Isaac Dunn

Dynamic Partial-Order Reduction for Model Checking

Computer Science Tripos – Part II

Clare College

May 2016

Proforma

Name:	Isaac Dunn
College:	Clare College
Project Title:	Dynamic Partial-Order Reduction for Model Checking
Examination:	Computer Science Tripos – Part II, June 2016
Word Count:	11,624[1]
Originator:	Isaac Dunn[2]
Supervisors:	Dr. Jonathan Hayman & Prof. Glynn Winskel

Original Aim of the Project

The original aim of the project was to implement a simple model-checking algorithm, and to understand and implement the dynamic partial-order reduction model-checking algorithm.

Work Completed

The simple and dynamic partial-order reduction model-checking algorithms were implemented, successfully verifying programs written in a concurrent programming language. Several extensions were also completed, including implementations of stateful model checking, sleep sets, static partial-order reduction, and visualisation of counter-example traces. A quantitative comparison of the performances of the implemented algorithms was also completed.

Special Difficulties

No special difficulties were encountered.

[1] The program `wc` was run on a plain-text file of Chapters 1–5, with diagram text manually removed.
[2] With guidance from Dr. Jonathan Hayman.

Declaration of Originality

I, Isaac Dunn of Clare College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed:

Date:

Acknowledgements

I would like to thank my supervisor, Dr. Jonathan Hayman, for all his help and guidance this year. I am also grateful to Prof. Glynn Winskel for agreeing to nominally supervise me, in case Dr. Hayman could not continue for the entire year. Finally, I would like to express my gratitude to anyone who has ever taught me anything, without whom I would know nothing.

Contents

Chapter 1

Introduction

After reading this chapter, you should understand:

- what model checking is and why it is useful;
- what the state explosion problem is;
- how partial-order reduction techniques address the state explosion problem; and
- the aims of my project.

1.1 Model Checking

There are many reasons for ensuring that software and hardware systems have as few defects as possible. Failure of a safety- or mission-critical system can have devastating consequences, but even the presence of bugs in everyday commercial systems has serious economic implications. In 2002, software bugs were estimated to cost the U.S. economy $59.5 billion per year [29].

Complementary to traditional testing and peer review, formal verification methods have been increasingly used to minimise the number of defects in a software or hardware system. A formal method uses automated mathematical analysis to increase the confidence that a system meets its specification, or shows that it does not.

Model checking is a verification method that arose in the early 1980s, motivated by the desire to automatically verify concurrent programs [6]. A model-checking algorithm performs an exhaustive enumeration of a given system's behaviour to show that it satisfies some property, usually expressed in a temporal logic.

The idea of applying Tarski fixed-point techniques [28] to automatically check that a program, expressed as a finite-state system, meets a specification expressed in a propositional temporal logic was independently introduced by Clarke and Emerson [7] and Queille and Sifakis [26]. Vardi and Wolper later devised an algorithm that performs model checking efficiently for the other main type of temporal logic, which treats time in terms of continuous paths rather than branching trees [33]. Other important milestones in the history of model checking include McMillan's symbolic model checking [22], which allowed much larger systems to be verified; bounded model checking, introduced by Biere et al. [2], which allowed model checking to find bugs in systems

too large for full verification; automatic counterexample-guided abstraction refinement by Clarke et al. [5], which allowed simpler but in some sense equivalent models (abstractions) to be easily used; and various partial-order techniques, discussed below.

Model checking has been used with considerable success in practice, for instance in finding a long-standing flaw in the Needham-Schroeder security protocol [19], integration into the IBM [27] and Amazon [24] development processes, and verification of software controlling a flood barrier near Rotterdam [17], a NASA Mars Rover [3], and an "industrial-size design of an ultra-modern satellite platform" [11].

1.2 The State Explosion Problem

Unlike conventional testing, which notoriously struggles to expose concurrency bugs, model checking can verify that concurrent programs are free of deadlocks, livelocks, or race conditions that could result in an error. However, model checking suffers from what is known as the *state explosion problem*.

Consider a concurrent system consisting of n asynchronous processes, each of which has k instructions to execute before it terminates. Unfortunately for model checking, there are

$$N = \frac{(kn)!}{(k!)^n}$$

different interleavings of the instructions from the different threads. If each different interleaving can result in a different final system state, then each of these must be checked, meaning that the running time of any model checker is N in the worse case. Figure 1.1 shows how N grows for $k = 1$.

1.3 Partial-Order Reduction

Model checking is still used in practice, thanks to various techniques that mitigate the state explosion problem. Partial-order reduction techniques reduce the number of interleavings that must be explored by using the fact that the order of execution of some instructions makes no difference to the end result. Fully exploiting this idea can very effectively mitigate the state explosion problem, as shown in Figure 1.2.

Valmari's stubborn sets [32], Peled's ample sets [25], and Godefroid's persistent and sleep sets [14] are instances of partial-order reduction; incidentally, the latter two are explored as extensions to my project.

1.4 My Project

At the 2005 Principles of Programming Languages (POPL) conference, Flanagan and Godefroid presented a paper titled Dynamic Partial-Order Reduction for Model Checking Software [13], introducing an algorithm which uses information gathered dynamically during its exploration of a state space to perform partial-order reduction more

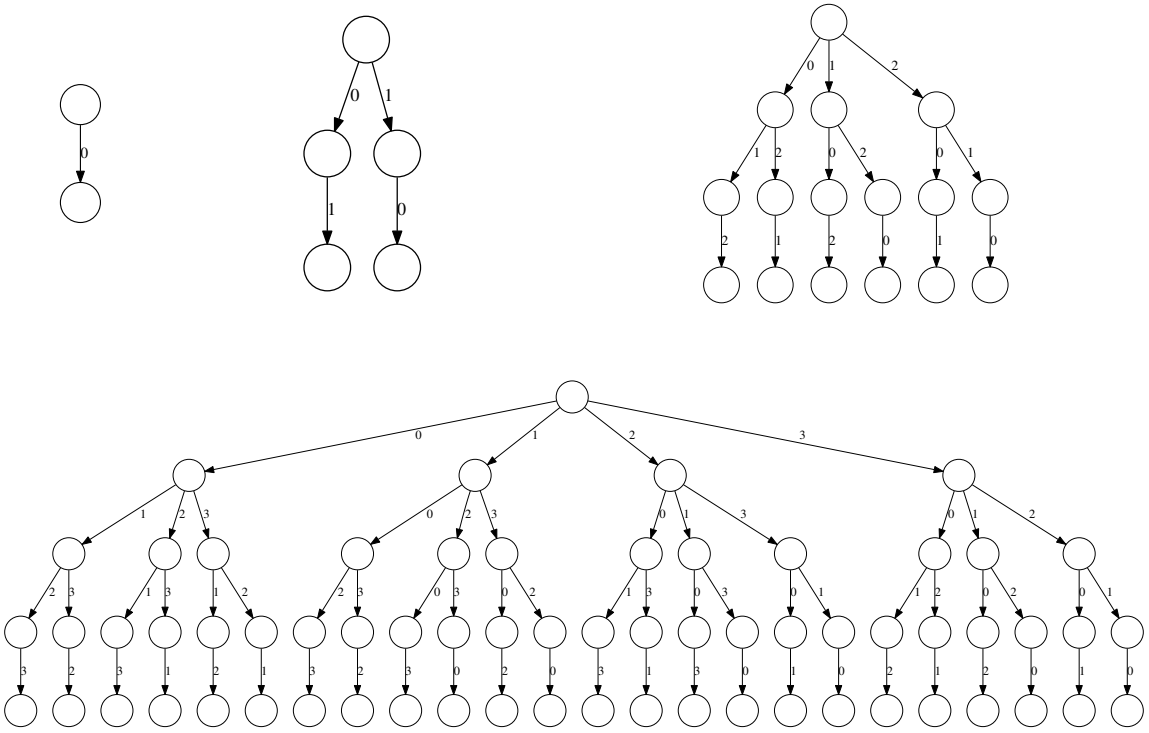


Figure 1.1: Diagrams showing the possible interleavings of one, two, three and four events.

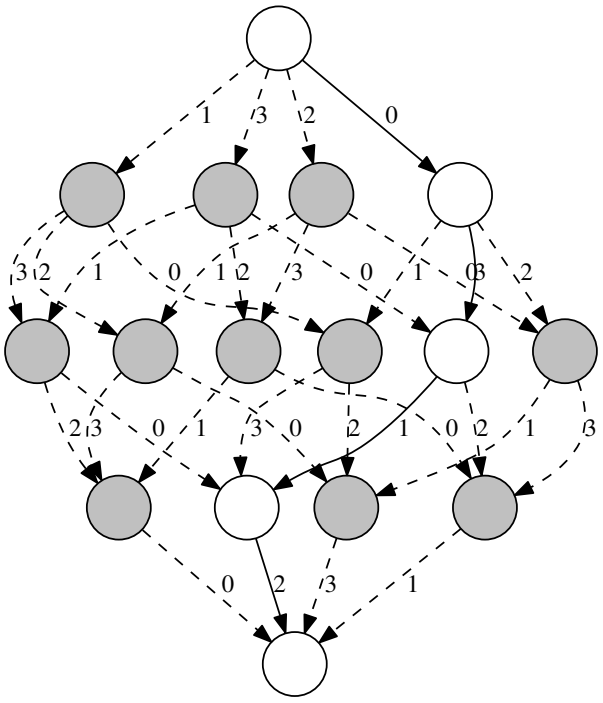


Figure 1.2: If the order of execution of four instructions or events makes no difference to the end result, it suffices to check only one interleaving.

effectively than existing algorithms. In particular, it initially assumes that all possible interleavings of instructions in a concurrent program will give the same result, and only explores alternative executions when it discovers pairs of instructions that might give different results when executed in a different order.

The original aim of my project was to fully implement the dynamic partial-order reduction (DPOR) algorithm presented in the POPL paper from scratch, and to evaluate its performance against a simple model-checker that performs no partial-order reduction. These goals were successfully achieved, and several other model-checking algorithms and techniques were also implemented as extensions. This allowed a fuller evaluation of the performance of DPOR in comparison to other techniques.

Chapter 2

Preparation

After reading this chapter, you should understand:

- the work I undertook before beginning the implementation of my project;
- the background theory of model checking relevant to my project;
- the persistent-set and sleep-set partial-order reduction techniques; and
- the basic operation of the DPOR algorithm.

2.1 Preparatory Work

2.1.1 Research and Understanding

In order to fully implement the POPL paper introducing dynamic partial-order reduction [13], I first needed to find and read academic papers covering the relevant background theory, and then read and understand the paper itself. This exploration of a new area of computer science was the main preparation that was necessary before I could begin implementing my project. The remaining sections of this chapter explain the core of this material that is essential to understand my project.

2.1.2 Requirements Analysis

Having understood the necessary theory, I refined the goals of my project to make them as concrete as possible. I decided on exactly what the model-checking algorithms would take as their inputs and give as their outputs, and explicitly set out the need to implement a programming language to express concurrent programs for model checking. I also wrote a detailed schedule for the development process.

2.1.3 Using OCaml

I implemented the project in OCaml. However, having only written very small programs in SML previously, I needed to understand both how OCaml differs from SML and how large programs should be structured using the OCaml module system. I

achieved these by making use of online resources and experimenting with small test programs.

2.2 Background Definitions

After reading this section, you should understand the definitions of the mathematical objects and structures manipulated by the algorithms in my project. These are adapted from the POPL paper [13].

2.2.1 Processes and States

We consider programs consisting of a finite set, $\mathcal{P}$, of concurrent threads or processes[1]. Each thread has its own local state, $s \in \mathcal{S}$, and there is some shared, globally-accessible state, $g \in \mathcal{G}$. The overall state of the system at any instant is therefore a member of the set $State = (\mathcal{P} \to \mathcal{S}) \times \mathcal{G}$. Local states s can be thought of as comprising the instructions and data for each process, and shared states g as a shared memory which the processes use for communication.

Each process executes a sequence of operations, each of which can operate on the thread's local state s or the shared state g. If an operation accesses g, it is said to be *visible*, else it is said to be *invisible*.

2.2.2 Transitions

A *transition* moves the system from one state to another, by performing a finite sequence of invisible operations of one process p, followed by a visible operation of the same process. This definition ignores the irrelevant interleavings of invisible operations, so that non-determinism in our model of concurrency arises only from the asynchronous interleavings of atomic visible operations. If process p has local state s, then its next transition $t_{p,s}$ is defined to be a partial function taking the current global state g and giving (s', g'), the next local state for p and the next shared state for the system. Let $\mathcal{T}$ denote the set of all transitions,

$$\mathcal{T} = \mathcal{G} \rightharpoonup (\mathcal{S} \times \mathcal{G}).$$

A transition $t_{p,s}$ is *enabled* in a state (l, g) if $l(p) = s$ and $t_{p,s}(g)$ is defined. If $t_{p,s}$ is enabled in (l, g) but not (l, g'), then thread p is *blocked* in state (l, g'). If $t_{p,s}$ is enabled in $\sigma = (l, g)$ and $t_{p,s}(g) = (s', g')$, then the execution of $t_{p,s}$ from σ results in the unique successor state $\sigma' = (l', g')$, where

$$l'(q) = \begin{cases} s' & \text{if } p = q, \\ l(q) & \text{if } p \neq q. \end{cases}$$

In this case we write $\sigma \xrightarrow{t_{p,s}} \sigma'$. We write $\longrightarrow^*$ to denote the transitive reflexive closure of $\longrightarrow$.

[1] I use "thread" and "process" interchangeably throughout.

In any state $\sigma = (l, g)$, $next(\sigma, p) = t_{p,l(p)}$ denotes the unique next transition to be executed by process p, and

$$enabled(\sigma) = \{t_{p,s} \in \mathcal{T} \mid t_{p,s} = next(\sigma, p) \wedge t_{p,s} \text{ is enabled in } \sigma\}$$

denotes the set of enabled transitions that can be executed from σ. A state σ is a *stopped state* if $enabled(\sigma) = \emptyset$. We write $proc(t_{p,s})$ for p, the process of the given transition.

2.2.3 Systems

The behaviour of an entire concurrent program is represented by a *transition system*, specified by a tuple $(State, \Delta, \sigma_0)$, where $\sigma_0 \in State$ is the initial state of the system and $\Delta \in State \to \mathbb{P}(State)$ is the *transition relation* defined by

$$\sigma' \in \Delta(\sigma) \iff \exists t \in \mathcal{T}.\, \sigma \xrightarrow{t} \sigma'.$$

2.3 Model Checking

After reading this section, you should understand exactly what I mean by a model-checking algorithm in the context of my project, and how a simple model-checking algorithm works.

2.3.1 Definition of Model Checking

As far as my project is concerned, model checking is the process of deciding whether a given transition system is *error-free* and *deadlock-free*. In other words, instead of being specified as a temporal logic formula, the specification of each program is fixed to these two properties, because the algorithms we consider sacrifice richness of specification for efficiency of verification.

To specify what it means for a system to be error-free, we extend each transition system with a relation $err : \mathcal{S} \to \mathbb{B}$, which decides whether any thread has encountered an error in some given state—for example, an assertion failing to hold. We then say that a transition system $(State, \Delta, \sigma_0, err)$ is *error-free* if $err(p)$ holds for no thread p in any reachable state. Formally,

$$\forall\, (l, g) \in State.\ \sigma_0 \longrightarrow^* (l, g) \implies \forall p \in \mathcal{P}.\, \neg\, err(l(p)).$$

We say that a transition system is *deadlock-free* if there are no blocked threads in any reachable stopped state. Formally,

$$\forall\, (l, g) \in State.\ (\sigma_0 \longrightarrow^* (l, g)) \wedge (enabled(l, g) = \emptyset) \implies \forall g' \in \mathcal{G}.\, enabled(l, g') = \emptyset.$$

2.3.2 Simple Model Checking

As the definitions of both error-free and deadlock-free can be expressed in the form

$$\forall \sigma \in State.\ \sigma_0 \longrightarrow^* \sigma \implies \Phi(\sigma)$$

for some predicate Φ, the simplest strategy is to perform an exhaustive search of the reachable state space, checking whether $\Phi_{\text{error}}(\sigma) \wedge \Phi_{\text{deadlock}}(\sigma)$ holds at each state σ encountered. The predicate Φ_{error} is that no thread has encountered an error, and Φ_{deadlock} is that if the state is a stopped state then no thread is blocked.

The exhaustive search is usually implemented as a stateless depth-first search, and therefore can only be applied to acyclic state spaces; the presence of a cycle would cause the non-termination of the search.

2.4 Partial-Order Reduction

Although a naïve exhaustive search is correct, its performance suffers due to the state explosion problem. After reading this section, you should understand the persistent-set and sleep-set partial-order reduction techniques, which aim to address this problem, and which are necessary to understand the DPOR algorithm and other algorithms that I implemented.

2.4.1 Independence

Two transitions are independent if their effects are commutative, and if executing one can never disable the other (see Figure 2.1). More formally, suppose that $\mathcal{T}$ is the set of transitions for some transition system, and that $I \subseteq \mathcal{T} \times \mathcal{T}$ is a reflexive and symmetric relation. We say that I is a valid *independence relation* whenever the following two conditions hold for all $(t_1, t_2) \in I$:

1. for all states $\sigma \in \mathit{State}$, if t_1 is enabled in σ and $\sigma \xrightarrow{t_1} \sigma'$, then t_2 is enabled in σ if and only if t_2 is enabled in σ'; and
2. for all states $\sigma \in \mathit{State}$, if both t_1 and t_2 are enabled in σ then there are states σ_1, σ_2 and σ' such that $\sigma \xrightarrow{t_1} \sigma_1 \xrightarrow{t_2} \sigma'$ and $\sigma \xrightarrow{t_2} \sigma_2 \xrightarrow{t_1} \sigma'$.

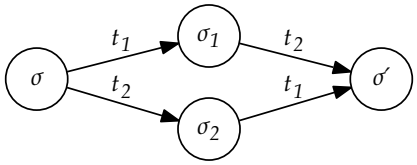
Two transitions t_1 and t_2 are said to be *independent* if there is a valid independence relation I such that $(t_1, t_2) \in I$, else they are said to be *dependent*. [13]

2.4.2 Persistent Sets

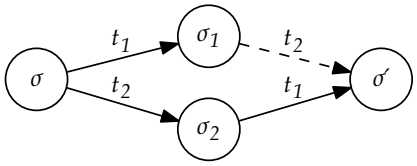
Partial-order reduction algorithms typically perform a depth-first search of the reachable state space, but at each state σ they explore only a subset $T \subseteq \mathit{enabled}(\sigma)$ of the possible transitions. The use of *persistent sets*, by Godefroid and Pirottin [16], is such a technique.

Definition

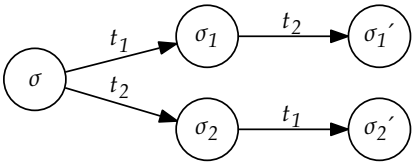
Informally, a set $T \subseteq \mathit{enabled}(\sigma)$ is persistent if every transition not in T reachable from σ by exploring only transitions not in T does not interact with the transitions in T.



(a) Transitions t_1 and t_2 are independent if they commute and cannot disable one another.



(b) If t_1 disables t_2 then t_1 and t_2 are not independent.



(c) If t_1 and t_2 do not commute then they are not independent.

Figure 2.1: Illustrations of transition independence.

Formally, a set $T \subseteq \mathit{enabled}(\sigma_1)$ is *persistent* in σ_1 if for all sequences of transitions

$$\sigma_1 \xrightarrow{t_1} \sigma_2 \xrightarrow{t_2} \ldots \xrightarrow{t_{n-1}} \sigma_n$$

such that $\forall i.\ t_i \notin T$, each t_i is independent with every transition in T.

An Example

In the state space shown in Figure 2.2, $T = \{t_1\}$ is a persistent set in the initial state, as every transition reachable following only transitions not in T is independent with t_1. Likewise, $T = \{t_2, t_3\}$ is a persistent set in the initial state, as the only transition reachable from the initial state without executing any transitions in T is t_1, and t_1 is independent with both t_2 and t_3. However, $T = \{t_2\}$ is not a persistent set in the initial state, as t_3 is reachable without executing t_2, but t_2 and t_3 are not independent. The set of all enabled transitions, $T = \mathit{enabled}(\sigma)$, is always a persistent set since there are no transitions that are reachable without executing one in T.

Soundness

It can be shown that if a stopped state is reachable by an exhaustive search of the state space, then it is reachable by a search exploring only the transitions in a persistent set at each state [15, Theorem 4.3]. Likewise, all reachable local states of each process are reachable in a persistent-set search [15, Theorem 6.14]. Therefore, a persistent-set search is sufficient for our definition of model checking (cf. Section 2.3.1).

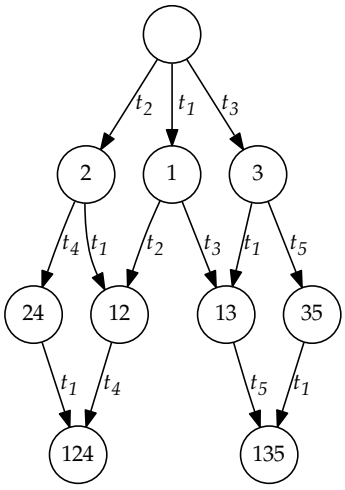


Figure 2.2: Diagram of an example state space, for the illustration of persistent sets. Transition t_1 is independent with the others, which are all dependent with one another.

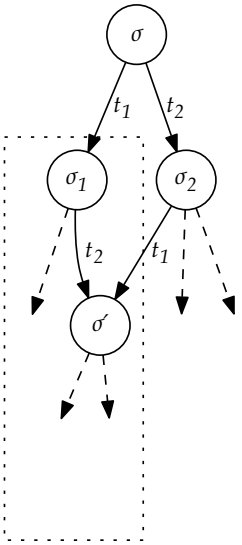


Figure 2.3: Diagram illustrating the idea behind sleep sets.

2.4.3 Sleep Sets

Another technique that addresses the state explosion problem is that of *sleep sets*, again by Godefroid [14]. Suppose that we are at state σ when performing a search of the state space (see Figure 2.3). Suppose we first explore t_1, reaching σ_1, and then t_2, reaching σ_2. If t_1 and t_2 are independent, then there is a state σ' reachable by both $\sigma \xrightarrow{t_1} \sigma_1 \xrightarrow{t_2} \sigma'$ and $\sigma \xrightarrow{t_2} \sigma_2 \xrightarrow{t_1} \sigma'$. Clearly, any stopped state reachable from σ' is reachable from σ_1. By assumption, when the search is at σ_2, we have already explored all the stopped states reachable from σ_1 (the dotted area in Figure 2.3), in particular those reachable from σ'. Therefore, there is no need to explore t_1 from σ_2.

In fact, there is no need to consider executing t_1 in our recursive exploration from σ_2 until we execute some transition t' which is *not* independent with t_1, since only after the execution of such a t' may exploring t_1 then lead to an unexplored area of the state space.

A sleep-set algorithm is one that maintains a set of transitions which, by the above argument, need not be explored from the current state.

2.5 Dynamic Partial-Order Reduction

After reading this section, you should understand roughly how the dynamic partial-order reduction algorithm works. The information in this section is adapted from the POPL paper [13].

2.5.1 Background Definitions

Transition Sequences

Instead of keeping track of just the current state, the DPOR algorithm keeps track of a *transition sequence* $\pi \in \mathcal{T}^*$. This transition sequence is implicitly executed from the initial state of the transition system, σ_0, so that in practice, $\pi = t_0, t_1, \ldots, t_{n-1}$ determines a sequence of states $\sigma_0, \sigma_1, \ldots, \sigma_n$ such that

$$\sigma_0 \xrightarrow{t_0} \sigma_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{n-1}} \sigma_n.$$

Given a transition sequence $\pi = t_0, t_1, \ldots, t_{n-1}$, we use the following notation:

π_i — the transition t_i;

$\pi.t$ — the transition sequence π extended with transition t;

$dom(\pi)$ — the set $\{i \in \mathbb{N} \mid 0 \leq i < n\}$;

$|\pi|$ — the length of the transition sequence, n;

$pre(\pi, i)$ — the state σ_i immediately preceding π_i; and

$last(\pi)$ — the final state, σ_n.

The Happens-Before Relations

Suppose that two adjacent transitions, π_i and π_{i+1}, are swapped in a transition sequence π to give a new transition sequence, and that π_i and π_{i+1} are independent. Because the order of execution of a pair of independent transitions is irrelevant in the sense that it results in the same state, the result of executing the two transition sequences must be the same (see Figure 2.4). It follows that any transition sequence obtained by making a finite number of swaps of adjacent independent transitions is equivalent in the same sense. Such equivalence classes are known as Mazurkiewicz traces [20], and are the basis of DPOR, since it is sufficient to explore only one member of each equivalence class.

It is useful to consider a "happens-before" relation that identifies pairs of transitions which cannot be swapped with one another to form an equivalent sequence. In

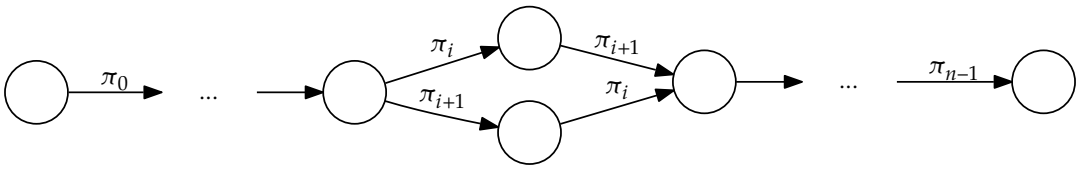


Figure 2.4: Illustration of equivalent transition sequences.

particular, the *happens-before* relation $\longrightarrow_\pi$ for a transition sequence π is the smallest relation on $dom(\pi)$ such that

1. if $i \leq j$ and π_i is dependent with π_j then $i \longrightarrow_\pi j$, and
2. $\longrightarrow_\pi$ is transitively closed.

By construction, the happens-before relation is a partial-order relation on the transitions in π, and the linearisations of $\longrightarrow_\pi$ are precisely the transition sequences equivalent to π.

The DPOR algorithm makes use of a variant of the happens-before relation, which guarantees that a process must execute its next transition by the end of the current transition sequence. Informally, if $i \hookrightarrow_\pi p$ then for some $j \geq i$, the process of π_j is p, both in π and any equivalent transition sequence. Formally, if $i \in dom(\pi)$ and $p \in \mathcal{P}$ then $i \hookrightarrow_\pi p$ if either

1. $proc(\pi_i) = p$, or
2. $\exists j \in dom(\pi).\ i < j \wedge proc(\pi_j) = p \wedge i \longrightarrow_\pi j$.

2.5.2 The Dynamic Partial-Order Reduction Algorithm

The DPOR algorithm uses information gathered dynamically during its exploration of a state space to explore only transitions in a persistent set from each state encountered on the search. The idea is that the extra information allows smaller persistent sets to be computed than would otherwise be possible, thereby making the search more efficient.

In particular, at each state σ, the next transition to execute, t, is chosen arbitrarily, and it is initially assumed that t will be the only transition to be explored from that state. However, $\{t\}$ is a persistent set only if t is independent of every transition reachable by exploring any transitions from σ other than t. Clearly, this is not true in general, so at each state σ' encountered beyond σ, if there is a transition $t' \in enabled(\sigma')$ such that t' is dependent with t, and t' may appear before t in an exploration from σ, then a backtracking point is added at σ to ensure that such an exploration is made. The addition of such backtracking points ensures that the set of transitions explored from each state is a persistent set.

The pseudocode for and a more detailed explanation of the DPOR algorithm are provided in Appendix A.1.

Chapter 3

Implementation

After reading this chapter, you should understand:

- the software development approach I used;
- the high-level structure and organisation of my project;
- the design and implementation of the programming language used as the object language for the project;
- the details of my implementation of the DPOR algorithm; and
- how I implemented some other model-checking algorithms.

3.1 Software Engineering

After reading this section, you should understand how I approached the development of my project, and what its high-level structure is.

3.1.1 Development Strategy

I used an incremental approach to the development of my project. This involved breaking down the project into several subgoals, each of which I implemented and tested in turn, before moving on to the next. The subgoals were to:

- design a concurrent programming language and create an interpreter for it;
- create a parser for that language;
- implement a simple model-checking algorithm;
- implement DPOR, assuming a language without locks for simplicity; and
- add locks to the DPOR implementation.

Each of the several extensions I completed was also developed as a new incremental subgoal.

The benefits of this approach include the frequent testing of each new component, the manageable size of each subgoal, and the creation of working software from the beginning.

3.1.2 Structure of the Project

The project was implemented in OCaml, which offers a few key advantages:

- a clean, functional style;
- a strong, safe type system, which prevents a large class of runtime errors;
- parametric polymorphism, a module system, and other higher-order programming features, facilitating abstraction and generalisation; and
- established libraries, providing efficient implementations of basic data structures and algorithms.

OCaml modules provide abstraction by encapsulating related definitions and hiding implementation details. *Signatures* are software interfaces providing declarations, and *structures* are implementations which provide definitions. A structure *matches* a signature if it has a definition for each declaration in the signature. *Functors* act as functions between modules.

In my project, there are five main signatures:

1. *Expression* contains information about the structure of expressions in the programming language of interest;
2. *Store* implements a mapping from locations to expressions for some implementation of *Expression*;
3. *Thread* contains the information about the reduction of expressions for one thread;
4. *Program* contains declarations combining threads to express a full concurrent program; and
5. *Checker* declares a function which performs model checking on a given concurrent program.

Since model checking is independent of any particular programming language, I used functors to minimise the work needed to use my project for a new language. From two structures matching *Expression* and *Thread*, functors automatically generate implementations of *Store*, *Program* and *Checker*, because I wrote each model-checking algorithm as a functor. A working implementation of each model-checking algorithm in my project can therefore be obtained for a new language just by specifying its syntax and semantics. This process is shown explicitly in Figure 3.2 for a hypothetical new language, NL.

3.2 Development of the Project Language

After reading this section, you should understand how I designed and implemented a programming language for the expression of programs to model check.

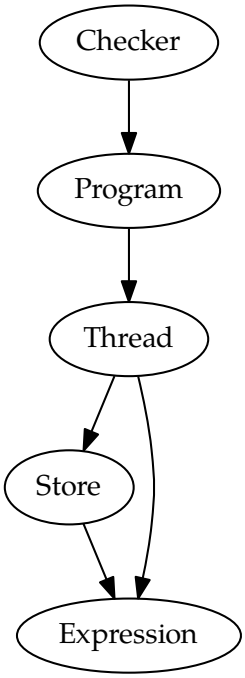


Figure 3.1: Diagram showing which signatures declare which other signatures as nested sub-structures.

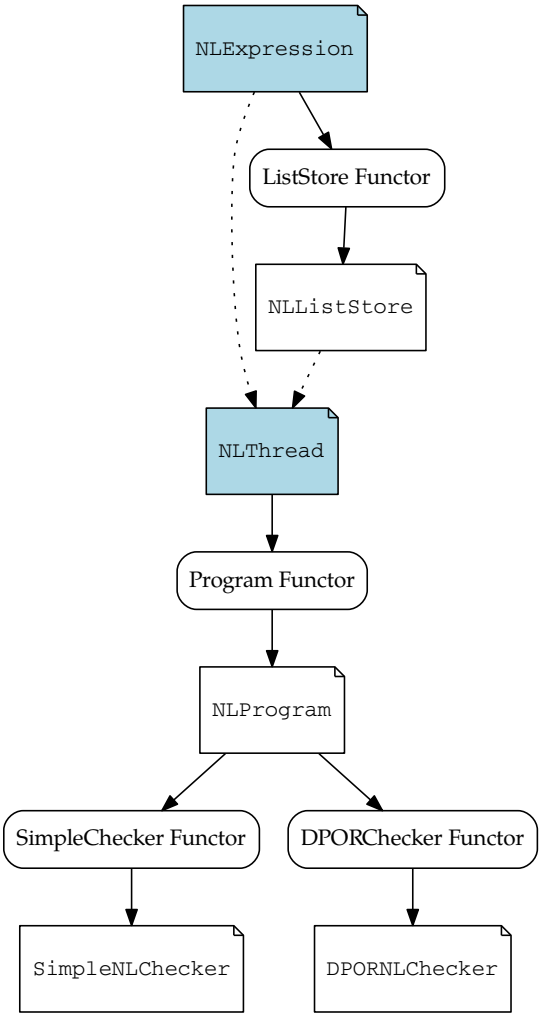


Figure 3.2: Diagram showing generation of structures using functors. The shaded files must be provided, the functors are part of my project, and the remaining files are generated using the functors.

3.2.1 Language Design

Although any concurrent thread-based programming language could be used to express transition systems for model checking, I chose to design and implement a simple ML-like language, PL (Project Language), to avoid dealing with the full complexity of an industrial language. When designing PL, my aim was to create a high-level language that had enough features to easily write practical concurrent programs, but was not unnecessarily complex.

PL is a Turing-complete functional language which operates on integers, Booleans and first-class functions. Thread-local reference cells provide some procedural functionality, and the usual structures manage flow control. There is an error construct, which allows easy syntactic checking of whether a thread has encountered an error. The shared state is a collection of variables that are shared between all threads, some of which can be locks, used to enforce mutual exclusion. There are four atomic operations that interact with shared variables:

1. a read operation returns the current value of a shared variable;
2. a compare-and-swap operation compares the value of a shared variable to a given value, and if they are the same, updates the shared variable to another given value and returns “true”, else it returns “false”;
3. a lock operation takes the given lock if available, else blocks until it becomes available; and
4. an unlock operation makes the given lock available.

Since the locking operation is the only one that can block, it is a trivial syntactic check to decide whether a given thread is blocked or not.

I did not write a type system for PL because I thought that the time to implement one would be greater than the time saved as a result. In hindsight, I think this was correct.

3.2.2 Parser

I decided that investing time in creating a parser for PL would pay itself off over the course of the project. A parser allowed me to write, for example,

```
let y = ref 1 in cas(Gx, 7 * !y - 3, 0),
```

instead of having to type the OCaml meta-language abstract syntax tree representing the structure of the PL expression,

```
Let (Ref (Integer 1), Cas (Global "x", Op (Op (Integer 7,
            Mult, Deref (Var 0)), Minus, Integer 3), Integer 0)).
```

Writing such expressions is not only extremely time-consuming but also error-prone. For the language to be of any practical use, a parser was a necessity.

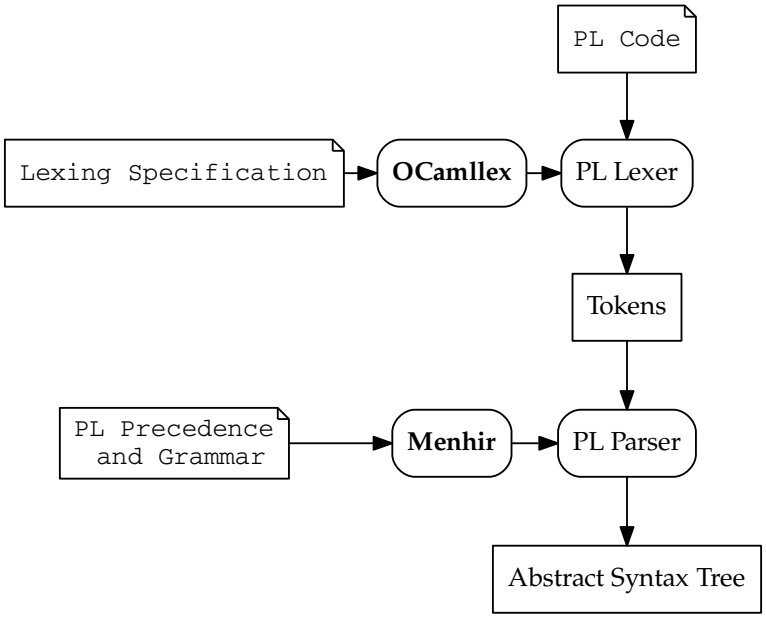


Figure 3.3: Diagram showing the parsing of a PL program.

To create a parser for PL, two program generators were used: OCamllex and Menhir. When given a file detailing a mapping from strings of characters to tokens, OCamllex produces a lexical analyser, which converts a sequence of characters into a sequence of tokens. When given a grammar and a set of precedence rules, Menhir produces a parser, which converts a sequence of tokens into an abstract syntax tree of the grammar. These were used in conjunction, as shown in Figure 3.3, to produce a full parser for PL.

3.2.3 Implementation

The Expression Module

Two datatypes are used to represent PL expressions: a "raw" expression, which uses strings to represent variables (this is the datatype that the parser produces), and a datatype which uses De Bruijn indices [4] to represent variables, allowing easier manipulation of expressions.

The remainder of the *Expression* module consists mainly of auxiliary functions which are necessary for the implementation of later modules, and either return a simple piece of information or perform a simple manipulation.

The Store Module

A functor, ListStore, is used. Given an *Expression* implementation, it returns a *Store* implementation, which represents stores using lists of (location, expression) pairs. Map-

pings are adding by appending a new pair to the front of the list, causing any old mapping for that location to become stale. A function is provided that walks through the list removing stale pairs, used to prevent the list from growing too long, or in situations when there must be no stale values in the list.

More efficient store implementations are possible—for example, one based on a hash table rather than a list—but I valued simplicity over efficiency in this case, to increase my confidence in its correctness.

The Thread Module

The *Thread* module consists primarily of the `next_step` and `next_transition` functions, which define the semantics of PL. Given an expression, `next_step` uses pattern matching to determine what the next atomic operation (if any) is for the reduction of that expression. Care must be taken to ensure that each step given here really is atomic, else race conditions could exist in real executions of programs that would not be found by model checking. This is not a concern for PL, since its only implementation is the one used for the model checker, but would be if an industrial language was used.

Given the definition of a transition (cf. Section 2.2.2), the `next_transition` function repeatedly calls the `next_step` function until some step accesses the shared store, at which point the resulting expression and updates to the stores are returned.

The Program Module

The *Program* module is a small functor which, when given a *Thread* implementation, provides the few additional definitions and auxiliary functions necessary to fully represent the state of transition systems (cf. Section 2.2). In particular, a state $\sigma \in State$ is represented by an instance of the `state` type:

```
type state = (ThrImp.ExpImp.expr * ThrImp.StoreImp.store) array *
    ThrImp.StoreImp.store;;
```

Testing

To ensure that the implementation of the PL language was correct, I wrote around twenty unit tests for the `next_transition` and `next_step` functions. These had good code coverage, and caught most of the bugs I found in the implementation of PL.

3.3 Dynamic Partial-Order Reduction

An overview of the DPOR algorithm was given in Section 2.5, with a more detailed explanation given in Appendix A.1. Several aspects of the implementation of DPOR are not obvious; after reading this section you should understand how I implemented all aspects of DPOR.

```
1:  procedure EXPLORE(π)
2:      let σ = last(π) in
3:      for all p ∈ P do
4:          UPDATEBACKTRACKSETS(π, next(σ, p))
5:      if enabled(σ) ≠ ∅ then
6:          let t = any t ∈ enabled(σ) in
7:          let backtrack(σ) = {t} in
8:          let done(σ) = ∅ in
9:          while done(σ) ≠ backtrack(σ) do
10:             let t = any t ∈ (backtrack(σ) \ done(σ)) in
11:             add t to done(σ);
12:             EXPLORE(π.t)
13:
14: procedure UPDATEBACKTRACKSETS(π, t_{p,s})
15:     let D = {i ∈ dom(π) | π_i is dependent with t_{p,s} and i ↛_π p} in
16:     if D ≠ ∅ then
17:         let σ_d = pre(π, max(D)) in
18:         if next(σ_d, p) ∈ enabled(σ_d) then add next(σ_d, p) to backtrack(σ_d)
19:         else add all of enabled(σ_d) to backtrack(σ_d)
20:
21: Initially: EXPLORE(∅)
```

Figure 3.4: The DPOR algorithm.

3.3.1 Implementation

This section explains how I translated the pseudocode in Figure 3.4 into an executable OCaml function, `check`, which is given in Appendix A.2 for reference.

Line 2 When implementing the simple model checker (cf. Section 3.5.1), I used the `fold_left` function from the library List module to repeatedly use the `apply_transition` from the *Program* module to apply the transitions in the transition sequence to the initial state. However, in this case I wrote a more general function, `pre`, which applies only the first n transitions in the sequence, thereby returning the state $pre(\pi, n)$ if $n < |\pi|$ or the state $last(\pi)$ if $n = |\pi|$, which is what we want in this case.

Lines 3–4 The `next_transition` function, provided in the *Thread* module, may return no transition at all (for instance in the case that a thread has terminated its computation), so the call to UPDATEBACKTRACKSETS is only made if `next_transition` does return a transition.

Lines 15–17 In my project, each transition t accesses only one shared object, denoted $\alpha(t)$, and it is assumed that two transitions are dependent if and only if they access the same shared object (or belong to the same process). It follows that the

most recent transition π_i such that π_i is dependent with $t_{p,s}$ and $i \not\rightarrow_\pi p$, if any, is precisely the last transition to access $\alpha(t_{p,s})$.

Therefore, as well as the transition sequence, π, `check` takes as an argument a function L, such that $L(o)$ is the index of the last transition that accessed shared object o, so that the index of the necessary backtracking point, if any, is $L(\alpha(t_{p,s}))$. However, if this transition happens-before t then no backtracking point at all is necessary, since there cannot be another transition that is dependent with t but does not happen-before t. To decide the happens-before relation, clock vectors are used, as described in Section 3.3.2.

Because $\sigma_d = pre(\pi, L(\alpha(t_{p,s})))$, the `pre` function mentioned above can be used to obtain σ_d.

Lines 18–19 The `next_transition` function returns a Boolean with each transition, indicating whether it is enabled in the given state, so that deciding whether $next(\sigma_d, p)$ is enabled is trivial. This also means it is easy to compute the enabled set of a state, by iterating through the processes and looking at whether the next transition of each is enabled.

Lines 5–6 If, while iterating through the processes in lines 3 and 4, any transition is found to be enabled, then a reference is set to it. If at line 5, that reference has been set at all, then it is chosen as the $t \in enabled(\sigma)$ in line 6, otherwise lines 6 to 12 are not executed.

Line 7 The *backtrack* sets for each state in the sequence of states implied by the transition sequence π must be stored in a data structure that allows them to be accessed and updated from all later recursive calls, so that they can add backtracking points. I decided to use a variable-length array for this, as arrays are mutable and have efficient access, but the maximum recursion depth (and so the longest fixed-length array necessary) cannot be predicted.

I therefore implemented a module which uses a fixed-length array to implement a variable-length array by copying the elements to a fresh array of double the previous size if the underlying array proves too short. Although this operation is expensive, the amortised cost of adding a new element to the array is constant. In this case, each element of the array is a *backtrack* set of a state, implemented as a list of processes whose next transitions are to be explored.

Lines 8–11 Instead of using a procedural while-loop, I implemented the iteration through the backtracking points using a tail-recursive function, taking a list of processes representing the *done* set as its argument. An auxiliary function returned a member p of the *backtrack* set not present in the *done* set, from which the transition t to explore is simply $next(\sigma, p)$. The recursive call to the tail-recursive function then has the old *done* set extended with p as its argument. If the auxiliary function can find no such p, then the tail-recursive function (and therefore the call to `check`) returns.

Line 12 Once the new arguments have been computed, a recursive call to `check` is made. Other than the clock vectors (see Section 3.3.2), the arguments that have to be computed are the new transition sequence, $\pi' = \pi.t$, and the new function, L', tracking which transitions have accessed each object:

$$L'(o) = \begin{cases} |\pi| & \text{if } o = \alpha(t); \\ L(o) & \text{otherwise.} \end{cases}$$

Line 13 Once all backtracking points have been explored, the result is returned. Appendix B.1 explains how errors and deadlocks are detected in the simple model checker, and the computation is very similar for the DPOR algorithm. The iteration through all threads on line 3 is used to also check whether any thread has encountered an error, is enabled, or is blocked. This information is combined with the results from the recursive calls to compute whether the state space from the current state onwards is error-free and deadlock-free.

3.3.2 Deciding The Happens-Before Relation

To decide the happens-before relation, $\hookrightarrow_\pi$, I used clock vectors. Although Flanagan and Godefroid outline how clock vectors can be used [13], they do not give the following detail in the reasoning.

Using Clock Vectors

As explained above, although the function L gives the index of the last transition that accessed a given shared object, which is the index of the potential backtracking point, we only need to add it if it did not happen-before the current transition. That is, if $L(\alpha(t_{p,s})) \not\hookrightarrow_\pi p$.

If, for each process q, we know the index k of the last transition of that process such that $k \hookrightarrow_\pi p$, then given any other transition π_i of process q, we know that $i \hookrightarrow_\pi p$ if and only if $i \leq k$. A clock vector c is a function from processes q to such indices k in the transition sequence, and we maintain one clock vector, c_p, per process p. To decide whether $i \hookrightarrow_\pi p$ for any i and p is simply to decide whether $i \leq c_p(q)$, where q is the process of transition π_i. This takes constant time.

Updating Process Clock Vectors

By passing correct clock vectors as arguments to `check`, then, it is straightforward to decide whether we need to add each potential backtracking point. Assuming that the correct value is passed for the initial call, (namely each $c_p(q) < 0$) we need to ensure that the clock vectors are correctly updated when a recursive call exploring a new transition, $t_{p,s}$, is made.

The only clock vector that needs to be updated is c_p, since $i \hookrightarrow_\pi p'$ if and only if $i \hookrightarrow_{\pi'} p'$, where $\pi' = \pi.t_{p,s}$ and $p' \neq p$. Let us denote the updated clock vector by

c'_p, and consider $c'_p(q)$, which must be at least $c_p(q)$, because if process p must happen-before the end of π then it must happen-before the end of π'. However, if there is a more recent transition of process q, say π_i that accesses the same shared object as the new transition $t_{p,s}$, then $c'_p(q)$ must be updated to the index of that transition, because π_i must happen-before $t_{p,s}$.

To perform this update, we need to keep track of the index k of the most recent transition of each process p that accesses each shared object o. We therefore also maintain a clock vector c_o for each shared object o, and set

$$c'_p(q) = max(c_p(q), c_{\alpha(t_{p,s})}(q)).$$

The only special case is when $q = p$, in which case $c'_p(p) = |\pi|$, since the last transition is now a transition of process p, and $|\pi|$ is the index of the last transition in π'. The full update for the clock vector is therefore

$$c'_p(q) = \begin{cases} |\pi| & \text{if } q = p; \\ max(c_p(q), c_{\alpha(t_{p,s})}(q)) & \text{if } q \neq p. \end{cases}$$

Updating Object Clock Vectors

We have seen how to use object clock vectors to update the process clock vectors, but the object clock vectors themselves must also be updated whenever a new transition is explored. In particular, if the clock vector for the object accessed by the transition must be updated so that the thread of the transition maps to that transition:

$$c'_{\alpha(t_{p,s})}(p) = |\pi|.$$

However, we can do better, by letting $c_o(q)$ be the index of the last transition of process q such that object o is guaranteed to be accessed between $c_o(q)$ and the last transition of the sequence (inclusive), whether or not $\pi_{c_o(q)}$ itself accesses o. This is sufficient for updating the process clock vectors as above, and also allows for higher indices to be stored than just the last transition of a process that accesses the object.

For example, suppose that, for some process $q \neq p$, $c_{\alpha(t_{p,s})}(q) < c_p(q)$. That is, more recently than the last transition of q that accessed the object $\alpha(t_{p,s})$, there was a transition of q that happens-before some transition of process p prior to the new transition $t_{p,s}$. Since any previous transition of process p must happen-before $t_{p,s}$, we know that the transition of index $c_p(q)$ must also happen-before $t_{p,s}$. Therefore, there is a transition (namely $t_{p,s}$) which accesses $\alpha(t_{p,s})$ and is guaranteed to happen after $\pi_{c_p(q)}$. So we can safely perform the following update:

$$c'_{\alpha(t_{p,s})}(q) = max(c_p(q), c_{\alpha(t_{p,s})}(q)),$$

meaning that the full update for the object clock vector is

$$c'_{\alpha(t_{p,s})}(q) = \begin{cases} |\pi| & \text{if } q = p; \\ max(c_p(q), c_{\alpha(t_{p,s})}(q)) & \text{if } q \neq p. \end{cases}$$

Noting that this is precisely the same clock vector as c'_p allows for a more efficient implementation of the updates.

3.4 Counter-Example Traces

One of the advantages of model checking is that, if an error is found, it is easy to give an example execution leading to that error—in our case, this is just the sequence of transitions leading to the error state. To give more insight into the reason for the error state being entered, the user could instead be given a *partial order* of the transitions which, whenever linearised into an execution, results in the error (or deadlock) state.

This partial order is simply the happens-before relation, as described in Section 2.5.1. As two transitions are dependent if and only if they belong to the same thread or if they access the same shared object, my algorithm executes the given transition sequence, using functions from processes and shared objects to positions in the sequence to track the last transition to each process and to access each shared object respectively. For each transition executed, two "happens-before" edges are added to a graph representing the partial order: one from the last transition belonging to the same process, and one from the last transition accessing the same shared object.

To show the partial order graph in an easily-interpretable way, I output a GraphViz file displaying the graph, with each node labelled with its process and the shared object it accesses. GraphViz is an open-source program which draws graphs. Figure 3.5 shows an example of such an automatically-generated graph.

3.5 Extensions For Evaluation

Having reached my goal of implementing the DPOR algorithm, I decided to implement some additional model-checking algorithms as extensions, to allow a comparative evaluation of their performances. After reading this section, you should understand how I implemented these other model-checking algorithms.

3.5.1 Simple Model Checker

Implementing and testing the simple model checker before DPOR (although not an extension) allowed me to sanity-check my PL implementation, practise with an easy model-checking algorithm first, and later verify the results of my DPOR implementation. It also provides a performance baseline for the evaluation of the other algorithms. As outlined in Section 2.3.2, it performs an exhaustive depth-first search of the state space, recursively deciding whether each state is error-free and deadlock-free. Appendix B.1 gives the pseudocode for this algorithm, and explains how it is decided whether an error or deadlock state has been reached.

3.5.2 Stateful Model Checking

If we keep track of which states we have already explored on a search, then when we re-encounter a state, we can cut short the search in the knowledge that we have already explored from that state, improving time efficiency at the cost of extra space.

Shared variables:

`n`, `w`, `x` and `y`, initially `0`

`t` and `z`, initially `false`

Thread 0:

```
t := z;
if t then x := y
```

Thread 1:

```
n := 1;
z := true
```

Thread 2:

```
y := 2;
if w = 1 & x = 2 then error
```

Thread 3:

```
w := n
```

Thread 4:

```
t := false
```

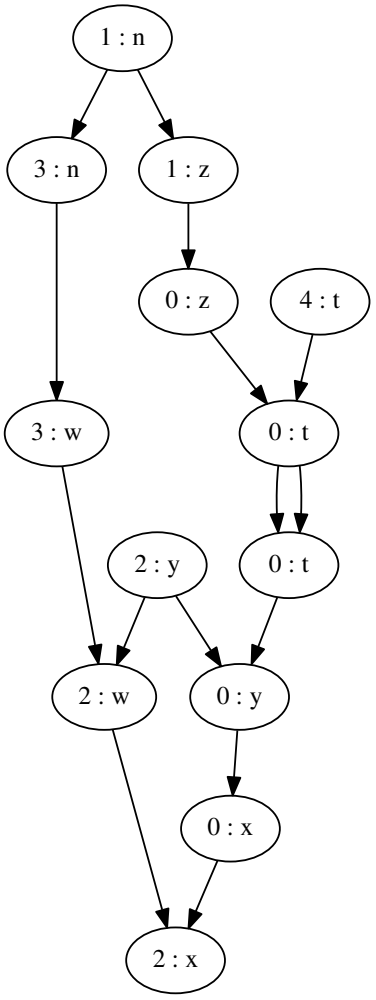


Figure 3.5: An example partial-order error trace. The assignments in the program are implemented in PL using compare-and-swap operations.

Simple Model Checker

Applying this technique to the simple model checking algorithm is straightforward: whenever a state is visited, add it to some data structure, then explore only those states which are not present in that data structure. As querying this structure and inserting a state to it happens at every state, we would like these operations to be efficient. The typical amortised costs of both these operations for a hash table are constant, so I used the OCaml library implementation of hash tables for this purpose.

Dynamic Partial-Order Reduction

Suppose that we applied this approach to the DPOR algorithm, and executed the resulting algorithm on the example shown in Figure 3.6. Consider the stage at which all the states in grey have been explored and so are present in the hash table, and the current transition sequence is the first transition of thread 1 followed by the first transition of thread 0. As this stage, the state labelled 2200 is encountered, which is already present in the hash table, so no exploration is made, and as no backtracking points have been added at either of the previous stages, the algorithm terminates.

However, this leaves part of the state space entirely unexplored (the dotted section in the diagram), meaning that the algorithm is unsound; any error or deadlock in a dotted state would go undetected. The reason for the unsoundness is the failure to introduce necessary backtracking points. If the search had not been cut short, then a backtracking point would have been added in the second state (1200), causing the dotted section to be explored.

Solving this by naïvely adding all possible backtracking points whenever the search is cut short turns out to give worse performance than plain DPOR. To use this technique, we need a method of conservatively deciding which backtracking points must be added for soundness, but not so conservatively that all practical benefits are lost.

Stateful Dynamic Partial-Order Reduction

One such method is to use a *visible operation dependency graph*, proposed by Yang et al. in an algorithm known as stateful dynamic partial-order reduction (SDPOR) [37]. The idea is that when we first explore a section of the state space, we make a note of the visible operations that may be executed, so that if we encounter that section again, we can cut short the search, and use our knowledge of the visible operations that may have been executed if the search was continued to add backtracking points as necessary.

More precisely, a visible operation dependency graph is a graph $G = \langle V, E \rangle$ whose nodes represent visible operations, and whose edges represent the ordering of those transitions; whenever a transition sequence of the form $\sigma_1 \xrightarrow{t} \sigma_2 \xrightarrow{t'} \sigma_3$ is encountered in a search of the state space, the edge (t, t') is added to the visible operation dependency graph G. Then, when a state σ is encountered which has already been explored on the search, sufficient backtracking points for soundness can be added by calling the UPDATEBACKTRACKSETS procedure on each transition in the set

$$\mathcal{U} = \{u \mid \exists t \in \mathit{enabled}(\sigma).\ u \text{ is reachable from node } t \text{ in graph } G\}.$$

Shared variables:	Thread 0:	Thread 1:
`m` and `n`, initially `1`	`m := 2;`	`n := 2;`
`x` and `y`, initially `0`	`x := n`	`y := m`

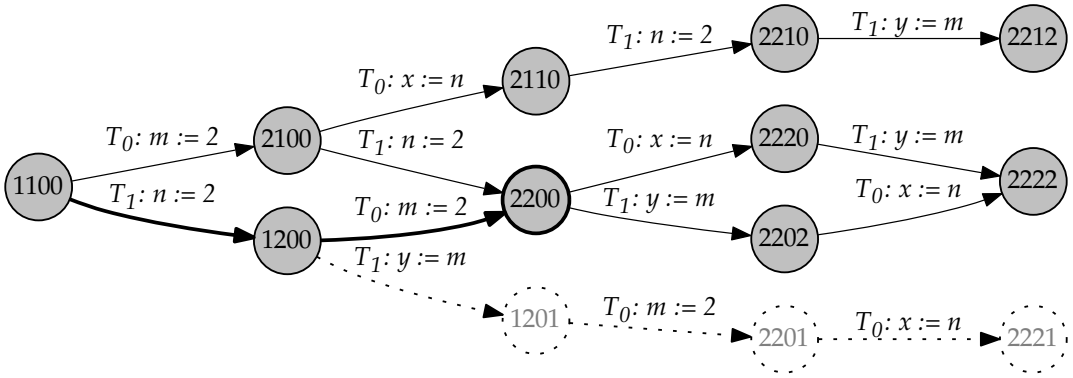


Figure 3.6: An example showing that naïvely cutting short the DPOR algorithm is unsound. Each state in the diagram shows the values of *mnxy*.

For its efficiency, SDPOR relies on the assumption that the sets $\mathcal{U}$ of reachable transitions are generally small in size.

The pseudocode for the SDPOR algorithm is shown in Appendix B.2. To implement the visible operation dependency graph G, I chose to use another hash table, mapping from transitions to lists of transitions. Calling UPDATEBACKTRACKSETS on each reachable node in G is implemented using a simple depth-first search.

3.5.3 Static Partial-Order Reduction

In order to evaluate the improvement in performance allowed by building persistent sets dynamically instead of statically, I decided to research and implement a static partial-order reduction (SPOR[1]) algorithm. However, I struggled to find one in the literature that I could understand and implement in the available time. I therefore designed my own rudimentary static persistent-set algorithm, which, while easy to understand and implement, usually returns unfortunately large persistent sets.

Before the algorithm begins its search, it compiles a table listing all the shared objects that each process may access, then, during the search, persistent sets T are constructed such that every process whose next transition is not in T never accesses any object that any process whose next transition is in T accesses. The pseudocode for this procedure is given in Appendix B.3. While it may seem that computing the objects the threads can access should be done at every state rather than just once, this is actually prohibitively expensive and offers only modest reductions in the sizes of the persistent sets.

[1]Not to be confused with SDPOR.

A limitation of my algorithm is that it assumes that each threads accesses only shared objects explicitly mentioned in its own local state so using shared objects to communicate references to other shared objects may result in unsound persistent sets being chosen.

3.5.4 Sleep Sets

To implement sleep sets, as explained in Section 2.4.3, I adapted my code for the simple model-checking algorithm to match the pseudocode given in Appendix B.4. As usual, I used a library hash table implementation, and used lists of process identifiers to represent sleep sets, since one integer takes up less memory than one representation of a transition, while providing sufficient information for this purpose.

Applying the sleep-set technique to the dynamic partial-order reduction algorithm involves keeping track of the sleep sets in the same way, and never adding backtracking points for processes that are in the sleep set of the relevant state. This is sound, although not as obviously as it might first seem [12, Theorem 1].

Chapter 4

Evaluation

After reading this chapter, you should understand:

- the extent to which I believe my implementations are correct, and the evidence motivating that belief;
- why being certain of the soundness of a model-checking system is not essential for it to be of practical use; and
- how the model-checking algorithms I implemented perform in relation to one another.

4.1 Soundness

Although the various model-checking algorithms have all been proven sound, bugs in my project could easily render my implementations of them incorrect. After reading this section, you should understand the extent to which I believe my implementations are correct and why, and how verification systems can be useful despite not being verified themselves.

4.1.1 Evidence for Soundness

Almost every program contains bugs—industry average is about 1 to 25 errors per 1,000 lines of delivered code [21]. My project consists of at least 3,250 lines of code, and there is no reason to think that I would produce fewer errors than the average professional software developer, so it is likely it contains several defects. However, I believe that all remaining bugs in my project probably relate to functionality not necessary for soundness. For instance, I would not be surprised if there were a bug in my implementation of PL, whereas I would not expect my implementation of DPOR to return an incorrect result. There are two main reasons for this belief: my review and understanding of the code, and my testing process.

Review of Source Code

OCaml is high-level enough that side-by-side comparison of my code and the corresponding pseudocode reveals the mapping of one onto the other. This makes it relatively easy to perform thought experiments to try to find differences in semantics between the two. During development, I spent time doing this, successfully discovering some bugs; at the end of the process, I am more confident in the correctness of my implementations as a result—especially the simple model checker, due to its simplicity.

Tests

As part of the development process, I wrote 45 varied test cases, each consisting of a PL program and the expected (*error-free*, *deadlock-free*) result of model checking that transition system. Running a model-checking algorithm on the program should give the expected result. If not, then either I expected the wrong result, or the implementation was incorrect. Because of my confidence in its correctness, I used the simple model checker to verify my expectations, so that any unexpected result implied the existence of a bug.

Although the tests were primarily designed to help me find bugs during development, they have the secondary benefit of increasing the confidence in my implementations—only obscure bugs could elude my tests, assuming that the code coverage is as good as I think.

4.1.2 Model Checking In Practice

Verification of my implementations would take months of effort, and is significantly out of the scope of my project. Although it may initially seem that a model-checking system without such a guarantee of its soundness is useless, this is not the case.

It can be argued [10] that proof of a statement is best understood as a social process by which a consensus arises in its truth, and that formal proofs are a relatively poor catalyst to this process, since they are so long and difficult to read. There are other ways that the confidence in a program's correctness can be increased, including conventional testing, successful long-term use, and performing cross-checking with other (ideally independently developed) implementations [1, 30].

In practice, factors such as performance and usability are at least as important as soundness. A system that catches 90% of bugs can still be a useful tool, especially if it catches errors which are difficult to expose using conventional testing, such as those caused by race conditions. Conversely, if a system is guaranteed to be sound, but takes an impractically long time to check any realistic program, or if it requires weeks of training before use, then it will not be used in practice. Similarly, a system that returns only a single Boolean result is less useful than one like my project, which points the developer in the direction of the bug if one is found.

Perhaps the clearest indication of the value of unverified model-checking systems is their widespread industrial use and resulting successes. [3, 8, 17, 18, 19, 23, 24] It is also noteworthy that, as of 2006, there were only two verified model-checking systems

despite model checking having been in use for over two decades, neither of which was widely used [1].

4.2 Performance

After reading this section, you should understand how effective the different model-checking algorithms are at addressing the state explosion problem, and how efficient my implementations of them are.

4.2.1 Methodology

Unfortunately, there is no concurrent program that is "typical" for verification by model checking. I have therefore had to pick some example programs for which to analyse the performance of the algorithms. For each of these examples, I varied some test parameter and recorded for each algorithm both the number of (not necessarily unique) transitions explored, which is a measure of how successful the algorithm is in addressing the state explosion problem, and also the execution time, which is what ultimately matters in practice.

The examples I chose (explained below) feature a lot of interaction between threads relative to many real-world programs, in which each thread will often perform a lot of work without interacting with any other threads. This means that the algorithms, particularly DPOR, are likely to be more effective on such programs.

The numbers of transitions explored were easily obtained by counting the numbers of recursive calls made. The execution times were measured using the `time` function in the Sys module packaged with OCaml. It returns measurements of time only in quanta of 0.004 seconds, so the accuracy of the time measurements is poor for executions not much longer than this time. The precision of measurements is good, though, with very similar results being returned for multiple executions of the same program. Having established this, I took only one measurement for each, preferring to focus effort on collecting data from more examples.

Example 1 – Repeated Accesses

Each thread in this example is assigned a particular shared object, which it accesses a fixed number of times. There are three test parameters: the total number of threads, the number of threads assigned to each shared object, and the number of accesses that each thread makes. This example allows the amount of dependence between the threads to be controlled by varying this second parameter.

Example 2 – Bounded Buffer

This is an instance of Dijksta's classic bounded-buffer problem [9], in which two threads operate on a shared fixed-length array of items, with the "producer" writing new items to the buffer and the "consumer" removing them. I used a buffer of size 8 and effected mutual exclusion using per-item locks. By varying the number of items that each thread processes as the test variable, this example explores how the algorithms perform as the number of instructions per thread increases on a more realistic example.

Example 3 – Indexer

Taken from the paper introducing DPOR [13, Figure 1], each thread "receives" a number of "messages", each of which is inserted into a shared hash table. If a slot is already occupied, the next free slot is tried instead. This example was chosen to allow comparison with the published results [13, Figure 8], and to evaluate the algorithms on a realistic use case.

Example 4 – File System

Also from the DPOR paper [13, Figure 7], this example is based an idiom found in the Frangipani file system [31]. Each thread searches for a free "disk block" for an "inode" to point to. This example was chosen for the same reasons as Example 3.

4.2.2 Results

For each example program, the results for all eight algorithms are presented on the same graph, in the style indicated in Figure 4.1. If the timing function returned 0.000s, I plotted a point at 0.002s for clarity, since $\log(0)$ is undefined. I stopped collecting data when execution times approached 100 seconds.

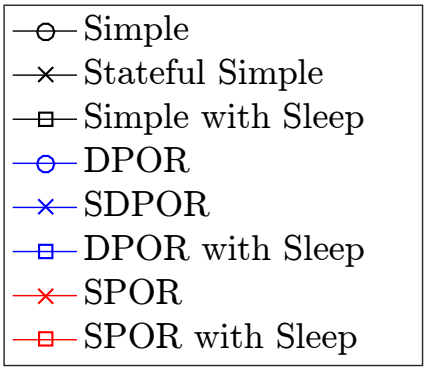
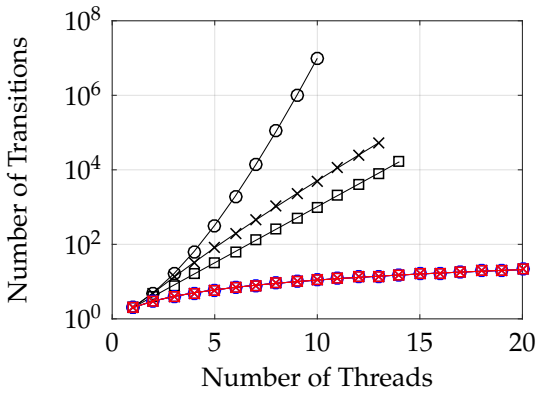
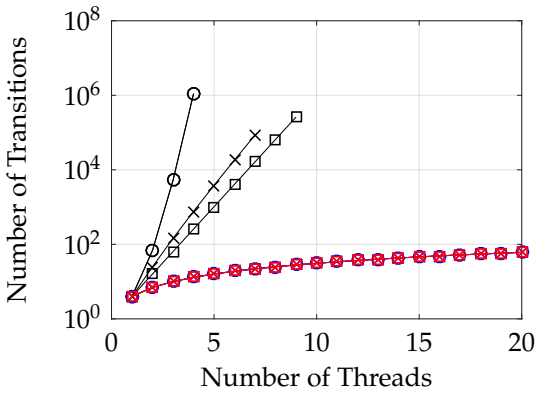


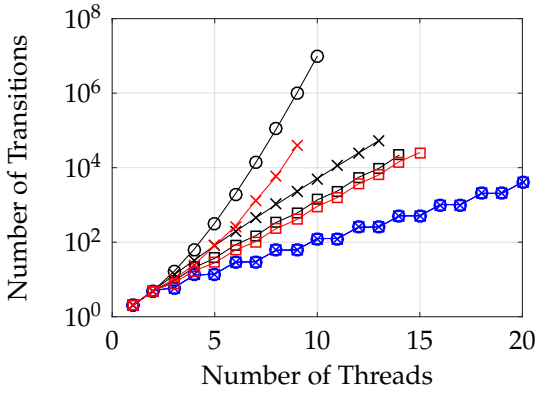
Figure 4.1: Key for model-checking algorithm performance graphs.



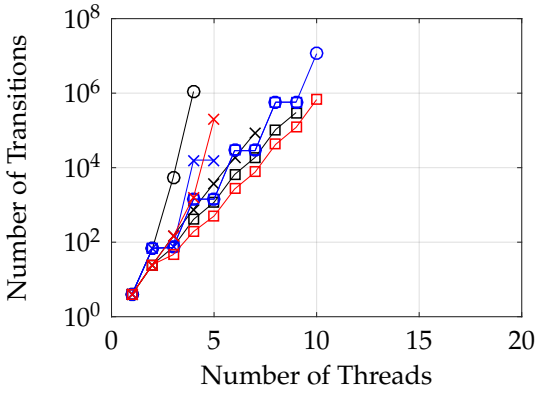
(a) 1 thread per object, each making 1 access.



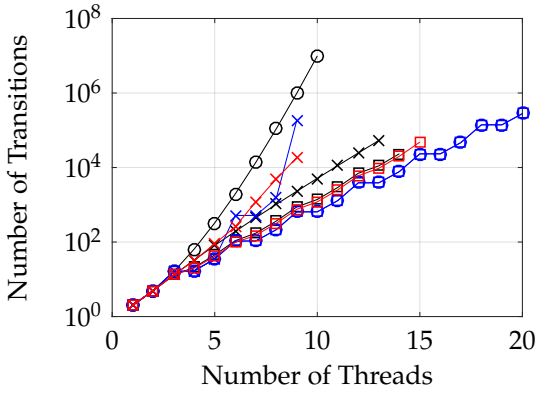
(b) 1 thread per object, each making 3 accesses.



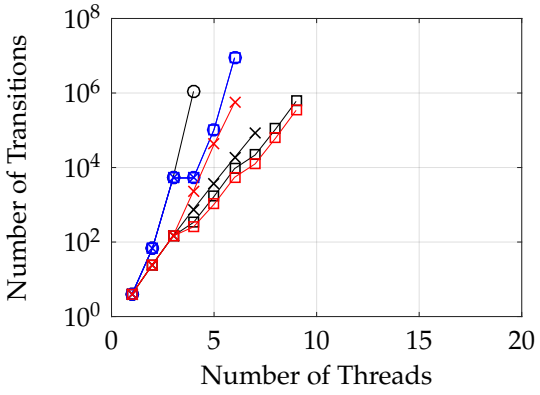
(c) 2 threads per object, each making 1 access.



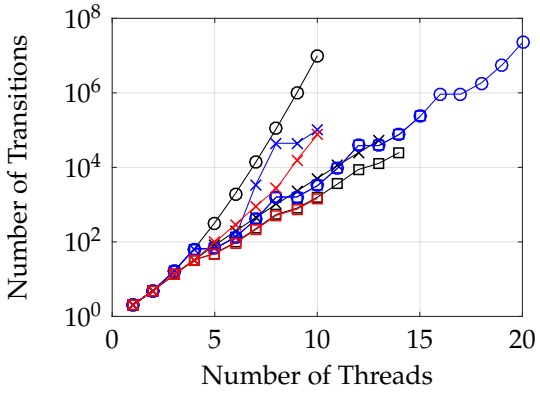
(d) 2 threads per object, each making 3 accesses.



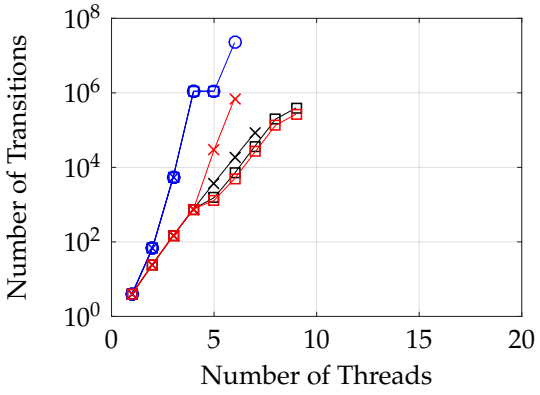
(e) 3 threads per object, each making 1 access.



(f) 3 threads per object, each making 3 accesses.

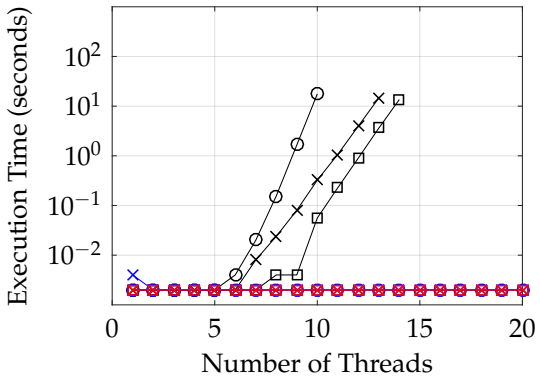


(g) 4 threads per object, each making 1 access.

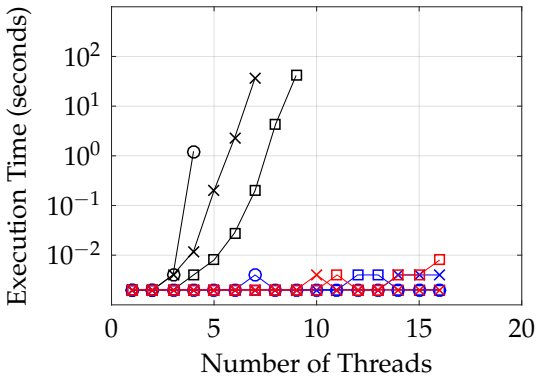


(h) 4 threads per object, each making 3 accesses.

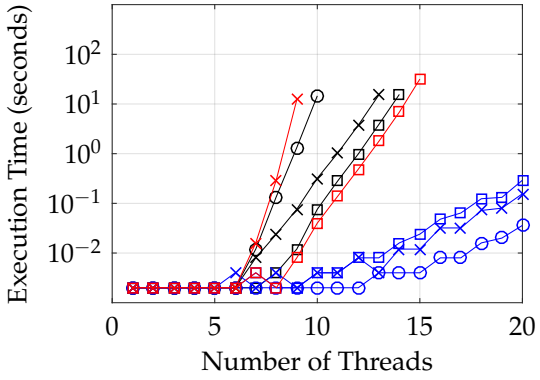
Figure 4.2: Numbers of transitions explored for the repeated-accesses example.



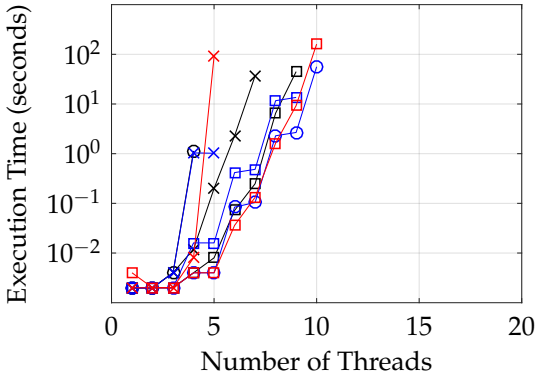
(a) 1 thread per object, each making 1 access.



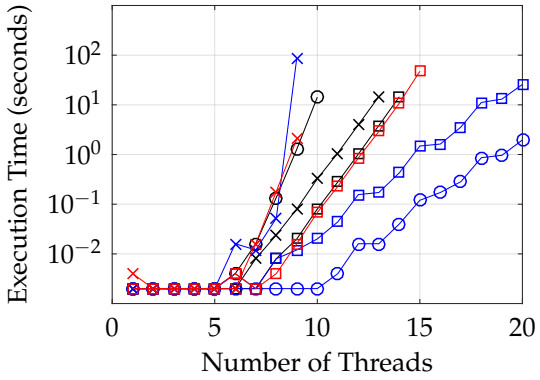
(b) 1 thread per object, each making 3 accesses.



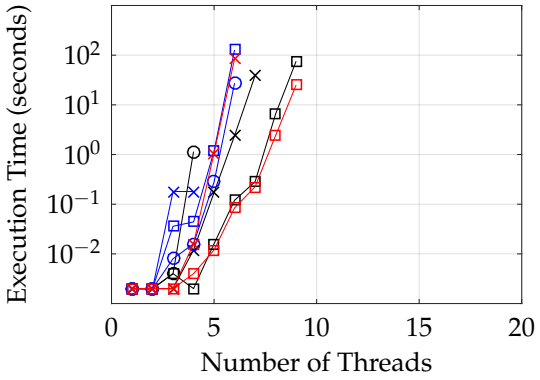
(c) 2 threads per object, each making 1 access.



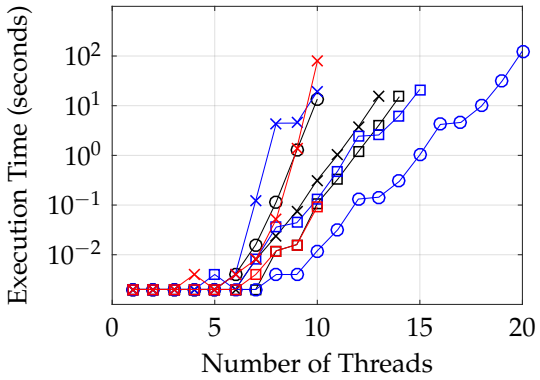
(d) 2 threads per object, each making 3 accesses.



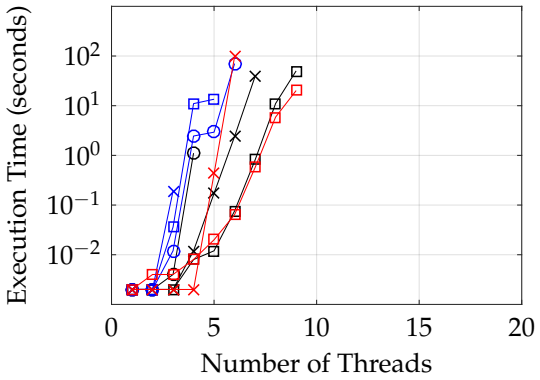
(e) 3 threads per object, each making 1 access.



(f) 3 threads per object, each making 3 accesses.



(g) 4 threads per object, each making 1 access.



(h) 4 threads per object, each making 3 accesses.

Figure 4.3: Execution times for the repeated-accesses example.

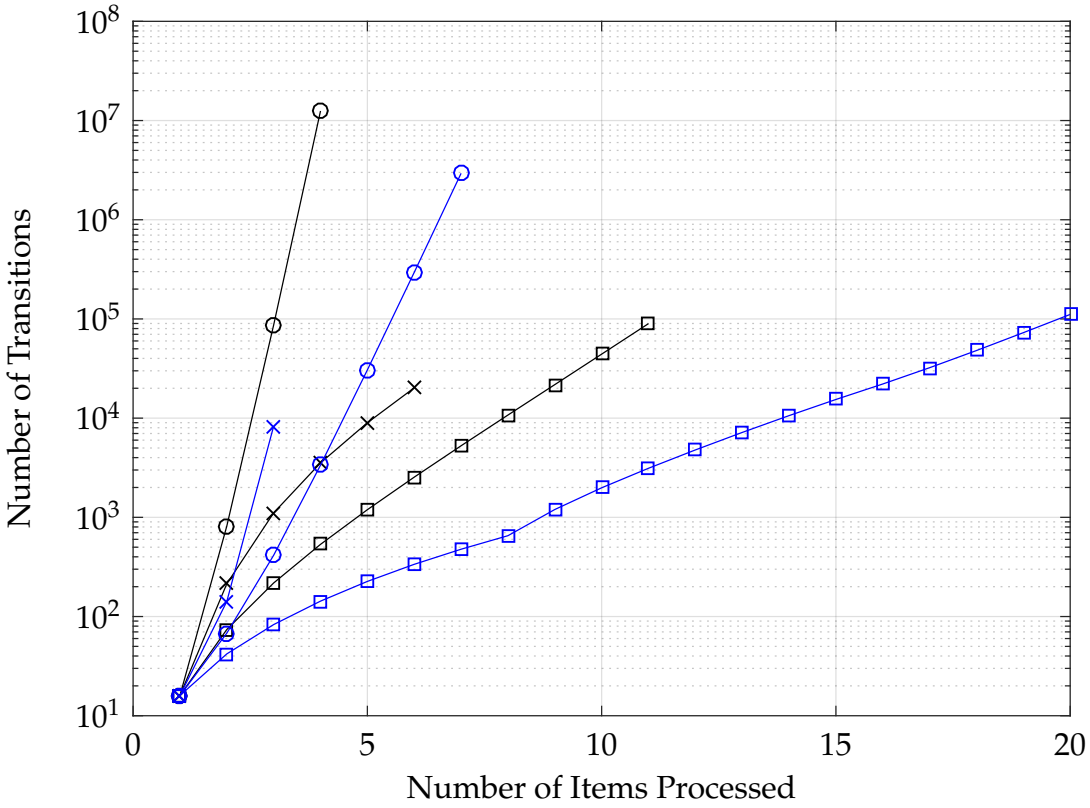


Figure 4.4: Numbers of transitions explored for the bounded-buffer example.

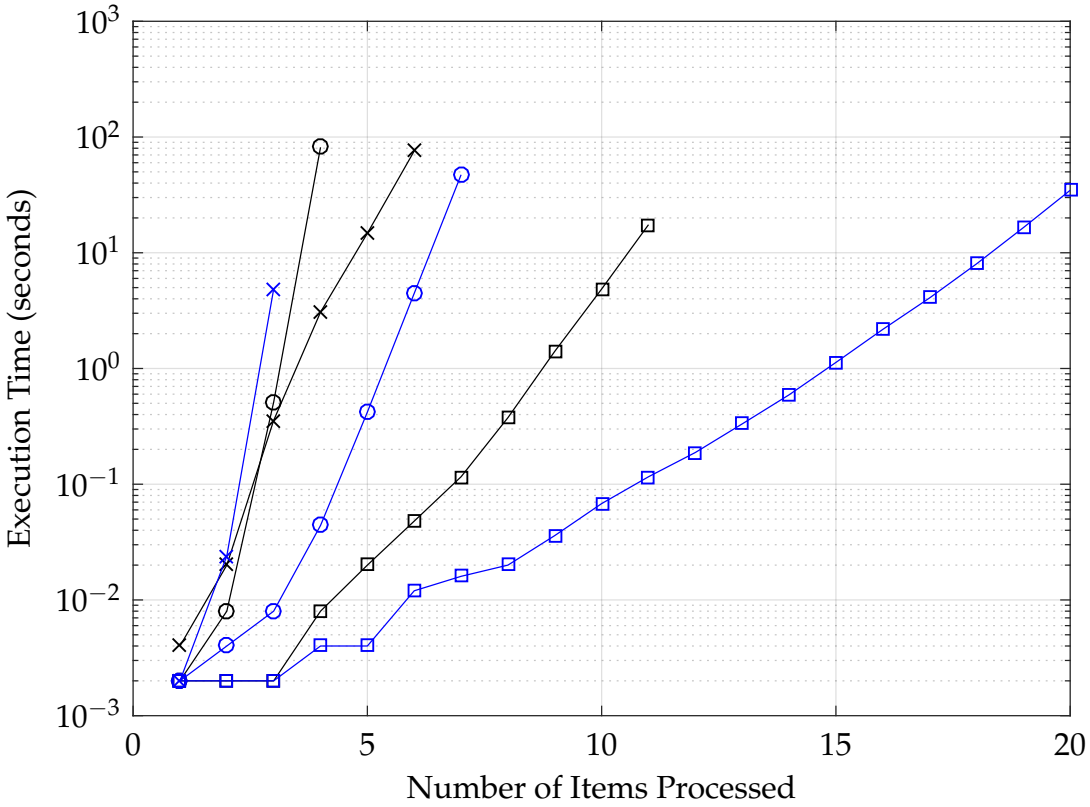


Figure 4.5: Execution times for the bounded-buffer example.

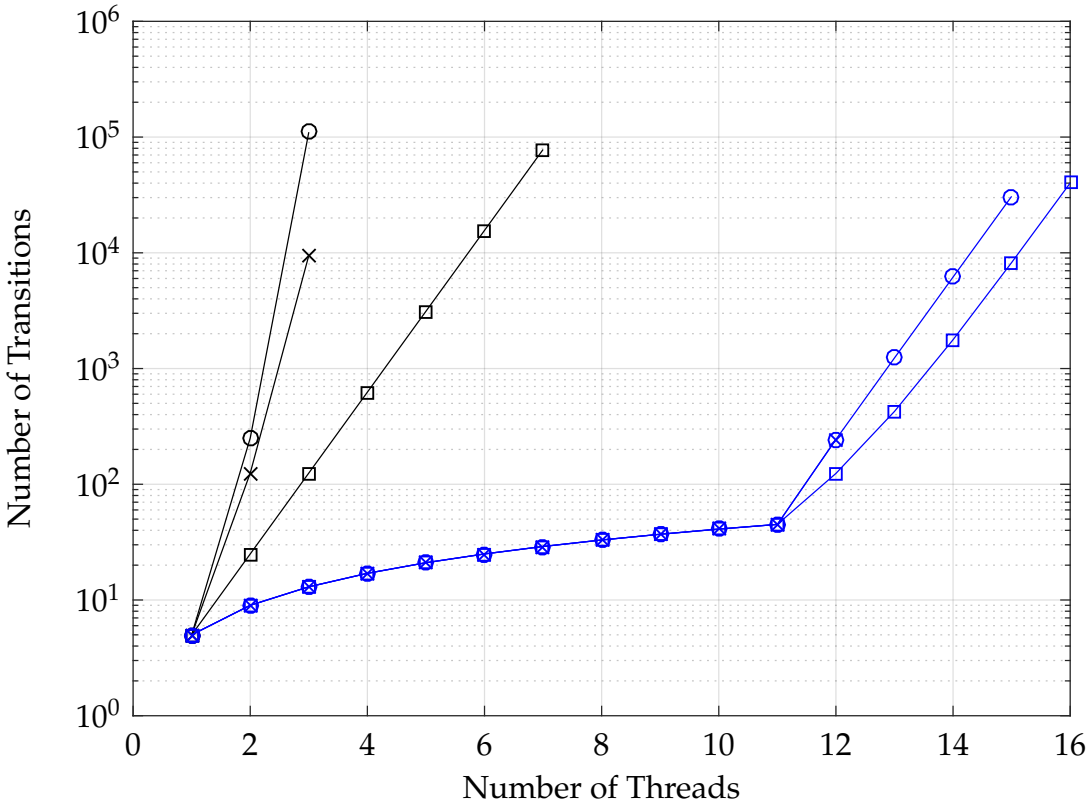


Figure 4.6: Numbers of transitions explored for the indexer example.

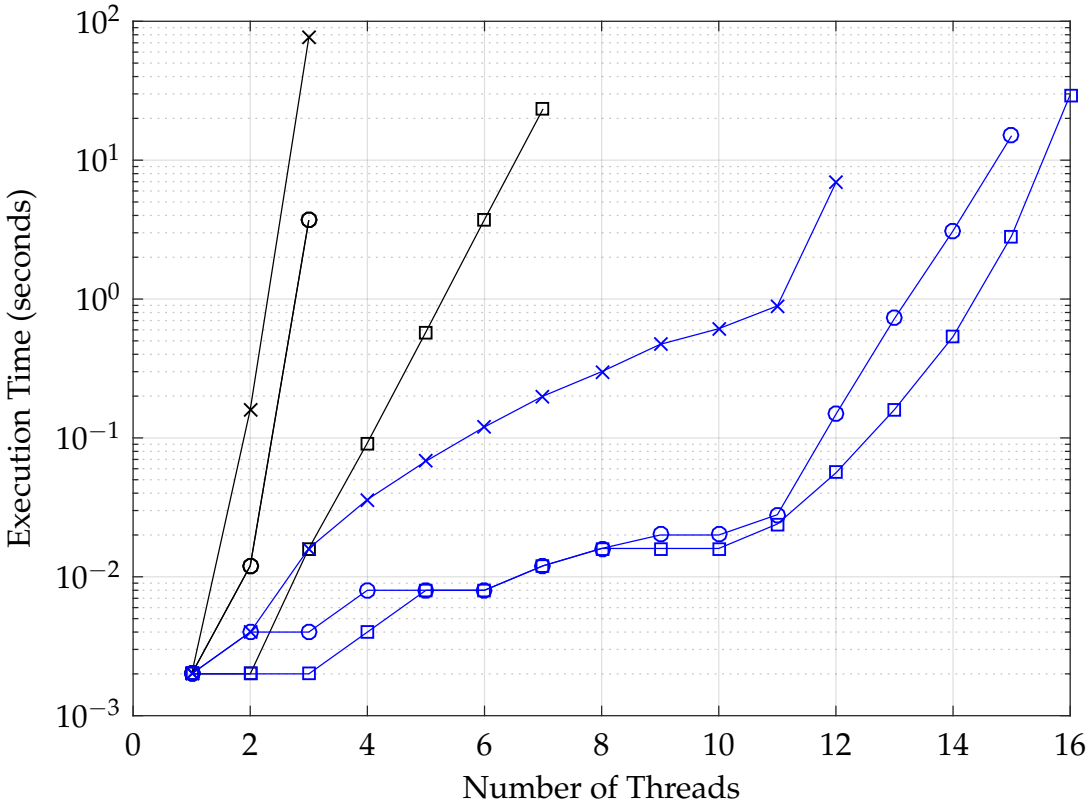


Figure 4.7: Execution times for the indexer example.

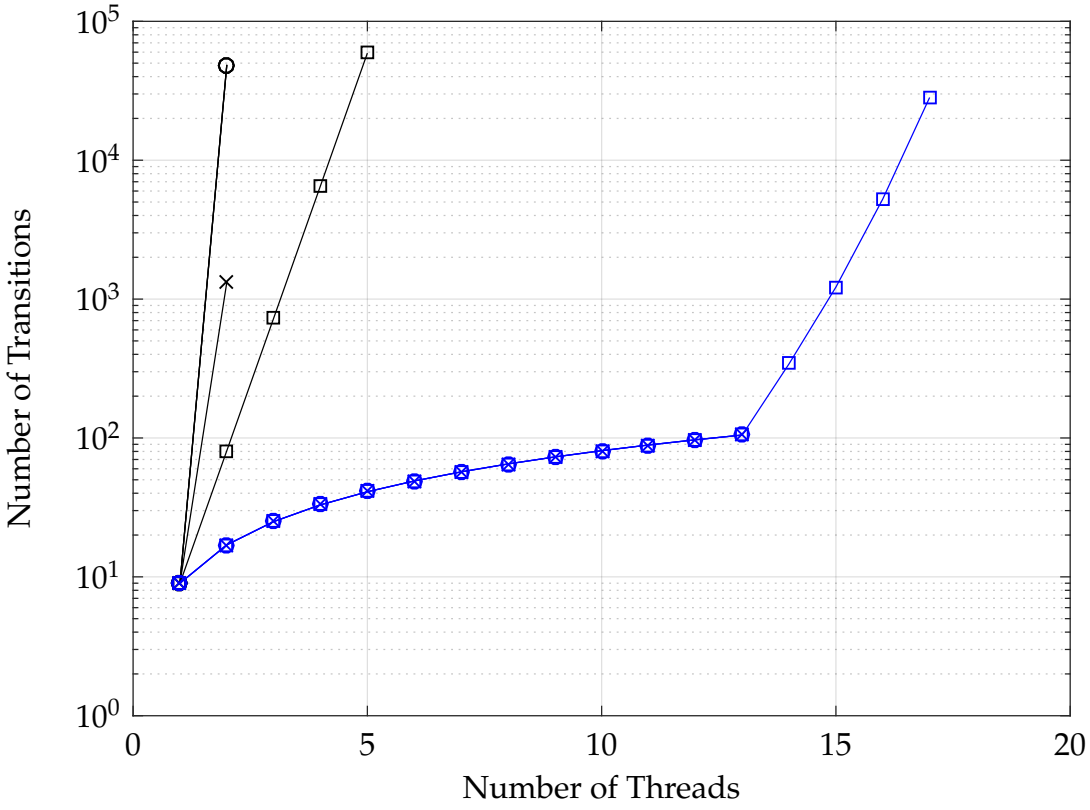


Figure 4.8: Numbers of transitions explored for the file-system example.

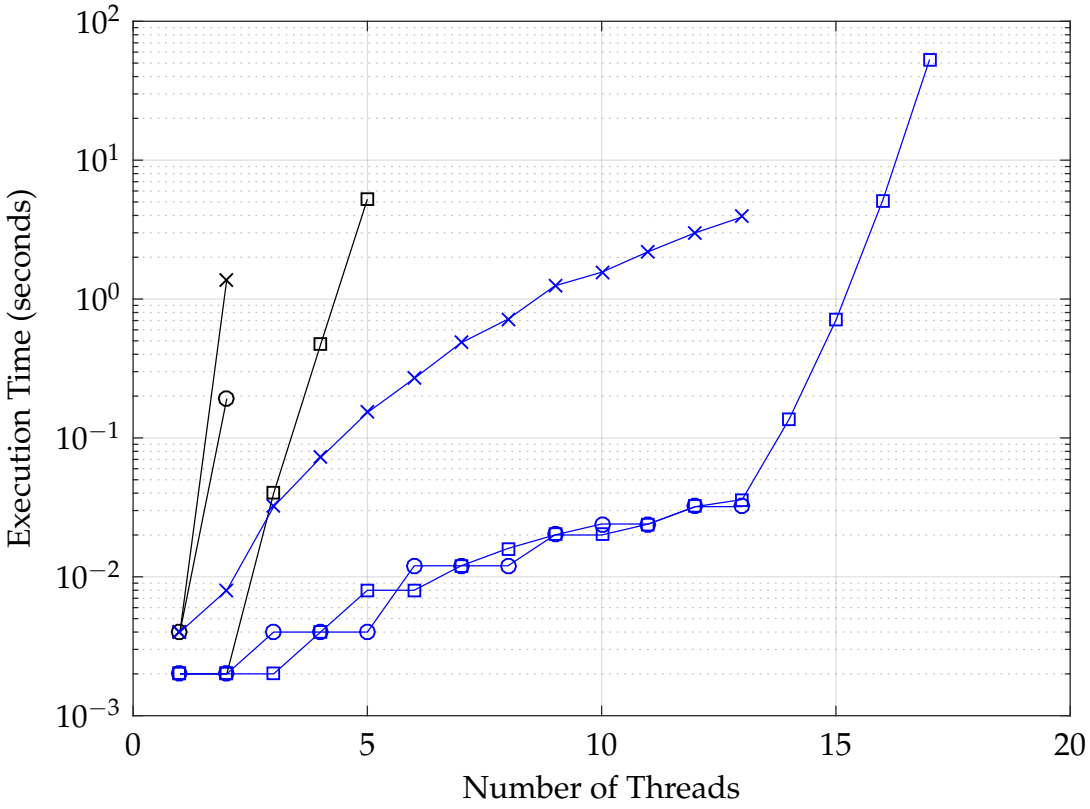


Figure 4.9: Execution times for the file-system example.

4.2.3 Performance of the Algorithms

Simple Model Checker

The simple model checker provides a baseline, exploring every path from the initial state to a stopped state exactly once. Due to the state explosion problem, this number increases exponentially with the complexity of the program being verified. As constant work is done per thread per transition explored, the execution times of the simple algorithm are proportional to the number of transitions it explores.

Stateful Simple Model Checking

The stateful simple model-checker can be seen to perform consistently well, in terms of reduction of the explored state space. If two transitions are "actually" independent, meaning that executing them in either order results in the same state, then the stateful algorithm should explore from only one of these; the prevalence of such pairs of transitions explains the success of this approach.

Despite the reductions in the numbers of transitions explored, the cost of the hash-table operations necessary for stateful checking means that the execution times are sometimes worse than for the simple algorithm—for example, in Figure 4.7. However, looking at Figure 4.3 suggests that this is only a short-term problem, and as the size of the state space increases, the benefits of stateful model checking dominate its costs.

Dynamic Partial-Order Reduction

When all transitions are independent, as in the first two cases of Figure 4.2, the DPOR algorithm only explores one execution path, with negligible cost. In the other cases in Figure 4.2, adding an extra thread is either of negligible cost (if and only if the new thread is independent of the others), or contributes a multiplicative increase to the cost of the search. These multiplicative increases are still not as severe as in the simple algorithm, though, because the new thread is independent of at least some of the existing threads.

In the bounded-buffer example (Figure 4.4), the DPOR algorithm is able to exploit the independence between the two threads (they are dependent only when they access the same item in the buffer) to slow the exponential cost caused by the state explosion problem.

The remaining two examples are more realistic in nature, and exemplify the benefits that DPOR can offer. In the indexer example (Figure 4.6), dependence between threads is caused only by collisions in the hash table—for the simple hash function used, this happens at twelve threads, but for realistic hash functions would happen much later. Since there are no race conditions if there are no collisions, only one execution path is explored for fewer than twelve threads. Similar behaviour is seen in Figure 4.8, in which only one path is explored until similar "collisions" when accessing a table occur.

Theoretically, DPOR does a fixed amount of work per thread per transition, including some memory operations to keep track of the backtrack sets. As the number of

transitions explored varies with the number of threads much more than the amount of work per transition explored does, I would expect the execution times to be roughly proportional to the numbers of transitions explored. Incidentally, the fact that this is consistently the case, and that the graphs of transitions explored match their published counterparts [13], is further evidence that my implementation is correct and not unnecessarily complex.

Stateful Dynamic Partial-Order Reduction

In the cases given above, the SDPOR algorithm consistently performs poorly, exploring at least as many transitions as the plain DPOR algorithm. However, in the examples given by Yang et al. [37], SDPOR consistently performs better. The explanation for this seems to be that their examples are much larger, meaning that when a state is re-encountered, a larger saving is typically made, while the number of backtracking points introduced does not increase in the same way. It would be interesting to investigate exactly when the SDPOR algorithm outperforms stateless DPOR.

However, as seen in Figures 4.7 and 4.9, the execution times for the SDPOR algorithm are much worse than what might be expected given the number of transitions it explores. This can be explained by the increasing costs of the hash-table operations performed at each state, since each state occupies quite a lot of memory.

As well as introducing SDPOR, Yang et al. also introduced a light-weight scheme for capturing the local states of threads, which they used in conjunction with SDPOR to improve its implementation efficiency. Had I implemented this additional technique, the average cost of the hash-table operations would have decreased, although the large numbers of transitions explored would not have been affected.

Sleep Sets

For the simple model-checker, sleep sets significantly reduce the rate of exponential increase in the number of transitions explored in every example, by preventing the unnecessary re-exploration of sections of the state space (more effectively than the simple stateful algorithm).

When applied to the DPOR algorithm, the results are more variable. While little benefit is seen from applying sleep sets to DPOR in Figures 4.3 and 4.6 (presumably because the DPOR algorithm has already fully exploited the independence between the threads), better results are seen in Figures 4.4 and 4.8, in which sleep sets work in conjunction with DPOR to greatly slow the rate of the exponential increase in the number of transitions explored. Although the effectiveness of the technique depends on the particular program being verified, it seems that on more realistic examples, it is highly effective.

There is a time and space cost in applying the sleep-set technique. In the repeated-accesses example, using sleep sets does not reduce the number of explored transitions when applied to DPOR, so this time cost can be clearly seen in Figure 4.3. However, in the other examples, and all realistic programs, these costs are insignificant compared to the considerable benefits available.

Static Partial-Order Reduction

When every thread accesses separate objects, then my SPOR algorithm returns persistent sets of size one, which is optimal, as shown in the first two cases of Figure 4.2. However, this is very much a corner case, and my SPOR algorithm usually offers only modest reductions in the number of transitions explored—as the number of threads sharing each object increase, the benefits of the algorithm decrease.

In particular, if the threads of a program cannot be partitioned such that the objects accessed by different partitions are disjoint, my SPOR algorithm offers no improvements at all. Unfortunately, this is the case for most concurrent programs, including my other examples, from which the SPOR results have therefore been omitted.

The extra work done per transition in implementing my SPOR algorithm is by design little (cf. Section 3.5.3), so execution times are roughly proportional to the number of transitions explored.

Once again, attempting to implement an SPOR algorithm mostly served to give me a greater understanding of the difficulty of trying to statically compute reasonably small persistent sets.

Chapter 5

Conclusions

5.1 Summary and Reflections

As well as meeting the aims of the project, I also managed to implement several extensions, allowing a fuller analysis of the benefits of DPOR in relation to other related techniques. The performance of the implemented algorithms was quantitatively evaluated, revealing that DPOR is often very effective at mitigating the state explosion problem, especially when used in conjunction with the sleep-set technique.

Despite the considerable challenges, particularly in understanding the model-checking algorithms and their underlying theory, the satisfaction of eventually achieving my goals was gratifying. Besides being a subject that I am interested in, I am glad that I chose this project because it has allowed me to develop the important skills of reading academic papers, translating high-level algorithms into executable programs, and developing software.

5.2 Further Work

Instead of assuming that any two processes that access the same object must be dependent, it is possible to distinguish between reads and writes to shared objects, allowing two consecutive reads to be identified as independent (in the sense of Section 2.4.1). Flanagan and Godefroid suggest that this could be implemented by keeping two clock vectors per shared object—one for reads and the other for writes [13]. It would be interesting to implement this algorithm and evaluate its performance in relation to the original DPOR algorithm.

I would also like to use my system to verify programs written in a widely-used language; as mentioned previously, the only work necessary for this is an implementation of the *Expression* and *Thread* modules. Given the complexity of some modern programming languages, though, this might not be trivial. A better approach for Java, for example, might be to implement a JVM which performs DPOR, inspired by Java PathFinder [34].

Finally, I would be be interested in spending time looking at recent developments in model checking that have been made since the introduction of DPOR.

Bibliography

[1] Bernhard Beckert and Vladimir Klebanov. Must program verification systems and calculi be verified? In *Proceedings, 3rd International Verification Workshop (VERIFY), Workshop at Federated Logic Conferences (FLoC), Seattle, USA*, 2006.

[2] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, UK, 1999. Springer-Verlag.

[3] Guillaume Brat, Doron Drusinsky, Dimitra Giannakopoulou, Allen Goldberg, Klaus Havelund, Mike Lowry, Corina Pasareanu, Arnaud Venet, Willem Visser, and Rich Washington. Experimental evaluation of verification and validation tools on martian rover software. *Formal Methods in System Design*, 25(2):167–198.

[4] N. G. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *INDAG. MATH*, 34:381–392, 1972.

[5] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. *Computer Aided Verification: 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000. Proceedings*, chapter Counterexample-Guided Abstraction Refinement, pages 154–169. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[6] Edmund M. Clarke. 25 years of model checking. chapter The Birth of Model Checking, pages 1–26. Springer-Verlag, Berlin, Heidelberg, 2008.

[7] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer-Verlag.

[8] Darren Cofer. Model checking: Cleared for take off. In *Proceedings of the 17th International SPIN Conference on Model Checking Software*, SPIN'10, pages 76–87, Berlin, Heidelberg, 2010. Springer-Verlag.

[9] Lee W Cooprider, F Heymans, Pierre-Jacques Courtois, and David L Parnas. Information streams sharing a finite buffer: other solutions. *Information Processing Letters*, 3(1):16–21, 1974.

[10] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Commun. ACM*, 22(5):271–280, May 1979.

[11] Marie-Aude Esteve, Joost-Pieter Katoen, Viet Yen Nguyen, Bart Postma, and Yuri Yushtein. Formal correctness, safety, dependability, and performance analysis of a satellite. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1022–1031, Piscataway, NJ, USA, 2012. IEEE Press.

[12] Cormac Flanagan and Patrice Godefroid. Addendum to dynamic partial-order reduction for model checking software. Online, 2005.

[13] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 110–121, New York, NY, USA, 2005. ACM.

[14] Patrice Godefroid. *Computer-Aided Verification: 2nd International Conference, CAV '90 New Brunswick, NJ, USA, June 18–21, 1990 Proceedings*, chapter Using partial orders to improve automatic verification methods, pages 176–185. Springer Berlin Heidelberg, Berlin, Heidelberg, 1991.

[15] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[16] Patrice Godefroid and Didier Pirottin. *Computer Aided Verification: 5th International Conference, CAV '93 Elounda, Greece, June 28–July 1, 1993 Proceedings*, chapter Refining dependencies improves partial-order verification methods (extended abstract), pages 438–449. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993.

[17] Pim Kars. The application of promela and spin in the bos project, 1996.

[18] R. P. Kurshan. *25 Years of Model Checking: History, Achievements, Perspectives*, chapter Verification Technology Transfer, pages 46–64. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[19] Gavin Lowe. *Tools and Algorithms for the Construction and Analysis of Systems: Second International Workshop, TACAS '96 Passau, Germany, March 27–29, 1996 Proceedings*, chapter Breaking and fixing the Needham-Schroeder Public-Key Protocol using FDR, pages 147–166. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[20] Antoni Mazurkiewicz. *Petri Nets: Applications and Relationships to Other Models of Concurrency: Advances in Petri Nets 1986, Part II Proceedings of an Advanced Course Bad Honnef, 8.–19. September 1986*, chapter Trace theory, pages 278–324. Springer Berlin Heidelberg, Berlin, Heidelberg, 1987.

[21] Steve McConnell. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA, 2004.

[22] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[23] Steven P. Miller. *Theorem Proving in Higher Order Logics: 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, chapter Will This Be Formal?, pages 6–11. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[24] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How amazon web services uses formal methods. *Commun. ACM*, 58(4):66–73, March 2015.

[25] Doron Peled. *Computer Aided Verification: 6th International Conference, CAV '94 Stanford, California, USA, June 21–23, 1994 Proceedings*, chapter Combining partial order reductions with on-the-fly model-checking, pages 377–390. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994.

[26] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, UK, 1982. Springer-Verlag.

[27] T. Schlipf, T. Buechner, R. Fritz, M. Helms, and J. Koehl. Formal verification made easy. *IBM J. Res. Dev.*, 41(4-5):567–576, July 1997.

[28] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5(2):285–309, 1955.

[29] G. Tassey. The economic impacts of inadequate infrastructure for software testing, 2002.

[30] Heikki Tauriainen and Keijo Heljanko. Testing ltl formula translation into büchi automata. *International Journal on Software Tools for Technology Transfer*, 4(1):57–70.

[31] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. *SIGOPS Oper. Syst. Rev.*, 31(5):224–237, October 1997.

[32] Antti Valmari. A stubborn attack on state explosion. In *Proceedings of the 2Nd International Workshop on Computer Aided Verification*, CAV '90, pages 156–165, London, UK, UK, 1991. Springer-Verlag.

[33] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency: Structure versus Automata, volume 1043 of Lecture Notes in Computer Science*, pages 238–266. Springer-Verlag, 1996.

[34] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[35] Anh Vo, Sarvani Vakkalanka, Michael DeLisi, Ganesh Gopalakrishnan, Robert M. Kirby, and Rajeev Thakur. Formal verification of practical mpi programs. *SIGPLAN Not.*, 44(4):261–270, February 2009.

[36] Yu Yang. *Efficient Dynamic Verification of Concurrent Programs*. PhD thesis, Salt Lake City, UT, USA, 2009. AAI3351199.

[37] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Efficient stateful dynamic partial order reduction. In *Proceedings of the 15th International Workshop on Model Checking Software*, SPIN '08, pages 288–305, Berlin, Heidelberg, 2008. Springer-Verlag.

Appendix A

Details of DPOR

A.1 Explanation of Theory

1: **procedure** EXPLORE(π)
2: **let** $\sigma = last(\pi)$ **in**
3: **for all** $p \in \mathcal{P}$ **do**
4: UPDATEBACKTRACKSETS(π, $next(\sigma, p)$)
5: **if** $enabled(\sigma) \neq \emptyset$ **then**
6: **let** t = any $t \in enabled(\sigma)$ **in**
7: **let** $backtrack(\sigma) = \{t\}$ **in**
8: **let** $done(\sigma) = \emptyset$ **in**
9: **while** $done(\sigma) \neq backtrack(\sigma)$ **do**
10: **let** t = any $t \in (backtrack(\sigma) \setminus done(\sigma))$ **in**
11: add t to $done(\sigma)$;
12: EXPLORE($\pi.t$)
13:
14: **procedure** UPDATEBACKTRACKSETS(π, $t_{p,s}$)
15: **let** $D = \{i \in dom(\pi) \mid \pi_i$ is dependent with $t_{p,s}$ and $i \not\leftrightarrow_\pi p\}$ **in**
16: **if** $D \neq \emptyset$ **then**
17: **let** $\sigma_d = pre(\pi, \max(D))$ **in**
18: **if** $next(\sigma_d, p) \in enabled(\sigma_d)$ **then** add $next(\sigma_d, p)$ to $backtrack(\sigma_d)$
19: **else** add all of $enabled(\sigma_d)$ to $backtrack(\sigma_d)$
20:
21: Initially: EXPLORE($\emptyset$)

Lines 1–2 The EXPLORE procedure implements dynamic partial-order reduction. The transition sequence π which is passed as its argument implicitly specifies the state from which the exploration is to take place, σ.

Lines 3–4 First, any necessary backtracking points which can be identified from this state are added to the *backtrack* sets of previous states. Backtracking points can be identified using information from each possible transition from this state, so the UPDATEBACKTRACKSETS procedure is called with each possible next transition as an argument.

Line 5 Having updated the *backtrack* sets of previous states, the search continues its exploration, but only if there are any enabled transitions to explore.

Lines 6–8 The recursive exploration proceeds by maintaining two sets of transitions: *backtrack* and *done*. The *backtrack* set contains those transitions from σ which should be explored, and is initialised to contained any single enabled transition. Recursive calls to EXPLORE may add more transitions to *backtrack*. The *done* set contains those transitions in *backtrack* that have been explored.

Line 9 The search from σ halts when every transition that needs to be explored (is in *backtrack*) has been explored (is in *done*).

Lines 10–12 If there is some transition that needs exploring, EXPLORE is recursively called with that transition extending the current transition sequence π, and that transition is added to *done*.

Line 14 The UPDATEBACKTRACKSETS procedure can use the information that a given transition, $t_{p,s}$ can be executed in σ to add necessary backtracking points.

Line 15 The set D contains the indices i of the transitions π_i in the sequence such that π_i and t are dependent, and $(i, p) \not\leftrightarrow_\pi$. This second condition means that $t_{p,s}$ is *not* guaranteed to happen after π_i, so D is the set of transitions which may have a race condition with $t_{p,s}$.

Line 16 If there are no transitions which have a race condition with $t_{p,s}$, then no backtracking points need to be added: as far as we know from $t_{p,s}$, if we had chosen any other choices of transitions to explore previously, we would have ended up with a transition sequence in the same equivalence class as π.

Line 17 If there are transitions in π which have a race condition with $t_{p,s}$, then it is sufficient to add a backtracking point at the state immediately before the most recent of these; the necessary backtracking points for the earlier states will be added by later (or previous) calls of EXPLORE.

Line 18 We know there is a race condition between $t_{p,s}$ and $next(\sigma_d, p)$, so we want to explore another transition sequence from σ_d in which $next(\sigma_d, p)$ appears before $t_{p,s}$. If $next(\sigma_d, p)$ is enabled in σ_d, then we certainly explore such a transition sequence by immediately exploring $next(\sigma_d, p)$ from σ_d.

Line 19 However, if we cannot immediately explore $next(\sigma_d, p)$ from σ_d because it is not enabled, then it is not obvious which transitions to explore from σ_d to achieve a transition sequence in which $next(\sigma_d, p)$ appears before $t_{p,s}$. The algorithm presented here does not attempt to narrow down which transitions might lead to such a transition sequence, and instead plays it safe and explores them all.

Lines 21 To perform model checking, the state space is explored from the initial state, σ_0, which is reached from σ_0 by executing no transitions, so EXPLORE is initially called on the empty sequence of transitions, $\emptyset$.

A.2 OCaml Implementation

Below is my OCaml implementation of the DPOR algorithm. Note that many of the comments are references to the original pseudocode [13, Figures 3–5], rather than the pseudocode in this dissertation, which is simplified for ease of understanding.

```
let rec check init_prog t_seq proc_cvs obj_cvs last_ti =
    (* Track execution info *)
    calls := !calls + 1;
    let depth = List.length t_seq in
    if depth > !max_depth then max_depth := depth;

    let error_free = ref true in
    let one_thread_can_advance = ref false in
    let all_stopped_threads_are_values = ref true in
    let recursive_calls_deadlock_free = ref true in

    (* Used to determine if there is some enabled p *)
    let transition_to_explore = ref None in

    let pre i =
        let rec pre_aux tran_list index state =
            if index = i then state else
                (pre_aux (List.tl tran_list) (index+1)
                    (ProgImp.apply_transition state (List.hd tran_list)))
        in pre_aux t_seq 0 init_prog in

    (* let s = last(S) *)
    let (tds, g) = pre depth in

    (* for all processes p *)
    for p = 0 to Array.length tds - 1 do

        let (e, s) = Array.get tds p in

        (* Is there next transition? *)
        match ProgImp.ThrImp.next_transition (e, s, g) with

          (* No futher transitions: check if this thread reaches a local
             error *)
          None -> let (reaches_err, reaches_nonval) =
                    ProgImp.ThrImp.check_local (e, s, g) in
                  if reaches_err then (
                    error_free := false;
                    if write_error_traces then ProgImp.output_hasse_image
                        ("errortraces/errortrace" ^ (string_of_int !calls) ^
                            ".gv") t_seq
```

```
              );
              if reaches_nonval then
                all_stopped_threads_are_values := false

        (* There is a transition *)
      | Some (next_t, enabled) ->
          if enabled then
              (* There is at least one transition to explore: this one *)
              (one_thread_can_advance := true;
              transition_to_explore := Some p)
          else all_stopped_threads_are_values := false;

          (* let i = L(alpha(next(s,p))) *)
          let i = last_ti next_t.T.g_loc in

          (* if i =/= ~-1 and not i <= C(p)(proc(Si)) *)
          if i > ~-1 then if
            i > Clockvector.get (proc_cvs p) (fst (List.nth t_seq i))
          then (
              let prei = pre i in
              let is_enabled p' (tds', g') =
                  let (e', s') = Array.get tds' p' in
                  match ProgImp.ThrImp.next_transition (e', s', g') with
                  Some (_, b) -> b
                | None -> false
              (* if p in enabled(pre(S, i)) *)
              in if is_enabled p prei
              (* then add p to backtrack(pre(S, i)) *)
              then Var_array.set backtracks i (p::(Var_array.get backtracks
                  i))
              (* else add enabled(pre(S,i)) to backtrack(pre(S,i)) *)
              else let rec ntoz n = if n = 0 then [0] else n::(ntoz (n-1))
                   in let en_in_prei q = is_enabled q prei
                   in Var_array.set backtracks i (List.filter en_in_prei
                         (ntoz (Array.length tds - 1)))
              )
  done;

  (* if there is some p in enabled(s) *)
  (* As we have no locks, all available transitions are enabled *)
  (match !transition_to_explore with None -> () | Some pi -> (

  (* backtrack(s) := {p} *)
  Var_array.append backtracks [pi];

  let rec find_p bs ds = match bs with
      [] -> None
```

```
  | x::xs -> if List.mem x ds then find_p xs ds else Some x in

(* let done = emptyset *)
(* while there is some p in backtrack(s) but not done *)
let rec whileloop po dones = match po with None -> () | Some p -> ((

let (e, s) = Array.get tds p in
match T.next_transition (e, s, g) with
    None -> raise (Failure "Should NEVER happen; there is a transition")
  | Some (next_t, enabled) -> (

if not enabled then raise (Failure "Only enabled transitions should be
    explored");

(* let S' = S.next(s,p) *)
let new_t_seq = t_seq @ [(p, next_t)] in

(* let o = alpha(next(s,p)) *)
let o = next_t.T.g_loc in

(* let cv = max(C(p),C(o))[p:=|S'|] *)
let cv = Clockvector.max (proc_cvs p) (obj_cvs o) in
Clockvector.set cv p (List.length new_t_seq);

(* let C' = C[p:=cv, o:=cv] *)
let new_proc_cvs pi = if pi = p then cv else proc_cvs pi in
let new_obj_cvs oi = if oi = o then cv else obj_cvs oi in

(* let L' = L[o:=|S'|] *)
let new_last_ti oi = if oi = o then List.length new_t_seq - 1 (* index *)
                               else last_ti oi in

(* Explore(S', C', L') *)
let (ef, df) = check init_prog new_t_seq new_proc_cvs new_obj_cvs
    new_last_ti in
error_free := !error_free && ef;
recursive_calls_deadlock_free := !recursive_calls_deadlock_free && df;

(* go back to top of while loop if necessary *)
(* also contains add p to done *)
)); whileloop (find_p (Var_array.get backtracks (Var_array.length
    backtracks - 1)) (p::dones)) (p::dones))
(* also contains let done = emptyset *)
in whileloop (find_p (Var_array.get backtracks (Var_array.length
    backtracks - 1)) []) [];
    Var_array.remove_last backtracks (* Decrement size of array to
        counter earlier increment *)));
```

```
      (!error_free, !recursive_calls_deadlock_free &&
         (!one_thread_can_advance || !all_stopped_threads_are_values))

  let error_and_deadlock_free (tds, g) =
```

Appendix B

Other Model-Checking Algorithms

B.1 Simple Model Checker

```
1:  procedure CHECK(σ_0, π)
2:      let (l, g) = last(π) in
3:      let error_free = true in
4:      let is_enabled_thread = false in
5:      let no_waiting_threads = true in
6:      let calls_deadlock_free = true in
7:      for all p ∈ P do
8:          if next(σ, p) ∈ enabled(σ) then
9:              is_enabled_thread := true;
10:             let (result_ef, result_df) = CHECK(σ_0, π.next(σ, p)) in
11:             error_free := error_free & result_ef;
12:             calls_deadlock_free := calls_deadlock_free & result_df
13:         else
14:             if err(l(p)) then error_free := false;
15:             if ∃g' ∈ G. next(σ, p) ∈ enabled(l, g') then no_waiting_threads := false
16:     let deadlock_free = calls_deadlock_free & (is_enabled_thread | no_waiting_threads) in
17:     return (error_free, deadlock_free)
```

The algorithm assumes that if a thread encounters an error, then the execution of that thread will not proceed, meaning that only threads which do not have any enabled transitions need to be checked for errors. The Boolean *error_free* is set to false if an error is encountered either in this state (line 14) or in a recursive call (line 11).

Deciding whether the state space from the current state onwards is deadlock-free is more complicated. The *calls_deadlock_free* keeps track of whether any recursive call reports a deadlock (line 12). The current state is not a deadlock state if either there is some thread with an enabled transition, or every thread had terminated its computation and is not waiting for some change in the shared state. If any transition is enabled, then *is_enabled_thread* is set to true (line 9). If any thread without an enabled transition is waiting for some change in the shared state (for instance a lock to become available), then *no_waiting_threads* is set to false. These results are combined in line 17.

B.2 Stateful Dynamic Partial-Order Reduction

Adapted from Yang et al. [37, Figure 5].

```
1:  let H = an empty hash table of states in
2:  let G = an empty graph on transitions in
3:
4:  procedure EXPLORE(π)
5:      let σ = last(π) in
6:      if σ ∈ H then
7:          let U = {u | ∃t ∈ enabled(σ). u is reachable from node t in G} in
8:          for all t ∈ U do UPDATEBACKTRACKSETS(π, t)
9:      else
10:         add σ into H;
11:         for all p ∈ P do UPDATEBACKTRACKSETS(π, next(σ, p));
12:         if enabled(σ) ≠ ∅ then
13:             let t = any t ∈ enabled(σ) in
14:             let backtrack(σ) = {t} in
15:             let done(σ) = ∅ in
16:             while done(σ) ≠ backtrack(σ) do
17:                 let t = any t ∈ (backtrack(σ) \ done(σ)) in
18:                 add t to done(σ);
19:                 if |π| > 0 then add edge (π_{|π|−1}, t) to G;
20:                 EXPLORE(π.t)
21:
22: procedure UPDATEBACKTRACKSETS(π, t_{p,s})
23:     let D = {i ∈ dom(π) | π_i is dependent with t_{p,s} and i ↛_π p} in
24:     if D ≠ ∅ then
25:         let σ_d = pre(π, max(D)) in
26:         if next(σ_d, p) ∈ enabled(σ_d) then add next(σ_d, p) to backtrack(σ_d)
27:         else add all of enabled(σ_d) to backtrack(σ_d)
28:
29: Initially: EXPLORE(∅)
```

B.3 Static Partial-Order Reduction

```
1: procedure PERSISTENTSET(σ)
2:     let E = {p}, for any p such that ∃t_{p,s} ∈ enabled(σ) in
3:     let O = {o | accesses(p, o)} in
4:     while E changes do
5:         for all q ∈ (P \ E) do
6:             if ∃o ∈ O. accesses(q, o) then
7:                 E := E ∪ {p};
8:                 O := O ∪ {o | accesses(q, o)}
9:     let T = {t | ∃p ∈ E. t = next(σ, p)} in
10:    if (T \ enabled(σ)) ≠ ∅ then return enabled(σ)
11:    else return T
```

B.4 Sleep Sets

This pseudocode is adapted from pseudocode given by Godefroid [15], the main difference being that Godefroid uses iteration and an explicit stack, rather than recursion.

```
1: let H = an empty hash table mapping states to sets of transitions in
2:
3: procedure CHECK(σ, Sleep)
4:     if σ ∉ H then
5:         add (σ, Sleep) to H;
6:         T := enabled(σ) \ Sleep
7:     else
8:         T := H(σ) \ Sleep;
9:         Sleep := Sleep ∩ H(σ);
10:        H(σ) := Sleep
11:    for all t ∈ T do
12:        let σ' = σ' such that σ -t-> σ' in
13:        let Sleep' = {t' ∈ Sleep | t' is independent with t} in
14:        CHECK(σ', Sleep');
15:        Sleep := Sleep ∪ {t}
16:
17: Initially: CHECK(σ_0, ∅)
```

Appendix C

Project Proposal

Introduction

Model checking is category of formal methods used to show that systems (such as hardware components and software modules) fit their specification. Unlike standard testing or simulation, model checking ensures that the given system will *always* behave as expected. This is achieved by constructing an abstract model of the system, and then automatically showing that this model has properties which are equivalent to the system specification.

However, especially when verifying concurrent software, model checking suffers from the *state explosion problem*—the size of the state space of the model grows exponentially with the number of components in the system. There have been several attempts to address this problem, but this project will explore one in particular: dynamic partial-order reduction.

Partial-order reduction attempts to reduce the size of the state space explored by considering only a subset of the possible interleavings of events, such that any interleaving not explored is, as far as the specification is concerned, equivalent to an interleaving that is explored. However, it can be difficult to work out which sets of interleavings are equivalent, so statically computed approximations of such sets are necessarily conservative. In some cases, this can lead to only a modest reduction in the size of the state space.

Dynamic partial-order reduction completes one full execution of the program, picking an arbitrary interleaving of events, while dynamically making note of backtracking points at which an alternative state transition leads to a path potentially not “equivalent" to that already traversed. These paths are explored until all backtracking points are exhausted; at this point, the program is shown to have met its specification as if the entire state space had been explored. The extra information gained from doing the computation dynamically means that often a significant reduction in the size of the state space explored can be achieved. The aim of this project is to build a system which carries out dynamic partial-order reduction model checking.

Starting Point

The project will be written from scratch, although all the fundamental theory and algorithms will be taken from published work.

Substance and Structure of the Project

The project breaks down into the following three sub-tasks:

1. Design and implement a simple language for the writing of concurrent software programs and the specification of simple safety constraints.
2. Understand and implement the core algorithm which identifies the necessary backtracking points.
3. Implement an algorithm which backtracks through the possible executions of the given program, using the above procedure to explore the interleavings of events which are necessary for soundness of verification.

Main Success Criterion

To demonstrate, through a few well-chosen simple examples, that the project successfully performs dynamic partial-order reduction model checking. Quantitative comparison should be made either with a model checking algorithm that does not use partial-order reduction to show the improvement in performance, or ideally with an existing implementation of dynamic partial-order reduction model checking to show that the performance is similar.

Possible Extensions

- Implement a static partial-order reduction model checker and compare.
- If the system does not meet its specification, provide a trace of a counterexample execution.
- Understand and implement various ameliorations (for example using a stateful backtracking algorithm, the “stack traversal" refinement or “sleep sets"), and evaluate performance improvement.
- Design and build a tool to visualise the execution of the backtracking state space exploration, perhaps even making a visual comparison between dynamic and static partial-order reduction on the same problem.

Plan of Work

Michaelmas Weeks 3–6

Study and understand relevant literature (especially the paper introducing dynamic partial- order reduction). This is a significant part of this relatively theory-heavy project. Specify syntax and semantics of a simple language for writing concurrent software programs.

Michaelmas Weeks 7–8

Implement, test and debug the language.

Christmas Vacation

Design, write, test and debug the main structure of the system, in particular the backtracking algorithm. The procedure that computes which backtracking points are necessary is to be approximated by a function which suggests that every possible interleaving should be examined.

Lent Weeks 1–3

Implement, test and debug algorithm that decides which backtracking points are necessary. Write and submit the required progress report.

Lent Weeks 4–5

Evaluate project to show that the main success criterion has been met.

Lent Weeks 6–8

If behind, catch up with timetable; otherwise, attempt extension tasks. Begin writing first draft of dissertation.

Easter Vacation

Continue work on dissertation. Deliver full draft to supervisors and director of studies. Proofread and submit dissertation before beginning of Easter Term.

Resources Required

For this project, I shall mainly use my own computer, an Acer laptop which runs Windows 10, has a 1.8 GHz dual-core processor, and 6 GB of memory. In the event of its failure, I will use the university's MCS machines. To mitigate the risk of loss of data, I will work in a Dropbox folder, so that my files are automatically backed up whenever my computer is connected to the Internet. I will also make at least weekly backups to both OneDrive and my university filespace.