PROBABILISTIC SYMMETRY REDUCTION

CHRISTOPHER POWER

SUBMITTED IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

SCHOOL OF COMPUTING SCIENCE

COLLEGE OF SCIENCE AND ENGINEERING
UNIVERSITY OF GLASGOW

APRIL 2012

Abstract

Model checking is a technique used for the formal verification of concurrent systems. A major hindrance to model checking is the so-called state space explosion problem where the number of states in a model grows exponentially as variables are added. This means even trivial systems can require millions of states to define and are often too large to feasibly verify. Fortunately, models often exhibit underlying replication which can be exploited to aid in verification. Exploiting this replication is known as symmetry reduction and has yielded considerable success in non probabilistic verification.

The main contribution of this thesis is to show how symmetry reduction techniques can be applied to explicit state probabilistic model checking. In probabilistic model checking the need for such techniques is particularly acute since it requires not only an exhaustive state-space exploration, but also a numerical solution phase to compute probabilities or other quantitative values.

The approach we take enables the automated detection of arbitrary data and component symmetries from a probabilistic specification. We define new techniques to exploit the identified symmetry and provide efficient generation of the quotient model. We prove the correctness of our approach, and demonstrate its viability by implementing a tool to apply symmetry reduction to an explicit state model checker.

Acknowledgments

I would like to thank my supervisor, Alice Miller, for her time, support and advice. I would also like to thank my Ph.D. examiners, David Parker and John O'Donnell. Their suggestions greatly enhanced my dissertation. Finally, thank you to everybody who supported me over the last four years.

Contents

1 Introduction **1**

2 Model Checking **4**

2.1 Introduction 4

2.2 Types of Models 6

2.2.1 Kripke Structures 6

2.2.2 Discrete Time Markov Chains (DTMC) 7

2.2.3 Markov Decision Processes (MDP) 8

2.3 Temporal Logic 10

2.3.1 Linear Time Temporal Logic (LTL) 10

2.3.2 Computation Tree Logic (CTL) 12

2.3.3 CTL* 13

2.3.4 Probabilistic Computational Tree Logic (PCTL) 15

2.4 Storage Schemes 16

2.4.1 Explicit State Model Checking 16

2.4.2 Symbolic Model Checking 17

2.4.3 Probabilistic Representations 21

2.5 Model Checking Algorithms 23

2.5.1 CTL Model Checking 23

2.5.2 LTL Model Checking 24

2.5.3 PCTL Model Checking 25

2.6 Model Checking Tools 29

2.6.1 State Space Explosion 31
2.7 Summary 32

3 Symmetry Reduction **33**
3.1 Group Theory 34
3.1.1 Groups, Subgroups and Homomorphisms 34
3.1.2 Permutation Group 35
3.1.3 Group Actions 37
3.2 Symmetry in Model Checking 37
3.2.1 Symmetry in Kripke Structures 37
3.2.2 Symmetry in Discrete Time Markov Chains 38
3.2.3 Symmetry in Markov Decision Process 39
3.3 Symmetry Reduction in Practice 39
3.3.1 Identifying Symmetry 40
3.3.2 User Specification of Symmetry 41
3.3.3 The Scalarset Approach 41
3.3.4 Automated Symmetry Detection 42
3.4 Exploiting Symmetry 43
3.4.1 Easy Classes of Symmetry 44
3.4.2 Multiple Representatives Approach 44
3.4.3 Strategies for Symmetry Reduction 45
3.5 Combining Symmetry Reduction with Symbolic Representation 46
3.6 Exploiting Symmetry in Less Symmetric Systems 47
3.7 Tools for Symmetry Reduction 48
3.7.1 Symmetry Reduction for Probabilistic Model Checking 49
3.8 Summary 50

4 Probabilistic Symmetric Systems Language **51**
4.1 Informal Introduction to PSS 52
4.1.1 Six Sided Die Example 53
4.1.2 Simple Mutual Example 54
4.1.3 A Peer to Peer Network Example 56

4.2 Formal Definition of PSS 60
4.2.1 Variable Declarations 62
4.2.2 Language Definition 63
4.2.3 Definition of Atomic Propositions 64
4.2.4 States of a Model Associated with a PSS Specification 64
4.2.5 Expression Evaluation 65
4.2.6 Guard Evaluation 66
4.2.7 Effect of Updates 67
4.3 Constructing the Discrete Time Markov Chain 67
4.4 Construction of the Markov Decision Process 69
4.5 Summary 70

5 Automated Symmetry Detection **71**

5.1 Automated Detection 72
5.1.1 Channel Diagram Associated with a PSS Specification 72
5.1.2 Examples of Channel Diagrams Associated with a PSS Specification 73
5.1.3 Channel Diagrams and Data Symmetries 77
5.1.4 Extended Channel Diagram Associated with a PSS Specification $\mathcal{P}$ 78
5.1.5 Examples of Extended Channel Diagrams Associated with a PSS Specification 79
5.1.6 Comparison of Channel Diagrams and Extended Channel Diagrams 84
5.1.7 Deriving an Extended Static Channel Diagram 85
5.2 Correspondence Proof 85
5.2.1 Action of Aut(ECD($\mathcal{P}$)) on $\mathcal{P}$ 86
5.2.2 Action of Aut(ECD($\mathcal{P}$)) on $\mathcal{D}$ and $\mathcal{M}$ 87
5.3 Largest Valid Symmetry Group 91
5.3.1 Reconstructing the Largest Valid Symmetry Group 92
5.3.2 Algorithm to Reconstruct the Largest Valid Symmetry Group . . . 93
5.4 Summary 95

6 Computing a Representative State **96**

6.1 A Model of Computation Without References 97
6.2 Full Enumeration 98

6.2.1 Applying a Permutation to a State . . . 99
6.3 Full Enumeration as a Constraint Satisfaction Problem . . . 104
6.3.1 A Simple Mapping . . . 104
6.3.2 Adding Further Constraints . . . 105
6.4 A Representative Function for Large Groups . . . 108
6.4.1 Local Search for Large Groups . . . 108
6.5 Exploiting The Structure of a Group . . . 110
6.5.1 Fully Symmetric Group . . . 111
6.5.2 Exploiting Group Homomorphisms . . . 111
6.5.3 Cyclic Group . . . 114
6.5.4 Direct Product . . . 117
6.5.5 Semi Direct Product . . . 120
6.5.6 Summary . . . 122

7 Constructing the Probabilistic Model 123
7.1 Algorithm to Construct a Quotient Probabilistic Model . . . 124
7.1.1 Implementation of Data Structures . . . 126
7.2 PCTL Model Checking . . . 129
7.2.1 Determining if a PCTL Formula is Invariant Under a Group G . . . 130
7.2.2 Largest Valid Subgroup . . . 131
7.3 Summary . . . 132

8 Results and Comparison 133
8.1 An Overview of the PSS Symmetry Reduction Tool . . . 134
8.2 An Overview of Automated Symmetry Detection . . . 135
8.2.1 Computing Graph Automorphisms . . . 136
8.2.2 Checking the Validity of an Element . . . 136
8.2.3 Calculating the Largest Valid Symmetry . . . 136
8.2.4 Experimental Results . . . 137
8.3 Computing State Representatives Experiments . . . 140
8.3.1 Application of an Element . . . 141
8.3.2 Enumeration . . . 141
8.3.3 Local Search . . . 143

8.4 Probabilistic Model Checking 145
8.5 Comparison with PRISM, PRISM-symm and GRIP 147
8.6 Summary 150

9 Conclusion and Future work **151**
9.1 Future Work 152

A Skeleton Code **154**
A.1 Simple Mutual Exclusion 154
A.2 Dining Philosophers 155
A.3 Network Infection 156
A.4 Monty Hall Problem 157
A.5 Resource Allocator 158
A.6 Three tiered architecture 159

Bibliography **159**

CHAPTER 1

Introduction

Model checking is a technique used in the formal verification of concurrent systems. To verify the correctness of a system, a model of the system is generated that contains all possible behaviours. This set of system behaviours can then be checked against a set of properties to ascertain if the system behaves as expected. Example properties include: “process 1 and process 2 are never in their critical sections simultaneously” or “it is always possible to restart the system”.

As the model checking process has became increasingly sophisticated, the range of systems that can be described and verified has increased. One prevalent example is the description of probabilistic systems. By extending a specification language to allow for transitions between states to be labelled with the likelihood that they will occur, properties such as: “the message will be delivered with probability 0.6” or “the probability of shutdown occurring is at most 0.02” can be verified. Probabilistic model checking has the advantage of providing efficient and rigorous methods for evaluating a wide range of quantitative properties.

A major hindrance to model checking is the so-called state space explosion problem. Holzmann [58] explains that although verification algorithms have a linear run time complexity, this is offset as the number of states in a model grows exponentially as variables are added. This means that even trivial real-life systems can require millions of states to define their behaviour.

The reason that non-probabilistic model checking has been so successful in the real world is that an enormous amount of work has been put into developing efficient implementation techniques. These include state compression [57], partial order reduction [52, 85], symmetry reduction [20] and symbolic storage, where states and transitions of a model are represented symbolically (as opposed to explicitly) in order to save space [16]. These techniques have allowed the verification of ever increasing complex systems and greatly enhanced the uptake of model checking.

In the probabilistic domain, alleviating the state space explosion problem is an active research area. Techniques including symbolic storage [6, 67], partial order reduction [7, 28] and bisimulation minimisation [63] have been investigated. Furthermore, some steps have been made in examining the application of probabilistic symmetry reduction to symbolically stored state-spaces [68, 31].

Symmetry reduction is a technique concerned with exploiting underlying regularities in the state space by only storing one representative of a class of states. If symmetry is known to be present in a model then model checking of certain properties can be performed over a quotient state-space. Importantly, symmetry reduction is implemented differently in explicit-state and symbolic model checkers, each with their own research challenges.

In explicitly represented systems the exploitation of symmetry can be highly profitable in terms of time and space. Recent work has focused on providing "push button" or automatic symmetry detection [30] and reduction in addition to widening the set of systems to which symmetry can be applied [94]. To our knowledge little work has been conducted on the application of symmetry reduction to probabilistic explicit state model checking. Therefore, we propose to investigate the application of symmetry reduction in probabilistic explicit state model checking. To present the results of this investigation the rest of the thesis is structured as follows:

We provide a review of model checking and symmetry reduction literature in Chapters 2 and 3 respectively and highlighted that no research has been conducted on the application of symmetry reduction to explicit state probabilistic model checking.

Our contribution begins in Chapter 4 where we formally defined a new probabilistic model specification language. The presented specification language is capable of exhibiting complex symmetry groups while being simple enough to allow the correctness of our detection and reduction techniques to be proved. We make use of the language in the remainder of the thesis to aid in the presentation of our results.

In Chapter 5 we introduced the extended channel diagram approach, which is the first technique we know of that can detect arbitrary component and data symmetries directly from

a probabilistic specification. The approach involves computing the symmetry group of the specification and using the presented techniques determines a subgroup of these symmetries which induces automorphisms of the underlying model that are valid for symmetry reduction.

In Chapter 6 we present a selection of new techniques to efficiently compute equivalence class representatives for certain classes of symmetry groups. We present enhancements that improve the average runtime of exhaustive search and where enumeration is infeasible, we considers a tailored made local search algorithm. For symmetry groups possessing identifiable structural properties we provide efficient techniques that do not require the exhaustive application of all elements in the symmetry group . We suggest techniques for the fully symmetric group, cyclic groups and groups that can be decomposed as an internal direct product or as an internal semi direct product.

In Chapter 7 we consider how to combine our presented techniques to construct a smaller quotient model directly from a probabilistic specification. Finally, in Chapter 8 we describe our model checker which implements the presented specification language, detection and reduction techniques. The model checker is used to test the viability of applying our approach of automated symmetry reduction to explicit state probabilistic model checking. For a variety of symmetric specifications we show significant runtime and memory savings can be made while performing probabilistic model checking.

CHAPTER 2

Model Checking

In this chapter we present established approaches to model checking. Section 2.1 introduces the notion of formal verification and defines the model checking process. In Section 2.2 we cover some common types of models and in Section 2.3 we cover a selection of temporal logics. Issues concerning underlying data structures and key algorithms are discussed in Section 2.4 and Section 2.5 respectively. The chapter closes with a review of relevant currently available model checkers and a discussion on the state space explosion problem.

2.1 Introduction

Computerised systems have become an integral part of our lives and technological progression has led to the scenario where these systems directly control safety critical applications. Systems of this type include aeroplane landing systems [80], nuclear reaction management [83] and medical systems [76]. In instances where human life is placed at direct risk, it is essential that controlling software functions correctly. However, even after decades of research, the best of traditional software development methodologies cannot guarantee a bug free system [60]. Furthermore, these methodologies cannot provide enough confidence to assert if a system correctly implements its requirements specification [12].

In software and hardware design of complex or safety critical systems, the majority of time

and cost is spent on the verification phase. This motivates research into techniques that ease verification, increasing both coverage and confidence. Formal methods is a technique that offers these desired attributes. Subtle errors that remain hidden after emulation, testing and simulation can potentially be revealed using formal methods.

One aspect of formal methods is system verification. System verification is the process of establishing whether a design or system satisfies certain properties. A bug occurs when the system does not satisfy one of the stated properties and the system is considered correct when all properties are satisfied. An overview of the verification process is depicted in Figure 2.1.

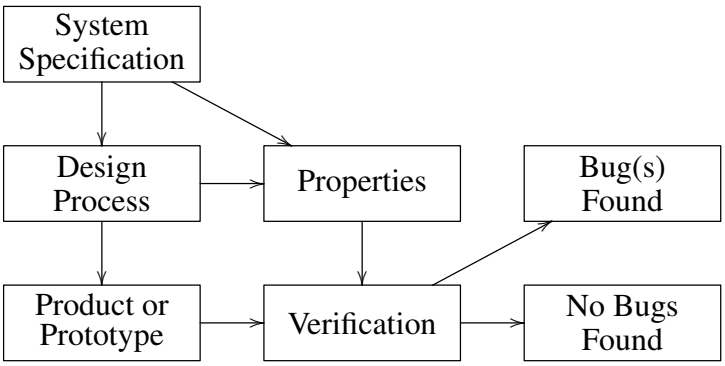


Figure 2.1: Schematic of system verification as presented in [9].

Academic research has proposed and developed several approaches to formal verification [24]. Two major classifications are theorem proving [49] and model checking [23]. In theorem proving the system specification and properties are described in terms of mathematical statements. Verification is conducted by proving properties based on the system specification. The proof is required to show how statements of the theorem are formally derived from axioms using inference rules. A powerful benefit of theorem proving is its ability to deal with systems represented by an infinite state space. The alternative formal verification methodology, model checking, is a completely automatic process that usually deals with finite state spaces.

A model checker [25] accepts two inputs, a model specification $\mathcal{P}$, described in a high level formalism, and a set of testable properties, ϕ. The model checker generates and exhaustively searches a finite state model $\mathcal{M}(\mathcal{P})$ to confirm if a property holds, or alternatively, reports it was in violation of the system specification. When a violation occurs, it is common for model checkers to generate a counter-example illustrating precisely why the property was invalid, Figure 2.2.

Bugs identified in the model of the system will hopefully reveal bugs in the system design. However, care must be taken to ensure properties correctly define the desired behaviour [60]. This model checking process is repeated with a refined specification or property until ϕ holds in all initial states of $\mathcal{M}(\mathcal{P})$. When this occurs it is said that $\mathcal{M}(\mathcal{P})$ satisfies ϕ, written

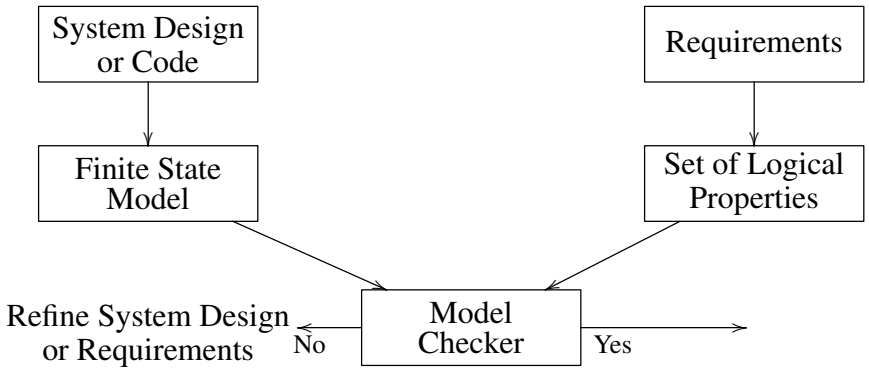


Figure 2.2: The model checking process.

$\mathcal{M}(\mathcal{P}) \models \phi$. Therefore the model checking process can be stated; given a model specification $\mathcal{P}$ and a property ϕ, does $\mathcal{M}(\mathcal{P}) \models \phi$?

2.2 Types of Models

A model $\mathcal{M}(\mathcal{P})$ describes in mathematical terms the behaviour of a system. A general set of mathematical structures, based on directed graphs, are used to describe the possible behaviours of a system specification. In the graph, nodes contain information about the system at an instant in time. These nodes are the states of the system, and transitions specify how the system can evolve from one state to another. The accumulation of all possible states and transitions is called the state space. In Section 2.2.1 we define a Kripke structure [15] as a method of describing these systems.

An additional category of model checker is one that enables the specification and verification of systems which exhibit random or probabilistic behaviour. This is achieved by labelling transitions between states with information about the likelihood that they will occur. A mathematical structure capable of describing the evolution of this category of system is a Markov Chain [65]. There are several Markov Chain variants, however we will only consider, Discrete Time Markov Chains (DTMC) (Section 2.2.2) and Markov Decision Processes (MDP) (Section 2.2.3) which extend the DTMC allowing for the specification of non-deterministic behaviour.

2.2.1 Kripke Structures

A Kripke structure [15] is commonly used to describe a finite state model. Let V = $\{v_1, v_2, \ldots, v_k\}$ be a finite set of variables, where v_i ranges over a finite non-empty set of possible values D_i. Then $D = D_1 \times D_2 \ldots \times D_k$ is the set of all possible system states.

Definition. A Kripke structure $\mathcal{M}$ over D is a tuple $\mathcal{M}$ = (S, S_0, R) where:

- S = D is a non-empty finite set of states.
- $S_0 \subseteq S$ is a set of initial states.
- $R \subseteq S \times S$ is a transition relation.

A path in $\mathcal{M}$, commencing from $s \in S$ is an infinite sequence of states $\pi = s_0, s_1, s_2, \ldots$ where $s_0 = s$ and for all $i > 0$, $(s_{i-1}, s_i) \in R$. It is common notation for a transition between two states s and s', to be written as $s \rightarrow s'$. Therefore, a state $s \in S$ is reachable if there is a path $s_0, s_1, \ldots, s, \ldots$ in $\mathcal{M}$ where $s_0 \in S_0$, and a transition $s \rightarrow s' \in R$ is reachable if s is a reachable state. Commonly, Kripke structures have a single initial state $s_0 \in S$ and in this instance the Kripke structure is $\mathcal{M}$ = (S, s_0, R)

To illustrate, consider the Kripke structure for the two process mutual exclusion specification depicted in Figure 2.3. Each process has three local states N, T and C and a single state variable, st_i for $i \in \{1, 2\}$. Therefore, V = $\{st_1, st_2\}$ and $D_1 = D_2 = \{N, T, C\}$. The values N, T and C denote that a process is in a neutral, trying or critical state and the behaviour of the model can be defined as follows. If process i is in the trying state and process j is not in the critical state, process i may move into the critical state, it is not possible for both processes to be simultaneously in the critical state and if a process requests access to a critical section, it will eventually be granted.

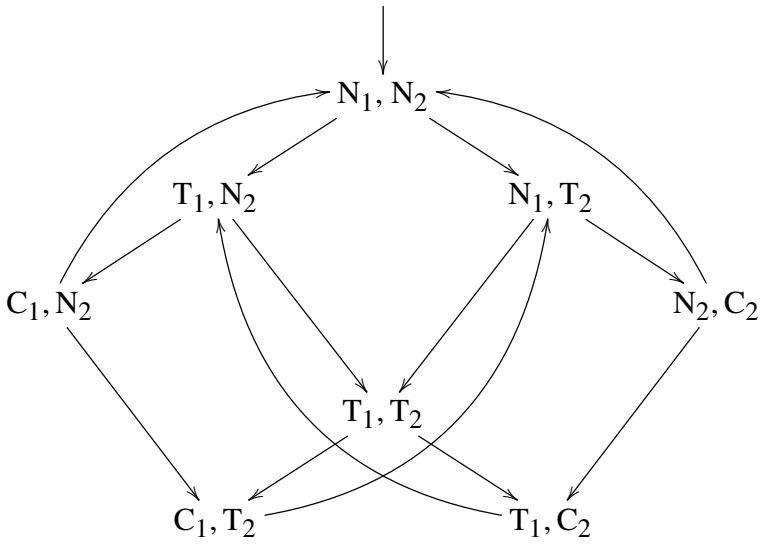


Figure 2.3: Kripke structure for the two process mutual exclusion specification.

2.2.2 Discrete Time Markov Chains (DTMC)

The simplest probabilistic models we consider are DTMCs [9]. This can be viewed as a state transition system with the probability of making a transition from one state to another

appended. As before, $V = \{v_1, v_2, \ldots, v_k\}$ is the finite set of variables each ranging over a domain D_i and $D = D_1 \times D_2 \ldots \times D_k$ is the set of all possible system states.

Definition. A Discrete Time Markov Chain $\mathcal{D}$ over D is a tuple $\mathcal{M} = (S, i_{init}, \mathbf{P})$ where:

- $S = D$ is a non-empty finite set of states.
- $i_{init} : S \to [0, 1]$ is an initial distribution, such that $\sum_{s \in S} i_{init}(s) = 1$.
- $\mathbf{P} : S \times S \to [0, 1]$ is a transition probability matrix such that for all states $s \in S$

$$\sum_{s' \in S} \mathbf{P}(s, s') = 1.$$

An entry in transition probability matrix $\mathbf{P}(s, s')$ determines the probability of moving between state s and state s'. The states s' for which $\mathbf{P}(s, s') > 0$ are possible successors to s. Therefore, $\sum_{s' \in S} \mathbf{P}(s, s') = 1$ for all states $s \in S$. To meet the requirement, $\sum_{s' \in S} \mathbf{P}(s, s') = 1$ for all states $s \in S$, all terminating states are appended with a transition to themselves with probability 1. The value $i_{init}(s)$ designates the probability of the system beginning in state s. States s, with $i_{init}(s) > 0$ are the initial states of the system. Commonly DTMCs have a single initial state $s_0 \in S$ and in this instance the DTMC is defined $\mathcal{D} = (S, s_0, \mathbf{P})$

A path π in $\mathcal{D}$ starting from s_0 is a non-empty sequence of states $s_0, s_1, s_2, \ldots$ where $s_i \in S$ and $\mathbf{P}(s_i, s_{i+1}) > 0$ for all $i \geq 0$. The execution path π can be finite, π_{fin}, or infinite, π_{inf}, with the i^{th} state being denoted $\pi(i)$. The notation Path(s) is the set of all path fragments initiating from the state s. Possible evolutions of the system are represented by paths. Therefore, to reason about the behaviour of the system, the probability that a path is taken must be calculated. For each state $s \in S$ a probability measure $Prob_s$ on Path(s) is defined. To facilitate this $\mathbf{P}(\pi_{fin}) = 1$ where $\pi_{fin} = s_0$ and $\mathbf{P}(\pi_{fin}) = \mathbf{P}(s_0, s_1) \cdot \mathbf{P}(s_1, s_2) \cdot \cdots \mathbf{P}(s_{n-1}, s_n)$ where $\pi_{fin} = s_0, s_1, \ldots, s_n$.

The cylinder set $C(\pi_{fin})$ is the set of all paths with prefix π_{fin} and $\sum_s$ is the smallest σ-algebra [1] on Path(s) containing all the sets $C(\pi_{fin})$. The probability measure $Prob_s$ on $\sum_s$ is the unique measure such that $Prob_s(C(\pi_{fin})) = \mathbf{P}(\pi_{fin})$. The probability of a system exhibiting a specific behaviour can be calculated by identifying the set of paths satisfying the condition and imposing upon them the measure $Prob_s$.

2.2.3 Markov Decision Processes (MDP)

MDPs [9] provide a mathematical framework for modelling decisions in situations where outcomes are partly random and partly under the control of the decision maker. An MDP is

[1] A π-algebra is a pair $(Outc, \varepsilon)$ where Outc is a nonempty set and $\varepsilon \subset 2^{Outc}$ a set consisting of subsets of Outc that contains the empty set and is closed under complementation and countable unions.

useful in the context of model checking as it allows the description of a number of probabilistic systems operating in parallel. Furthermore, non-deterministic transitions can be specified when the exact probability distribution is unknown, or irrelevant. , let V and D follow the same definition as given for Kripke structures and DTMCs.

Definition. A Markov Decision Process $\mathcal{M}$ over D is a tuple $\mathcal{M}$ = (S, i_{init}, Steps) where:

- S = D is a non-empty finite set of states.
- $i_{init} : S \to [0, 1]$ is an initial distribution, such that $\sum_{s \in S} i_{init}(s) = 1$.
- Steps : $S \to 2^{Dist(s)}$ is a transition function.

The definition is similar to that of a DTMC but transition probability matrix **P** is replaced by Steps. For a state $s \in S$, Steps(s) is the set of non-deterministic choices available in s. Specifically, Steps(s) is a function mapping states onto a finite, non-empty subset of probability distributions and takes the form $\mu : S \to [0, 1]$ where $\sum_{s \in S} \mu(s) = 1$.

An execution path through an MDP is a non-empty sequence $s_0 \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_2} \dots$ where $s_i \in S$, $\mu_{i+1} \in$ Steps(s_i) and $\mu_{i+1}(s_{i+1}) > 0$ for all $i \geq 0$. In keeping with the notation presented for a DTMC, π_{fin} is a finite execution path, π_{inf} an infinite path and the i^{th} state of a path denoted $\pi(i)$. The notation Path(s) is the set of all path fragments initiating from the state s.

An execution path of an MDP requires both non-deterministic and probabilistic transitions to be resolved. Non-deterministic choices are made by an adversary where the decision is determined by the choices made in all previous runs. An adversary A can be defined formally as a function that maps every finite path π_{fin} in an MDP onto a distribution $A(\pi_{fin}) \in$ Steps(last(π_{fin})) where last(π_{fin}) is the final state of the finite path. For convenience the subset of Path(s) which corresponds to adversary A is denoted $Path^A(s)$.

The behaviour of the MDP for an adversary A is probabilistic in nature. It can therefore be defined by a DTMC where the probability of non-deterministic transitions are given by a probability distribution selected by A. Drawing conclusions about an MDP's behaviour for a given adversary is meaningless. Meaningful results, in the form of maximum and minimum probabilities can be ascertained by computing over all possible adversaries. When computing the maximum or minimum probability for an observed event it is often beneficial to use adversaries with a notion of fairness. For example, in a non-deterministic scheduler it is impossible to draw meaningful conclusions if every process does not eventually get a chance to proceed. A path π is fair, if for states s occurring infinitely often in π each choice $\mu \in$ Steps(s) is taken infinitely often. Additionally, an adversary A is fair if $Prob_s^A(\{\pi \in Path^A(s) \mid \pi \text{ is fair}\}) = 1$ for all $s \in S$.

2.3 Temporal Logic

It is common for the set of testable properties ϕ to be expressed using a temporal logic. Temporal logics provide a formal language to reason about the behavioural properties of parallel programs and more generally reactive systems. Their application in Computing Science was pioneered by Pnueli [87] who argued that existing techniques for verification were not adequate for concurrent, reactive systems. Temporal logics alleviated this issue by allowing the user to reason about related events at different moments in a system's execution. Temporal logic formalisms may be classified according to their particular view of time; Linear Time where for every given path every state has a unique successor, and Branching Time, where every state has several successors, Figure 2.4.

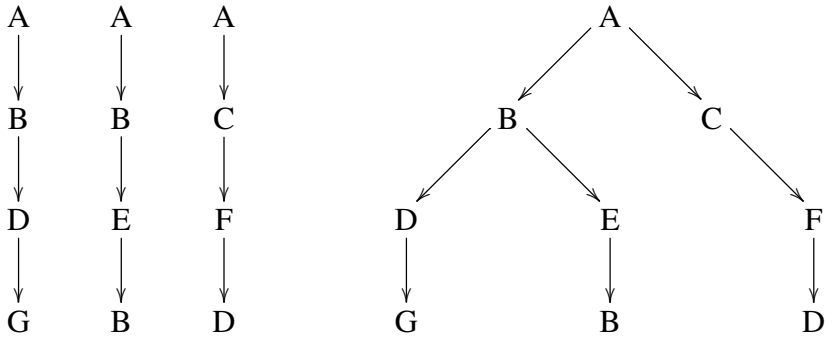


Figure 2.4: Comparative views of time in temporal logics.

2.3.1 Linear Time Temporal Logic (LTL)

Linear Time Temporal Logic [87] allows the future behaviour of a system to be reasoned about. A collection of paths represent the set of possible future behaviours, any one of which might be the actual outcome of the systems execution. To aid in behavioural descriptions a set of atoms, which state facts that may hold in a system are used. The choice of atoms is dependent on the system being described but general examples include "Queue 4 is full" and "Process 3 is active".

Linear Time Temporal Logic, where ϕ is the formula and p is any propositional atom from a set of atoms, has the following syntax in Backus Naur Form:

$$\phi := \top \mid \bot \mid \mathrm{p} \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \mathrm{X}\,\phi \mid \mathrm{F}\,\phi \mid \mathrm{G}\,\phi \mid \phi\,\mathrm{U}\,\phi \mid \phi\,\mathrm{W}\,\phi \mid \phi\,\mathrm{R}\,\phi$$

The symbols X, F, G, U, R, and W are temporal connectives where X reasons about the next state, F some future state and G all future states. Symbols U, R and W are known as Until,

Release and Weak-until, respectively. Take a Kripke structure $\mathcal{M}$ and a path $\pi = s_0, s_1, s_2, \ldots$. Whether π satisfies a LTL formula is defined by the satisfaction relation $\models$. For the path π we define s_0 as the first state in the path and, for all $i \geq 0$, π_i is the suffix of π starting from state s_i. The relation $\models$ is defied inductively below;

- $\pi \models \top$.
- $\pi \not\models \bot$.
- $\pi \models p$ iff p is true in s_0.
- $\pi \models \neg\phi$ iff $\pi \not\models \phi$.
- $\pi \models \phi_1 \wedge \phi_2$ iff $\pi \models \phi_1$ and $\pi \models \phi_2$.
- $\pi \models \phi_1 \vee \phi_2$ iff $\pi \models \phi_1$ or $\pi \models \phi_2$.
- $\pi \models \phi_1 \rightarrow \phi_2$ iff $\pi \models \phi_2$ whenever $\pi \models \phi_1$.
- $\pi \models X\ \phi$ iff $\pi_1 \models \phi$.
- $\pi \models G\ \phi$ iff, for all $i \geq 1$ $\pi_i \models \phi$.
- $\pi \models F\ \phi$ iff, there is some $i \geq 1$ such that $\pi_i \models \phi$.
- $\pi \models \phi\ U\ \psi$ iff, there is some $i \geq 1$ such that $\pi_i \models \psi$ and for all $j = 1, \ldots, i-1$ we have $\pi_j \models \phi$.
- $\pi \models \phi\ W\ \psi$ iff, either there is some $i \geq 1$ such that $\pi_i \models \psi$ and for all $j = 1, \ldots, i-1$ we have $\pi_j \models \phi$, or for all $k \geq 1$ we have $\pi_k \models \phi$.
- $\pi \models \phi\ R\ \psi$ iff, either there is some $i \geq 1$ such that $\pi_i \models \phi$ and for all $j = 1, \ldots, i-1$ we have $\pi_j \models \psi$, and for all $k \geq 1$ we have $\pi_k \models \psi$.

Therefore, take a Kripke structure $\mathcal{M}$, $s \in S$ and a LTL formula ϕ. We write $\mathcal{M}$,s $\models \phi$. if, for every execution path π of $\mathcal{M}$ starting at s, we have $\pi \models \phi$.

Example LTL Properties

Referring to the Kripke structure for the mutual exclusion problem defined in Figure 2.3 an attempt to verify the correctness of the solution can be made by checking it against a set of LTL properties:

- **Safety**: The requirement that only one process is in the critical section at any time is expressible by the formula, $G\ \neg(C_1 \wedge C_2)$. This condition is satisfied in the initial state and all subsequent states. However, this alone does not prove the correctness of the specification. A protocol that prevented any process from ever entering a critical section would also satisfy the formula.
- **Liveness**: When a process requests entry to a critical section, the request will eventually be granted. The requirement is expressed as $G\ (T_1 \rightarrow F\ (C_1))$. However, a counter example can be generated to show this formula is not valid. Starting at the initial state

there is a path where T_1 becomes true but C_1 remains false, N_1, $N_2 \rightarrow T_1$, $N_2 \rightarrow T_1$, $T_2 \rightarrow T_1$, $C_2 \rightarrow T_1$, $N_2 \rightarrow \ldots$. This path arises as the state T_1, T_2, where both processes are in a trying state gives a choice of which process moves to the critical section. This problem can be alleviated by remodelling the system to allow the state T_1, T_2 to appear twice in the transition system. Each occurrence implicitly indicating which process will proceed to a critical state.

- **Non-blocking**: A process can always request entry to its critical section. For example, in process one we would like to attest that for every state satisfying N_1, there is a successor satisfying T_1. However, statements of this nature are not expressible in LTL as the logic does not contain an existence quantifier on paths.

2.3.2 Computation Tree Logic (CTL)

Computation Tree Logic [19], is a branching-time logic, meaning that its model of time is a tree-like structure in which the future is not determined. CTL provides the previously covered LTL temporal operators U, F, G and X in addition to new quantifiers A and E. These quantifiers express All paths, and Exists a path, respectively. This allows properties such as, "there is a reachable state satisfying q" or "from all reachable states satisfying p, it is possible to maintain p continuously until reaching a state satisfying q" to be expressed.

CTL, where ϕ is the state formula and p is any propositional atom from a set of Atoms, has the following syntax in Backnus Naur Form:

$$\phi := \top \mid \bot \mid \text{p} \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \text{AX}\ \phi \mid \text{EX}\ \phi \mid \text{AF}\ \phi \mid \text{EF}\ \phi \mid \text{AG}\ \phi \mid \text{EG}\ \phi \mid \text{A}\ [\phi\ \text{U}\ \phi] \mid \text{E}\ [\phi\ \text{U}\ \phi]$$

CTL temporal connectives are formed from a pair of symbols that are indivisible, the first being either an A or E and the second being one of the connectives X, F, G, or U. The symbols X, F, G and U cannot occur without being preceded by an A or an E and similarly, every A or E must be accompanied by X, F, G and U.

For a Kripke structure $\mathcal{M}$, a state s $\in$ S and a CTL formula ϕ, the relation $\mathcal{M}$, s $\models \phi$ can be defined by structural induction on ϕ in the following manner:

- $\mathcal{M}$,s $\models \top$.
- $\mathcal{M}$,s $\not\models \bot$.
- $\mathcal{M}$,s $\models$ p if p is true in s.
- $\mathcal{M}$,s $\models \neg\phi$ iff $\mathcal{M}$, s $\not\models \phi$.
- $\mathcal{M}$,s $\models \phi_1 \wedge \phi_2$ iff $\mathcal{M}$, s $\models \phi_1$ and $\mathcal{M}$, s $\models \phi_2$.

- $\mathcal{M}$,s $\models \phi_1 \vee \phi_2$ iff $\mathcal{M}$, s $\models \phi_1$ or $\mathcal{M}$, s $\models \phi_2$.
- $\mathcal{M}$,s $\models \phi_1 \rightarrow \phi_2$ iff $\mathcal{M}$,s $\not\models \phi_1$ or $\mathcal{M}$,s $\models \phi_2$.
- $\mathcal{M}$,s $\models$ AX ϕ iff for all s_1 such that s $\rightarrow s_1$ we have $\mathcal{M}$,$s_1 \models \phi$.
- $\mathcal{M}$,s $\models$ EX ϕ iff for some s_1 such that s $\rightarrow s_1$ we have $\mathcal{M}$,$s_1 \models \phi$.
- $\mathcal{M}$,s $\models$ AG ϕ iff for all paths $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow$ ... where s_1 equals s, and all s_i along the path we have $\mathcal{M}$,$s_i \models \phi$.
- $\mathcal{M}$,s $\models$ EG ϕ iff there is a path $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow$ ... where s_1 equals s, and for all s_i along the path we have $\mathcal{M}$,$s_i \models \phi$.
- $\mathcal{M}$,s $\models$ AF ϕ if for all paths $s_1 \rightarrow s_2 \rightarrow$ ... where s_1 equals s, there is some s_i, such that $\mathcal{M}$,$s_i \models \phi$.
- $\mathcal{M}$,s $\models$ EF ϕ iff there is a path $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow$ ... where s_1 equals s, and for all s_i along the path we have $\mathcal{M}$,$s_i \models \phi$.
- $\mathcal{M}$,s $\models$ A [ϕ_1 U ϕ_2] iff for all path $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow$ ... where s_1 equals s, that path satisfies ϕ_1 U ϕ_2.
- $\mathcal{M}$,s $\models$ E [ϕ_1 U ϕ_2] iff there is a path $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow$ ... where s_1 equals s, and that path satisfies ϕ_1 U ϕ_2.

Example CTL Properties

Referring to the Kripke structure for the mutual exclusion problem defined in Figure 2.2, we can attempt to check the correctness of the solution by verifying it against a set of CTL properties:

- **Safety**: The requirement that only one process is in the critical section at any time is expressed by the formula, AG ($\neg$($C_1 \wedge C_2$)). This condition is satisfied in the initial state and all subsequent states. As with LTL this formula does not cover a protocol that prevents all processes from ever entering the critical section.
- **Liveness**: Having reached its trying region a process will eventually progress to its critical section. For process one the formula AG ($T_1 \rightarrow$ AF (C_1))) would assert this property. A counter example to this property would be, starting at the initial state, a path where T_1 is true and C_1 is false, $N_1, N_2 \rightarrow T_1, N_2 \rightarrow C_1, N_2 \rightarrow N_1, N_2 \rightarrow \ldots$
- **Non-blocking**: A final consideration is that a process can always request entry to its critical section. For process one an appropriate formula would be AG ($N_1 \rightarrow$ EX (T_1)).

2.3.3 CTL*

The Computational tree logic [48] CTL*, combines the operators of both branching and linear time logics. The expressive power of LTL and CTL is combined, by removing the

constraint that temporal operators X, U, F and G have to be associated with path quantifiers A and E. In fact path quantifiers can be associated with any possible combination of linear operators. The syntax of CTL* involves two classes of formulae: state formulae, which are evaluated in states where p is any atomic formulae and α any path formula

$$\phi := \top \mid \mathrm{p} \mid \neg\phi \mid \phi \wedge \phi \mid \mathrm{A}[\alpha] \mid \mathrm{E}[\alpha]$$

and path formulae, which are evaluated along paths where ϕ is any state formula

$$\alpha := \phi \mid \neg\alpha \mid \alpha \wedge \alpha \mid \alpha \,\mathrm{U}\, \alpha \mid \mathrm{G}\,\alpha \mid \mathrm{F}\,\alpha \mid \mathrm{X}\,\alpha$$

CTL* is more expressive than LTL and CTL. It enables properties such as, "along all paths, either p is true until r, or q is true until r" (A [(p U e) $\vee$ (q U r)]) to be expressed. Furthermore, the logics of LTL and CTL are both subsets of CTL*. This may be surprising as LTL does not include the path operators A and E. However, in LTL we implicitly consider all paths for a given formula α and this is semantically equivalent to the CTL* formula A[α].

Expressive powers of LTL, CTL and CTL*.

Figure 2.5 shows the relationship between the expressive powers of LTL, CTL and CTL*. There is a considerable amount of literature comparing Linear-Time and Branching-Time logics [11, 70] and the question of which is better often arises. We have shown that the expressive powers of CTL* is greater than either of them. However, its implementation is computationally expensive and is therefore rarely used. The choice between LTL and CTL depends on the application and personal preference.

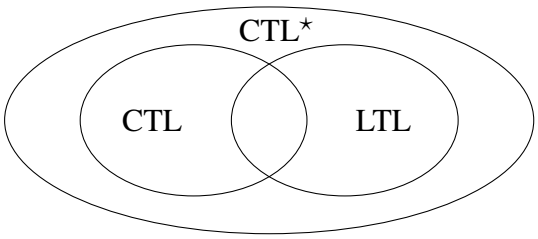


Figure 2.5: Relationship between the expressive powers of LTL, CTL and CTL*.

CTL allows explicit quantification over paths. However, it does not allow a specific selection of paths to be specified. For example, in LTL we can express that "all paths which have a p in some state also have a q" (F (p) $\rightarrow$ F (q)). It is not possible to write this in CTL because of the constraint that every F has an associated A or E. However, both languages can in some instances express the same formula. An example of this is the property that "any p is eventually followed by a q" expressible in CTL as AG (p $\rightarrow$ F (q)) and G (p $\rightarrow$ AF (q)) in LTL [61].

As mentioned previously, LTL formulae are evaluated on paths and properties that assert the existence of a path are not expressible. In terms of verification this problem can be alleviated

by considering the complement property [61]. However, a property that contains universal and existential path quantifiers in general cannot be expressed using this approach.

2.3.4 Probabilistic Computational Tree Logic (PCTL)

Probabilistic temporal logics are required to reason about probabilistic models. PCTL [53], is a probabilistic extension of CTL and therefore an example of Branching time logic. PCTL formulae can be interpreted over a DTMC or an MDP and incorporates timing information into properties. The syntax of PCTL where p is any propositional atom from a set of Atoms, prob in [0, 1], $k \in \mathbb{N}$ and $\bowtie$ is chosen from the set $\{\leq, <, >, \geq\}$ follows; [2]

$$\phi := \top \mid p \mid \neg\phi \mid \phi \wedge \phi \mid P_{\bowtie prob}\,[\psi]$$
$$\psi := X\,\phi \mid \phi\ U^{\leq k}\ \phi \mid \phi\ U\ \phi$$

In the syntax of PCTL a state s satisfies the probabilistic path formula $P_{\bowtie prob}\,[\psi]$ if the probability of leaving s via a path satisfying ψ is in the interval specified by $\bowtie$ prob. The path formulae X ϕ is true if ϕ is satisfied in the next state; ϕ_1 U ϕ_2 is true if ϕ_2 is satisfied at some point in the future and ϕ_1 is true until that point. Finally, $\phi_1\ U^{\leq k}\ \phi_2$ is true if ϕ_2 is satisfied within k time-steps and ϕ_1 is true until that point. The semantics of PCTL differs over DTMCs and MDPs. The relevant semantics will be defined in the proceeding sections.

PCTL over DTMCs

A property of a model is always expressed as a state formula. Therefore, for a DTMC $\mathcal{D}$, a state $s \in S$ and PCTL formula ϕ, we write $\mathcal{D}, s \models \phi$ or $s \models \phi$ to say that ϕ holds in s. The set of all states in which ϕ holds $\{s \in S \mid s \models \phi\}$ is denoted Sat(ϕ). Similarly, we write $\mathcal{D}, \pi \models \psi$ or $\pi \models \psi$ if path formula ψ holds for path π. Finally, for a path ϕ and a state $s \in S$, $P_s(\psi) = Prob_s(\{\pi \in Path(s) | \mathcal{D}, \pi \models \psi\})$. Formally, the semantics of PCTL over DTMCs are defined in the following manner:

- $\mathcal{D}, s \models \top$.
- $\mathcal{D}, s \models p$ if p is true in $\mathcal{D}$, s.
- $\mathcal{D}, s \models \neg\phi$ iff $\mathcal{D}, s \not\models \phi$.
- $\mathcal{D}, s \models \phi_1 \wedge \phi_2$ iff $\mathcal{D}, s \models \phi_1$ and $\mathcal{D}, s \models \phi_2$.
- $\mathcal{D}, s \models P_{\bowtie prob}\,[\psi]$ iff $P_s(\psi) \bowtie prob$.
- $\mathcal{D}, \pi \models X\,\phi$ iff $\mathcal{D}, \pi_1 \models \phi$.
- $\mathcal{D}, \pi \models \phi\ U^{\leq k}\ \psi$ iff for some $i \leq k$, $\mathcal{D}, \pi_i \models \psi$ and $\mathcal{D}, \pi_j \models \phi$ for all $0 \leq j < i$.
- $\mathcal{D}, \pi \models \phi\ U\ \psi$ iff for some $k \geq 0$, $\mathcal{D}, \pi \models \phi\ U^{\leq k}\ \psi$.

[2] The existential and universal quantification in CTL are replaced by a single probabilistic operator $P_{\bowtie\ prob}[\psi]$ and the additional operators covered in CTL can be defined using only the presented PCTL operators.

PCTL over MDPs

In the instance of MDPs the semantic definition of a path formula is identical to that of DTMCs. However, the probability of a set of paths differs as it can only be computed for a specific adversary. Given an MDP $\mathcal{M}$, the probability of a path from s satisfying path formula ψ under adversary A is denoted $\mathrm{P}_{\mathrm{s}}^{\mathrm{A}}(\psi) = \mathrm{Prob}_{\mathrm{s}}^{\mathrm{A}}(\{\pi \in \mathrm{Path}^{\mathrm{A}}(\mathrm{s}) | \mathcal{M}, \pi \models \psi\})$. To formally define the semantics of the PCTL formula $\mathrm{P}_{\bowtie \mathrm{prob}}[\psi]$ a set of adversaries Adv is selected and quantified over. It follows that the satisfaction relation is parameterised by Adv. Therefore, a state s satisfies the formula $\mathrm{P}_{\bowtie \mathrm{prob}}[\psi]$ if $\mathrm{P}_{\mathrm{s}}^{\mathrm{A}}(\psi) \bowtie$ prob for all adversaries $\mathrm{A} \in$ Adv. Formally the semantics of PCTL over MDPs are defined in the following manner:

- $\mathcal{M}, \mathrm{s} \models_{\mathrm{Adv}} \top$.
- $\mathcal{M}, \mathrm{s} \models_{\mathrm{Adv}}$ p if p is true in $\mathcal{M}$, s.
- $\mathcal{M}, \mathrm{s} \models_{\mathrm{Adv}} \neg\phi$ iff $\mathcal{M}, \mathrm{s} \not\models_{\mathrm{Adv}} \phi$.
- $\mathcal{M}, \mathrm{s} \models_{\mathrm{Adv}} \phi_1 \wedge \phi_2$ iff $\mathcal{M}, \mathrm{s} \models_{\mathrm{Adv}} \phi_1$ and $\mathcal{M}, \mathrm{s} \models_{\mathrm{Adv}} \phi_2$.
- $\mathcal{M}, \mathrm{s} \models \mathrm{P}_{\bowtie \mathrm{prob}}[\psi]$ iff $\mathrm{P}_{\mathrm{s}}^{\mathrm{A}}(\psi) \bowtie$ prob for all $\mathrm{A} \in$ Adv.
- $\mathcal{M}, \pi \models_{\mathrm{Adv}} \mathrm{X}\ \phi$ iff $\mathcal{M}, \pi_1 \models_{\mathrm{Adv}} \phi$.
- $\mathcal{M}, \pi \models_{\mathrm{Adv}} \phi\ \mathrm{U}^{\leq \mathrm{k}}\ \psi$ iff for some $\mathrm{i} \leq \mathrm{k}$, $\mathcal{M}, \pi_{\mathrm{i}} \models_{\mathrm{Adv}} \psi$ and $\mathcal{D}, \pi_{\mathrm{j}} \models_{\mathrm{Adv}} \phi$ for all $0 \leq \mathrm{j} < \mathrm{i}$.
- $\mathcal{M}, \pi \models_{\mathrm{Adv}} \phi\ \mathrm{U}\ \psi$ iff for some $\mathrm{k} \geq 0$, $\mathcal{M}, \pi \models_{\mathrm{Adv}} \phi\ \mathrm{U}^{\leq \mathrm{k}}\ \psi$.

2.4 Storage Schemes

We have discussed the mathematics underlying models in Section 2.2. However the question of how to generate and represent this underlying structure within computer memory remains. The two major encoding schemes employed in model checking are explicit and symbolic state representations.

2.4.1 Explicit State Model Checking

As the mathematical structures discussed in Section 2.2 are similar to a graph, it is sensible to use well known graph data structures to encode them. To this end, early implementations of model checkers used adjacency lists to represent transitions and a dictionary to look up atomic propositions true in a state. This approach had a major disadvantage. Since all possible system states must be pre-computed, an intractable state space is often generated for even simple systems.

Fortunately, it is often unnecessary to generate the entire state space. Given some initial state, there is no need to consider unreachable states and in practice, the unreachable part of the state space is significant. An improved methodology would be, given an initial state, compute its successors based upon rules in the program specification. While generating the state space in this manner, algorithms require some method of storing states encountered. Each encountered state is encoded as a state vector and stored in a data structure, e.g., a hash table. A technique where each state is represented explicitly in its own piece of memory is called an explicit-state encoding. The standard algorithm for explicit state space exploration [58] is outlined in Figure 2.6:

```
1.   reached := unexplored := {s0}
2.   While unexplored ≠ ∅
3.   {
4.       remove a state s from unexplored
5.       for each transition s → s'
6.       {
7.           if s' = error
8.           {
9.               stop and report error
10.          }
11.          if s' ∉ reached
12.          {
13.              add s' to reached and unexplored
14.          }
15.      }
16.  }
```

Figure 2.6: Standard algorithm for explicit state space exploration.

2.4.2 Symbolic Model Checking

The use of Binary Decision Diagrams (BDD) [4] as an alternative storage scheme can result in the verification of significantly larger systems. Model checking using BDDs is called symbolic model checking. The name highlights the fact that individual states are not explicitly enumerated and stored. Alternatively, the sets of states which satisfy a formula being checked are encoded symbolically.

Binary Decision Diagrams (BDD).

BDDs are a representation of Boolean functions and an alternate encoding of binary decision trees. A binary decision tree is comprised of non-terminal nodes labelled with Boolean variables x, y, z, ... and terminal nodes labelled with values 0 or 1. Let T be a finite binary

decision tree that defines a unique Boolean function of the variables in non-terminal nodes as follows. Given an assignment of 0 and 1 to the Boolean variables occurring in T, start at the root of the tree and at each node take the dashed line when the truth value of the variable is 0. Alternately, exit via the solid line. The value of the reached terminal node gives the truth value of the function. Figure 2.7 illustrates for the Boolean function $f(x_1, x_2, x_3) = (x_1x_2 + x_2x_3 + x_3x_1)$.

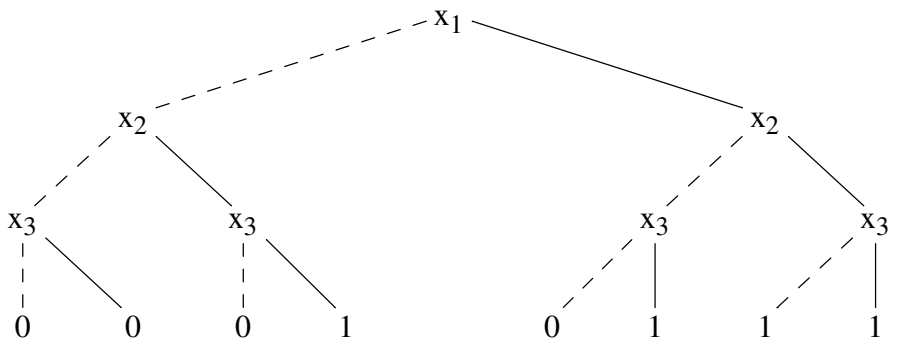


Figure 2.7: Representing Boolean function using BDDs.

Binary decision trees and truth tables are comparable in terms of size. Therefore, a function depending on n Boolean variables will generate a binary decision tree with a minimum of $2^{n+1}-1$ nodes. Binary decision trees often contain redundancy in the form of duplication that can be exploited to produce a more compact representation. Three approaches to reducing the size of a binary decision tree are outlined below and illustrated in Figure 2.8:

(a) **Remove Duplicate Terminals**. Choose two representative terminal vertices, one for constant 0 and the other for the constant 1. All arcs going to a 0 terminal are mapped to the single representative terminal 0. Similarly all arcs to a 1 terminal are mapped to the 1 representative terminal.

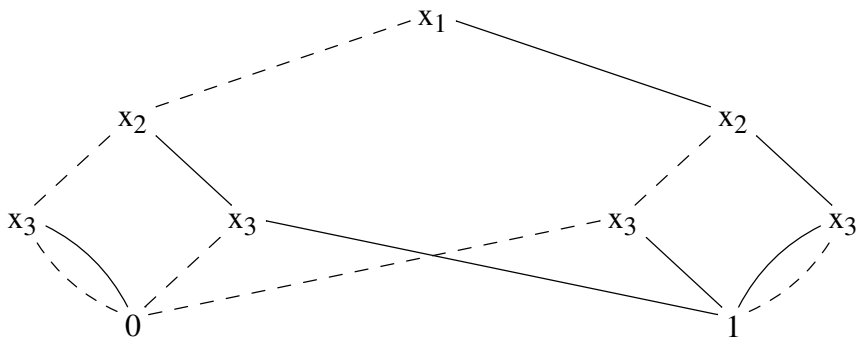
(b) **Remove Duplicate Nonterminals**. If two distinct nodes in the tree are roots of identical sub trees, then one of them can be removed and all its incoming edges are redirected to the remaining instance.

(c) **Remove Redundant Tests**. If both outgoing edges of a node n point to the same node m, then we eliminate that node n, sending all its incoming edges to m.

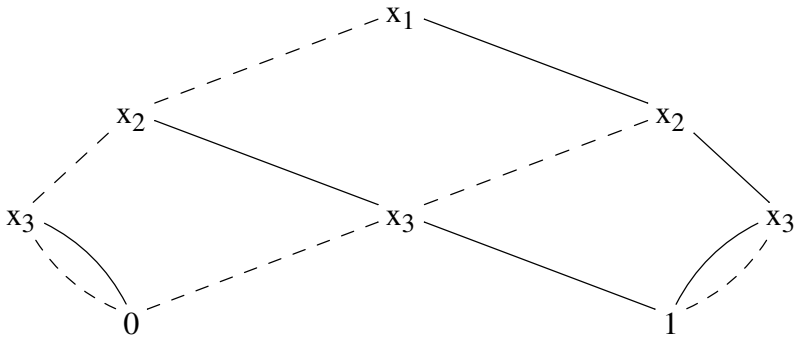
The exulting structures are examples of BDDs which are more general than binary decision trees as they allow sharing of leaves.

Ordered Binary Decision Diagrams (OBDD)

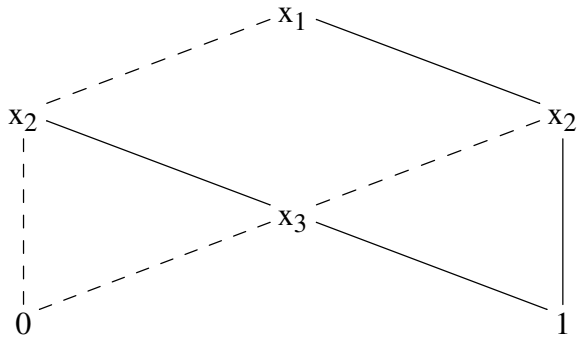
BDDs can provide a compact representation of Boolean functions. However, a BDD may have multiple occurrences of a Boolean variable along a path. This leads to the scenario



(a) Eliminate duplicate terminals



(b) Eliminate duplicate nonterminals



(c) Eliminate redundant tests

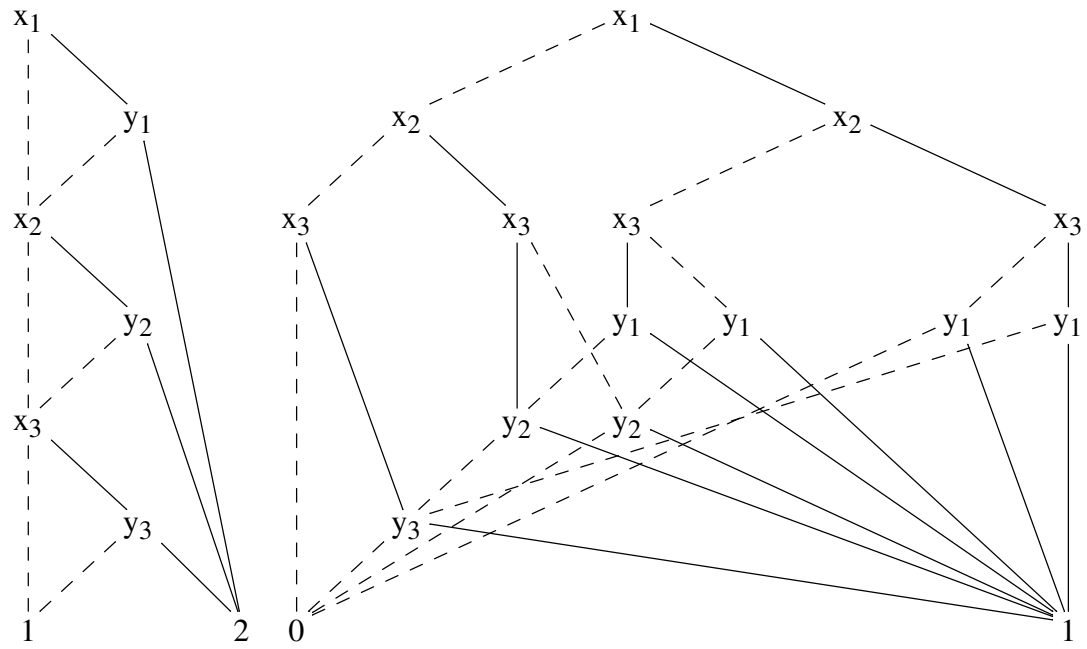
Figure 2.8: Reduction rules for BDDs [61].

where different diagrams can be produced for the same function. Therefore, testing equivalence between two BDDs is NP hard [14]. This problem can be resolved by imposing an ordering on the variables.

Let $[x_1, \ldots, x_n]$ be an ordered list of unique variables and let B be a BDD. B has ordering $[x_1, \ldots, x_n]$ if all variables in B occur in the list and every occurrence of x_i along any path in B is followed by x_j where $i < j$. Figure 2.7 has variable ordering $[x_1, x_2, x_3]$. It follows from the definition of an OBDD that multiple occurrences of any variable along a path are not permitted.

The size and form of an OBDD representing a function is heavily dependent on the vari-

able ordering. This is illustrated in Figure 2.9 which shows two OBDD for the function $(x_1y_1 + x_2y_2 + x_3y_3)$. Figure 2.9a implements the ordering $[x_1, y_1, x_2, y_2, x_3, y_3]$ and results in an OBDD with 8 vertices. Figure 2.9b implements ordering $[x_1, x_2, x_3, y_1, y_2, y_3]$ and yields an OBDD with 16 nodes. Although finding the optimal ordering is itself a computationally expensive problem, there are good heuristics which will usually produce a fairly good ordering [41].



(a) Ordering x_1, y_1, x_2, y_2, x_3, y_3

(b) Ordering x_1, x_2, x_3, y_1, y_2, y_3

Figure 2.9: OBDDs for the function $(x_1y_1 + x_2y_2 + x_3y_3)$ [61]

The overhead of determining and imposing an ordering is deemed acceptable as a OBDD representing a given function using a given ordering is unique. In other words, if B and B' represent the same Boolean function and during construction of the OBDD the same ordering was imposed, the resulting structure will be identical. It follows that checking whether two OBDDs represent the same function is simply a matter of checking whether they have the same structure.

Representing Subsets of the Set of States

Now that OBDDs have been defined the question is how to use them to represent various subsets of states. Let S be a finite set of states. Elements of S must be encoded as Boolean values. This is achieved by assigning elements $s \in S$ a unique vector of Boolean values $(v_1, v_2, \ldots, v_n)$ where each $v_i \in \{0, 1\}$.

A subset $T \subset S$ can be denoted by the Boolean function f_T, mapping the characterising function onto 1 if $s \in T$ and 0 otherwise. There are 2^n possible Boolean vectors of length

n and so n should be chosen such that $2n - 1 < |S| \leq 2^n$, where $|S|$ is the number of elements in S. For example the set $P = \{(0, 1, 0) (1, 0, 1)\}$ has the characteristic function $f_P = (\neg v_1 \wedge v_2 \wedge \neg v_3) \vee (v_1 \wedge \neg v_2 \wedge v_3)$.

When S represents the set of states of a transition system, a subset of atoms can be used to provide a unique Boolean vector for each $s \in S$. Let P(Atoms) be a set of subsets of Atoms with ordering $x_1, x_2, \ldots, x_n$, state $s \in S$ can be represented by the the vector $(v_1, v_2, \ldots, v_n)$, where, for each i, v_i equals 1 if the atom is valid in the state or 0 otherwise. Consequently this state is represented by an OBDD for the Boolean function $(l_1 \cdot l_2 \cdot \cdots l_n)$ where l_i is 1 if x_i is a valid atom and 0 otherwise. The set of states $s_1, s_2, \ldots, s_m$ is represented by the OBDD of the Boolean function $(l_{11} \cdot l_{12} \cdots l_{1n} + l_{21} \cdot l_{22} \cdots l_{2n} + \cdots + l_{m1} \cdot l_{m2} \cdots l_{mn})$.

Representing the Transition Relation

A transition relation is defined as $R \subseteq S \times S$. In the section above it was shown that subsets of a given finite set may be represented as OBDDs by considering the characteristic function of a binary encoding. To encode a transition relation two copies of a Boolean vector are required. Thus, the transition $s \rightarrow t$ is represented by the pair of Boolean vectors $(v_1, v_2, \ldots, v_n)$, $(v'_1, v'_2, \ldots, v'_n)$. As an OBDD, the transition is represented by Boolean function $(l_1 \cdot l_2 \cdots l_n) \cdot (l'_1 \cdot l'_2 \cdots l'_n)$ and a set of transitions is the + of such formulae.

The key idea behind applying OBDDs to finite systems is to take a system specification and synthesise the OBDD directly, without having to go via intermediate representations. Fortunately, certain specification languages enforce updates to variable to be defined using the variables current value [69, 25]. This type of variable update description can be compiled into a set of Boolean functions. Therefore, given the OBDD for a set of states and the transition relation, one step successors and one step predecessors can be computed using BDD symbolic based algorithms. This can be done repeatedly to explore all reachable states [4].

2.4.3 Probabilistic Representations

Discrete Time Markov Chains can be described as large sparse real-valued matrices where a sparse matrix is one populated mostly by the value 0. The naive data structure for a matrix is a two-dimensional array where entries can be accessed via two indices. To encode a $m \times n$ matrix, memory is required to store all $(m \times n)$ entries even when the majority hold the value 0. Therefore, it is memory efficient to only store non-zero entries.

The most common scheme employed by model checkers is a row-major sparse matrix encoding [69, 17]. This data structure stores information on non zero entries using three arrays, *row*, *column* and *value*. The *value* array stores the actual value of all entries in the matrix

while *row*, stores the column index of each matrix entry. Both these arrays are row ordered. The final array, *row*, provides a means of indexing into the column array that in turn reveals the value of a desired element.

For example, an entry in position (r, c) is found by indexing into locations r and r + 1 in *row*. The located values are used in turn to index into *column* positions *column*[*row*[r]] and *column*[*row*[r+1]−1]. The values contained between and including these indices are checked to see if they equal c. If the value c is not present, then (r, c) = 0. If it is present, then (r, c) is non-zero and its value can be found by looking up the value at the same position in the value array. An example of this encoding is outlined in Figure 2.10;

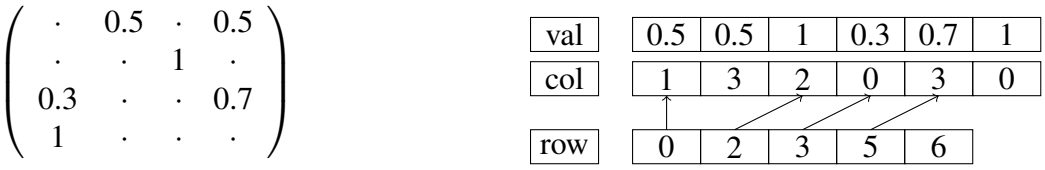


Figure 2.10: A four state DTMC and its sparse storage [82].

An MDP can be represented by a matrix where states that allow a non deterministic choice between several probabilistic distributions are described using several rows. The row major scheme can be modified to allow this encoding. As before, the data structure uses the three arrays, *row*, *column* and *value* in addition to the array *nc*. The implementation of *value* and *column* are identical. However, two levels of indexing are required to represent states and available non-deterministic choices. This additional indexing is provided by the *nc* array. An example of this encoding is outlined Figure 2.11.

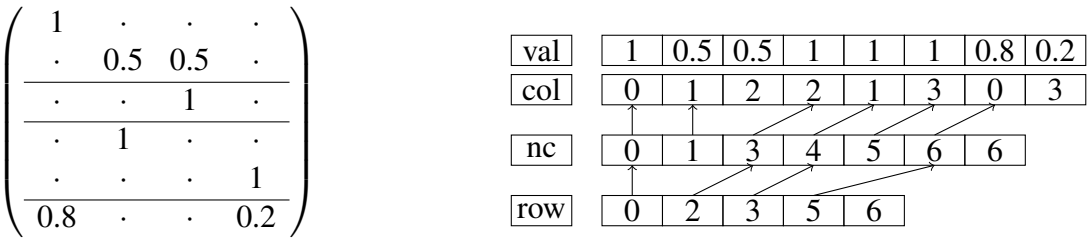


Figure 2.11: A four state MDP and its sparse storage [82].

The row-major data structure facilitates a compact representation and quick access to matrix elements. Its main downside is the expense involved in structure modifications. However, the scheme is well suited for model checking as many of the desired matrix operations can be performed efficiently [42].

The use of symbolic data structures for the probabilistic case are most commonly implemented by extending BDD based representations to allow functions which can take any

value, not just 0 or 1. This extension is termed Multiple Terminal Binary Decision Diagrams MTBDDs and allows a BDD with more than two terminal nodes. Furthermore, it has been shown that MTBDDs can represent matrices over any finite set as well as implementing standard matrix operations, such as scalar multiplication, matrix addition and matrix multiplication [50].

2.5 Model Checking Algorithms

With the semantic definitions for LTL, CTL and PCTL presented in Section 2.3 and an overview of representation schemes provided in Section 2.4, we will outline how the representation schemes can be manipulated to provide answers to posed logical formulae.

2.5.1 CTL Model Checking

The CTL model checking algorithm [22] answers the question $\mathcal{M}, s_0 \models \phi$ by returning all states of $\mathcal{M}$ that satisfy condition ϕ. From the returned set of states it is easy to check if $\mathcal{M}, s_0 \models \phi$ by verifying that the returned set contains s_0. The model checking algorithm for CTL is a labelling algorithm that marks states which satisfy sub-formulae of the formula to be checked. Fortunately, the algorithm does not have to handle every CTL connective as any formula can be reformulated using only $\bot$, $\neg$ and $\wedge$ and temporal connectives and AF , EU and EX.

The CTL algorithm takes a model $\mathcal{M}$ and a CTL formula ϕ as input. Immediately, ϕ is reformulated and states of $\mathcal{M}$ are labelled with sub formulae of ϕ that are satisfied in the state. The algorithm begins with the smallest sub-formulae and works outwards towards ϕ. Suppose ψ is a sub-formula of ϕ and states satisfying sub-formulae of ψ have already been labelled. The states required to be labelled with ψ can be determined by a case analysis. If ψ is:

- $\bot$: then no states are labelled with $\bot$.
- p: where p is an atom from the set of Atoms then label the state with p is true in the state.
- $\psi_1 \wedge \psi_1$: label s with $\psi_1 \wedge \psi_1$ if s is already labelled both with ψ_1 and with ψ_2 .
- $\neg\psi_1$: label s with $\neg\psi_1$ if s is not already labelled with ψ_1 .
- AF ψ_1:
  - If any state s is labelled with ψ_1 , label it with AF ψ_1.

  - Repeat: label any state with AF ψ_1 if all successor states are labelled with AF ψ_1 , until there is no change.

- E [ψ_1 U ψ_2]:
  - If any state s is labelled with ψ_2 , label it with E [ψ_1 U ψ_2].
  - Repeat: label any state with E [ψ_1 U ψ_2] if it is labelled with ψ_1 and at least one of its successors is labelled with E [ψ_1 U ψ_2] , until there is no change.

- EX ψ: label any state with EX ψ if one of its successors is labelled with ψ.

Once all sub-formulae of ϕ including ϕ itself have been labelled to states, the set of states labelled with ϕ are returned. The complexity of this algorithm is O(f $\cdot$ V $\cdot$ (V + E)), where f is the number of formula connectives, V the number of states and E the number of transitions [25].

2.5.2 LTL Model Checking

The state-labelling approach of CTL is not appropriate for LTL model checking as sub-formulae are evaluated along paths of a system and not states. While numerous implementations of slightly different LTL model checking algorithm appear in the literature [54, 58], nearly all follow the same strategy [61]. These algorithms take a model $\mathcal{M}$, a state s $\in$ S and a LTL formula ϕ as input and determine whether $\mathcal{M}$, s $\models \phi$ using the following steps:

1. Construct an automaton for the formula $\neg\phi$, denoted $A_{\neg\phi}$. $A_{\neg\phi}$ accepts a trace that is a sequence of valuations of the propositional atoms. Thus, the automaton $A_{\neg\phi}$ encodes all the traces satisfying the property $A_{\neg\phi}$. A trace can be generated for any path.
2. Combine $A_{\neg\phi}$ and system model $\mathcal{M}$. This results in a new automaton that has all the paths of $A_{\neg\phi}$ and $\mathcal{M}$. In practice the new system is constructed by letting the system model $\mathcal{M}$ and the automaton for the formula $A_{\neg\phi}$ to take alternate progressing steps.
3. Attempt to find a path from the starting state to a set of final states defined by $A_{\neg\phi}$. If such a path is found it can be interpreted to mean that M, s $\not\models \phi$. In this instance a counterexample can be extracted from the located path.

The complexity of this algorithm is $O(|S| + |R| \cdot 2^{O(|\phi|)})$ and is therefore exponential in the length of the formula to be checked [58]. In the worst case this means that LTL model checking is significantly more complex than CTL model checking. Fortunately, the worst case is rarely achieved and in practice there is little run time difference between the algorithms.

2.5.3 PCTL Model Checking

The PCTL model checking algorithm [53] is similar to the CTL model checking algorithm detailed in Section 2.5.1. However, it is now necessary to compute relevant probabilities. For model checking operator $P_{\bowtie prob}[\psi]$ applied to a DTMC the probability of a path leaving each state s satisfying the path formula ψ must be computed. This may require a calculation involving the operators $P_s(X\ \phi)$, $P_s(\phi\ U^{\leq k}\ \psi)$ and $P_s(\phi\ U\ \psi)$. With this information it is then possible to compute $Sat(P_{\bowtie prob}[\psi]$ as $\{s \in S \mid P_s(\psi) \bowtie prob\}$.

The Next Operator

Observe that $P_s(X\ \phi)$ is the sum of the probabilities of reaching a state in the next transition where ϕ holds i.e. $\sum_{s' \in Sat(\phi)} \mathbf{P}(s, s')$. Let $\underline{\phi}$ be a vector indexed by states where $\underline{\phi}(s) = 1$ if $s \models \phi$ and $\underline{\phi}(s) = 0$ if $s \not\models \phi$. Vector $\underline{P(X\ \phi)}$ of required probabilities can be calculated by the matrix-vector multiplication $\underline{P \cdot \phi}$.

The Bounded Until Operator

To perform the calculation associated with this operator the set of states are divided into the three disjoint sets: $S^{no} = S \setminus (Sat(\phi_1) \cup Sat(\phi_2))$, $S^{yes} = Sat(\phi_2)$ and $S^{?} = S \setminus (S^{no} \cup S^{yes})$. The sets S^{yes} and S^{no} contain the states where $P_s(\phi U^{\leq k} \phi)$ equals 1 and 0 respectively and $S^{?}$ contains all other states. For the set of states $S^{?}$, we have

$$P_s(\phi_1 U^{\leq k} \phi_2) = \begin{cases} 0 & \text{if } k = 0 \\ \sum_{s' \in S} \mathbf{P}(s, s') \cdot P_s(\phi_1 U^{\leq k-1}\ \phi_2) & \text{if } k \geq 1 \end{cases}$$

Let $\underline{P_s(\phi\ U^{\leq k}\ \phi)}$ be a state indexed vector and by defining the matrix $\mathbf{P'}$ as follows

$$\mathbf{P'}(s, s') = \begin{cases} \mathbf{P'}(s, s') & \text{if} \in S^{?} \\ 1 & \text{if } s \in S^{yes} \text{ and } s = s' \\ 0 & \text{if } s \in S^{no} \end{cases}$$

the required probabilities can be computed in the following manner. If $k = 0$ and $s \in S^{yes}$, $P_0(s) = 1$ and if $s \in S^{no}$, $P_0(s) = 0$. In the instance where $k \geq 1$ vector $\underline{P_s(\phi_1 U^{\leq k-1}\ \phi_2)}$ can be calculated by k matrix-vector multiplication $\mathbf{P'} \cdot \underline{P_s(\phi_1 U^{\leq k-1}\ \phi_2)}$.

The Until Operator

As with the bounded until operator, all states are divided into the three disjoint sets, S^{yes}, S^{no} and $S^{?}$. The sets are defined as above, however sets S^{yes}, S^{no} are extended to contain all

states for which $P_s(\phi_1 \, U \, \phi_2)$ are respectively, 1 or 0. Set S^{no} is calculated by first computing the set of states reachable with non-zero probability satisfying ϕ_2 whose predecessors do not satisfying ϕ_1. Subtracting these states from set S, produces the set of states with 0 probability. Set S^{yes} similarly calculates the set of states reachable with probability less than 1, that satisfy ϕ_2 whose predecessors do not satisfy ϕ_1. By subtracting these states from S, the set of states satisfying the operator with probability 1 is determined.

The reason behind the pre-computation of S^{yes}, S^{no} is that it ensures a unique solution to the linear equation system and reduces the set of states in $S^?$, for which probabilities must be computed numerically. Furthermore, the model checking of qualitative properties where the probability bound is 1 or 0 requires no further computation. The final set $S^?$ can be calculated by solving the linear equation

$$P_s(\phi_1 U \phi_2) = \begin{cases} \sum_{s' \in S} \mathbf{P}(s, s') \cdot P_s(\phi_1 U \phi_2) & \text{if } s \in S^? \\ 1 & \text{if } s \in S^{yes} \\ 0 & \text{if } s \in S^{no} \end{cases}$$

To reconstruct the problem in the form $\mathbf{A}\underline{x} = \underline{b}$. Let $\underline{b}$ be the state indexed vector where $\underline{b}(s) = 1$ if $s \in S^{yes}$ and $\underline{b}(s) = 0$ if $s \in S^{no}$, and $\mathbf{A} = \mathbf{I} - \mathbf{P'}$ where $\mathbf{I}$ is the the identity matrix and matrix $\mathbf{P'}$ is as defined below;

$$\mathbf{P'}(s, s') = \begin{cases} \mathbf{P}(s, s') & \text{if } s \in S^? \\ 0 & \text{if } s \in S^{yes} \\ 0 & \text{if } s \in S^{no} \end{cases}$$

The linear equation system $\mathbf{A}\underline{x} = \underline{b}$ can then be solved using direct methods, such as Gaussian elimination, or iterative methods, such as Jacobi, Gauss-Seidel or the Power method [93]. In model checking it is common to have to manage large models and therefore iterative methods are preferred. Gauss-Seidel typically outperforms Jacobi due to faster convergence and has the added benefit of only needing to store a single solution vector. Both of these methods usually outperform the Power method. However the Power method has guaranteed convergence.

PCTL model checking over MDPs

For an MDP computing the probabilities of the PCTL operators: next, bounded until and until differs as all adversaries $A \in Adv$ have to be accounted for. To determine if a probability bound holds either the maximum or minimum probability for the PCTL formula is calculated depending whether the relational operator defines an upper or lower bound. Furthermore, the

calculation method for maximum and minimum probabilities changes depending whether the set of all adversaries or just the set of fair adversaries is considered. In practice this consideration only affects the until operator as fairness only places restrictions on the long-run behaviour of the system.

The Next Operator

For the PCTL next operator two cases must be considered;

$$P_s^{max}(X\ \phi) = max_{\mu \in Steps(s)} \{\sum_{s' \in Sat(\phi)} \mu(s')\}$$
$$P_s^{min}(X\ \phi) = min_{\mu \in Steps(s)} \{\sum_{s' \in Sat(\phi)} \mu(s')\}$$

Let $m = \sum_{s \in S} |Steps(s)|$, the total number of nondeterministic choices in all states of the MDP. The function Steps can be represented as an $m \times |S|$ matrix. Let $\underline{\phi}$ be a state-indexed vector where $\underline{\phi}(s) = 1$ if $s \models \phi$ and $\underline{\phi}(s) = 0$ if $s \not\models \phi$. The calculation of either $P_s^{max}(X\ \phi)$ or $P_s^{min}(X\ \phi)$ can be carried out in two steps:

1. The matrix-vector multiplication $Steps \cdot \underline{\phi}$ results in a vector of length m.
2. From this vector select the maximum or minimum value given for each state depending on the operator being calculated. The results in a new vector with length $|S|$.

Bounded Until Operator

As was the case in DTMCs the set of states is divided into the three disjoint subsets: $S^{no} = S \setminus (Sat(\phi_1) \cup Sat(\phi_2))$, $S^{yes} = Sat(\phi_2)$ and $S^{?} = S \setminus (S^{no} \cup S^{yes})$. S^{yes} and S^{no} contain the set of states for which $P_s^{max} = (\phi_1\ U^{\leq k}\ \phi_2)$ or $P_s^{min} = (\phi_1\ U^{\leq k}\ \phi_2)$ equal 1 or 0 respectively. $S^{?}$ contains the remaining states and there are two cases;

$$P_s^{max}(\phi_1 U^{\leq k} \phi_2) = \begin{cases} 0 & \text{if } k = 0 \\ max_{\mu \in Steps(s)}\{\sum_{s' \in S} \mu(s') \cdot P_s^{max}(\phi_1 U^{\leq k-1} \phi_2)\} & \textit{if } k \geq 1 \end{cases}$$

$$P_s^{min}(\phi_1 U^{\leq k} \phi_2) = \begin{cases} 0 & \textit{if } k = 0 \\ min_{\mu \in Steps(s)}\{\sum_{s' \in S} \mu(s') \cdot P_s^{min}(\phi_1 U^{\leq k-1} \phi_2)\} & \text{if } k \geq 1 \end{cases}$$

Using the same matrix representation as above the computation of $P_s^{max}(\phi_1\ U^{\leq k}\ \phi_2)$ or $P_s^{min}(\phi_1\ U^{\leq k}\ \phi_2)$ can be carried out in k iterations. Every iteration comprises of one matrix-vector multiplication and the selection of the maximum or minimum value.

The Until Operator

For the PCTL operator until we must compute either $P_s^{max} = P_s^{max}(\phi_1 \text{ U } \phi_2)$ or $P_s^{min} = P_s^{min}(\phi_1 \text{ U } \phi_2)$. Two cases must be considered, all adversaries and fair adversaries. For clarity P_s^A will be used to denote $P_s^A(\phi_1 \text{ U } \phi_2)$. Beginning with the case for the set of all adversaries the set of states are divided into the three disjoint subsets $S^{no} = S \setminus (Sat(\phi_1) \cup Sat(\phi_2))$, $S^{yes} = Sat(\phi_2)$ and $S^? = S \setminus (S^{no} \cup S^{yes})$.

For P_s^{max}, S^{no} contains the set of states for which $P_s^A = 0$ for every adversary A. The calculation of P_s^{max} proceeds by first computing the set of states reachable with non-zero probability under some adversary that satisfy ϕ_2 and whose predecessors satisfied ϕ_1. Removing these states from set S produces the set of states with probability 0. For P_s^{min} , S^{no} contains all states for which $P_s^A = 0$ for some adversary A and is computed in a similar fashion.

The algorithm for the computation of P_s^{max} in the set S^{yes} is more complex but works on the same principle of calculating the states reachable with probability less than 1 and subtracting from set S. The algorithm depends on two nested loops .The outer loop computes a set of states R and by the end of its execution will contain all states where $P_s^A = 1$ for some adversary A. With each iteration of the outer loop, invalid states that were identified by the inner loop are removed. They are determined as the states which can no longer reach a state where $Sat(\phi_2)$ without passing through a state not in $Sat(\phi_1)$ or a previously removed state. For p_s^{min} , S^{no} is assumed to be the set of states $Sat(\phi_2)$ for which P_s^{min} is trivially 1.

The minimum and maximum probabilities for the remaining states $S^?$ can either be computed using value iteration or solved by reduction to a linear optimisation problem. Linear optimisation problems can be solved using classic techniques such as the Simplex, Ellipsoid method or Interior point method [72] that yield an exact solution in a finite number of steps. However, these direct methods are not well suited to problems of the size commonly handled. Therefore the problem must be reformed to allow an iterative approach to provide the solution to P_s^{max} and P_s^{min}, where $P_s^{max} = \lim_{n\to\infty} P_s^{max(n)}$:

$$P_s^{max(n)} = \begin{cases} 0 & \text{if } s \in S^{no} \\ 1 & \text{if } s \in S^{yes} \\ 0 & \text{if } s \in S^? \text{ and } n = 0 \\ \max_{\mu \in Steps(s)} \{\sum_{s' \in S} \mu(s') \cdot P_s^{max(n-1)}\} & \text{if } s \in S^? \text{ and } n > 0 \end{cases}$$

and where $P_s^{min} = \lim_{n\to\infty} P_s^{min(n)}$:

$$P_s^{min(n)} = \begin{cases} 0 & \text{if } s \in S^{no} \\ 1 & \text{if } s \in S^{yes} \\ 0 & \text{if } s \in S^{?} \text{ and } n = 0 \\ \min_{\mu \in Steps(s)} \{\sum_{s' \in S} \mu(s') \cdot P_s^{min(n-1)}\} & \text{if } s \in S^{?} \text{ and } n > 0 \end{cases}$$

The values of P^{max} and P^{min} can be approximated by an iterative computation, stopping when some convergence criteria has been satisfied. There is a strong similarity between a single iteration of this method and one required for the bounded until operator. Hence, assuming a similar representation scheme each iteration can be performed using one matrix-vector multiplication and selection of the appropriate maximum or minimum value.

Until Operator (Fair Adversaries)

This section covers the calculation of P^{max} and P^{min} over fair adversaries for the PCTL operator until. The process of computing P^{max} remains unchanged. By considering a more restricted class of adversaries, the maximum probability clearly cannot increase. Furthermore, the probability does not decrease as fairness only places restrictions on infinite behaviour and for a path to satisfy an until formula only some finite, initial portion of it is relevant.

When computing P^{min}, the minimum probability over all fair adversaries can be higher than the minimum for all adversaries. However, the new minimum probability can be ascertained without much additional effort. By considering the probability that $\phi_1 \cup \phi_2$ is not satisfied, a maximal probability calculation is obtained. This allows the same method from the previous section to be used. The desired probabilities can then be obtained by subtracting the calculated probabilities from 1 [10].

2.6 Model Checking Tools

SPIN is an open-source model checker developed at Bell Labs for the verification of non-probabilistic concurrently executing processes [58]. Models are described in a high level language called Promela that consists of global variables, channel declarations and process type declarations, together with an initialisation process. Properties can be defined using LTL, which in turn is translated into an automaton to provide an efficient verification implementation. Additionally, the option of hand constructing a more expressive property directly as an automaton is available. The verifier can perform both depth and breadth first search over the state-space to check for absence of deadlock, or satisfaction of safety properties. As is traditional, if a property is violated a counter example is generated.

The combination of the Promela language and SPIN verifier has been widely used in safety critical systems. Examples including the verification of control algorithms for a movable storm surge barrier where manual management was considered too high a risk [92]. A more recent example is verification of the resource arbiter used to manage all motors on the Mars Exploration Rovers [59].

Another example of an explicit-state model checker is Murϕ [29]. The Murϕ specification language consists of infinitely executing guarded commands and unlike SPIN there is no provision for temporal logics. The verifier can only check the state-space for absence of deadlock, or satisfaction of assert statements. Murϕ has seen most use in the design of cache coherence algorithms and protocols.

For non-probabilistic symbolic model checkers Symbolic Model Verifier, SMV is the benchmark implementation [25]. SMV uses an OBDD based algorithm for the verification of CTL properties against a specification written in the SMV language. The SMV language supports finite data structures such as Booleans, scalars and fixed arrays. The primary purpose of the language is the specification of the transition relation which in turn allows the generation of the OBDD directly from the language description. Perhaps the most recognised application of SMV is in the verification of the Cache Coherence Protocols for Distributed File Systems [25].

In the probabilistic realm a tool for Qualitative and Quantitative Linear Time analysis of Reactive Systems, LiQuor [17], was developed by the University of Bonn. LiQuor is a tool for verifying probabilistic reactive systems specified in ProbMela, a probabilistic guarded command language inspired by the modelling language Promela [8]. Like SPIN, LiQuor relies on the automata-based approach to model check linear time properties. The underlying data structure is explicit in nature and is used to encode an MDP. In design, LiQuor is an amalgamation of many separate tools, the ProbMela compiler, Cocktail providing the user interface and Appetizer for user driven simulation.

The Markov Reward Model Checker [64] provides model checking of CSL over CTMCs and PCTL over DTMCs. Both Markov Chains are stored in explicit data structures such as sparse matrices. The transition matrices for the DTMCs or CTMCs to be analysed are also specified explicitly: the user provides a list of all the states and transitions which make up the model. The format in which checker accepts information allows it to be easily integrated with other tools.

Finally, the Probabilistic Symbolic Model Checker, PRISM [69], was developed at the University of Birmingham. It is a probabilistic model checker that employs automatic formal verification techniques to enable the analysis of stochastic systems. PRISM provides support

for three types of probabilistic models; DTMCs, MDPs and CTMCs. The basic underlying data structures of PRISM are BDDs and MTBDDs. However, PRISM provides three distinct engines. The first is a pure MTBDD implementation, the second is explicit, based on sparse matrices; and the third uses a hybrid symbolic, explicit approach. Properties of models are written in the PRISM property specification language, based on the three probabilistic temporal logics PCTL and LTL for DTMCs, PCTL for MDPs, and CSL for CTMCs. Further, analytical abilities include the power to enhance the richness of models, by the assignment of costs and rewards to certain model behaviours, i.e transitions within the model. PRISM also provides wide ranging support for the automated analysis of quantitative properties.

2.6.1 State Space Explosion

As mentioned in Section 2.1 a major problem which limits the application of model checking is the of state-space explosion. Although verification algorithms usually have polynomial run time complexity, this is offset as the number of states in a model grows exponentially with the number of variables. This means that even trivial real-life systems can require many millions of states to define their behaviour.

To illustrate this consider a system composed of ten identical processes that contain three Boolean variables and five bounded integers in the range $\{ 0, \ldots, 9 \}$ [9]. A system with this arrangement will consists of $10 \cdot 2^3 \cdot 10^5 = 8,000,000$ states. Now consider if an array of 50 bit elements are added to the program. Now $800,000,000 \cdot 2^{50}$ states are required to describe all behaviours of the system.

Three main approaches have been identified for tackling this problem:

1. Translation of a specification into a form that captures the same essential behaviour, but results in a smaller model. This includes techniques such as design abstraction and source code or communication structure optimisation.
2. Reducing the size of memory required to store a state. The most successful technique of this kind is symbolic model checking; while state compression and supertrace verification have also proved useful in practice.
3. Minimise the number of states that must be checked to verify a property. Techniques include on-the-fly model checking, partial-order reduction, symmetry reduction, abstraction and compositional reasoning.

2.7 Summary

In this chapter we have provided a detailed account of current approaches to model checking, starting with theory and working through to implementation. The chapter closes by mentioning the state space explosion problem and the wide variety of techniques that have been proposed to combat it. In the next chapter we will specifically focus on the state space management technique of symmetry reduction.

CHAPTER 3

Symmetry Reduction

This chapter presents established approaches for the state space management technique of Symmetry Reduction. Section 3.1 introduces the notion of Symmetry Reduction with Section 3.2 providing a mathematical explanation for its application in the context of model checking. Issues concerning the identification and application of symmetry are discussed in Section 3.3 and Section 3.4 respectively. The chapter closes with a review of currently available tools that implement an approach to symmetry reduction.

Concurrent systems often contain replication and as a result model checking algorithms may spend a significant proportion of time searching over equivalent areas of the state space. Consider a system comprised of numerous processes running the same program. The only distinguishable difference between them is the process name. In this instance processes can be viewed as interchangeable and any transformation that consistently swaps them throughout the system will not impact the overall set of system behaviours. Once a symmetry has been established the question becomes how to exploit the knowledge during verification.

Continuing the example, every system state does not have to be individually encountered and stored, they can be collapsed into one representative state in a reduced system. Therefore, given n processes, potentially n! original states can be collapsed and their behaviours represented by a single new state in a reduced system. Consider the mutual exclusion protocol outlined in Figure 2.3. This example consists of 2 identical processes, excluding process identifiers, and clearly demonstrates the existence of symmetry within a Kripke structure.

Furthermore, the mutual exclusion property AG $(\neg(C_1 \wedge C_2))$ holds for any state A, B and B, A, where A and B take a value from the set {N, T, C}. If state A, B satisfies the mutual exclusion property so does B, A.

3.1 Group Theory

Symmetries of a model structure form a group and the survey of symmetry reduction techniques in this chapter, and the techniques develop throughout the thesis require some definitions and results from group theory. This section covers basic definitions [73, 56] and provides a specific overview of Permutation and Symmetric Groups.

3.1.1 Groups, Subgroups and Homomorphisms

Definition. A group $(G, *)$ contains a set S with a closed binary operation $*$ such that the group axioms hold;

1. For every $x, y, z \in G$, we have $(x * y) * z = x * (y * z)$.
2. There is an element $e \in G$ such that for every $x \in G$, $e * x = x * e = x$.
3. For each $x \in G$ there is an element $x' \in G$ such that $x * x' = x' * x = e$.

The number of distinct elements in a finite group G, is called the order of the group and is denoted by $|G|$. The identity element is denoted by e or e_G if ambiguous, and the inverse of an element x of G is denoted by x^{-1} . Finally, the binary operation $*$, usually a composition of mappings is written xy for $x * y$.

Definition. A group is called abelian or commutative if it satisfies the additional property, $x * y = y * x$ for all elements $xy \in G$.

Definition. A subset H of a group G is called a subgroup of G if the following conditions are satisfied:

1. $e_G \in H$.
2. if $x, y \in H$ then $xy \in H$.
3. if $x \in H$, then $x^{-1} \in H$.

If H is a subgroup of G it can be written $H \leq G$. Additionally, a subgroup H is proper subgroup if $H \neq G$ and is written $H < G$.

Definition. Let H be a subgroup of G and $g \in G$ then the subset $gH = \{gh \mid h \in H\} \subseteq G$ is called the left coset of H. Similarly the subset $Hg = \{hg \mid h \in H\} \subseteq G$ is the right coset of H. The set of left cosets of H is denoted G/H and the set of right cosets $H\backslash G$.

Definition. Let A be any subset of a group G. The subgroup of G generated by A, denoted $\langle A \rangle$, or $\langle x_1, x_2 \ldots, x_n \rangle$ if $A = \{x_1, x_2, \ldots, x_n\}$ is the intersection of all subgroups of G containing A.

The group $\langle A \rangle$ is the smallest subgroup of G that contains A. It is often the case in computational applications that the generating set $\{x_1, x_2, \ldots, x_n\}$ has an ordering that is defined implicitly by the subscript of x_i

Definition. Let $(G, *)$ and $(H, \circ)$ be groups. A function $f : G \to H$ is a homomorphism if for all $a, b \in G$,

$$f(a * b) = f(a) \circ f(b)$$

A homomorphism σ is called a monomorphism, epimorphism or isomorphism if it is an injection, surjection or bijection, respectively. Groups G and H are isomorphic if there is an isomorphism $\sigma : G \to H$, denoted $G \cong H$. Finally an isomorphism from group G onto itself is called an automorphism.

3.1.2 Permutation Group

Definition. If X is a non empty set, a permutation of X is a bijection $\alpha : X \to X$ The set of all permutations of X is denoted S_X

Let X be a set. The symmetric group, Sym(X), consists of all the bijections from X to X, with map compositions as the group operator. A recurring special case is where X is finite, consisting of the first n natural numbers. Under these conditions the symmetric group is termed S_n.

Any bijection α can be denoted by two rows;

$$\alpha = \begin{pmatrix} 1 & 2 & \ldots & n \\ \alpha 1 & \alpha 2 & \ldots & \alpha n \end{pmatrix},$$

and the bottom row is a rearrangment of $\{1, 2, \ldots, n\}$. If Γ is a set of size n, then there is a bijection $\theta : \Gamma \to \{1, 2, \ldots, n\}$ which induces a mapping $\overline{\theta} : \mathrm{Sym}(\Gamma) \to S_n$ via $(\sigma)\overline{\theta} = \theta^{-1} \circ \sigma \circ \theta$ for every $\sigma \in \mathrm{Sym}(\Gamma)$. For example the Symmetric group S_3 is;

$$\begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}, \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix},$$

$$\begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix}, \begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{pmatrix}, \begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix}$$

The two rowed notation is not always convinent to work with or present. For this reason we introduce an equivalent notation.

Definition. If $x \in X$ and $\alpha \in S_X$, then α fixes x if $\alpha(x) = x$ and α moves x if $\alpha(x) \neq x$.

Definition. Let $i_1, i_2, \ldots, i_r$ be distinct integers between 1 and n. If $\alpha \in S_n$ fixes the remaining n – r integers and if

$$\alpha(i_1) = i_2, \alpha(i_2) = i_3, \ldots, \alpha(i_{r-1}) = i_r, \alpha(i_r) = i_1$$

then α is an r-cycle of length r. This is denoted $(i_1\ i_2 \cdots i_r)$

For example the following permutation can be expressed as (015)(2)(364)

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 5 & 2 & 6 & 3 & 0 & 4 \end{pmatrix}$$

Every 1-cycle fixes every element of X. Therfore, all cycles of length one are equal to the identity. Let $\alpha \in$ Sym(X). If α = identity then we write id for α as usual. A transposition is an element of Sym(Ω) that exchanges two elements and fixes all the others, in other words a cycle of length 2. Since any cycle can be written as a product of transpositions, it follows that any permutation on set Ω can be written as a product of transpositions.

Definition. A permutation group G on the set X is a subgroup of Sym(X).

Definition. Let G act on the set Ω. The equivalence relation $\{\sim G\}$ or $\sim$ if the context is clear is defined on Ω by $\alpha \sim \beta$ if and only if there exists a $g \in G$ with $\beta = \alpha^g$. The equivalence classes of $\sim$ are called the orbits of G on Ω. The orbit of a specific element $\alpha \in \Omega$ is denoted by α^G

Definition. Let G act on the set Ω, $\alpha \in \Omega$ and let $\Delta \subseteq \Omega$

- The stabiliser of $\alpha \in G$ is, $\{g \in G \,|\, \alpha^g = \alpha\}$ and is denoted by G_α.
- The setwise stabiliser $\{g \in G \,|\, \alpha^g \in \Delta \text{ for all } \alpha \in \Delta\}$ of $\Delta \in G$ is denoted by G_Δ.
- The pointwise stabiliser $\{g \in G \,|\, \alpha^g = \alpha \text{ for all } \alpha \in \Delta\}$ of $\Delta \in G$ is denoted by $G_{(\Delta)}$.

3.1.3 Group Actions

In the practical application of symmetry reduction, a fundamental idea is that a group of permutations acting on a set induces a group of permutations acting on a different larger set. In the context of model checking this takes the form of a group of process identifier permutations inducing a group of permutations on a set of states.

Definition. A group G acts on the non-empty set X if to each $\alpha \in$ G and x $\in$ X there corresponds a unique element α(x) $\in$ X and that, for all x $\in$ X and $\alpha, \beta \in$ G, $\alpha\beta$(x) = $\alpha(\beta(x))$.

Let G act on X. Then to each $\alpha \in$ G there corresponds an element $\rho_\alpha \in$ Sym(X) defined by ρ_α : x $\rightarrow \alpha$(x), and the map ρ : G $\rightarrow$ Sym(X) defined by $\rho : \alpha \rightarrow \rho_\alpha$ is a homomorphism.

3.2 Symmetry in Model Checking

3.2.1 Symmetry in Kripke Structures

A system contains symmetries if its set of transitions remains invariant when individual or groups of variables are interchanged by certain permutations. For a Kripke structure, $\mathcal{M}$ = (S, s_0, R), a permutation α : S $\rightarrow$ S that maintains the transition relation and initial state is termed an automorphism of $\mathcal{M}$. An automorphism α satisfies the following conditions:

- For all s, t $\in$ S, (s, t) $\in$ R $\Rightarrow$ (α(s), α(t)) $\in$ R.
- $\alpha(s_0) = s_0$.

The set of all automorphisms of $\mathcal{M}$ forms a group, where the operator is a mapping, and is denoted Aut($\mathcal{M}$). For G $\leq$ Aut($\mathcal{M}$), the orbits of S under G can be used to construct a quotient Kripke structure $\mathcal{M}_G$. In the case where G is non trivial, $\mathcal{M}_G$ will be smaller than $\mathcal{M}$.

Definition. Let $\mathcal{M}$ be a Kripke structure, and G an automorphism group of $\mathcal{M}$. The quotient structure [77] $\mathcal{M}_G = (S_G, s^0_G, R_G)$ is defined as:

- $S_G = \{rep_G(s) : s \in S\}$, where $rep_G(s)$ is a unique representative of s^G.
- $s^0_G = rep_G(s_0)$.
- $R_G = \{(rep_G(s), rep_G(t)) : (s, t) \in R\}$.

If a path $\pi = s_0, s_1, \ldots$ is found in $\mathcal{M}$ then $\overline{\pi} = rep_G(s_0), rep_G(s_1), \ldots$ is a path in $\mathcal{M}_G$. Furthermore, a path $\overline{\pi} = \overline{s_0}, \overline{s_1}, \ldots$ in $\mathcal{M}_G$ has a corresponding path in $\mathcal{M}$, $\pi' = s'_0, s'_1, \ldots$

such that $s'_0 = s_0$ and for all $i \geq 1$, $s'_i \in \overline{s}_i^G$. Proofs of these statements have been presented and show a bidirectional correspondence between paths of $\mathcal{M}$ and the quotient structure $\mathcal{M}_G$ for any group of automorphisms G of $\mathcal{M}$ [77].

Therefore, for a model $\mathcal{M}$ and its quotient model $\mathcal{M}_G$ with respect to a group G, $\mathcal{M}, s \models \phi \Leftrightarrow \mathcal{M}_G, rep_G(s) \models \phi$ for every symmetric CTL* formula. A CTL* formula ϕ is symmetric if for each propositional subformula f, and for all $\alpha \in G$, $s \models f \Leftrightarrow \alpha(s) \models f$. It follows that $\mathcal{M} \models \phi \Leftrightarrow \mathcal{M}_G \models \phi$.

3.2.2 Symmetry in Discrete Time Markov Chains

For a DTMC, $\mathcal{D} = (S, s_0, \mathbf{P})$, a permutation $\alpha : S \rightarrow S$ that maintains the transition probabilities and initial state is termed an automorphism of $\mathcal{D}$. An automorphism α satisfies the following conditions:

- For all $s, t \in S$, $\mathbf{P}(s, t) = \mathbf{P}(\alpha(s), \alpha(t))$.
- $\alpha(s_0) = s_0$.

The set of all automorphisms of $\mathcal{D}$ forms a group where the operator is a mapping and is denoted Aut($\mathcal{D}$). For $G \leq Aut(\mathcal{D})$, the orbits of S under G can be used to construct a quotient DTMC $\mathcal{D}_G$

Definition. Let $\mathcal{D}$ be a DTMC, and G be an automorphism group of $\mathcal{D}$. The quotient structure [77] $\mathcal{D}_G = (S_G, s_G^0, \mathbf{P}_G)$ is defined as:

- $S_G = \{rep_G(s) : s \in S\}$, where $rep_G(s)$ is a unique representative of s^G.
- $s_G^0 = rep_G(s_0)$.
- $\mathbf{P}_G(rep_G(s), rep_G(t)) = \sum_{x \in t^G} \mathbf{P}(rep_G(s), x)$.

If a path $\pi = s_0, s_1, \ldots$ is found in $\mathcal{D}$ then $\overline{\pi} = rep_G(s_0), rep_G(s_1), \ldots$ is a path in $\mathcal{D}_G$. Furthermore, a path $\overline{\pi} = \overline{s_0}, \overline{s_1}, \ldots$ in $\mathcal{D}_G$ has a corresponding path in $\mathcal{D}$, $\pi' = s'_0, s'_1, \ldots$ such that $s'_0 = s_0$ and for all $i \geq 1$, $s'_i \in \overline{s}_i^G$. Proofs of these statements have been presented and show a bidirectional correspondence between paths of $\mathcal{D}$ and the quotient structure $\mathcal{D}_G$ for any group of automorphisms G of $\mathcal{D}$ [77].

Therefore for a model $\mathcal{D}$ and its quotient model $\mathcal{D}_G$ with respect to a group G, $\mathcal{D}, s \models \phi \Leftrightarrow \mathcal{D}_G, rep_G(s) \models \phi$ for every symmetric PCTL formula. A PCTL formula ϕ is symmetric if for each propositional subformula f, and for all $\alpha \in G$, $s \models f \Leftrightarrow \alpha(s) \models f$. It follows that $\mathcal{D} \models \phi \Leftrightarrow \mathcal{D}_G \models \phi$.

3.2.3 Symmetry in Markov Decision Process

For an MDP, $\mathcal{M}$ = (S, s_0, Steps), a permutation $\alpha : S \to S$ that maintains the transition probabilities and initial states is termed an automorphism of $\mathcal{M}$. An automorphism α satisfies the following conditions:

- For all s, t $\in$ S for which there exists a $\mu \in$ Steps and $\mu(t) > 0$, there exists $\mu' \in$ (Steps(α(s))) such that $\mu'(\alpha(t)) = \mu(t)$.
- $\alpha(s_0) = s_0$.

The set of all automorphisms of $\mathcal{M}$ forms a group where the operator is a mapping and is denoted Aut($\mathcal{M}$). For G $\leq$ Aut($\mathcal{M}$), the orbits of S under G can be used to construct a quotient MDP $\mathcal{M}_G$.

Definition. Let $\mathcal{M}$ be a MDP, and G be an automorphism group of $\mathcal{M}$. The quotient structure [77] $\mathcal{M}_G$ = (S_G, s^0_G, $Steps_G$) is defined as:

- $S_G = \{rep_G(s) : s \in S\}$, where $rep_G(s)$ is a unique representative of s^G .
- $S^0_G = rep_G(s_0)$.
- for each $rep_G(s) \in S_G$ and $\mu \in$ Steps($rep_G(s)$), $Steps_G(rep_G(s))$ contains a distribution $\overline{\mu} \in$ Dist(S_G) where, for $rep_G(t) \in S_G$, $\overline{\mu}(rep_G(t)) = \sum_{x \in t^G} \mu(x)$.

If a path $\pi = s_0 \xrightarrow{\mu_1} s_1, \xrightarrow{\mu_2} \ldots$ is found in $\mathcal{M}$ then $\overline{\pi} = rep_G(s_0) \xrightarrow{\overline{\mu}_1} rep_G(s_1) \xrightarrow{\overline{\mu}_2} \ldots$ is a path in $\mathcal{M}_G$. Furthermore, a path $\overline{\pi} = \bar{s}_0 \xrightarrow{\overline{\mu}_1} \bar{s}_1 \xrightarrow{\overline{\mu}_2} \ldots$ has a corresponding path in $\mathcal{M}$, $\pi' = s'_0 \xrightarrow{\mu'_1} s'_1 \xrightarrow{\mu'_2} \ldots$ such that $s'_0 = s_0$, $s'_i \in \bar{s}_i^G$. Proofs of these statements have been presented and show a bidirectional correspondence between paths of $\mathcal{M}$ and the quotient structure $\mathcal{M}_G$ for any group of automorphisms G of $\mathcal{M}$ [77].

Therefore for a model $\mathcal{M}$ and its quotient model $\mathcal{M}_G$ with respect to a group G, M, s $\models \phi \Leftrightarrow$ M_G, $rep_G(s) \models \phi$ for every symmetric PCTL formula. It follows that $\mathcal{M} \models \phi \Leftrightarrow M_G \models \phi$.

3.3 Symmetry Reduction in Practice

The key idea of symmetry reduction is to exploit the underlying structure of the transition system in an attempt to reduce the size of the state space. The state space exploration algorithm, outlined in Figure 2.6, can be adapted to exploit equivalence between states. The adapted algorithm, presented in Figure 3.1, allows a quotient state space to be incrementally constructed if symmetries are known in advance. The major benefit of this approach is that the quotient structure may be built even though the original structure is too large to be explored.

```
1.   reached := unexplored := {rep(s_0)}
2.   While unexplored ≠ ∅
3.   {
4.       remove a state s from unexplored
5.       for each transition s → s'
6.       {
7.           if s' = error
8.           {
9.               stop and report error
10.          }
11.          if rep(s') ∉ reached
12.          {
13.              add rep(s') to reached and unexplored
14.          }
15.      }
16.  }
```

Figure 3.1: Standard algorithm for explicit state space exploration with symmetry reduction.

The algorithm begins by removing the initial state from unexplored and determining its successors. For each successor state the algorithm computes the representative and checks if it has been encountered before. In the case where the representative state has not been encountered it is added to the unexplored stack and reached hash table. Once the status of all successor states has been determined the algorithm obtains the next state from the unexplored stack. Once the unexplored stack is empty the state space has been fully explored.

This approach to the quotient structure construction poses two major questions:

1. How is symmetry identified?
2. How are equivalent states reduced to a representative state?

3.3.1 Identifying Symmetry

A simple method for the identification of symmetry is to take a model specification and construct its Kripke structure $\mathcal{M}$. By subjecting $\mathcal{M}$ to a standard symmetry detection algorithm [27], the resulting information can be used in the construction of the quotient structure. This naive approach has two flaws;

1. The state space explosion problem is the primary motivation for research into reduction techniques. Therefore, the need to fully construct the state space in order to run the algorithm is self defeating. If a model can be constructed, it is reasonable to assume the state space is tractable and no reduction is required. Furthermore, once the state space

becomes intractable, symmetries can no longer be detected and a reduced structure cannot be constructed.

2. The identification of automorphisms in a Kripke structure is as hard as solving the graph isomorphism problem [81]. For a large state space this is a time consuming processes.

To exploit the possible benefits of symmetry reduction other methods must be considered.

3.3.2 User Specification of Symmetry

The problems of the naive approach can be side-stepped by placing the burden of symmetry detection on the user. The manual specification of symmetry groups requires no overhead for detection and affords the specification of groups that would be potentially missed by the detection algorithm. However, as with all manual specification techniques, this methodology is prone to user error and requires expert knowledge of the domain.

A solution that requires no upfront detection or expert domain knowledge is to restrict the specification language in a way that guarantees all generated models are symmetric. This approach is taken by the symmetry based model checker, SMC [91]. All specifications are required to be fully symmetric and this is achieved by forcing behaviour to be defined in a single process that can in turn be instantiated multiple times.

Similarly, the Symmetric Probabilistic Specication Language (SPSL) [33] defines a subset of the PRISM language designed to guarantee applicability of the generic representatives approach (see Section 3.5). SPSL restrict specification to containing families of symmetric processes whose behaviour can be defined using multiple local variables. The GRIP tool [32] translates SPSL specifications into a generic reduced form, the semantics of which are isomorphic to the original specications symmetrically reduced model. Therefore, the generic reduced specification can be used to model check symmetric properties of the original model.

3.3.3 The Scalarset Approach

An alternative approach to language restriction is the annotation of the system specification with a data type known as a Scalarset [81]. A Scalarset acts as a documentation of the symmetry present in the model and loosens the requirement on the system being fully symmetric.

Definition. A scalarset is an integer sub-range with the following set of restricted operations:

1. A scalarset term is always a variable reference.

2. Two scalarset variables of the same type can only be compared using the = operator.
3. A scalarset variable can only be assigned a value from another scalarset variable of the same type.
4. If a scalarset variable is used in the index of a for statement, the result must not be affected by the order of iteration execution.
5. An array with a scalarset index type can only be manipulated by a scalarset variable of the same type.

This definition ensures that any values defined using the scalarset datatype can be consistently permuted in all states in the state-space. Additionally, the definition is given in terms of a generic language and can be adapted to any specific setting. The original implementation was provided in the Murϕ description language [29], however, scalarsets have been defined and extended for the Promela specification language [13]. These extensions are able to handle queues as well as allowing multiple scalar sets to be defined in a single specification.

A major benefit of the scalarset data types are that violations to the restricted set of operations can be detected in polynomial time [13]. However, the definition only allows the description of systems that exhibit total symmetries. For example, the data type could be utilised in the description of a system of processes connected as a clique but not as a ring or tree. Extensions to the core datatype have been proposed as a means of specifying systems with ring structures, however, these have not been implemented. The scalarset datatype and extensions all share the same downside. Inherently the user must identify symmetry in the model and select an appropriate data type to specify their presence. This means that symmetry reduction using scalarsets is not a "push button" reduction technique.

3.3.4 Automated Symmetry Detection

Consider a model of a system $\mathcal{M}$ that consists of a finite number of parallel executing processes, identical up to renaming and communicating via shared variables. A subgroup of Aut($\mathcal{M}$) can be automatically calculated from the communication structure of the program [47]. The automatic computation of the subgroup is feasible as it is usually small compared to Aut($\mathcal{M}$).

To construct the communication structure, let I be a set of finite numbers, $\{0, 1, 2, \ldots n\}$, representing process identifiers. For a system specification $P = \|_{i \in I} p_i$, the communication structure is an undirected graph CS = (I, E), where $\{i, j\} \in E$ if and only if processes p_i and p_j share a variable. Originally this approach only applied to systems where variables were

shared by at most two processes of the same type. However, this restriction has since been lifted by considering the coloured hypergraph of a shared variable program [47].

In the message-passing paradigm, where processes communicate by sending messages across a data type known as a channel, structural symmetries of a model can be automatically extracted from program text in the form of a static channel diagram [40]. The static channel diagram can be thought of as a static approximation of a specification's communication structure. Communication arising from the dynamic passing of channel references is not considered. Furthermore, any edge in the diagram may be the direct result of updates which are not executable in the final model. Currently this approach is implemented in SymmExtractor [30], an automatic symmetry detection tool for the Promela [58] specification language.

SymmExtractor takes a Promela specification as input and by analysing a subsequently constructed abstract syntax tree, generates a static channel diagram. This diagram provides input to the Saucy [27] program, deriving its automorphisms. These generators are checked individually against the specification to see if they induce valid automorphisms of the associated model. Starting with the set of candidate generators which are valid, the largest possible subgroup of candidate symmetries which are all valid is computed. In the worst case this can be an algorithmically expensive operation.

This approach has the benefit of being able to detect arbitrary component symmetries arising from the communication structure of a specification. The only requirement is that the specification satisfies certain restrictions that can be automatically checked, and are less strict than those imposed by the scalarset data type or the SMC input language [91].

3.4 Exploiting Symmetry

The second problem is once symmetries have been identified, how are they then utilised to check if the current state is equivalent to one already encountered. In general this involves searching for a canonical state representation of the current state. This is known as the constructive orbit problem and has been shown to be at least as hard as testing for graph isomorphism for which currently no polynomial algorithms are known [47]. Furthermore, this operation must be performed for every state encountered during exploration.

Definition. Let G be a group acting on the set $\{1, 2, \ldots, n\}$. For two vectors $x, y \in \mathbb{Z}^n$ the orbit problem is the process of determining if there exists a permutation $\alpha \in G$ such that $y = \alpha(x)$.

Despite these problems the orbit problem can be efficiently solved for certain symmetry

groups [78] and directly avoided through the use of heuristics suited to the graph isomorphism problem [26]. A further way to alleviate the orbit problem is to lift the restriction on mapping all equivalent states to a single representative. However, this requires a delicate balance between speed and storage requirements.

3.4.1 Easy Classes of Symmetry

For the following classes of automorphism group G the orbit problem can be solved in polynomial time [20] where n denoted the number of processes;

- A group whose order is polynomial in n. The representative state can be computed by enumerating the orbits of the state. Examples include cyclic or dihedral groups
- The symmetric group S_n - a representative state in the form of the lexicographically smallest element of the orbit can be obtained by sorting the state-vector.
- A group that is the disjoint product or wreath product of groups that themselves are solvable in polynomial time. The representative can be found by solving the orbit problem independently for each subgroup.
- A group generated by transpositions.

3.4.2 Multiple Representatives Approach

The requirement that every element of a given orbit s^G is mapped to a single representative ensures that symmetry reduction is optimal in terms of space. While permitting multiple representatives per orbit may diminish potential reduction, it greatly reduces the complexity involved in calculating a representative state [13]. This relaxation creates a quotient structure that captures all system behaviours and is therefore a sound reduction technique. Therefore, as long as the set of representatives remains small this approach to symmetry reduction is viable.

By allowing multiple representatives, the selection of the minimal lexicographical representative of a state is no longer appropriate. Instead a representative function is chosen by a normalisation function which maps all states to states no larger than themselves. A good normalisation function is defined as one that maps a state to the minimum or close to the minimal orbit representative. This normalisation function provides an approximate solution to the orbit problem.

3.4.3 Strategies for Symmetry Reduction

The simplest approach to calculating a representative state in an orbit is to construct all states in the orbit and select the lexicographical minimum. If the group is small then this is a feasible strategy and provides an optimal symmetry reduction strategy. The Symmetric Spin [13] package provides an enumeration strategy, however it optimises this approach by generating permutations incrementally by composing successive transpositions. The enumeration strategy has also been generalised to apply to arbitrary groups using stabiliser chains [36]. The use of stabiliser chains enables faster calculation of α(s) and only requires the storage of coset representatives. A variation on this strategy can be employed when a model checker explores the state space using a depth-first search algorithm. Instead of calculating the lexicographical minimum to be the representative the first element of an orbit encountered during search is chosen [91].

In fully symmetric systems the minimum state representative can be easily obtained by sorting the tuple lexicographically. Unfortunately, for some commonly occurring symmetry groups this simple sorting strategy is not applicable. For example, in a client server specification a group may permute server components along with their associated blocks of client components. While permutations of server components are isomorphic to the symmetric group S_n, a minimal ordering cannot be obtained by simply sorting the vector [47]. To solve this problem a minimising set X that can be obtained from a larger group G is defined. The minimal representative can be simply calculated by iterating over X until a fixed-point is reached. This approach has been seen to be viable for a large class of groups which are isomorphic to S_n.

Finally, certain kinds of symmetry groups can be decomposed as a product of subgroups. For certain decompositions the orbit problem can be solved separately for each subgroup the solutions being combined to provide a solution for the whole group. In the instance where a group permutes disjoint sets of components the group can be described as the disjoint product of the groups. If the symmetry group partitions the components into subsets for which there is analogous symmetry, and symmetry between the subsets, then the group can be described as the wreath product of the group. In [39] techniques capable of detecting, before search, whether G can be decomposed have been presented and their viability shown through implementation in the TopSpin tool.

If G is a large group and the strategies above prove infeasible an approximate symmetry reduction strategy must be deployed. One approach to providing an approximate solution is to split the state vector into two parts. Representatives of an orbit can be obtained by lexicographically sorting the leftmost part of the vector relative to the splitting point. [13].

The trade off between speed and reduction can be tuned by varying the split point. Approximate solutions have also been provided through the use of heuristics such as hill-climbing local search and their viability shown through the exploration of state spaces associated with various configurations of a hypercube network [36].

3.5 Combining Symmetry Reduction with Symbolic Representation

So far, all discussion on the application of symmetry have dealt with an explicit representation scheme. This is due to the inherent problems of combining symmetry reduction with symbolic storage schemes. When using BDDs as a data structure, checking state equivalence requires an increase in memory footprint. To implement equivalence checking symbolically, a propositional formula must be defined that detects whether two arguments are symmetry-equivalent. This formula has the form $f(s_1, \ldots, s_n, s'_1, \ldots, s'_n)$ and evaluates to true if the vector $(s_1, \ldots, s_n)$ is a permutation of the vector $(s'_1, \ldots, s'_n)$. For many symmetry groups that commonly occur in model checking, the BDD of formula f is intractable in terms of size [21].

One approach to alleviating this problem is to allow multiple representative states from each orbit. To constrain the size increase of the model, representatives are selected based on a specific subset of automorphisms [44]. However, in practice allowing multiple representatives still produces models of intractable size. An approach that directly avoids construction of the orbit relation involves determining orbit representatives dynamically during fixed point iterations. This is achieved by computing transition images from the unreduced structure and mapping the new states to their representatives. However, the dynamic calculation of a representative state is computationally infeasible for certain specification types [45].

When a specification defines a fully symmetric system, generic representatives [44] can be used to avoid construction of the orbit relation by translating the specification into a reduced specification. A generic representative indicates how many processes are in each local state. For example, in a mutual exclusion specification with three processes the states (N, N, T), (T, N, N) and (N, T, N) are all equivalent and expressed by the generic representative (2N 1T). The semantics of the translated specification, that now uses sets of counters to generically represent the state of processes, are isomorphic to the original specification symmetrically reduced model. Therefore, the generic reduced specification can be used to model check symmetric properties of the original model.

3.6 Exploiting Symmetry in Less Symmetric Systems

In practice many systems are comprised of a set of similar but not identical processes, the condition α(R) = R is not satisfied for all process permutations. Therefore, a partial symmetry system can be defined as, for most transitions r $\in$ R and most permutations α, the condition α(r) $\in$ R holds. An example of a partially symmetric system is the readers-writers problem [94], where reader and writer processes access a shared resource. Asymmetry is introduced into the system as a writer process always has priority over a reader when both are trying to access the shared resource. Therefore, readers can be permuted, writers can be permuted, but readers cannot be interchanged with writers. However, the state graph is symmetric in every sense except for transitions from a state where two processes are attempting to access the shared resource. To exploit similarity in partially symmetric systems different classes of symmetry have been defined, these include near or rough symmetry [43] and virtual symmetry [46].

In the case of near symmetry, let $\mathcal{M}$ be a system model and $\mathcal{I}$ its associated set of process identifiers. A permutation $\alpha \in$ Sym(I) is defined to be a near automorphism if, for every transition s $\rightarrow$ t in $\mathcal{M}$, either α(s) $\rightarrow$ α(t) is a transition in $\mathcal{M}$ or s is totally symmetric with respect to Aut($\mathcal{M}$). System $\mathcal{M}$ is nearly symmetric if a group of near automorphisms G_n can be identified. In the case where G_r is a subgroup of Sym(I), $\mathcal{M}$ can be considered roughly symmetric with respect to G_r, if for states s and s$'$ in the same orbit, any transition from s is matched by a transition from s$'$ where the transition is initiated by a process with a higher priority. Finally, if $\mathcal{M}$ is a nearly (roughly) symmetric model with respect to group G_n(G_r) then symmetry reduction with respect to G_n(G_r) preserves all symmetric CTL properties [43]. In the case of both near and rough symmetry it is unclear how to verify it on a high-level system description

Virtual symmetry [46] subsumes the notion of both near and rough symmetry. Where rough symmetries allow the specification of systems with static priorities, virtual symmetry affords the specification of systems where resources are shared according to dynamic priorities. Using the terminology of [46] the symmetrisation R^G of a transition relation R by a group G is defined by: R^G = $\{\alpha(s) \rightarrow \alpha(t) : \alpha \in G$ and s $\rightarrow \in$ R. Symmetrising a transition relation involves adding the transitions missing due to asymmetry present in the system. A structure $\mathcal{M}$ is therefore said to be virtually symmetric with respect to a group G_v acting on S if for any s $\rightarrow$ t $\in R^{G_v}$, there exists $\alpha \in G_v$ such that s $\rightarrow \alpha$(t) $\in$ R.

An additional approach to providing the reduction of a partially symmetric system is to annotate each state with information about whether and how symmetry is violated along its path. More precisely, the annotation is a partition of the set of all component indices: if the

path to the state contains a transition that distinguishes two components, their indices are put into different partition cells. Only components in the same cell can be permuted during future explorations from the state. An algorithm that adapts to this state information has been defined and produced a quotient structure that is not bisimulation equivalent to the original. This allows the analysis of systems with respect to safety properties.

3.7 Tools for Symmetry Reduction

As previously mentioned the Murϕ specification language [29] provided the first definition and implementation of the scalarset data type. From the scalarset data type the automorphism group for the state space is determined and the lexicographically smallest member of each orbit is used as the representative. The viability of Murϕ has been used to verify a number of highly symmetric algorithms including Peterson's n-process mutual exclusion algorithm and a lock implementation for the Stanford DASH multiprocessor.

Continuing with explicit state implementations the symmetry based model checker SMC [91] was created specifically for the specification and verification of highly symmetric systems. Symmetry is easily detectable due to input language restriction and the first state of an orbit encountered during search is selected as the representative. SMC has the major advantage of being the only model checker that can be used to effectively verify liveness properties under both strong and weak fairness assumptions. This is achieved by annotating the quotient structures with additional information that not only allows the original structure M to be retrieved from the quotient structure, but it is also possible to check properties expressed in indexed CTL.

Symmetric SPIN [13] is a tool that brings symmetry reduction to the popular SPIN model checker via the scalarset data type. Symmetric SPIN avoids direct modification of the Promela language with the scalarset data type and requires all information to be outlined in a separate user generated file. A script is then used to modify the generated verifier adding a representative function that computes a lexicographical minimum representative via a canonicalisation function or returns an approximate minimal representation via a normalisation function. Experimental results [13] have shown that for certain models the factor of reduction gained are close to the theoretical limit. Furthermore, it has been shown that symmetry can be used in conjunction with the partial-order reduction.

TopSPIN [39]provides another symmetry reduction implementation to SPIN but differs from other approaches by providing a means for automated symmetry detection. This is implemented through the extraction of static channel diagrams directly from the specification. Ef-

ficient reduction is achieved by the provision of four strategies termed: enumeration, local-search, fast, and segmented. The selection of algorithm is based on the information provided by static channel diagram analysis. A current limitation of TopSPIN is its restriction to verification of assertions, LTL properties cannot presently be verified.

The symbolic model checker, SMV [75] has received an implementation of symmetry reduction via the use of scalarsets. In addition, temporal case splitting is used to break a given property down into a parameterised set of assertions in an attempt to avert the problems inherent with the combination of symmetry and symbolic storage schemes. However, this approach has its own problems, termed the case explosion. By declaring variables as scalarsets, assertions can be sorted into equivalence classes and it can be shown that it is only necessary to check a representative subset of assertions. Virtual symmetry has been successfully combined with the generic representatives approach for the case where processes are fully interchangeable with respect to virtual symmetry. This allows symmetry-reduced symbolic model checking of partially symmetric systems, using the NuSMV [18] model checker. The question of whether virtual symmetry can be verified efficiently is still an open question as it seem to incur a cost proportional to the size of the unreduced Kripke structure

SYMM [20] is a symbolic model checker constructed with the intention of exploring symmetry reduction. SYMM utilises a small and simple specification language based on a shared variable model of computation and allows the verification of CTL properties. Symmetries are required to be input by the user and the explosion problem is avoided by allowing multiple orbit representatives approach. SYMM has been used to verify the IEEE Futurebus arbiter protocol [20].

3.7.1 Symmetry Reduction for Probabilistic Model Checking

The PRISM-symm tool [68] has recently been integrated into the PRISM model checker [69]. This provides PRISM with inbuilt symmetry reduction capabilities, which are implemented using ideas from dynamic symmetry reduction [45]. To operate, the tool requires users to specify the number of modules that appear before and after a block of symmetric modules. Therefore, reduction can only be provided when full symmetry is present between a series of modules. Furthermore, PRISM does not check if the provided symmetries are correct and consequently a degree of expert knowledge is required.

Case studies conducted using PRISM-symm have shown that a substantial decrease in the number of reachable states can be achieved. However, in one instance the size of the MTBDD was shown to increase by a factor of ten, but in other experiments it decreased by a factor of more than two. Nevertheless, the results show that symmetry reduction can be effectively

applied to probabilistic model checking. When considering highly symmetric systems, the benefits of the reduced reachable states often outweigh any downside of the larger MTBDD.

PRISM can also be used to provide symmetry reduction by means of the GRIP tool [32] (Generic Representatives in PRISM). The tool translates a restricted subset of the PRISM language, called Symmetric Probabilistic Specification Language [33] to a reduced counter-abstract form that can be subjected to analysis by PRISM. Case studies conducted using GRIP have also yielded a substantial decrease in the number of reachable states. However, as with PRISM-symm both smaller and larger MTBDD sizes have been observed.

When directly compared, GRIP is typically faster for models that contain a large number of simple modules, whereas PRISM-symm performs better on models constructed from a small number of more complex modules [33]. Furthermore, while GRIP only operates on a subset of the PRISM language, and therefore cannot be applied to all of PRISM's features, any symmetry reduction provided by the tool is known to be correct. Therefore, these approaches have shown that symmetry reduction can be successfully applied to probabilistic model checking. However, both techniques are restricted to operating on symmetric systems and GRIP is further restricted to a operating on a subset of the PRISM language.

Finally, no research on the application of symmetry reduction to probabilistic explicit state model checking could be identified. However, tools such as TopSPIN [39] have shown that symmetry reduction can be effectively applied to systems that exhibit arbitrary component symmetries. Therefore, the application of symmetry reduction to explicit state probabilistic model checking, may not require the types of restrictions imposed by GRIP and PRISM-symm.

3.8 Summary

If a concurrent system is comprised of many replicated processes then checking a model of the system may involve redundant search over equivalent, or symmetric, areas of the state space. Symmetry reduction is concerned with exploiting these underlying regularities by only storing one representative of a structure. For highly symmetric systems, this can result in a reduction factor exponential to the number of system components.

We have given an overview of symmetry reduction techniques for model checking and of the currently available tools. The survey clearly identifies a lack of research into the application of symmetry reduction techniques to explicit state probabilistic model checking. This issue is the focus of the remainder of the thesis.

CHAPTER 4

Probabilistic Symmetric Systems Language

This chapter introduces a new probabilistic specification language: Probabilistic Symmetric Systems Language, henceforth referred to as PSS. The motivation for defining a new specification language was driven by the absence of an existing language that meets our specific requirements. Nevertheless, PSS is influenced by the PRISM [69] and ProbMela [8] specification languages. It draws upon their syntax and language features to allow the specification of probabilistic models that are naturally compatible with language level symmetry detection techniques.

PSS is a small language and shares the following common features with ProbMela: parameterised processes, channels, arrays, reference types, global and local variables. However it does not have some of the language features ProbMela inherited from Promela [58] such as enumerated types and user defined record types [8]. ProbMela was not considered an appropriate language for our purposes as its large set of language features would make the rigorous proof of any symmetry detection technique infeasible. A potential solution would be to restrict and consider a subset of ProbMela, an approach mirroring the definition of the Promela-lite [37] specification language. However, the result of that would be a non-intuitive specification language that hides information required by the symmetry detection technique proposed in Chapter 5. For example, the type of a channel cannot be readily determined from its declaration [34].

While PSS is similar to the PRISM language, it possesses additional features such as the

previously mentioned channel and reference types. These data types are included as they have been used in a previous approach to symmetry detection that was capable of capturing arbitrary components symmetries [37]. As this thesis focuses on the application of symmetry reduction to explicit state probabilistic model checking, the ability to capture arbitrary symmetry groups is desirable. Therefore, PSS is designed to include language features used in the previous approach, as it will serve as a basis for our own symmetry detection techniques.

Furthermore, the Symmetric Probabilistic Specification Language [33], a subset of PRISM defined to guarantee application of a generic representatives approach, is not appropriate for our needs. The language is restricted to defining specifications that consist of multiple families of identical processes. Therefore, specifications will not contain the more complex forms of symmetries we desire to capture and exploit.

Section 4.1 provides an informal introduction to PSS by means of example, followed by the formal definition of PSS grammar in Section 4.2. The chapter concludes in Section 4.3 with the semantics of PSS that define how a DTMC or MDP is constructed from a PSS specification.

4.1 Informal Introduction to PSS

The major elements of a PSS specification are processes and variables, with every specification containing a set of global variables and processes. In turn, each process possesses a unique set of local variables that cannot be modified or read by another process. The state of a process is determined by the current value of its local variables and the state of the model is determined by the current value of global variables in conjunction with the state of the processes.

The set of behaviours a model associated with a specification may exhibit is defined using a set of commands located in every process. Each command consists of a guard and a set of updates. The guard determines if the model is in an appropriate state for its updates to be executed. An execution takes the form of a modification to at least one local or global variable. By definition this transitions the model into a new state.

In the instance where multiple updates are available, only one is selected and executed to create a new state. The probability with which an update is selected is given in the PSS specification. Using a number of small examples, PSS language features will be presented in the following sections.

4.1.1 Six Sided Die Example

The PSS code in Figure 4.1 defines the 13 state DTMC that models the behaviour of a six sided die using fair coins. In a PSS specification, a process contains the declaration of a local variable set. A variable declaration consists of a type, name and initial value. This example involves two variables, state and die of type integer initialised to the value 0. After the local variable definitions is a set of guarded commands with the form:

$$\text{guard} \rightarrow p_1 : u_1 + p_2 : u_2 + \ldots + p_n : u_n;$$

where the p_i are probabilities and the u_i are updates, $1 \leq i \leq n$. This form of command directly follows the style of the PRISM specification language.

```
1.  dtmc
2.
3.  Process die()
4.  {
5.    int state := 0;
6.    int die := 0;
7.
8.    state == 0 → 0.5 : (state := 1) + 0.5 : (state := 2);
9.    state == 1 → 0.5 : (state := 3) + 0.5 : (state := 4);
10.   state == 2 → 0.5 : (state := 5) + 0.5 : (state := 6);
11.   state == 3 → 0.5 : (state := 1) + 0.5 : (state := 7; dice := 1);
12.   state == 4 → 0.5 : (state := 7; die := 2) + 0.5 : (state := 7; die := 3);
13.   state == 5 → 0.5 : (state := 7; die := 4) + 0.5 : (state := 7; die := 5);
14.   state == 6 → 0.5 : (state := 2) + 0.5 : (state := 7; die := 6);
15.   state == 7 → (state := 7);
16. }
```

Figure 4.1: PSS code modelling the behaviour of a six sided die using fair coins [1].

A guard is a Boolean function of global and local variables. As line 1 of the PSS specification in Figure 4.1 defines the model as a DTMC, all guards listed with the process must be disjoint. Command updates describes changes that can occur to this set of global or local variables, resulting in a transition between states. The first command in this PSS specification is given on line 8 and describes the behaviours **Process** die() may exhibit when the variable state has value 0.

$$\text{state} == 0 \rightarrow 0.5 : (\text{state} := 1) + 0.5 : (\text{state} := 2);$$

The possible behaviours given by the two updates are that the variable state become equal to 1 or 2. In a command, each unique and independent u_i is surrounded by brackets and separated by the + character. When a choice is available between updates each is assigned a probability that is required to sum to one. Therefore, when the integer variable state takes the value 0, executing the guard will change its value to 1 with probability 0.5, or 2 with probability 0.5.

Line 14 provides an example of a command consisting of a single update. In this instance the probability is omitted and is assumed to equal 1.

$$\text{state} == 7 \rightarrow (\text{state} := 7);$$

Furthermore, an update can change the value of more than one variable. In fact, an update can change the value of all global variables and variables local to the current process. The second update of the command given on line 12 gives an example in which the value of state is changed to 7 and die to 1 in a single step. To indicate that updates are a single action they are enclosed in brackets and separated by a semi colon.

$$\text{s} == 3 \rightarrow 0.5 : (\text{state} := 1) + 0.5 : (\text{state} := 7; \text{dice} := 1);$$

4.1.2 Simple Mutual Example

The PSS code in Figure 4.2 models the behaviour of a simple mutual exclusion problem. Although defining a different type of model, the layout of the specification is similar to that given in Figure 4.1. The PSS specification begins with the **mdp** declaration immediately followed by the declarations of a global variable set, containing the integer variables x and y both initialised to 0, and two processes with no local variables.

Lines 9 and 16 give the first example of a guard whose boolean function accepts more then one variable as input.

$$\text{x} == 1 \wedge \text{y} \neq 2 \rightarrow (\text{x} := 2);$$
$$\text{y} == 1 \wedge \text{x} \neq 2 \rightarrow (\text{y} := 2);$$

PSS allows a range of simple propositions to be defined using $<$, $\leq$, $\neg$, ==, $\geq$, $>$ and combined using the operators $\wedge$ and $\vee$.

Similarly, updates can make use of arithmetic expressions that combine operators from the set $\{+, -, \star, \div\}$, global variables and local variables from the same process. The second

```
1.  mdp
2.
3.  int x := 0;
4.  int y := 0;
5.
6.  Process mexc1()
7.  {
8.    x == 0 → 0.8 : (x := 0) + 0.2 : (x := x + 1);
9.    x == 1 ∧ y ≠ 2 → (x := 2);
10.   x == 2 → 0.5 : (x := 2) + 0.5 : (x := x - 2);
11. }
12
13. Process mexc2()
14. {
15.   y == 0 → 0.8 : (y := 0) + 0.2 : (y := y + 1);
16.   y == 1 ∧ x ≠ 2 → (y := 2);
17.   y == 2 → 0.5 : (y := 2) + 0.5 : (y := y - 2);
18. }
```

Figure 4.2: PSS code modelling the behaviour of a mutual exclusion problem [3].

update of the command given on line 17 is an example of an arithmetic operation being used in an update.

$$y == 2 \rightarrow 0.5 : (y := 2) + 0.5 : (y := y - 2);$$

The specification is for an MDP, as seen in line 1. In every state a single process must be non-deterministically selected to execute a command. When global variables x and y both equal 0, if **Process** mexc1() is selected the command given on line 8 will be executed and x will be set to 1 with probability 0.8 or 0 with probability 0.2. On the other hand if **Process** mexc2() is selected the command given on line 15 will be executed and y will be set to 1 with probability 0.8 or 0 with probability 0.2. This PSS specification defines a non-deterministic choice between two probability distributions when both variables x and y are equal to 0. This non-deterministic choice is illustrated in the partial MDP depicted in Figure 4.3.

As in the PRISM specification language [69] PSS supports local nondeterminism. In this case the requirement for all guards to be disjoint when specifying a DTMC is dropped. In the example, suppose the command set of **Process** mexc1() is extended to include

$$x == 0 \rightarrow 0.5 : (x := 1) + 0.5 : (y := 2);$$

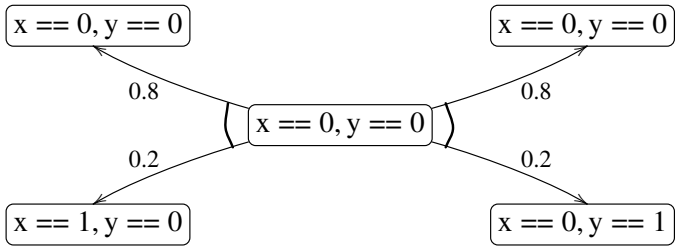


Figure 4.3: Non-deterministic choice between two probability distributions when x == 0 and y == 0.

This PSS specification now contains a non-deterministic choice between three probability distributions when both variables x and y are equal to 0. These are the the two previous distributions and a new distribution in which x is set to 1 with probability 0.5 and y is set to 2 with probability 0.5. This three way non-deterministic choice is illustrated in the partial MDP depicted in Figure 4.4.

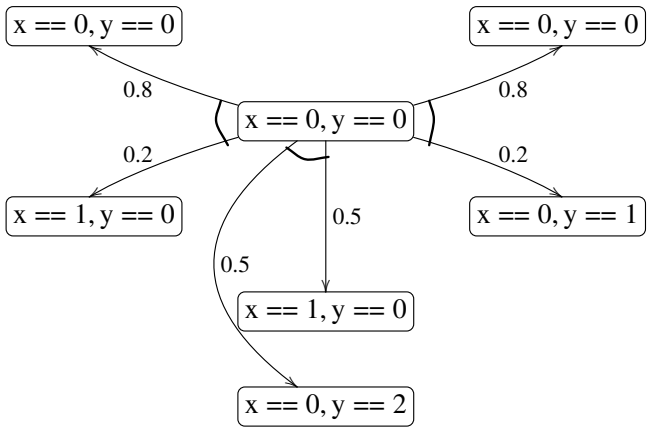


Figure 4.4: Non-deterministic choice between three probability distributions when x == 0 and y == 0.

4.1.3 A Peer to Peer Network Example

To illustrate additional features of PSS we give a simple peer to peer (p2p) network specified in PSS, see Figure 4.5. The specification defines 3 client processes that share a single transfer medium and introduces 4 further language features:

- **Personal Identification Types**: - Variables declared with the type pid provide a natural means of referencing other processes in the specification. To accommodate referencing, every process contains a predefined local variable of type pid, called _pid that is

```
1.  mdp
2.
3.  chan medium [3] of {pid, pid};
4.  int[3] full;
5.
6.  Process client(chan in)
7.  {
8.    int[3] message;
9.    int from;
10.   int to;
11.
12.   len(in) > 0 → (in?from,to);
13.   too == _pid → (message[from] := 1);
14.   _pid == 0 ∧ full[0] == 0 ∧ full[1] == 0 ∧ full[2] == 0 → 0.5 : (in!1,_pid) +
                                                                0.5 : (in!2,_pid);
15.   _pid == 1 ∧ full[0] == 0 ∧ full[1] == 0 ∧ full[2] == 0 → 0.5 : (in!0,_pid) +
                                                                0.5:(in!2,_pid);
16.   _pid == 2 ∧ full[0] == 0 ∧ full[1] == 0 ∧ full[2] == 0 → 0.5 : (in!0,_pid) +
                                                                0.5 : (in!1,_pid);
17.   message[0] + message[1] + message[2] == 3 → (full[_pid] := 1);
18. }
19.
20. Initial{client(medium); client(medium); client(medium);}
```

Figure 4.5: PSS code modelling the behaviour of a simple peer to peer network.

assigned a unique numerical value. The guard of the command given on line 13 of the specification illustrates this feature

$$\text{to} == \_\text{pid} \rightarrow (\text{message[from]} := 1);$$

to determine whether the current value of the variable to is equal to the process's unique numerical reference.

- **Arrays**: - Line 4 of the specification shows the declaration of an array containing three variables of type integer.

int[3] message;

As in most languages, PSS arrays are indexed from 0 with the number of elements in an array specified at declaration with an integer constant. The language provides a

simple and straightforward way to initialise arrays at declaration time by enclosing the initial values in curly braces { }. The following command shows how to initialise an array with message[0] == 1, message[1] == 2 and message[2] == 5.

int[3] message = {1, 2, 5};

If no initialising values are provided at the time of declaration, the variables contained within the array automatically default to the initial value associated with their type. In the p2p specification no initialising value is specified for the message[] array, consequently the three integer variables it contains are initialised with the default value 0.

An example of the access and update of array elements is provided by the command given on line 17. The guard of the command accesses every element in the message array to determine if their accumulated value is equal to 3. If this is true, the process's unique pid value is used to index into a second array called full[] and set the value of the variable at at this location to 1.

message[0] + message[1] + message[2] == 3 → (full[_pid] := 1);

- **Channel Types**: - Channels provide a natural means of modelling inter process communication. A channel acts as a first-in first-out queue accepting messages of a specific format. On line 3 a channel is constructed with three slots that accept messages consisting of two fields of type pid.

chan medium [3] of {pid, pid};

Once declared channels can be written to and read from using the ! and ? operators respectively. Data written to or read from a channel must be of the correct type. An example of this can be seen on line 12 where the variables from and to, declared with type pid, match the type format of the message.

len(in) > 0 → in?from,to;

This command checks if there is at least one message on the channel. If this is true, the message at the head of the channel is removed and its values placed into the appropriately typed variables. Similarly, the update of the command given on line 14, takes two appropriately typed variables and places their values in a message that is appended to the tail of a channel queue.

· · · → 0.5 : (in!1,_pid) + 0.5 : (in!2,_pid);

If the channel is full the command will not execute. If a user wants to guarantee that a channel write update will execute when there is enough space on the channel to accept a sequence of messages, a proposition of the following form can be added to the guard.

$$\text{len(channel)} < \text{cap(channel)} - \text{required space}$$

Similarly a user can specify the behaviour a model will exhibit when there is not enough space on the channel by adding a proposition of the following form to the guard.

$$\text{len(channel)} > \text{cap(channel)} - \text{required space}$$

- **Initial**: - The Initial operator allows a parameterised process to be instantiated multiple times and passed any required parameter values to construct the initial state of the system. The use of the initial operator can be seen on line 20

  Initial {client(medium); client(medium); client(medium);}

  where 3 client processes are created and passed the global channel medium. Line 6 of the PSS specification reveals that the channel variable in is an alias for the channel variable medium.

  Process client(**chan** in)

  When reading the specification all occurrences of in can be directly replaced by medium. As all the variable mentioned in the specification have a defined initial value, the first state of the system can be constructed.

Now that the additional language features have been explained the behaviour of the PSS specification given in Figure 4.5 can be described. Each of the 3 client processes holds a unique segment of a message, the goal being to reconstruct the full message. Each client transmits their data to one of the other clients, the destination client being determined probabilistically. Message segments are transmitted via a shared channel medium and each client periodically polls medium to determine if it is the intended recipient of any present segment. If this is the case, the client reads the data, from medium and adds it to the appropriate message slot. Once a client has reconstructed the full message it sets a flag and the program terminates.

4.2 Formal Definition of PSS

In order to specify the PSS language, we employ a context-free grammar (or Backus-Naur Form) [79]. The specification of the PSS language is given in Figure 4.6. A context-free grammar consists of the following four elements:

- A set of terminal symbols often referred to as tokens. Terminals are used to define the base symbols of a language. Common base symbols are variable names, digits, reserved language words and logical operations such as $<$ or $\geq$. In the PSS language definition (see Figure 4.6) boldface strings represent terminals.
- A set of non-terminal symbols often referred to as syntactic variables. Syntactic variables are used to define a set of terminal and further non-terminal symbols that can be used to replace the variable. The way in which the variable can be replaced is given by production rules. In Figure 4.6 *italicised* strings represent non-terminal symbols.
- A set of production rules. A production rule begins with a non-terminal symbol called the head of the production, a separating arrow, and a sequence of terminals and/or non-terminals called the body of the production. For notational convenience, terminal and non-terminal symbols appearing in the body can be grouped together. These groupings are called alternatives and are separated by the symbol |, which is read as "or".

  $$\text{head} \rightarrow \text{alternative}_1 \mid \text{alternative}_2 \mid \ldots \mid \text{alternative}_\text{n}$$

  A production rule states that the head must be replaced by one of the alternatives.
- A starting non-terminal symbol. In Figure 4.6, the head of the first production rule is the starting non-terminal symbol.

Although not required by the definition of a context-free grammar, the provision of recursion can be simplified through the use of the following symbol modifiers:

- ? : symbols appearing in the body of a production, or groups of symbols enclosed in parenthesis, can be omitted or appear at most once. Note that ? appears as a superscript symbol and should not be confused with our symbol for read.
- * : symbols appearing in the body of a production, or groups of symbols enclosed in parenthesis, can be omitted or appear any number of times.
- + : symbols appearing in the body of a production, or groups of symbols enclosed in parenthesis, must appear at least once. Note that + appears as a superscript symbol and should not be confused with the separator of updates in PSS commands.

A context-free grammar definition gives a list of production rules that can be used to generate the set of all strings that are part of the language. Consequently, if a string cannot be constructed using the available rules it is not a valid part of the language.

specification → (*channel* | *variable* | *array*)* *process initial*

channel → **chan** *name* **= [** *number* **] of** {(*type* | *ichan*) (, *type* | *ichan*)* }**;**

variable → *type name* (**:=** *number*)$^?$ **;**

array → (*type* | *chan*) **[** *number* **]** *name* **=** {*number* (, *number*)* }**;**

type → **int** | **pid**

ichan → **chan** {(*type* | *ichan*) (, *type* | *ichan*)*}

process → **Process** *name* ((**chan** | *type*) *name*
(, (**chan** | *type*) *name*)*) { *body* }

body → (*channel* | *variable* | *array*) *statement*

statement → *guard* → *update***;**

guard → *expression logicop expression* | **!***guard* | *guard* **&&** *guard*
| (*guard* || guard) | (*guard*)

update → *probability* : *choice* (**;** *choice*)* (+ *probability choice* (**;** *choice*)*)

choice → *name* **:=** *expression* | *name* **?** *name* (, *name*)* |
name **!** *name* (, *name*)*

initial → **Initial** { *name* (*argument*, (*argument*)*);
(*name* (*argument*, (*argument*)*);)*

argument → name | *number* | **null**

expression → *name* | *number* | _**pid** | **null**
len(*name*) | (*expression*) | *expression mathop expression*

logicop → **==** | **!=** | **¡** | **¡=** | **¿** | **¿=**

mathop → **+** | – | * | **mod** | **/**

name → **an alpha numeric string that must start with a letter**

number → **an integer**

probability → **a decimal number between 0 and 1**

Figure 4.6: Context free grammar definition of the PSS specification language.

4.2.1 Variable Declarations

The PSS specification language supports two primitive data types, int and pid which represent integer and process identifiers respectively. Both integer and process identifier variable declarations follow the same format: a keyword indicating the data type, followed by an alphanumeric identifier and an optional initial value.

$$(\text{int} \mid \text{pid})\ \text{identifier}\ (:= \text{integer})^{?};$$

When no initial value is specified, variables are given the default value zero. Both integer and process identifier types accept a finite range of values as this guarantees the construction of a finite state space on which the model checking algorithms detailed in Section 2.5 can operate. Integers use 4 bytes of storage to support values in the range of –2, 147, 483, 648 to +2, 147, 483, 647 and process identifiers use 1 byte of storage to support values in the range of 0 to 256. While being restricted to a finite state space is a clear limitation, the approach still allows for the specification and verification of complex systems.

Channel variables provide a way to specify inter process communication. The PSS channel syntax modifies the traditional style found in Promela to allow the structure of a message to be determined solely from examination of the channel declaration. Channels are declared using the reserved keyword chan followed by an alphanumeric identifier, a channel capacity and the structure of the message accepted by the channel. This message structure takes the form of a comma-separated list of type names. We refer to such a channel as chan{T}, where T denotes a comma-separated list of types.

$$\text{chan name} = [\text{integer}]\ \text{of}\ \{\ \text{type-list}\ \};$$

The types which comprise T may themselves be channel types, in addition to primitive integers and pid types. From the syntax definition when T contains a channel type, the type-list of the internal channel must be explicitly defined. For example, the channel declaration;

$$\text{chan y [2] of } \{\text{int, chan}\{\text{int, int}\}, \text{pid}\}$$

allows channels declared with the type-list {int, int} to be passed over it.

The reason for this verbose style of declaration will become clear in Chapter 5 when we consider the automatic detection of symmetry directly from the specification. Furthermore, this style allows for simplified type checking on channel read and write operations. Unlike

Promela, channel operations of the wrong type are immediately obvious in the PSS specification language and can be detected at compile time [34].

In a PSS specification c?msg and c!msg denote a read and write of a message from/to channel c. Note that cap(c) returns the capacity of channel c. We use $[m_1, \ldots, m_k]$ $(1 \leq k \leq cap(c))$ to denote the queue of a channel containing k messages, [] an empty queue and c[k] the k^{th} message in c. The writing of a message m to a channel whose queue is currently $[m_1, \ldots, m_k]$ (and not already full) results in the queue $[m_1, \ldots, m_k, m]$ whereas reading a message from said channel results in the queue $[m_2, \ldots, m_k]$. If a channel queue is currently $[m_1, \ldots, m_{cap(c)}]$ writing a message m will result in an unchanged queue $[m_1, \ldots, m_{cap(c)}]$.

The reserved keywords len and cap enable the current length and maximum length of a channel queue to be determined. These two operators can be combined to construct a guard that will allow a command to execute if a channel c has a specific capacity, see Figure 4.7.

Guard	**Meaning**
$len(c) > 0$	c contains at least one message
$len(c) < cap(c)$	c has at least one space
$len(c) < cap(c) - n$	c has at least n spaces

Figure 4.7: Example uses of the cap and len operators.

An array declaration in PSS takes the form

type name [number];

where type $\in \{$int, pid, chan$\}$, name is a valid identifier and enclosed in square brackets [] is the size of the array. The first element in an array always has index zero. An array in PSS is not a type, its only job is to provide a simplified manner of creating and managing multiple elements of the same type.

4.2.2 Language Definition

A PSS specification is composed from a set of global variables, Var_{global} and a set of processes $\{Proc_i | 1 \leq i \leq n\}$ for some $n > 0$ and a special operator **Initial**. Each process, $Proc_i$, consists of a set of local variable declarations, Var_i, and a set of commands Cd_i. Let $Var_{proc} = \cup_i Var_i$, then the set of all variables $Var = Var_{proc} \cup Var_{global}$. Finally, for every $v \in Var$, let $\overline{v}$ denote the initial value of v.

Process templates have the form **Process** name (param_list) {body} and in the style of Promela are initiated within a single operator, in our case Initial; i.e. **Initial** {name(param_list);

. . . }. The **Initial** operator determines the number of processes that comprise the final model and passes any values required to ensure all local variables have an initial value. All processes are created simultaneously in the first state, and each running process has a unique non-negative process identifier. The value assigned to the identifier is based on the order the processes appear in the **Initial** operator starting from 0. Each process can refer to its own pid via the predefined local variable _pid.

The behaviour of process Proc_i is determined by its associated set of commands Cd_i. Each command $\sigma \in \text{Cd}_i$ contains a guard g and a set of pairs (p_j, u_j) where $p_j \in \mathbb{R}_{>0}$ and u_j is an update. A guard g is a boolean function taking input from the sets $\text{Var}_{\text{global}}$ and Var_i and each update u_j consists of changes to these same sets. Finally, p_j attaches a probability to each update that determine the likelihood that an update will occur. It is required that, for each j, $p_j \in (0, 1]$ and that $\sum_j p_j = 1$.

4.2.3 Definition of Atomic Propositions

A set of atomic propositions AP for a PSS specification can now be defined. Let D represent a finite data domain and $x \in \text{Var}$ a variable with type pid or int. Then for each $d \in D$, $(x = d) \in \text{AP}$.

Let $c \in \text{Var}$ be a variable declared with type channel. Using the notation introduced in Section 4.2.1, $c = [m_1, \ldots, m_k]$, $(0 \leq k \leq \text{cap}(c))$ denotes a channel queue containing k messages where c[k] indexes the k^{th} message. If c is a channel accepting a message of type T, then for all $0 < k \leq \text{cap}(c)$, $(c[k] = \text{msg}) \in \text{AP}$ for all $\text{msg} \in T$.

4.2.4 States of a Model Associated with a PSS Specification

Let S be the set of potential states in a model $\mathcal{M}$ associated with a PSS specification $\mathcal{P}$. Then S consists of every possible assignment of values to variables and channels declared in $\mathcal{P}$. As the range of values supported by int and pid is finite, S is finite. It follows that a state $s \in S$ of a specification $\mathcal{P}$ can be expressed using a set of atomic propositions.

Let Proc_i be a process in specification $\mathcal{M}$. For $x \in \text{Var}_i$, the notation p[i].x indicates the local variable x of process Proc_i where i is the process's unique pid value and p is its name. For $x \in \text{Var}_{\text{global}}$, x can be unambiguously referred to using only its name. Consider the outline PSS specification given in Figure 4.8.

This skeleton specification contains three channels, A, B, C, and a global int variable called count. Furthermore, the specification contains two instantiations of the user process. Placing

```
1.  dtmc
2.
3.  chan A [2] of {pid, chan{int}};
4.  chan B [1] of {int};
5.  chan C [1] of {int};
6.  int count;
7.
8.  Process user (chan {pid, chan{int}} in; chan {int} out)
9.  {
10.   ...
11. }
12.
13. Initial{user(A,B); user(A,C);}
```

Figure 4.8: Skeleton code of a potential PSS specification.

channels and global variables in the order they appear in the specification, and ordering the local variables of $user_1$ before $user_2$, an example state s would be expressed as:

$$\begin{aligned} s = \{&(A[1] = [(1, B)]), (A[2] = \text{null}), (B[1] = [3]), (C[1] = [2]), (count = 1), \\ &(user[1].in = A), (user[1].out = B), (user[1].\_pid = 0), \\ &(user[2].in = A), (user[2].out = C), (user[2].\_pid = 1)\}. \end{aligned}$$

The initial state of the specification can be constructed from the initial values assigned to all variables, and parameter values passed to process in the **Initial** process.

4.2.5 Expression Evaluation

In Figure 4.6, the syntax of expressions is given. This section shows how an expression is evaluated in the context of a state $s \in S$. We adopt the same format as used in the description of Promela-lite [37]. A function $eval_{p,i}$ (s, e) accepts an expression e and a state s and returns the effect of evaluating e at state $s \in S$ in the context of $Proc_i$. Let $s \in S$ be a state in a PSS specification $\mathcal{P}$. If e is:

- x where $(x = a) \in s$, $eval_{p,i}(s, e) = a$.
- p[i].x where $((p[i].x) = a) \in s$, $eval_{p,i}(s, e) = a$.
- len(c) where $(c = [m_1, \ldots, m_k]) \in s$, $(0 \leq k \leq cap(c))$, $eval_{p,i}(s, e) = k$.

- len(p[i].c) where (p[i].c = $[m_1, \ldots, m_k]$) $\in$ s, ($0 \leq k \leq$ cap(p[i].c)), $eval_{p,i}(s, e) = k$.
- a where $a \in \mathbb{Z}$, $eval_{p,i}(s, e) = a$.
- _pid, $eval_{p,i}(s, e) = i$.
- (g) where g in an expression, $eval_{p,i}(s, e) = eval_{p,i}(s, g)$.
- $e_1 \circ e_2$ where $\circ \in \{+, -, \star, \div\}$ and e_1 and e_2 only contain integer variables, $eval_{p,i}(s, e) = eval_{p,i}(s, e_1) \circ eval_{p,i}(s, e_2)$.

As variables of type int represent a finite range of values, expressions must handle the case where they return a value outside this range. This scenario is handled in the same way as implemented in the SPIN model checker [58]. Let min(int) and max(int) denote the minimum and maximum values storable in an integer variable, where min(int) is a negative number. To ensure the result of the expression $e_1 \circ e_2$ falls outside of the accepted range, it is evaluated as follows;

$$((eval_{p,i}(s, e_1) \circ eval_{p,i}(s, e_2) + |min|) \mid (max - min)) - |min$$

The calculation of max(int) + 1 returns min(int). Variables of type pid cannot be evaluated as part of an arithmetic expressions as they are strictly for process referencing.

4.2.6 Guard Evaluation

The syntax of guards is given in Figure 4.6. This section shows how a guard is evaluated in the context of a state $s \in S$. The function $eval_{p,i}$ is used to determine whether a guard holds in a given state. For a guard g appearing in the context of process p with pid value i, $s \models_{p,i} g$ returns true if and only if g evaluated at $s \in S$ in this context is satisfied. The relation $\models_{p,i}$ is defined as follows:

- $s \models_{p,i} e_1 \bowtie e_2$ is satisfied $\Leftrightarrow eval_{p,i}(s, e_1) \bowtie eval_{p,i}(s, e_2)$ where $\bowtie \in \{==, \neq, <, \leq, >, \geq\}$.
- $s \models_{p,i} \neg g \Leftrightarrow s \not\models_{p,i} g$.
- $s \models_{p,i} g_1 \,\&\&\, g_2 \Leftrightarrow s \models_{p,i} g_1$ and $s \models_{p,i} g_2$.
- $s \models_{p,i} g_1 \,||\, g_2 \Leftrightarrow s \models_{p,i} g_1$ or $s \models_{p,i} g_2$.
- $s \models_{p,i} (g) \Leftrightarrow s \models_{p,i} g$.

4.2.7 Effect of Updates

Let x $\in$ Var represent either a global variable x or local variable p[i].x with type chan or int that appears in a PSS specification $\mathcal{P}$. In Figure 4.6, the syntax of possible updates is given. For a subset of allowed updates u, the effect of executing u on a state s in the context of process i is given by a function $\text{exec}_{p,i}(s, u)$. The function $\text{exec}_{p,i}(s, u)$ and the conditions under which u can be applied is defined as follows:

- If x = a at s, e is an expression and u is an update of the form x := e, then $\text{exec}_{p,i}(s, u)$ is $(s\setminus (x) = a)) \cup \{x = \text{eval}_{p,i}(s, e))\}$.
- If $x = [m_1, \ldots, m_n]$ at s and u is an update of the form $x\,!\,e_1, e_2, \ldots, e_k$, then $\text{exec}_{p,i}(s, u)$ when $s \models_{p_i} \text{len}(x) < \text{cap}(x)$ is $(s\setminus\{(c = [m_1, \ldots, m_k]\}) \cup \{(c = [m_1, \ldots, m_n, (\text{eval}_{p,i}(s, e_1), \text{eval}_{p,i}(s, e_2), \ldots, \text{eval}_{p,i}(s, e_k))]\}$.
- If $x = [m_1, \ldots, m_n]$ at s, for an update u of the form $x\,!\,e_1, e_2, \ldots, e_k$, $\text{exec}_{p,i}(s, u)$ when $s \models_{p_i} \text{len}(x) == \text{cap}(x)$ is (s).
- If $x = [(a_{1,1}, \ldots, a_{1,k}), \ldots, m_n]$ at s and u is an update of the form $x\,?\,e_1, e_2, \ldots, e_k$, then $\text{exec}_{p,i}(s, u)$ when $s \models_{p_i} \text{len}(x) > 0$ and $x_j = y_j$ at s for $(1 \leq j \leq k)$ is $(s\setminus\{(x = [(a_{1,1}, \ldots, a_{1,k}), \ldots, m_n], (x_1 = y_1), (x_2 = y_2), \ldots, (x_k = y_k)\} \cup \{(x = [a_2, \ldots, a_m]), (x_1 = a_{1,1}), (x_2 = a_{1,2}), \ldots, (x_k = a_{1,k})\}$.
- If x = [] at s, for an update u of the form $x\,?\,e_1, e_2, \ldots, e_k$, $\text{exec}_{p,i}(s, u)$ when $s \models_{p_i}$ len(x) == 0 is (s).

A common problem with language definitions is that they permit strings that lead to invalid operations. A classic example is divisions by 0. The context free grammar definition of the PSS language given in Figure 4.6 is no exception. It permits update strings that have no defined update rule or expressions that cannot be evaluated.

Therefore, the thesis restricts itself to considering PSS specifications that have a constructible initial state and all executable updates and guards match one of the rules given in the proceeding sections. In Chapter 8 we describe the implementation of the PSS model checker and indicate ways in which malformed specifications are identified and reported to the user before runtime.

4.3 Constructing the Discrete Time Markov Chain

Examining the definition of a DTMC given in Section 2.2.2, the construction of a DTMC from an appropriate PSS requires the creation of a set of states S, an initial state and a probability transition matrix **P** must be defined.

Then the set of all states S consists of every possible assignment of values to variables and channels declared in $\mathcal{P}$. As covered in Section 4.2.4 an individual state $s \in S$ can be expressed as a an ordered tuple of variables. With global variables ordered as they appear in the specification and variables in process placed in instantiation order, a single state can be defined $s = (\text{Var}_{\text{global}} \cup \text{Var}_1 \cup \ldots \cup \text{Var}_n)$ for $n > 0$. If $\overline{\text{Var}_i}$ and $\overline{\text{Var}}_{\text{global}}$ are sets of variables $\{\overline{v}_1, \ldots, \overline{v}_k\}$ and $\{\overline{w}_1, \ldots, \overline{w}_q\}$ respectively, the initial state is $s_o = (\overline{\text{Var}}_{\text{global}} \cup \overline{\text{Var}_1} \cup \ldots \cup \overline{\text{Var}_n})$.

To create **P**, the behaviour of each process Proc_i must be determined. For Proc_i consider a command $\sigma \in \text{Cd}_i$ where $\sigma = (g, (p_1, u_1), \ldots, (p_n, u_n))$. As g is a predicate over the set of variables $\text{Var}_{\text{global}}$ and Var_i, it defines a subset of S hence referred to as $S_g = \{s \in S \mid s \models g\}$. An update u_j describes changes that occur to variable sets $\text{Var}_{\text{global}}$ and Var_i. It follows that u_j can be thought of as a function from S_g to the set S_{new} of states created by applying the update (see Section 4.2.7).

Using the p_j value associated with each update u_j, command σ defines, for each $s \in S_g$, a function $\mu : S_{\text{new}} \to \mathbb{R}_{\geq 0}$ where for each $t_i \in S_{\text{new}}$

$$\mu(t_i) = \sum_{\{j \,|\, \text{exec}_{p,i}(s,u_j)=t_i\}} p_j$$

To determine the behaviour of Proc_i in every state, we combine this information for all commands $\sigma \in \text{Cd}_i$. We denote this by a function $\mathbf{P}_{i,\text{ind}} : S \times S_{\text{new}} \to [0, 1]$ where for each $s \in S$ and $t_i \in S_{\text{new}}$

$$\mathbf{P}_{i,\text{ind}}(s, t_i) = \begin{cases} \mu(t_i) & \text{if } s \in S_g \text{ for some } \sigma \in \text{Cd}_i \\ 0 & \text{otherwise} \end{cases}$$

As there is no synchronisation between processes a command in the specification corresponds to a transition, or set of transitions, in the DTMC. We define the effect that process Proc_i transitions have on the full model using the function $P_i : S \times S \to [0, 1]$. For states $s = (s_1, \ldots, s_m) \in S$ and $t = (t_1, \ldots, t_m) \in S$

$$\mathbf{P}_i(s, t) = \begin{cases} \mathbf{P}_{i,\text{ind}}(s, t_i) & \text{if } s_j = t_j \text{ for all } 1 \leq j \neq i \leq m \\ 0 & \text{otherwise} \end{cases}$$

Finally, we define the probability transition matrix **P** of the overall model. In each global state, some subset of the processes can independently make transitions. We assume a uniform probability of each process being scheduled. Hence we define $\mathbf{P} : S \times S \to [0, 1]$ where p_s is the number of processes which can make a transition in state s as:

$$\mathbf{P}(s,t) = 1/p_s(\sum_{i=1}^{m} P_i(s,t))$$

4.4 Construction of the Markov Decision Process

To construct an MDP from a PSS specification a similar approach is taken. Let the state space S, a state $s \in S$, initial state $\bar{s}$, a command $c \in Cd_i$, the set S_{new}, the set S_g and functions $\mu : S_{new} \to \mathbb{R} \geq 0$ be defined exactly as found in Section 4.3.

From the definition of a MDP, see Section 2.2.3, the set of guards in a process are not necessarily disjoint and a set of probability distributions may be enabled in a single state. To accommodate this a function $Steps : S \to 2^{Dist(S)}$, maps each individual state $s \in S$ to a finite, non-empty subset of Dist(S). Therefore, each process $Proc_i$, has an associated function $Steps_{i,ind} : S \to 2^{Dist(S_{new})}$ that links each state $s \in S$ to a set of probability distributions over the set S_{new}

$$Steps_{i,ind}(s) = \{\mu \mid c \in Cd_i \text{ and } s \in S_g\}$$

To generalise the function so it provides a probability distribution over the set of all states S, let the variable sets for all other processes remain unchanged. Only the variable sets Var_i and Var_g are modified. The required function is denoted $Steps_i : S \to 2^{Dist(S)}$.

Let $s = (Var_{global} \cup Var_1 \cup \ldots \; Var_n)$ be a state $s \in S$ then the function $Steps_i(s)$ is defined in the following manner. For each $\mu_i \in Steps_{i,ind}(s)$, $Steps_i(s)$ is the set of probability distribution $\mu \in Dist(S)$, where for any state $t = (Var_{global} \cup Var_1 \cup \ldots \cup Var_n) \in S$:

$$\mu(t) = \begin{cases} \mu_i(t_i) & \text{if } s_j = t_j \text{ for all } 1 \leq j \neq i \leq m \\ 0 & \text{otherwise} \end{cases}$$

To provide the function required by the definition $Steps : S \to 2^{Dist(S)}$ the above functions are combined for every process in the specification;

$$Steps(s) = \bigcup_{i=1}^{m} Steps_i(s)$$

This is appropriate as scheduling between processes is non-deterministic.

4.5 Summary

In order to allow for the development of our automated symmetry reduction techniques we have introduced the Probabilistic Symmetric Systems Language. The grammar and full semantics of PSS were presented in Section 4.2 and Section 4.3 respectively. We make extensive use of the PSS language in the remainder of the thesis to aid in the presentation and proof of our results.

CHAPTER 5

Automated Symmetry Detection

This chapter introduces the Extended Channel Diagram (ECD) of a PSS specification $\mathcal{P}$ and formally establishes a correspondence between automorphisms of an ECD of a PSS specification $\mathcal{P}$ and automorphisms of the probabilistic models constructible from $\mathcal{P}$. In contrast to previous approaches to symmetry detection [47, 30, 13], the ECD approach is the first technique we know of that can detect arbitrary component and data symmetries directly from a specification. Therefore, it offers distinct advantages over techniques such as scalar sets [13], which only captures data symmetries when all variables are interchangeable, and the static channel diagram approach [30], which only captures arbitrary component symmetries. Therefore, the ECD approach has the potential to capture a larger group of symmetries, and as a result the possibility of mapping more states to a single or smaller number of representative states arrises.

Like previous techniques designed to detect arbitrary component symmetry [30] this approach can be fully automated and requires no additional information from the user. Furthermore, we assert that capturing data and component symmetries enables a user to write specifications in the way they desire. They are not forced to needlessly place information within processes to take advantage of component symmetries.

To summarise, the ECD approach allows a set of potential automorphisms of a model associated with a PSS specification to be generated. Provided that the automorphisms meet a small set of restrictions then they are valid for symmetry reduction. From the set of valid

automorphisms we show how a potentially larger set of automorphisms valid for reduction can be calculated.

5.1 Automated Detection

An approach capable of detecting arbitrary component symmetries directly from the channel based specification language Promela-lite has previously been described [37]. This approach generates a diagram, known as a static channel diagram, of communication that may potentially occur between processes in the specification's underlying Kripke structure. The automorphisms of this potential communication diagram correspond to a set of Kripke structure automorphisms, some of which may be valid for symmetry reduction. This set of potential symmetries are subsequently narrowed to provide a set of automorphisms of the associated Kripke structure guaranteed to be valid for symmetry reduction.

We now show how the static channel diagram approach can be applied to the PSS specification language to detect potential symmetries valid for reduction in the underlying probabilistic model. Here we note that, unlike Promela-lite, the PSS language does not support communication arising from the dynamic passing of channel references [38]. Therefore, all channels in PSS are static and henceforth we simply refer to the technique as a channel diagram.

5.1.1 Channel Diagram Associated with a PSS Specification

Let $\mathcal{P}$ be a PSS specification with $n > 0$ processes and let $V_P = \{1, 2, \ldots, n\}$ be the set of process identifiers and V_C the set of channel identifiers in $\mathcal{P}$. For $i \in V_P$ let process(i) denote the name of process i, and for $c \in V_c$ let chan(c) denote the comma separated list of types accepted by c (see Section 4.2.1).

Definition. The channel diagram associated with $\mathcal{P}$ is a coloured, bipartite digraph $C(\mathcal{P}) = (V, E, C)$ where V, E and C are the sets of vertices, edges and colours and:

- $V = V_P \cup V_C$
- For $i \in V_P$ and $c \in V_C$
  - $(i, c) \in E$ if and only if process(i) has a statement involving an update that includes a write operation to channel c;
  - $(c, i) \in E$ if and only if process(i) has a statement involving an update that includes a read operation from channel c.

- C is a colouring function that colours all vertices according to the type of the associated processes and channels.

An automorphism of a channel diagram is a bijection $\alpha : V \rightarrow V$ which satisfies the following three conditions:

- $\forall$ i, j $\in$ V, (i, j) $\in$ E $\Rightarrow$ (α(i), α(j) $\in$ E)
- $\forall$ i $\in$ V, C(i) = C(α(i))

The second condition ensure that only processes and channels of the same colouring can be mapped to each other.

In the diagrammatic presentation, processes are represented by circles and channels by double lined quadrilaterals. Either the name of a process (process(i)) or the comma separated list of types accepted by the message of channel c (chan(c)) are used to label the vertices. As the colouring function uses process(i) and chan(c) to determine the colour of a vertex, there is a one to one correspondence between vertex labels and colours.

5.1.2 Examples of Channel Diagrams Associated with a PSS Specification

This section illustrates the concept of a channel diagram by examining the C($\mathcal{P}$) for a variety of PSS specifications $\mathcal{P}$. In turn we discuss the set of automorphisms Aut(C($\mathcal{P}$)) and how they relate to automorphisms in the associated probabilistic model.

Simple Mutual Exclusion Specification

The first example is the simple mutual exclusion specification $\mathcal{P}$ given in Figure 5.1. The channel diagram C($\mathcal{P}$) contains two identically labeled vertices, one for each mex process.

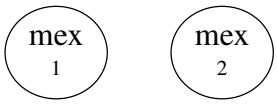


Figure 5.1: Channel diagram for a two process mutual exclusion specification.

As specification $\mathcal{P}$ does not contain any channels C($\mathcal{P}$) does not contain any edges. Using integer subscripts to uniquely identify each vertex, the automorphisms of the diagram are

$$\text{Aut}(\text{C}(\mathcal{P})) = \{(1, 2)\}.$$

How a permutation $\alpha \in$ Aut(C($\mathcal{P}$) corresponds to a permutation of the associated probabilistic model is formally defined later in this chapter. Here we note, in this instance a permutation of the process indices in C($\mathcal{P}$) corresponds to a permutation of processes in the states of the associated probabilistic model.

Figure 5.2 shows the general form of the channel diagram for an n process mutual exclusion specification. For this diagram the automorphism group is S_n (i.e. all permutations of processes).

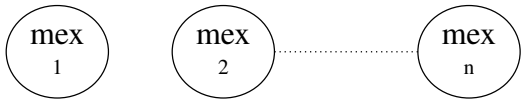


Figure 5.2: General form of the $\mathcal{C}(\mathcal{P})$ for a n process mutual exclusion specification.

Dining Philosophers Specification

The well known dining philosophers problem [71] provides a simple description of process deadlock occurring in operating systems. A common description follows;

A group of N philosophers have congregated at a circular table to eat and discuss philosophy. To eat, a philosopher needs two forks, but there is only a total of N forks at the table. A deadlock would arise if every philosopher held a left fork and waited indefinitely for a right fork. Conversely, holding a right fork and waiting for a left fork will also result in deadlock. In general a deadlock is reached when there is a cycle of unwarranted requests. Philosopher P_1 is waiting for a fork grabbed by philosopher P_2 who is waiting for the fork of philosopher P_3 and so on. The channel diagram C($\mathcal{P}$) for the PSS specification $\mathcal{P}$ containing three philosophers is shown in Figure 5.3.

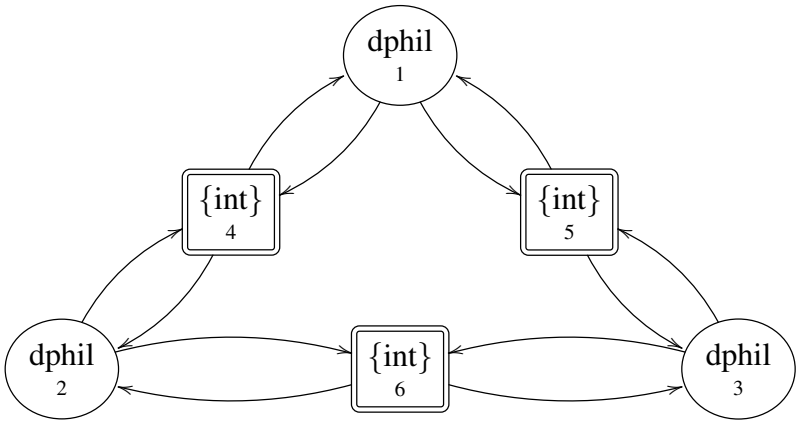


Figure 5.3: Channel diagram for a PSS dinning philosophers specification with three philosophers.

The channel diagram contains three identically coloured circular vertices, one for each philosopher and three identically coloured double lined quadrilateral vertices, one for each channel. Every philosopher can read and write from two channels and this is reflected by the edges present in $C(\mathcal{P})$. Using integer subscripts to uniquely identify each vertex, the automorphisms of the diagram are given by the group generators

$$\mathrm{Aut}(C(\mathcal{P})) = \{(1, 2)(5, 6), (2, 3)(4, 5)\}.$$

As in the preceding example, a permutation of process indices in $C(\mathcal{P})$ corresponds to a permutation of the processes present in the states of the associated probabilistic model. However, every permutation of process indices in $C(\mathcal{P})$ is accompanied by a permutation of channel names that leaves a process index attached to the same channel names. In a similar manner, permutations of channel names in $C(\mathcal{P})$ correspond to a permutation of channel variables present in the states of the associated probabilistic model.

The general form of the channel diagram associated with an n process dining philosophers is isomorphic to the automorphism group S_n (i.e. all permutations of processes).

Network Infection Specification

Appendix A.3 gives the skeleton PSS specification describing the progress made by a computer virus as it infects a network. The specification describes a network of computer nodes arranged in an N $\times$ N grid. Computer nodes have a direct connection to nodes located to their direct north, south, east or west. The nodes located at the border of the network are only connected to two or three other nodes. This requires the specification to have three different descriptions for nodes, to accommodate how many connections the node has. In the channel diagram, processes able to connect to two others will be coloured "C2", processes able to connect to three others will be coloured "C3" and processed able to connect to four others will be coloured "C4" (see Figure 5.4).

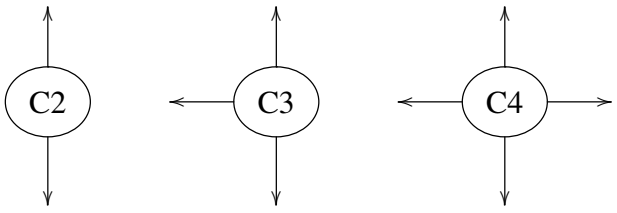


Figure 5.4: Examples of computer process colourings.

The specification describes the scenario in which a computer virus is initially present in one of the edge computer nodes. Once the "inception node" is non-deterministically selected,

the virus remains in this node and repeatedly attacks any neighbouring computer node that is currently not infected. To infect a new computer node the virus must try first to pass through the node's firewall and if successful, try to infect the node. For both of these steps, there is a probabilistically determined chance of success. The channel diagram C($\mathcal{P}$) for the PSS specification $\mathcal{P}$ with a 3×3 grid of computer nodes is shown in Figure 5.5.

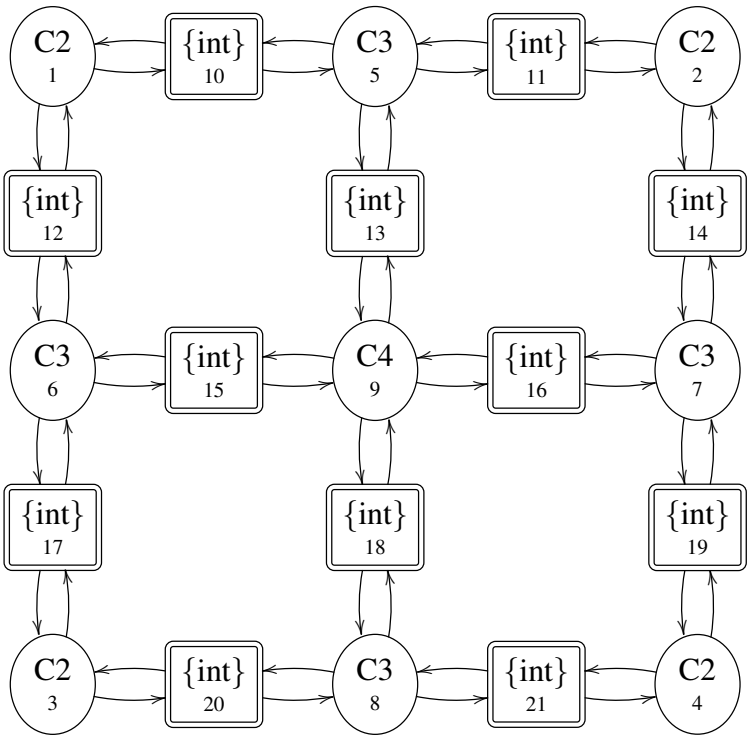


Figure 5.5: Channel diagram of a 3×3 network infection specification.

The channel diagram contains 21 vertices. There are 9 circular processes each taking one of the three available colours and 12 identically coloured channels. Every process vertex has both an incoming and outgoing edge to their linked channel vertex. Intuitively this is because each computer can be infected by or infect a neighbouring computer.

However, the computer that is initially infected will never read from any of the channels it is connected to, it will only try to infect its neighbours. This illustrates an important point of channel diagrams, they capture the potential communication allowed by the specification and not the actual communication in the underlying probabilistic model. The automorphisms of the diagram, using integer subscripts to uniquely identify each vertex, are given by the group generators

$$\begin{aligned}\mathrm{Aut}(\mathrm{C}(\mathcal{P})) = \{&(1,\,2)(6,\,7)(3,\,4)(10,\,11)(12,\,14)(15,\,16)(17,\,19)(20,\,21),\\ &(1,\,3)(5,\,8)(2,\,4)(12,\,17)(10,\,20)(13,\,18)(11,\,21)(14\,,19),\\ &(1,\,3)(5,\,6)(10,\,17)(11,\,12)(13,\,15)(3,\,4)(6,\,8)(17\,,20)(12\,,21)(15\,,18)\\ &(4,\,2)(8,\,7)(20,\,19)(21,\,14)(18,\,16)(2,\,1)(7,\,5)(19\,,11)(14\,,10)(16\,,13)\}\end{aligned}$$

It follows from Aut(C($\mathcal{P}$)) that any re-arrangement of processes and channels caused by flipping the diagram on an x or y axis can be provided by a permutation $\alpha \in$ Aut(C($\mathcal{P}$). As before, a permutation of process indices and channel names in C($\mathcal{P}$) correspond to a permutation of process and channel variables present in the states of the associated probabilistic model.

Finally, a regular polygon with n sides has 2n different symmetries: n rotational symmetries and n reflectional symmetries. These rotations and reflections make up the dihedral group D_n. Indeed, the general form of the channel diagram for an N $\times$ N specification is the automorphism group D_8.

5.1.3 Channel Diagrams and Data Symmetries

While this technique is well suited for detecting a set of candidate symmetries, it does not consider any type of symmetry other than component symmetries. This is unfortunate as specifications often contain large data structures and even when exploiting component symmetries it may be impossible to check that a property holds for every assignment of values. As with potential component symmetries detected from a specification, potential data symmetries present within a specification's data structures may also be detected. By capturing any additional data symmetries the possibility of mapping more states to a single or smaller number of representative states in the underlying model arises. Furthermore, capturing data and component symmetries allows users to write specifications in a more natural manner. They are not unduly forced to place data within either a process or channel to take advantage of component symmetry.

Examining the simple mutual exclusion PSS specification presented in Figure 4.2, it is clear that no data is stored within a process. Processes provide a list of commands and the data is stored in the globally defined integer variables. It follows that the number of states in the probabilistic model is determined by the values that can be assigned to these global variables. Figure 5.1 shows the channel diagram for this specification and it was asserted in Section 5.1.1 that a permutation of the channel diagram corresponded to a permutation of processes present in the states of the probabilistic model. As processes do not determine the number of states, the use of any component symmetries will not reduce the number of states in the probabilistic model.

5.1.4 Extended Channel Diagram Associated with a PSS Specification $\mathcal{P}$

To capture potential data and component symmetries, the definition of a channel diagram is extended and an algorithm to extract the newly defined diagram from a PSS specification is described in Figure 5.11. Furthermore, in Section 5.2 we define the correspondence between automorphisms of an ECD extracted from a specification $\mathcal{P}$ and automorphisms of the associated probabilistic model. In addition, we prove that if a permutation $\alpha \in$ ECD meets a small number of restrictions its corresponding probabilistic model permutation can be used for symmetry reduction.

An ECD directly extends channel diagrams by including vertices for global variables and in certain circumstances an edge between a process and global variable vertex is included. An edge between a process and global variable vertex is included in the diagram if a process can potentially update the variables value. Conversely an edge between a global variable and process vertex is included if an update made by the process uses the value stored in the variable. As an ECD is a direct extension of channel diagrams, the subsequent results including the proof given in Section 5.2 are true for channel diagrams extracted from a PSS specification.

Let $\mathcal{P}$ be a PSS specification with $n > 0$ processes, and let $V_P = \{1, 2, \ldots, n\}$ be the set of process identifiers, V_C the set of channel identifiers and V_G the set of global variable identifiers in $\mathcal{P}$. For $i \in V_P$ let process(i) denote the name of process i, for $c \in V_c$ let chan(c) denote the comma separated list of types accepted by c and for $x \in V_G$ let type(x) denote the type of variable x (see Section 4.2.1).

Definition. The extended channel diagram associated with $\mathcal{P}$ is a coloured, tripartite digraph ECD($\mathcal{P}$) = (V, E, C) where V, E and C are the sets of vertices, edges and colours and:

- $V = V_P \cup V_C \cup V_G$
- For $i \in V_P$ and $c \in V_C$
  - (i, c) $\in$ E if and only if process(i) has a statement involving an update that includes a write operation to channel c;
  - (c, i) $\in$ E if and only if process(i) has a statement involving an update that includes a read operation from channel c.
- For $i \in V_P$, $x \in V_G$
  - (i, x) $\in$ E if and only if process(i) has a statement involving an update that includes an assignment to variable x
  - (x, i) $\in$ E if and only if process(i) has a statement involving an update that includes changes to a variable via an expression involving x.

- C is a colouring function that colours all vertices according to process(i), chan(c) and type(x).

An automorphism of an ECD is a bijection α : V $\rightarrow$ V which satisfies the following four conditions:

- $\forall$ i, j $\in$ V, (i, j) $\in$ E $\Rightarrow$ (α(i), α(j) $\in$ E)
- $\forall$i $\in$ V, C(i) = C(α(i))

The second condition ensure that only processes and channels of the same colouring can be mapped to each other.

Like channel diagrams we present ECDs diagrammatically. As before, processes are represented by circles and channels by double lined quadrilaterals. In an ECD, single lined quadrilaterals are used to depict a global variable. Either the name of a process (process(i)), the comma separated list of types accepted by the message of channel c (chan(c)), or the type of variable x (type(x)) are used to label the vertices. As seen in the channel diagram examples there is a one to one correspondence between vertex labels and colours.

5.1.5 Examples of Extended Channel Diagrams Associated with a PSS Specification

This section illustrates the concept of an ECD by examining the ECD($\mathcal{P}$) constructed from a set of PSS specifications. In turn we discuss the set of automorphisms Aut(ECD($\mathcal{P}$)) and how they relate to automorphisms in the associated probabilistic model.

Simple Mutual Exclusion Specification

The advantage of capturing potential data symmetries is made clear by re-examining the PSS specification $\mathcal{P}$ describing a simple mutual exclusion problem, Figure 4.2. The ECD($\mathcal{P}$) for a two process specification is shown in Figure 5.6.

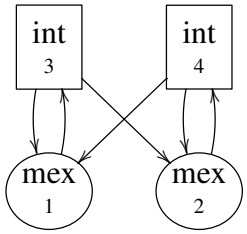


Figure 5.6: Extended channel diagram for a two process mutual exclusion specification.

As in the channel diagram, the ECD($\mathcal{P}$) for this specification contains two identically coloured vertices, one for each of the two identically typed processes. However, it now includes two quadrilateral variable vertices and edges between process and variable vertices to indicate the reading and writing of values to the variables in the specification. The automorphism group of the diagram, using the integer subscripts to uniquely identify each vertex is

$$\text{Aut}(\text{ECD}(\mathcal{P})) = \{(1, 2)(3, 4)\}.$$

Any arrangement of process vertices that leaves the variable vertices it shares an edge with attached can be provided with a permutation $\alpha \in \text{Aut}(\text{ECD}(\mathcal{P}))$. In contrast to simple channel diagrams, the corresponding model permutations now act on both process components and data in the state tuple. As previously asserted, the size of the state space is defined by the combination of values that global variables can take. The ECD has captured a potential symmetry that acts on these variables and consequently we have the possibility of constructing a smaller quotient structure is provided.

Figure 5.7 shows the general form of the ECD for an n process mutual exclusion specification. In contrast to channel diagrams the automorphism group is now isomorphic to S_n.

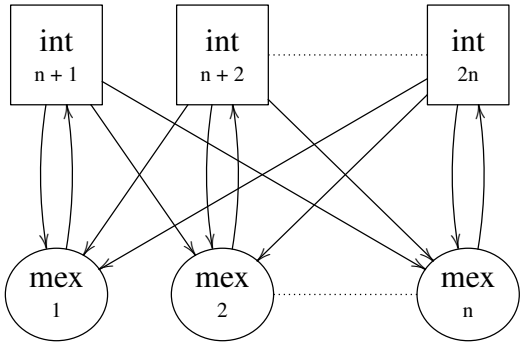


Figure 5.7: General form of ECD($\mathcal{P}$) for an n process mutual exclusion specification.

Dining Philosophers Specification

Examples of dining philosophers specifications are commonly presented in literature using global variables to model the state of the forks. One form of the specification in PSS is presented in Appendix A.2 and the associated ECD is shown in Figure 5.8.

The resulting diagram is similar to the channel diagram example but variable vertices directly replace the channel vertices. While the ECD has not captured a larger set of potential symmetries it has given the user the choice on how to formulate the specification. The burden has been removed from the the user on where to place data.

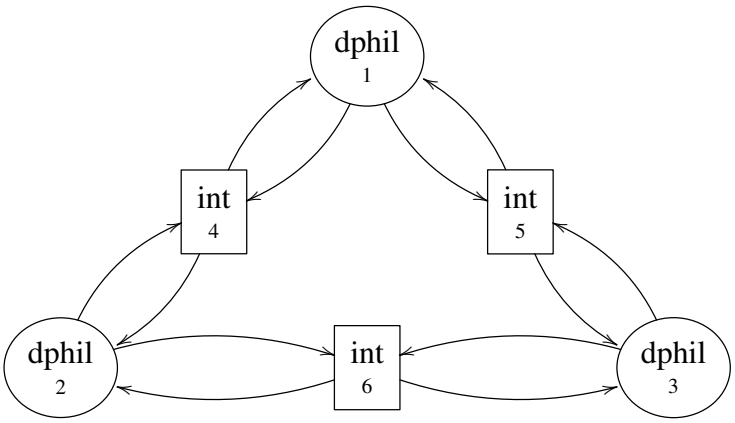


Figure 5.8: Extended channel diagram for a PSS dinning philosophers specification with three philosophers.

Monty Hall Problem

A further advantage of capturing potential data symmetries is that symmetries valid for reduction can be obtained from a specification containing a single process. This is not possible using channel diagrams as a specification with a single process will not contain any component symmetries. An example of a problem that would naturally be modelled using a single process would be the Monty Hall problem.

The Monty Hall problem can be stated as follows: There are three doors, behind two are goats and behind the third is a car. A contestant is asked to select a door and their prize is whatever lies behind it. Before the door is opened to reveal the prize, Monty Hall, who knows what's behind all the doors opens one of the other doors to reveal a goat. The contestant is now given the option to change their selection to the other door or stick with their original choice. The Monty Hall problem is to decide the optimal choice for the contestant.

The ECD($\mathcal{P}$) associated with specification $\mathcal{P}$ is shown in Figure 5.9.

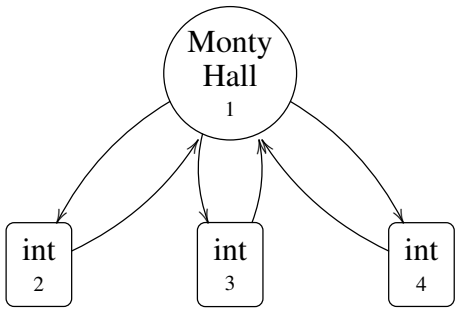


Figure 5.9: Extended channel diagram for a PSS monty hall specification.

As expected the ECD contains a single vertex for the command process and three identically coloured vertices for the global integers that represent the doors. Edges between the process

and variable vertices indicate the initial setting and subsequent revelation of what lies behind the doors as the game progresses. Using integer subscripts to uniquely identify each vertex, the automorphisms of the diagram are

$$\mathrm{Aut}(\mathrm{ECD}(\mathcal{P})) = \{(2,\,3)(3,\,4)\}.$$

Any arrangement of the variable vertices can be provided by a permutation $\alpha \in \mathrm{Aut}(\mathrm{ECD}(\mathcal{P}))$. This corresponds to a permutation of the integer variables in the states of the underlying probabilistic model. While the Monty Hall problem is commonly described using only three doors the problem can be generalised to use any number of doors. In this general form the automorphism group is isomorphic to S_n where n is the number of doors.

Here we note, as there is only one component, a user may legitimately declare all variables within the scope of the process. This would invalidate the detection technique as vertices are only present for global variables and processes. However, in a one process specification, with no ill effect, all variables can be made global in the background. This allows the ECD technique to be applied without prompting the user to restructure the specification.

In this and the previous examples, the data symmetries that have been identified could be expressed using scalarsets. However, the ECD technique can capture them automatically with no user input. Furthermore, the ECD is capable of capturing data symmetries not expressible using scalarsets. We show an example this in the following specification.

Resource Allocator Specification

A resource allocator accepts access requests from computers and can grant a single computer the right to use the resource. Each client has a priority level, and when multiple requests occur the resource allocator grants access to the computer with the highest priority. If several requests are made with the same priority the resource allocator chooses non- deterministically which to satisfy.

Communication between a computer and the resource allocator occurs over a channel, which a computer can use to send an access request message to the allocator. When the allocator decides which computer it will grant access to the resource it sends back a confirmation message on the same channel. Once the computer finishes using the resource it sends a finished message to the allocator. The allocator is then free to grant another computer access to the resource.

To add further complexity some of the computers are able to share the resource, thus bypassing the access decision made by the allocator. If computer i is configured to share with client

j then on receiving an access granted message, client i uses the resource and when finished gives the resource to client j. When client j finishes using the resource it sends it to another client in the chain. This resource passing continues until the resource is returned to Client i, which sends a finished message to the resource allocator

The inter communication between processes is modelled using global variables to pass the resource around the chain of computers. The skeleton PSS specification $\mathcal{P}$ given in Appendix A.5 specifies 9 computers with the same priority level and three of the clients have been set up to share the resource. The ECD($\mathcal{P}$) associated with specification $\mathcal{P}$ is shown in Figure 5.10.

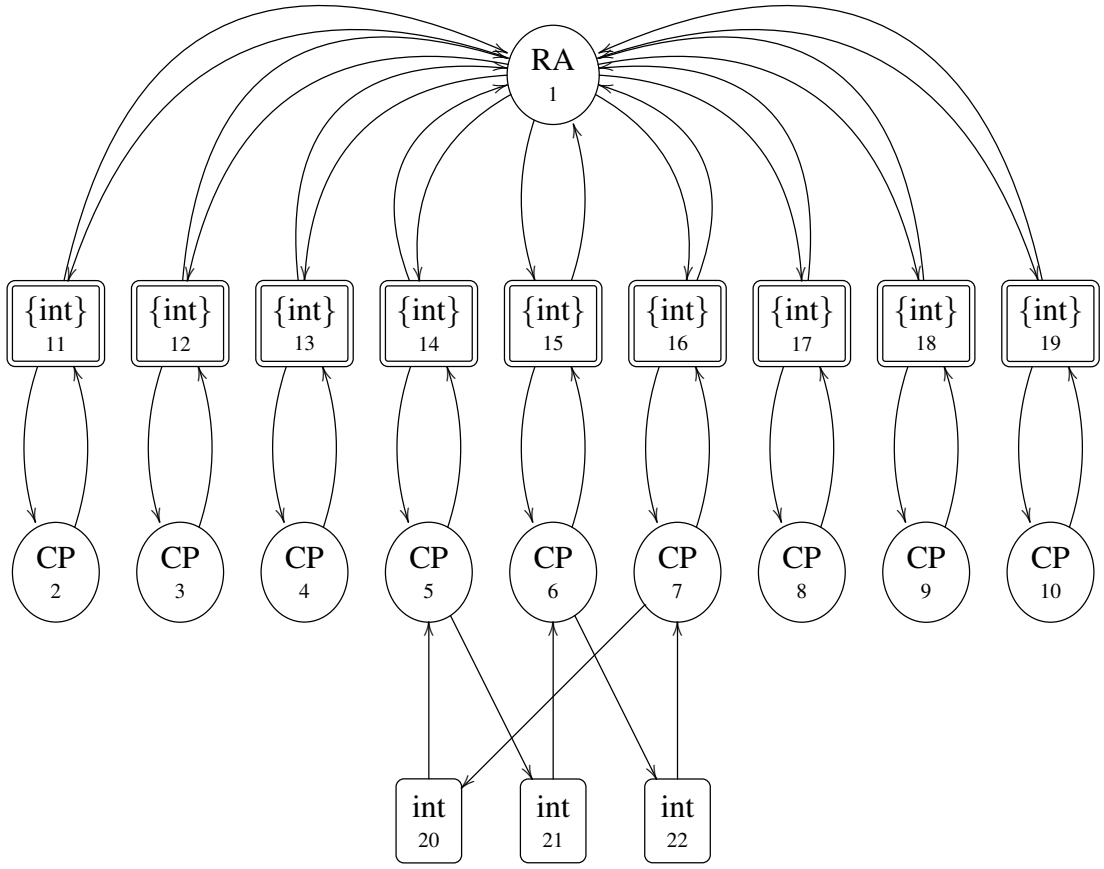


Figure 5.10: Extended channel diagram for a PSS resource allocator specification with 9 computers with the same priority level and three of the clients have been set up to share the resource.

The ECD contains 22 vertices. There are 9 identically coloured circular vertices for each computer process and a single distinctly coloured circular vertex for the resource allocator. Every process vertex has an incoming and outgoing edge to a channel vertex, caused by the communication present in the specification. As specified, three of the clients have links to global variable vertices that in the specification allows them to independently share the resources. Using integer subscripts to uniquely identify each vertex, the automorphisms of the diagram are given by the group generators

$$\text{Aut}(\text{C}(\mathcal{P})) = \{(2, 3)(11, 12), (3, 4)(12, 13), (4, 8)(13, 17), (8, 9)(17, 18), (9, 10)(18, 19), (5, 6, 7)(20, 21, 22)\}$$

Any arrangement of the process vertices with index values 2, 3, 4, 8, 9 and 10 that leaves a process vertex with incoming and outgoing edges to the same channel vertex can be provided by a permutation $\alpha \in \text{Aut}(\text{ECD}(\mathcal{P}))$. Due to the inclusion of three global variable vertices, these process vertices cannot be permuted with the remaining process vertices in the ECD. This is because there is now a cyclic relationship between computers 5, 6 and 7 due to the configuration of the additional sharing functionality provided by the global variables. The automorphism group captured by the ECD is describes as $\text{C}_3 \times \text{S}_6$. It is not possible to specify this product of symmetric and cyclic groups using scalarsets as cyclic symmetries cannot be handled by either technique.

5.1.6 Comparison of Channel Diagrams and Extended Channel Diagrams

The ECD provides several distinct advantages over both channel diagrams and scalarsets:

- The technique allows users to write specifications in a more natural manner. The user is not unduly forced to place data within a process or channel to take advantage of component symmetry.
- When a specification contains a single process it exhibits no component symmetry. By employing extended channel diagrams a large set of potential data symmetries may be detected. In this instance $|\text{Aut}(\text{ECD}(\mathcal{P}))| \geq |\text{Aut}(\text{C}(\mathcal{P}))|$.
- ECDs capture arbitrary potential data symmetries. This is a distinct advantage over a technique such as scalar sets were any data symmetry must exhibit full symmetry.
- When directly comparing the techniques on a PSS specification $\mathcal{P}$ containing both component and data symmetries the order of $\text{Aut}(\text{ECD}(\mathcal{P}))$ and $\text{Aut}(\text{C}(\mathcal{P}))$ may be equal. In this instance a permutation $\alpha \in \text{Aut}(\text{ECD}(\mathcal{P}))$ may act on a larger number of state components giving arise to the possibility of more states being mapped to a single or smaller number of representative states.
- When directly comparing the techniques on a PSS specification $\mathcal{P}$ that contains only component, symmetries, both techniques will return the same group of automorphisms. The $\text{ECD}(\mathcal{P})$ is a true extension of $\text{C}(\mathcal{P})$ and no information is lost.
- When directly comparing the techniques on a PSS specification $\mathcal{P}$ that contains more than one process, $|\text{Aut}(\text{C}(\mathcal{P}))| \geq |\text{Aut}(\text{ECD}(\mathcal{P}))|$. This is due to the fact that for any

set of n similarly typed processes the maximum size of the group of automorphisms is S_n. The inclusion of variable or channel vertices can only restrict the number of automorphisms.

The final point can be illustrated using the resource allocator example. Examining ECD associated with the specification (see Figure 5.10), the group of automorphisms was described as $C_3 \times S_6$. The channel diagram $C(\mathcal{P})$ of the resource allocator specification would be the same without the global variables and their incoming and outgoing edges. The automorphism group captured by the channel diagram is describes as S_9, as any arrangement of computer vertices can be provided by a permutation $\alpha \in C(\mathcal{P})$.

As asserted the addition of variable edges cannot increase the number of ways a group of similarly typed processes can be permuted. It is clear that $|S_9| \geq |C_3 \times S_6|$. This leads to the conclusion that all automorphisms captured by a channel diagram or extended channel diagram are not necessarily valid for reduction. This is the topic of Section 5.2.

5.1.7 Deriving an Extended Static Channel Diagram

Given a PSS specification $\mathcal{P}$, Aut(ECD($\mathcal{P}$)) can be derived from a single pass of $\mathcal{P}$. The node set and colouring can be immediately deduced from the inspection of variable declarations and the creation of processes in the **Initial** process definition statement. Edges in the diagram are generated using the algorithm presented in Figure 5.11.

The complexity of deriving ECD($\mathcal{P}$) from $\mathcal{P}$ is linear in the size of $\mathcal{P}$. We now show how the elements of Aut(ECD($\mathcal{P}$)) act on $\mathcal{P}$, and on the probabilistic model generated by the specification.

5.2 Correspondence Proof

This section contains the main result of the chapter and shows how automorphisms of an ECD($\mathcal{P}$) extracted from a PSS specification $\mathcal{P}$ can be used to define an automorphism acting on the set of states S of the associated DTMC $\mathcal{D}$ or MDP $\mathcal{M}$. Provided that the automorphisms meet a small set of restrictions then they are valid for symmetry reduction.

```
1.  for all (g → u) ∈ P
2.  {
3.    if u contains a statement referencing c ∈ V_C
4.    {
5.      for all i ∈ V_P
6.      {
7.        if u contains a channel write operation on c
8.        {
9.          E := E ∪ (i, c)
10.       }
11.       if u involves a channel read operation on c
12.       {
13.         E := E ∪ (c, i)
14.       }
15.     }
16.   }
17.   if u contains a statement referencing x ∈ V_G
18.   {
19.     for all i ∈ V_P
20.     {
21.       if u contains a statement of the form x := e for some expression e
22.       {
23.         E := E ∪ (i, x)
24.       }
25.       if u contains an update of the form y := e where expression e refers to global variable x
26.       {
27.         E := E ∪ (x, i)
28.       }
29.     }
30.   }
31. }
```

Figure 5.11: Algorithm to extract an ECD from a PSS specification in a single pass.

5.2.1 Action of Aut(ECD($\mathcal{P}$)) on $\mathcal{P}$

We now introduce the notion of equivalence between PSS specifications. Two PSS specifications P_1 and P_2 are equivalent, denoted $P_1 \equiv P_2$, if they are the same up to rearrangement of updates in commands, and of process initiation statements within the **Initial** process. Equivalent specifications define identical behaviour and it follows that the underlying probabilistic model for equivalent programs is the same.

Let $\mathcal{P}$ be a PSS specification with extended channel diagram ECD($\mathcal{P}$) and let $\alpha \in$ Aut(ECD($\mathcal{P}$))

be a permutation. To construct $\alpha(\mathcal{P})$ from $\mathcal{P}$ each channel name c and global variable name x that occur in expression e is replaced by channel name α(c) and global variable name α(x) respectively. However, if x is the name of an array, the desired location can be indexed by another variable or by a value, val $\in \{1, \ldots, n\}$. Let y be a variable of type int or pid, then x[y] is replaced by α(x)[α(y)]. If array x is indexed by a val, then x[val] is replaced by α(x)[α(val)].

Furthermore, assignment statements of the form x := val, Boolean expressions of the form x == val or val == x, where x $\in$ Var with type pid and val $\in \{1, \ldots, n\}$, are replaced by x := α(val), x == α(val) or α(val) == x respectively. Finally α acts on the order of statements that appear in the **Initial** process, with a statement that appears in position i being moved to position α(i).

Permutation α is said to be valid (for $\mathcal{P}$) if $\alpha(\mathcal{P}) \equiv \mathcal{P}$, and a subgroup H of Aut(ECD($\mathcal{P}$)) is valid (for $\mathcal{P}$) if every $\alpha \in$ H is valid for $\mathcal{P}$. It follows that if α is valid for $\mathcal{P}$ and σ_i = (g, (prob$_1$, u$_1$), ..., (prob$_k$, u$_k$)) is a command associated with Proc$_i$ in $\mathcal{P}$, then $\sigma_{\alpha(i)}$ = (α(g), (prob$_1$, α(u$_1$)), ..., (prob$_k$, α(u$_k$))) is a command associated with Proc$_{\alpha(i)}$ in $\alpha(\mathcal{P})$.

5.2.2 Action of Aut(ECD($\mathcal{P}$)) on $\mathcal{D}$ and $\mathcal{M}$

For an element $\alpha \in$ Aut(ECD($\mathcal{P}$)) we define a corresponding mapping α^* which is a permutation of the DTMC $\mathcal{D}$ or MDP $\mathcal{M}$ constructed from $\mathcal{P}$. In order to define the action of the permutation on the states we first define the action of α on the set of atomic propositions:

- For (x = val) $\in$ AP, with x $\in$ Var$_{\text{Global}}$ and type int, α(x = val) = (α(x) = val).
- For (x = val) $\in$ AP, with x $\in$ Var$_{\text{Global}}$ and type pid, α(x = val) = (α(x) = α(val)).
- For (c[k] = msg) $\in$ AP for some channel c $\in$ Var, α(c[i] = msg) = (c[α(i)] = α(msg)). Here α acts on msg by permuting the value of each field of msg.
- Let p[i].x denote local variable x of Proc$_i$ and consider a proposition of the form p[i].x = val. Since α preserves the colouring of processes according to their type and name, process α(i) is also an instantiation of Proc$_i$ and therefore the local variable p[α(i)].x exists. Therefore:
  - if x has type pid or chan, α(p[i].x = val) = (p[α(i)].x = α(val)).
  - if x has type int, α(p[i].x = val) = (p[α(i)].x = val).

Since a state is uniquely defined by its labelling function, for any s $\in$ S, α^* permutes a state's labelling function in such away that L(α^*(s)) = $\{\alpha(ap) : ap \in L(s)\}$.

Correspondence Proof

Theorem 5.1. Let $\mathcal{P}$ be a PSS specification with extended channel diagram ECD($\mathcal{P}$) and associated DTMC structure $\mathcal{D}$ or MDP structure $\mathcal{M}$. If $\alpha \in$ Aut(ECD($\mathcal{P}$)) is valid for $\mathcal{P}$, then $\alpha^* \in$ Aut($\mathcal{D}$) or $\alpha^* \in$ Aut($\mathcal{M}$)

We first consider the action of $\alpha \in$ Aut(ECD($\mathcal{P}$)) on the guards of an PSS specification $\mathcal{P}$.

Lemma 1. Let s $\in$ S and g a guard of process Proc_i. If $\alpha \in$ Aut(ECD($\mathcal{P}$)), then g holds at s if and only if α(g) holds at α^*(s).

Proof of Lemma 1. If g = true then α(g) = true and the result holds. When g = ap for ap $\in$ AP, then α(g) = α(ap). From the definition of α^*, ap $\in$ L(s) $\Leftrightarrow$ α(ap) $\in$ L(α^*(s)) and the result holds. If g = $\neg$ap, α(g) = $\neg\alpha$(ap)

$$\begin{aligned} \neg\text{ap} \in \text{L(s)} &\Leftrightarrow \text{ap} \notin \text{L(s)} \\ &\Leftrightarrow \alpha(\text{ap}) \notin \text{L}(\alpha^*(\text{s})) \\ &\Leftrightarrow \neg\alpha(\text{ap}) \in \text{L}(\alpha^*(\text{s})) \end{aligned}$$

and the result holds. □

If g = a $\vee$ b or g = a $\wedge$ b for propositional sub formulas a and b the proofs follow by structural induction.

We now show that if executing an update $u_i \in \text{Proc}_i$ results in a transition from state s to state t, executing $\alpha(u_i)$ results in a transition from α^*(s) to α^*(t). This relationship is depicted in Figure 5.12

$$\begin{array}{ccc} \text{s} & \xrightarrow{\alpha^\star} & \alpha^\star(\text{s}) \\ {\scriptstyle u_i}\downarrow & & \downarrow{\scriptstyle \alpha(u_i)} \\ \text{t} & \xrightarrow{\alpha^\star} & \alpha^\star(\text{t}) \end{array}$$

Figure 5.12: Relationship been transitions and permutations in a DTMC or MDP.

Lemma 2. Let $\alpha \in$ Aut(C($\mathcal{P}$). If s $\rightarrow$ t is a transition associated with u_i, and α^*(s) $\rightarrow$ t$'$ is the corresponding transition associated with update $u_{\alpha(i)}$, then t$'$ = α^*(t).

Proof of Lemma 2. Suppose that u_i is a global int variable update of the form x := val$'$. Consequently, (x = val) $\in$ L(s), (x = val$'$) $\in$ L(t) and L(t) = (L(s)$\backslash\{$(x = val)$\}$)$\cup\{$(x = val$'$)$\}$. Then $u_{\alpha(i)}$ is a variable update α(x := val$'$) and α(x := val) $\in$ L(α^*(s)). Therefore

$$\begin{aligned} L(t') &= (L(\alpha^*(s))\backslash\{\alpha(x = val)\}) \cup \{\alpha(x = val')\} \\ &= (L(\alpha^*(s))\backslash\{(\alpha(x) = val)\}) \cup \{(\alpha(x) = val')\} \\ &= (L(\alpha^*(t)) \\ t' &= \alpha^*(t) \end{aligned}$$

Let u_i be a global pid variable update of the form $x := val'$. Consequently, $(x = val) \in L(s)$, $(x = val') \in L(t)$ and $L(t) = (L(s)\backslash\{(x = val)\}) \cup \{(x = val')\}$. Then $u_{\alpha(i)}$ is a variable update $\alpha(x := val')$ and $\alpha(x = val) \in L(\alpha^*(s))$. Therefore

$$\begin{aligned} L(t') &= (L(\alpha^*(s))\backslash\{\alpha(x = val)\}) \cup \{\alpha(x = val')\} \\ &= (L(\alpha^*(s))\backslash\{(\alpha(x) = \alpha(val))\}) \cup \{(\alpha(x) = \alpha(val')\} \\ &= (L(\alpha^*(t)) \\ t' &= \alpha^*(t) \end{aligned}$$

Let u_i be a local int variable update of the form $p[i].x := val'$. Consequently, $(p[i].x = val) \in L(s)$, $(p[i].x = val') \in L(t)$ and $L(t) = (L(s)\backslash\{(p[i].x = val)\}) \cup \{(p[i].x = val')\}$. Then $u_{\alpha(i)}$ is a variable update $\alpha(p[i].x := val')$ and $\alpha(p[i].x = val) \in L(\alpha^*(s))$. Therefore

$$\begin{aligned} L(t') &= (L(\alpha^*(s))\backslash\{\alpha(p[i].x = val)\}) \cup \{\alpha(p[i].x = val')\} \\ &= (L(\alpha^*(s))\backslash\{(p[\alpha(i)].x = val))\}) \cup \{p[\alpha(i)].x = val')\} \\ &= (L(\alpha^*(t)) \\ t' &= \alpha^*(t) \end{aligned}$$

Let u_i be a local pid variable update of the form $p[i].x := val'$. Consequently, $(p[i].x = val) \in L(s)$, $(p[i].x = val') \in L(t)$ and $L(t) = (L(s)\backslash\{(p[i].x = val)\}) \cup \{(p[i].x = val')\}$. Then $u_{\alpha(i)}$ is a variable update $\alpha(p[i].x := val')$ and $\alpha(p[i].x = val) \in L(\alpha^*(s))$. Therefore

$$\begin{aligned} L(t') &= (L(\alpha^*(s))\backslash\{\alpha(p[i].x = \alpha(val))\}) \cup \{\alpha(p[i].x = \alpha(val'))\} \\ &= (L(\alpha^*(s))\backslash\{(p[\alpha(i)].x = \alpha(val))\}) \cup \{p[\alpha(i)].x = \alpha(val'))\} \\ &= (L(\alpha^*(t)) \\ t' &= \alpha^*(t) \end{aligned}$$

Let u_i be a channel variable update of the form $c_j!msg$. Consequently, $(c_j = [msg_1, \ldots, msg_k], len(c_j) = k) \in L(s)$, $(c_j = [msg_1, \ldots, msg_k, msg_{k+1}], len(c_j) = k + 1) \in L(t)$ and $L(t) = (L(s)\backslash\{(c_j = [msg_1, \ldots, msg_k]), len(c_j = k)\} \cup \{(\alpha(c_j = [msg_1, \ldots, msg_k, msg_{k+1}]), len(\alpha(c_j) = k + 1)\}$. Then $u_{\alpha(i)}$ is a channel write update $\alpha(c_j!msg)$ and $(\alpha(c_j = [msg_1, \ldots, msg_k]), len(\alpha(c_j) = k) \in L(\alpha^*(s))$. Therefore

$$\begin{aligned} L(t') &= (L(\alpha^*(s))\backslash\{(\alpha(c_j = [msg_1, \ldots, msg_k]), len(\alpha(c_j = k))\})\cup \\ &\quad \{(\alpha(c_j = [msg_1, \ldots, msg_k, msg_{k+1}]), len(\alpha(c_j) = k+1)\} \\ L(t') &= (L(\alpha^*(s))\backslash\{(c_{\alpha(j)} = [\alpha(msg_1), \ldots, \alpha(msg_k)]), len(c_{\alpha(j)} = k)\})\cup \\ &\quad \{(c_{\alpha(j)} = [\alpha(msg_1), \ldots, \alpha(msg_k), \alpha(msg_{k+1})]), len(c_{\alpha(j)}) = k+1)\} \\ t' &= \alpha^*(t) \end{aligned}$$

Let u_i be a channel variable update of the form c_j?msg. Consequently, ($c_j = [msg_1, \ldots, msg_k]$, $len(c_j) = k) \in L(s)$, ($c_j = [msg_1, \ldots, msg_{k-1}]$, $len(c_j) = k-1) \in L(t)$ and $L(t) = (L(s)\backslash\{(c_j = [msg_1, \ldots, msg_k]), len(c_j = k)\} \cup \{(\alpha(c_j = [msg_1, \ldots, msg_{k-1}]), len(\alpha(c_j) = k-1)\}$. Then $u_{\alpha(i)}$ is a channel write update $\alpha(c_j!msg)$ and ($\alpha(c_j = [msg_1, \ldots, msg_k])$, $len(\alpha(c_j) = k) \in L(\alpha^*(s))$. Therefore :

$$\begin{aligned} L(t') &= (L(\alpha^*(s))\backslash\{(\alpha(c_j = [msg_1, \ldots, msg_k]), len(\alpha(c_j = k))\})\cup \\ &\quad \{(\alpha(c_j = [msg_1, \ldots, msg_k]), len(\alpha(c_j) = k-1)\} \\ L(t') &= (L(\alpha^*(s))\backslash\{(c_{\alpha(j)} = [\alpha(msg_1), \ldots, \alpha(msg_k)]), len(c_{\alpha(j)} = k)\})\cup \\ &\quad \{(c_{\alpha(j)} = [\alpha(msg_1), \ldots, \alpha(msg_k)]), len(c_{\alpha(j)}) = k-1)\} \\ t' &= \alpha^*(t) \end{aligned}$$

Finally if u_i is a sequence of expressions they are all executed simultaneously and $t' = \alpha^*(t)$.

□

Proof of Theorem 5.1. We must show that $s_0 = \alpha(s_0)$ and $\forall s, t \in S, \mathbf{P}(s, t) = \mathbf{P}(\alpha(s), \alpha(t))$.

To begin, for a proposition ($x = val) \in s_0$ we must also show that $\alpha((x = val)) \in s_0$. For each variable x, ($x = x_0) \in s_0$, where x_0 is the initial value assigned to x upon declaration.

- If x is of type int then we have $\alpha(x_0) = x_0$ and $\alpha(x)$ must exist or $\alpha(\mathcal{P}) \not\equiv \mathcal{P}$). Therefore $(\alpha(x = x_0) = (\alpha(x) = \alpha(x_0)) = (\alpha(x) = (x_0))) \in s_0$.
- If x is of type pid then $\alpha(x_0)$ and $\alpha(x)$ must exist or $\alpha(\mathcal{P}) \not\equiv \mathcal{P}$). Therefore ($\alpha(x = x_0) = (\alpha(x) = \alpha(x_0)) \in s_0$.

Furthermore, the initial state, all channels are empty, so for any channel c, the propositions (c = []) and $\alpha((c = [])) = (\alpha(c) = [])$ both belong to s_0.

For the second requirement suppose that $\mathbf{P}(s, t) = 0$. Then there is no command σ such that g holds at s, and t is the result of applying an update u of σ to s. Suppose that $\mathbf{P}(\alpha^*(s), \alpha^*(t)) > 0$ then for some command $\sigma_i = (g, (p_1, u_1), \ldots, (p_k, u_k))$ of $Proc_i$ in $\mathcal{P}$, g holds at $\alpha^*(s)$ and for some j, $\alpha^*(t)$ is the result of applying u_j to $\alpha^*(s)$. But then, since α is valid, so is α^{-1} and so $\alpha^{-1}(\sigma_i)$ is a command of $\mathcal{P}$. By Lemma 1, $\alpha^{-1}(g)$ holds at s and t is the result of applying $\alpha^{-1}(u_j)$ to s. Hence $\mathbf{P}(s, t) > 0$, which is a contradiction.

If $\mathbf{P}(s, t) = p_i$. then there is a command σ such that g holds at s, and t is the result of applying an update u of σ to s. By Lemma 1, the guard $\alpha(g)$ holds at $\alpha^{\star}(s)$, and by Lemma 2, execution of the updates of $\alpha(\sigma)$ lead to state $\alpha(t)$. Furthermore, p_i is the result of an expression e formed exclusively from global or local variables of type integer. By the rules presented in Section 4.2.7, for a global or local integer y = val, $\alpha(y = val) = (\alpha(y) = val)$). It follows that $\alpha(e = p_i) = (\alpha(e) = p_i)$). Therefore $\forall s, t \in S, \mathbf{P}(s, t) = \mathbf{P}(\alpha(s), \alpha(t))$. □

5.3 Largest Valid Symmetry Group

The ECD($\mathcal{P}$) of a program is typically a small graph which can be easily extracted from the PSS specification. Additionally, checking for a permutation $\alpha \in$ Aut(ECD($\mathcal{P}$)) whether $\alpha(\mathcal{P}) \equiv \mathcal{P}$ can be implemented efficiently is the topic of Chapter 8. Thus, using Theorem 5.1, it is possible to obtain a group of model automorphisms, that are valid for symmetry reduction.

However, this group of valid symmetries may not be as large as possible, this is easily conveyed by means of an example. Recall the resource allocator example. If its specification were altered so the resource allocator blocked access from computers with pid value 3 and 8, the automorphism generators of the ECD associated with the modified specification would still be calculated as

$$\begin{aligned}\text{Aut(C}(\mathcal{P})) = \{\ &(2, 3)(11, 12), (3, 4)(12, 13), (4, 8)(13, 17), (8, 9)(17, 18),\\ &(9, 10)(18, 19), (5, 6, 7)(20, 21, 22)\ \}.\end{aligned}$$

In this instance generators that do not fix Proc_4 and Proc_8 are not valid for symmetry reduction. Let α be such a generator, the declaration blockedclient = 4 in $\mathcal{P}$ is replaced with blockedclient = $\alpha(4)$ which does not appear on the specification, thus $\alpha(\mathcal{P}) \not\equiv \mathcal{P}$. A similar argument applies to the statement blockedclient = 8. The other generators are valid for $\mathcal{P}$ and subsequently a valid group for reduction is

$$\text{H} = \{\ (2, 3)(11, 12), (9, 10)(18, 19), (5, 6, 7)(20, 21, 22)\}.$$

However, consider the group

$$\text{G} = \{(2, 3)(11, 12), (3, 9)(12, 18), (9, 10)(18, 19), (5, 6, 7)(20, 21, 22)\ \}.$$

Every generator of G is valid in $\mathcal{P}$ and H $\subset$ G. Therefore, G is not the largest valid subgroup of Aut(ECD($\mathcal{P}$)).

5.3.1 Reconstructing the Largest Valid Symmetry Group

This problem of calculating a larger group valid for reduction has previously been tackled using a group theoretic approach [30]. The algorithm for finding the largest valid subset starts with a known valid subgroup H of Aut(ECD($\mathcal{P}$)) and adds valid coset representatives to the generators of H to obtain successively larger valid subgroups. Once all coset representatives of the group H have been checked the largest valid set will have been determined. The algorithm performs badly [30] if the initial group H is small, and Aut(ECD($\mathcal{P}$)) is very large. In such cases the number of right coset representatives to consider is, in the worst case, |Aut(ECD($\mathcal{P}$))| / |H|.

We propose an alternative algorithm in which the behaviour is linked with the level of symmetry exhibited in the specification. To achieve this, our approach considers what actions are permitted in the specification that later induce an invalid permutation $\alpha \in$ Aut(ECD($\mathcal{P}$)). To prevent an invalid permutation α appearing in Aut(ECD($\mathcal{P}$)), the vertices it permutes can no longer share the same colour. The affected vertices are recoloured by searching for valid elements $\alpha \in$ Aut(ECD($\mathcal{P}$)) that permute at least one of the vertices with another whose colour is already known. If a valid element is identified, the vertices it permutes are given the known colour and if no such element is found, the unmatched vertices are each given a unique colour. In regards to algorithmic behaviour, the more symmetric the specification, the less vertex re-colourings will be required. Finally, once ECD($\mathcal{P}$) has been re-coloured, Aut(ECD($\mathcal{P}$) is the largest valid symmetry.

Specification Statements that Lead to Invalid Symmetries

An element $\alpha \in$ Aut(ECD($\mathcal{P}$)) can fail to meet the requirement that $\alpha(\mathcal{P}) \equiv \mathcal{P}$ if:

- a variable has not been symmetrically initialised (e.g. the blockclient variable in Section 5.3).
- process Proc_{i} contains an executable command σ while process $\text{Proc}_{\alpha(\text{i})}$ cannot execute the command $\alpha(\sigma)$.

Consequently in a PSS specifications identically named processes may only be partially symmetric.

5.3.2 Algorithm to Reconstruct the Largest Valid Symmetry Group

To account for partially symmetric processes the ECD has to be modified so vertices corresponding to partially symmetric processes no longer share the same colour. This will result in the transposition of the processes no longer being provided by a permutation in the graph automorphism group. To achieve this we use the following algorithm.

Let Work be the the set of all process vertices $v_i \in ECD(\mathcal{P})$ $(0 < i < |ECD(\mathcal{P}|)$. Select a process vertex $pv_i \in$ Work, remove it from the set and assign it a distinct colour. The set of process vertices X_i that it can be transposed with is calculated using the generators of Aut(ECD($\mathcal{P}$)). This facility is provided by the graph automorphism package GAP. Next locate an element $\alpha \in$ Aut(ECD($\mathcal{P}$) that transposes v_i with a vertex in $xv_i \in X_i$. If applying the permutation to the specification results in $\alpha(\mathcal{P}) \equiv \mathcal{P}$ then the processes are symmetric and the vertices should share the same colour. On the other hand if $\alpha(p) \not\equiv p$ then the process are only partially symmetric and the vertices must have a distinct colour. If the process share the same colour, then the process xv_i is removed from the sets X_i and Work. If it must have a different colour then it is only removed from the set X_i. An element $\alpha \in$ Aut(ECD($\mathcal{P}$)) that permutes pv_i with a vertex remaining in X_i is obtained. When X_i is empty all the vertices that share the same colour as pv_i have been determined. This process is repeated for every vertices remaining in Work.

The final output of the algorithm are sets of vertices that must be identically coloured. No finer colouring of the sets exists and consequently the automorphisms of the newly coloured ECD is the largest valid symmetry.

Lemma 3. α is a valid permutation if and only if α^{-1} is valid.

Lemma 4. If α and β are valid permutations. The permutation $\alpha\beta$ is valid.

Lemma 5. If α is a valid permutation and β is an invalid permutations. The permutation $\alpha\beta$ is invalid.

Proof. Let Z_i $(0 < i < |ECD(\mathcal{P}|))$ be a collection of sets with each set containing a set of process vertices that share the same colour.

Let a_1 be the first element in the set Z_1. If Z_1 contains more elements b_1, c_1 ..., then a_1 has been directly compared for validity with every other element in the set. Even though, elements b_1 and c_1 have not been directly compared the permutation $(b_1, c_1) = (b_1, a)_1^{-1}(a_1, c_1)$. By Lemma 3 and Lemma 4$(b_1, c_1)$ is valid and the vertices corresponding to b_1 and c_1 share the same colour. A similar argument holds between any two elements in the set and it follows that all elements in the set share the same colour.

Let a_2, b_2, ... be element in the set Z_2. During the execution of the algorithm the permutations that provided the transposition of vertices (a_1, a_2), (a_1, b_2), ... were found to be invalid. Any other element $\beta \in Z_1$ can reach any element $\gamma \in Z_2$ with the permutation $(\beta, \gamma) = (\beta, a_1), (a_1, \gamma)$ which is a valid permutation followed by an invalid permutation. By Lemma 5 this is an invalid permutation and Lemma 3 shows the symmetric case is invalid. Therefore, there is no valid permutation to transpose any vertex in Z_1 and Z_2. The vertices in the two sets must share different colours and a similar argument holds between any two sets and it follows that all sets must have a distinct colouring.

As there are no valid permutations that can transpose any two vertex in any two different sets, the graph can have no finer colouring. The automorphisms of the diagram are therefore the largest valid symmetry. □

Example

Applying the algorithm to modified resource allocator specification splits the processes into 3 sets: Z_1 ={2, 3, 9, 10}, Z_2 = {4, 8}, Z_3 ={5, 6, 7}. The result of using these sets to recolour the ECD is shown in Figure 5.13. Using integer subscripts to uniquely identify each vertex, the automorphisms of the diagram are given by the generators

$$\begin{aligned}\mathrm{Aut}(C(\mathcal{P})) = \{&(2, 3)(11, 12), (3, 9)(12, 18), (9, 10)(18, 19),\\ &(4, 8)(13, 17)(5, 6, 7)(20, 21, 22)\}.\end{aligned}$$

which is the largest valid symmetry group.

Complexity of the Algorithm

In the worst case the algorithm will execute its outer loop n times and require a total of n^2 process comparisons. This behaviour is exhibited if there are n identically named processes and in fact none of them are symmetric. Therefore, for each loop iteration only one process is removed from the work set. As the symmetry between identically named processes increases, the average number of vertices removed from the work set on each iteration will also increase. It follows that a more symmetric specification will require on average a smaller number of comparisons. In Section 8.2.4 experimental results are provided that detail the run time performance of our symmetry detection techniques, including this algorithm, when applied to a variety of PSS specifications.

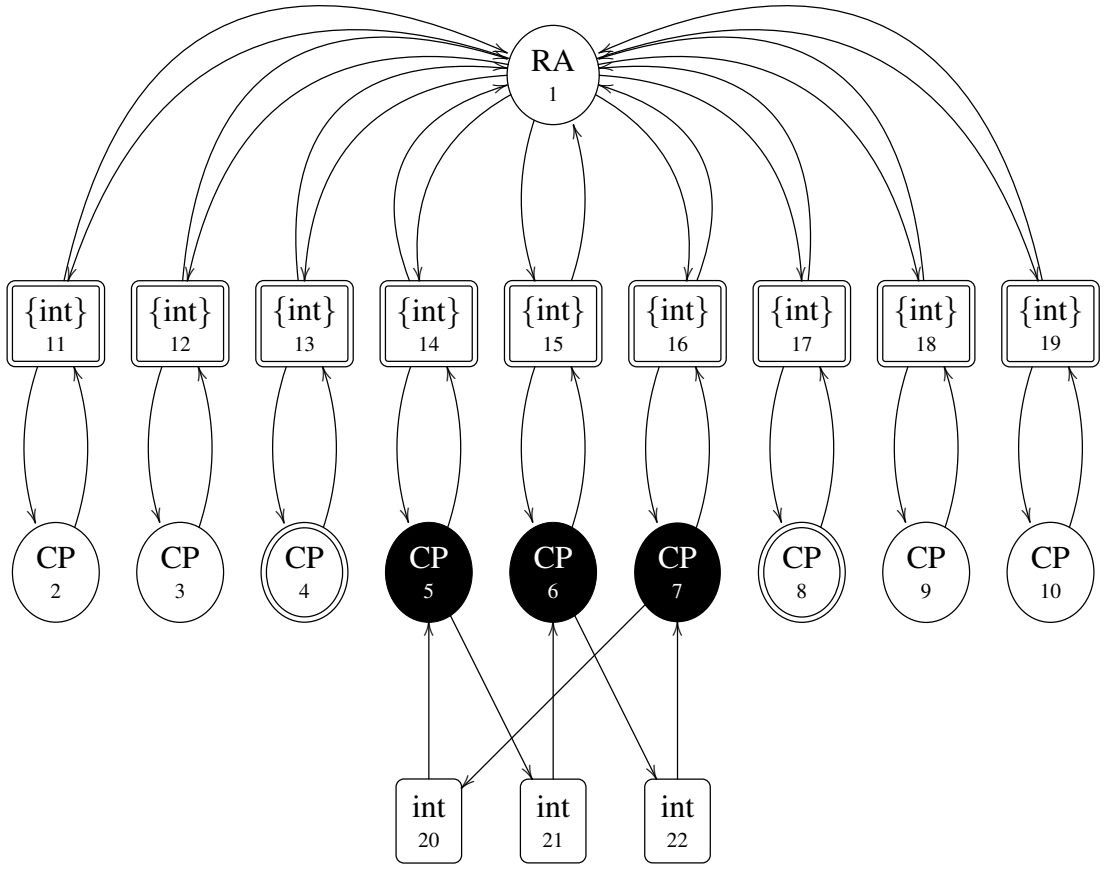


Figure 5.13: A recoloured extended channel diagram for a PSS resource allocator specification.

5.4 Summary

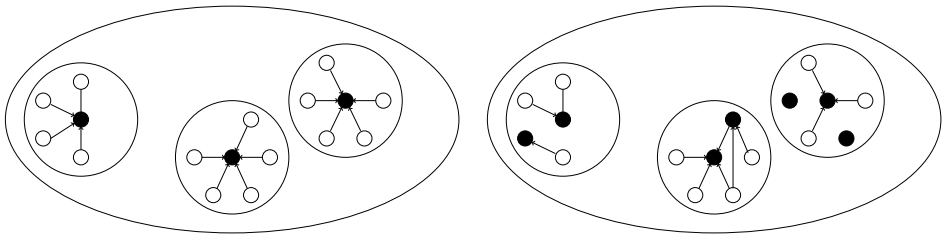
We have introduced the extended channel diagram approach, which is the first technique we know of that can detect arbitrary component and data symmetries directly from a probabilistic specification. To summarise, the ECD approach allows a set of potential automorphisms of a model associated with a PSS specification to be generated. Provided that the automorphisms meet a small set of restrictions then they are valid for symmetry reduction. From the set of valid automorphisms we show how a potentially larger set of automorphisms valid for reduction can be calculated.

CHAPTER 6

Computing a Representative State

Once a set of symmetries viable for reduction has been deduced, they can be applied to a state to transform it into another state in its orbit. A common approach to symmetry reduction is to use a representative function, *rep*(s , G) that, given a state s and a permutation group G, returns a state from the orbit of s. Clearly, if *rep*(s, G) = *rep*(t, G) then states s and t are equivalent. The goal of a representative function is to always return the same or a small number of states in a state orbit.

The single state scenario is depicted in Figure 6.1a and the multiple representatives scenario in Figure 6.1b. Single or multiple representatives are shown as black disks and a single state may be associated with multiple representatives. Permitting multiple representatives per orbit is a cosmetic change that simplifies the orbit problem only to the extent that the benefits of symmetry reduction are diminished.



(a) Single representative (b) Multiple representative

Figure 6.1

In this chapter we present various strategies to calculate a representative state and begin with a simple exhaustive calculation approach. An implementation enhancement that improves the average runtime complexity of exhaustive search is presented in Section 6.2 and the problem is explored from a new angle by mapping it to a constraint satisfaction problem in Section 6.3. Section 6.4 covers groups where enumeration is infeasible and considers a tailored made local search algorithm to map a state to a small number of representatives. Finally the chapter considers exploiting structural properties of the group G to calculate a state representative. We suggest techniques for the fully symmetric group, cyclic groups and groups that can be decomposed as an internal direct product or as an internal semi direct product.

In Chapter 5 we covered a technique to automatically calculate a set of valid symmetries from a PSS specification. However, the techniques presented in the remainder of this chapter make no assumption about the relationship between a state s and how the group G was calculated.

6.1 A Model of Computation Without References

To simplify the presentation of results a state s is presented as an array of integer variables. Each integer corresponds to a single state component such as a process, global variable or channel. Formally, a system of n components can be defined in terms of a set of n variables, each with a finite domain $L \subset \mathbb{Z}$.

Let $V = \{v_1, v_2, \ldots, v_l\}$ be the set of variables in a state s, and D_i the finite domain of v_i ($1 \leq i \leq l$). By letting m equal the largest domain value, $m = \max(|D_1|, |D_2|, \ldots, |D_l|)$, and with no loss of information $D_i = \{1, 2, \ldots, m\}$. Furthermore, V can be partitioned into n subsets, $V_1, V_2, \ldots, V_n$, for some $n > 0$, where each subset V_i contains variables that constitute a single state component i. For $1 \leq i \leq n$, $V_i = \{v_{i,1}, v_{i,2}, \ldots, v_{i,l_i}\}$ and for some $v_{i,j} \in V$ and $l_i > 0$, $\sum_{i=1}^{n} l_i = l$. It follows

- if state component i is a variable, the set V_i contains a single variable.
- if state component i is a buffered channel with capacity t, the set V_i contains t variables, one for each buffer position.
- if state component i is a process then the set V_i contains all local variables declared within that process. Then $D = \prod_{i=1}^{n} \prod_{j=1}^{l_i} D'_{i,j}$ where $D'_{i,j}$ is the domain of $v_{i,j}$, for $1 \leq i \leq n$, $1 \leq j \leq l_i$.

Let f be the size of the largest subset V_i, a map $\theta_i : D \rightarrow 1, 2, \ldots, m^f$ can be defined for all subsets in the following manner. For any state $s = (d_{1,1}, d_{1,2}, \ldots, d_{1,l}, \ldots, d_{n,1}, d_{n,2},$

$\ldots, d_{n,l}) \in D$, $\theta_i(d) = \sum_{j=1}^{l_i} d_{i,j} m^{j-1}$. It follows that a state s can be defined as an array of integers, $\theta(s) = (\theta_1(s), \theta_2(s), \ldots, \theta_n(s))$.

Therefore, given an array of integers that represent state $s = (x_1, x_2, \ldots, x_n) \in L^n$, a group $G \leq S_n$ and an element $\alpha \in G$, the state $\alpha(s)$ can be defined as $\alpha(s) = (x_{\alpha^{-1}(1)}, x_{\alpha^{-1}(2)}, \ldots, x_{\alpha^{-1}(n)})$.

Finally, as a state is mapped to a vector of integers, there is a natural lexicographical total ordering on states. Throughout this chapter we use $\min[s]_G$ to denote the lexicographical minimal state in the orbit of state s under group G.

6.2 Full Enumeration

A straightforward and well known approach to computing the state $\min[s]_G$ is to consider every state in $[s]_G$, and return the lexicographically smallest. This is achieved by obtaining all elements $\alpha \in G$ and applying them in turn to state s to compute $\alpha(s)$. This results in the computation of $[s]_G$ and from this set the state $\min[s]_G$ can be returned as the representative state. This strategy provides exact symmetry reduction but is only feasible if the number of elements in G is small. An algorithm capable of performing this strategy is shown in Figure 6.2.

```
1. Procedure minimise(State original, Group G)
2. {
3.     State temp := null
4.     minimum := original
5.     for (Permutation α : G)
6.     {
7.         State temp := apply(original, α)
8.         if (temp < minimum)
9.         {
10.            minimum := temp
11.        }
12.    }
13.    return minimum
14.}
```

Figure 6.2: An algorithm to compute the lexicographical minimal state using full enumeration.

This algorithm is well documented and implemented in several model checkers to provide symmetry reduction [13]. By examining the literature and available source code the im-

plementations do not exploit a known property of state space exploration. For all states, excluding the initial state, the components updated between transitions are known. If an element $\alpha \in$ G does not transpose at least one of the updated components, the calculation of α(s) is not required.

Example 1. Let state s = [1, 3, 5, 6] and state t = [1, 7, 5, 6] and assume that t can be reached from s through an update to component s[2]. It is clear that state components t[1], t[3] and t[4] are still in sorted order. For group G = S_4, Figure 6.3 lists all elements in G and those whose only action is to reorder sorted elements in state vector t are stricken through.

~~()~~, ~~(3,4)~~, (2,3), (2,3,4), (2,4,3), (2,4), (1,2), (1,2)(3,4), (1,2,3), (1,2,3,4), (1,2,4,3), (1,2,4), ~~(1,3,2)~~, (1,3,4,2), ~~(1,3)~~, ~~(1,3,4)~~, (1,3)(2,4), (1,3,2,4), (1,4,3,2), (1,4,2), ~~(1,4,3)~~, ~~(1,4)~~, (1,4,2,3), (1,4)(2,3)

Figure 6.3: A list of elements in the Group S_4. Stricken through elements are not applied to the state.

To implement this amendment all elements in G are independently examined before state space exploration. If $\alpha \in$ G transposes component i, it is placed in a set moved_i. During search if state t contains updated components indexed by values n, m and o, the elements in the set $\text{moved}_n \cup \text{moved}_m \cup \text{moved}_o$ are applied to state t. The pre-calculation of the sets moved_i is feasible as the total number of elements in G is small. However, significant savings can be made in the time taken for state space exploration as there is no correlation between the order of |G| and the number of states in the state space.

6.2.1 Applying a Permutation to a State

Line 7 of the algorithm shown in Figure 6.2 makes use of a function *apply*(State, Permutation) that applies a permutation α to a state s. Two ways in which this function can be implemented are: as a series of transpositions, or as a single direct application of the permutation α to the state.

Application as a Series of Transpositions

Several implementations apply permutations as a series of transpositions [13, 35], as any permutation $\alpha \in S_n$ can be represented as a product of at most n – 1 transpositions [55]. Given such a representation for α, the state α(s) can be computed by applying a series of transpositions in sequential order. This approach is illustrated in the algorithm presented in Figure 6.4.

```
1.Procedure apply(State original, Permutation permutation)
2.{
3.    State result := original
4.    int position := 0
5.    while (position < permutation.length)
6.          if (position ≠ permutation[position])
7.          {
8.             swap(result, position, permutation[position])
9.             swap(permutation, position, permutation[position])
10.         } else
11.         {
12.            position++
13.         }
14.   }
15.   return result
16.}
```

Figure 6.4: Algorithm to apply a permutation as a series of transpositions.

The algorithm covers the calculation of transpositions from a permutation and the application of the transpositions to a given state. To apply the transposition the algorithm makes use of a simple procedure that swaps two elements in an array. This is detailed in Figure 6.5 and requires an overhead of O(1) additional space.

```
1.Procedure swap(Array original, int i, int j)
2.{
3.    int temp = original[j]
4.    vector[j] = vector[i]
5.    vector[i] = temp
6.}
```

Figure 6.5: Algorithm to transpose two element in an array.

The *apply*(Array, Permutation) function assumes that a permutation and a state are given as an array of integers. The function iterates over all elements in the permutation (line 5) and determines if the indexed value is equal to the index (line 6). If this is true, the state component is in the correct position and the algorithm moves to check the same condition on the next index (line 12). In the case where the index and the indexed value are not equal a transposition must be applied to both the state and permutation arrays (lines 8 and 9). The elements to be transposed are: the element residing in the current index, and the element indexed by this value in the permutation array.

After application of the *swap*(Array, int , int) procedure the algorithm checks if the value of the current element is equals to its index (line 5). As before, if true the algorithm moves to

the next index and if false another transposition must be applied. When the function reaches the final index the permutation has been successfully applied as a series of transpositions.

Example 2. Let the state s = [5, 9, 2, 1, 3, 6] and the desired permutation of indices is p = [5, 3, 2, 1, 0, 4].

- Starting from the initial position in the permutation vector, the index (0) is not equal to the value it indexes (5). A transposition of the elements in positions 0 and 5 of the state and permutation array is required. After application the element residing in index 5 has been placed in its final position.

$$\text{p} = [4, 3, 2, 1, 0, 5]$$
$$\text{s} = [6, 9, 2, 1, 3, 5]$$

- The function checks that the index (0) in the permutation vector is not equal to the value it indexes (4). A transposition of the elements in positions 0 and 4 of the state and permutation array is required. After application the element residing in index 4 has been placed in its final position.

$$\text{p} = [0, 3, 2, 1, 4, 5]$$
$$\text{s} = [3, 9, 2, 1, 6, 5]$$

- The function checks and the index (0) in the permutation vector is equal to the value it indexes (0). The element residing in index 0 of the state array is in the correct position and the algorithm moves to compare the next index.
- The function checks that the index (1) in the permutation array is not equal to the value it indexes (3). A transposition of the elements in positions 1 and 3 of the state and permutation vector is required. After application the element residing in index 3 has been placed in its final position.

$$\text{p} = [0, 1, 2, 3, 4, 5]$$
$$\text{s} = [3, 1, 2, 9, 6, 5]$$

- The function checks that the index (1) in the permutation array is equal to the value it indexes (1). The element residing in index 1 of the state vector is in the correct position and the algorithm moves to compare the next index.
- The procedure will terminate after checking all remaining permutation array indices and concluding they equal the value of the element they index.

Upon termination it is clear that all elements in the state have been permuted to the correct position by applying a series of transpositions. As each swap operation places at least one component in the correct position, and the final swap places two components in the correct position, the application of a permutation to a state requires no more than N−1 transpositions.

```
1. Procedure apply(State original, Permutation permutation)
2. {
3.     State result := State(original.length)
4.     for (int i = 0; i < original.length; i++)
5.     {
6.         result[permutation[i]] := original[i]
7.     }
8.     return result
9.}
```

Figure 6.6: Application of a permutation directly to a state vector

Direct Application of a Permutation to a State

An alternative approach is to apply the permutation directly to a state. This approach is illustrated in the *apply*(State, Permutation) procedure presented in Figure 6.6.

The *apply*(State, Permutation) procedure assumes that a permutation and a state are given as an array of integers. The function generates a new result array with the same length as the state array (line 3). Subsequently, the function iterates over all elements in the state array (line 4) and copies the element directly to its final position in the copy array. The final position of element i is determined by the value of the element i indexes in the permutation array (line 6). The direct application of a permutation requires a single memory allocation and n operations to copy each element to the correct position in the new result state.

Example 3. Let the state s = [5, 9, 2, 1, 3, 6] and the desired permutation of indices is p = [5, 3, 2, 1, 0, 4].

- A new empty state array result = [null, null, null, null, null, null] is created with the same length as state array s
- Index 0 indexes into the state array (5) and the permutation array (5). This determines that the function copies the state component with value 5 into index 5 in the state array result.

  result = [null, null, null, null, null, 5]

- Index 1 indexes into the state array (9) and the permutation array (3). This determines that the function copies the state component with value 9 into index 3 in the state array result.

  result = [null, null, null, 9, null, 5]

- Index 2 indexes into the state array (2) and the permutation array (2). This determines that the function copies the state component with value 2 into index 2 in the state array result.

  result = [null, null, 2, 9, null, 5]

- Index 3 indexes into the state array (1) and the permutation array (1). This determines that the function copies the state component with value 1 into index 1 in the state array result.

  result = [null, 1, 2, 9, null, 5]

- Index 4 indexes into the state array (3) and the permutation array (0). This determines that the function copies the state component with value 3 into index 0 in the state array result.

  result = [3, 1, 2, 9, null, 5]

- Index 4 indexes into the state array (6) and the permutation array (4). This determines that the function copies the state component with value 6 into index 4 in the state array result.

  result = [3, 1, 2, 9, 6, 5]

Upon termination it is clear that all components in the state have been permuted to the correct position by directly applying the permutation to the state array.

Comparision of Permutation Applications

Preliminary experiments indicated, that for sets of randomly generated states and permutations, the direct application of elements becomes more efficient as the number of state component increases. However, the number of components in a state exceeded 500 before a statistically significant runtime difference was noted. This size of state is unrealistic in model checking due to the state space explosions problem and in practice, there may be no difference between the techniques. In Chapter 8 we repeat these experiments on a selection of PSS specifications with elements generated form the specification symmetry group.

6.3 Full Enumeration as a Constraint Satisfaction Problem

In this section we consider a novel approach to calculating $\min[s]_G$ by mapping the problem to a constraint satisfaction problem (CSP). This approach may yield runtime improvements as $\min[s]_G$ can be determined using search and propagation techniques common to the field of constraint programming [90]. To begin, some basic definitions are introduced.

Definition. A constraint satisfaction problem consists of:

- a set of variables $X = \{x_1, \ldots, x_n\}$.
- each variable x_i has a finite domain D_i of possible values.
- and a set of constraints restricting the values that variables can simultaneously take.

6.3.1 A Simple Mapping

From the definition, a CSP requires a set of variables each of which is associated with a set of possible values. Given a state array of length n where $n > 0$ our mapping to a CSP consists of a set of variables $X = \{x_1, \ldots, x_n\}$. To determine the domain D_i of variable x_i, all elements $\alpha \in G$ are examined. For every $\alpha(s)$ the state component residing in index i ($1 \leq i \leq n$) is added to the domain D_i of variable x_i.

Example 4. Let the state s = [5, 9, 2, 1, 3, 6] and the group G contain the permutations [1, 0, 3, 2, 4, 5], [2, 1, 0, 3, 4, 5], [2, 3, 0, 1, 4, 5] and the identity permutation [0, 1, 2, 3, 4, 5]. Then the CSP contains the following set of variables and domains:

$$x_0 = \{2, 5, 9\},\ x_1 = \{1, 5, 9\},\ x_2 = \{1, 2, 5\},\ x_3 = \{1, 2, 5, 9\},\ x_4 = \{3\},\ x_5 = \{6\}$$

Similarly, constraints are constructed by examining how elements in G act on the components in a state s. If a permutation $\alpha \in G$ moves a component residing in position s[i] to position $\alpha(s[i]) = s[j]$, then variable $v_j == s[i]$ under this permutation. This can be restated as $v_j == \alpha^{-1}(s[j])$.

It follows, that a permutation defines the values that variables may take. This information can be used to construct a constraint stating, for all $\alpha_i \in G$ ($1 \leq i \leq |G|$);

$$\begin{gathered} v_0 == \alpha_1^{-1}(s[0])\ \&\&\ v_1 == \alpha_1^{-1}(s[1])\ \&\&\ \ldots \&\&\ v_n == \alpha_1^{-1}(s[n])\ \big|\big| \\ v_0 == \alpha_2^{-1}(s[0])\ \&\&\ v_1 == \alpha_2^{-1}(s[1])\ \&\&\ \ldots \&\&\ v_n == \alpha_2^{-1}(s[n])\ \big|\big| \\ \vdots \\ v_0 == \alpha_n^{-1}(s[0])\ \&\&\ v_1 == \alpha_n^{-1}(s[1])\ \&\&\ \ldots \&\&\ v_n == \alpha_n^{-1}(s[n]) \end{gathered}$$

Example 5. For a state s = [5, 9, 2, 1, 3, 6] and permutations [1, 0, 3, 2, 4, 5], [2, 1, 0, 3, 4, 5], [2, 3, 0, 1, 4, 5] and [0, 1, 2, 3, 4, 5], the following constraint is created.

$$\begin{aligned}
&x_0 == 9 \;\&\&\; x_1 == 5 \;\&\&\; x_2 == 1 \;\&\&\; x_3 == 2 \;\&\&\; x_4 == 3 \;\&\&\; x_5 == 6 \;||\\
&x_0 == 2 \;\&\&\; x_1 == 9 \;\&\&\; x_2 == 5 \;\&\&\; x_3 == 1 \;\&\&\; x_4 == 3 \;\&\&\; x_5 == 6 \;||\\
&x_0 == 2 \;\&\&\; x_1 == 1 \;\&\&\; x_2 == 5 \;\&\&\; x_3 == 9 \;\&\&\; x_4 == 3 \;\&\&\; x_5 == 6 \;||\\
&x_0 == 5 \;\&\&\; x_1 == 9 \;\&\&\; x_2 == 2 \;\&\&\; x_3 == 1 \;\&\&\; x_4 == 3 \;\&\&\; x_5 == 6
\end{aligned}$$

This constraint defines the set of values that variables can simultaneously take, and an assignment of variables that satisfy the constraint is equivalent to a state $\alpha(s)$. At this stage all the elements required to map the calculation of $S_{[G]}$ into a CSP have been provided. A simple method of finding a solution to a CSP is depth-first search and in the field of constraint programming this is called simple backtracking search. An outline of the algorithm is given in Figure 6.7.

1. Select a variable that has not been assigned a value.
2. Assign it a value from its domain.
3. Test if the current partial solution satisfies all relevant constraints:
   - if true, return to step 1.
   - if false, choose another value from this variables domain; if there are no more values, backtrack to a previous variable which still has values to be tried.

Figure 6.7: Outline of a simple backtracking search algorithm.

The backtracking search algorithm continues until all variable value assignments that satisfy the set of constraints have been determined. These solutions can subsequently be compared and the minimal lexicographical solution returned as the state $\min[s]_G$. For the variables and domains given in Example 4 and the constraint calculated in Example 5, Figure 6.8 shows the state space searched by the simple backtracking algorithm to provide all solutions of the CSP.

6.3.2 Adding Further Constraints

As the ultimate goal is to calculate the state $\min[s]_G$, the variable v_0 should never be assigned any value other than the smallest value in its domain. This can be achieved by posting the constraint

$$v_1 = \text{minimum}(\alpha_1^{-1}(\text{vec}_1), \alpha_2^{-1}(\text{vec}_1) \ldots \alpha_n^{-1}(\text{vec}_1))$$

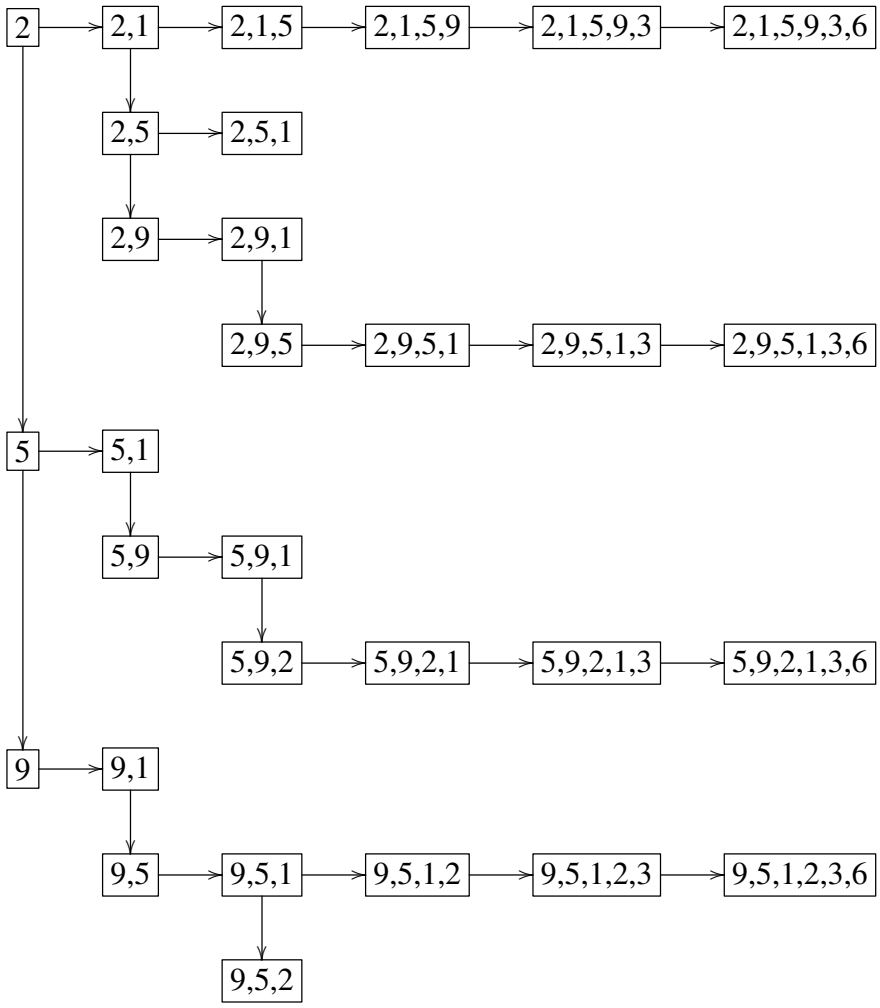


Figure 6.8: The state space searched by the simple backtracking algorithm to provide all solutions to the CSP. Where the value in the first position is assigned to variable x_0, the value in the second position is assigned to variable x_1, ...

or wiping all values, except the smallest, from the domain of v_0.

For the variables and domains given in Example 4, the constraint calculated in Example 5 and when the domain of v_0 has been modified to contain only the smallest value, Figure 6.9 show the state space searched by the simple backtracking algorithm. In this instance only two solutions have to be compared to determine the lexicographical minimum state.

A problem with this simple update, is that a group G may not transpose the first component to reside in an alternative position. In this instance the domain of v_0 is a single value and the size of the domain cannot be reduced. Therefore, the application of the update will not reduce the required search effort. This problem can be solved by generalising the approach to remove all values except the smallest from the first variable to have domain size greater than one.

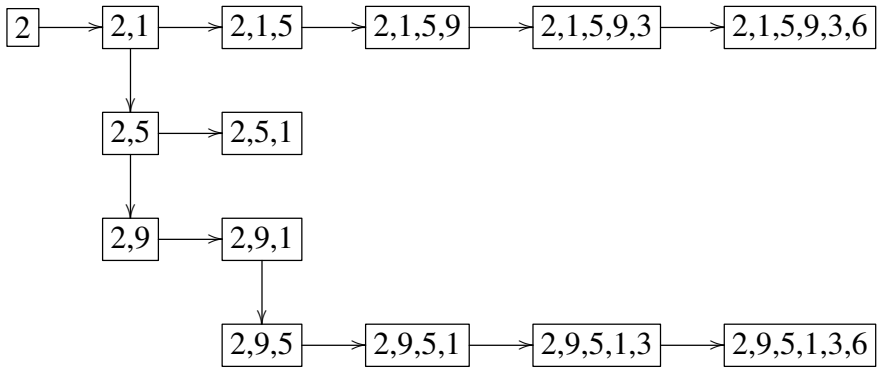


Figure 6.9: The state space searched by the simple backtracking algorithm to provide all solutions to the CSP. Where the value in the first position is assigned to variable x_0, the value in the second position is assigned to variable $x_1, \ldots$

As previously stated, only permutations that transpose state components updated during a transition need be applied. This information can be incorporated into the CSP as follows. If a component residing in position s[i] of the state array has been updated, then the constraint

$$v_i \neq s[i]$$

requires any solution of the CSP to assign its current value to another variable. In other words the component must be moved to a different position in the state.

If no valid solution is found, the current state must be the lexicographical minimum. In the event that more than one component is updated, then for every updated component s[j], s[k] ... s[l] the following constraint is posted;

$$v_k \neq s[k] \;||\; v_j \neq s[j] \;||\; \ldots \;||\; v_l \neq s[l]$$

Therefore, at least one updated component must be moved to a different position in the state. As before, if no valid solution is found the current state is the lexicographical minimum. In Example 5 the lexicographical minimal solution was equivalent to the state [2, 1, 5, 9, 3, 6]. If a transition from this state updated component 2, creating state [2, 1, 7, 9, 3, 6], the CSP would contain the following variables with domains;

$$x_0 = \{2\},\ x_1 = \{1, 7, 9\},\ x_2 = \{1, 2, 7\},\ x_3 = \{1, 2, 7, 9\},\ x_4 = \{3\},\ x_5 = \{6\},$$

and constants;

- $x_0 == 9$ && $x_1 == 7$ && $x_2 == 1$ && $x_3 == 2$ && $x_4 == 3$ && $x_5 == 6$ ||
  $x_0 == 2$ && $x_1 == 9$ && $x_2 == 7$ && $x_3 == 1$ && $x_4 == 3$ && $x_5 == 6$ ||

$x_0 == 2 \;\&\&\; x_1 == 1 \;\&\&\; x_2 == 7 \;\&\&\; x_3 == 9 \;\&\&\; x_4 == 3 \;\&\&\; x_5 == 6 \;||$
$x_0 == 7 \;\&\&\; x_1 == 9 \;\&\&\; x_2 == 2 \;\&\&\; x_3 == 1 \;\&\&\; x_4 == 3 \;\&\&\; x_5 == 6$

- $v_2 \neq 7$

Figure 6.10 illustrates the state space searched by the simple backtracking algorithm to provide all solutions to the CSP. As no solution was found, the current state is the lexicographical minimum.

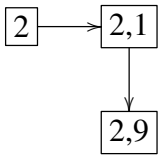


Figure 6.10: The state space searched by the simple backtracking algorithm to provide all solutions to the CSP. Where the value in the first position is assigned to variable x_0, the value in the second position is assigned to variable x_1, ...

6.4 A Representative Function for Large Groups

If a symmetry group G contains a large number of elements, full enumeration or strategies based on this approach are no longer computationally feasible. This scenario requires the use of a representative function that attempts to incrementally move a state towards the lexicographical minimum using only a subset of element in G.

6.4.1 Local Search for Large Groups

Given a starting state s and a subset of elements in a group suitable for symmetry reduction, H $\subset$ G. Figure 6.11 outlines a procedure that attempts to move a state towards the lexicographical minimum.

The *localSearch*(State, Group) function operates by applying all available elements to the current state and returning the minimal state from this set (line 6). This operation can be provided using the *minimise*(State, Group) function detailed in Figure 6.2. If a lexicographically smaller state is calculated the operation is repeated using the new state. This continues until the *minimise*(State, Group) function fails to return a state lexicographically smaller than the current state (line 8).

```
1. Procedure localSearch(State current, Group H)
2. {
3.     State update := current;
4.     do
5.     {
6.         current := update;
7.         update := minimise(current, H)
8.     }  while (update ≠ current)
9.     return update
10.}
```

Figure 6.11: Local search of $[s]_G$ using elements in $H \subset G$.

Generating a Subset of Elements

Using GAP, a subset of element in a group G can be randomly generated and used to minimise the current state. However, all the generated elements are not equally likely to progress a given state towards the lexicographical minimum. The logic behind this assertion is that an element that transposes the two left most components in a state will make less progress than an element that transposes a selection of components towards the left hand side of the state.

To justify this claim 100 elements were randomly generated from the group S_{10} and applied to $100,000$ randomly generated states of length 10 using the *localSearch*(State, Group) function. During search the number of times the application of an element produced the update state in line 7 of the function was tracked. Figure 6.12 provides an ordered plot showing how many times a specific element was used.

It is clear that a small number of elements are used significantly more than the others. Therefore, an improvement could potentially be made by replacing the less successful elements with new ones as search progresses. Two approaches we consider to provide replacement elements are:

- randomly select new elements from the group G.
- use the most successful elements to generate new elements.

We propose that after every ten thousand states, elements that produced a minimal state less than the mean number of times are replaced. Replacing these elements with a new selection of random elements is self explanatory and no further detail is provided.

In the second technique the remaining successful permutations are multiplied with random elements from the group G until a total of a 100 elements are available. As these new

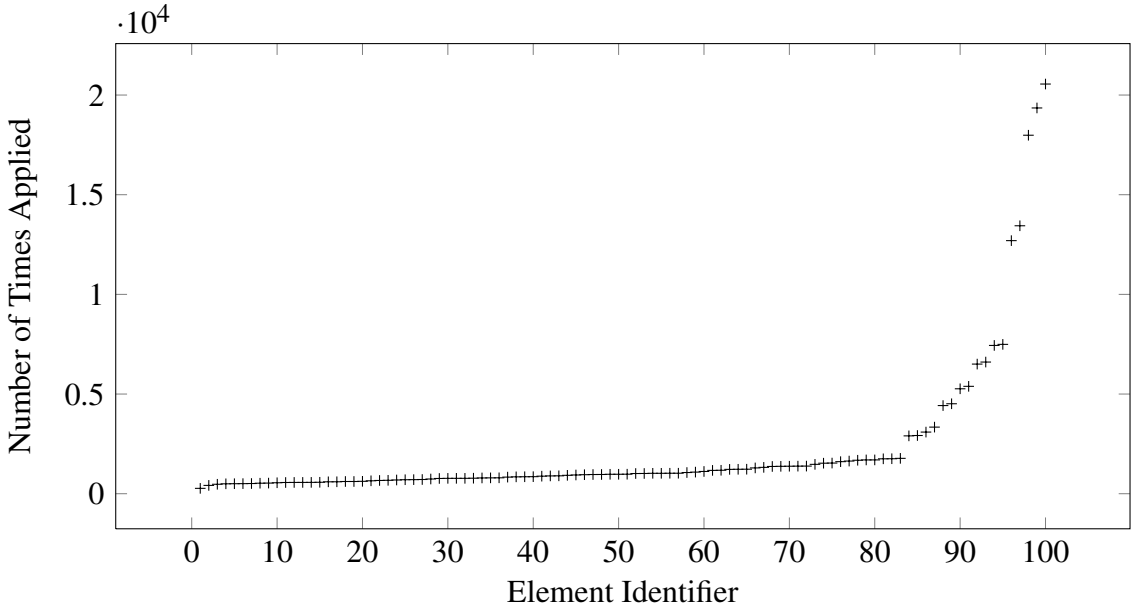


Figure 6.12: Number of times an element is applied during local search.

elements contain some of the previously successful transpositions, it is hypothesised that these elements will be better suited to mapping a state towards the lexicographical minimum.

To compare the two techniques, a set of 100 elements are created and both techniques are applied to the same set of 1,000,000 randomly generated states. Upon termination both techniques will have created a new set of 100 elements tailored to the problem instance. Figure 6.13 provides an ordered plot showing how many times elements from both sets were selected when applied to the set of 1,000,000 states.

While both techniques provide a more rounded set of elements compared to the initial set, it is clear that using previously successful elements to generate new elements results in the most general set of permutations. In this section we have focused on randomly generated instances of the Constructive Orbit Problem (see Section 3.4), Chapter 8 covers the application of the techniques to real models and shows that the more general sets produce smaller quotient structures.

6.5 Exploiting The Structure of a Group

The previously presented techniques have been general and applicable for any group G. However, information about the structure of G may allow specialised representative functions to be designed that solve the COP faster than techniques based on full enumeration. We now detail techniques that efficiently solve the COP when G is the fully symmetric group, isomorphic to the fully symmetric group, a cyclic group or can be decomposed as an internal

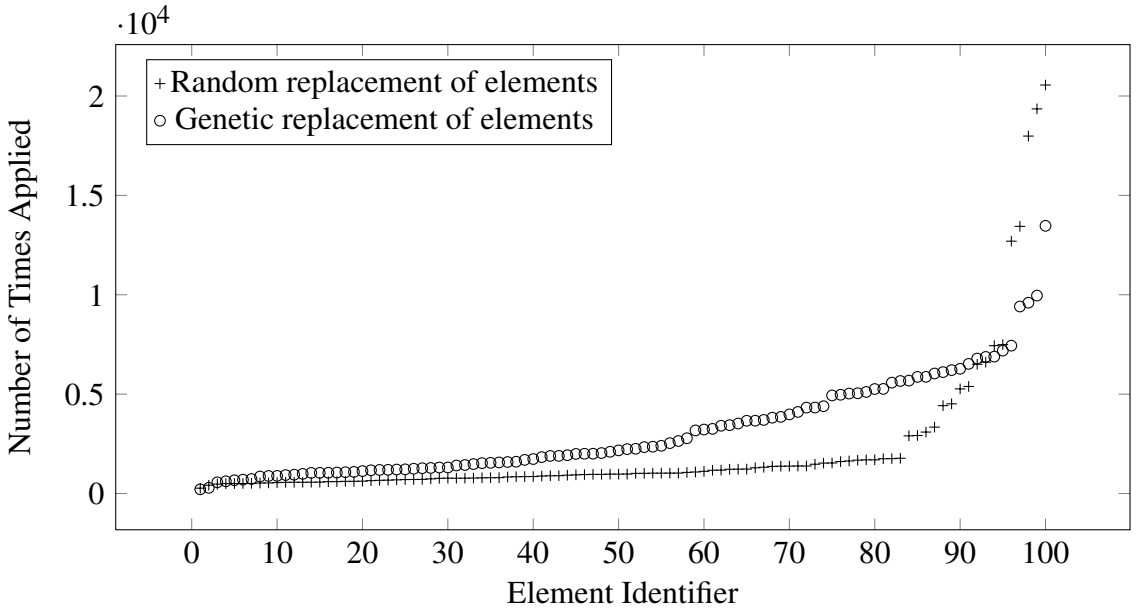


Figure 6.13: Comparing the number of times an element in the final set of 100 elements is selected.

direct product or as an internal semi direct product.

6.5.1 Fully Symmetric Group

The first group we consider is the symmetric group of degree n which contains all permutations of n symbols. In this instance the lexicographical minimum can be calculated by sorting the state vector. As sorting can be performed in polynomial time an efficient solution for the COP when $G = S_n$ is known.

Example 6. For a system with four components, sorting equivalent states [3, 2, 1, 3] and [3, 3, 2, 1] yields the state [1, 2, 3, 3], which is clearly the smallest state in the orbit.

To improve upon this approach we introduce a new strategy which exploits the property that components containing the same value after an update are in sorted order.

6.5.2 Exploiting Group Homomorphisms

For models that exhibit full symmetry between components, the smallest state in the orbit can be efficiently computed using the techniques presented in Section 6.5.1. However, many of the groups detailed in Chapter 4 and encountered in model checking are only isomorphic to the symmetric group. In this instance the efficient approach outlined above cannot be directly applied to a state.

Consider the subgroup

$$H = \{(1, 2)(4, 7)(5, 8)(6, 9), (2, 3)(7, 10)(8, 11)(9, 12)\}$$

which permutes components 1, 2 and 3, with their linked blocks of components (see Figure 6.14). It is clear that H is isomorphic to S_3, the symmetric group on 3 objects. Yet we cannot compute the minimal representative of a state by sorting s since this is equivalent to applying an element $\alpha \in S_{12}$ to s, which may not belong to the group H. Therefore, we present an approach to efficiently solve the COP for groups isomorphic to the fully symmetric group.

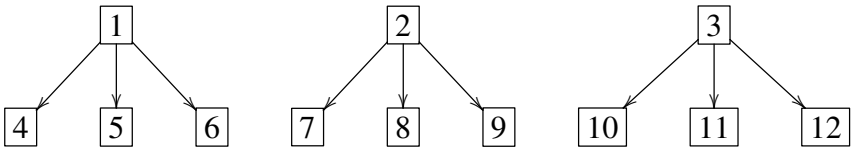


Figure 6.14: An example arrangment of component blocks.

A homomorphism between groups G_1 and G_2 has two important properties: it preserves the binary operation of G_1 and there is a one to one correspondence between the groups (see Section 3.1.1). It follows that any action group G_1 has on a state must be preserved by G_2.

Theorem 6.1. If a homomorphism can be constructed between a Group G_1 and G_2 that obeys a small number of restrictions, the lexicographical minimal state can be computed without requiring the full enumeration of elements in G_1. The set of restrictions follow:

1. Let the orbit of component x be denoted G_x, then $\forall x \in S$, $|G_x|$ is the same.
2. Let n be the number components, then there are $n/|G_x|$ distinct orbits.
3. G_2 is the group $S_{|G_x|}$.
4. Let $G_i (1 \leq i \leq n/|G_x|)$ be a distinct orbit. Exactly one component from G_i is mapped to a component in G_2.

Proof of Theorem 6.1. Let G_2 be the symmetric group S_n and G_1 a group with y distinct disjoint orbits G_i $(1 \leq i \leq y)$. By restriction 4 only one component in G_i can be mapped to a component of S_n. Therefore, all components that can be moved to the left most position in s reside in separate component of S_n. Furthermore, by restrictions 2 and 3 all components in S_n have one been mapped at most one component that can appear in the left most position.

Let G_i contain the component currently in the left most position of the state. A permutation $\alpha \in S_n$ acts on s to permute one component in G_i to reside in the first position of the state. The minimal state can be constructed by lexicographically sorting the components in G_i

and applying an element $\alpha \in S_n$ to state s that provides this ordering of components. It follows that for a group that is isomorphic to the symmetric group of size n, that meets our restrictions, $\min[s]_G$ can be calculated by sorting a set of n components and applying a single element $\alpha \in G_2$. □

It is easy to construct a mapping that meets our restrictions for a group isomorphic to S_n. However, this does not guarantee that the mapping will be a homomorphism. To test if a constructed mapping is a homomorphism we use GAP. It is important to note that testing this property may be computationally inefficient and the literature provides no indication of when poor performance may arise. In our implementation and testing, Chapter 8, the performance was acceptable for groups commonly encountered during model checking.

Example 7. Recall group H given at the start of the section. Group H meets restriction 1 and 2 as it acts on a set of 12 components and has four distinct orbits of size 3. The 4 distinct orbits are:

$$H_1 = \{1, 2, 3\}$$
$$H_2 = \{4, 7, 10\}$$
$$H_3 = \{5, 8, 11\}$$
$$H_4 = \{6, 9, 12\}.$$

It follows from restriction 3 that G_2 is the symmetric group S_3 and a mapping between the groups that obeys restriction 4 is shown in Figure 6.15. The mapping has been tested using GAP and is a homomorphism between H and S_3.

For a state s = (6, 10, 3, 4, 5, 7, 8, 11, 9, 2, 1, 12) the components that can appear in the first position of the state, indexed by orbit H_1, are [6, 10, 3]. The lexicographical ordering is [3 ,6, 10] and can be provided by the permutation (1, 2, 3) $\in S_3$. Applying this permutation to s results in s = (3, 6, 10, 2, 1, 12, 4, 5, 7, 8, 11, 9) which is verifiable as $\min[s]_{s_3}$. It is easy to check that this state can be provided by the element (1, 2, 3)(4, 7, 10)(5, 8, 11)(6, 9, 12) $\in$ H and using enumeration that $\min[s]_H$ = (3, 6, 10, 2, 1, 12, 4, 5, 7, over other approaches to probabilsitic symemtry reduction., 11, 9).

An added benefit of this technique is that when a new state is constructed, the minimum representative need only be calculated if a component indexed by an orbit containing the left most component has been updated.

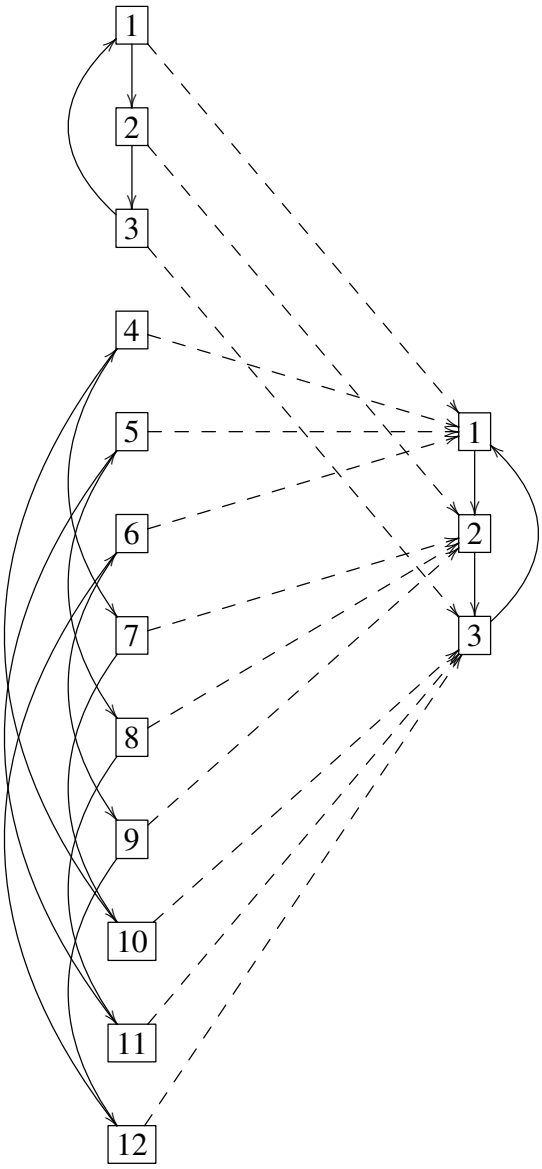


Figure 6.15: A mapping of components between groups that meets the set of restrictions. One of the actions provided by G_1 and G_2 on their set of components is provided to show the mapping is a homomorphism.

6.5.3 Cyclic Group

The elements of a cyclic group C_n act on a state by shifting all components to the left by a certain distance. It follows that for a state acted upon by a cyclic permutation group, any one of its components can be shifted to reside in the first position in the state. Furthermore, as all components are shifted when applying a permutation, it will always require the application of exactly N – 1 transpositions to calculate the new state.

As asserted, any component in the state can be shifted to occupy index 0. There is no need to apply any permutation that does not shift a component with the smallest value to index 0.

To identify the smallest component a simple algorithm is used to traverse the state and store the set of indexes containing the smallest valued components, FigureZ6.16.

```
1.Procedure minElements(State original)
2.{
3.    int min = original[0]
4.    LinkedList result = null
5.    for (int i = 0; i < original.length; i++)
6.    {
7.        if (original[i] = min)
8.        add(result, i)
9.    }   elsif (original[i] < min)
10.       result = null
11.       add(result, i)
12.   }
13.   return result
14.}
```

Figure 6.16: An algorithm to compute the lexicographical minimal state when considering a cylic group.

Once a set of indexes containing minimal components have been identified, a permutation in C_n that shifts this component to index 0 can be applied. The resulting states are then compared to determine which is the lexicographical minimum. In the worst case this could still result in all elements in the cyclic group being applied to the state.

Example 8. For the state vector [1, 1, 1, 1] acted upon by the group C_4 the algorithm will return the set of all indices.

To prevent the need for the application of multiple elements from the cyclic group we add some further restrictions. In other words, when multiple components in the state contain the same value, we have to select a single one to be shifted to reside in position 0. This can be achieved in part using the following restriction.

1. If consecutive components in the state vector contain the same value, then the component at the start of the sequence will be selected as the minimal component.

   Example 9. For the vector [2, 1, 1, 2] index 1 would contain the minimal component to be shifted to position 0.

   (a) For the purpose of this calculation the vector is treated as a cycle.

   Example 10. For the vector [1, 1, 2, 1] index 3 would contain the minimal component required to be shifted to position 0.

(b) If all components in the state have the same value and the size of the cycle is the same as the length of the state then the current state is the minimal representative.

However, even with this restriction the algorithm may still return multiple indexes. This occurs if the state contains more than one sequence of minimal elements. Further restrictions are therefore required.

Example 11. For the state [1, 1, 2, 1, 1, 3, 1, 1, 2] index 0, 3 and 6 would contain the minimal components shifted to position 0 and compared.

1. If multiple minimal sequences are identified then return the index of the sequence with the longest length.

   Example 12. For the state [1, 2, 1, 1, 3, 1, 1, 2] index 3 and 6 would contain the minimal element that would be shifted to position 0 and compared. This is an improvement as index 0 is no longer checked.

2. When multiple indices are returned, it indicates that there are identical sequences of identical length in the state. Let x_i be the index of the component residing at the end of a minimal sequence, then the components indexed by x_{i+1} are compared and the index with the smallest component is returned. In the event that these two components have the same value, the next two indices are continually compared until a minimal component is found.

   Example 13. For the vector [1, 2, 1, 1, 3, 1, 1, 2] index 5 would contain the minimal components that would be shifted to position 0.

   Example 14. For the vector [1, 1, 3, 4, 1, 1, 3, 5] index 0 would contain the minimal component that would be shifted to position 0.

3. The only remaining case to consider is when the sequence of compared objects are identical and the current components to be compared are in fact the first component of the other sequence. In this instance more than one sequence is still viable and it is irrelevant which index is returned.

   Example 15. For the vector [1,1, 3, 4, 5 1, 1, 3, 4, 5] index 0 would contain the minimal element that would be shifted to position 0.

By obeying these restrictions the application of only one permutation is required to map a state to is minimum representative. However, the need for the application of this permutation can be removed by a simple modification to the hashing function. This is covered in Chapter 8 where the implementation of the algorithms are discussed.

6.5.4 Direct Product

It would be useful to start with a group G and break it down into a product of smaller groups. When a group can be factored in a specific manner a representative state could be calculated by applying elements from the smaller groups instead of the single larger group.

Definition. If H and K are groups, then their direct product, denoted by $H \times K$, is the group with elements all orders pairs (h, k) where $h \in H$ and $k \in K$, and with operation

$$(h, k)(h', k') = (hh', kk').$$

The above definition shows how to multiply two groups together to form a single larger group. However, we are interested in starting with a direct product and factoring it into a set of smaller groups.

Theorem 6.2. Let G be a group with normal subgroups H and K. If $HK = G$ and $H \cap K = 1$, then $G \cong H \times K$

If a group can be factored in accordance to Theorem 6.2 then it is hypothesised if $G = H_1 \times H_2 \times \cdots \times H_k$ then the $\min[s]_G = \min[\ldots \min[\min[s]_{H_1}]_{H_2} \ldots]_{H_k}$

Example 16. It is easy to show that the group K_3 decomposes as the direct product $K_3 = H_1 \times H_2 \times H_3$ where:

- $K_3 = \{(1, 2)(3, 4)(5, 6)(7, 8), (1, 3)(2, 4)(5, 7)(6, 8), (1, 5)(2, 6)(3, 7)(4, 8)\}$
- $H_1 = \{((1, 2)(3, 4)(5, 6)(7, 8)\}$
- $H_2 = \{((1, 3)(2, 4)(5, 7)(6, 8)\}$
- $H_3 = \{((1, 5)(2, 6)(3, 7)(4, 8)\}$.

Let s = (4, 3, 2, 4, 4, 5, 5, 1) and using enumeration we calculate the lexicographical minimal state to be $\min[s]_{K_3} = (1, 5, 5, 4, 4, 2, 3, 4)$. However, by independently considering the subgroups $\min[\min[\min[s]_{H_i}]_{H_j}]_{H_k}] = (3, 4, 4, 2, 5, 4, 1, 5)$ for any distinct $i, j, k \in \{1, 2, 3\}$.

This example illustrates that state s cannot be minimised by independently considering the subgroups H_1, H_2 and H_3, no matter the order of application. To proceed we examine how the subgroups act on a state and attempt to identify an alternative representative that can be calculated by the independent application of the subgroups. To achieve this requires the introduction of further theorems and results from group theory.

Theorem 6.3. If $N \lhd G$, then the cosets of N in G form a group, denoted by G/N of order [G : N]

As the internal direct product contains two normal subgroups, each can be used to form a smaller group often referred to as a factor group. These factor groups can easily be constructed using Corollary 1.

Corollary 1. If $N \lhd G$, then the function $v : G \to G/N$ is a surjective homomorphism with kernel N.

Therefore, given a group $G \times H$ two subgroups of smaller order can be constructed: $(G \times H)/G$ and $(G \times H)/H$. To understand the structure of these groups and how they act on a state we introduce two of the fundamental isomorphism theorems.

Theorem 6.4 (First Isomorphism Theorem)**.** Let $f : G \to H$ be a homomorphism with kernel K. Then K is a normal subgroup of G and $G/K \cong$ imf.

The first isomorphism theorem shows there is no significant difference between a factor group and a homomorphic image

Theorem 6.5 (Second Isomorphism Theorem)**.** Let N and T be subgroups of G with N normal. Then $N \cup T$ is normal in T and $T/(N \cup T) \cong NT/N$

The direct product contains two normal factor group and consequently;

Theorem 6.6. If $A \lhd H$ and $B \lhd K$, then $A \times B \lhd H \times K$ and $H \times K/A \times B \cong (H/A) \times (K/B)$

It immediately follows if, $N \lhd H$, then $N \times 1 \lhd H \times K$ and

Corollary 2. If $G = H \times K$, then $G/H \times 1 \cong G$.

Corollary 2 asserts, given the group $G \times H$, the groups $(G \times H)/G \times 1 \cong H$ and $(G \times H)/H \times 1 \cong G$ can be constructed. By the first isomorphism theorem these factor groups define a homomorphism that collapses cosets with several elements into a single element. This collapsing of elements partitions the set of state components into a set of mutual disjoint component subsets. Where the components collapsed into each subset are those indexed by cosets of the factor group. We refer to these disjoint subgroups as part of a partition.

Hence, $(G \times H)/G \times 1 \cong H$ permutes a partitions of components in state s defined by the cosets of G and $(G \times H)/H \times 1 \cong G$ permutes a partitions of components in state s defined by the cosets of G. Once the groups isomorphic to G or H sets an ordering of parts, no action provided by the other group can alter this ordering.

An Alternative Representative

Therefore, each subgroup partitions components in a state into a set of disjoint parts. Each part consists of the set of components indexed by a coset that was collapsed into a single element. A representative state could be calculated for every state in $s_{[G\times H]}$, if subgroups isomorphic to G and H could independently return the same ordering of parts. A simple and effective way to lexicographically order parts is based on the sum of their components.

Here we note that components indexed by a part do not necessarily appear in consecutive order in a state. As a result we define an ordering of parts based on the location of their left most component. To illustrate, let a and b be parts, if a is ordered before part b then the left most component indexed by part a appears before any component indexed by part b.

Example 17. Recall group K_3, that decomposes as the direct product $K_3 = H_1 \times H_2 \times H_3$, and state s = (4, 3, 2, 4, 4, 5, 5, 1).

Group H_1 permutes two parts in state s. In Figure 6.17 the components in s indexed by the part are highlighted by squares and components indexed by the second part by circles.

([4] (3) [2] (4) [4] (5) [5] (1))

Figure 6.17: The partition of the components under the group H_1

The sum of components in the "square" part is 15 while the sum of components in the "circle" coset is 13. Using enumeration the lexicographical minimal part ordering for group H_1 is provided by state s = (3, 4, 4, 2, 5, 4, 1, 5).

Group H_2 permutes two part in state s. In Figure 6.18 the components in s indexed by the first part are highlighted by squares and components indexed by the second part by circles.

([3] [4] (4) (2) [5] [4] (1) (5))

Figure 6.18: The partition of the components under the group H_2

The sum of components in the "square" part is 16 while the sum of components in the "circle" part is 12. Using enumeration the lexicographical minimal coset ordering for group H_2 is provided by state s = (4, 2, 3, 4, 1, 5, 5, 4).

The group H_3 permutes two cosets in state s. In Figure 6.19 the components in state s indexed by one coset are highlighted by squares and the the components indexed by the

other in circles.

([4] [2] [3] [4] ① ⑤ ⑤ ④)

Figure 6.19: The partition of the components under the group H_1

The sum of components in the "square" coset is 13 while the sum of components in the "circle" coset is 15. Using enumeration the lexicographical minimal coset ordering for group H_3 is provided by state s = (1, 5, 5, 4, 4, 2, 3, 4).

Let $\text{cmin}[s]_G$ denote the lexicographical minimal coset ordered state in the orbit of state s under group G. The example shows $\text{cmin}[\text{cmin}[\text{cmin}[s]_{H_3}]_{H_2}]_{H_1}]$ = (1, 5, 5, 4, 4, 2, 3, 4) and as required $\text{cmin}[\text{cmin}[\text{cmin}[s]_{H_i}]_{H_j}]_{H_k}]$ = (1, 5, 5, 4, 4, 2, 3, 4) for any distinct i, j, k $\in \{1, 2, 3\}$.

6.5.5 Semi Direct Product

In Section 6.5.4, we saw if G is the internal direct product of H and K, then a state representative could be efficiently computed. We now show a modification to calculate a representative when the restriction of both subgroups being normal is lifted.

Definition. Let K be a (not necessarily normal) subgroup of G. Then a subgroup $Q \trianglelefteq G$ is a compliment of K in G if $K \cap Q = 1$ and $KG = G$

Definition. (Semi Direct Product) A group G is a semi direct product of K by Q, denoted by $G = K \times Q$, if $K \trianglelefteq G$ and K has a compliment $Q_1 \cong Q$.

In what follows we denote elements of K by letters a, b, c in the first half of the alphabet, and we denote elements of Q by letters x,y,z at the end of the alphabet. For elements $x, y \in G$, $x = a_1z_1$ and $y = a_2z_2$ and it follows that $xy = a_1z_1a_2z_2$. If we were dealing with a direct product, were elements commute, xy could be simply written as $xy = a_1a_2z_1z_2$. This was one of the major reasons a representative can be computed from subgroups. An element xy can always be reconstructed by applying elements of group K followed by Q. For our current group if we tried to make the same swap of elements we get

$$xy = a_1z_1a_2z_1^{-1}a_2^{-1}a_2z_1z_2.$$

It is clear that $z_1z_2 \in Q$ and in hope of applying a similar technique we must justify that $a_1z_1a_2z_1^{-1}a_2^{-1}a_2 \in K$. By cancelling terms

$$a_1 z_1 a_2 z_1^{-1} a_2^{-1} a_2 = a_1 z_1 a_2 z_1^{-1}$$

and since $a_1 \in K$, $a_1 z_1 a_2 z_1^{-1} \in K$ when $z_1 a_2 z_1^{-1} \in K$. Therefore, each element k of K gives rise to an automorphism of H via conjugation:

$$\mu(z) = zaz^{-1}$$

So, the interaction of H and K is expressed by the homomorphism μ. If there is no interaction, the product is direct and the homomorphism is trivial. If, on the other hand, there is interaction, the product is not direct, one of the two groups is not normal, they do not commute with each other, and the homomorphism we have defined tells us exactly how the group structure deviates from that of a direct product. With $x = a_1 z_1$ and $y = a_2 z_2$, as before we have

$$xy = a_1 \mu_{z_1}(a_2) z_1 z_2$$

To formalise;

Lemma 6. If G is a semi direct product of K by Q, then there is a homomorphism $\theta : Q \to$ Aut(K), defined by $\theta_x = \gamma_x |K$; that is, for all $x \in Q$ and $a \in K$,

$$\theta_1(a) = a \qquad \text{and} \qquad \theta_x(\theta_y(a)) = \theta_{xy}(a)$$

Definition. Given a group Q and K and a homomorphism $\theta : Q \to$ Aut(K), define G = KQ to be the set of all orders pairs $(a, x) \in X \times Q$ equipped with the operation

$$(a, x)(b, y) = (a\theta_x(b), xy)$$

Theorem 6.7. Given groups Q and K and a homomorphism $\theta : Q \to$ Aut(K), then G = KQ is a semi direct product of K by Q that realises θ

Intuitively, realising θ is a way of describing how K is normal in G. Therefore, given a groups isomorphic to G and K and a homomorphism $\theta : Q \to$ Aut(K) a state can be representative can be efficiently computed. In the worst case the representative calculation applies all elements of G, all elements of K and a single mapping to the state.

6.5.6 Summary

In this chapter we present a selection of new techniques to efficiently compute equivalence class representatives for certain classes of symmetry groups. We present enhancements that improve the average runtime of exhaustive search and where enumeration is infeasible, we consider a tailored local search algorithm. For symmetry groups possessing identifiable structural properties we provide efficient techniques that do not require the exhaustive application of all elements in the symmetry group . We suggest techniques for the fully symmetric group, cyclic groups and groups that can be decomposed as an internal direct product or as an internal semi direct product.

CHAPTER 7

Constructing the Probabilistic Model

In Chapter 4 we introduced the PSS language and with the presented semantics formally defined how a specification $\mathcal{P}$ can be used to construct a DTMC or MDP. This was followed by the introduction of the ECD where we established a correspondence between automorphisms of an ECD generated from $\mathcal{P}$ and automorphisms of the probabilistic model constructible from $\mathcal{P}$. In addition, we proved that if the identified automorphisms met a small set of restrictions they were valid for symmetry reduction. In this Chapter we consider how to use $\mathcal{P}$ and G to construct the probabilistic model directly from $\mathcal{P}$.

As noted in Chapter 6, the construction of a quotient structure requires the use of a representative function. For the discussion presented within this chapter we assume for any state $s \in S$ the representative function rep(s, G) always returns the same state in $[s]_G$. For convenience we denote this unique representative state as $\min[s]_G$.

Group G contains a set of permutations $\pi : S \to S$ which act on the state space while preserving the transition function (see Section 3.2). To meet the definition of a quotient DTMC (see Section 3.2.2) we require that $\mathbf{P}(\pi(s), \pi(t) = \mathbf{P}(s, t)$ for all $s, t \in S$. In the case of MDPs (see Section 3.2.3), for each $s \in S$ and each distribution $\mu \in \text{Steps}(s)$, there must be a distribution $\mu' \in \text{Steps}(\pi(s))$ such that $\mu'(\pi(s')) = \mu(s')$ for all $s' \in S$. Therefore, the construction of a quotient DTMC and MDP can then be carried out in the following manner:

- For a DTMC (S, **P**) we build (S_G, $\mathbf{P}_G$) and for each ($\min[s]_G$, $\min[t]_G$) $\in S_G$

$$\mathbf{P}_G(\min[s]_G, \min[t]_G) = \sum_{t \in S | rep(t,G) = \min[t]_G} \mathbf{P}(\min[s]_G, t) \tag{7.1}$$

- For an MDP (S, Steps), the quotient model is (S_G, $Steps_G$). To meet this requirement if $\min[s]_G \in S_G$, then $Steps_G(\min[s]_G)$ contains a distribution μ_G if and only if there exists $\mu \in Steps(\min[s]_G)$ such that for each $\min[t]_G \in S_G$

$$\mu_G(\min[t]_G) \sum_{t \in S | rep(t,G) = \min[t]_G} \mu(t) \tag{7.2}$$

Constructing a probabilistic model in this manner results in a quotient model that is equivalent to the original in the context of strong probabilistic bi simulation. In Section 7.1 we detail an algorithm to construct the quotient probabilistic model directly from a PSS specification. The algorithm makes use of two sets to traverse the state space and in Section 7.1.1 we consider several common implementations of the sets and assess their compatibility with symmetry reduction techniques.

Finally, a PCTL formula made from a set of atomic propositions AP preserved by G performs equivalently on the quotient DTMC and MDP. However, a user may desire to use a PCTL formula where the set of atomic propositions AP are not preserved by G. In Section 7.2 we address this issue by presenting a method to construct the smallest quotient model on which a PCTL formula will perform equivalently.

7.1 Algorithm to Construct a Quotient Probabilistic Model

In this section we outline an algorithm to construct the quotient probabilistic model directly from a PSS specification $\mathcal{P}$. The algorithm is obtained by modifying graph traversal algorithms commonly employed in model checking to explore the state space. Two of the most common traversal algorithms are depth first search (DFS) and breadth first search (BFS) both of which are based on the skeleton algorithm given in Figure 7.1.

As is common in the field of model checking the set R is implemented as a hash table and U is either a stack or queue. If U is a stack then the algorithm performs a DFS traversal of the state space and conversely, if U is a queue the algorithm performs a BFS traversal. Intuitively the set R is the set of states that have been visited, while U keeps track of all states that still have to be explored. Irrespective of the implementation of the set U, the algorithm determines all states reachable from the starting state s_0.

When a state s is removed from U, it is processed and all sub states in a single probabilistic

```
1.  List T := null
2.  R := U := {rep(s0)}
3.  While U ≠ ∅
4.  {
5.      remove a state s from U
6.      for each transition s → s'
7.      {
8.          if s' = error
9.          {
10.             stop and report error
11.         }
12.         if rep(s', G) ∉ R
13.         {
14.             add rep(s', G) to R and U
15.         }
16.         add rep(s', G) to T
17.     }
18. storeTransitions(T)
19. }
```

Figure 7.1: Graph Reachability Analysis.

distribution are returned. If the algorithm is exploring a DTMC there is only one distribution and consequently s is processed a single time. On the other hand, if the algorithm is exploring an MDP, s may define multiple distributions. In this instance s is processed once for every distribution it defines.

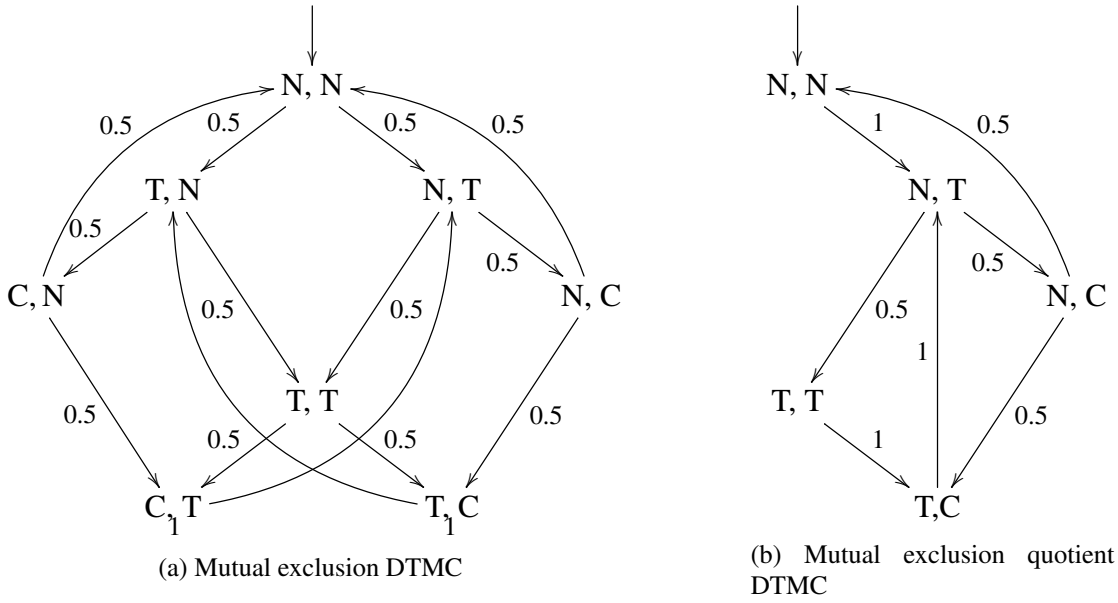
The function rep(t, G) is applied in turn to all sub states and the result, $\min[t]_G$, is stored in a list T. In addition, if $\min[t]_G$ is not in R then it is added to U to be explored in the future. Once a representative state has been calculated for all sub states in a distribution, the function storeTransition(List) is called with argument T, which has type List.

The storeTransition(List) removes a single state t from T and determines how many times a state identical to t occurs in T. The matching occurrences are removed from T and the number of matches are used to determine $\mathbf{P}(s, t)$, which is the calculation of (7.1) as required. The calculation is repeated until T is empty. Furthermore, storeTransition(List) will be called once for all probabilistic distributions defined by s, allowing for the calculation of (7.2).

Finally, the algorithm is guaranteed to terminate as each edge is taken only once. On termination, R contains all vertices that are reachable from s_0 in the quotient graph.

Example 18. Figure 7.2a illustrates the DTMC constructible from the simple mutual exclusion PSS specification given in Figure 4.2. The automorphism group $G = \{(1, 2)\}$ was calculated from the ECD associated with this specification (see Section 5.1.4) and it is easy

to check that G is in fact the largest group valid for symmetry reduction. Using this information the algorithm given in Figure 7.1 directly constructs the quotient DTMC presented in Figure 7.2b.



(a) Mutual exclusion DTMC

(b) Mutual exclusion quotient DTMC

Figure 7.2

7.1.1 Implementation of Data Structures

As previously mentioned in the field of model checking the set R is implemented as a hash table and U is either a stack or queue. Here we consider several common implementations of the sets and assess their compatibility with symmetry reduction techniques.

Stack and Queue

In probabilistic model checking before any property can be checked, the full, or in our case the quotient state space must be explored. When directly comparing the two traversal algorithms, in terms of explicit state model checking, the DFS is more space efficient. On average a DFS traversal requires less states to be held concurrently in memory.

In probabilistic model checking no benefit is gained by using a BFS to explore the state space. The ability of a BFS to return the shortest path that violates a property does not translate to the probabilistic domain. Therefore, we always choose to implement the set as a stack.

The most common optimisation is to only store the state at the top of the stack in full. Every

other state in the stack can be represented as a list detailing how it differs from the state stored directly above it. This technique is called a ”state delta” and is routinely used in explicit state model checkers. For the implementation of our algorithm in Chapter 8, we will implement the set as a stack that uses state deltas to reduce its footprint in memory.

Hash Function

Hashing is a storage technique where data records are stored in a distributed manner over a fixed address range. To retrieve a data record, a hashing function is used to compute the address where the record may be located in memory. To achieve this behaviour a hash function h accepts a parameter called a key k. The value h(k) is called a hash address and indicates where a data record associated with key k may be located. A hash function h can be defined

$$h : K \rightarrow \{0, \ldots, n\}$$

which is not injective. Subsequently two distinct keys k and k' may have the same hash value, $h(k) = h(k')$.

Assuming a hash table contains a key k and a new key k' is inserted where $h(k) = h(k')$, the result is a hash collision and the key k' must be stored in an alternative location as $h(k')$ is occupied. A simple solution is to chain collided keys. This can be achieved by making each element in the hash table the first node of a linked list containing collided keys. This is not the only way to handle the key collision problem but in the context of model checking it is an appropriate solution.

When using a chaining scheme, each record in a hash table requires r bytes to store and p bytes to reference with a pointer. Therefore, the memory requirement for this scheme where s is the number of states in the reachable state space and n is the size of the hash table is given by the equation

$$\text{Required Memory} = (s \times p) + ((r + p) \times n)$$

Assuming a memory consumption of 1 KByte per state and a 64 bit addressing scheme, a model of 1 million states would require approximately 1016MByte of storage space. Due to these large memory requirements explicit state model checkers tend to provide this hashing scheme as a user selected option and not as the default method. For our needs this simplistic

scheme is not appropriate. In Chapter 8 we aim to analyse the viability of symmetry reduction in a probabilistic setting which requires models comprising of significantly more than one million states. Therefore, a more space efficient hashing scheme must be considered.

A common scheme that provides a more space efficient approach is known as Bitstate hashing. The key idea behind bit state hashing is to store a single bit that indicates if a state s has been visited. Upon encountering a state s, if the value of the bit indexed by h(s) equals;

- 1, then state s may have been encountered.
- 0, then state s has not been encountered.

As the hash table no longer stores complete states, collisions cannot be resolved. Consequently some states will be incorrectly dropped and in addition, their successor states will not be explored if they are only reachable through the dropped state. To adequately guarantee that a state space has been fully explored the probability of a collision occurring must be small. Let n be the number of slots in the hash table, and as $n/s \rightarrow 0$ the number of conflicts expected to occur tends to 0.

Therefore, to provide low collision rates, the hash table should be as large as possible even though the majority of the slots will hold the value 0. This results in a waste of memory. Analysis has shown a hash table occupying 500 MByte of RAM would only allow the exploration of approximately 29,032 states with a 0.9 probability that a collision has not occurred [66]. While this scheme lifts the requirement of state storage, the memory required to minimise conflicts to an acceptable level is infeasible.

A hashing scheme that attempts to alleviate this downside is Multiple Bit hashing, which considers the application of k independent single bit hash functions. As before, a single bit hash function calculates a position in the hash table that contains either a 0 or 1. However, for a single function, h(s) = 1 does not indicate that state s may have been visited. In multi bit hashing a state s may already have been encountered, if and only if for all k hash functions

$$H[h_1(s)] = 1 \wedge H[h_2(s)] = 1 \wedge \cdots \wedge H[h_k(s)] = 1.$$

As k hashing functions are applied to every state, the volume of wasted memory is lowered and for an appropriate value of k the size of the hash table can be reduced. However, if k is too large a value the hash table will quickly saturate, increasing the probability of collisions. Conversely, if the value of k is too small, the hash table must be large in order to achieve a low conflict probability.

Previous model checkers have used 2 and 20 independent hashing functions. However, analysis has shown for a modern computer architecture with a 64bit addressing scheme, the use

of 30 hashing functions would be more appropriate. Under the suggested approach, a hash table occupying 500 MByte of RAM would allow the exploration of 52.168 million states, with a 0.9999 probability that a collision has not occurred [66].

This memory improvement is achieved at the cost of a substantially increased runtime. This additional runtime results from the computation, comparison and toggling of multiple bit positions per state. Symmetry reduction is already a CPU intensive operation and adding the requirement of computing 30 hash functions for each state may lead to an unacceptable runtime. Even though this common hashing scheme has been implemented in several model checkers, it may not be compatible with a model checker which provides symmetry reduction.

Therefore, we examine the use of a hash compaction scheme that aims to simulate multi bit hashing. In a hash compaction scheme each state s is mapped to a string of b bits by applying a hash function h . The compressed state description of b bits is then hashed using a separate function h' and stored in the resultant location. Using this scheme two states s_i and s_j can produce a hash collision if their compressed state descriptions are equal i.e. $h(s_i) = h(s_j)$. This will result in the identical descriptions being hashed to the same location in the table. In cases where states collide on slots $h'(h(s_i) = h'h((s_j))$, but $h(s_i) \neq h(s_j)$ a linked list of chained state descriptions are used to resolve the conflict.

It is intuitively clear that this hash compaction scheme has a lower time complexity compared to the 30 multiple bit scheme as only 2 hash functions are employed. On the other hand, b bytes of memory are now required for every slot in the hash table. Fortunately, analysis of this hashing scheme has shown that a hash table occupying 1024 MByte of RAM, with a state descriptor of 9 bytes, would allow the exploration of approximately 97 million states with a 0.9999 probability that a collision has not occurred.

We believe hash compaction is an appropriate scheme to be paired with a model checker that provides symmetry reduction capabilities. While more advanced schemes could be considered, hash compaction will allow models in excess of 100 million states to be constructed while not negatively impacting the runtime of the reduction techniques detailed in Chapter 6. This enables us to draw informed conclusions about the viability of symmetry reduction in the probabilistic domain.

7.2 PCTL Model Checking

Once a quotient probabilistic model $\mathcal{M}_G$ has been constructed using the algorithm given in Figure 7.1 and if a PCTL formula ϕ is invariant with respect to G then $\mathcal{M}_G$ and ϕ can be used

for probabilistic model checking. The result will be identical to that obtained by applying probabilistic model checking to $\mathcal{M}$ and ϕ (see Section 3.2).

In Section 2.3.4 the syntax and semantics of PCTL were covered and all expressible properties were assertions that returned "yes" or "no" answers. An extension to PCTL, implemented in the PRISM model checker [69], allows properties to take the form $P_{=?}[\psi]$ that returns the probability that some behaviour of a model is observed. We introduce this extensions here as it adds clarity to the proceeding examples.

Consider the DTMC constructed from the two process mutual exclusion specification, Figure 7.2a. A simple PCTL property suitable for probabilistic model checking is P =?[$G^{\leq 3}$ x = 3] which determines the probability that within 3 time steps process 1 will have been in a critical state. Using probabilistic model checking the property is satisfied with 0.375 probability. A second PCTL formula P =?[$G^{\leq 3}$ x = 3 & y = 3] determines the probability that within 3 time steps process 1 or 2 will have been in a critical state. Probabilistic model checking gives a 0.75 probability that the property will be satisfied.

Now consider the quotient DTMC constructed from the same specification, Figure 7.2b. The property P =?[$G^{\leq 3}$ x = 3] is not satisfied by any path in the quotient probabilistic model. On the other hand, the property P =?[$G^{\leq 3}$ x = 3 & y = 3] is satisfied with 0.75 probability, the same as in the unreduced probabilistic model. The key difference between the two properties is that property one is not invariant under the group G used to construct the quotient probabilistic model.

7.2.1 Determining if a PCTL Formula is Invariant Under a Group G

As we are concerned with providing an automated approach to symmetry reduction, the user cannot be expected to provide a PCTL formula compatible with the quotient model. To check if a formula is appropriate we introduce the notion of equivalence between PCTL formulas. Two PCTL formula ϕ_1 and ϕ_2 are equivalent, denoted $\phi_1 \equiv \phi_2$, if they are the same up to rearrangement of atoms in a maximal propositional sub formula. This requires we check that for all $\alpha \in$ G, $\alpha(\phi) \equiv \phi$. If this is true the formula ϕ can be posed to a quotient structure created using the group G.

Action of G on ϕ

Group G is Aut(ECD($\mathcal{P}$)) and in Section 5.2.2 the action that an element $\alpha \in$ Aut(ECD($\mathcal{P}$)) has on the set of atomic propositions in a state was defined. As a PCTL formula is a set

of state and path formula over a set of atoms (see Section 2.5.3), the formula $\alpha(\phi)$ can be constructed from ϕ by application of the rules given in Section 5.2.2.

7.2.2 Largest Valid Subgroup

If a PCTL formula ϕ is not appropriate for model checking using the quotient probabilistic model, it may be invariant under a group H that is a subset of G. In other words $(\alpha(\phi) \equiv \phi)$ for a set of elements $\alpha \in \mathrm{H} \subset \mathrm{G}$ and the group H can be used to construct a quotient structure in which ϕ holds.

While H is valid for reduction it may not be the largest group valid in both the specification and ϕ. Starting with the valid subgroup H the algorithm given in Figure 7.3 computes the largest valid subgroup. The algorithm was developed to calculate the largest valid symmetry in a Promela-lite specification and was implemented in the SymmExtactor tool [38]. Our only modification appears on line 7 where we check the validity of the element against a PSS specification and the given PCTL formula.

```
1.  X := generators of Aut(ECD(𝒫))
2.  H := ⟨{α ∈ X : α(P) ≡ P}⟩
3.  U := representatives of right cosets of H in Aut(ECD(𝒫)) except H
4.  while U ≠ ∅
5.  {
6.      U := U\{α}
7.      if (α(𝒫) ≡ 𝒫 && α(φ) ≡ φ)
8.      {
9           H := ⟨H⋃{α}
10.         if |Aut(ECD(𝒫))| / |H| < |U|
11.         {
12.             U := representatives of right cosets of H in Aut(ECD(𝒫)) except H
13.         }
14.     }
15. }
```

Figure 7.3: Algorithm to calculate the largest valid subgroup.

The algorithm starts with a known valid subgroup H, and adds valid coset representatives, computed using GAP, to the generators of H to obtain successively larger valid subgroup. Once all coset representatives of the the group H have been checked the largest valid set will have been determined.

It will often be the case that the user wishes to run multiple PCTL formulas in succession

without reconstructing a probabilistic model. If this is the case the largest group that is invariant under all formula and the specification must be calculated and used to construct a quotient chain. This can be achieved using the Algorithm in Figure 7.3 to calculate the largest valid group obtained from the specification against the first PCTL formula. The resultant group can then be used to find the subgroup with respect to the next formula and so on until the algorithm has been run against all formulas.

7.3 Summary

In this chapter we have combined our presented techniques to construct a smaller quotient model directly from a probabilistic specification. In addition, we have ensured that the reduced model will be compatible with a set of PCTL formula when probabilistic model checking is applied.

CHAPTER 8

Results and Comparison

In this chapter we describe the implementation of the Probabilistic Symmetric Systems symmetry reduction tool which we have developed. The PSS tool accepts two inputs: a PSS specification (introduced in Chapter 4) and a set of PCTL properties. Using the automated symmetry detection techniques detailed in Chapter 5, the strategies presented in Chapter 6 and the exploration algorithm of Chapter 7, the quotient probabilistic model is constructed.

An overview of the PSS tool implementation is provided in Section 8.1 and we discuss how it integrates with GAP [51], bliss [62] and PRISM [69] to provide an automated approach to probabilistic symmetry reduction. Section 8.2 focuses on the efficient implementation of automated symmetry detection using the extended channel diagram techniques of Chapter 5. In Section 8.3 we detail the implementation of the techniques described in Chapter 6 and where appropriate provide experimental results to highlight their effectiveness. Finally, we present experimental results which demonstrate how our techniques perform on a varied selection of PSS specifications and compare them to other probabilistic symmetry reduction techniques. All specification and properties used within this chapter, as well as source code for a preliminary versions of our tool, can be found at [88].

8.1 An Overview of the PSS Symmetry Reduction Tool

The PSS symmetry reduction tool is implemented primarily in the JAVA programming language and interfaces with a selection of tools written in C++. An overview of the PSS symmetry reduction tools architecture and how it integrates with various other tools is illustrated in Figure 8.1.

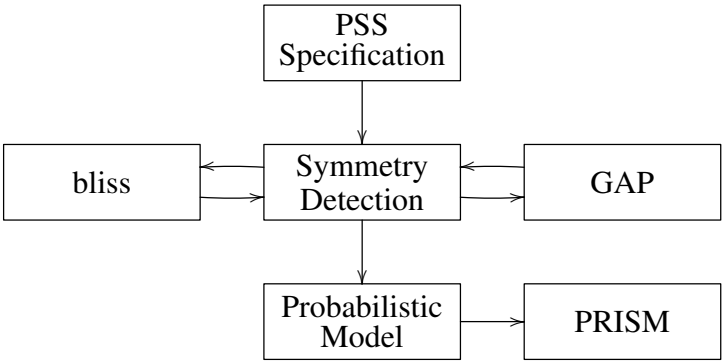


Figure 8.1: An overview of the PSS symmetry reduction tool's architecture.

When a PSS specification $\mathcal{P}$ is passed as a command line argument, the specifications syntax is checked for conformity against the PSS BNF given in Section 4.2.2 and its variables type checked. The tokenizer, parser and abstract syntax tree representation of the input specification required to check these restrictions were created with the aid of ANTLR [84]. These are the only restrictions checked at compile time. If a PSS specification contains a command with no associated rule (see Sections 4.2.5 – 4.2.7), the error will not be revealed until the command is executed at runtime.

For a specification $\mathcal{P}$ that passes the compile time checks, the PSS tool extracts the extended channel diagram ECD($\mathcal{P}$) and calculates the largest valid subgroup G $\leq$ Aut(ECD($\mathcal{P}$)) with respect to $\mathcal{P}$. The set of PCTL formula given as input is checked for invariance under G and formulas that did not pass the test are used in conjunction with G to compute a new group H $\subset$ G that all PCTL formula are invariant under. The resultant group H is the largest group valid for symmetry reduction with respect to $\mathcal{P}$ and the set of PCTL formula.

The structural properties of group H are analysed using GAP [51], which is a system for computational discrete algebra, with particular emphasis on Computational Group Theory. The results of the analysis determine which representative function will be selected. Once the implementation of the rep(s, H) function has been finalised, the quotient state space is constructed. Additionally, during exploration a flat file description of the probabilistic model is generated. This flat file description and the set of PCTL formula are accepted as input by the PRISM model checker [69] and using its explicit-state probabilistic model checking library, implemented using sparse matrix data structures, probabilistic model checking is

performed.

To allow the effectiveness of our techniques to be determined, the PSS symmetry reduction tool can be forced to construct the unreduced probabilistic model associated with specification $\mathcal{P}$. In fact, any group G can be provided as input for symmetry reduction bypassing the automated detection process. This may be desirable when the group G is known from a previous calculation or an expert user has manually identified a larger symmetry that our tool did not detect. Finally, the function rep(s, G) can be manually selected to facilitate a direct comparison of techniques.

8.2 An Overview of Automated Symmetry Detection

Our automated symmetry detection technique requires the abstract syntax tree representation of the specification and the types of all channels, variables and processes in the specification. The extended channel diagram ECD($\mathcal{P}$) of specification $\mathcal{P}$ is extracted using the algorithm given in Figure 5.11 and its automorphisms are calculated using bliss [62], an open source tool for computing automorphism groups and canonical forms of graphs. If any of the automorphisms are not valid with respect to $\mathcal{P}$, the algorithm given in Section 5.3.2 re-colours ECD($\mathcal{P}$) and bliss[62] computes the automorphisms of the updated graph. The result, Aut(ECD($\mathcal{P}$)) is the largest group valid for symmetry reduction with respect to specification $\mathcal{P}$.

This automatic symmetry detection process is summarised in Figure 8.2 and in the remainder of the section we discuss various aspects of the technique's implementation.

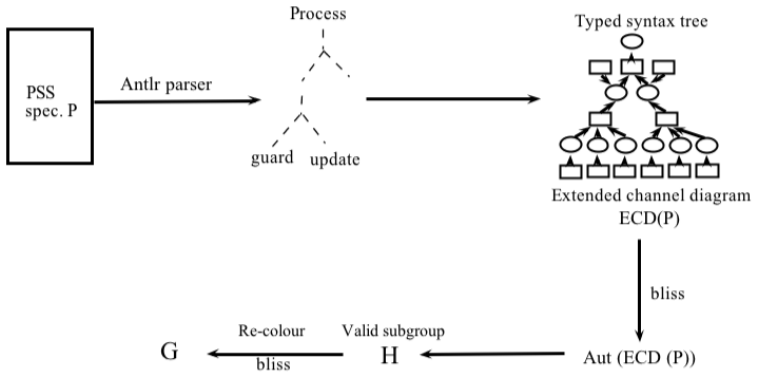


Figure 8.2: The automated symmetry detection process.

8.2.1 Computing Graph Automorphisms

A number of open source tools that calculate graph automorphism were considered for use in our implementation. The three candidate tools were bliss [62], nauty [74], and saucy [27]. In preliminary experiments, for the size and type of graphs we consider, no tool has a runtime advantage. We ultimately selected bliss as it provides a Java language wrapper for its C++ interface. As a result, bliss was the easiest tool to program with and integrate into our Java based symmetry reduction tool.

While the Java wrapper is more convenient to use than the C++ interface, it is not suited for performance critical software. However, the graphs we present to bliss are not computationally challenging and no impact on runtime performance was noted.

8.2.2 Checking the Validity of an Element

To check the validity of an element we must determine if $\alpha(\mathcal{P}) \equiv \mathcal{P}$. The implementation of this check requires the application of an element α to $\mathcal{P}$ and the ability to determine if two specifications are equivalent.

To apply an element we create a copy of the abstract syntax tree representation of specification $\mathcal{P}$, which we denote $\mathcal{P}'$. The specification $\alpha(\mathcal{P}')$ is obtained by performing an in order traversal of the abstract syntax tree and replacing every channel name c and global variable name x with α(c) and α(x) respectively. If val $\in \{1, \ldots, n\}$ is being assigned or compared to a pid variable it is replaced by α(val). Finally α acts on the order of statements that appear in the **Initial** process with a statement that appears in position i being moved to position α(i).

Once the specification $\alpha(\mathcal{P}')$ has been obtained, checking whether $\alpha(\mathcal{P}') \equiv \mathcal{P}$ involves a second in order traversal over both specification. Each specification is normalised into a predictable form, by lexicographically sorting the operands in commutative operators, the order of updates in a command and the order of statements in the **Initial** process. If after normalisation $\alpha(\mathcal{P}') \equiv \mathcal{P}$, element α is valid for symmetry reduction.

8.2.3 Calculating the Largest Valid Symmetry

In order to compute the largest subgroup of Aut(ECD($\mathcal{P}$)) valid for symmetry reduction the algorithm given Section 5.3.2 is used. The algorithm uses bliss [62] to calculate graph automorphisms and the implementation discussed in Section 8.2.2 to check the validity of elements.

A key part of the algorithm is the selection of an element in Aut(ECD($\mathcal{P}$)) that permutes two process vertices. The simplest way to achieve this is to search through all the elements in Aut(ECD($\mathcal{P}$)) until an element that performs the required permutation is identified. This procedure is inefficient as it requires multiple searches of Aut(ECD($\mathcal{P}$)). Therefore, we identify all vertices that a single process vertex must be permuted with and locate a set of elements to perform the permutations in a single search of Aut(ECD($\mathcal{P}$)).

To determine if a set of PCTL formulas is invariant under the largest valid symmetry calculated from the specification, we use an implementation of the algorithm given in Figure 7.3. We do not provide experimental results for this algorithm as its behaviour is well documented [37]. For our purpose, if the PCTL formulas are invariant under a group that shares the majority of generators with the group calculated from the specification, the overhead of the algorithm is negligible.

8.2.4 Experimental Results

We now present experimental data gathered from the application of our symmetry detection techniques to a variety of probabilistic specifications. In addition, we extracted channel diagrams from each specification so any runtime costs incurred by our extensions could be viewed. All experiments were conducted using a PC with a 2.4GHz Intel Core 2 Duo processor and 8Gb of main memory, however, the experiments were restricted to running on a single core. The experiments considered a simple mutual exclusion, dining philosophers, resource allocator, three-tiered architecture and a network infection specification. Furthermore, the experiments were performed on several configurations of each specification. The results are presented in Table 8.1

The simple mutual exclusion and dining philosopher specifications are based on the skeleton PSS code presented in Appendix A.1 and Appendix A.2 respectively. For varying configurations of the specification, n denotes the number of symmetric processes. The ECD for a simple mutual exclusion specification where n = 2 was shown in Figure 5.6 and for a n = 3 dinning philosopher specification the associated ECD was shown in Figure 5.8.

The resource allocator specifications are based on the skeleton PSS code presented in Appendix A.5. For varying configurations of the specification, we give sets of processes that share the same priority level and a final set denoting processes that can independently share the resource. For example, the configuration $\{1, 2, 3\}\{4, 5, 6\}\{7, 8, 9\}\{\}$ indicates a resource allocator specification with no sharing and three priority levels. The ECD of a resource allocator specification with configuration $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}\{4, 5, 6\}$ was shown in Figure 5.10.

The three-tiered architecture specifications are based on the skeleton PSS code presented in Appendix A.6. For varying configurations of the specification, we present a list of numbers indicating how many client processes are connected to a single server process. The configuration 5–5–5 indicates a three-tiered architecture with 3 servers and 15 clients, where 5 clients are connected to each server.

Finally, network infection specifications are based on the skeleton PSS code presented in Appendix A.3. For varying specification configurations, n × n denotes the layout of computer processes in the network grid. The ECD of a specification with configuration 3×3 was shown in Figure 5.5.

	$\mathcal{C}(\mathcal{P})$					ECD($\mathcal{P}$)				
Configuration $\mathcal{P}$	$\lvert\text{Aut}(\mathcal{C}(\mathcal{P}))\rvert$	$\lvert\text{G}\rvert$	$\lvert\text{H}\rvert$	bliss	largest	$\lvert\text{Aut}(\text{ECD}(\mathcal{P}))\rvert$	$\lvert\text{G}\rvert$	$\lvert\text{H}\rvert$	bliss	largest
Simple Mutual Exclusion										
5	120	120	120	0.01	0.01	120	120	120	0.01	0.01
10	3.6×10^6	3.6×10^6	3.6×10^6	0.01	0.1	3.6×10^6	3.6×10^6	3.6×10^6	0.01	0.1
20	2.4×10^{18}	2.4×10^{18}	2.4×10^{18}	0.01	0.36	2.4×10^{18}	2.4×10^{18}	2.4×10^{18}	0.01	0.38
40	8.1×10^{37}	8.1×10^{37}	8.1×10^{37}	0.01	0.94	8.1×10^{37}	8.1×10^{37}	8.1×10^{37}	0.01	0.95
Dinning Philosophers										
5	120	120	120	0.03	0.05	120	120	120	0.03	0.05
10	3.6×10^6	3.6×10^6	3.6×10^6	0.03	0.17	3.6×10^6	3.6×10^6	3.6×10^6	0.03	0.24
20	2.4×10^{18}	2.4×10^{18}	2.4×10^{18}	0.04	0.44	2.4×10^{18}	2.4×10^{18}	2.4×10^{18}	0.03	0.64
40	8.1×10^{37}	8.1×10^{37}	8.1×10^{37}	0.04	1.20	8.1×10^{37}	8.1×10^{37}	8.1×10^{37}	0.04	1.85
Three Tiered Architecture										
2–2–2	48	48	48	0.01	0.04	48	48	48	0.01	0.04
3–3–3	1296	1296	1296	0.02	0.05	1296	1296	1296	0.02	0.05
4–4–4	82944	82944	82944	0.02	0.09	82944	82944	82944	0.02	0.09
5–5–5	2.0×10^7	2.0×10^7	2.0×10^7	0.02	0.13	2.0×10^7	2.0×10^7	2.0×10^7	0.02	0.13
5–5–4–4–3–3	2.3×10^8	2.3×10^8	2.3×10^8	0.05	0.28	2.3×10^8	2.3×10^8	2.3×10^8	0.05	0.38
5–5–5–5–5	2.9×10^{11}	2.9×10^{11}	2.9×10^{11}	0.07	0.72	2.9×10^{11}	2.9×10^{11}	2.9×10^{11}	0.07	1.15
Resource Allocator										
{1,2,3}{4,5,6}{7,8,9}{}	362880	1296	1296	0.02	12.33	362880	1296	1296	0.02	12.67
{1,2}{3,4}{5,6}{7,8}{9}{}	326880	384	384	0.02	15.16	326880	384	384	0.02	15.14
{1}{2}{3}{4}{5}{6}{7}{8}{9}{}	362880	0	0	0.00	36.52	362880	0	0	0.00	36.10
{1,3,5,7,9}{2,4,6,8,10}{}	3.6×10^6	0	14400	0.04	21.65	3.6×10^6	0	14400	0.04	21.73
{1,2,3}{4,5,6}{7,8,9}{4,5,6}	362880	4	108	0.02	13.05	2160	108	108	0.01	4.96
{1,2,3}{4,5,6}{7,8,9}{2,5,8}	362880	0	8	0.02	16.21	2160	8	8	0.01	8.99
Network Infection										
3×3	4	4	4	0.01	0.01	4	4	4	0.01	0.01
4×4	4	4	4	0.02	0.01	4	4	4	0.01	0.01
5×5	4	4	4	0.03	0.02	4	4	4	0.02	0.02

Table 8.1: Experimental results showing that automated symmetry detection can be efficiently implemented and that our data detection extensions do not result in a significant computational overheard. Table headings provide the following measurements:

- Configuration – The specification from which the rows results are generated.
- $\lvert\text{Aut}(\mathcal{C}(\mathcal{P}))\rvert$ – The size of the automorphism group of the channel diagram associated with configuration $\mathcal{P}$.
- $\lvert\text{Aut}(\text{ECD}(\mathcal{P}))\rvert$ – The size of the automorphism group of the extended channel diagram associated with configuration $\mathcal{P}$.
- $\lvert\text{G}\rvert$ – The size of the subgroup generated by valid generators of either Aut(ECD($\mathcal{P}$)) or Aut($\mathcal{C}(\mathcal{P})$).
- $\lvert\text{H}\rvert$ – The size of the largest valid subgroup of Aut(ECD($\mathcal{P}$)) or Aut($\mathcal{C}(\mathcal{P})$).
- bliss – The time taken in seconds for bliss to compute Aut(ECD($\mathcal{P}$)) or Aut($\mathcal{C}(\mathcal{P})$).
- largest – The time taken in seconds to compute $\lvert\text{H}\rvert$ using our algorithm.

From Table 8.1, all configurations of the mutual exclusion specification highlight the effi-

ciency of both automated detection techniques, when all generators are valid, irrespective of specification size. For this family of specifications our data detection extensions are equally efficient compared to channel diagrams and no impact on runtime is observed.

Configurations of the dining philosopher specification reveal a similar result. However, a small increase in runtime is observed for calculation of the largest group valid for symmetry reduction when using ECD($\mathcal{P}$). As the order of ECD($\mathcal{P}$) and $\mathcal{C}(\mathcal{P})$ are equal for all configurations, the time taken to apply an element of ECD($\mathcal{P}$) must be measurably longer. Despite this, the data extensions are still efficient and in this instance will provide greater levels of reduction. In practice, for specifications that include data symmetries the extensions will provide greater levels of reduction. Nevertheless, it is possible to write contrived specifications where this is not the case.

Configurations of the three-tiered architecture provide results for the application of our technique to specifications that do not exhibit full symmetry between processes. The results confirm that our extensions can be efficiently applied when all generators of Aut(ECD($\mathcal{P}$)) are valid, irrespective of the type of symmetry exhibited by the specification. As before, a small increase in runtime is incurred by our extensions when calculating the largest group valid for symmetry reduction. However, even for the largest 5–5–5–5–5 configuration the increased run time is negligible and the data extensions will provide more reduction.

For all configurations of the resource allocator specification, the order of G is less than the order of Aut(ECD($\mathcal{P}$)) or Aut($\mathcal{C}(\mathcal{P})$), indicating that a re-colouring of the graph was required. As asserted, the run time of the re-colouring algorithm (see Section 5.3.2) is linked with the level of symmetry exhibited in the specification. As the level of symmetry is reduced over the first 3 configurations the run time of the algorithm increases. However, the runtime of the algorithm is acceptable and not increased by our data detection extensions. This result is expected as these examples do not contain resource sharing, which is provided by global variables.

A disparity between the order of ECD($\mathcal{P}$) and $\mathcal{C}(\mathcal{P})$ is provided by the final two configurations. As the inclusion of data symmetries can only restrict the number of graph automorphisms, the search effort required by our algorithm is reduced. Therefore, we find that computing the largest valid symmetry is markedly quicker in the instance that out extensions more accurately captured the actual symmetries present in the specification.

Finally the results given in network inspection reveal that increasing the size of a specification does not necessarily increase the number of automorphisms in the captured group. Nevertheless, the results reiterate the previous findings. Our automated approach to capturing data and component symmetries can be efficiently applied to a probabilistic specification

language. When compared to the original channel diagram approach, in the worst case a small increase in runtime is observed. However, the small increase in runtime is justified by a potential increase in state space reduction.

8.3 Computing State Representatives Experiments

In Chapter 6 the representative functions we proposed assumed that state components did not hold references to other components. However, the PSS language supports referencing through use of a pid variable. The inclusion of component referencing does not invalidate the techniques presented in Chapter 6. However, when applied to a state generated from a PSS specification, the returned state may not be the lexicographical minimum. To alleviate this problem, we provide exact symmetry reduction by implementing the solution used in the TopSPIN [38] symmetry reduction package.

Let I = $\{1, 2, \ldots, n\}$ be a set of component identifiers and assume a component can be split into two disjoint sections. Using terminology adapted from SymmSpin [13] we refer to the sections as either a control or reference section. A control section is comprised from the values of local variables which are not references to other components and can be abstractly represented as a single integer (see Section 6.1). Conversely, a reference section is comprised from the value of local variables which reference other components i.e. pid values in the PSS language. Hence, the set of control sections are identical to a model of computation without references (see Section 6.1) and can be represented as an array of integers.

It follows that we can apply our representative strategy to control sections of a state and obtain a state representative. In this representative state, the relative ordering of components with equally valued control sections is irrelevant. However, these components may have reference sections with different values. Therefore, to guarantee the representative function returns the lexicographical minimum state, we must consider all orderings of components with equally valued control sections. From this set of control equivalent states, we return the lexicographical minimum with respect to reference sections.

The permutations required to generate the set of control equivalent states are obtained using GAP. The set of elements are applied via enumeration and the lexicographical minimum state with respect to reference sections is returned as the representative. In the remainder of this section we present experiment data on the techniques described in Chapter 6 to assess their effectiveness when applied to a model specification.

8.3.1 Application of an Element

In Section 6.2.1 we detailed two strategies for applying a permutation α to a state s: as a series of transpositions, or as a single direct application. Preliminary experiments indicated, for sets of randomly generated states and permutations, the direct application of permutations becomes more efficient as the number of state components increase.

To repeat the comparison in a realistic setting we use a basic full enumeration algorithm (see Section 6.2) to explore the quotient state space associated with a selection of PSS specifications and configurations. The selected specifications are mutual exclusion, resource allocator and three-tiered architecture, as each exhibits a distinct type of symmetry. The dining philosopher specification is omitted as the symmetry it exhibits is almost identical to that of the mutual exclusion specification and the results will be similar. In addition, results for the network infection specification are not provided, as irrespective of configuration only 4 permutations are applied to each state.

To conduct the experiment we explore the quotient state space associated with each specification twice. The first time applying each element used by the enumeration algorithm directly to a state, and the second time applying each element as a series of transpositions. The time taken to explore the quotient state space using both techniques is provided in Table 8.2. Verification attempts which exceed available resources, or do not terminate within 12 hours are indicated by -.

From Table 8.2, only for the smallest mutual exclusion configurations are the exploration times similar. For all other models the direct application of permutations provides a consistent slight increase in speed. The results suggest that the direct application of permutations is a better approach, and for all proceeding experiments we apply permutations in this manner. However, these results may be implementation specific and cannot be generalised. Nevertheless, they reveal that the application of a permutation as a series of transpositions cannot be assumed as faster as it requires a maximum of n – 1 operations.

8.3.2 Enumeration

In Section 6.2 and Section 6.3 we considered two representative functions based on a full enumeration strategy. The first representative function attempted to narrow the set of permutations applied to each state and the second mapped the calculation to a constraint satisfaction problem.

To provide a comparison between these representative functions, a base line time is set using

Configuration	States	|H|	Direct	Transposition
Mutual Exclusion				
5	12	120	0.10	0.10
10	22	3.6×10^6	11364	12672
15	-	3.0×10^{12}	-	-
20	-	2.4×10^{18}	-	-
Resource Allocator				
{1,2,3}{4,5,6}{}	1584	36	8.01	8.14
{1,3,5,7}{2,4,6,8}{}	3947	576	319	347
{1,2,3,4,5}{6,7,8,9,10}{}	8311	14400	15009	16511
Three-tiered Architecture				
2–2–2	12884	48	89	90
3–3–3	48,737	1296	8867	8965
4–4–4	-	82944	-	-
5–5–5	-	2.0×10^7	-	-

Table 8.2: Experimental results showing the direct application of permutations to a state provide a consistent slight increase in speed. Table headings give the following measurements:

- Configuration – The specification from which the rows results are generated.
- States – The number of states in the quotient state space.
- |H| – The size of the largest group valid for symmetry reduction
- Direct – The time in seconds taken to generate the quotient state space when directly applying permutations.
- Transposition – The time in seconds taken to generate the quotient state space when applying permutations as a series of transpositions.

the basic full enumeration algorithm to explore the quotient state space associated with a selection of PSS specifications $\mathcal{P}$. The selected specifications are mutual exclusion, resource allocator and three-tiered architecture, as each exhibits a distinct type of symmetry. Subsequently, we explore each quotient state space using the restriction of elements and the CSP mapping to provide comparable times. The findings of the experimental results are provided in Table 8.3. Verification attempts which exceed available resources, or do not terminate within 12 hours, are indicated by -.

For configurations of the simple mutual exclusion model we observe that attempting to narrow the set of applied elements is a valid strategy that improves runtime by a constant factor. A similar result is seen in the resource allocator model, however, as we alter configurations to reduce the size of |H|, the runtime improvements rapidly diminish. For configurations of the three-tiered architecture model, no runtime improvement is provided. The results suggest that attempting to narrow the set of elements is profitable when the model exhibits full symmetry between components. However, for models with this property a more efficient

Configuration	States	|H|	Direct	Restrict	CSP
Mutual Exclusion					
5	12	120	0.10	0.7	0.16
10	22	3.6×10^6	11364	8684	8517
15	-	-	-	-	-
20	-	-	-	-	-
Resource Allocator					
{1,2,3}{4,5,6}{}	1584	36	8.01	7.61	7.59
{1,3,5,7}{2,4,6,8}{}	3947	576	319	255	211
{1,2,3,4,5}{6,7,8,9,10}{}	8311	14400	15009	11256	10373
Three-tiered Architecture					
2–2–2	12884	48	89	89	81
3–3–3	48737	1296	8867	8867	8672
4–4–4	-	82944	-	-	-
5–5–5	-	2.0×10^7	-	-	-

Table 8.3: Experimental results showing that the Restrict and CSP representative calculation offer slight runtime improvements. Table headings give the following measurements:

- Configuration – The specification from which the rows results are generated.
- States – The number of states in the quotient state space.
- |H| – The size of the largest group valid for symmetry reduction
- Direct – The time in seconds taken to generate the quotient state space when directly applying permutations.
- Restrict – The time in seconds taken to generate the quotient state space when directly applying a potentially narrowed set of permutations.
- CSP – The time in seconds taken to generate the quotient state space when applying permutations as a CSP.

representative function can normally be generated (see Section 6.5.2).

In addition, our novel approach to solving full enumeration as a constraint satisfaction problem yields a slight runtime improvement across all specifications and configurations. However, this runtime improvement is not significant enough to allow for larger configurations of any specification to be generated. Therefore, using information about the structure of G to design representative functions is of vital importance.

8.3.3 Local Search

In Section 6.4 we hypothesised that improvements in the level of reduction could be obtained by replacing unsuccessful elements during hill climbing local search. The two replacement strategies we considered were:

- randomly selecting new elements from the group G.
- using the most successful elements to generate new elements.

To provide a comparison between these techniques the smallest quotient state space associated with a selection of PSS specifications $\mathcal{P}$ was generated using the full enumeration algorithm. As before, the selected specifications are mutual exclusion, resource allocator and three-tiered architecture, as each exhibits a distinct type of symmetry. Using the element replacement conditions discussed in Section 6.4.1 we subsequently explore each state space using hill climbing local search and both replacement strategies. The findings of the experimental results are provided in Table 8.4, verification attempts which exceed available resources, or do not terminate within 12 hours, are indicated by -.

Configuration	States	Random	Successful
Mutual Exclusion			
5	12	67	67
10	22	4117	4117
15	-	116947	97525
20	-	422169	374193
Resource Allocator			
{1,2,3}{4,5,6}{}	1584	5562	5562
{1,3,5,7}{2,4,6,8}{}	3947	7134	7016
{1,2,3,4,5}{6,7,8,9}{}	8311	87352	77473
Three-tiered Architecture			
2–2–2	12884	54646	54646
3–3–3	48737	107396	104399
4–4–4	-	274765	239662
5–5–5	-	446925	416429

Table 8.4: Experimental results showing that using successful elements to generate new elements is a more effective approach. Table headings give the following measurements:

- Configuration – The specification from which the rows results are generated.
- States – The number of states in the quotient state space generated using the full enumeration algorithm.
- Random – The number of states in the quotient model when randomly selecting new elements from the group G.
- Successful – The number of states in the quotient state space when using the most successful elements to generate new elements.

As predicted, the results in Table 8.4 show that using previously successful elements to generate new elements gives the largest level of reduction. Furthermore, while the level of reduction is less than that obtained by full enumeration the technique is considerably faster and was able to generate a quotient state space in cases where full enumeration failed.

8.4 Probabilistic Model Checking

Here we present experimental results which demonstrate the effectiveness of applying our symmetry reduction techniques to probabilistic model checking. Our experiments considered configurations of a simple mutual exclusion, dining philosophers, resource allocator, three-tiered architecture and a network infection specification. Therefore, the performance of the reduction techniques can be viewed across a varied selection of PSS specifications.

For each specification and configuration, we give the number of states and the time taken to generate the unreduced state space. The number of states and the time taken to generate the quotient state space when using full enumeration, and the representative function selected from the analyses of group G are also provided. In addition, the time taken for the PRISM model checkers explicit engine to verify a relevant PCTL property on the unreduced and quotient state space is given to aid the comparison. The results are presented in Table 8.5 and verification attempts which exceed available resources, or do not terminate within 12 hours, are indicated by -.

When our reduction techniques are applied to the mutual exclusion specification, GAP detects that the group of symmetries associated with all configurations are isomorphic to the group S_n. Therefore, the selected representative function is a homomorphic mapping between these groups and the group S_n. For this family of specifications, limitations of full enumeration are clear. For all configurations, the full enumeration approach takes longer than simply exploring the full unreduced state space. Furthermore, for larger configurations, full enumeration fails to generate the quotient state space within the experimental time limit. However, homomorphic mappings exploit the structure G to provide an efficient approach to symmetry reduction. For all configurations, the time taken to explore the quotient state space is significantly less than the time required to explore the full unreduced state space.

Similarly, when our reduction techniques are applied to the dining philosophers specification, GAP detects that the group of symmetries associated with all configurations are isomorphic to the group S_n. Once again, the selected representative function is a homomorphic mapping between these groups and the group S_n. Therefore, the conclusions that can be drawn from this set of experiments are the same as the mutual exclusion set. However, they clearly illustrate that even when applying symmetry reduction to a fully symmetric specification, the quotient state space can quickly become intractable. Nevertheless, the selected technique allows the quotient state space to be constructed for larger configurations.

For all considered configurations of the three-tiered architecture specification, GAP detects that the group of symmetries associated with the specification are formed from the direct

Configuration	Full		Enumeration		Selected		PRISM PCTL	
	States	Time	States	Time	States	Time	Full	Quotient
Mutual Exclusion								
5	113	0.08	12	0.16	12	0.01	0.01	0.01
10	6145	0.16	22	8517	22	0.03	0.34	0.01
15	278529	7	-	-	32	0.04	8	0.01
20	1.2×10^7	511	-	-	42	0.04	818	0.01
Dining Philosophers								
3	956	0.21	184	2	184	0.07	0.03	0.01
6	917424	11	-	-	33304	134	12	5
9	-	-	-	-	8.74×10^6	3973	-	23
12	-	-	-	-	-	-	-	-
Three-tiered Architecture								
2–2–2	1.0×10^6	37	12884	81	12884	64	8	0.1
3–3–3	2.5×10^7	6544	48737	8672	48737	182	172	0.3
4–4–4	-	-	-	-	1.3×10^6	2642	-	6.3
Resource Allocator								
{1,2,3,4,5,6} {}	-	-	86314	716	86314	18	-	0.71
{1,2,3}{4,5,6}{}	19511	0.1	1584	8	2134	3	0.42	0.23
{1,3,5,7}{2,4,6,8}{}	217395	3	3947	319	13947	15	0.51	0.03
{1,2,3,4,5}{6,7,8,9,10}{}	2.2×10^6	17	8311	4009	332164	85	17	0.03
Network Infection								
3×3	1125	0.01	375	0.02	375	0.02	2	1
4×4	103105	1	34369	1	34369	1	7	3
5×5	-	-	-	-	-	-	-	-

Table 8.5: Experimental results comparing several approaches to state probabilistic model checking. Table headings give the following measurements:

- Configuration – The specification from which the rows results are generated.
- Full – The number of states in the unreduced state space and the time taken in seconds to construct it.
- Enumeration – The number of states in the quotient state space and the time taken in seconds to construct it using a full enumeration strategy.
- Selected – The number of states in the quotient state space and the time taken in seconds to construct it using a strategy selected by analysing the group of symmetries present within the associated specification.
- PRISM PCTL – The time in seconds taken for the PRISM model checkers explicit engine to verify a relevant PCTL property on the full and quotient state space.

product of various subgroups. While GAP is able to describe the structure of the group, it cannot efficiently return the subgroups involved in the direct product. The only means we are aware of is via a brute force factorisation to obtain all normal subgroups of G. For non trivial groups this calculation can be time prohibitive and as a result we provide the group decomposition by hand. This does not invalidate the technique, as an algorithm capable of performing direct product decompositions in polynomial time is known [95] but not implemented as part of the tool.

As previously indicated, the full enumeration approach takes longer than simply exploring the unreduced state space. However, the selected representative function shows the viability of our symmetry reduction techniques, even when applied to specifications that do not exhibit full symmetry between components. The reduction technique allows the quotient state space to be constructed for the larger 4-4-4 configuration and for the 2-2-2 configuration, the time required for state space exploration is notably reduced.

For resource allocator specification with configuration $\{1, 2, 3, 4, 5, 6\}\{\}$, GAP classifies the symmetry group associated with the specification as isomorphic to S_n and a homomorphic mapping is generated. For the remainder of the configurations, GAP detects that the associated symmetries are formed from a disjoint product. While our techniques could be independently applied to the resultant subgroups, we have not implemented a disjoint decomposition algorithm [37] and the local hill climbing search strategy is selected to provide reduction. The results show that when no tailored representative function can be selected, the heuristic approach provides significant run time benefits over the full enumeration approach. However, the trade off is a larger state space.

Finally, when applying our techniques to the network infection specification, GAP detects that the group of symmetries associated with all configurations are dihedral groups. As the number of elements in dihedral groups are small, symmetry reduction is provided using full enumeration. In this instance, the time taken to construct the unreduced and quotient state space are similar. However, the resultant quotient structure contains significantly less states and therefore, the computational effort required to solve PCTL properties will be reduced.

These results demonstrate the effectiveness of our symmetry reduction techniques. For all specifications, except the resource allocator, the time taken to explore the quotient state space using the selected method was similar or quicker than exploring the full state space. In addition, the probabilistic model checking provided by PRISM's explicit engine was notably faster when performed using the quotient state space. When comparing the combined time of model construction and property checking, our symmetry reduction techniques provide a substantial decrease in runtime. Furthermore, in some instances the application of our techniques allowed probabilistic model checking to be performed on larger model configurations.

8.5 Comparison with PRISM, PRISM-symm and GRIP

In the previous sections we have provided experimental evidence that shows symmetry reduction can be effectively applied to probabilistic explicit state model checking. However, in Section 3.7.1 of the literature review, we identified the tools PRISM-symm [68] and

GRIP [32], which combine symmetry reduction and probabilistic symbolic model checking techniques. Therefore, we provide a comparison between the techniques that aims to uncover the strengths and weaknesses of the differing approaches.

The main challenge to conducting a fair comparison is overcoming the differences in input languages. While PSS and PRISM are similar, they are not directly equivalent. Furthermore, PRISM-symm and GRIP only operate on a subset of the PRISM language and all models are required to be fully symmetric. To conduct the comparison, we have identified three specifications: a randomised consensus protocol [5], Rabin's mutual exclusion [89] and a minimum space shared memory leader election protocol [86]. These specifications were selected as they are available online [2], have been used in previous comparisons between PRISM-symm and GRIP [33], and only require minor modifications to be translated into valid PSS specifications.

To translate the specifications into valid PSS specifications, local variables declared within PRISM modules are declared as global integer variables. The translation is required as the PRISM language allows a command within a module to read the value of local variables declared within other modules. This type of behaviour is prohibited in PSS, but declaring the variable as a global integer allows its value to be read and does not hinder the application of our symmetry detection and reduction techniques. All other changes are minor syntactical modifications.

The results of the comparison are presented in Table 8.6 and verification attempts which exceed available resources, or do not terminate within 12 hours, are indicated by -. For the model building process, PRISM translates a specification into an MTBDD representation, the set of all reachable states are computed and any state not reachable from the initial state is removed. PRISM-symm follows the same construction process, but subsequently applies symmetry reduction to the MTBDD representation. GRIP applies language-level symmetry reduction to the given specification, resulting in a reduced generic specification which is passed to PRISM for construction. The PSS tool starts from the equivalent PSS specification and applies our symmetry detection techniques. As the mentioned specifications are isomorphic to the group S_n, the selected representative function is a homomorphic mapping. This representative function is then utilised in the construction of the reduced model. Finally, each constructed model is checked using PRISM and the fastest technique for each tool is used.

The results show the time required by our techniques to construct the quotient model is significantly longer than the time required by the other approaches. Furthermore, our technique fails to construct the quotient state space when considering larger configurations. However, model checking times are favourable. For the consensus and leader election protocols, the

Configurations	States		Build Time				Model Checking Time			
	Full	Reduced	PRISM	PRISM-symm	GRIP	PSS	PRISM	PRISM-symm	GRIP	PSS
Consensus										
12	1.2×10^{11}	339729	1	1	2	128	16993	8	5	0.63
14	5.0×10^{12}	747243	3	3	4	374	-	121	79	3
16	2.1×10^{14}	1.5×10^{6}	7	9	7	-	-	847	456	-
Rabin										
4	201828	11130	1	2	7	236	0.5	0.12	0.6	0.03
6	1.3×10^{8}	356592	4	10	28	681	0.29	0.39	0.32	0.12
8	4.5×10^{10}	4.1×10^{6}	11	36	62	-	0.97	1	1	-
Leader										
60	4.2×10^{28}	1891	2	12	1	11	47	3	0.03	0.01
100	5.2×10^{47}	5151	8	137	1	32	889	14	0.18	0.01
140	6.3×10^{66}	10011	26	897	1	60	4051	46	0.37	0.07
Three-tiered Architecture										
2–2–2	430080	6 144	0.66	N/A	N/A	tf	307	N/A	N/A	0.43
3–3–3	1.9×10^{6}	26388	3	N/A	N/A	tf	3698	N/A	N/A	2
4–4–4	1.0×10^{8}	1.0×10^{6}	27	N/A	N/A	tf	-	N/A	N/A	6

Table 8.6: Experimental results showing a comparison between several approaches to probabilistic symmetry reduction. Table headings give the following measurements:

- Configuration – The specification from which the rows results are generated.
- States – The number of states in the quotient and unreduced state space.
- Build Time – The time in seconds required by each technique to build the model.
- Model Check Time – The time taken in seconds for the PRISM model checker to verify a relevant PCTL property against each model.

model checking times are either comparable or faster than GRIP, which in turn is faster than PRISM-symm and PRISM. For configurations of Rabin's mutual exclusion specification, model checking times are similar for all techniques.

While results suggest that GRIP outperforms PRISM-symm, this is a miss leading result. Typically, GRIP is faster for models that contain a large number of simple modules, whereas PRISM-symm performs better on models constructed from a small number of more complex modules [33]. This result is not shown, as our comparisons purposively avoid specifications with complex modules to allow translation into PSS. Therefore, when considering fully symmetric specifications GRIP and PRISM-symm have the advantage of being able to verify properties against larger models.

Nevertheless, a major advantage of our technique is its ability to be applied to specifications that do not exhibit full symmetry. In the final experiment of Table 8.6, results from a simplified three-tiered architecture specification are presented. In this instance, our techniques enable model checking to be successfully performed in instances where PRISM's sparse engine failed and GRIP and PRISM-symm could not be applied. Therefore, when considering specifications that exhibit more complex forms of symmetry, our approach offers a clear advantage.

8.6 Summary

In this chapter we have implemented the techniques discussed within the thesis and the resulting tool is used to test the viability of our approach to automated symmetry reduction. We find that our automated approach to capturing data and component symmetries can be efficiently applied to a probabilistic specification language. Furthermore, for a variety of symmetric specifications we show the significant runtime and memory savings can be made while performing probabilistic model checking. Furthermore, when considering specifications that exhibit complex forms of symmetry, our approach offers a clear advantage.

CHAPTER 9

Conclusion and Future work

The main contribution of this thesis was to show how symmetry reduction techniques can be applied to explicit state probabilistic model checking. We proved the correctness of our approach, and demonstrated its viability by implementing our techniques in a symmetry reduction tool.

Our contribution began in Chapter 4 where we formally defined the Probabilistic Symmetric Systems Language, presenting a full language grammar and semantics. The language was specifically designed to allow the creation of models exhibiting complex symmetry groups while being simple enough to allow rigorous proof. Throughout the thesis the language has been extensively used to specify the required models and prove the correctness of our techniques.

In Chapter 5 we introduced an approach to symmetry detection that can automatically detect arbitrary component and data symmetries directly from our probabilistic specification language. Given a specification, this approach extracts a diagrammatic representation of communication that may occur between components. Automorphisms of this diagram, which we refer to as an extended channel diagram, were shown to correspond to automorphisms in the underlying probabilistic model. Provided these automorphisms meet a small set of restrictions, they are valid for symmetry reduction. Finally, from this set of valid automorphisms we showed how a potentially larger valid set could be be calculated.

In Chapter 6 we presented new techniques to efficiently compute equivalence class representatives for certain classes of symmetry groups identified in Chapter 5. The presented techniques included a novel mapping of exhaustive search to a constraint satisfaction problem and a hill climbing local search algorithm. Additionally, we suggested efficient techniques to handle fully symmetric groups, cyclic groups and groups that could be decomposed as an internal direct product or as an internal semi direct product.

In Chapter 7 we outlined an algorithm to construct the quotient probabilistic model directly from a PSS specification. The algorithm was obtained by modifying graph traversal algorithms commonly employed in model checking. Several common implementations of data structures were assessed to determine their compatibility with our symmetry reduction techniques.

In Chapter 8 we implemented our techniques and tested their viability. Experimental results showed our automated approach to capturing data and component symmetries could be efficiently applied to a wide family of probabilistic specifications. These included various configurations of a simple mutual exclusion, dining philosophers, resource allocator, three-tiered architecture and network infection specification. Furthermore, for the same specifications, we demonstrated significant runtime and state space gains by applying our symmetry reduction techniques to probabilistic model checking. Finally, we compared our techniques to other documented approaches to probabilistic symmetry reduction and conclude that when considering specifications that exhibit complex forms of symmetry, our approach offered a clear advantage.

9.1 Future Work

In Chapter 6 we illustrated that for groups decomposable as an internal direct or semi direct product, a state representative can be obtained by considering their subgroups in isolation. The main issue with this approach is how to efficiently obtaining the subgroups without resorting to brute force factorisation. While a polynomial time algorithm is known for the direct product, no such algorithm is available for a semi direct product. However, a possible decomposition may be indicated by examining the structure of the ECD. Being able to quickly decompose the groups would greatly increase the practicality of the approach.

Furthermore, some symmetry reduction strategies are clearly parallelisable. If we have n processor cores, group G could be split into n equally-sized disjoint subsets. Each core could independently apply enumeration and the set of n potential representatives compared to obtain the lexicographical minimum.

However, more research is required into parallel algorithms that can exploit the structural properties of a symmetry group. An ideal candidate would be the parallel application of the subgroups obtained from a direct product decomposition. While we noted that the groups could be considered in series, how they could be considered in parallel requires research.

Finally, probabilistic partial order reduction is an alternative reduction technique that has been applied to probabilistic explicit state model checking. Partial order reduction attempts to construct a reduced state space by removing redundancies in the transition system. Therefore, partial order reduction reduction works on transitions while symmetry focuses on states. The combination of these two techniques is theoretically possible and would provide a novel extension to our current implementation.

APPENDIX A

Skeleton Code

In this appendix we provide skeleton example of the PSS specifications mentioned during the course of the thesis. The presentation aims to highlight the component and data symmetries present within the specifications structure.

A.1 Simple Mutual Exclusion

```
1. dtmc
2.
3. pid[3] full;
4.
5. Process mex()
6. {
7. }
8.
9. Initial{mex(); mex();}
```

A.2 Dining Philosophers

```
1.  mdp
2.
2.  chan fork1 [1] of {int};
3.  chan fork2 [1] of {int};
4.  chan fork3 [1] of {int};
6.
7.  Process dphil(chan left, chan right)
8.  {
9.  }
10.
11. Initial{dphil(fork3, fork1); dphil(fork1, fork2); dphil(fork2, fork3);}
```

```
1.  mdp
2.
3.  pid[3] fork;
6.
7.  Process dphil()
8.  {
9.  }
10.
11. Initial{dphil(); dphil(); dphil();}
```

A.3 Network Infection

```
1.  mdp
2.
2.  chan link1 [1] of {int};
3.  chan link2 [1] of {int};
4.  chan link3 [1] of {int};
5.  chan link4 [1] of {int};
6.  chan link5 [1] of {int};
7.  chan link6 [1] of {int};
8.  chan link7 [1] of {int};
9.  chan link8 [1] of {int};
10. chan link9 [1] of {int};
11. chan link10 [1] of {int};
12. chan link11 [1] of {int};
13. chan link12 [1] of {int};
14.
15. Process C2(chan con1, chan con2)
16. {
17. }
18.
19. Process C3(chan con1, chan con2, chan con3)
20. {
21. }
22.
22. Process C4(chan con1, chan con2, chan con3, chan con4)
24. {
25. }
26.
27. Initial {C2(link1, link3); C2(link5, link2); C2(link8, link9);
            C2(link12, link10); C3(link2, link4,link1);C3(link3, link6,link8);
            C3(link10, link7, link5); C3(link11, link9, link12);
            C4(link6, link4, link7, link9);}
```

A.4 Monty Hall Problem

```
1. dtmc
2.
3. int[3] door;
4.
5. Process MontyHall()
6. {
7. }
8.
9. Initial{MontyHall();}
```

A.5 Resource Allocator

```
1.  mdp
2.
3.  int[3] resource;
4.  chan link1 [1] of {int};
5.  chan link2 [1] of {int};
6.  chan link3 [1] of {int};
7.  chan link4 [1] of {int};
8.  chan link5 [1] of {int};
9.  chan link6 [1] of {int};
10. chan link7 [1] of {int};
11. chan link8 [1] of {int};
12. chan link9 [1] of {int};
13.
14. Process RA(chan con1, chan con2, chan con3,
                chan con4, chan con5, chan con6,
                chan con7, chan con8, chan con9)
15. {
16. }
17.
18. Process CP(chan con1)
19. {
20. }
21.
22. Process C4(chan con1, chan con2, chan con3, chan con4)
23. {
24. }
25.
26. Initial {CP(link1, link2, link3, link4, link5, link6, link7, link8, link9);
            CP(link1); CP(link2); CP(link3); CP(link4); CP(link5);
            CP(link6); CP(link7); CP(link8); CP(link9);}
```

A.6 Three tiered architecture

```
1. mdp
2.
4. chan link1 [1] of {int};
5. chan link2 [1] of {int};
6. chan link3 [1] of {int};
7. chan link4 [1] of {int};
8. chan link5 [1] of {int};
9. chan link6 [1] of {int};
10.
11. Process DB(chan con1, chan con2)
12. {
13. }
14.
15. Process Server(chan con1, chan con2, chan con3)
16. {
17. }
18.
19. Process Client(chan con1)
20. {
21. }
22.
23. Initial {DB(link1, link2); Server(link1, link3, link4);Server(link2, link5, link6)
    Client(link3); Client(link4); Client(link5); Client(link6)}
```

Bibliography

[1] "Dice programs," [accessed 20-March-2012]. [Online]. Available: http://www.prismmodelchecker.org/casestudies/dice.php

[2] "Experimental specifications," [accessed 20-March-2012]. [Online]. Available: http://www.prismmodelchecker.org/grip/

[3] "Two process mutual exclusion," [accessed 20-March-2012]. [Online]. Available: http://www.prismmodelchecker.org/manual/ThePRISMLanguage/Example1

[4] S. B. Akers, "Binary decision diagrams," *IEEE Transactions on Computers*, vol. 100, no. 27, pp. 509–516, 1978.

[5] J. Aspnes and M. Herlihy, "Fast randomized consensus using shared memory," *Journal of Algorithms*, vol. 15, no. 1, pp. 441–460, 1990.

[6] C. Baier, E. Clarke, V. Hartonas-Garmhausen, M. Kwiatkowska, and M. Ryan, "Symbolic model checking for probabilistic processes," in *Proceedings of the* 24$^{\text{th}}$ *International Colloquium on Automata, Languages and Programming*, ser. Lecture Notes in Computer Science, P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, Eds., vol. 1256. Springer, 1997, pp. 430 – 440.

[7] C. Baier, M. Groesser, and F. Ciesinski, "Partial order reduction for probabilistic systems," in *Proceedings of the* 1$^{\text{st}}$ *International Conference on Quantitative Evaluation of Systems*. IEEE Computer Society Press, 2004, pp. 230 – 239.

[8] C. Baier, F. Ciesinski, and M. Grosser, "ProbMela and verification of Markov decision processes," *ACM SIGMETRICS Performance Evaluation Review*, vol. 32, pp. 22–27, 2005.

[9] C. Baier and J.-P. Katoen, *Principles of Model Checking*. MIT Press, 2008.

[10] C. Baier and M. Kwiatkowska, "Model checking for a probabilistic branching time logic with fairness," *Distributed Computing*, vol. 11, pp. 125–155, 1998.

[11] M. Ben-Ari, Z. Manna, and A. Pnueli, "The temporal logic of branching time," in *Proceedings of the* 8th *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '81. ACM, 1981, pp. 164–176.

[12] J. Bosch, *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley Professional, 2000.

[13] D. Bošnački, D. Dams, and L. Holenderski, "Symmetric Spin," *International Journal on Software Tools for Technology Transfer*, vol. 4, no. 1, pp. 92–106, 2002.

[14] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a BDD package," in *Proceedings of the* 27th *ACM/IEEE Conference on Design Automation*. IEEE/ACM, ACM Press, 1991, pp. 40–45.

[15] M. C. Browne, E. M. Clarke, and O. Grumberg, "Characterizing finite Kripke structures in propositional temporal logic," *Theoretical Computer Science*, vol. 59, pp. 115–131, 1988.

[16] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang, "Symbolic model checking: 10^{20} states and beyond," in *Proceedings of the* 5th *Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1990, pp. 428 – 439.

[17] F. Ciesinski and C. Baier, "LiQuor: A tool for qualitative and quantitative linear time analysis of reactive systems," in *Proceedings of the* 3rd *International Conference on the Quantitative Evaluation of Systems*. IEEE Computer Society, 2006, pp. 131–132.

[18] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv 2: An opensource tool for symbolic model checking," in *Computer Aided Verification*. Springer, 2002, pp. 241–268.

[19] E. Clarke and E. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," *Proceedings of the Workshop on Logic of Programs Logics of Programs*, vol. 131, pp. 234 – 248, 1981.

[20] E. Clarke, E. Emerson, S. Jha, and A. Sistla, "Symmetry reductions in model checking," in *Proceedings of Computer Aided Verification*, A. Hu and M. Vardi, Eds., vol. 1427. Springer, 1998, pp. 147–158.

[21] E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen, "Symbolic model checking," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, R. Alur and T. Henzinger, Eds. Springer Berlin / Heidelberg, 1996, vol. 1102, pp. 419–422.

[22] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems*, vol. 8, pp. 244–263, 1986.

[23] E. M. Clarke, Emerson, E. Allen, and J. Sifakis, "Model checking: Algorithmic verification and debugging," *Communications of the ACM – Scratch Programming for All*, vol. 52, no. 11, pp. 74–84, 2009.

[24] E. M. Clarke and J. M. Wing, "Formal methods: State of the art and future directions," *ACM Computing Surveys*, vol. 28, pp. 626–643, 1996.

[25] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 1999.

[26] D. Corneil and D. Kirkpatrick, "A theoretical analysis of various heuristics for the graph isomorphism problem," *SIAM Journal on Computing*, vol. 9, pp. 281–297, 1980.

[27] P. Darga, K. Sakallah, and I. Markov, "Faster symmetry discovery using sparsity of symmetries," in *Proceedings of the* 45^{th} *Annual Design Automation Conference*. ACM, 2008, pp. 149–154.

[28] P. D'Argenio and P. Niebert, "Partial order reduction on concurrent probabilistic programs," in *Proceedings of the* 1^{st} *International Conference on Quantitative Evaluation of Systems*. IEEE Computer Society Press, 2004, pp. 240 – 249.

[29] D. Dill, A. Drexler, A. Hu, and C. Yang, "Protocol verification as a hardware design aid," in *IEEE International Conference on Computer Design: VLSI in Computers and Processors*. IEEE Computer Society, 1992, pp. 522–525.

[30] A. Donaldson and A. Miller, "Automatic symmetry detection for model checking using computational group theory," in *FM 2005: Formal Methods*, ser. Lecture Notes in Computer Science, J. Fitzgerald, I. Hayes, and A. Tarlecki, Eds. Springer Berlin / Heidelberg, 2005, vol. 3582, pp. 631–631.

[31] A. Donaldson and A. Miller, "Symmetry reduction for probabilistic model checking using generic representatives," in *Proceedings of the* 4th *International Symposium on Automated Technology for Verification and Analysis*, ser. Lecture Notes in Computer Science, S. Graf and W. Zhang, Eds., vol. 4218. Springer, 2006, pp. 9 – 23.

[32] A. Donaldson, A. Miller, and D. Parker, "GRIP: Generic representatives in PRISM," in *Proceedings of the* 4th *International Conference on Quantitative Evaluation of Systems (QEST'07)*. IEEE Computer Society, 2007, pp. 115–116.

[33] A. Donaldson, A. Miller, and D. Parker, "Language-level symmetry reduction for probabilistic model checking," in *Proceedings of the* 6th *International Conference on Quantitative Evaluation of Systems*. IEEE Computer Society, 2009, pp. 289–298.

[34] A. Donaldson and S. Gay, "ETCH: An enhanced type checking tool for Promela," in *Model Checking Software*, ser. Lecture Notes in Computer Science, P. Godefroid, Ed. Springer Berlin / Heidelberg, 2005, vol. 3639, pp. 902–902.

[35] A. Donaldson and A. Miller, "A computational group theoretic symmetry reduction package for the SPIN model checker," in *Algebraic Methodology and Software Technology*, ser. Lecture Notes in Computer Science, M. Johnson and V. Vene, Eds. Springer Berlin / Heidelberg, 2006, vol. 4019, pp. 374–380.

[36] A. Donaldson and A. Miller, "Exact and approximate strategies for symmetry reduction in model checking," in *FM 2006: Formal Methods*, ser. Lecture Notes in Computer Science, J. Misra, T. Nipkow, and E. Sekerinski, Eds. Springer Berlin / Heidelberg, 2006, vol. 4085, pp. 541–556.

[37] A. Donaldson and A. Miller, "Automatic symmetry detection for Promela," *Journal of Automated Reasoning*, vol. 41, pp. 251–293, 2008.

[38] A. F. Donaldson, "Automatic techniques for detecting and exploiting symmetry in model checking," Ph.D. dissertation, University of Glasgow, 2007.

[39] A. F. Donaldson and A. Miller, "Extending symmetry reduction techniques to a realistic model of computation," *Electronic Notes in Theoretical Computer Science*, vol. 185, pp. 63–76, 2007.

[40] A. F. Donaldson, A. Miller, and M. Calder, "Finding symmetry in models of concurrent systems by static channel diagram analysis," *Electronic Notes in Theoretical Computer Science*, vol. 128, no. 6, pp. 161–177, 2005.

[41] R. Drechsler, N. Gockel, and B. Becker, "Learning heuristics for OBDD minimization by evolutionary algorithms," in *Proceedings of the* 4th *International Conference on*

Parallel Problem Solving from Nature, ser. PPSN IV. Springer-Verlag, 1996, pp. 730–739.

[42] I. S. Duff, R. G. Grimes, and J. G. Lewis, "Sparse matrix test problems," *ACM Transactions Mathematical Software*, vol. 15, pp. 1–14, 1989.

[43] E. Emerson and R. Trefler, "From asymmetry to full symmetry: New techniques for symmetry reduction in model checking," *Correct Hardware Design and Verification Methods*, pp. 704–704, 1999.

[44] E. Emerson and T. Wahl, "On combining symmetry reduction and symbolic representation for efficient model checking," in *Correct Hardware Design and Verification Methods*, ser. Lecture Notes in Computer Science, D. Geist and E. Tronci, Eds. Springer Berlin / Heidelberg, 2003, vol. 2860, pp. 216–230.

[45] E. Emerson and T. Wahl, "Dynamic symmetry reduction," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, N. Halbwachs and L. Zuck, Eds. Springer Berlin / Heidelberg, 2005, vol. 3440, pp. 382–396.

[46] E. A. Emerson, J. W. Havlicek, and R. J. Trefler, "Virtual symmetry reduction," *Proceedings of the* 15th *Annual IEEE Symposium on Logic in Computer Science*, pp. 121–131, 2000.

[47] E. A. Emerson and A. P. Sistla, "Symmetry and model checking," *Formal Methods in System Design*, vol. 9, pp. 105–131, 1996.

[48] E. Emerson and J. Halpern, ""Sometimes" and "not never" revisited: on branching versus linear time temporal logic," *Journal of the ACM*, vol. 33, no. 1, pp. 151–178, 1986.

[49] M. Fitting, *First-order Logic and Automated Theorem Proving (2nd ed.)*. Springer-Verlag New York, Inc., 1996.

[50] M. Fujita, P. C. McGeer, and J. C.-Y. Yang, "Multi-terminal binary decision diagrams: An efficient datastructure for matrix representation," *Formal Methods System Design*, vol. 10, pp. 149–169, 1997.

[51] *GAP – Groups, Algorithms, and Programming, Version 4.4.12*, The GAP Group, 2008. [Online]. Available: http://www.gap-system.org

[52] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, J. v. Leeuwen, J. Hartmanis, and G. Goos, Eds. Springer, 1996.

[53] H. Hansson and B. Jonsson, "A logic for reasoning about time and reliability," *Formal Aspects of Computing*, vol. 6, no. 5, pp. 512–535, 1994.

[54] K. Heljanko and I. Niemela, "Bounded LTL model checking with stable models," *Theory and Practice of Logic Programming*, vol. 3, pp. 519–550, 2003.

[55] I. Herstein, *Topics in Algebra*. John Wiley & Sons, 1975.

[56] D. Holt, B. Eick, and E. O'Brien, *Handbook of Computational Group Theory*. CRC Press, 2005, vol. 24.

[57] G. Holzmann, "State compression in SPIN: Recursive indexing and compression training runs," in *Proceedings of the* 3rd *International SPIN Workshop*, 1997, pp. 1–10.

[58] G. Holzmann, *The SPIN Model Checker – Primer and Reference Manual*. Addison-Wesley Professional, 2004.

[59] G. Holzmann and R. Joshi, "Model-driven software verification," in *Model Checking Software*, ser. Lecture Notes in Computer Science, S. Graf and L. Mounier, Eds. Springer Berlin / Heidelberg, 2004, vol. 2989, pp. 76–91.

[60] G. J. Holzmann, "The logic of bugs," in *Proceedings of the* 10th *ACM SIGSOFT Symposium on Foundations of Software Engineering*, ser. SIGSOFT '02/FSE-10. ACM, 2002, pp. 81–87.

[61] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2004.

[62] T. Junttila and P. Kaski, "Engineering an efficient canonical labeling tool for large and sparse graphs," in *Proceedings of the* 9th *Workshop on Algorithm Engineering and Experiments and the* 4th *Workshop on Analytic Algorithms and Combinatorics*, D. Applegate, G. S. Brodal, D. Panario, and R. Sedgewick, Eds. Society for Industrial and Applied Mathematics, 2007, pp. 135–149.

[63] J.-P. Katoen, T. Kemna, I. Zapreev, and D. Jansen, "Bisimulation minimisation mostly speeds up probabilistic model checking," in *Proceedings of the* 13th *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, O. Grumberg and M. Huth, Eds., vol. 4424. Springer, 2007, pp. 87 – 101.

[64] J. P. Katoen, M. Khattri, and I. S. Zapreev, "A Markov reward model checker," in *The* 2nd *International Conference on Quantitative Evaluation of Systems*. IEEE Computer Society, 2005, pp. 243–245.

[65] J. Kemeny and J. Snell, *Finite Markov Chains*. Springer, 1960.

[66] M. Kuntz and K. Lampka, "Probabilistic methods in state space analysis," in *Validation of Stochastic Systems*, ser. Lecture Notes in Computer Science, C. Baier, B. Haverkort, H. Hermanns, J.-P. Katoen, and M. Siegle, Eds. Springer Berlin / Heidelberg, 2004, vol. 2925, pp. 251–266.

[67] M. Kwiatkowska, G. Norman, and D. Parker, "Probabilistic symbolic model checking with PRISM: A hybrid approach," in *Proceedings of the* 8th *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, J.-P. Katoen and P. Stevens, Eds., vol. 2280. Springer, 2002, pp. 52–66.

[68] M. Kwiatkowska, G. Norman, and D. Parker, "Symmetry reduction for probabilistic model checking," in *Proceedings of the* 18th *International Conference on Computer Aided Verification*, ser. Lecture Notes in Computer Science, T. Ball and R. Jones, Eds., vol. 4114. IEEE Computer Society Press, 2006, pp. 234 – 248.

[69] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *Proceedings of the* 23rd *International Conference on Computer Aided Verification*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 585–591.

[70] L. Lamport, ""Sometime" is sometimes not "never": On the temporal logic of programs," in *Proceedings of the* 7th *ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, ser. POPL '80. ACM, 1980, pp. 174–185.

[71] D. Lehmann and M. Rabin, "On the advantage of free choice: A symmetric and fully distributed solution to the dining philosophers problem (extended abstract)," in *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages*, 1981, pp. 133–138.

[72] D. Luenberger and Y. Ye, *Linear and Nonlinear Programming*. Springer Verlag, 2008, vol. 116.

[73] R. Lyndon and P. Schupp, *Combinatorial Group Theory*. Springer Verlag, 1977, vol. 89.

[74] B. McKay, "nauty Users Guide (version 2.4)," *Computer Science Department, Australian National University*, 2007.

[75] K. L. McMillan, *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[76] P. Merloz, J. Tonetti, L. Pittet, M. Coulomb, S. Lavallee, J. Troccaz, P. Cinquin, and P. Sautot, "Computer-assisted spine surgery," *Computer Aided Surgery*, vol. 3, no. 6, pp. 297–305, 1998.

[77] A. Miller and A. Donaldson, "Property preservation in quotient structures," Technical Report, Department of Computing Science, University of Glasgow, Tech. Rep., 2009.

[78] A. Miller, A. Donaldson, and M. Calder, "Symmetry in temporal logic model checking," *ACM Computing Surveys*, vol. 38, no. 3, 2006.

[79] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[80] F. Neuman and J. Foster, *Investigation of a Digital Automatic Aircraft Landing System in Turbulence*. United States. National Aeronautics and Space Administration and Ames Research Center, 1970.

[81] C. Norris Ip and D. Dill, "Better verification through symmetry," *Formal Methods in System Design*, vol. 9, no. 1, pp. 41–75, 1996.

[82] D. Parker, "Implementation of symbolic model checking for probabilistic systems," Ph.D. dissertation, University of Birmingham, 2002.

[83] D. Parnas, G. Asmis, and J. Madey, "Assessment of safety-critical software in nuclear power plants," *Nuclear Safety*, vol. 32, no. 2, pp. 189–198, 1991.

[84] T. J. Parr and R. W. Quong, "ANTLR: A predicated-LL(k) parser generator," *Software Practice and Experience*, vol. 25, pp. 789–810, 1994.

[85] D. Peled, "Combining partial order reductions with on-the-fly model checking," in *Computer Aided Verification*. Springer, 1994, pp. 377–390.

[86] A. Pnueli and L. Zuck, "Verification of multiprocess probabilistic protocols," *Distributed Computing*, vol. 1, no. 1, pp. 53–72, 1986.

[87] A. Pnueli, "The temporal logic of pograms," in *Proceedings of the* 18^{th} *Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, 1977, pp. 46–57.

[88] C. Power, *Probabilistic Symmetry Reduction*. [Online]. Available: https://bitbucket.org/powerc/pss

[89] M. Rabin, "N-process mutual exclusion with bounded waiting by $4 \log_2 \mathrm{N}$-valued shared variable," *Journal of Computer and System Sciences*, vol. 25, no. 1, pp. 66–75, 1982.

[90] F. Rossi, P. Van Beek, and T. Walsh, *Handbook of Constraint Programming*. Elsevier Science, 2006, vol. 35.

[91] A. P. Sistla, V. Gyuris, and E. A. Emerson, "SMC: A symmetry-based model checker for verification of safety and liveness properties," *ACM Transactions on Software Engineering and Methodology*, vol. 9, no. 2, pp. 133–166, 2000.

[92] J. Tretmans, K. Wijbrans, and M. Chaudron, "Software engineering with formal methods: The development of a storm surge barrier control system revisiting seven myths of formal methods," *Formal Methods in System Design*, vol. 19, pp. 195–215, 2001.

[93] R. S. Varga, *Matrix Iterative Analysis*. Springer, 2000, vol. 27.

[94] T. Wahl, "Adaptive symmetry reduction," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, W. Damm and H. Hermanns, Eds. Springer, 2007, vol. 4590, pp. 393–405.

[95] J. Wilson, "Finding direct product decompositions in polynomial time," *Arxiv preprint arXiv:1005.0548*, 2010.