

Verifying concurrent software using metaheuristic  
search techniques

Qualifying Dissertation

Jan Staunton

Department of Computer Science  
University of York

July 2, 2009

## Abstract

Processors with multiple execution cores, dubbed multi-core processors, are becoming increasingly prominent. The trend of fabricating an increasing number of execution cores on a single chip is not likely to end soon with Intel predicting that an 80 core CPU will be commercially available within the next decade. The trend can be seen in many embedded systems, where multiple processors may be required in order to provide a responsive user experience whilst meeting energy constraints.

Developing software to exploit multiple processors is a challenging task, and testing that software even more so. Testing of concurrent software can be a daunting task, as behaviour can differ between subsequent executions. Errors may manifest in some executions and not others. A promising approach to this problem is to search the state space of the concurrent program. However, this approach exhibits scalability issues as state spaces can be large. Metaheuristic search techniques may be able to avoid this problem, allowing efficient testing of concurrent software.

This report outlines the problem area and some previously reported solutions. Described in this report are verification mechanisms that explore the state space of a concurrent program, and an outline of metaheuristic search techniques. This report proposes the use of a previously untested metaheuristic search technique, in combination with techniques found in previously reported literature.

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Multi-processor Systems . . . . .	5
1.2	Utilising multiple cores . . . . .	5
1.3	Concurrent software verification . . . . .	6
1.4	Metaheuristic search techniques . . . . .	6
1.5	Motivation for research . . . . .	7
1.6	Research goals . . . . .	7
1.7	Structure of this report . . . . .	7
<b>2</b>	<b>Concurrent Software Verification</b>	<b>8</b>
2.1	Introduction to concurrent software . . . . .	8
2.1.1	Scheduling . . . . .	9
2.1.2	Competition for resources . . . . .	10
2.1.3	Communication . . . . .	11
2.2	Prominent faults in concurrent software . . . . .	12
2.2.1	Deadlock . . . . .	13
2.2.2	Starvation . . . . .	17
2.2.3	Livelock . . . . .	17
2.2.4	Data races . . . . .	18
2.3	Concurrent software verification . . . . .	19
2.3.1	Coverage metrics . . . . .	19
2.4	Static verification techniques . . . . .	20
2.4.1	Lockset analysis . . . . .	20
2.4.2	Data-flow analysis . . . . .	21
2.4.3	Annotation-based theorem proving . . . . .	21
2.4.4	Summary of static verification techniques . . . . .	21
2.5	Dynamic verification techniques . . . . .	22
2.5.1	Schedule altering techniques . . . . .	22
2.5.2	Runtime monitoring . . . . .	23
2.5.3	Summary of dynamic verification techniques . . . . .	23
2.6	Formal methods . . . . .	23
2.6.1	Specification . . . . .	23
2.7	Summary . . . . .	24
<b>3</b>	<b>Model Checking</b>	<b>26</b>
3.1	Introduction . . . . .	26
3.2	Building the model . . . . .	27
3.2.1	Transition systems . . . . .	27

3.2.2	Paths . . . . .	29
3.2.3	Reachable states . . . . .	29
3.2.4	Modelling a concurrent system . . . . .	29
3.2.5	Reactive systems . . . . .	31
3.2.6	Modelling software . . . . .	31
3.2.7	Modelling concurrent software . . . . .	36
3.2.8	Expressing models . . . . .	36
3.3	Specification of properties . . . . .	37
3.3.1	Temporal logics . . . . .	37
3.3.2	Linear temporal logic . . . . .	38
3.3.3	Concurrency and fairness . . . . .	39
3.3.4	Branching temporal logic . . . . .	41
3.3.5	Computation tree logic . . . . .	41
3.3.6	Extended computation tree logic . . . . .	42
3.3.7	Other path description languages . . . . .	42
3.4	Verification . . . . .	43
3.4.1	Complete model checking mechanisms . . . . .	44
3.4.2	On-the-fly model checking . . . . .	45
3.4.3	Guided model checking . . . . .	45
3.5	The strengths of model checking . . . . .	46
3.5.1	General technique . . . . .	46
3.5.2	Partial verification . . . . .	46
3.5.3	Useful in debugging . . . . .	46
3.5.4	Potential for automation . . . . .	46
3.6	Issues with model checking . . . . .	46
3.6.1	Model inaccuracy . . . . .	46
3.6.2	Somewhat limited to control oriented verification . . . . .	47
3.6.3	State-space explosion problem . . . . .	47
3.6.4	Other issues . . . . .	47
3.7	Tackling the issues . . . . .	48
3.7.1	Reducing the likelihood of model inaccuracy . . . . .	48
3.7.2	Reducing the state-space . . . . .	48
3.8	Summary . . . . .	49
<b>4</b>	<b>Metaheuristic search</b>	<b>51</b>
4.1	Introduction . . . . .	51
4.1.1	Metaheuristic search . . . . .	52
4.2	Local search . . . . .	52
4.2.1	Convergence and optima . . . . .	53
4.2.2	Landscapes . . . . .	53
4.2.3	Fitness function . . . . .	55
4.2.4	Simulated annealing . . . . .	55
4.3	Other local search mechanisms . . . . .	55
4.4	Population-based search . . . . .	56
4.4.1	Evolutionary algorithms . . . . .	56
4.4.2	Genetic algorithms . . . . .	57
4.5	Estimation of distribution algorithms . . . . .	61
4.5.1	Example EDA . . . . .	61
4.5.2	Types of EDA . . . . .	62
4.5.3	Strengths . . . . .	63

4.5.4	Weaknesses . . . . .	63
4.6	Summary . . . . .	63
<b>5</b>	<b>Previous work applying search to find concurrency bugs</b>	<b>64</b>
5.1	Searching for deadlock . . . . .	64
5.1.1	Genetic algorithms . . . . .	65
5.1.2	Ant colony optimisation . . . . .	66
5.2	Searching for liveness errors . . . . .	67
5.2.1	Ant colony optimisation . . . . .	67
5.3	Heuristics . . . . .	68
5.3.1	Using static analysis . . . . .	68
5.3.2	User annotations . . . . .	68
5.3.3	Learning from mistakes . . . . .	68
5.4	Reducing the state space . . . . .	69
5.4.1	Reducing OBDD size . . . . .	69
5.5	Summary and conclusion . . . . .	69
<b>6</b>	<b>Proposal</b>	<b>71</b>
6.1	Research strands . . . . .	71
6.1.1	EDA-based model checking . . . . .	71
6.1.2	Modelling the execution space . . . . .	72
6.1.3	Scalability . . . . .	72
6.1.4	Exploiting traditional techniques . . . . .	73
6.1.5	Thesis write-up . . . . .	73
6.2	Tools and support . . . . .	73
6.2.1	Model checking frameworks . . . . .	73
6.2.2	Metaheuristic search frameworks . . . . .	74
6.2.3	Summary . . . . .	75
6.3	Benchmark problems . . . . .	75
6.4	Computational requirements . . . . .	75
6.4.1	CPU requirements . . . . .	76
6.4.2	Memory requirements . . . . .	76
6.4.3	Distributed computation . . . . .	76
6.4.4	Available hardware . . . . .	77
6.4.5	Project management . . . . .	78
6.5	Summary . . . . .	78

# List of Figures

2.1	Round robin schedule for processes A, B and C . . . . .	9
2.2	wait and signal implementation pseudocode . . . . .	11
2.3	Mutual exclusion implementation using semaphores . . . . .	12
2.4	Resource allocation graph for resources 1 through 5 and processes 1 through 5, loosely based on [Silberschatz et al., 2004] . . . . .	13
2.5	Wait-for graph for processes 1 through 5, loosely based on [Sil- berschatz et al., 2004] . . . . .	14
2.6	Wait-for cycle depiction . . . . .	15
2.7	Dining Philosophers problem . . . . .	16
2.8	Naive dining philosopher behaviour . . . . .	17
2.9	Data race example code . . . . .	18
2.10	Nasty interleaving . . . . .	18
3.1	Traffic light transition system, based on the traffic light example found in [Baier et al., 2008] . . . . .	28
3.2	Asynchronous Concurrent traffic light transition system . . . . .	30
3.3	Synchronous Concurrent traffic light transition system . . . . .	32
3.4	Software system in C code . . . . .	34
3.5	Program graph for C fragment . . . . .	35
3.6	Expanded program graph for C fragment . . . . .	35
3.7	Semantics of the $\models$ relation, from [Baier et al., 2008] . . . . .	38
4.1	Example landscapes . . . . .	54
4.2	Algorithm of a vanilla EA . . . . .	57
4.3	Algorithm of a vanilla EA . . . . .	59
4.4	Genetic operators . . . . .	60
4.5	Algorithm of a vanilla EDA . . . . .	62
5.1	Fitness function for deadlock from [Alba et al., 2008] . . . . .	66

# Chapter 1

## Introduction

### 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 not showing any sign of slowing down. At the time of writing, it is common to find two-core (dual-core) processors in most modern machines, and some instances four-core (quad-core) processors. 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 during general interactive tasks, and this seems to implicitly scale with additional cores. However, for application developers to see the benefit of multi-core technology, explicit division of computation over multiple processors is required.

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 mechanisms within a language to co-ordinate access to shared resources.

### 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 the potential for *deadlock* and *race conditions*. Automatically verifying that a concurrent software program is free from such errors is a difficult problem. 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, making dynamic testing difficult. Race conditions are particularly notorious, as execution can continue in the presence of race conditions with inaccurate results. Static analysis techniques can highlight the potential for concurrent faults through examination of the source code, but suffer from produce spurious errors to the point that trust is lost in such techniques.

Another approach to concurrent software verification is to examine all possible executions of a concurrent program in order to discover errors. This requires the building of a state space represented as an automaton. Each state in the automaton consists of the composition of the states of each process within the program. Edges between each state represent the progression of a single process within 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, large enough to exhaust the resources available to a developer.

### 1.4 Metaheuristic search techniques

It has been shown that software engineering tasks, particularly software verification, can be couched as an optimisation problem [Clarke et al., 2003, McMinn, 2004]. Using this formulation of the problem, metaheuristic search techniques can be applied. Metaheuristic search techniques have been used to solve hard optimisation problems. Some example techniques include local search, 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 5 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. Little is known as to what constitutes a difficult concurrent bug for a particular search technique to detect. No investigation can be carried out as to how these techniques can reveal insights into the behaviour of concurrent programs.

## 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 method. 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 and race conditions. These can include finding violations of liveness properties (discussed thoroughly in chapter 3).

## 1.7 Structure of this report

The structure of this document is as follows. Chapter two introduces concurrent software and various methods of verifying concurrent software. Chapter 3 outlines model checking, an effective mechanism for verifying properties of concurrent systems. Chapter 4 describes some metaheuristic search mechanisms, and chapter 5 describes some previous work applying these mechanisms to the problem of finding errors in concurrent systems. The final chapter details a proposal for research.

## Chapter 2

# Concurrent Software Verification

### 2.1 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.

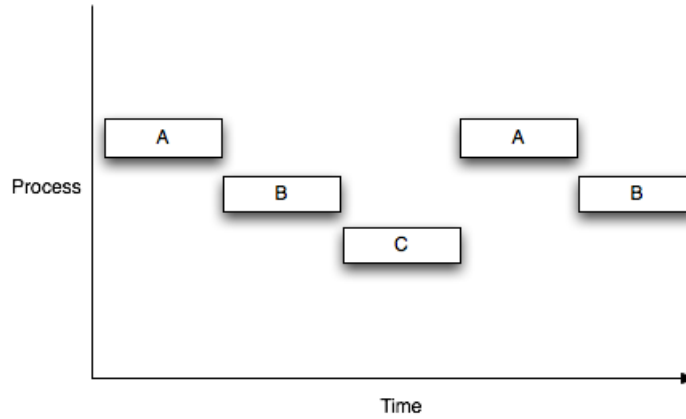
A sequential software program consists of a single process. Repeated executions of a sequential program with a particular input will yield the same behaviour each time, unless there is some stochastic element to the program<sup>1</sup>. In mainstream programming languages such as C and Java a process 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 conjunction 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 that event is a popular method of implementation. Using this model, the process can respond to the event independently from the other processes in the software, yielding a quick reaction to the event. 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

---

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

Figure 2.1: Round robin schedule for processes A, B and C



shall be used interchangeably throughout this report.

### 2.1.1 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 interleaved on a single processor according to some scheduling policy, the simplest schedule being a round robin schedule (Figure 2.1). 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 exist, 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 previously determined next turn of the process. When executing multiple processes on a single processor, pre-emption allows for fast reaction times in reactive software systems allowing for quick responses to events in the environment.

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 ( $c$ ) processors executing a concurrent software program, the processes that make up said concurrent software program can be divided between the  $c$  processors. In this environment, up to  $c$  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 execu-

tion 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 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].

### 2.1.2 Competition for resources

The scheduling phenomena described above highlights how processes can compete for a resource, namely a fixed number of processors on which to be executed. The scheduler divides the resource according to some policy amongst the individual processes. The processor is a special type of resource that is implicitly shared amongst each of the individual processes. Some problems, however, may require explicit sharing of resources. An example situation would be access to a particular output device or file, where only one process may access the resource at any time. In this situation, processes are said to be accessing the resource in *mutual exclusion*.

When a process is accessing the resource in mutual exclusion, the resource is *locked* to all other processes, and said process unlocks the resource once it has finished using it. A process can only lock an unlocked resource. A process typically waits for a resource to become unlocked if it tries to acquire a resource that is locked. A great many bugs in concurrent software programs arise from the locking and unlocking of resources. For instance, failing to obtain a lock on a resource before using it can lead to multiple processes accessing the

Figure 2.2: wait and signal implementation pseudocode

```
1
2 wait(S) {
3     while S <= 0 {
4         ; //noop
5     }
6     S--;
7 }
8
9 signal(S) {
10    S++;
11 }
```

same resource simultaneously, leading to *interference* (discussed later in data races).

### 2.1.3 Communication

The simplest concurrent software program consists of  $n$  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. In general, a concurrent software program is more complex if any of the processes within the software program are dependent on progress of each other. Dependencies between and process  $A$  and a process  $B$  are caused by *communication* between  $A$  and  $B$ . In order for process  $A$  and  $B$  to communicate, the two processes must *synchronise* their actions in order to share information. Various synchronisation mechanisms exist, and they are outlined below.

#### Semaphores

One of the most basic synchronisation structures is the *semaphore* [Dijkstra, 1968]. Semaphores can be used to mediate access a finite number  $n$  of a resource. A semaphore  $s$  representing the resource is shared between a number of processes, and is initialised with  $n$ . A process can access  $s$  via two mechanisms, `wait` and `signal`. The definitions of these two procedures are described in figure 2.2. Both `wait` and `signal` are implemented atomically.

For a process  $P$  to access a resource represented by semaphore  $s$ ,  $P$  calls `wait(s)`. If one of the resource is free, then  $P$  decrements  $s$  and uses that resource.  $P$  is said to have acquired a *lock* on the resource. Otherwise,  $P$  waits until one of the resource is free. When  $P$  is done with the resource,  $P$  calls `signal(s)`, signalling that a resource is free for another process to use.  $P$  is said to have *unlocked* the resource. The simplest form of a semaphore is the binary semaphore, which can be used to mediate critical section access. A binary semaphore  $m$  is initialised with the value 1, and is shared amongst a number of processes.  $m$  is a *mutex* in this case, and can then be used by processes to guarantee mutually exclusive access to critical sections. A critical section is the block of code that utilises the shared resource. Figure 2.3 demonstrates this.

Figure 2.3: Mutual exclusion implementation using semaphores

```
1
2 //Do some other work
3
4 wait(m);
5
6 //Do critical section work exclusively
7
8 signal(m);
9
10 //Do some other work...
```

### Monitors

The use of a mutex to implement mutual exclusion is an effective mechanism. However, incorrect usage of a mutex (omitting a signal(S) for instance) can lead to subtle errors. In order to aid in the implementation of mutual exclusion, *monitors* can be used. A monitor is an object that can be safely used by multiple processes. A monitor object  $m$  consists of variables that make up the state of the object, plus procedures that can manipulate the state. The guarantee is that execution of the procedures of  $m$  are done so in mutual exclusion. Rather than having to explicitly synchronise using a mutex, the monitor object handles this implicitly, reducing the potential for synchronisation errors. Monitors can be extended to implement other synchronisation schemes, and good treatment of monitors can be found in [Burns and Wellings, 2001, Silberschatz et al., 2004]. The Java programming language implements a form of monitors via the `synchronised` keyword.

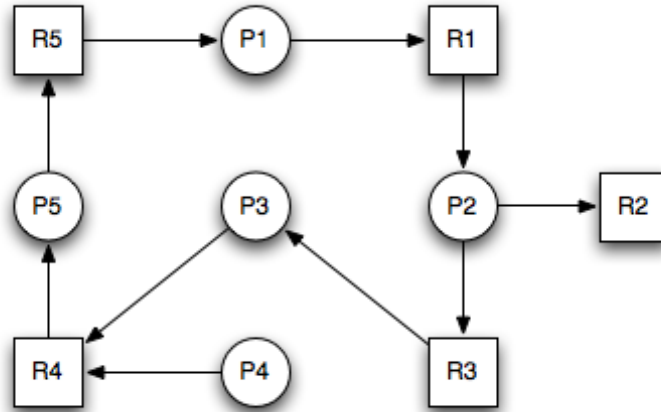
### Message passing

Message-based communication is another form of communication that can be used for process communication. Message-passing 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]. [Burns and Wellings, 2001] gives a good overview of the various message passing schemes, and the languages that implement them.

## 2.2 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 pro-

Figure 2.4: Resource allocation graph for resources 1 through 5 and processes 1 through 5, loosely based on [Silberschatz et al., 2004]



cesses. 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, and I prefer this view of looking at concurrent bugs in software programs.

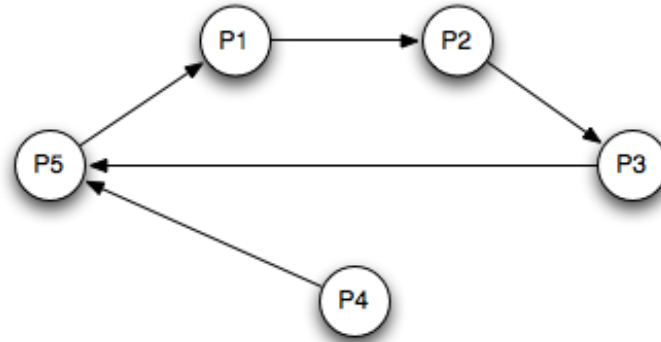
In this section, I shall outline some of the prominent concurrent faults that occur in real concurrent programs along with examples.

### 2.2.1 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  $R_i$ . 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 2.4.

This graph can be 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 waiting for a resource that  $P_j$  currently holds. A wait-for graph that is equivalent to the resource allocation graph in figure 2.4 is depicted in figure 2.5.

Figure 2.5: Wait-for graph for processes 1 through 5, loosely based on [Silberschatz et al., 2004]



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  $i^{\text{th}}$  ( $0 \leq i \leq n$ ) process is waiting on a resource held by process  $i + 1$ , modulo the number of processes  $n$ . Figure 2.6 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 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.

Figure 2.6: Wait-for cycle depiction

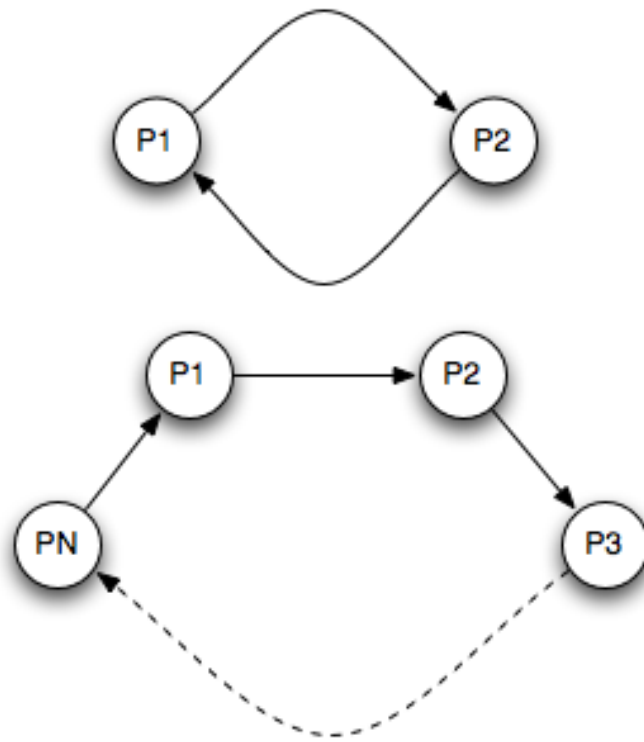
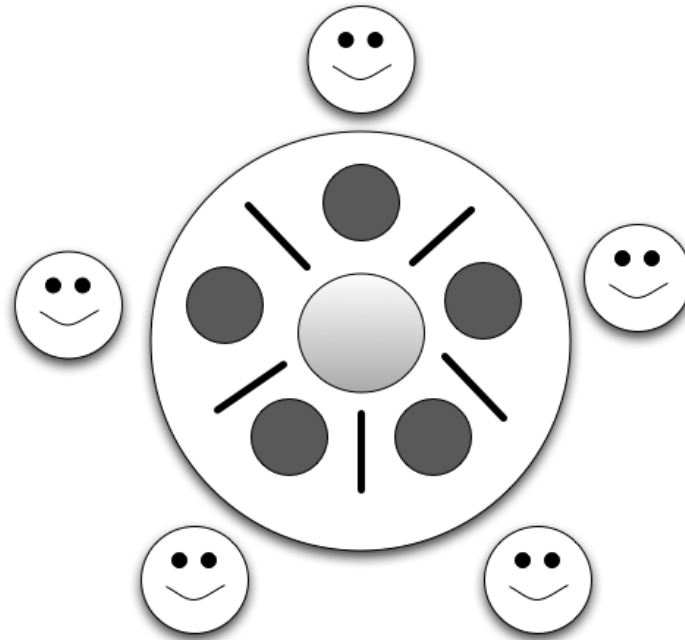


Figure 2.7: Dining Philosophers problem



### Dining philosophers problem

The Dining philosophers problem [Dijkstra, 1968] is by far the most famous example of a potential deadlock situation. The situation is a set of philosophers share a circular dining table (Figure 2.7), 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 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 resources 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 2.8.

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

Figure 2.8: Naive dining philosopher behaviour

```
1
2 while (hungry) {
3     acquire_left_fork ();
4     acquire_right_fork ();
5     eat ();
6     release_left_fork ();
7     release_right_fork ();
8 }
```

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). 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 condition for deadlock.

### 2.2.2 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.

### 2.2.3 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 performing any useful processing. Livelock situations have a much more loosely defined structure, typically dependant on the structure of the system in question. This makes testing for livelock a more difficult challenge, as it is difficult to define in the general case.

Figure 2.9: Data race example code

```
1
2 int tmp_var = x;
3 tmp_var = tmp_var + 50;
4 x = tmp_var;
```

Figure 2.10: Nasty interleaving

```
1
2 Thread one: int tmp_var = x;
3 Thread two: int tmp_var = x;
4 Thread two: tmp_var = tmp_var + 50;
5 Thread one: tmp_var = tmp_var + 50;
6 Thread one: x = tmp_var;
7 Thread two: x = tmp_var;
```

## 2.2.4 Data races

Race conditions occur when two more more processes access a shared memory object without proper co-ordination, usually in the form of synchronisation. The overriding symptom of a race condition is that the manipulation of a shared memory location is dependent on the interleaving of actions that manipulate it. Race conditions are particularly notorious as a concurrent software program can potentially continue processing in the face of race conditions, whereas deadlock has an overriding and obvious show stopping behaviour.

The most simple example of a data race is as follows. A concurrent Java program is made up of two threads/processes that access the integer variable  $x$ , initially set to 0. Both threads apply the operations outlined in figure 2.9, each with a unique local  $tmp\_var$ .

The intuitive view of this operation is that the first thread will make a temporary copy of  $x$ , increment the temporary copy, and write the new value back to  $x$ , followed by second thread doing the same, yielding  $x == 100$ . However, it is possible for the interleaving outlined in figure 2.10 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 2.9 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 on some of the synchronisation mechanisms highlighted on page 11.

Data race errors can be more subtle than the exceptionally verbose 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 4 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 = x+1`. 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).

## 2.3 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 rigorous 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 increases the difficulty of testing concurrent software over testing sequential software. For instance, [Beizer, 1990] describe a great number of methods that rely on the deterministic nature of input/output relationships defined by a program. This 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 eating.

### 2.3.1 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. To satisfy developers that a program has been exercised fully, 100% coverage of branches or statements may suffice.

100% path coverage of a multi-threaded program requires that all possible interleavings over all possible inputs are exercised. The set of all possible execu-

tions of a concurrent program can be extremely large, exponentially increasing 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 interleaving coverage criteria, 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.

The following sections are dedicated to techniques that attempt to tackle the problem of verifying or testing concurrent software programs.

## 2.4 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. Static analysis is a fundamental tool in compilers, which use static analysis techniques for type checking and dead code detection. 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 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.

### 2.4.1 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 target 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.

### 2.4.2 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.

### 2.4.3 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 that clarify potential ambiguities. 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 provided are to aid the programmer in further specifying the design, and the tools automatically check if the current implementation can invalidate that design.

### 2.4.4 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 [Artho and Biere, 2001, Artho and Havelund, 2003, Engler and Ashcraft, 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, having to sift

through a large amount of potentially false positives. A concrete interleaving of processes within a piece of software that can lead to a concurrent bug is also absent from error reports, an invaluable piece of information when debugging the error. 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.

## 2.5 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.

### 2.5.1 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 the 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 be generated automatically and inexpensively. Additionally, the unit tests generated can be used as part of a regression suite [Beizer, 1990].

## 2.5.2 Runtime monitoring

Runtime monitoring techniques perform instrumentation on a particular concurrent program 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 use 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.

## 2.5.3 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 suffer from a few drawbacks. The major one is that execution of the program can be very expensive. Due to the non-determinism 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.

## 2.6 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 high level specifications of systems can be examined in order to verify particular properties. The specification of a system can be thought of as a model of the system. Various methods of specification/modelling and analysis exist, and some are described briefly below.

### 2.6.1 Specification

The most prominent method to express the behaviour of a concurrent 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. Examples of process algebras are CSP and CCS [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]. A standard programming language such as Java can be considered a modelling language, but with less clear manipulation operators when compared with a process algebra.

## 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, which can be represented as a transition system (discussed in chapter 3), is extracted from the model and exhaustively checked for conformance with a specification. This process is known as *model checking* Baier et al. [2008], Clarke et al. [2000]. If a violation is found, a concrete trace of events leading up the error can be reported.

Transition systems can be extracted from program source code and checked in a similar fashion. 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 a program. This eliminates some of the downfalls of traditional static and dynamic analysis. 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, has explicit control of the scheduling of the processes of a system.

The major problem with exhaustive searching of a state space is that the state space of concurrent systems can be large. It is typical for the state space of practical systems to be too large to fit into memory. Some techniques have been devised to overcome this problem. A detailed overview of model checking is given in chapter 3.

## 2.7 Summary

This chapter has outlined the basics of concurrent programming, and the issues of concurrent software verification. The problem of finding concurrent bugs is 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.

A lot of the informal verifications techniques discussed are specific to a particular type of concurrent fault. `raceFinder`, for instance, is a tool dedicated to checking for potential race conditions and nothing else. This means that a number of techniques must be used in order to verify various properties, possibly requiring integration of a number of different frameworks causing high cost during the verification process.

One technique stands out from the crowd in this respect and that is model checking. Model checking allows for the automatic verification of a wide variety of properties all under a single framework. Model checking can be applied to software source code, eliminating the disconnect from the artefact that some formal methods can introduce. Model checking can produce a concrete interleaving of process actions that lead to a particular fault, an invaluable facility in the debugging process. Model checking, in the authors opinion, represents one of the most promising mechanisms for efficiently detecting concurrent faults in software programs. In the next chapter, model checking is described in detail.

## Chapter 3

# Model Checking

### 3.1 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 et al., 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 automata 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 et al., 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 et al., 2008], a modern reference with specific coverage of software model check-

ing. 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 reference for the major concepts of model checking, with [Clarke et al., 2000] cited in all related works.

## 3.2 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 et al., 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 information encoding information about those states. [Baier et al., 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 be derived. The language used is dependant on which model checking tool one is using. The SPIN model checker [Baier et al., 2008, Holzmann, 1997, 2004] 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.

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.

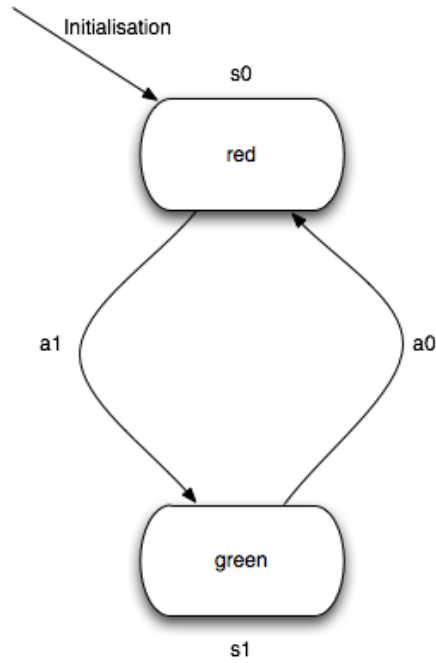
### 3.2.1 Transition systems

To model the execution of a system, a transition system [Baier et al., 2008] can be constructed. A transition system is made up of states and transitions between those states. A state encodes some information about the system [Baier et al., 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* [Baier et al., 2008, Clarke et al., 2000].

**Definition 1. Transition system**, definition from [Baier et al., 2008].

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

Figure 3.1: Traffic light transition system, based on the traffic light example found in [Baier et al., 2008]



- $S$  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  $s \xrightarrow{a} s'$ , where  $a \in Act$ . A state  $s$  can be a successor state of itself.

For example, figure 3.1 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 3.1 is deterministic.

$AP$  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  $s_1 \rightarrow green \in L$ .

### 3.2.2 Paths

A *path fragment*  $\rho$  through a transition system from states  $s_0$  to  $s_n$  is a 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.

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

If a path fragment  $\rho$  contains an initial state, then  $\rho$  is said to be *initial* [Baier et al., 2008]. If  $\rho$  ends with a terminal state or is an infinite path, then  $\rho$  is said to be *maximal* [Baier et al., 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 et al., 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 can be one 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  $\rho^\omega$  denotes a infinite path that consists of  $\rho$  repeated infinitely.

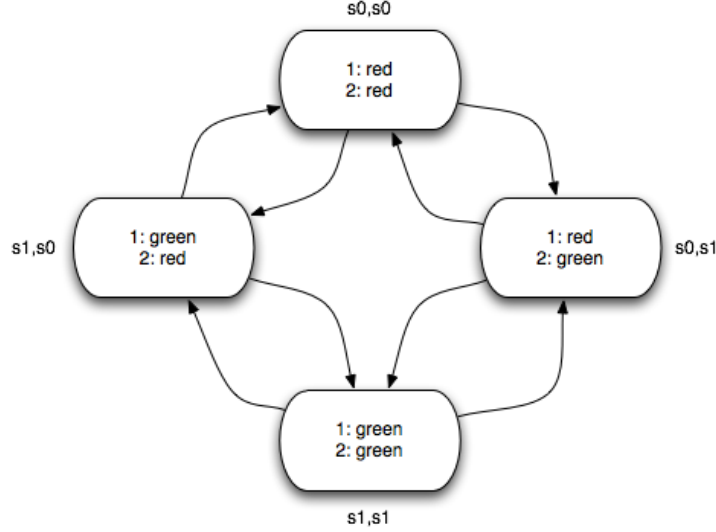
### 3.2.3 Reachable states

A state  $s$  is said to be reachable if there is an initial path that includes  $s$ . [Baier et al., 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  $\rightarrow$ .  $Reach(s)$  where  $s \in S$  can also be defined as all the states reachable from state  $s$  via the transition relation.

### 3.2.4 Modelling a concurrent system

Concurrent systems are systems in which multiple subsystems are active simultaneously, and each of these subsystems can potentially interact with each other. For instance, a software program can be thought of as a system, and multithreaded software program can be thought of as a concurrent system in

Figure 3.2: Asynchronous Concurrent traffic light transition system



which each thread is a subsystem. In the context of the traffic light model, we can construct a concurrent traffic light system in which two or more sequential traffic light models are active simultaneously.

The simplest method of composing the two or more sequential traffic light models is to interleave the executions of the models. This is expressed using the notation  $|||$ . Figure 3.2 shows the composition of two independent sequential traffic light models,  $TrafficLightSystem_1 ||| TrafficLightSystem_2$ . Each state in the concurrent model is described by the composition of the states of each individual subsystem. The actions of the individual models are also composed, leading a non-deterministic choice between the actions of each subsystem at any state in the system. A transition in a transition system of this nature corresponds to the transition of exactly one subsystem. [Baier et al., 2008] describes this as the *one-processor view* of interleaving, where no assumption can be made about which subsystem executes next and each subsystem executes asynchronously. This form of composition is useful when modelling asynchronous software [Baier et al., 2008] and asynchronous circuits [Clarke et al., 2000]. The formal definition of the asynchronous  $|||$  is defined in definition 1.

**Definition 2. Asynchronous interleaving operator  $|||$** , definition from [Baier et al., 2008].

The interleaving of two transition systems  $TS_1$  and  $TS_2$  where

$TS_1$  is a tuple  $(S_1, Act_1, \longrightarrow_1, I_1, AP_1, L_1)$  and  $TS_2$  is a tuple  $(S_2, Act_2, \longrightarrow_2, I_2, AP_2, L_2)$

is defined as the transition system

$TS_1 ||| TS_2 = (S_1 \times S_2, Act_1 \cup Act_2, \longrightarrow, I_1 \times I_2, AP_1 \cup AP_2, L)$

where the transition relation  $\longrightarrow$  is defined by the rules

$$\frac{s_1 \xrightarrow{a} s'_1}{\langle s_1, s_2 \rangle \xrightarrow{a} \langle s'_1, s_2 \rangle} \quad \text{and} \quad \frac{s_2 \xrightarrow{a} s'_2}{\langle s_1, s_2 \rangle \xrightarrow{a} \langle s_1, s'_2 \rangle}$$

and the labelling of each state is defined by  $L(\langle s_1, s_2 \rangle) = L(s_1) \cup L(s_2)$ .

Another possible composition of the models is a synchronous composition, where each transition consists of all concurrent subsystems transitioning simultaneously. In a synchronous concurrent transition system, a transition corresponds to a transition in all subsystems. This type of modelling is illustrated in figure 3.3, which shows a transition system for a synchronous concurrent traffic light system where the two lights are out of phase. The state space is reduced in size due to the restriction on the schedule of execution. Synchronous concurrent modelling is particularly useful when modelling synchronous hardware digital circuits because hardware circuits typically transition in a lock-step fashion according to a global clocking mechanism [Clarke et al., 2000].

### 3.2.5 Reactive systems

A typical traffic light system may allow users to press a wait button to signal the traffic light to enter the red state in order to allow safe crossing of a road. A traffic light system that allows some degree of control from users in the environment is an example of a *reactive system* [Manna and Pnueli, 1992]. A reactive system is a system that reacts to events in the environment and typically does not terminate. When modelling a system that does not terminate, the corresponding transition system will typically have no terminating states, i.e. every state has a possible successor state according to the transition relation.

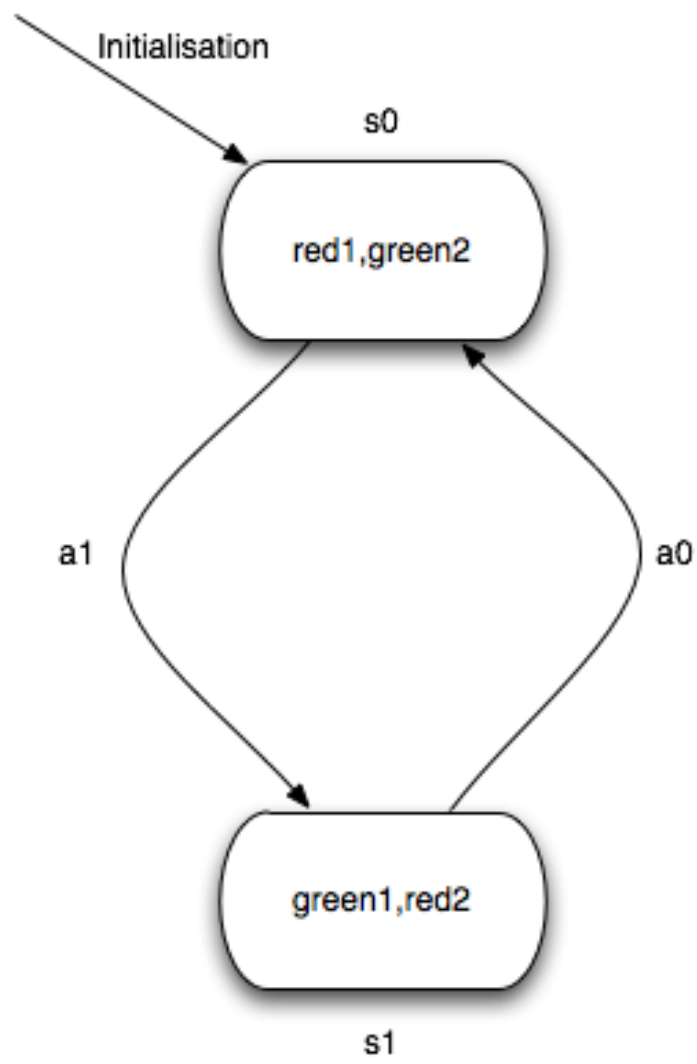
Modelling concurrent systems that react to events in an environment presents additional challenges. The traffic light system, for example, may allow users to press a wait button to signal the traffic light to enter the red state in order to allow safe crossing of a road. The wait button can be pressed at any time, asynchronously with respect to the execution of the traffic light system. When verifying this kind of system, little can be assumed about the schedule of the asynchronous button pressing.

### 3.2.6 Modelling software

The modelling discussed so far is useful for verifying control oriented models. When using model checking for the verification of software, the formalism used for modelling can be adapted to fit the purpose in order to increase the clarity of the transition system. In order to do this, the *program graph* [Baier et al., 2008] concept shall be introduced in order to simplify the modelling of software.

A typical software system will store typed variables  $Var$  of a certain domain, that represent a value over a particular range. For example, a software system may store an 8-bit *integer* variable with a range of 0-255. The control-flow of a program is typically dictated by conditions on the variables  $Var$  that the software stores, making the system more data-dependant than systems discussed previously.

Figure 3.3: Synchronous Concurrent traffic light transition system



A typical software system will transition through a number of states. The transitions between the states are dependent on conditions  $Cond(Var)$  over the typed variables  $Var$ .  $Cond(Var)$  is the set of all possible boolean conditions over variables in  $Var$ . The state of a software system consists of the location in the software program (e.g. program counter or line of code), and the evaluation  $Eval(Var)$  of any variables the program is storing. A software system is initialised in a starting location, with an initial evaluation of the variables.

The program graph concept is used to model typical software systems. The formal definition of a program graph is given below.

**Definition 3. Program graph**, definition from [Baier et al., 2008].

A *program graph* modelling a system storing typed variables  $Var$  is a tuple  $(Loc, Act, Effect, \hookrightarrow, Loc_0, g_0)$  where

- $Loc$  is a set of program locations
- $Act$  is a set of actions.
- $Effect : Act \times Eval(Var) \rightarrow Eval(Var)$  is the effect function that acts on variables  $Var$
- $\hookrightarrow \subseteq Loc \times Cond(Var) \times Act \times Loc$  is the conditional transition relation.
- $Loc_0 \subseteq Loc$  is the set of initial program locations.
- $g_0 \in Cond(Var)$  is the initial condition of the system.

$Loc$  is a set of program locations in a program of interest. This can correspond to line numbers in source code, or memory locations for assembly code.  $Act$  is a set of action labels. The  $Effect$  function describes the effect of actions in  $Act$  when applied to particular  $Eval(Var)$ , yielding a potentially different  $Eval(Var)$ . The conditional transition relation  $\hookrightarrow$  describes a set of guarded transitions, where the guards can refer to conditions in  $Cond(Var)$ . Each transition in  $\hookrightarrow$  can have associated actions that are executed if the transition is acted upon. Transitions occur between program locations.  $Loc_0$  is a set of initial program locations and  $g_0$  is the initial condition of the program graph.

To illustrate the program graph concept, consider the software program in figure 3.4. Figure 3.4 shows a simple program written in C. To model this software system as a program graph,  $Var = x$  as the software system stores only the variable  $x$ . The locations in the code that are important are those that dictate the control flow of the system. In this case,  $Loc = 8, 10|13, 17$  where the numbers refer to the line numbers of the code listing. Each of these lines of code expresses conditions on the current state of the program.  $10|13$  represents the multi-branching if statement beginning on line 10.

In this example, there are three explicit actions.  $Act = x ++, say\_hello(), say\_goodbye()$ . The effect function is as follows:  $Effect =$

- $Effect(x ++, \eta) = \eta[x := x + 1]$
- $Effect(say\_hello(), \eta) = \eta$
- $Effect(say\_goodbye(), \eta) = \eta$

Figure 3.4: Software system in C code

```

1 #include <stdio.h>
2
3 int x;
4
5 int main(int argc, char ** argv)
6 {
7     x = 0;
8     while (x < 2) {
9         x++;
10        if (x == 1) {
11            say_hello ();
12        }
13        else if (x == 2) {
14            say_goodbye ();
15        }
16    }
17    return 0;
18 }

```

where  $\eta$  is the current evaluation of variables  $Var$ . The notation  $\eta[x := x+1]$  represents the addition of 1 to  $x$  and keeping the current value of all other variables in  $Var$ . In this program graph,  $x++$  adds 1 to the variable  $x \in Var$  and both  $say\_hello()$  and  $say\_goodbye()$  have no effect on any program variables. The *Effect* functions describes the effect of all actions  $\alpha \in Act$  on the program variables  $Var$ .

The conditional transition relation  $\hookrightarrow$  describes the transitions possible from the current program location  $l$  depending on the current evaluation of the variables  $\eta$ . The conditional transition relation is a set of guarded transitions from location  $l$  to  $l'$ , with a potentially empty set of associated actions. This is written  $l \xrightarrow{g:\alpha} l'$  where  $g \in Cond(Var)$ ,  $\alpha \in Act$  and  $l, l' \in Loc$ . To model the fragment of C code,  $\hookrightarrow =$

- $8 \xrightarrow{x < 2: x++} 10|13$
- $10|13 \xrightarrow{x=1: say\_hello()} 8$
- $10|13 \xrightarrow{x=2: say\_goodbye()} 8$
- $8 \xrightarrow{x \geq 2} 17$

The first member of the relation describes the behaviour of the while loop, if  $x$  is less than 2 then increment  $x$ . In addition to altering the value of  $x$ , the rule makes the transition from location 8 to location 10|13. This transition is active if the guard evaluates to true. If more than one transition is active, then a transition is chosen non-deterministically from the active transitions. The second and third members of the relation describe the behaviour of the if/elseif control structure and both transitions progress to the location 8. The

Figure 3.5: Program graph for C fragment

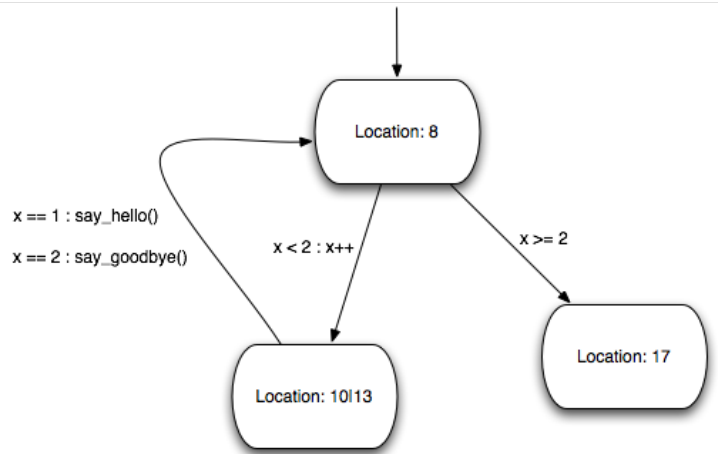
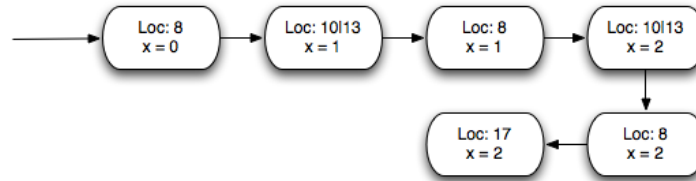


Figure 3.6: Expanded program graph for C fragment



last member describes the termination criterion of the program. This transition progresses to location 17 from which no transition is possible. When a state has no active transition, then that state is a terminal state.

The initial program location of the code that is of interest is line 8. Line 8 is where the main part of the control flow begins, after the initialisation of the variable  $x$ . Consequently, for the program graph model of the software system,  $Loc_0 = 8$ . The program is initialised outside of the main while loop, initialising the variable  $x$  to 0. In the program graph model, the initial condition  $g_0 = x = 0$ .

### Unfolding the program graph

Once the program graph is formalised, it must then be expanded into a model that more closely resembles a transition system in order to be checked. The states of the underlying transition system consists of a program location, and some evaluation of the variables in the system. The initial states of the transition system are made up of legal instantiations of the program graph [Baier et al., 2008]. For example, the initial state of the expanded transition system is

encoded by location 8, and the variable evaluation  $x = 0$  ( $\langle 8, x = 0 \rangle$ ).

The successor states to the initial state are generated by expanding the initial state according to the guarded transitions of the program graph. The set of successor states to the initial state is made up of the destination locations of all active conditional transitions that originate from the initial state, coupled with the new evaluation of *Var* as determined by the effect function associated with those transitions.

For example, in the software program example above, the only initial state is the state  $\langle 8, x = 0 \rangle$ . The evaluation of variables  $x = 0$  activates only one guard, the transition  $8 \xrightarrow{x < 2: x++} 10|13$ . The successor state is therefore made up of the program location 10|13 and the new evaluation of *Var* according to the effect function,  $x = 1$ . This yields the state  $\langle 10|13, x = 1 \rangle$ . If there are multiple guards active, i.e. the program graph is nondeterministic, then all possible successor states are expanded. This process is recursively applied until a previously expanded state is reached, or a terminal state is reached. Similar procedures to the unfolding a program graph are applied to modelling languages in order to obtain a transition system.

This process is demonstrated in figures 3.5 and 3.6. Figure 3.5 shows the program graph in diagrammatical form, and figure 3.6 shows the expanded transition system. In the program graph figure (Figure 3.5), each curved rectangle represents a program location, and each edge is labelled with the associated guard and actions. The expanded program graph (Figure 3.6), which would be generated in the verification stage, has just one execution in this case. In the expanded program graph, each curved rectangle is a state consisting of the program location and a valuation on variables. When the program graph is unfolded, a state is a program location in *Loc* coupled with an evaluation of variables  $Eval(Var)$ .

### 3.2.7 Modelling concurrent software

For the sake of brevity, a brief description of the modelling of concurrent software can be afforded here. The process of modelling concurrent software is similar to the process of constructing transition systems consisting of two or more processes. Program graphs are interleaved in order to produce a product program graph from which a transition system can be extracted. Guards on actions must be respected during the process. For a full treatment of this process, I refer the reader to [Baier et al., 2008]. [Baier et al., 2008] describes mechanisms for interprocess communication, including the modelling of message-passing and shared variable communication.

### 3.2.8 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.

### 3.3 Specification of properties

Once a model of the system has been constructed, the properties of the model to be verified must be expressed. Specifications are commonly formalised in a temporal logic [Baier et al., 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 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 both lights are red must be reachable”.

[Baier et al., 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* [Baier et al., 2008, Clarke et al., 2000]. An event  $e$  in this context is a particular path fragment  $\rho$ , 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 infinite paths that do not satisfy the given liveness property [Baier et al., 2008].

#### 3.3.1 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 [Baier et al., 2008, Clarke et al., 2000]. Temporal logics can be roughly split into two categories based on their notions of time. The first category is *linear*, where it is assumed that there is at most one possible successor state from the current state. The second is *branching*, where it is assumed that any number of successor states can arise from the current state through non-deterministic choice.

Most temporal logics allow the expression of the following two modalities [Baier et al., 2008, Clarke et al., 2000]. The first,  $\diamond$ , means eventually in the future from the current time. An example usage of this operator is  $\diamond a \wedge b$ , meaning that the proposition  $a \wedge b$  will eventually be satisfied by a state reachable from the “current” state. The second,  $\square$ , means always in the future from now onwards.  $\square a^b$  expresses that  $a^b$  holds true in every state reachable from the

Figure 3.7: Semantics of the  $\models$  relation, from [Baier et al., 2008]

- $\rho \models true$  - All paths satisfy *true*
- $\rho \models a$  iff  $a \in A_0$  - The first state satisfies *a*
- $\rho \models \varphi_1 \wedge \varphi_2$  iff  $\rho \models \varphi_1$  and  $\rho \models \varphi_2$
- $\rho \models \neg\varphi$  iff  $\rho \not\models \varphi$
- $\rho \models \bigcirc\varphi$  iff  $\rho[1] \models \varphi$
- $\rho \models \varphi_1 \bigcup \varphi_2$  iff  $\exists j \geq 0 \mid \rho[j] \models \varphi_2$  and  $\rho[i] \models \varphi_1$ , for all  $0 \leq i < j$ .

current state, including the current state.

### 3.3.2 Linear temporal logic

Linear Temporal Logic (LTL) is a temporal logic which is, from the name, in the linear category of temporal logics. LTL is made up of the traditional logical connectives such as  $\wedge$ ,  $\vee$  and  $\neg$  and temporal modalities [Clarke et al., 2000]. The two main temporal modalities are the *next* operator  $\bigcirc$  and the *until* operator  $\bigcup$ . One can refer to anything in the set  $AP$  when specifying properties. For example, if one wished to specify that some program variable  $a$  is less than 3 in the next step, it would be written as  $\bigcirc a < 3$ . Definition 4 shows the grammar of LTL, in BNF.

**Definition 4. LTL Grammar**, definition from [Baier et al., 2008].

$$\varphi ::= true \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc\varphi \mid \varphi_1 \bigcup \varphi_2$$

LTL formulae refer to paths in a transition system, and are either satisfied or violated by paths in a transition system [Baier et al., 2008]. Since the logic is referring to atomic propositions in  $AP$ , a path  $\rho$  in this instance is a sequence of atomic propositions  $\rho = A_0, A_1, A_2, \dots$ , where each  $A_i$  is an atomic proposition made up of propositions in  $AP$ . A particular  $A_i$  is analogous with a state in the system. The notation  $\rho[j\dots]$  refers to the path  $\rho[j\dots] = A_j, A_{j+1}, \dots$ . A LTL formula  $\varphi$  describes a set of paths. A path  $\rho$  satisfies  $\varphi$  if  $\rho$  is in the set of paths that  $\varphi$  describes. This is noted using the *satisfaction relation*  $\rho \models \varphi$ , meaning that the LTL formula  $\varphi$  is satisfied by  $\rho$ , or  $\rho$  models  $\varphi$ . The semantics of the satisfaction relation are described in figure 3.7.

It was discussed earlier that temporal logics allow the expression of the eventually and always operators. Expressing the operators  $\diamond$  and  $\square$  can be realised using the basic  $\bigcirc$  and  $\bigcup$  stated above.  $\diamond\varphi$  is defined as  $true \bigcup \varphi$ .  $\square\varphi$  is defined as  $\neg(\diamond\neg\varphi)$ .

I shall demonstrate the semantics of LTL using the concurrent traffic light model as an example. Figure 3.3 shows a synchronous variant of the concurrent traffic light system. In this system, two traffic lights are modelled out of phase. The set  $AP$  for this particular model is the set  $AP = red_1, green_1, red_2, green_2$ , and  $red_i$  indicates that traffic light  $i$ , 1 or 2, is in the red phase and so on.

A safety requirement of this particular system may be that “it is always the case that both traffic lights are not green at the same time”. This can be expressed in LTL using the  $\Box$  (“always”) operator, with appropriate atomic propositions.

$$\varphi_{\text{greensafe}} = \Box(\neg\text{green}_1 \vee \neg\text{green}_2)$$

During the verification phase, a transition system must be checked such that all paths from all initial states satisfy the safety property. In this case, the model must be checked such that all paths from state  $s_0 \models \varphi_{\text{greensafe}}$ . In this example, there is only one path through the system,  $\rho = (\text{red}_1 \wedge \text{green}_2), (\text{green}_1 \wedge \text{red}_2)$  and there is no state in  $\rho$  that violates  $\varphi_{\text{greensafe}}$ . Therefore, the transition system modelling the synchronous traffic light system  $TS_{\text{sync}} \models \varphi_{\text{greensafe}}$ , because all paths from all initial states in  $TS_{\text{sync}}$  satisfy  $\varphi_{\text{greensafe}}$ . If one was to check whether  $\varphi_{\text{greensafe}}$  was satisfied by the asynchronous concurrent traffic light model depicted in figure 3.2, we would find that the model violates  $\varphi_{\text{greensafe}}$ , because the transition system of the asynchronous concurrent traffic light model exhibits the following path:

$$(\text{red}_1 \wedge \text{red}_2), (\text{green}_1 \wedge \text{red}_2), (\text{green}_1 \wedge \text{green}_2)$$

As another example, let  $\varphi_{\text{eventuallygreen}} = \Diamond\text{green}_1$ . This LTL formula expresses that it is always the case that eventually traffic light 1 goes green. The synchronous traffic light model  $TS_{\text{sync}} \models \varphi_{\text{eventuallygreen}}$  because from the initial state  $s_0$  in  $TS_{\text{sync}}$ , the only path from  $s_0$  passes through a state where  $\text{green}_1$  is *true*. However, the asynchronous model in its current form does not satisfy  $\varphi_{\text{eventuallygreen}}$  since there is an infinite path

$$((\text{red}_1 \wedge \text{red}_2), (\text{red}_1 \wedge \text{green}_2), (\text{red}_1 \wedge \text{red}_2), (\text{red}_1 \wedge \text{green}_2))^\omega$$

The asynchronous model is capable of satisfying  $\varphi_{\text{eventuallygreen}}$ , however it is possible that on any particular execution of the system, only the subsystem of traffic light 2 transitions, starving traffic light 1. Starvation of a process or subsystem in a concurrent system is when a process is able to make progress, but is not “chosen” to do so. When modelling the traffic light system, the starvation of traffic light 1 could be considered and unrealistic execution, and measures must be taken to mitigate such an execution. For the asynchronous traffic light system to satisfy  $\varphi_{\text{eventuallygreen}}$ , additional assumptions must be made about the fairness of the transition system.

### 3.3.3 Concurrency and fairness

The notion of *fairness* [Baier et al., 2008, Clarke et al., 2000] when modelling a concurrent system refers to the choice of transitions made in a non-deterministic transition system, such as the asynchronous traffic light model depicted in figure 3.2. In this example, it was shown above that there was the potential for an *unfair* path that starves traffic light 1 from executing, even though there are active transitions allowing traffic light 1 to progress. This path made it impossible to satisfy  $\varphi_{\text{eventuallygreen}}$ .

To remedy the starvation problem, a set of *fairness assumptions* or *fairness constraints* must be enacted to ensure that starvation of subsystems does not

occur [Baier et al., 2008]. One requires that each subsystem of a concurrent system gets a *fair* chance to transition when it is possible to do so. In the traffic light example, one may require that each traffic light transitions *infinitely often*. This is equivalent to expressing that at no point from now on will either process be starved for an infinite amount of time [Baier et al., 2008, Clarke et al., 2000]. All paths that exhibit this behaviour are called *fair paths* [Baier et al., 2008]. Typically, when verifying concurrent systems using model checking, one is only interested in verifying the fair paths of the system, as unfair paths are often the unfeasible executions of the system. Therefore, one wishes to check a fair transition system, a transition system in which the only possible paths are fair paths [Baier et al., 2008].

Fairness constraints express sets of executions of transition systems in which the transition of subsystems are fair. Fairness constraints can be expressed in terms of the actions *Act* or states *S* of a transition system. There are various notions of fairness one can apply to a model. The condition that each traffic light can transition infinitely often without precondition is known as *unconditional fairness*, sometimes known as impartiality [Baier et al., 2008]. Two other major modes of fairness are strong-fairness, sometimes known as compassion, and weak-fairness, sometimes known as justice. Strong-fairness is that every process that can transition infinitely often does so infinitely often. Weak-fairness is every process that is continuously able to transition from a time onwards will do so infinitely often [Baier et al., 2008].

In LTL specifications, fairness constraints can be expressed in LTL. The expressions are as follows.  $\Phi$  and  $\Psi$  in the following are propositional formulae referring the atomic propositions in a given transition system. The following definitions are from [Baier et al., 2008].

$$\text{unconditional fairness} = \Box\Diamond\Psi$$

The unconditional fairness condition says that it is always the case that  $\Psi$  is eventually satisfied. An example is  $(\Box\Diamond\text{green}_1) \wedge (\Box\Diamond\text{green}_2)$  which states that it should always be the case that eventually  $\text{green}_1$  is satisfied, and likewise for  $\text{green}_2$ . Another way of stating unconditional fairness with respect to a traffic light system is that each of the traffic lights will cycle through their respective phases infinitely often, unconditionally [Baier et al., 2008].

$$\text{strong fairness} = \Box\Diamond\Phi \rightarrow \Box\Diamond\Psi$$

Strong fairness states that if it is always the case that eventually  $\Phi$  is satisfied, then it is always the case that eventually  $\Psi$  is satisfied. With regards to a reactive system, this is equivalent to saying that if a subsystem in a concurrent system is able to transition infinitely often, then it will do so infinitely often. In this sense, strong fairness can be used to resolve contention between processes [Baier et al., 2008], such as access to mutually exclusive critical sections.

$$\text{weak fairness} = \Diamond\Box\Phi \rightarrow \Box\Diamond\Psi$$

Weak fairness states that if eventually it is always the case that  $\Phi$  is satisfied, then it is always the case that eventually  $\Psi$  is satisfied. With regards to a reactive concurrent system, if a subsystem becomes continuously enabled at some time step, then that subsystem must be able to transition infinitely often

[Baier et al., 2008]. Continuously enabled means that at no time is the subsystem not enabled.

Fairness constraints can be combined via conjunction. Fairness constraints can be constructed using a mixture of unconditional, strong and weak fairness sub-constraints. Let *fair* be some fairness constraint. *fair* describes a set of paths that are considered to be fair paths. In terms of LTL specifications, *fair* describes the set of paths that are fair from a state *s*. When performing the verification stage of model checking (described later in section 3.4), verifying a transition system whilst considering fairness involves checking LTL properties against only the fair paths described by a fairness constraint *fair*.

The notion of fairness is in most cases dependant on the model one is verifying, and one must choose appropriately according to the reasonable usage of the system being modelled [Baier et al., 2008]. One must be careful when choosing fairness constraints. Too relaxed constraints can lead to the inability to verify liveness properties such as the  $\varphi_{eventuallygreen}$ . Conversely, too restrictive constraints may hide feasible paths in the modelled system that violate safety properties, such as race conditions that arise from an unforeseen interleaving of processes.

### 3.3.4 Branching temporal logic

Branching temporal logics (BTL) differ from linear temporal logics in their notions of time. As seen above, LTL refers to sequences of states, where there is only one possible successor state from each state. Branching temporal logics allow for reasoning about many possible successor states from any state. BTL refers to trees of states, allowing the expression of some properties that cannot be expressed in LTL [Baier et al., 2008]. A tree of states can be obtained from a transition system by unfolding the transition system from any initial state, much like the unfolding of a program graph described earlier. This unfolded tree represents all possible executions from that initial state. A traversal of the state tree root at the initial state of a transition system is a possible execution of that transition system from the rooted initial state.

BTL allows for reasoning over some or all computations from a state  $s \in S$  using existential and universal operators. For example,  $\exists \diamond \varphi$  means that there is at least one path that contains a state that satisfies  $\varphi$ .  $\forall \diamond \varphi$  means that all possible paths from a state contain a state that satisfies  $\varphi$ .

### 3.3.5 Computation tree logic

As the name suggests, Computation Tree Logic (CTL) is a branching temporal logic capable of reasoning about trees of executions. It is one of the more prominent branching temporal logics available, although others have been studied. CTL allows for the expression of important system properties that are not possible under LTL. CTL was originally used for checking concurrent systems as part of [Clarke and Emerson, 1981, Queille and Sifakis, 1982] and both works describe efficient algorithms for checking CTL properties.

CTL formulae consist of terms that refer to states as well as path operators, and makes an explicit distinction between the two [Baier et al., 2008]. CTL state formulae refer to atomic propositions and add existential and universal quantifiers over path formulae. CTL path formulae consist of LTL operators  $\bigcirc$  and  $\bigcup$

and both are semantically equivalent to their respective LTL counterparts. In CTL, all path formulae must be preceded by universal or existential quantifiers to be legal state formulae. The grammar for CTL is shown in definition 5.

**Definition 5. CTL Grammar**, definition from [Baier et al., 2008].

CTL state formulae:

$$\Phi ::= true \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \exists\varphi \mid \forall\varphi$$

where  $a \in AP$ . CTL path formulae:

$$\varphi ::= \bigcirc\Phi \mid \Phi_1 \bigcup \Phi_2$$

where  $\Phi, \Phi_1$  and  $\Phi_2$  are valid CTL state formulae.

An example of a CTL formula is  $\exists true \bigcup green_1$ , which expresses that there exists a path that contains a state that satisfies  $green_1$ . Both the synchronous and asynchronous traffic light models satisfy this CTL property, because a path can be found from the initial states of either model to a state that satisfies  $green_1$ . In another example, consider the formula  $\forall true \bigcup green_1$  that expresses that all possible paths must include a state that satisfies  $green_1$ . The synchronous traffic light model again satisfies this property, since the only path in that model includes a state that satisfies  $green_1$ . However, the asynchronous model includes infinite paths that do not satisfy  $green_1$  and therefore does not satisfy the universal property. This particular CTL property suffers from the same issue as the LTL property  $\varphi_{eventuallygreen}$ , in that fairness becomes an issue.

Fairness in CTL is handled in a different way to that of LTL, where fairness can be expressed in terms of an LTL formula. In CTL, rather than express the fairness constraints in the property being verified, the semantics of the existential and universal quantifiers are altered so that only fair paths are considered. To this effect, existential path quantifiers become “there exists a fair path” and universal path quantifiers become “for all fair paths” [Baier et al., 2008].

### 3.3.6 Extended computation tree logic

Extended computation tree logic (CTL\*) is a temporal logic that subsumes both LTL and CTL. CTL\* overcomes the limitations of CTL by allowing nesting of path formulae without any existential or universal qualification [Baier et al., 2008]. For example, the formula  $\forall \bigcirc \bigcirc green_1$  is a valid CTL\* formula, but is not a valid CTL formula due to the grammar restrictions. CTL\* has been shown to be more expressive than linear temporal logic, yet have the same complexity when it comes to verifying properties [Clarke et al., 2000, 1986].

### 3.3.7 Other path description languages

Whilst temporal logics are not the only way of specifying paths in a transition system, they offer a way of concisely expressing temporal properties over paths.

Temporal logics have mathematically sound manipulation rules for the composition and re-factoring of temporal logic formulae [Baier et al., 2008, Clarke et al., 2000].

Extensions to the traditional temporal logics have been proposed that can reason about the timing constraints of a system. Timed Computation Tree Logic (TCTL) coupled with appropriate extensions to the transition system concept can be used to achieve verification of timing constraints on system models [Baier et al., 2008], and linear time equivalents exist.

### 3.4 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  $\Phi$  [Baier et al., 2008, Clarke et al., 2000]. 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. *False positives* and *false negatives* are possible due to inadequate modelling or inaccurate specification. 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.

#### Checking invariant properties

A invariant is a property of the form  $\Box\Phi$  [Baier et al., 2008]. A property of this form describes that all states that are reachable from the initial state of the transition system must satisfy the condition  $\Phi$ .  $\Phi$  is a formula over the atomic propositions in  $AP$ , and can only refer to states, not paths or path fragments [Baier et al., 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 et al., 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  $\Phi$ . If the algorithm exhaustively checks all states and finds no violation of  $\Phi$ , then the algorithm returns *true*, indicating that  $TS \models \Phi$ .

### 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 in this form is known as a *bad prefix* [Baier et al., 2008].

The goal of checking a model for violations of a safety property  $\Phi_{safe}$  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 et al., 2008]. If a path  $\pi \in BadPrefixesTS$  is found, then  $\pi$  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 et al., 2008].

### Checking liveness properties

Liveness properties express that “something good will happen”. In order for a transition system to satisfy a liveness property  $\Phi$ , the transition system must be checked for the absence of a path that does not satisfy  $\Phi$ . 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 et al., 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.

## 3.4.1 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. [Baier et al., 2008, Clarke et al., 2000, Merz, 2001] excellent resources for these types of mechanisms. For the sake of brevity, they are not included here as they require in-depth explanations.

### 3.4.2 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 et al., 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.

### 3.4.3 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. When the problem is searching for violation of specifications, rather than exhaustively proving correctness, this allows guided model checking to potentially handle much larger models.

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., 2001]. 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 checking 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.

The better heuristics encode information about the target domain [Russell et al., 1995]. For example, when verifying software programs, it may help to use structural heuristics [Groce and Visser, 2002] in order to “head toward” parts of the state-space deemed interesting. Domain-independent heuristics exist, such as the hamming distance heuristic that can be used for symbolic model checking methods [Yang and Dill, 1998]. Similar heuristics can be used in the metaheuristic search techniques discussed in chapter 4.

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 when given an *admissible* heuristic will return the shortest path an error state. A heuristic is admissible if it does not overestimate the distance to an error state [Russell et al., 1995].

## 3.5 The strengths of model checking

### 3.5.1 General technique

Although model checking was born out of the need for an automatic verification technique for very specific problems [Clarke and Emerson, 1981, Queille and Sifakis, 1982], the fundamental aspects have been adapted used on a wide variety of challenges.

### 3.5.2 Partial verification

Model checking can be used to verify partial specifications [Baier et al., 2008, Clarke et al., 2000]. For example, a user may want to verify that a particular variable in a software program never exceeds a certain value. The user can specify this property alone and execute the verification stage, potentially on a model that has abstractions appropriate to the property being checked. This result can then be composed with other results as part of a larger verification effort.

### 3.5.3 Useful in debugging

The information generated from the model checking process, in the form of counterexamples, can be used to refine the model and or specification [Baier et al., 2008, Clarke et al., 2000]. A counterexample provides a precise sequence of events that lead to a property being violated. In an industrial setting, one can see model checkers used purely as a debugging tool, using the model checker as a simulated execution engine to find common software errors. The model checkers, in this case, are “curried” with a configuration relevant to the target language and problem at hand.

### 3.5.4 Potential for automation

In certain situations, the model checking process can be completely automatic, requiring little human interaction or skill [Baier et al., 2008]. For example, one can envisage developing a Java program, and have said program checked for defects, such a program deadlock, using a modified Java PathFinder [Visser et al., 2003]. In this situation, the model is generated automatically from the Java source code. The specification, for example ‘the system must not deadlock’, is an off-the-shelf specification catering from common software defects. The verification stage is also automatic, and could be left overnight to present a summary of the process for a developer in the morning.

## 3.6 Issues with model checking

### 3.6.1 Model inaccuracy

Model checking suffers from a number of problems. The first is that only a model of of the system is checked, not the actual system itself [Baier et al., 2008]. If the model of the system is not an accurate description of the system’s

behaviour, then performing verification on said model will yield misleading results. Misleading results can manifest themselves in the form of *false positives* and *false negatives*. As a consequence of this risk, organisations looking for high assurance during the V&V process will likely manually scrutinise the output of a model checker heavily to ensure model and specification accuracy. Also, further checking is needed for the actual construction of the system itself, regarding issues in traditional software testing such as acceptance and validation [Baier et al., 2008].

### 3.6.2 Somewhat limited to control oriented verification

Model checking runs into issues when verifying systems that are data intensive [Baier et al., 2008]. For instance, verifying a system that operates on a real-valued input will result in a great number of paths to check due to the density of the data type. Also, one cringes at the thought of having to model check a traditional database system, with the huge amount of states a database can be in. Typically, non-trivial verification tasks involve a great number of large to infinite domain data types.

### 3.6.3 State-space explosion problem

One can see from the program graph example before that for a program graph, the underlying explicit state transition system can grow in size. [Baier et al., 2008] summarises the expansions of various abstractions from a transition system, and how they can expand in systems with great numbers of states. The verification of non-trivial systems will, in most cases, require a model with a significant number of states [Baier et al., 2008].

The numbers of states, in general, increases exponentially as the number of variables and branches increases [Baier et al., 2008]. The problem is even more troublesome when analysing concurrent systems, such as software systems with multiple threads of control. When modelling concurrent systems, the subsystems are combined using a product of the underlying transition system. This means that the state-space grows exponentially with the number of subsystems. [Baier et al., 2008] outlines the issues when unfolding an abstract system into the concrete transition system, and shows how the resulting transition system grows with respect to certain features of the abstract model.

The complexity of the algorithms to check these models has also been shown to be sensitive to the number of states in a model. To check a model for safety properties, the entire state-space must be examined in order to guarantee safety. If the model grows exponentially, then the time to generate and check the model increases as well as the space to store the states of the model. This issue is known within the community as the *state-space explosion problem* [Baier et al., 2008, Clarke et al., 2000]. The majority of recent research focuses on the reduction of the state-space to allow the verification of larger, realistic systems. Methods of tackling this problem are discussed in later sections.

### 3.6.4 Other issues

Whilst the level of skill required by model checking may be less than that of a formal proof and refinement procedure, model checking still requires skills in

order to efficiently represent the model of the system [Baier et al., 2008]. Domain experts will likely be required in order to effectively abstract the system with minimal risk of model inaccuracy [Baier et al., 2008]. Also, skill is required when stating the specification in the various specification languages available [Baier et al., 2008]. Model checking is not a tool for checking that the specification is correct.

## 3.7 Tackling the issues

### 3.7.1 Reducing the likelihood of model inaccuracy

Mitigating the effects of this problem depends upon how the model itself is generated. If the model is generated automatically, through automatic translation or compilation as shown in the JPF [Visser et al., 2003], then the accuracy of the model depends solely on the accuracy of the translation. However, if the generation of the model has some manual aspect, the potential for inaccuracy is difficult to mitigate.

### 3.7.2 Reducing the state-space

One of the main focuses of model checking research is the reduction of the state-space generated during the verification stage. The methods one can use for reducing the state space of a model can depend the system one is trying to model, as some techniques focus on removing unnecessary detail from the system model. However, state space reducing mechanisms exist that are more general and be applied to wide variety of systems.

#### Symbolic model checking

Symbolic model checking aims to reduce the resources required for model checking by representing the state space symbolically rather than explicitly enumerating the states of a system [Burch et al., 1990]. Various forms of symbolic model checking mechanisms have surfaced over the years. Symbolic model checking has allowed the checking of huge models, with, the almost infamously touted,  $10^{20}$  states [Burch et al., 1990]. [Baier et al., 2008] describes this as model checking a system of sets of states and transitions between those sets.

The original symbolic model checking work was based upon *ordered binary decision diagrams* [Bryant, 1986] representations, and exploited known algorithms for analysing them. This approach encodes a transition system in a binary form and offers efficient algorithms for determining the satisfaction of various specifications. Symbolic approaches to model checking software have been proposed, namely a symbolic version [Anand et al., 2007] of the explicit state model checker JPF [Visser et al., 2003]. Since then, other representations have been realised. [Biere et al., 1999] describe a representation based upon propositional logic, translating a graph structure algorithmically to a propositional logic formula. The model checking procedure, in this instance, is reduced to satisfying the newly generated formula [Biere et al., 1999].

### Partial order reduction

The subsystems of a concurrent system may spend some time processing locally, i.e. not interacting with other processes. For example, a server process may perform local processing to do some housekeeping or indexing in preparation for a future request. When verifying properties of a particular class, for example synchronisation properties, local processing in subsystems may become irrelevant and therefore can be removed from the system model. POR is particularly useful when verifying synchronisation properties of a concurrent system, as an exhaustive partial order reduction can yield the synchronisation structure of a concurrent system, eliminating all other irrelevant information.

Partial order reduction (POR) is a technique for reducing the number of states and transitions in a concurrent system model [Peled, 1998]. The technique aims to reduce the amount of harmless interleaving transitions in a model. The interleaving of transitions  $P$  and  $Q$  are considered harmless when the execution of  $P$  or  $Q$  in either order displays equivalent behaviour with respect to a specification.  $P$  and  $Q$  are sets of transitions in the transition system of two subsystems in the concurrent model. If the interleaving of transitions  $P$  and  $Q$  is deemed to be harmless, then a model checker only needs to check one possible interleaving, conceptually collapsing every possible interleaving of  $P$  and  $Q$  into a single representative interleaving [Peled, 1993]. Algorithms for POR exist for off-line and on-the-fly model checking [Peled, 1996].

### Abstraction

Abstraction is the process of removing irrelevant information from a model with regards to the specification under scrutiny. This process can be highly manual in nature, potentially requiring domain experts as well as model checking experts. When model checking software, there is the potential for traditional static analysis techniques to be used. Partial order reduction can be considered as an abstraction process, representing many interleavings as a single relevant interleaving. This process can be automated, and has been in the JPF model checker.

### Model decomposition

Depending on the system under scrutiny, it may be possible to verify parts of the system model and then compose the results to gain assurance of the overall product [Clarke et al., 1989]. For example, a system may exhibit explicit modes, a mode being a particular behaviour during operation. For example, when landing an aircraft, an automated aircraft control system may switch to a landing mode, as opposed to a cruise mode when cruising. Systems that consist of many identical processes can also be targeted in this manner.

## 3.8 Summary

In this chapter, I have shown how model checking techniques can be used to verify properties of concurrent systems. I have described the application of model checking to software, and highlighted some problems (and some solutions) with the mechanism. I have shown how model checking can be used to verify

a wide range of properties, and how model checking can be automated when verifying concurrent software.

The verification stage techniques can be divided into two categories. The first category are the techniques that perform a full conformance check with the specification, exhaustively checking the execution space for conformance. The second category is the techniques that aim at finding errors when a transition system is too large to check exhaustively, such as the guided model checking techniques described above. Guided model checking techniques can provide a systematic way of detecting and reproducing errors in concurrent software automatically. Guided model checking can be thought of as sampling promising areas of the execution space in order to discover an error.

The second category is of high interest to the author. Using guided model checking techniques can provide underlying components of a general concurrent debugger in commonly used programming environments such as Eclipse of Microsoft's Visual Studio. The JPF [Visser et al., 2003] project has an Eclipse plug-in that can be used to model check parts of a Java program. The Banda/Bogor [Corbett et al., 2000, Robby and Hatcliff., 2003] project also has similar features for Eclipse. The use of tools such as these can provide excellent debugging information for a programmer, showing exact interleavings of threads or processes that cause a concurrent bug. This technique holds advantages over the static and dynamic techniques described in the previous chapter, allowing for the discovery of concurrent faults with potentially less computational effort.

In the next chapter, I will introduce mechanisms that can be used for searching over the execution space of a concurrent program. These mechanisms can be used in a similar way to how DFS and A\* search are used in guided model checking, effectively sampling areas of the execution space that are more likely to contain an error. The techniques described go beyond the local view approach of traditional guided model checking algorithms, potentially combining information from multiple paths in the execution space.

## Chapter 4

# Metaheuristic search

### 4.1 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. An example problem may be to find the fittest solution possible in a solution space. 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 *MaxOnes*( $v$ ) problem, also known as One-Max [Eiben and Smith, 2003] or the bit counting problem [Chen et al., 2002]. *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, *MaxOnes*(100) = 1 and *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 *MaxOnes*( $w$ ) is the length of vector  $w$ . The *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  $n$ . *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 *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. *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.

### 4.1.1 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.

## 4.2 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  $C$  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 the 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 by 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 [Reeves, 1993, Russell et al., 1995].

### 4.2.1 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 optima is defined as a point in the search space that is not a global optima and whose neighbourhood consists of only less fit solutions [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 optima. 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 optima.

### 4.2.2 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 4.1(c). 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

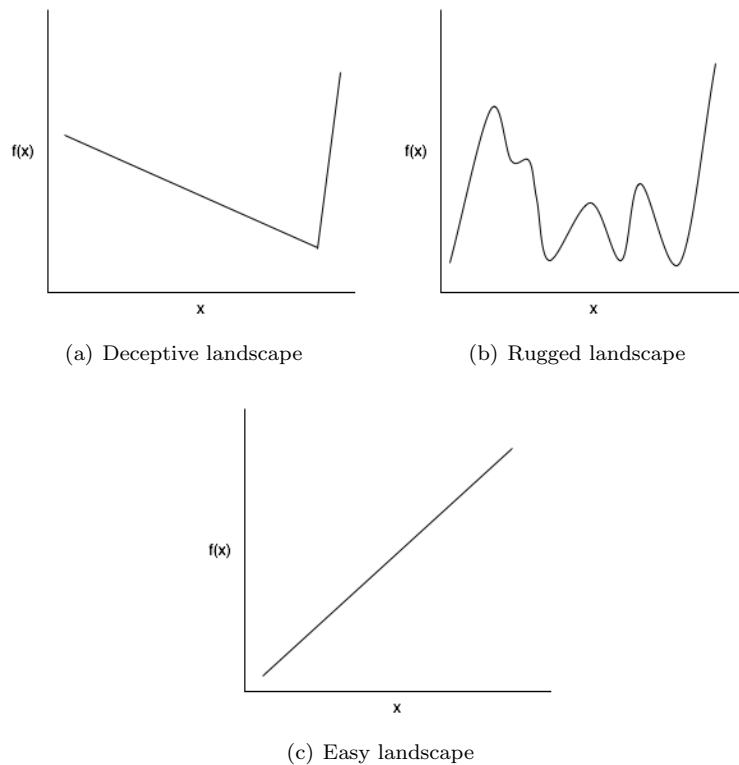


Figure 4.1: Example landscapes

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 optima, 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 4.1(a). 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 4.1(b) depicts a rugged landscape, and figure 4.1(c) 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 optima increases over a landscape that contains fewer local optima.

### 4.2.3 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, a change in a single bit can have a massive change in the phenotype. For instance, when flipping the most significant bit of  $0100b = 4d$ , the resultant integer is  $1100b = 12d$ . However, when using a gray coding scheme, a single bit change can yield a smaller change in the phenotype. Using this scheme, the neighbourhood of a particular solution  $x$  is more likely to be of similar fitness to  $x$ .

### 4.2.4 Simulated annealing

In order to increase the chances of converging on the global optima, 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* [Kirkpatrick 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.

## 4.3 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 stores restricted movements to previously explored parts of the search space, or other

restricted areas. 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 if the move is in the tabu list.

## 4.4 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].

### 4.4.1 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 creature's 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.

Figure 4.2: 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 }
```

A mutation operator takes a single solution and outputs a similar but different solution.

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 4.2 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.

#### 4.4.2 Genetic algorithms

Genetic algorithms (GAs) are one example of an EA, using selection pressure and simulated reproduction to sample a solution space. Genetic algorithms are a population-based metaheuristic search technique. Genetic algorithms were born out of work by Holland et al [Holland, 1975] and have become one of the more prominent optimisation techniques with numerous triumphs over various search and optimisation problems. Although the original technique was developed and studied for processing bit strings, the method provides a general framework for searching over other solution representations.

GAs use observations of how the genetic material (*genotype*) of living crea-

tures influences the physical attributes of those living creatures (the *phenotype*) [Reeves, 1993]. Features in the phenotype are determined by a decoding of features in the genotype. To combine desirable features in the phenotype, one must combine the the relevant aspects of the genotype. When breeding dogs for example, one will choose to combine the genetic material of parents with desirable characteristics in the hope that the offspring of said parents exhibit some combination of those characteristics.

### Anatomy of a GA

The basic concepts and components of a GA will now be described. GAs operate on a population of solutions in the solution space, biologically analogous to genotypes in the natural world. I shall refer to genotypes as individuals, solutions or genotypes from here onwards. A genotype describes the features a solution exhibits, and is typically decomposable to allow for recombination. A genotype is made up of genes, For example, a possible genotype of the *MaxOnes* problem is the same representation that can be used for local search, a bit string.

GAs refer to a phenotype, the result of some translation process applied to a genotype. The phenotype is a decoding of the genotype, analogous to the decoding of DNA to the physical characteristics of living creatures in the natural world. In order to assess the fitness of a genotype, it is translated into a phenotype and the phenotype is exercised. In the *MaxOnes* example, the translation and exercising process is combined into the fitness function described in the local search section. This approach is common in GA usage.

A GA transitions through a series of time steps, or iterations, much like local search. The population at iteration  $n$  is referred to as *generation  $n$*  in the context of GAs. Initially (generation 0), the population consists of a random sample of solutions in the solution space, or can be seeded with the best known solutions. At each generation, the individuals in the population are translated into their respective phenotypes and evaluated according to the fitness function. Then, individuals are selected to participate in the next generation. This selection process is probabilistic in nature and is typically biased toward the fitter individuals in the population.

The selected individuals are then subjected to *genetic operators*, instances of the recombination and mutation operators of an EA. In traditional GAs, a round of reproduction occurs where the selected individuals are combined to form new children. This is analogous to breeding, forming children in the natural world. The genetic operators combine features of the parent individuals in a meaningful way in the hope of yielding children with the characteristics that make their parents fit. The selection and breeding process is repeated until a new population is generated. This population becomes the next generation.

Once the next generation is constructed, a mutation operator is applied to the new population. The mutation operator, given an individual, returns a slightly modified variant of that individual. The mutation operator is applied to each individual post breeding with some probability. The probability of mutation is typically low. Mutation is seen as way of creating new genetic material within the population, sampling from new parts of the solution space. The process of evaluation, selection, reproduction and mutation continues until some termination criteria are met. The termination criteria can be expressed in similar terms to local search. The pseudo-code for a typical GA is shown in

Figure 4.3: 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 genetic operators
10         new_pop = GEN_OP(parents);
11         //New solutions become new generation
12         population = new_pop;
13     }

```

figure 4.3, and closely resembles the EA pseudo-code depicted in figure 4.2.

There are numerous selection mechanisms to choose from when using GAs. The selection mechanism is typically stochastic, but biased toward to the fitter members of the population. One of the more widely used mechanisms is tournament selection. Tournament selection randomly chooses  $n$  individuals from the population and pits them against each other. The fittest member of the  $n$  individuals is selected for breeding in the next generation. Other mechanisms include fitness proportionate selection, also known as roulette wheel selection, where each individual is assigned a probability of selection proportional to their fitness relative to the rest of the population. There are a variety of selection mechanisms, and this aspect of a GA is generally regarded as a parameter of the algorithm.

Genetic operators for breeding are chosen that facilitate the greatest chance of meaningful combination of the fitter attributes of the parents  $a$  and  $b$ . Typically, some form of *crossover operator* is employed. A crossover operator swaps features between parents, creating one or more children. An example crossover operator for combining bit strings is *single-point crossover*. Single-point crossover combines one half of parent  $a$  with the other half of parent  $b$ , and vice versa, to create two children. Mutation is akin to sampling the neighbourhood of a parent to obtain a mutant and replacing said parent with the mutant. For example, when using a bit string genotype, flipping a randomly selected bit can yield a mutant.. Figure 4.4 shows example genetic operators manipulating bit string genotypes.

### Elitism

During the course of an evolutionary run, good solutions have the potential to be destroyed during recombination and mutation stages. It may be desirable to propagate a representative of the best or  $n$  best solutions from one generation to the next, to avoid “losing” these solutions. This approach is known as *elitism* [Eiben and Smith, 2003]. This amounts to maintaining a representative of the best solutions found so far in the population at all times. This could be achieved

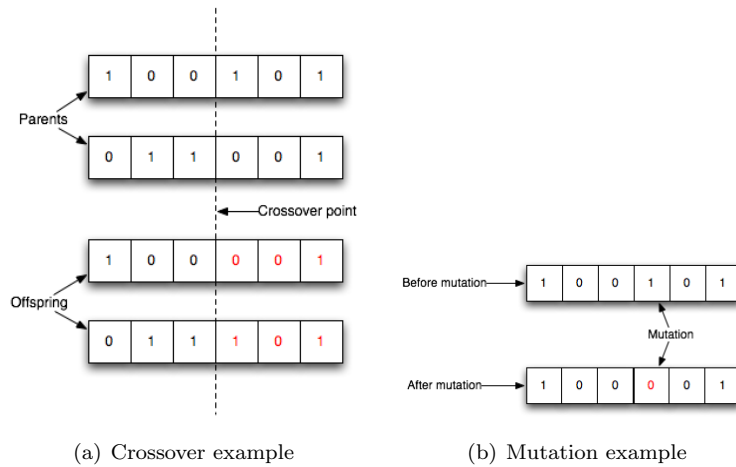


Figure 4.4: Genetic operators

by replacing the lowest ranking solution in the population with the best found so far before selection occurs. This approach exploits a small part of the history of an evolutionary run [Eiben and Smith, 2003].

### GA theory

In Holland's original work [Holland, 1975], *schemata* were introduced as a concept to aid in the analysis of the functioning of a GA. Schemata, plural for schema, define sets of bit strings. For example, the schema  $10*$  defines the set of bit strings  $(100, 101)$  and the schema  $101$  defines the set of bit strings  $(101)$ . The  $*$  in this notation defines a *don't care*, which can be either a 0 or a 1. Schemata have an *order* and a *length*. The order is defined as the number of defined bits in a schema, and the length is defined as the distance between the first and last defined bits. For example,  $1 * 1$  is of order 2 and length 2, and  $1 * * 0 2$  is of order 3 and length 4.

Bit string individuals in a population can be instances of many schemata. For example, the individual  $10$  is an instance of  $10$ ,  $1*$ ,  $*0$  and  $**$ . A bit string individual of length  $n$  is an instance of  $2^n$  different schemata. A population of  $P$  individuals can contain up to  $P2^n$ , however it is likely that schemata are present in more than one individual, reducing this number. The fitness of a schema  $s$  is the average fitness of the individuals in the population that are instances of schema  $s$ .

Holland argued that the ability of a GA to form good solutions can be attributed to how schemata propagate through the population during the course of a GA execution. Holland showed that over successive generations with selection pressure and genetic operators, a GA increases the number of instances of fit schema within the population. A GA is implicitly evaluating all of the schema represented in the population, and combining the fit schema using the crossover operator. *Schema theorem* describes models for the propagation of schema during the course of a GA run, showing that relatively fit, short low order schemata will have increased number of instances in successive generations. A GA does

this without explicitly storing information about individual schemata. The ability of a GA to test many schemata in few generations is referred to as *intrinsic parallelism*, also known as *implicit parallelism*.

Schemata can be thought of as sub-problems that are composed to form a larger solution [Whitley, 1991]. The assumption that an individual containing two fit schemata will be fitter than an individual containing only one of those schemata is called the *building-block hypothesis* [Reeves, 1993]. Building blocks are high quality solutions to sub-problems, and are described as above average fitness schemata when using bit strings. Fit schemata of low order and low length are more desirable than schemata of equal fitness but higher order and higher length. The argument is that low order, low length schema are less likely to be destroyed by genetic operations such as crossover and mutation. Various mechanisms can be employed to shorten the length of schema in order to make them more robust to genetic operations.

Schema theorem as laid out by Holland [Holland, 1975] applies only to bit string genotypes. Whilst ultimately all population representations are processed as bit strings on a standard binary computer, abstractions from bit string schema theory are needed for other representations. For example, the genetic programming community have developed a GP specific adaption of schema [Langdon and Poli, 2002]. Despite the criticisms of schema theory, the ability of a GA to construct good solutions in a building-block type fashion is supported by a substantial amount of empirical evidence.

## 4.5 Estimation of distribution algorithms

Estimation of distribution algorithms (EDAs) [Mühlenbein and Paafi, 1996] operate on a probabilistic model of the solution space. 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].

### 4.5.1 Example EDA

To describe the function of an EDA, the simplest form of an EDA will be described. The simple EDA mimics the behaviour of a simple genetic algorithm. The algorithm samples solution spaces described by strings of bits. The EDA works in a very similar fashion to a simple GA, and the pseudo-code for the algorithm can be found in figure 4.5. Most EDAs follow the algorithm outlined in figure 4.5 [Pelikan et al., 2002].

The algorithm holds an initial model of a good 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 real-valued number  $r$  between 0 and 1 is generated. If  $r < m_i$  then  $c_i$  is set to 1,  $r \geq m_i$  then  $c_i$

Figure 4.5: 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     }

```

is set to 0. Each  $c_i$  in the model represents the probability of the  $i^{\text{th}}$  bit of a candidate solution being a 1.

The candidate solution set is generated to be of adequate size. The larger the sample, the more accurate the model. 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  $c_i$ s, where  $c$  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....

### 4.5.2 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, and the above is example is of the UDMA variety Miihlenbein and Paafi [1996]. 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].

A step up from this kind of model is to assume some kind of pair-wise interaction occurs 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]. 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].

### 4.5.3 Strengths

#### Explicit building blocks

EDAs can automatically and explicitly learn relationships between variables in the solution space. For instance, the BOA algorithm can be seen as representing building blocks as sets of connected nodes in the bayesian network. BOA can do this without any domain knowledge specified by the user. Other EDAs, such as MIMIC, can learn more basic relationships between variables in the solution space.

#### Resource efficiency

EDAs have the potential to be resource efficient. The cGA algorithm [Harik et al., 1998], for instance, aims at reducing the memory requirements for binary string problems with little interaction between variables. However, this advantage breaks down when considering more complex models, such as BOA. The data structures required to learn and store a bayesian network far surpass the resources required for a cGA or UDMA approach.

### 4.5.4 Weaknesses

#### Choice of model

The choice of model to build and exploit in an EDA can be difficult. Simple choices are available for solutions spaces that can be encoded as bit strings and a vector of numbers. However, representing programs and other structures require great effort. [Salustowicz and Schmidhuber, 1997] and [Sastry and Goldberg, 2003] have both made inroads into the modelling of program structures. Great effort may be required in order to find a suitable modelling mechanism for a new target domain.

#### Sample diversity

[Yuan and Gallagher, 2005] have shown how the diversity of the sample generated by the sampling mechanism can be crucial in the performance of an EDA. Attention must be paid to this problem in order to avoid problems with early convergence. The treatment of this problem is not always clear, especially when dealing with new problem domains requiring novel modelling mechanisms.

## 4.6 Summary

In this chapter I presented a brief outline of a few metaheuristic search techniques. In this next chapter, I will outline how these techniques have been applied to the problem of model checking, and conclude by highlighting some gaps in the research.

## Chapter 5

# Previous work applying search to find concurrency bugs

### 5.1 Searching for deadlock

The property “a system must never deadlock” 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.

In order to focus the traversal on states which are more like 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 few resources in order to obtain a precise ordering of events that leads to the deadlock of a system.

### 5.1.1 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 approaches highlight the huge reduction in computational resources required for 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.

#### 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 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  $n$ th 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  $i$ th element  $v_i$ ,  $v_i$  is simply multiplied by the number of transitions available at the 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. [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.

#### 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

Figure 5.1: Fitness function for deadlock from [Alba et al., 2008]  
 $f(x) = \text{deadlock} + \text{numblocked} + \frac{1}{1+\text{pathlen}}$

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 5.1. In this equation, *deadlock* is set to 1 when the final state is in deadlock, *numblocked* is the number of blocked 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.

## Evaluation

Both works report that using a GA to find deadlock can result in using fewer computational resources than an exhaustive state-space traversal. It was shown that a GA based approach to find deadlock managed to discover errors in systems that were too large to handle using an exhaustive search of the state-space. However, little effort was given to determine whether a GA based approach has any advantage over guided model checking approaches such as best-first search and A\*. This is especially worrying since the fitness functions used are directly applicable to a guided model checking algorithm. [Godefroid and Khurshid, 2004] did evaluate the approach against a completely random search of the state-space, showing a clear advantage for the GA approach. [Alba et al., 2008] provide a brief comparison of their approach to the work of [Godefroid and Khurshid, 2004] showing some improvement over the earlier work.

### 5.1.2 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 least costing paths through graphs that represent a solution. A good description of the technique can be found in [Dorigo and Di Caro, 1999]. In this instance, the ant agents are tasked with co-operatively discovering short paths to safety violations in a transition system.

[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.

[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].

## 5.2 Searching for liveness errors

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.

### 5.2.1 Ant colony optimisation

[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 [Baier et al., 2008, Vardf and Wolper, 1986]. In this approach, the negation of the liveness property is transformed into a Büchi automaton, 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.

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.

[Alba and Chicano, 2008] report excellent results compared to traditional exhaustive mechanisms such as depth-first search, and a depth-first search that uses the Promela construct that eliminates part of the search space. 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. However, it is not clear how well the ACOhg-live algorithm would do without the special Promela features, and this may be crucial as the approach may not translate well to model checking other systems.

## 5.3 Heuristics

Work on heuristics for guided model checking will be generally applicable to incomplete mechanisms such as GAs. For instance, any of the heuristics mentioned below can be used in addition to the generic fitness function described in figure 5.1.

### 5.3.1 Using static analysis

Program slicing has been used in order to guide a model checking algorithm Millett and Teitelbaum [1998]. Program slices highlight parts of code that are influential in leading to a condition at a specified point of interest in the code [Tip, 1994, Weiser, 1981]. For example, one may generate a program slice that consists of parts of the code that are likely to lead to a deadlock condition. The highlighted code constitutes a *program slice*, and is usually an estimate [Weiser, 1981]. A model checker can favour paths in a model that execute parts of a given slice, leading to fewer states being expanded when searching for an erroneous state.

### 5.3.2 User annotations

[Yang and Dill, 1998] propose the use of *guideposts*, annotations of the model given by a system designer, in order to further guide the search toward suspicious parts of the state-space. The guideposts can be in the form of assertions in the model code, such as conditions on variables, or just simple tags in the code. The more guideposts a state and the prefix path activate (satisfy), the higher the probability of an error being present and the guided search should expand that state. [Yang and Dill, 1998] show experimental results indicating that the use of guideposts can reduce the computational effort required to find erroneous states.

### 5.3.3 Learning from mistakes

[Seppi et al., 2006] present an interesting work that proposes a metaheuristic that can be used in exhaustive guided model checkers that assign rankings to states as the search progresses. The technique attempts to learn from mistakes by characterising heuristic ranking errors in the form of a probability density function. Information is gathered from the sibling states of a state  $s$  under scrutiny. The siblings of a state  $x$  are all the children of the parent of  $x$ .

The authors claim that the set of the children of any parent are closely related, and that the distribution of a set of heuristic estimates can be used as an indicator as to how accurate a heuristic is. The authors use the distribution of heuristic values for the set of children to revise the estimate of each individual child. The estimate of each child is “pulled” toward the mean of the sibling set

of that child. The revised heuristic is then used to determine which child is expanded next.

The authors present empirical evidence for the efficacy of the technique. The authors apply the mechanism to a guided search strategy using an inadmissible heuristic, naming the algorithm Bayesian Heuristic Search (BHS). The inadmissible heuristic is a property-based heuristic and is detailed in the paper [Seppi et al., 2006]. The authors compare BHS to a standard breadth-first search and a best-first search based on the same inadmissible heuristic, checking a set of 22 modelled systems. The authors show that the revised heuristic estimates lead to fewer states being explored before finding an error on the majority of the systems checked (15 out of 22 systems). The BHS approach also allows models that do not fit into memory to be checked, by focusing the search effectively toward areas of the search space likely to contain errors.

The authors note that this result could have wider implications, and highlight the possible application to search algorithms such as random walk. A similar concept could be applied to population-based search algorithms, given some mechanism to determine how similar solutions are that is not the fitness function. However, the overheads for such an approach may outweigh the benefits and requires further investigation. The approach does create a new parameter which represents the general variance of the heuristic over all states in the model. The authors treat this as a tuneable parameter, and give some analysis as to the significance of the parameter.

## 5.4 Reducing the state space

### 5.4.1 Reducing OBDD size

When producing a OBDD representation of a system for symbolic model checking, the order of the variables or function arguments in the OBDD dictates the number of nodes in the OBDD. [Costa et al., 2001] apply a GA to search over the possible orderings, minimising the number of nodes in the diagram. The ordering of variables is encoded in a permutation based chromosome, and specialist crossover and mutation operators are employed. [Costa et al., 2001] shows experimentally that more efficient OBDDs can be generated from solutions obtained from one of the prominent traditional heuristic mechanisms in the field. No evidence is given as to whether the GA method starting from scratch can perform equally well.

## 5.5 Summary and conclusion

The work applying metaheuristic search techniques to concurrent verification seems to be preliminary in nature, showing that the techniques can find faults in concurrent systems. Some of the work has shown that the use of optimisations from model checking literature can be used to improve the efficacy of the respective techniques. Some work has been done on using observations of the history of the search process to further enhance a search.

I find it surprising that there is a lack of attempts at using local search on this problem. Researchers seem to have skipped this method altogether, favouring the more prominent population based approaches available. Work is

required to determine whether local search can play a role in finding safety and liveness errors. From the literature I have read, I have seen little on what makes a concurrent fault difficult to detect. Are some examples of deadlock harder to detect than others? Are there metrics that define the difficulty of finding a particular concurrent fault?

From reading the model checking literature and reviewing preliminary work, I believe a major aspect to be tackled at the moment is the solution representation. With the exception of ACO, an algorithm designed for searching graphs, previous work focuses on explicit path-based representations of the solution space. With the state of the art approach, it seems that traditional crossover and mutation operators can become hugely destructive. Because of this, recombination of solutions may not be effective, hampering the search effort.

I believe that the application of EDAs in this area could be fruitful. An EDA in this context would build a model of promising areas of the state space relevant to some goal (e.g. finding safety violations). How the state space is modelled is an open question. A probabilistic model of the explicit path space could be constructed, in a similar way to the GA approaches mentioned above. There is the opportunity to explicitly model interleavings of process actions. This can be done using a probability distribution over strings using a representation such as probabilistic finite automata [Ron et al., 1995]. Using this approach, bad prefixes to safety violations could be explicitly modelled.

The use of metaheuristic search techniques in the presence of traditional state space reduction methods could lead to greater scalability. [Chicano and Alba, 2008b] have shown that partial order reduction can be used in conjunction with ACO to produce counterexamples effectively. The use of metaheuristic mechanisms and symbolic state space reduction methods could prove fruitful, as symbolic methods can yield dramatic reductions in state-space size.

Finally, gaining insight into concurrent faults using visualisation and effective comprehension of the search process could be useful. Effective reduction of complex interleavings into data a developer quickly can understand could allow for greater assurance in the behaviour of a particular software program. With visualisation comes the opportunity for interaction, allowing a human participant to guide the search. For instance, a developer could dynamically add guideposts to areas of the state-space that looks promising.

# Chapter 6

## Proposal

### 6.1 Research strands

The underlying hypothesis that I will investigate during this research is:

*Probabilistic Model Building Genetic Algorithms (PMBGAs or EDAs) are an effective mechanism for searching transition systems.*

In order to provide evidence as to the validity of this hypothesis, I propose 4 strands of research. During each strand of work I will aim to write and submit one or more papers to relevant conferences.

1. Explore the applicability of EDA-based techniques to verifying properties of concurrent systems using model checking techniques.
2. Compare various methods of modelling the execution space of a system.
3. Evaluating the scalability of the approach on systems with a large number of explicit states.
4. Explore the efficacy of EDA-based techniques in combination with traditional state reducing methods as well as static and dynamic analysis techniques.

#### 6.1.1 EDA-based model checking

This strand will be the initial focus of the work. An EDA-based model checking approach will be applied to a series of benchmark problems and compared with existing techniques. The method will use EDA techniques to build a model of the execution space can describe areas of interest in the real execution space. The result of this work will be a proof-of-concept design and implementation along with scientifically rigorous evaluation, demonstrating the applicability of model building based approaches to verifying properties of concurrent systems. The work will be demonstrated on a set of example problems, some of which are outlined in section 6.3.

The hypothesis to be investigated as part of this strand is as follows:

*An EDA-based model checking approach can detect concurrent faults using fewer resources than state of the art methods..*

Resources include memory, disk space and time. During the course of this work, I will demonstrate the applicability of EDA-based model checking techniques to the two main classes of temporal property, safety and liveness. I will compare the approach against known techniques, such as guided model checking and the GA and ACO approaches outlined in the previous work chapter. Care and attention must be placed on avoiding some of the pitfalls of EDA-based approaches, such as lack of diversity in the sampling mechanism.

I estimate that this strand of work will take 6 months to complete.

### 6.1.2 Modelling the execution space

This strand of research will build upon the implementation derived from the EDA-based model checking strand. The work as part of this strand will focus on comparing different ways of modelling the execution space of a concurrent program. A systematic comparison of approaches will be performed. The approaches to be compared will include modelling the execution space as explicit paths, interleavings of actions, as well as state based modelling. Of particular interest will be the modelling of the execution space as a distribution over strings of varying length. The lengths will range from what can be thought of as a Markov chain-like model (strings of length one) to large probabilistic finite automata based approaches.

The hypothesis to be investigated as part of this strand is as follows:

*As the size of a transition system increases, the complexity of the model required to effectively detect errors also increases..*

To investigate this hypothesis, a large amount of empirical analysis may be required. A set of systems ranging from simple to complex will be required. The metric of complexity of a system will need to be clearly defined, possibly in terms of the size of the state space or the subtlety of the error. The complexity of the execution space modelling mechanisms must also be defined, possibly in terms of granularity of the approach or the amount of computational resources used. One interesting question here is that are there notable phase changes with regards to the complexity of a transition system that affect the sophistication of the model required to search it?

The computational requirements for this kind of experimentation are discussed in section 6.4. I estimate that this strand of work will take 4 months to complete.

### 6.1.3 Scalability

This strand of research will investigate the scalability of the work produced so far. The target of the work will be large systems in terms of lines of code. The DEOS operating system (discussed later) would be a suitable target here, consisting of many lines of code with a large number of program states. Other benchmark problems, such as the dining philosophers problems, could be scaled up to a large size in order to truly exercise the method.

The hypothesis to be investigated as part of this strand is as follows:

*An EDA-based model checking mechanism can outperform the state of the art on large explicit state model checking problems.*

Performance can be measured in terms of resource usage, time required or number of states visited. The state of the art in this case is traditional deterministic model checking techniques, guided model checking techniques and the work outlined in the chapter 5.

I estimate that this strand of work will take 4 months to complete.

#### **6.1.4 Exploiting traditional techniques**

The use of metaheuristic search techniques with symbolic model checking methods will be the focus of this strand. A scientifically rigorous comparison of an EDA-based symbolic model checking techniques against traditional and heuristic symbolic model checking techniques will be performed. Other potential avenues of research consist of using dynamic and static analysis techniques to aid the search process.

The hypothesis to be investigated as part of this strand is as follows:

*EDA-based symbolic model checking techniques outperform traditional symbolic model checking techniques in finding property violations.*

Performance can be measured in terms of resource usage, time required or number of states visited. I estimate that this strand of research will require 6 months to complete.

#### **6.1.5 Thesis write-up**

I estimate that 6 months will be required to write up the thesis.

## **6.2 Tools and support**

### **6.2.1 Model checking frameworks**

A number of packages implementing model checking techniques are freely available complete with source code. The main tools targeted in the literature are designed with heuristic search extensions in mind. The languages used to implement the popular model checking frameworks are a mix of high level modern languages, such as Java and C. This allows for a modular approach to the implementation of any necessary work, allowing for easy extension of existing code.

#### **Java PathFinder**

The Java PathFinder (JPF) [Havelund, 1999, Visser et al., 2003] model checking tool is a popular target for the literature discussed in the previous work chapter. JPF performs explicit-state on-the-fly model checking procedures on models expressed in the Java language, a popular mainstream language. JPF includes implementations of advanced model checking mechanisms such as on-the-fly partial order reduction. The distribution of JPF also includes some benchmark problems, such as an implementation of the dining philosophers problem with the potential for deadlock. The tool is open source with an active community

continuing development. Extensions to JPF have been proposed that operate over symbolic search spaces [Anand et al., 2007].

### **HSF-SPIN**

HSF-SPIN [Edelkamp et al., 2001] is an extension to the SPIN model checker that incorporates directed model checking techniques. HSF-SPIN checks Promela specifications for conformance to LTL properties, and is aimed at verifying distributed software systems. The implementation is open source and coded in ANSI C, but does not seem to be actively maintained. However, the last update to HSF-SPIN is stable and is used in some previous work applying heuristic techniques to model checking.

## **6.2.2 Metaheuristic search frameworks**

Using an existing framework has numerous advantages over a bespoke solution. An existing framework is more likely to have been tested by a wider user base, and used in previous research. A widely tested framework is likely to produce more reliable results when running experiments, minimising the risk of coding errors affecting conclusions. It is favourable in my opinion to use a general framework for evolutionary computation with extensions relevant to the proposed work. What follows is a short overview of some available frameworks.

### **ECJ**

ECJ [Luke et al., 2007] is a framework that implements a wide variety of evolutionary search mechanisms in the Java programming language. The framework is general and flexible enough to support a wide variety of metaheuristic paradigms, including ant colony optimisation and multi-objective optimisation techniques. The framework also supports multi-processor computing, both locally on a multi-core computer and over TCP/IP. The ECJ framework has parameter file support, facilitating large scale experimentation.

I have worked with ECJ during my undergraduate studies and I am highly skilled in extending the framework to suit various needs. I am also highly skilled with the Java programming language, and this knowledge will be useful when extending ECJ for the proposed work. ECJ as of June 2009 does not contain any EDA or PMBGA implementations, but preliminary implementation work has given me confidence in my ability to extend ECJ in a short amount of time.

### **Open BEAGLE**

A number of C++ evolutionary toolkits exists, the most notable being Open BEAGLE [Gagne and Parizeau, 2002]. Open BEAGLE (OB) offers much the same features as ECJ, and is easily extensible. OB also offers parameter files for easy tweaking of algorithm parameters. There may also be notable efficiency gains over Java based frameworks, as architecture specific optimisation can be achieved. I have some skill with C-based languages, making Open BEAGLE a suitable candidate framework.

### 6.2.3 Summary

For the proposed work, the implementation will require the bridging of search and model checking frameworks. The choice of evolutionary framework, therefore, will largely depend on language compatibility with the chosen model checking framework. There is a clear match between the Java based frameworks, ECJ and JPF, and between the C-based frameworks, Open BEAGLE and HSF-SPIN. If a framework turns out to be lacking the flexibility required to achieve a certain goal, a bespoke solution would not be difficult to implement as I have the relevant skills in either language.

## 6.3 Benchmark problems

In order to evaluate any approach against previous work, a set of benchmark problems must be obtained. The benchmark problems in this case must consist of a system with a documented concurrent bug. In order to demonstrate the scalability of the approach, the systems examined must exhibit large state spaces. In order to show that an approach has industrial applicability, it must be evaluated on real industrial systems, or systems exhibiting problems of interest to industry. This may be difficult to define, but is discussed briefly below.

The previous work chapter of this report highlighted work in the area of applying search techniques to the verification of properties in concurrent systems. In the highlighted work, various problems are used to aid in the comparison of techniques. For instance, the dining philosophers problem is a well known deadlock problem, and models are available for the JPF model checker and Promela [Alba and Chicano, 2007, Alba et al., 2008]. Other instances of well-known deadlock problems are available for testing, including the stable marriage problem. There are a number of models available that exhibit liveness violations, and these are used when comparing approaches to finding liveness errors in large models.

In order to show the applicability of a particular approach to real-world problems, an evaluation of the approach on real-world industrial examples is needed. [Eytani et al., 2007] have compiled a set of concurrent programs with bugs created by students as part of an assignment. The suite of programs exhibit a wide variety of bugs, including deadlock and race conditions. The JPF tool includes an implementation of a real-time operating system [Visser et al., 2003], consisting of 1000 lines of code. The model also includes an environment for the OS to interact with, and exhibits a subtle concurrent bug. Demonstrating the efficacy of an approach on examples such as these will increase confidence in the real-world applicability of said approach.

## 6.4 Computational requirements

The search techniques discussed in this report can place high demands on computational resources. In addition, the search techniques discussed are also stochastic in nature, leading to variation in the results between different executions. In order to accurately describe the behaviour of any of these techniques, experiments must be repeated independently enough times in order to be confident in any estimated parameters from the results obtained.

In addition to this, the target problem of model checking can also place demands on computational resources, particularly when the transition system generated by a model has a large number of states. Fortunately, tools and literature are available in order for me to estimate the amount of resources likely to be required for the proposed work.

#### 6.4.1 CPU requirements

The CPU requirements of this proposal must take into account the model checking and metaheuristic search processing requirements, which can be large. [Alba et al., 2008] report times of ranging from 0.01s to around 50s for the run time of their GA-based experiments on moderate computational hardware comparable to the staff workstations of the department. Similar times are reported for the ACO-based experiments [Alba and Chicano, 2007]. I estimate that running times of experiments will be around this number or possibly slightly more.

#### 6.4.2 Memory requirements

[Alba et al., 2008] have quoted moderate amounts of memory usage whilst performing their experiments. For instance, when executing the GA based technique on a dining philosophers model with 8 philosophers, the overall memory usage for a population size of 50 was around 100MB. This included the memory overhead for the GA, but the majority of the memory usage resulted from the expansion of states in the JPF model checker. The authors note that it may be possible to gain more efficiency by changing the way JPF handles the state expansion. Similar memory requirements are posted in [Chicano and Alba, 2008a] and in both works a single computer was used to perform the experiments.

[Visser et al., 2003] report on the model checking of a large real-time operating system, DEOS, consisting of 1000+ lines of code. A Java implementation of the DEOS operating system is given as an example in the JPF model checker, as well as the dining philosophers example used in [Alba et al., 2008]. The DEOS example is much larger in terms of lines of code than the dining philosophers, ranging from 1000+ to around 20+ respectively. It is not clear as to how much memory experimenting with the DEOS system would consume, but in my opinion it is likely to be in the order of gigabytes rather than megabytes. This will pose a substantial challenge to any metaheuristic approach, and may require some of the distributed solutions mentioned below.

#### 6.4.3 Distributed computation

The figures quoted above are encouraging for the smaller scale experimentation, allowing for some of the earlier work in the proposal to be executed on a single computer. However, the distribution of computational work will be necessary if any large scale experimentation is to be achieved.

#### Distributed experimentation

When running a large number of experiments, it is possible to run a number of isolated experiments concurrently depending on the available resources. A number of tools exists for doing this kind of distributed computing, for example the

Sun Grid Engine [Microsystems), 2001]. SGE is implemented across a number of servers in the department, and the tool offers automation and results collection facilities. With this kind of framework available, it is not unfeasible to run large numbers of experiments in order to investigate the proposed hypotheses.

### **Distributing the search process**

It is possible to distribute the computation of the search algorithms mentioned in this report across a number of processors. Conceptually, this can happen in a number of ways and is typically defined by the granularity of the distribution. One method is to maintain *islands* of populations, and distribute the islands amongst the processors. Each processor is therefore processing an entire population, and can exchange information about the best solutions in their respective models. There are options for migration to occur between the islands, exchanging parts of the model in some fashion. The islands can exchange model information in a lock step fashion, i.e. every  $n$  generations, or asynchronously according to some schedule.

Of particular interest in this proposal is the distribution of the fitness function, i.e. the model checking phase during evaluation. This can be realised using a master-slave model of distributed computation, where a master server stores the model and deals with the updating of the model, and slave nodes sample the model and evaluate the sample individuals. This results in processors evaluating individual paths in the execution space. In this model, the samples generated from the current model are explicitly shared amongst the available processors. When using an island model, sample evaluation is implicitly distributed.

### **Distributing the model checking phase**

Whilst it is possible to distribute the path exploration phase of the model checking process, most effort focuses on distributing the state space amongst processors for the checking of properties [Barnat et al., 2001]. This kind of distribution is aimed at large scale transition systems, and the verification of a variety of temporal properties. The work proposed in this report, however, uses the model checking tool as a kind of simulator. Distributing the computation at this level will likely be fruitless. In my opinion, it is best to focus on the higher granularities of distribution, such as distributed experimentation or master slave models.

## **6.4.4 Available hardware**

### **Personal hardware**

My personal laptop should be more than sufficient to handle the small scale experimentation proposed. The machine contains 4GB of RAM and a fast CPU. In addition to this, staff workstations and a number of compute servers can be used for small scale experimentation.

### **Department compute servers**

For distributed computation, the department as well as the SEBASE project have a number of compute servers available for use. The machines consist of

a number of cores with shared memory, linked by TCP/IP networking. The available hardware will more than adequately support the possible distribution mechanisms discussed above. Also available is the White Rose Grid [Dew et al., 2003], a Beowulf cluster offering a large amount of computational resources that can be used for the computational work of this proposal.

#### **Commodity graphics hardware**

GPGPU [Luebke et al., 2004] based computing facilities are another possible way to cut down the computational complexity of an experiment. Technologies like CUDA and OpenCL can be leveraged to distribute computation amongst the many cores of a graphics processing unit. Technologies as such as these may be useful when implementing the experimental aspect of the proposed work.

#### **6.4.5 Project management**

In order to manage the code and write-up of my research, I shall use a distributed version control system. This will allow backups before any major changes are made, mitigating the risk of disastrous changes.

### **6.5 Summary**

This report has described the chosen field of research, giving an overview of the literature to the problem of verifying concurrent systems. Also outlined is the field of metaheuristic search mechanisms, including some applications of said mechanisms to the verification of concurrent systems. A proposal for further research is given, including a summary of the resources required.

# Bibliography

- 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, 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. *Lecture notes in computer science*, pages 68–79, 2003.
- S. Anand, C.S. Pasareanu, and W. Visser. JPF-SE: A symbolic execution extension to Java PathFinder. *Lecture Notes in Computer Science*, 4424:134, 2007.
- 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. *Lecture notes in computer science*, pages 297–308, 2003.
- C. Baier, J.P. Katoen, and I. NetLibrary. *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. 1994.
- J. Barnat, L. Brim, and J. Stribrna. Distributed LTL model-checking in SPIN. *Lecture Notes in Computer Science*, pages 200–216, 2001.
- 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. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207. Springer-Verlag London, UK, 1999.
- 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.
- R.E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.
- JR Burch, EM Clarke, KL McMillan, DL Dill, and LJ Hwang. Symbolic model checking: 10 20 states and beyond. In *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on*, pages 428–439, 1990.
- 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 *Evolutionary Computation, 2008. CEC 2008. (IEEE World Congress on Computational Intelligence)*. *IEEE Congress on*, pages 2997–3004, 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. ISSN 0020-0190. doi: <http://dx.doi.org/10.1016/j.ipl.2007.11.015>.
- E. Clarke, D. Long, and K. McMillan. Compositional model checking. In *Proceedings of the Fourth Annual Symposium on Logic in computer science table of contents*, pages 353–362. IEEE Press Piscataway, NJ, USA, 1989.
- Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, January 2000. ISBN 0262032708.
- 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.
- EM Clarke, EA Emerson, and AP Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

- 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.
- J. Clarke, 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.
- EG Coffman and MJ Elphick. System Deadlocks. *Computing*, 3:67–78, 1971.
- JC Corbett, MB Hatcliff, J. Laubach, S. Pasareanu, and C.S.R.H. Zheng. Banderas: Extracting finite-state models from Java source code. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 439–448, 2000.
- U. Costa, D. Deharbe, and A. Moreira. Advances in BDD reduction using parallel genetic algorithm. In *Proceedings of the 10 th International Workshop on Logic Synthesis (IWLS)*, 2001.
- 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. 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.
- PM Dew, JG Schmidt, M. Thompson, and P. Morris. The White Rose Grid: Practice and Experience. In *UK eScience-All Hands Meeting*. Citeseer, 2003.
- 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 meta-heuristic. 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, 2001.
- 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. ISBN 0-7695-1926-1.
- 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. ISSN 0362-1340. doi: <http://doi.acm.org/10.1145/543552.512558>.
- C. Gagne and M. Parizeau. Open BEAGLE: A new versatile C++ framework for evolutionary computation. In *Late-Breaking Papers of the 2002 Genetic and Evolutionary Computation Conference (GECCO 2002), New York (NY), USA*. Citeseer, 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.
- James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005. ISBN 0321246780.
- A. Groce and W. Visser. Model checking Java programs using structural heuristics. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 12–21. ACM New York, NY, USA, 2002.
- 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. *Lecture notes in computer science*, pages 152–152, 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.
- S. Kirkpatrick, 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.
- 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.
- D. Luebke, M. Harris, J. Krüger, T. Purcell, N. Govindaraju, I. Buck, C. Woolley, and A. Lefohn. GPGPU: general purpose computation on graphics hardware. In *International Conference on Computer Graphics and Interactive Techniques*. ACM New York, NY, USA, 2004.
- Sean Luke, Liviu Panait, Gabriel Balan, and Et. Ecj 16: A java-based evolutionary computation research system, 2007.
- J. Magee and J. Kramer. *Concurrency: State Models And Java Programs*. Wiley New York, 2006.
- Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. Springer, 1992.
- P. McMinn. Search-based software test data generation: a survey. *Software Testing, Verification & Reliability*, 14(2):105–156, 2004.
- S. Merz. Model checking: A tutorial overview. *Lecture Notes In Computer Science*, pages 3–38, 2001.

- W. Gentzsch (Sun Microsystems). Sun grid engine: Towards creating a compute power grid. In *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, page 35, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1010-8.
- H. Mühlenbein and G. Paafi. From recombination of genes to the estimation of distributions I. Binary parameters. *Parallel Problem Solving from Nature-PPSN IV*, pages 178–187, 1996.
- L.I. Millett and T. Teitelbaum. Slicing Promela and its applications to model checking, simulation, and protocol understanding. In *Proceedings of the 4th International SPIN Workshop*, pages 75–83, 1998.
- D. Peled. All from one, one for all: on model checking using representatives. *Lecture Notes in Computer Science*, pages 409–409, 1993.
- D. Peled. Combining partial order reductions with on-the-fly model-checking. *Formal Methods in System Design*, 8(1):39–64, 1996.
- Doron A. Peled. Ten years of partial order reduction. *Lecture notes in computer science*, pages 17–28, 1998.
- 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.
- 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.
- M B. Dwyer Robby and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *Proceedings of the 9th European Software Engineering Conference held jointly with the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 267–276, 2003.
- D. Ron, Y. Singer, and N. Tishby. On the learnability and usage of acyclic probabilistic finite automata. In *Proceedings of the eighth annual conference on Computational learning theory*, pages 31–40. ACM New York, NY, USA, 1995.
- 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.
- R. Salustowicz and J. Schmidhuber. Probabilistic incremental program evolution. *Evolutionary Computation*, 5(2):123–141, 1997.
- K. Sastry and D.E. Goldberg. Probabilistic model building and competent genetic programming. *Genetic Programming Theory and Practice*, pages 205–220, 2003.
- 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.
- K. Seppi, M. Jones, and P. Lamborn. Guided model checking with a bayesian meta-heuristic. *Fundamenta Informaticae*, 70(1):111–126, 2006.
- Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley, December 2004. ISBN 0471694665. URL <http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0471694665>.
- 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.
- B. Yuan and M. Gallagher. On the importance of diversity maintenance in estimation of distribution algorithms. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 719–726. ACM New York, NY, USA, 2005.

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