undesirable behaviour (such as slow response times). The EDA in effect would be learning fit sequences of actions that cause the desired behaviours, and problematic sequences may be subtle in nature (much like finding faults in concurrent software).

11.7 summary

In this chapter, I have summarised the empirical work carried out as part of this thesis, reiterating how the evidence discussed in Chapters 8, 9 and 10 increases confidence in the validity of the hypotheses outlined in Chapter 2. I have highlighted the novel contributions of this thesis, and briefly discussed some limitations. I believe that the methods established in this thesis have the potential for further work, and I have briefly outlined some potential ways forward. It is a dream of mine to see techniques such as these churning away in the background of my IDE on my future 128 core system of the future, and I hope that the work produced as part of this helps toward that goal.

BIBLIOGRAPHY

N. Aan de Brugh, V. Nguyen, and T. Ruys. Moonwalker: Verification of .NET programs. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 170–173, 2009.

E. Alba and F. Chicano. Finding safety errors with ACO. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1066–1073. ACM Press New York, NY, USA, 2007.

E. Alba and F. Chicano. Searching for liveness property violations in concurrent systems with ACO. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1727–1734. ACM New York, NY, USA, 2008.

E. Alba and J.M. Troya. Genetic Algorithms for Protocol Validation. In *Parallel Problem Solving from Nature — PPSN IV*, volume 1141 of *Lecture Notes in Computer Science*, pages 870–879. Springer Berlin / Heidelberg, 1996.

E. Alba, F. Chicano, M. Ferreira, and J. Gomez-Pulido. Finding deadlocks in large concurrent java programs using genetic algorithms. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1735–1742. ACM New York, NY, USA, 2008.

P. Amey and B. Dobbing. High Integrity ravenscar. In *Reliable Software Technologies — Ada-Europe 2003*, volume 2655 of *Lecture Notes in Computer Science*, pages 637–637. Springer Berlin / Heidelberg, 2003.

C. Artho and A. Biere. Applying static analysis to large-scale, multi-threaded Javaprograms. In *Software Engineering Conference, 2001. Proceedings. 2001 Australian*, pages 68–75, 2001.

C. Artho and K. Havelund. Applying Jlint to Space Exploration Software. In *Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 61–75. Springer Berlin / Heidelberg, 2003.

C. Baier and J.P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

S. Baluja. Population-Based Incremental Learning: A Method for Integrating Genetic Search Based Function Optimization and Competitive Learning. Technical Report CMU-CS-94-163, Carnegie Mellon University Pittsburgh, PA, USA, 1994.

B. Beizer. *Software system testing and quality assurance*. Van Nostrand Reinhold Company, 1984.

B. Beizer. *Software testing techniques*. Van Nostrand Reinhold Co. New York, NY, USA, 1990.

Y. Ben-Asher, E. Farchi, and Y. Eytani. Heuristics for finding concurrent bugs. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*. IEEE Computer Society Washington, DC, USA, 2003.

A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur. Applications of synchronization coverage. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 206–212. ACM New York, NY, USA, 2005.

A. Burns and A.J. Wellings. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Addison Wesley, 2001.

J.H. Chen, D.E. Goldberg, S.Y. Ho, and K. Sastry. Fitness Inheritance In Multi-objective Optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference table of contents*, pages 319–326. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2002.

F. Chicano and E. Alba. Finding liveness errors with ACO. In *Proceedings of the World Conference on Computational Intelligence*, pages 3002–3009, 2008a.

Francisco Chicano and Enrique Alba. Ant colony optimization with partial order reduction for discovering safety property violations in concurrent models. *Inf. Process. Lett.*, 106(6):221–231, 2008b.

Francisco Chicano, Marco Ferreira, and Enrique Alba. Comparing metaheuristic algorithms for error detection in java programs. In Myra Cohen and Mel Ó Cinnéide, editors, *Search Based Software Engineering*, volume 6956 of *Lecture Notes in Computer Science*, pages 82–96. Springer Berlin / Heidelberg, 2011.

J. Clark, JJ Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, et al. Reformulating software engineering as a search problem. In *Software, IEE Proceedings-[see also Software Engineering, IEE Proceedings]*, volume 150, pages 161–175, 2003.

E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, 2004.

Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, January 2000.

E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logics of Programs*, pages 52–71. Springer, 1981.

E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K.L. McMillan, and L.A. Ness. Verification of the Futurebus+ cache coherence protocol. *Formal Methods in System Design*, 6(2):217–232, 1995.

EG Coffman and MJ Elphick. System Deadlocks. *Computing*, 3: 67–78, 1971.

P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM New York, NY, USA, 1977.

C. Darwin. *On the origin of species by means of natural selection, or the preservation of favoured races in the struggle for life.* New York, D. Appleton and company, 1860.

J.S. de Bonet, C.L. Isbell Jr, P. Viola, M.C. Mozer, M.I. Jordan, and T. Petsche. MIMIC: Finding Optima by Estimating Probability Densities. *Advances in Neural Information Processing Systems*, 9: 424, 1997.

D.L. Detlefs. An overview of the Extended Static Checking system. In *Proceedings of the First Workshop on Formal Methods in Software Practice*, pages 1–9, 1996.

Edsger W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

M. Dorigo and G. Di Caro. Ant colony optimization: a new metaheuristic. In *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, volume 2, 1999.

S. Edelkamp, A.L. Lafuente, and S. Leue. Directed explicit model checking with HSF-SPIN. In *Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 57–79. Springer-Verlag New York, Inc. New York, NY, USA, 2001a.

S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Protocol verification with heuristic search. In *AAAI-Spring Symposium on Model-based Validation Intelligence*, pages 75–83, 2001b.

S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology Transfer (STTT)*, 5(2):247–267, 2004.

O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded Java programs. *Concurrency and Computation: Practice and Experience*, 15, 2003.

A.E. Eiben and J.E. Smith. *Introduction to evolutionary computing*. Springer, 2003.

E.A. Emerson. *Branching time temporal logic and the design of correct concurrent programs*. Harvard University, 1981.

D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. *ACM SIGOPS Operating Systems Review*, 37(5):237–252, 2003.

Y. Eytani, K. Havelund, S.D. Stoller, and S. Ur. Towards a framework and a benchmark for testing tools for multi-threaded programs. *Concurrency and Computation: Practice and Experience*, 19(3), 2007.

Eitan Farchi, Yarden Nir, and Shmuel Ur. Concurrent bug patterns and how to test them. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 286.2, Washington, DC, USA, 2003. IEEE Computer Society.

M. Ferreira, F. Chicano, E. Alba, D.L. y Ciencias, and J.A. Gomez-Pulido. Detecting protocol errors using particle swarm optimization with java pathfinder. In *Proceedings of the High Performance Computing & Simulation Conference*, pages 319–325, 2008.

C. Flanagan and S.N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. *Science of Computer Programming*, 71(2):89–109, 2008.

Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. *SIGPLAN Not.*, 37(5):234–245, 2002.

F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers and Operations research*, 13(5): 533–549, 1986.

P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(2):117–127, 2004.

D.E. Goldberg. *Genetic algorithms in search, optimization and machine learning*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1989.

James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The 3rd Edition*. Addison-Wesley Professional, 2005.

GR Harik, FG Lobo, DE Goldberg, S.G.C. Syst, and M. View. The compact genetic algorithm. In *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on*, pages 523–528, 1998.

K. Havelund. Java PathFinder, a translator from Java to Promela. In *Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *Lecture Notes in Computer Science*, pages 152–152. Springer Berlin / Heidelberg, 1999.

CAR Hoare and CAR Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21:666–677, 1985.

J.H. Holland. Adaptation in natural and artificial system. *Ann Arbor, MI: University of Michigan Press*, 1975.

G.J. Holzmann. The SPIN model-checker. *Proceeding FORTE 1999*, 28:481–497, 1997.

G.J. Holzmann. *The SPIN model checker: Primer and reference manual*. Addison-Wesley Professional, 2004.

Intel Corp. Intel Research Advances 'Era Of Tera', February 2007. URL http://www.intel.com/pressroom/archive/releases/20070204comp.htm.

G. Jones and M.H. Goldsmith. *Programming in OCCAM2*. Prentice Hall, 1988.

J. Kennedy and R. Eberhart. Particle Swarm Optimization. In *Neural Networks, 1995. Proceedings., IEEE International Conference on*, volume 4, pages 1942–1948. IEEE, 1995.

S. Kirkpatric, CD Gelatt, and MP Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

G. Koch. Discovering multi-core: extending the benefits of moore's law. *Technology*, page 1, 2005.

J.R. Koza. *Genetic programming: on the programming of computers by means of natural selection*. MIT press, 1992.

W.B. Langdon and R. Poli. *Foundations of genetic programming*. Springer, 2002.

K.R.M. Leino and G. Nelson. An extended static checker for Modula-3. In *Proceedings of the 7th International Conference on Compiler Construction*, pages 302–305. Springer-Verlag London, UK, 1998.

S. Lu, W. Jiang, and Y. Zhou. A study of interleaving coverage criteria. In *Foundations of Software Engineering*, pages 533–536. ACM New York, NY, USA, 2007.

J. Magee and J. Kramer. *Concurrency: State Models And Java Programs*. Wiley New York, 2006.

P. McMinn. Search-based software test data generation: a survey. *Software Testing, Verification & Reliability*, 14(2):105–156, 2004.

D. Merkle, M. Middendorf, and H. Schmeck. Ant colony optimization for resource-constrained project scheduling. *Evolutionary Computation, IEEE Transactions on*, 6(4):333–346, 2002.

S. Merz. Model checking: A tutorial overview. In *Modeling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes In Computer Science*, pages 3–38. Springer Berlin / Heidelberg, 2001.

R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

H. Mühlenbein and G. Paaß. From recombination of genes to the estimation of distributions I. Binary parameters. *Parallel Problem Solving from Nature-PPSN IV*, pages 178–187, 1996.

J. Očenášek. Parallel estimation of distribution algorithms. *Doctroral dessertation, Faculty of Information Technology, Brno University of Technology*, 2002.

R.S. Parpinelli, H.S. Lopes, and A.A. Freitas. Data mining with an ant colony optimization algorithm. *Evolutionary Computation, IEEE Transactions on*, 6(4):321–332, 2002.

M. Pelikan, D.E. Goldberg, and E. Cantu-Paz. Linkage problem, distribution estimation, and Bayesian networks. *Evolutionary Computation*, 8(3):311–340, 2000.

M. Pelikan, D.E. Goldberg, and F.G. Lobo. A survey of optimization by building and using probabilistic models. *Computational optimization and applications*, 21(1):5–20, 2002.

R. Poli and N.F. McPhee. A Linear Estimation-of-Distribution GP System. In *Genetic Programming*, volume 4971 of *Lecture Notes in Computer Science*, pages 206–217. Springer Berlin / Heidelberg, 2008.

W. Pugh and N. Ayewah. Unit testing concurrent software. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 513–516. ACM New York, NY, USA, 2007.

J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351. Springer, 1982.

C.R. Reeves. *Modern heuristic techniques for combinatorial problems*. John Wiley & Sons, Inc. New York, NY, USA, 1993.

A.W. Roscoe, C.A.R. Hoare, and R. Bird. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.

S.J. Russell, P. Norvig, J.F. Canny, J. Malik, and D.D. Edwards. *Artificial Intelligence: A Modern Approach*. Prentice Hall Englewood Cliffs, NJ, 1995.

C. Ryan, JJ Collins, and M.O. Neill. Grammatical evolution: Evolving programs for an arbitrary language. In *Genetic Programming*, volume 1391 of *Lecture Notes in Computer Science*, pages 83–96. Springer Berlin / Heidelberg, 1998.

R. Salustowicz and J. Schmidhuber. Probabilistic incremental program evolution. *Evolutionary Computation*, 5(2):123–141, 1997.

S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15 (4):391–411, 1997.

Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley, December 2004.

K.M. Sim and W.H. Sun. Ant colony optimization for routing and load-balancing: survey and new directions. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 33(5):560–572, 2003.

Jan Staunton and John Clark. Applications of model reuse when using estimation of distribution algorithms to test concurrent software. In Myra Cohen and Mel Ó Cinnéide, editors, *Search Based Software Engineering*, volume 6956 of *Lecture Notes in Computer Science*, pages 97–111. Springer Berlin / Heidelberg, 2011a.

Jan Staunton and John A. Clark. Searching for Safety Violations Using Estimation of Distribution Algorithms. *Software Testing Verification and Validation Workshop, IEEE International Conference*

on Software Testing, Verification, and Validation, pages 212–221, 2010.

Jan Staunton and John A. Clark. Finding short counterexamples in promela models using estimation of distribution algorithms. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, GECCO '11, pages 1923–1930, New York, NY, USA, 2011b. ACM.

S.D. Stoller. Testing concurrent Java programs using randomized scheduling. *Electronic Notes in Theoretical Computer Science*, 70 (4):142–157, 2002.

F. Tip. *A Survey of Program Slicing Techniques*. Centrum voor Wiskunde en Informatica, 1994.

M.Y. Vardf and P. Wolper. An Automata—Theoretic Approach to Automatic Program Verification (Preliminary Report). In *Symposium on Logic in Computer Science: Proceedings: Cambridge, Massachusetts, June 16-18, 1986*, page 332. IEEE Computer Society Press, 1986.

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

M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press Piscataway, NJ, USA, 1981.

L.D. Whitley. Fundamental principles of deception in genetic search. *Foundations of genetic algorithms*, 1(3):221–241, 1991.

S. Wright. The roles of mutation, inbreeding, crossbreeding, and selection in evolution. In *Proc of the 6th International Congress of Genetics*, volume 1, pages 356–366, 1932.

C.H. Yang and D.L. Dill. Validation with guided search of the state space. In *Proceedings of the 35th annual conference on Design automation*, pages 599–604. ACM New York, NY, USA, 1998.

J. Zhao. Slicing Concurrent Java Programs. In *Proceedings of the 7th IEEE International Workshop on Program Comprehension*, pages 126–133, 1999.

COLOPHON

This thesis was typeset with $\LaTeX 2_\varepsilon$ using Hermann Zapf's *Palatino* and *Euler* type faces (Type 1 PostScript fonts *URW Palladio L* and *FPL* were used). The listings are typeset in *Bera Mono*, originally developed by Bitstream, Inc. as "Bitstream Vera". (Type 1 PostScript fonts were made available by Malte Rosenau and Ulrich Dirr.)

NOTE: The custom size of the textblock was calculated using the directions given by Mr. Bringhurst (pages 26–29 and 175/176). 10 pt Palatino needs 133.21 pt for the string "abcdefghijklmnopqrstuvwxyz". This yields a good line length between 24–26 pc (288–312 pt). Using a "*double square textblock*" with a 1:2 ratio this results in a textblock of 312:624 pt (which includes the headline in this design). A good alternative would be the "*golden section textblock*" with a ratio of 1:1.62, here 312:505.44 pt. For comparison, `DIV9` of the `typearea` package results in a line length of 389 pt (32.4 pc), which is by far too long. However, this information will only be of interest for hardcore pseudo-typographers like me.

To make your own calculations, use the following commands and look up the corresponding lengths in the book:

```
\settowidth{\abcd}{abcdefghijklmnopqrstuvwxyz}
\the\abcd\ % prints the value of the length
```

Please see the file `classicthesis.sty` for some precalculated values for Palatino and Minion.

145.86469pt

Final Version as of 19th October 2012 at 10:14.