UNIVERSITY
of
GLASGOW

Automatic Techniques for Detecting and Exploiting Symmetry in Model Checking

Alastair F. Donaldson

April 5, 2007

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy at the University of Glasgow

Department of Computing Science
University of Glasgow

Abstract

Model checking is an increasingly popular technique for the formal verification of concurrent systems. The application of model checking is limited due to the *state-space explosion problem* – as the number of components represented by a model increases, the worst case size of the associated state-space grows exponentially. As such, models of realistic systems are often too large to feasibly check. Over the last 15 years, symmetry reduction techniques for model checking have been developed and, in a restricted setting, have been shown to be effective in reducing the state-space explosion problem. Current techniques can handle limited kinds of symmetry, e.g. full symmetry between identical components in a concurrent system. They avoid the problem of automatic symmetry detection by requiring the user to specify the presence of symmetry in a model (explicitly, or by annotating the associated specification using additional language keywords), or by restricting the input language of a model checker so that *only* symmetric systems can be specified. Additionally, computing unique representatives for each symmetric equivalence class is easy for these limited kinds of symmetry.

We present a theoretical framework for symmetry reduction which can be applied to explicit state model checking. The framework includes techniques for *automatic* symmetry detection using computational group theory, which can be applied with no additional user input. These techniques detect structural symmetries induced by the topology of a concurrent system, so our framework includes exact and approximate techniques to efficiently exploit *arbitrary* symmetry groups which may arise in this way. These techniques are also based on computational group theoretic methods.

We prove that our framework is logically sound, and demonstrate its general applicability to explicit state model checking. By providing a new symmetry reduction package for the SPIN model checker, we demonstrate that our framework can be feasibly implemented as part of a system which is widely used in both industry and academia. Through a study of SPIN users, we assess the usability of our automatic symmetry detection techniques in practice.

Contents

List of Figures

Chapter 1

Introduction

Over the last 25 years, temporal logic model checking [32, 26, 128, 132, 143] has become one of the most popular techniques for formal verification of concurrent hardware and software systems. Given a finite-state model which captures the essential behaviour of a concurrent system, and a temporal logic property which describes some requirement of the system, a model checking algorithm determines whether or not the property holds in the initial state (or states) of the model. Furthermore, if the property does *not* hold, the model checker outputs a *counter-example* – a behaviour of the model which violates the given property. Model checkers can therefore be used to automatically find subtle defects in complex concurrent system designs, or to prove the absence of certain defects, increasing confidence in the system. The fact that model checking is, in principle, a fully automated technique makes it more appealing to designers than other formal methods such as development by specification and refinement, or mechanical theorem proving.

Although model checking has proved successful in both industry and academia, the technique is hindered by the *state-space explosion problem*. This is where, in the worst case, the number of reachable states of a model grows exponentially with the number of components of the system being modelled. Consider a system comprised of n identical components, each of which occupies one of k local states, for some $n, k > 0$. A state of a model of this system can be viewed as a tuple $(l_1, l_2, \ldots, l_n)$, where $l_i \in \{1, 2, \ldots, k\}, (1 \leq i \leq n)$. Thus there are k^n potential states in the model. Although in practice it is unusual for *every* state to be reachable, it is typical for the number of reachable states to approach this upper limit. This means that memory and time constraints often prohibit model checking properties of systems with many components.

A lot of model checking research concentrates on approaches to reduce the state-space explosion problem. Techniques such as symbolic model checking [18, 126], partial order reduction [67, 135], abstraction [30] and symmetry reduction [14, 27, 31, 55, 102] have been successfully used in the verification of large systems.

Symmetry reduction is applicable when a system contains replicated components. Such replication, or *symmetry*, can result in portions of the state-space of a model of the system being *equivalent* up to rearrangement of component identifiers. If symmetry is known to be present in a specification then model checking of certain properties can be performed over a *quotient* model, which is generally smaller than the unreduced model. The quotient model is usually constructed by converting each state encountered during search to a unique representative of its symmetric equivalence class. There are two main problems which must be overcome for a symmetry reduction technique to be useful: it must be possible to derive symmetries of a model from its associated high-level specification, and an efficient method of computing equivalence class representatives must be available.

Existing techniques for exploiting symmetry in model checking assume that symmetries of a model are either known *a priori* [31], coded into the model through the use of special keywords [14, 102], or guaranteed to exist by restricting the input language so that there is full symmetry between multiple instances of a parameterised component [163]. The first two approaches are potentially prone to error, and compromise the automation of model checking, which is one of its main strengths as a verification technique. With the third approach, the specification language is designed to suit one particular state-space reduction technique, which may restrict the style of specifications, and typically only full symmetry between identical components can be captured in this way. Ideally, a model checking tool should be able to detect symmetry automatically from a high level system description.

The problem of computing equivalence class representatives is usually avoided by only providing support for full symmetry, since in this special case representatives can be efficiently computed using techniques based on sorting. However, many other kinds of symmetry commonly occur in models of concurrent systems with a regular structure. For example, cyclic/dihedral groups are typically associated with systems which have uni/bi-directional ring structures, and wreath product groups occur when dealing with tree topologies. Efficient strategies for representative computation have been proposed for symmetry groups which are known to have certain structural properties [27]. However, an *automated* solution to the problem of classifying the structure of *any* group so that an appropriate strategy can be chosen is required.

1.1 Contribution and Structure of the Thesis

We provide a review of model checking and symmetry reduction literature in Chapters 2 and 3 respectively. In Chapter 4 we present a selection of examples for which symmetry detection and/or reduction using existing techniques is either difficult, or impossible. The rest of the thesis is divided into two parts, which respectively addresses research problems in automatic symmetry detection, and efficient exploitation of symmetry.

Chapters 5–8 are concerned with techniques for automatic symmetry detection. Examples from Chapter 4 are used in Chapter 5 to highlight a correspondence between symmetries of the communication structure and symmetries of the model associated with a specification. We develop automated symmetry detection techniques for message passing specification languages in Chapter 7, using a small language which captures the essential features of the widely used Promela language. The approach involves computing the symmetry group of the *static channel diagram* of a specification (a graphical representation of potential communication in the underlying model), and using a computational group theoretic algorithm to compute a subgroup of these symmetries which induces automorphisms of the underlying model. In Chapter 8 we describe *SymmExtractor*, an implementation of these techniques for Promela, using the computational group theoretic package GAP. We evaluate the usability of SymmExtractor using a set of Promela specifications written as solutions to two student assessed exercises.

The problem of efficiently exploiting symmetries during model checking is addressed in Chapters 9 – 11. In Chapter 9 we extend existing results on efficiently computing equivalence class representatives for certain kinds of symmetry group under a simple model of computation, and present a computational group theoretic approach to classifying an arbitrary symmetry group so that an appropriate symmetry reduction strategy can be chosen. Given a set of group generators, the classification algorithm analyses the structure of the group, identifying it as a wreath or disjoint product of subgroups (which are in turn analysed), or as a basic symmetry group. For certain kinds of basic symmetry groups, exact, efficient symmetry reduction strategies are available. Otherwise we propose an approximate strategy based on hillclimbing local search. This strategy does not provide optimal reduction, but is sound, as well as being fast in practice. For symmetry groups which decompose as a product of basic groups, a composite symmetry reduction strategy is selected. In Chapter 10 we then consider a more realistic model of computation, and show that exact symmetry reduction strategies under the simple model of computation are no longer guaranteed to provide optimal reduction. We show how to extend these strategies to achieve optimality, at the expense of polynomial time. In Chapter 11 we describe TopSPIN, a symmetry reduction package for the SPIN model checker, which incorporates our (detection and reduction) techniques. We show significant reductions in verification time and space requirements for model checking safety properties for a variety of examples.

1.2 Thesis Website and Source Forge

The results in this thesis are illustrated using a variety of specifications of various concurrent systems. Some of these are given in Appendix A, but all are available online at the following URL:

`http://www.dcs.gla.ac.uk/people/personal/ally/thesis/`

Release distributions of the three software tools presented in the thesis, SPIN-to-GRAPE, SymmExtractor and TopSPIN, can also be downloaded from the above URL. The tools are open source and their source code can be downloaded from Source Forge:

`https://sourceforge.net/projects/symmetryglasgow/`

1.3 Notation for Equality and Assignment

Throughout the thesis we make extensive use of the Promela specification language. Promela follows the C convention of using == to denote the boolean equality operator and = assignment. For example, `x==5` is a boolean expression which evaluates to *true* iff x has the value 5. On the other hand, `x=5` is a statement which assigns x to the value 5.

When writing mathematical equations and presenting algorithms, we prefer to use = to denote the equality operator, and := to denote assignment (the approach used by languages such as Ada and Pascal). Therefore the meaning of == and := is unambiguous, but the meaning of = depends on whether it occurs in a Promela (or Promela-Lite) code fragment. The SMC language, discussed in Section 3.3.3, uses = and == in the same way as Promela.

1.4 Acknowledgment of Published Work

Much of the original material in this thesis has been published by the author in a selection of co-authored papers.

The survey of symmetry reduction techniques presented in Chapter 3 appears in [130]; the SPIN-to-GRAPE tool of Chapter 4 was first presented in [49]. The automatic symmetry detection techniques of Chapter 7 are published (in a preliminary form) in [48] and [42], in which the SymmExtractor tool is also introduced. The type reconstruction algorithm used by SymmExtractor (see Section 8.2) was developed as part of the ETCH type checker [41]. Chapters 9 and 10 introduce strategies for symmetry reduction which have been published in [44] and [46] respectively, while the TopSPIN symmetry reduction package is described in [43].

However, the content of this thesis is the work of the author, incorporating supervisory suggestions.

Published work not included in the thesis

We have published three papers related to symmetry reduction in model checking, the content of which are not included here. The topics covered by these papers are:

a symmetry reduction technique specific to featured networks [129], an approach to symmetry reduction for probabilistic, symbolic model checking [45], and a comparison of techniques for exploiting symmetry in model checking and constraint programming [47].

While these papers address interesting problems related to the role of symmetry reduction in formal verification, they do not fit into the suite of automatic, general techniques for exploiting symmetry which we present here.

Chapter 2

Model Checking and the State Space Explosion Problem

In this chapter we formally present temporal logic model checking, introducing the Kripke structure formalism used to model a concurrent system, together with the logic *CTL** and its commonly used sub-logics, *CTL* and *LTL*. We give an overview of some standard model checking algorithms and tools. In particular, we describe the Promela specification language and its bespoke model checker, SPIN, which are referred to frequently in Chapters 4–11. The chapter concludes with a discussion of techniques which have been developed to combat the state-space explosion problem.

We begin by describing the use of model checking in the development of reliable concurrent systems.

2.1 The Model Checking Process

Verification of a concurrent system design by temporal logic model checking traditionally involves first specifying the behaviour of the system at an appropriate level abstraction. The specification $\mathcal{P}$ is described using a high level formalism (often similar to a programming language), the semantics of which are an associated *finite state* model, $\mathcal{M}(\mathcal{P})$. A requirement of the system is specified as a temporal logic property, ϕ.

A software tool called a *model checker* then exhaustively searches the finite state model $\mathcal{M}(\mathcal{P})$, checking whether ϕ holds at each initial state. If ϕ does not hold at some initial state, an error trace or *counter-example* is reported. Manual examination of this counter-example by the system designer can reveal that $\mathcal{P}$ does not adequately specify the behaviour of the system, that ϕ does not accurately describe the given requirement, or that there is an error (bug) in the design. In this case, either $\mathcal{P}$, ϕ, or the system design (and thus also $\mathcal{P}$ and possibly ϕ) must be modified, and re-checked. This process is repeated until the model checker reports that ϕ holds in every initial state of $\mathcal{M}(\mathcal{P})$, in which case we say $\mathcal{M}(\mathcal{P})$ satisfies ϕ, written $\mathcal{M}(\mathcal{P}) \models \phi$. The model checking process is illustrated by Figure 2.1.

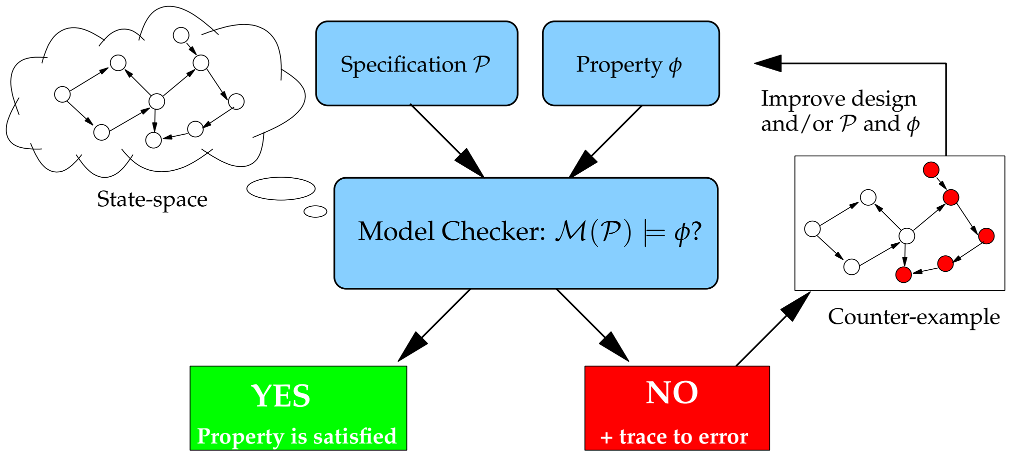


Figure 2.1: The model checking process.

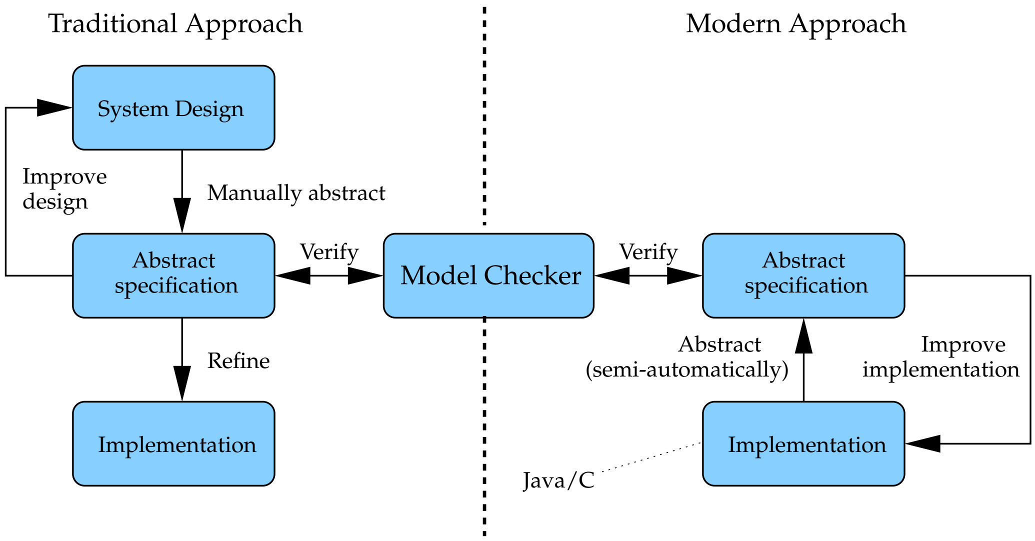


Figure 2.2: Traditional and modern approaches to model checking in the development of systems (adapted from [154]).

Assuming that the specification and temporal properties have been constructed with care, successful verification by model checking increases confidence in the system design, which can then be refined towards an implementation. This traditional approach is illustrated in the left hand side of Figure 2.2.

In practice, software is often developed rapidly, without much initial testing or verification. In this case there is a need to apply model checking techniques to the source code of an existing system, in an attempt to correct logical design flaws. Semi-automatic abstraction techniques are used to extract a specification and logical properties from source code so that the model checking process can be applied. This modern approach is illustrated on the right hand side of Figure 2.2.

2.2 Kripke Structures and Temporal Logic

As discussed above, the model checking problem involves determining whether or not a finite state model describing the behaviour of a concurrent system satisfies a temporal logic formula specifying a desired safety or liveness property of the system. A *Kripke structure* is the common formalism for representing a finite state model, and temporal logic formulas are usually expressed in (a sub-logic of) $CTL^\star$, or the μ-calculus.

Let $V = \{v_1, v_2, \ldots, v_k\}$ be a finite set of system variables, where each v_i ranges over a finite non-empty set D_i of possible values. Then $D = D_1 \times D_2 \times \cdots \times D_k$ is the set of all possible system states. A Kripke structure is defined in terms of D as follows:

Definition 1 *A Kripke structure $\mathcal{M}$ over D is a tuple $\mathcal{M} = (S, S_0, R)$ where:*

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

A path in $\mathcal{M}$ from a state $s \in S$ is an infinite sequence of states $\pi = s_0, s_1, s_2, \ldots$ where $s_0 = s$, such that for all $i > 0$, $(s_{i-1}, s_i) \in R$. For states s and t, it is common to denote the transition (s, t) by $s \rightarrow t$. A state $s \in S$ is *reachable* if there is a path $s_0, s_1, \ldots, s, \ldots$ in $\mathcal{M}$ where $s_0 \in S_0$. A transition $(s, t) \in R$ is *reachable* if s is a reachable state.

We usually deal with Kripke structures which have a single initial state $s_0 \in S$, in which case we write $\mathcal{M} = (S, s_0, R)$.[1]

Figure 2.3 shows the reachable part of a Kripke structure for a model of two process mutual exclusion. The model consists of two processes, each with three local states N, T and C. Each process has a single state variable, st_i say ($i \in \{1, 2\}$). Here $V = \{st_1, st_2\}$ and $D_1 = D_2 = \{N, T, C\}$. The values N, T and C denote that a process is in the *neutral*, *trying* or *critical* state respectively. For $A \in \{N, T, C\}$ we abbreviate the proposition $st_i = A$ by A_i. Only if process i is in the trying state (i.e. T_i holds) and process $j \neq i$ is *not* in the critical state (i.e. $\neg C_j$ holds) can process i can move into the critical state. Thus in the model it is not possible for both processes to be in the critical state. That is, the mutual exclusion property holds. Note that there is a single initial state (indicated by an incoming edge with no predecessor state in Figure 2.3). In the initial state both processes are in the neutral state.

1. Following the convention of e.g. [30, 55, 57, 59], Definition 1 does not include a labelling function. Such a structure is sometimes referred to simply as a *transition system* [30]. We could equivalently define states as being labelled with atomic propositions of the form $(v_i = d_i)$ (where $d_i \in D_i$) [32]. However, the above notation in which states *are* valuations of variables (and thus are implicitly labelled) is convenient for presentation of our results, and is close to the representation of states used by explicit-state model checkers.

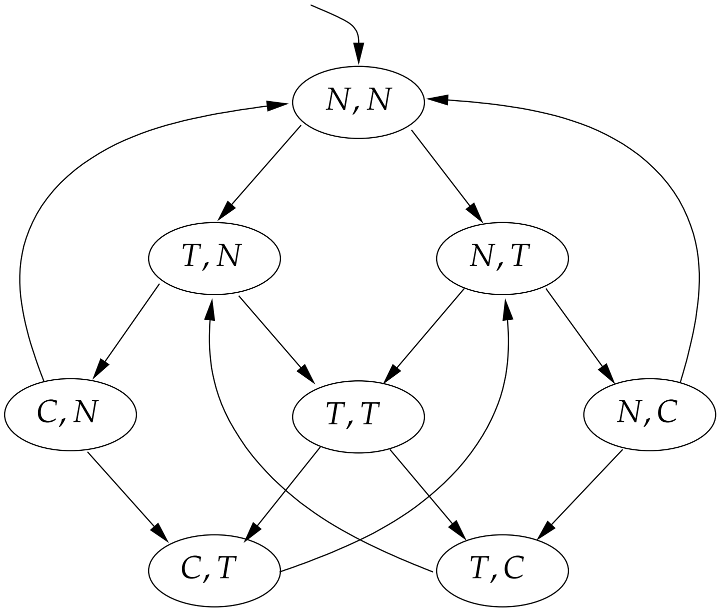


Figure 2.3: Kripke structure for two-process mutual exclusion.

2.2.1 *CTL**

To express properties of Kripke structures we introduce the branching time temporal logic *CTL**. The set of *CTL** state and path formulas are defined inductively over a finite set of propositions over system variables. The quantifiers **A** and **E** are used to denote *for all paths*, and *for some path* respectively (where $\mathbf{E}\phi = \neg\mathbf{A}\neg\phi$). In addition, **X**, **U**, **F** and **G** represent the standard *next-time*, *strong until* (see e.g. [92]), *eventually* and *always* operators (where $\mathbf{E}\phi = \mathit{true}\mathbf{U}\phi$, and $\mathbf{G}\phi = \neg\mathbf{F}\neg\phi$ respectively). Note that we use $p \Rightarrow q$ to denote $\neg p \vee q$ in the standard way. Let V and D_i, ($1 \leq i \leq k$) be as above. Then:

- *true*, *false*, $(v_i = d_i)$ and $(v_i \neq d_i)$ (for all $v_i \in V$, $d_i \in D_i$) are state formulas
- if ϕ and ψ are state formulas, then so are $\neg\phi$, $\phi \wedge \psi$ and $\phi \vee \psi$
- if ϕ is a path formula, then $\mathbf{A}\phi$ and $\mathbf{E}\phi$ are state formulas
- any state formula ϕ is also a path formula
- if ϕ and ψ are path formulas, then so are $\neg\phi$, $\phi \wedge \psi$ and $\phi \vee \psi$, $\mathbf{X}\phi$, $\phi\mathbf{U}\psi$, $\mathbf{F}\phi$ and $\mathbf{G}\phi$.

Given (path or state) formulas ϕ and ψ, ψ is a *sub-formula* of ϕ, written $\psi \subseteq \phi$, if either $\psi = \phi$, ψ is an operand to one of the operators appearing in ϕ, or ψ is bound to a quantifier appearing in ψ. The sub-formula ψ is *propositional* if it is a state formula which does not include **A** or **E**. A *maximal* propositional sub-formula of ϕ is a propositional sub-formula ψ such that if $\psi \subseteq \psi' \subseteq \phi$, where ϕ' is also a propositional sub-formula, then $\psi = \psi'$.

The logic *CTL** is the set of all *state* formulas. For a Kripke structure $\mathcal{M}$, if the *CTL** formula ϕ holds at a state $s \in S$ then we write $\mathcal{M}, s \models \phi$ (or simply

$s \models \phi$ when the identity of the model is clear from the context). Otherwise we write $\mathcal{M}, s \not\models s$. The relation $\models$ is defined inductively below. Note that for a path $\pi = s_0, s_1, \ldots$ we define $\mathit{first}(\pi) = s_0$ and, for all $i \geq 0$, π_i is the suffix of π starting from state s_i.

- $s \models \mathit{true}$, and $s \not\models \mathit{false}$
- $s \models (v_i = d_i)$ if and only if $s = (e_1, e_2, \ldots, e_k)$ and $e_i = d_i$
- $s \models (v_i \neq d_i)$ if and only if $s = (e_1, e_2, \ldots, e_k)$ and $e_i \neq d_i$
- $s \models \neg\phi$ if and only if $s \not\models \phi$
- $s \models \phi \wedge \psi$ if and only if $s \models \phi$ and $s \models \psi$
- $s \models \phi \vee \psi$ if and only if $s \models \phi$ or $s \models \psi$
- $s \models \mathbf{A}\phi$ if and only if $\pi \models \phi$ for every path π starting at s
- $s \models \mathbf{E}\phi$ if and only if $\pi \models \phi$ for some path π starting at s
- $\pi \models \phi$, for any state formula ϕ, if and only if $\mathit{first}(\pi) \models \phi$
- $\pi \models \neg\phi$ if and only if $\pi \not\models \phi$
- $\pi \models \phi \wedge \psi$ if and only if $\pi \models \phi$ and $\phi \models \psi$
- $\pi \models \phi \vee \psi$ if and only if $\pi \models \phi$ or $\pi \models \psi$
- $\pi \models \phi\mathbf{U}\psi$ if and only if, for some $i \geq 0$, $\pi_i \models \psi$ and $\pi_j \models \phi$ for all $0 \leq j < i$
- $\pi \models \mathbf{X}\phi$ if and only if $\pi_1 \models \phi$
- $\pi \models \mathbf{F}\phi$ if and only if $\pi_i \models \phi$, for some $i \geq 0$
- $\pi \models \mathbf{G}\phi$ if and only if $\pi_i \models \phi$, for all $i \geq 0$.

Model checking involves determining the satisfaction of a temporal logic formula by a Kripke structure. The *model checking problem* can be specified *globally* or *locally* as follows [132]:

Global model checking problem – Given a Kripke structure $\mathcal{M}$ and a $CTL^\star$ formula ϕ, determine the set of states in $\mathcal{M}$ that satisfy ϕ (i.e. determine $\{s \in S : \mathcal{M}, s \models \phi\}$).

Local model checking problem – Given a Kripke structure $\mathcal{M}$, a $CTL^\star$ formula ϕ and a state s in $\mathcal{M}$, determine whether s satisfies ϕ (i.e. $\mathcal{M}, s \models \phi$).

Recall the set S_0 of initial states of a Kripke structure $\mathcal{M}$. In practice we are typically interested in whether the initial states of a model satisfy a given property, so we say that the model $\mathcal{M}$ satisfies the $CTL^\star$ property ϕ, denoted $\mathcal{M} \models \phi$, if $\mathcal{M}, s \models \phi$ for all $s \in S_0$.

Returning to the mutual exclusion example of Figure 2.3, we can express the mutual exclusion property formally in $CTL^\star$ as follows:

Property 1 $\mathbf{AG}(\neg(C_1 \wedge C_2))$.

The Kripke structure clearly satisfies this property as (C, C) is not a reachable state. Property 1 is a *safety* property – it asserts that something (bad) never happens. A *liveness* property on the other hand expresses that eventually something (good) must happen during an execution. For example, Property 2 below states that, once reaching its *trying* region, a process will eventually progress to its critical section (the *progress* property):

Property 2 $\mathbf{AG}(T_1 \Rightarrow (\mathbf{F}C_1))$.

To see that the Kripke structure does *not* satisfy this property, consider the infinite path starting at (N, N), followed repeatedly by the cycle $(T, N), (T, T), (T, C), (T, N)$. Process 1 waits in the trying region forever along this infinite path, violating Property 2. Thus this path is a *counter-example* which proves that $\mathcal{M}, (N_1N_1) \not\models$ Property 2.

We now define two sub-logics of $CTL^\star$ which are commonly used in applications of model checking.

CTL

The logic *CTL* (Computation Tree Logic) is the sub-logic of $CTL^\star$ in which the temporal operators **X**, **U**, **F** and **G** must be immediately preceded by a path quantifier. For example the so-called *reset* property, $\mathbf{AG}(\mathbf{EF}\ \mathit{Restart})$, which asserts that from any state it is possible to get to the *Restart* state, is a *CTL* property. Efficient model checking algorithms exist for this sub-logic (see Section 2.3.1), which is expressive enough for the needs of most hardware verification problems, and thus is used almost exclusively in this area.

LTL

The logic *LTL* (Linear Temporal Logic) is obtained by restricting the set of ($CTL^\star$) formulas to those of the form $\mathbf{A}\phi$, where ϕ does not contain **A** or **E**. It cannot express e.g. the *reset* property (see above). On the other hand, the property $\mathbf{A}(\mathbf{FG}\ \mathit{Leader})$, which states that eventually the proposition *Leader* will hold forever, can be expressed in *LTL* but not *CTL*. Although the model checking problem for *LTL* is *NP*-hard [32], *LTL* model checking can be performed *on-the-fly* using an automata-theoretic approach (see Section 2.3.2) which can be very efficient in practice. *LTL* is applied almost exclusively in *software* verification.

Figure 2.4 illustrates the relationship between *CTL*, *LTL* and $CTL^\star$. Example properties (adapted from [32]) in $CTL \cap LTL$, $CTL \setminus LTL$, $LTL \setminus CTL$ and $CTL^\star \setminus (CTL \cup LTL)$ are shown. For a debate on the relative benefits of *CTL* vs. *LTL* see [92].

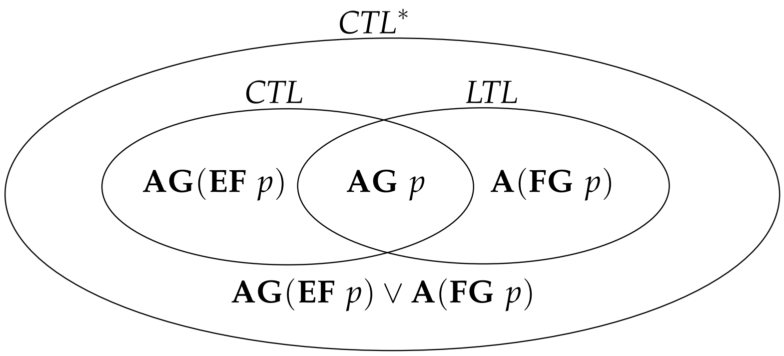


Figure 2.4: Relationship between the temporal logic $CTL^\star$ and its sub-logics CTL and LTL, with example properties.

2.2.2 μ-calculus

It is worth noting that properties of transition systems can also be expressed in the propositional μ-calculus [110]. This powerful language is obtained by extending Hennessy-Milner logic (a simple modal logic) [79] with fixpoint operators. The μ-calculus is of interest to researchers in formal verification as many temporal logics (e.g. $CTL^\star$) can be encoded into it.

Although symmetry reduction techniques have been shown to be compatible with μ-calculus model checking [55], we restrict our attention to $CTL^\star$ and its sub-logics, which are expressive enough to describe most properties of interest, and are supported by widely used model checkers such as SPIN and SMV.

2.3 Model Checking Algorithms

We now describe standard explicit-state model checking algorithms for CTL and LTL, and indicate how they can be combined for $CTL^\star$ model checking.

2.3.1 *CTL* model checking

The model checking algorithm for CTL [28, 143] works by successively marking states which satisfy sub-formulas of the formula to be checked, starting with propositional sub-formulas which are trivial to check. The particular form of the algorithm used depends on the formula. For illustration, we give here an example of how the algorithm proceeds to check formula ϕ, where ϕ is $\mathbf{A}(\phi_1 \mathbf{U} \phi_2)$.

For a state s, $s \models \phi$ if and only if either s satisfies ϕ_2 or s has at least one successor, s satisfies ϕ_1 and all successors of s satisfy ϕ. Initially all states are marked to indicate whether they satisfy ϕ_1 and/or ϕ_2. States which satisfy ϕ_2 can immediately be marked as satisfying ϕ. Each state is also marked with a number (*nb* say), denoting how many successors have yet to be marked as satisfying ϕ. Initially for each state s, *nb* is set to 0 if $s \models \phi$, or to the number of successors of s otherwise. In the latter case, each time a successor of s is marked as satisfying ϕ, *nb* is decremented by one. When $nb = 0$ for s, clearly $s \models \phi$. When no states can be remarked, the algorithm terminates. If, at this point, all initial states are marked as satisfying ϕ, then $\mathcal{M} \models \phi$.

The algorithm for determining whether a *CTL* formula ϕ holds in a state s of $\mathcal{M}$ is linear in the size of the formula and the Kripke structure – the complexity is $O(|\phi| \cdot (|S| + |R|))$, where $|\phi|$ is the length of ϕ [28]. An extension of the algorithm which only considers fair computations (see Section 3.6.2) is presented in [28].

2.3.2 Automata-theoretic *LTL* model checking

The model checking problem for *LTL* can be restated as: "given $\mathcal{M}$ and ϕ, does there exist a path of $\mathcal{M}$ that does not satisfy ϕ?" One approach to *LTL* model checking is the tableau approach described in [132]. However, we concentrate here on the more efficient automata-theoretic approach [118, 173].

Definition 2 *A finite state automaton (FSA) $\mathcal{A}$ is a tuple $\mathcal{A} = (S, s_0, L, T, F)$ where:*

1. *S is a non-empty, finite set of states*
2. *$s_0 \in S$ is an initial state*
3. *L is a finite set of labels*
4. *$T \subseteq S \times L \times S$ is a set of transitions*
5. *$F \subseteq S$ is a set of final states.*

A run of $\mathcal{A}$ is an ordered, possibly infinite, sequence of transitions

$$(s_0, l_0, s_1), (s_1, l_1, s_2), \ldots$$

where $s_i \in S$ and $l_i \in L$ for all $i > 0$. An accepting run of $\mathcal{A}$ is a finite run in which the final transition (s_{n-1}, l_{n-1}, s_n) has the property that $s_n \in F$.

In order to reason about infinite runs of an automaton, alternative notions of acceptance, e.g. Büchi acceptance, are required. We say that an infinite run (of an FSA) is an accepting ω-run (i.e. it satisfies Büchi acceptance) if and only if some state in F is visited infinitely often in the run. A Büchi automaton is an FSA defined over infinite runs (together with the associated notion of Büchi acceptance).

Every *LTL* formula can be represented as a Büchi automaton (see for example [174], and references therein). In order to verify an *LTL* property $\mathbf{A}\phi$, a model checker must show that all paths of a model $\mathcal{M}$ satisfy ϕ (alternatively, find a counter-example, namely a path which *does not* satisfy ϕ). To do this, an automaton $\mathcal{A}$ representing the reachable states of $\mathcal{M}$ is constructed, together with an automaton $\mathcal{B}_{\neg\phi}$ which accepts all paths for which $\neg\phi$ holds. The asynchronous product of the two automata, $\mathcal{A}'$ is constructed. (In practice $\mathcal{A}'$ is usually constructed implicitly, by letting $\mathcal{A}$ and $\mathcal{B}_{\neg\phi}$ take alternate steps). Any accepting run of $\mathcal{A}'$ signifies an error. If there are no accepting runs, $\mathcal{M} \models \phi$. Generally to prove *LTL* properties, a depth-first search is used. As the search progresses, all states visited are stored (in a reduced form) in a hash array (or heap), and states along the current path are pushed on to the stack.

If the property ϕ to be verified is a safety property, say $\phi = \mathbf{AG}\ \psi$, where ψ does not contain the until operator $\mathbf{U}$, then a depth-first search of $\mathcal{A}'$ is used. If a state is encountered at which ψ is false, then ϕ is false and the current path (the current contents of the stack) provides a counter-example. If, on the other hand, ϕ is a liveness property, then determining the truth, or otherwise, of ϕ relies on the ability to detect the presence of infinite accepting runs in $\mathcal{A}'$. This is achieved either by using the classic approach of Tarjan [169] in which the strongly connected components are constructed and analysed separately for acceptance runs, or via a nested depth-first search [35]. A nested depth-first search is more efficient than the classic approach in that it is not necessary to produce all acceptance runs, just a single acceptance cycle (if one exists). Suppose, for example ϕ is $\mathbf{A}(\mathbf{GF}\ p)$, for some proposition p. From any state s reached during an initial search at which $\neg p$ holds, a second search is initiated to check for paths leading back to s, during which p remains false. If no such path exists, the original search resumes from s.

The complexity of *LTL* model checking is exponential in the length of the formula to be checked: $O((|S| + |R|) \cdot 2^{O(|\phi|)})$. This is because the worst case automaton generated from an *LTL* formula ϕ may have $2^{|\phi|}$ states. Although in the worst case this means that *LTL* model checking is much harder than *CTL* model checking, in most practical cases there is little performance difference [92, Appendix B].

2.3.3 Model checking for *CTL**

Model checking for *CTL** was first introduced in [28]. A method for checking *CTL** properties [54] involves the use of an *LTL* model checker on the sub-formulas of the property to be checked. The complexity of *CTL** model checking is the same as for *LTL* model checking. However, due to the automata-theoretic approach for *LTL* model checking and the efficient *CTL* model checking algorithm, most model checkers are used to verify either *CTL* or *LTL* properties, but not both.

2.4 Promela and SPIN

Clearly it would be impractical to model complex concurrent systems directly as Kripke structures. In practice, a system is described using a high-level specification formalism which has Kripke structure semantics. A model checking tool takes as input a specification of a concurrent system, together with a property in some temporal logic. Using algorithms such as those outlined in Section 2.3, together with appropriate state-space reduction techniques (see Section 2.6), the tool checks whether or not the associated model satisfies the property, providing a counter-example if the result is negative. Certain properties (such as absence of deadlock, or basic safety properties which can be expressed using specification-level assertions) can be checked without a temporal property.

The model checker SPIN (simple Promela interpreter) allows *LTL* reasoning about specifications written in Promela (process meta language). SPIN has been widely used in industry and academia for reasoning about communications protocols. In this section we give an overview of Promela and SPIN, which are used for implementation and examples throughout Chapters 4–11. For an excellent Promela language reference, see [65]. Full details of SPIN and Promela can be found in the SPIN reference manual [92]. In Section 2.5 we briefly describe a selection of other model checking tools.

2.4.1 Promela

Promela is an imperative style specification language geared towards the description of network protocols. In general, a Promela specification consists of a series of global variables, channel declarations and process type (proctype) declarations, together with an initialisation process. Desired logical properties of a specification are either presented using assertions embedded in the body of a proctype, or via a *never claim* – a special additional process which can be used for the verification of *LTL* properties.[2]

Each proctype in a Promela specification can be viewed as a finite automaton (see Section 2.3.2), and the model associated with this specification is the asynchronous product of the automata for all proctype instantiations. This global automaton can be viewed as a Kripke structure, so we talk about the Kripke structure, rather than the automaton, associated with a Promela specification.

Variables and channels

Promela includes the following primitive data types: *bit*, *byte*, *short* and *int* (numeric types); *pid* (a type for storing process identifier values), and *bool* (for boolean values). Names for messages in a protocol can be defined using a single enumeration, called *mtype*. For example, the declaration:

```
mtype = {request,ack,grant,deny}
```

defines four distinct message names for use in a protocol. User-defined record types can be constructed using the `typedef` keyword. The declaration

```
typedef message { pid sender; pid receiver; mtype body;
bit encrypted }
```

defines a record type, *message*, with four fields: *sender* and *receiver* (which have type *pid*), *body* (an enumeration), and *encrypted* (a *bit*). Single-dimensional arrays can be

2. Temporal properties can also be expressed using *progress* and *accept* labels in the body of a proctype. We do not discuss these here.

declared using C-like syntax. For example:

```
message A[5]
```

defines an array of length five, with element type `message`. Two-dimensional arrays can be declared indirectly using an array of a record type which includes an array type as one of its fields.

To facilitate the specification of protocols, Promela includes a `chan` data type to describe both synchronous and buffered channels. A channel declaration can have one of three forms. A declaration `chan` $\langle name \rangle$ `=` `[`x`]` `of` `{`$\langle type \rangle_1$`,` $\langle type \rangle_2$`, ...,` $\langle type \rangle_k$`}` ($x \geq 0$, $k > 0$) defines a *new* channel (referred to by $\langle name \rangle$). Each message to be sent on this channel must be a tuple of values, where the value at position i has type $\langle type \rangle_i$ ($1 \leq i \leq k$). We refer to the elements of this tuple as *message fields*. If $x > 0$ then the declaration defines a buffered, first-in first-out channel of length x. If $x = 0$ then communication on the channel is synchronous. The component of the channel declaration of the form `[`x`]` `of` `{`$\langle type \rangle_1$`,` $\langle type \rangle_2$`, ...,` $\langle type \rangle_k$`}` is called a channel *initialiser*. A channel declaration `chan` $\langle name \rangle_1$ `=` $\langle name \rangle_2$, on the other hand, does not define a new channel. Rather it defines a new channel reference, $\langle name \rangle_1$, which refers to the channel referred to by $\langle name \rangle_2$ (the name associated with a previous channel declaration). Finally, a declaration `chan` $\langle name \rangle$ defines a channel reference which is initially null. A useful feature of Promela is that, like the π-calculus [157], it supports the declaration of first class channels: the type `chan` may be given as a message field type in a channel initialiser, so that channel references can be passed on the channel. This allows for specifications with *dynamic* communication structures.

We say that a channel variable is *globally instantiated* if it is declared in global scope (outwith any proctype definition), and has a channel initialiser.

A (non-channel) global variable declaration may be prefixed by the `hidden` keyword. This indicates to SPIN that the variable is a "scratch" variable, used only for intermediate computation within `atomic` or `d_step` blocks (see below). Accordingly, to save memory, SPIN does not include the values of hidden variables in the data structure used to represent a state of the model associated with a specification. It is the responsibility of the user to ensure that hidden variables are used correctly; SPIN cannot check this automatically. It is particularly convenient to declare global constant data structures (e.g. fixed lookup tables) as `hidden`, so that they are not duplicated in every state of the global state space.

Processes and statements

A Promela *proctype* is a parameterised process definition. A proctype consists of a name, an optional list of parameters and local variable declarations, and an ordered list of statements. Each proctype includes a built-in, read-only variable called `_pid`, which records the identifier of a process (a non-negative integer). In addition, each proctype includes an implicit *program counter* variable, which stores the current position of execution within the proctype body. This variable cannot be explicitly referred to. However, particular positions in the proctype body can be marked using labels, which can then be used for control flow via `goto` statements (as in the C language).

A specification usually includes a designated `init` process which is automatically instantiated at the start of verification, and which may instantiate further processes via `run` statements.[3] A `run` statement consists of a proctype name, and a list of actual parameters for the given proctype. Execution of a `run` statement causes an instance of the given proctype to be added to the pool of running processes. The `init` process is assigned `_pid` value 0 by SPIN, and processes identifiers are thereafter assigned in order of instantiation.

The simple statements in a proctype fall into three categories: expressions, updates, and communication statements. An *expression* is a boolean expression over local and global variables, using the standard equality operators == and `!=`, relational operators <, <=, > and >=, and logical operators `&&`, `||` and `!`. Boolean expressions may also test the state of a buffered channel c using the `len` operator (which returns the length of c); the operators `full`, `empty`, `nfull` and `nempty` which determine whether c is full, empty, not full or not empty respectively,[4], or via a channel poll expression (see [92] for details). Upon reaching an expression statement, a process may not continue execution until the expression evaluates to *true*. When this is the case, execution of the statement has no side-effects. An *update* is a statement of the form $\langle$*variable*$\rangle$=$\langle$*expr*$\rangle$. Such a statement is always executable (as long as the expression does not involve division by zero), and updates the value of the given variable with the result of the expression.

A *communication* statement involves sending on or receiving from a channel. A *send* statement has the form $\langle$*chan*$\rangle$`!`$\langle$*expressions*$\rangle$, where $\langle$*chan*$\rangle$ is a channel variable and $\langle$*expressions*$\rangle$ is a comma-separated list of expressions. The type of each expression must match the type of the corresponding message field of the channel to which the variable refers. A statement of this form is executable either if the channel is buffered and not full, or if the channel is synchronous and there is another process ready to receive on the channel. Sending on a buffered channel has the effect of adding a message to the buffer, and sending on a synchronous channel causes the list of expression values to be written to a corresponding list of variables offered by the receiving process. A *receive* statement has the form $\langle$*chan*$\rangle$`?`$\langle$*variables*$\rangle$, where $\langle$*chan*$\rangle$ is as before, and $\langle$*variables*$\rangle$ is a comma-separated list of distinct variables. A receive statement is executable either if the channel is buffered and not empty, or if the channel is synchronous and there is another process ready to send a message on the channel. Receiving on a buffered channel causes the given list of

3. Processes may also be instantiated using the `active` keyword – see [92] for details.
4. The provision of both `full` and `nfull` (similarly `empty` and `nempty`) is necessary since, for reasons described in [92], it is illegal to write `!full(`c`)`.

variables to be assigned to the associated field values of the next message on the buffer, which is also removed from the buffer. Receiving on a synchronous channel causes the list of variables to be overwritten by the (evaluated) list of expressions offered by the associated sender process.

Our description of communication statements has not covered various features, including: non-destructive channel reading; sorted-send and random-receive operations; the `eval` operator, and the built-in, write-only '`_`' variable. These features are fully documented in the reference manual [92].

Control flow

The most basic control flow operator in Promela is '`;`', which denotes *sequence* (as in most imperative languages). Following languages such as Pascal, '`;`' is intended as a statement *separator* rather than a statement *terminator*, so strictly should *not* appear at the end of a list of statements. However, the SPIN implementation relaxes this condition, and a terminating semi-colon is optional. Any occurrence of '`;`' can be equivalently replaced with the alternative separator '`->`'. However, '`->`' is usually used to express a compound statement of the form *guard* `->` *update*.

To describe a system at an appropriate level of abstraction it is often convenient to specify that a particular sequence of statements should be executed as a single update. This can achieved using a `d_step` (deterministic step) or `atomic` block. A `d_step` block consists of one or more non-blocking, deterministic statements to be executed as a single transition. Examples of blocking statements include channel operations, expression statements (described in detail below), and `run` statements (which may block due to an upper limit of 256 running processes imposed by SPIN). In addition, it is not legal for a `goto` or `break` statement (described below) to potentially cause a jump out of a `d_step` block. An `atomic` block is similar, but it is permissible for statements within an `atomic` block to involve non-deterministic choice, potentially block execution of the process, or cause a jump out of the block. The use of `d_step` over `atomic`, when applicable, results in more efficient use of memory during verification.

Repetitive choice can be specified using a compound statement of the form `do` ⟨*options*⟩ `od`. The ⟨*options*⟩ part of this construct is a list of Promela fragments, separated by the `::` token. A process executes a `do..od` statement by repeatedly executing one of the options, if any are executable. A `break` or `goto` statement may be used to jump out of a `do..od` loop. Non-repetitive choice can be specified similarly using an `if..fi` construct. Examples of `do..od` and `if..fi` are provided in Figure 2.5. The `if..fi` example shows that the guards which determine executability of each option need not be mutually exclusive: if the guard `(x==4)` evaluates to true then either of the statement sequences which start with this guard can be executed. The `else` keyword can be used to assert that a particular option should only be chosen if no other options are executable. The `if..fi` example

```
if
  :: link!5 -> ...              do
  :: (x==4) -> goto finish          :: (counter<N) -> ...
  :: (x==4) -> ...                  :: (counter==N) -> break
  :: else -> skip               od
fi
 ...
finish:
```

Figure 2.5: Condition, repetition and `goto` statements in Promela.

```
mtype = {N,T,C}
mtype st[6]=N;

proctype user() {
  do
    :: d_step { st[_pid]==N -> st[_pid]=T }
    :: d_step { st[_pid]==T &&
        (st[1]!=C && st[2]!=C && st[3]!=C && st[4]!=C && st[5]!=C)
          -> st[_pid]=C
       }
    :: d_step { st[_pid]==C -> st[_pid]=N }
  od
}

init {
  atomic {
    run user();
    run user();
    run user();
    run user();
    run user();
  }
}
```

Figure 2.6: Promela specification of mutual exclusion with 5 processes.

also illustrates the way flow of control can be organised using traditional `goto` statements and labels.

Example

We illustrate some of the features of Promela using the simple specification shown in Figure 2.6, which is a five-process version of the mutual exclusion protocol described in Section 2.2. The specification consists of: an enumerated type definition for the symbolic constants `N`, `T` and `C`; a global array `st` which is used to hold the state of each process; a *user* proctype, and an `init` process which instantiates a number of *user* processes.

The body of the *user* proctype consists of a single `do..od` statement. Each option in this loop is a `d_step` block which is in turn comprised of a guard (e.g. `st[_pid]==N`) followed by an update (e.g. `st[_pid]=T`). Each block is executable at a given state if its associated guard evaluates to true. A *user* process proceeds by repeatedly executing one of the `d_step` blocks, if any are executable. In this example, the options within the `do..od` statement are mutually exclusive.

The `init` process consists of a sequence of `run` statements, each of which instantiates a *user* process. The `run` statements are contained within an `atomic`

```
never {
T0_init:
        if
        :: (!(st[1]==C) && st[1]==T) -> goto accept_S4
        :: (1) -> goto T0_init
        fi;
accept_S4:
        if
        :: (!(st[1]==C)) -> goto accept_S4
        fi;
}
```

Figure 2.7: Example *never claim* for the *LTL* property $\mathbf{AG}(T_1 \Rightarrow (\mathbf{F}C_1))$.

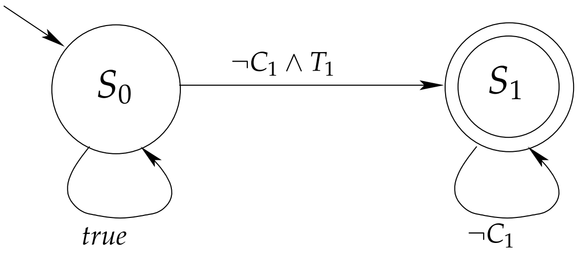


Figure 2.8: Büchi automaton representing the formula $\neg\mathbf{AG}(T_1 \Rightarrow (\mathbf{F}C_1))$.

block, to indicate that they should be executed as an indivisible block.

Note that a Promela array with length $l > 0$ is indexed using integers in the range $0 \ldots (l-1)$. However, in the mutual exclusion example, the five *user* processes have `_pid` variables with values in the range 1–5. Therefore the array `st` is declared with length 6, and position 0 of the array is unused.

Figure 2.6 does not illustrate the declaration and use of channels. Appendices A.2 and A.3 contain Promela specifications which include synchronous and buffered channel declarations respectively.

2.4.2 Reasoning about Promela specifications

As mentioned above, simple logical properties of a Promela specification can be expressed using `assert` statements embedded in the body of proctypes, and more complex *LTL* properties can be expressed using a *never claim* process. The never claim corresponding to an *LTL* property ϕ is a fragment of Promela code equivalent to a Büchi automaton representing the formula $\neg\phi$. Figure 2.7 shows the never claim used to verify the progress property (Property 2, Section 2.2.1), for the simple mutual exclusion example. The propositions T_1 and C_1 in the property are represented by propositions `st[1]==T` and `st[1]==C` respectively in the never claim. Figure 2.8 shows the associated Büchi automaton for $\neg\mathbf{AG}(T_1 \Rightarrow (\mathbf{F}C_1))$. States s_0 and s_1 of the automaton correspond to the labels `T0_init` and `accept_S4` of Figure 2.7 respectively. A never claim can include an expression of the form *name*`[`*i*`]@`*label* to refer that the program counter of process *i*, an instantiation of proctype *name*, is at the position of the specified *label*.

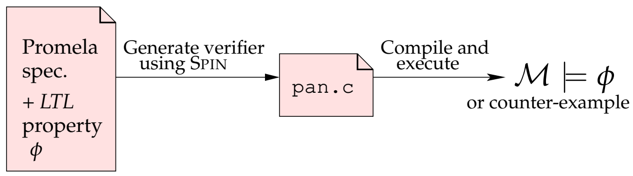


Figure 2.9: The SPIN verification process.

Given a Promela specification (optionally including an associated never claim), SPIN generates a C program, `pan.c`. This program is called the *verifier* generated by SPIN. It includes data structures to represent states of the model associated with the input specification, and search algorithms for exploration of the state-space. The *LTL* model checking algorithm is based on the approach described in Section 2.3.2. Routines to implement various state-space reduction techniques (some of which are discussed in Section 2.6) are also incorporated in `pan.c`. As well as checking properties of a specification expressed using assertions and a never claim, SPIN can be used to search for *deadlock* states (from which no transitions originate).

In order to obtain a verification result, `pan.c` must be compiled and executed. Figure 2.9 illustrates the process of *LTL* property verification using SPIN. Note that a conclusive verification result will only be obtained if memory permits. When checking a large state-space, the verifier may terminate having exhausted available memory without finding an error.

2.4.3 Features of SPIN

A variety of built-in state-space reduction techniques are provided by SPIN. The model checker also supports simulation of Promela specifications though a user interface.

SPIN uses on-the-fly verification and partial-order reduction techniques (discussed in Section 2.6.3) to reduce the number of states which need to be explored during model checking. Additionally, the tool provides data-flow optimisation to identify points in the specification where variables become *dead*, and techniques for statement merging, both of which help reduce verification complexity.

To reduce the per-state storage requirement, SPIN provides three state compression options (see Section 2.6.2), and automatically eliminates write-only variables from the state-vector.

Support for sophisticated simulation of Promela specifications is provided via the XSPIN user interface. Execution of a specification may be simulated randomly or interactively, or may be guided by a counter-example generated by a verification attempt. The interface allows a user to step through a simulation run, tracking the values of global and local variables and channels. In interactive mode, nondeterministic choices are resolved by the user. Channel-based communication be-

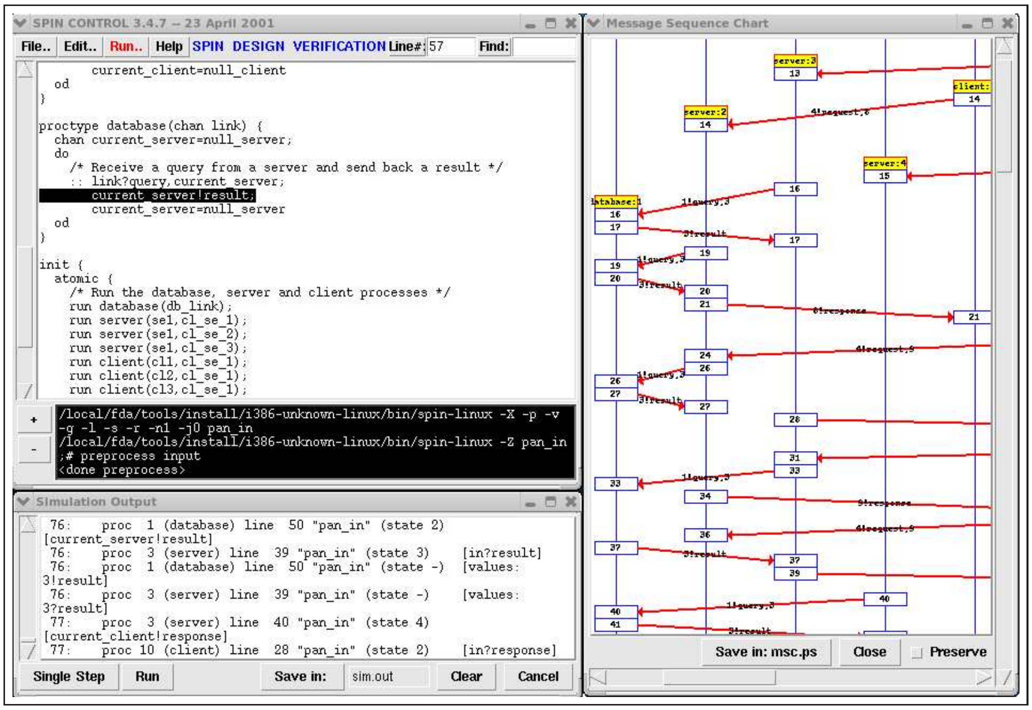


Figure 2.10: Simulation of a Promela specification using message sequence charts.

tween processes may also be graphically illustrated using message sequence charts (MSCs). An MSC represents each process by a vertical *time-line* with a top box indicating the name of the process. Messages between processes are represented by diagonal arrows between time-lines, and indicate a partially ordered set of communication events. MSCs support visualisation of complex communications protocols, and can be a useful aid when understanding counter-examples produced by a model checker [136].

Figure 2.10 shows a screen-shot of the XSPIN interface. A Promela specification is loaded into the top-left pane. The bottom-left pane shows the status of the current simulation run, and the right-hand pane shows an MSC for the simulation. A user can also choose to display the current values of global and local variables in a separate window.

2.5 Other model checkers

We broadly classify model checkers into three categories: *standard* checkers, which check logical properties of high level specifications; *real time/probabilistic* checkers, which allow performance evaluation, and *direct* model checkers, which aim to verify source code.

2.5.1 Standard model checkers

The explicit-state model checker Murϕ [40] uses a language based on a collection of guarded commands (condition/action rules), which are executed repeatedly in an infinite loop. The imperative-style language incorporates new data types, including *multiset* (for describing a bounded set of values whose order is irrelevant to the behaviour of the description) and *scalarset* (for describing a subrange whose elements can be freely permuted; see Section 3.3.2). The verifier performs a depth- or breadth-first search over the state-space to check for absence of deadlock, or satisfaction of safety properties expressed using *assert* statements, or *invariants*. More complex temporal properties cannot be verified.

The tool COSPAN [112] uses an automata-theoretic approach to model checking. The system to be verified is modelled as a collection of coordinating processes described in the S/R (selection/resolution) modelling language. The verifier supports both on-the-fly explicit-state search and symbolic search using binary decision diagrams (BDDs – see Section 2.6.2).

The most successful BDD-based symbolic model checker (see Section 2.6.2) is the *CTL* model checker SMV [126]. Systems are described using the SMV language, which has a precise semantics relating input specifications to their expressions as boolean formulas. SMV supports synchronous and asynchronous communication, and provides for modular descriptions of re-usable components. NuSMV [25] is a re-implemented and extended version of SMV, which includes a textual interaction shell and graphical user interface, as well as techniques for model partitioning and *LTL* model checking.

An enhanced version of SMV, RuleBase [8] is an industry-oriented tool for the verification of hardware designs. In an effort to make the specification of *CTL* properties easier for the non-expert, RuleBase supports its own language, Sugar, as well as standard hardware description languages such as VHDL and Verilog.

In Sections 3.9.1 and 3.9.2 we discuss the implementation of symmetry reduction techniques in standard model checking tools.

2.5.2 Real time and probabilistic model checkers

When modelling certain critical systems, it is essential to include some notion of time. If time is considered to increase in discrete steps (discrete-time), then existing model checkers can be readily extended [3]. The most widely used *dense* real-time model checker (in which time is viewed as increasing continuously) is UPPAAL [115]. Models are expressed as timed automata and properties defined in UPPAAL logic, a subset of timed computation tree logic (*TCTL*). UPPAAL uses a combination of on-the-fly and symbolic techniques so as to reduce the verification problem to that of manipulating and solving constraints. Another real-time model checker is KRONOS [183] which is used to analyse real-time systems modelled in several timed process description formalisms. A real-time extension to COSPAN [4] allows real-time constraints to be expressed by associating lower and upper bounds on the time spent by a process in a local state.

The probabilistic model checker PRISM (probabilistic symbolic model checker) [113, 83, 151] allows reasoning about models of probabilistic systems. The tool supports discrete- and continuous-time Markov chains, as well as Markov decision processes, which allow both probabilistic and non-deterministic behaviour. Properties are written in terms of probabilistic computation tree logic (*PCTL*), or continuous stochastic logic (*CSL*). Models can also be specified using PEPA (performance evaluation process algebra) [82] and converted to PRISM.

We discuss symmetry reduction implementations for real time and probabilistic model checking tools in Section 3.9.3.

2.5.3 Direct model checking tools

Finite state model checking traditionally requires the manual construction of a model, via a specification language, which is then converted into a Kripke structure for model checking. Recently there has been much interest in applying model checking *directly* to program source code written in languages such as Java [158] and C [107]. Early approaches to model checking Java software, e.g. Java PathFinder [77], involved the direct translation of Java code into Promela, and subsequent verification using SPIN. Thus these approaches were restricted to programs containing features supported by both Java and Promela (this is not the case for floating point numbers, for example).

The BANDERA tool [34] avoids direct translation by instead extracting an abstracted finite-state model from Java source code. This model is then translated into a suitable modelling language (Promela or SMV) and model checked accordingly. A second-generation Java PathFinder tool [176] makes extensive use of the BANDERA abstraction techniques, and works directly with Java bytecode.

The dSPIN tool [38] is an extension of SPIN which has been designed for modelling and verifying object-oriented software (in particular Java programs). In addition to the usual features available with SPIN, the dSPIN tool allows for the dynamic creation of heap objects.

The Bogor model checking framework [146] is used to check sequential and concurrent Java programs. Behavioural aspects of a program to be verified are first specified in JML (Java modelling language), which, together with the original Java program, is then translated into a lower-level specification for verification. Bogor exploits the canonical heap representation of dSPIN and is implemented as a plug-in for the Eclipse [33] integrated development environment.

Various tools address the problem of direct model checking of C code. For example, BLAST (Berkeley lazy abstraction software verification tool) [80] uses counter-example guided abstraction refinement (see Section 2.6.3) for proving the correctness of software. Microsoft's SDV (static driver verifier) tool uses the SLAM [5] analysis engine to analyse the source code of Windows device drivers. SDV involves a similar abstraction, verification and refinement loop to that of BLAST.

The VeriSoft model checker [68] is used to verify concurrent processes executing C code. Systematic search of the state-space allows the user to check for deadlock, assertion violations and livelocks. A *stateless* search is used, whereby only states along the current path are stored, together with as many states as possible in the remaining available memory. As a result it is theoretically possible to verify systems of any size. However, the same path may be explored many times, and so search can be very slow.

Recent versions of the SPIN tool allow C code to be embedded into Promela specifications. This feature allows Promela specifications to be automatically extracted from C code [91].

Symmetry reduction techniques have been used in various direct model checkers, as we discuss in Section 3.9.4.

2.6 Tackling the State-space Explosion Problem

As noted in Chapter 1, the major problem which limits the application of model checking is that of *state-space explosion* – as the number of components in a specification of a concurrent system increases, the associated model suffers combinatorial growth, quickly becoming too large to feasibly check. Since its conception, much research in model checking has concentrated on combatting the state-space explosion problem, and a variety of techniques have been proposed.

We identify three approaches to tackling the problem. The first approach involves (usually manual) conversion of a specification into a more efficient specification which captures the same essential behaviour, but has a smaller associated model. Techniques such as design abstraction and source code or communication structure optimisation follow this approach, and are discussed in Section 2.6.1. The second approach relies on a compact representation of states. The most successful technique of this kind is *symbolic model checking*; while state compression and *supertrace* verification have also proved useful in practice. These are discussed in Section 2.6.2. The final approach, discussed in Section 2.6.3, involves reducing the number of states which must be checked to verify a property, without specification-level modification. Techniques include on-the-fly model checking, partial-order reduction, symmetry reduction, abstraction and compositional reasoning.

2.6.1 Specification-level abstraction

The following reduction techniques are applied at the source code level *before* verification. They can therefore be used in conjunction with other state-space reduction techniques applied during verification.

Design abstraction

As discussed in Section 2.1 and illustrated by Figure 2.2, traditional model checking involves manual construction of an abstract high level specification which captures the behaviour of the system under verification. The size of the state-space associated with a specification depends crucially on the level of this abstraction. For example, a data-oriented abstraction of a communications protocol which distinguishes the contents of individual packets will give rise to a much larger state-space than a control-oriented abstraction where packet contents are not specified. Thus good *design abstraction* is one of the key techniques for developing specifications which have tractable associated state-spaces. According to Holzmann [92]:

> *Choosing the right level of abstraction can mean the difference between a tractable model with provable properties and an intractable model that is only amenable to simulation, testing, or manual review.*

A good design abstraction results in the construction of the smallest sufficient (associated) model whilst still allowing verification to be performed [92]. Design abstraction is usually a manual process. However, techniques based on program slicing [171] can be used to automatically remove fragments of a specification which cannot affect the temporal property to be verified [92]. This process is arguably a form of design abstraction.

Source code optimisation

Common modelling pitfalls can lead to unnecessary state space explosion. For example, neglecting to reset a counter variable at the end of a loop can result in many states which are identical except for the counter value. Assuming that the counter has no further use after the loop and will be reset if the loop is executed again, this duplication is redundant and could easily be avoided.

When working with large models it may also be possible to reduce the size of the state vector (the portion of memory required to represent a state) through careful use of advanced specification language features. For detailed source code optimisation strategies for Promela, see [92, 152, 153]. Certain modelling pitfalls can be compensated for by using automatic data-flow optimisation techniques (traditionally used by optimising compilers).

The distinction between design abstraction and source code optimisation is that changing the level of design abstraction may allow the elimination of data (variables) from a specification. Source code optimisation on the other hand involves appropriate management of data so that, at a given level of abstraction, the state-space is minimised.

Communication structure optimisation

The choice of communication structure for a specification can significantly affect the size of the state-space of the underlying model, when buffered channels are used. For example, modelling communication between two processes using two dedicated channels, rather than a single shared channel, increases the number of messages which may potentially be in transit, thus resulting in a larger state-space. While a complex communication structure may be necessary to faithfully model a given system, the truth of a particular temporal property may not be affected by the choice of communication structure. In some cases it is possible to check a property over a *smaller* model with a *simpler* communication structure if it can be shown that the behaviour of the original model relevant to the property can be *emulated* by the reduced model [155, 156]. A similar approach is suggested in [89].

2.6.2 Compact state-space representation

Symbolic model checking

Symbolic model checking [18] is a method by which states and transitions of a model are represented *symbolically* (as opposed to *explicitly*) in order to save space. A particular symbolic approach (namely BDD-based encoding) has proved especially successful for the verification of *CTL* properties for very large systems [126].

A binary decision tree is a structure that is used to represent a boolean formula. Any assignment of truth values to the variables of the formula corresponds to a path down the tree from the root node to a terminal node, which is labelled either true or false. The value of this label determines the value of the function for this assignment of variables. A binary decision diagram (BDD) is obtained from a binary decision tree by merging isomorphic subtrees and identical terminals. Any set of states can be encoded as a BDD. Indeed, if S is a set of states encoded as a set of boolean tuples (on a set X), then for any fixed ordering of the elements of X, there is a unique BDD representing S [17].

An ordered binary decision diagram (OBDD) is a BDD which has a total ordering applied to the variables labelling the vertices of the diagram. The size of the OBDD can vary greatly depending on the ordering used. Heuristics have been developed to find efficient orderings for a given formula (when such an ordering exists). Determining whether a given ordering is more efficient than another ordering is *NP*-complete [11].

For a Kripke structure, both the set of states and the set of transitions can be represented by BDDs. All *possible* states are encoded, as opposed to all *reachable* states. As the superfluous states are unreachable, they do not affect the result of model checking. Indeed, their presence may lead to a simplification of certain BDDs. In addition, it is possible to first compute the reachable states, R say, and then restrict the CTL model checking algorithm to R.

State compression

The SPIN model checker provides two *lossless* compression techniques – they guarantee exhaustive search if memory permits. These techniques reduce the amount of memory required to store each state of a model, and thus allow larger state-spaces to be explored.

The first compression technique is known as *collapse* compression [86]. This method works by storing separate state components for each process in the system, and a separate component for the global data objects of the system (channels and global variables). A global state is then composed from these state components using a small, unique index for each.

The *collapse* method of compression can provide significant reduction in state-space storage requirements, and is fast. However, for models with large state-spaces, SPIN provides a heavy-weight compression scheme using a minimised automaton representation of the state-space, somewhat similar to a BDD [90]. As the state-space of a model is searched, the model checker builds an automaton which recognises states which have been previously seen. Thus on reaching a state, if the state is recognised by the automaton then the state has already been encountered, and search can backtrack. Otherwise the automaton is modified to recognise this new state in the future. The automaton is in general much smaller than the full state-space of the model, and in some cases memory requirements of the verifier are exponentially smaller than for standard search. However, the minimised automaton approach is considerably slower than search without compression, or search using *collapse* compression.

For maximum lossless compression the two state compression techniques can be combined.

Supertrace verification

The compression techniques discussed above are both *lossless*, that is they guarantee exhaustive search if memory permits. In many applications of model checking, finding errors is the main focus of verification rather than proving *absence* of errors. For such applications SPIN provides a *lossy* compression technique called *supertrace verification* [87] (also known as *bitstate hashing*). This technique is useful for exploration of large state-spaces, but does not guarantee full state-space coverage, as we discuss below.

During search (without the supertrace technique), SPIN uses a hash table to store the state-space. When a state is encountered, a hash table lookup determines whether the state has been seen before; if it has not then it is added to a linked-list of states at its hash table slot. Supertrace verification is based on the observation that if the number of hash table slots greatly exceeds the number of reachable states then, assuming a good quality hash function, each state can be stored in a separate slot. In this case, each slot of the hash table could be represented by a single bit – if a state is hashed to a slot in the table which is set to one then it can be assumed that this state has been seen before. This method vastly reduces the memory required to store states, and also leads to efficient hash table operations as linked-lists no longer need to be searched.

Although the probability of hash collision is low, when such collisions happen the model checking algorithm will erroneously assume that a state has been visited before. Thus supertrace mode does not guarantee 100% coverage of the state-space, but can often provide good coverage of a state-space much larger than could be explored using standard search. A variant of supertrace mode, also implemented in SPIN is the *hash-compact* method [181].

2.6.3 Reducing state-space size

Symmetry reduction

Symmetry reduction for model checking is the main topic of this thesis. In Chapter 3 we provide a detailed summary of symmetry reduction theory, and a thorough survey of existing techniques and tools.

On-the-fly model checking

It is not always necessary to build the entire state-space in order to determine whether or not a specification satisfies a given property.

If the property to be checked is *false*, only part of the state-space needs to be constructed, up to the point at which an error state (safety property) or a violating cycle (liveness property) is discovered. However, if there are no errors, the entire reachable part of the state-space must be constructed. This means that although debugging can be performed relatively easily, property verification very quickly becomes prohibitive.

On-the-fly methods are most suitable for explicit-state model checking algorithms based depth-first search, and have been developed to check *LTL*, *CTL* and $CTL^\star$ properties [173, 175, 10]. SPIN is an example of an on-the-fly *LTL* model checker. Approaches for combining on-the-fly techniques with symbolic model checking are restricted to the checking of safety properties [9].

Partial-order reduction

The explosion of states and transitions in a model results from the interleaving of actions of distinct processes in all possible orders. In general, the consideration of such interleavings is crucial: bugs in concurrent systems often arise due to unexpected ordering of actions. However, if a set of transitions are entirely independent and are *invisible* with respect to the property being verified, the order in which they are executed does not affect the overall behaviour of the system. (A transition is invisible with respect to a property ϕ if the truth of ϕ is unaffected by the transition.) Partial-order reduction [53, 67, 135] exploits this fact, and considers only one representative ordering for any set of concurrently enabled, independent, invisible transitions.

Partial-order reduction methods rely on determining a suitable subset of transitions to be considered at every state. As a result, rather than exploring a structure $\mathcal{M}$ an equivalent (usually smaller) structure $\mathcal{M}'$ is explored, with fewer transitions and fewer states. The equivalence in this case is ϕ-stutter equivalence (where ϕ is the property to be checked). If we regard states as being labelled with propositions (as discussed in Footnote 1, page 19) then for two paths π_1 and π_2, let π_1^ϕ and π_2^ϕ be the paths obtained from π_1 and π_2 by restricting the set of labels to the propositions contained in ϕ. Then π_1^ϕ and π_2^ϕ are said to be ϕ-stutter equivalent if they can both be reduced to a common path π by repeated application of the stuttering operator. (The stuttering operator replaces two successive occurrences of a state s in a path by a single occurrence.)

Note that partial-order reduction can only be used to check properties which are closed under stuttering. All *LTL* properties which to not use the next-time (**X**) operator are closed under stuttering.

For some systems, where all actions are dependent on one another, partial-order reduction cannot offer any improvement in verification space or time. In many realistic cases however, partial-order reduction can be extremely effective. For example, for some systems the growth of the state-space as the number of processes increases is reduced from exponential to polynomial when partial-order methods are used. In others the global state-space may increase with the growth of a parameter whereas the size of the reduced state-space remains unchanged [66].

Data abstraction

In Section 2.6.1 we discussed the use of *design abstraction* for modelling a system at an appropriate level of detail so that property verification is tractable. We now discuss the more precise, formal notion of *data abstraction*. Data abstraction works by reducing the number of states in a model by restricting the set of values which variables may take. The resulting reduced Kripke structure $\widehat{\mathcal{M}}$ is an abstraction of the original structure $\mathcal{M}$: every execution in $\mathcal{M}$ has a corresponding execution in $\widehat{\mathcal{M}}$. This idea has been formalised by various authors (e.g. [30, 36, 111]). We now summarise the approach presented in [30].

Recall that a Kripke structure is defined over a set of variables $V = \{v_1, v_2, \ldots, v_k\}$, and that D_i denotes the domain of possible values for v_i ($1 \leq i \leq k$). Formally, for each $1 \leq i \leq k$, let $h_i : D_i \rightarrow \widehat{D_i}$ be a surjection. The surjection h_i maps values of variable v_i onto an *abstract* domain $\widehat{D_i}$. Let $\widehat{D} = \widehat{D_1} \times \widehat{D_2} \times \cdots \times \widehat{D_k}$. Then $h : D \rightarrow \widehat{D}$ defined by $h((d_1, d_2, \ldots, d_k)) = (h_1(d_1), h_2(d_2), \ldots, h_k(d_k))$ is a surjection mapping a state $s \in S$ $(= D)$ to an abstract state $\widehat{s} \in \widehat{D}$. This surjection can be

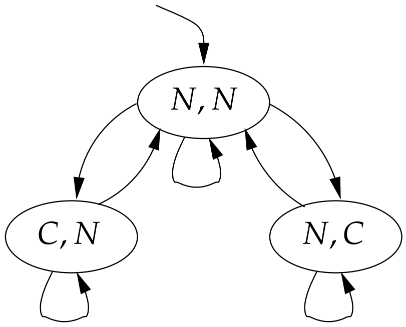


Figure 2.11: Mutual exclusion model reduced via abstraction.

used to define a minimal abstract Kripke structure, each state of which is the image of a set of concrete states under h.

Definition 3 *$\widehat{\mathcal{M}}$ is the abstract Kripke structure over $\widehat{D}$ given by:*

- $\widehat{S} = \widehat{D}$
- $\widehat{S_0} = \{h(s) : s \in S_0\}$
- $\widehat{R} = \{(h(s), h(t)) : (s,t) \in R\}$.

The abstract Kripke structure $\widehat{\mathcal{M}}$ may be significantly smaller than $\mathcal{M}$. If ϕ is a *CTL** formula over $\widehat{\mathcal{M}}$ then a corresponding formula $\mathcal{C}(\phi)$ can be interpreted over $\mathcal{M}$. The formula $\mathcal{C}(\phi)$ is obtained by replacing every state sub-formula $(v_i = \widehat{d_i})$ with the disjunction $\bigvee\{(v_i = d_i) : h(d_i) = \widehat{d_i}\}$, and $(v_i \neq \widehat{d_i})$ with $\neg\mathcal{C}(v_i = \widehat{d_i})$.

The sub-logic of *CTL** consisting of formulas which do not use the path quantifier **E** is denoted *ACTL**. Most temporal properties of interest in verification problems can be expressed in *ACTL** (or even *ACTL*, the corresponding restriction of *CTL*). The next theorem shows that certain *ACTL** properties of $\mathcal{M}$ can be proved by checking $\widehat{\mathcal{M}}$.

Theorem 1 *Let ϕ be an ACTL* formula over $\widehat{\mathcal{M}}$. Then $\widehat{\mathcal{M}} \models \phi \Rightarrow \mathcal{M} \models \mathcal{C}(\phi)$.*

With certain additional conditions on h, this result can be extended to apply to *CTL**.

We illustrate the abstraction approach using the mutual exclusion example. Recall from Section 2.2 that $D_i = \{N, T, C\}$ for $i \in \{1, 2\}$. Let $\widehat{D_i} = \{N, C\}$, and define $h_i(N) = N$, $h_i(T) = N$ and $h_i(C) = C$ for $i \in \{1, 2\}$. This abstraction maps both the *neutral* and *trying* regions onto a single *neutral* region. Figure 2.11 shows the abstract Kripke structure $\widehat{\mathcal{M}}$ corresponding to the Kripke structure of Figure 2.3 under this abstraction.

Let $\phi = \mathbf{AG}(\neg(C_1 \wedge C_2))$ (ϕ is Property 1 of Section 2.2.1). Clearly $\phi \in$ *ACTL**, and $\mathcal{C}(\phi) = \phi$ (since our abstraction does not affect the critical region).

Thus ϕ can be checked over $\widehat{\mathcal{M}}$. Note that $\widehat{\mathcal{M}}$ satisfies the $CTL^\star$ formula $\psi = \mathbf{EG}(N_1 \wedge N_2)$ – there is a path where the transition $(N, N) \rightarrow (N, N)$ is repeated forever. This formula is not in $ACTL^\star$, so we cannot conclude that the formula $\mathcal{C}(\psi) = \mathbf{EG}((N_1 \vee T_1) \wedge (N_2 \vee T_2))$ is satisfied by $\mathcal{M}$ (it is easy to check that it is not).

Note that Theorem 1 does not state that we can *disprove* properties of $\mathcal{M}$ by model checking $\widehat{\mathcal{M}}$. Indeed, given a counter-example for a property ϕ in $\widehat{\mathcal{M}}$, there may be no corresponding counter-example for $\mathcal{C}(\phi)$ in $\mathcal{M}$. For example, $\widehat{\mathcal{M}} \not\models \mathbf{AF}(C_1 \vee C_2)$, which states that the critical section will eventually be reached by one of the processes. Again this is shown via the path where $(N, N) \rightarrow (N, N)$ is repeated forever. It is easy to check that there is no corresponding counter example in $\mathcal{M}$. This spurious counter-example is called a *false negative*.

Abstraction techniques have been used in conjunction with symbolic model checking to verify designs of industrial complexity. However, the user is required to manually specify the abstraction functions. This requires significant insight into the verification problem, compromising the automation of model checking.

Recently there has been progress towards automating the use of abstraction as a state-space reduction technique. Counter-example guided abstraction refinement (CEGAR) [29] is an iterative process where a reduced model is derived from a high level specification using coarse (even arbitrary) abstraction functions. An $ACTL^\star$ property ϕ is checked over this abstract model. If ϕ holds then the truth of $\mathcal{C}(\phi)$ for the unreduced model is established, otherwise a counter-example in the abstract model is reported. An algorithm is used to check whether a corresponding concrete counter-example exists in the original model. If so, then the falsity of $\mathcal{C}(\phi)$ has been established. Otherwise the counter-example is refuted, and information obtained from this counter-example analysis used to *refine* the abstract model, resulting in a larger (but still abstract) state-space which will not admit the spurious counter-example. In practice, the refinement will prohibit a whole class of spurious counter-examples. This process is repeated until a result is obtained, or the abstract model cannot be refined any further without exceeding resources. The CEGAR process is illustrated in Figure 2.12 (adapted from [160]). CEGAR is at the heart of the SLAM and BLAST software model checkers [5, 80].

Compositional verification

A concurrent distributed system is usually comprised of a number of components executing in parallel. It may be possible to verify that a property holds for a model of the system by checking components of the system individually using an *assume-guarantee* proof strategy [140]. With this verification strategy, the typical syntax of a property is: $\langle\phi\rangle\mathcal{M}\langle\psi\rangle$, where ϕ and ψ are temporal formulas. This property states that if $\mathcal{M}$ is a model such that $\mathcal{M} \models \phi$ then it must be the case that $\mathcal{M} \models \psi$. The parallel composition of components $\mathcal{M}_1$ and $\mathcal{M}_2$ can then be checked using the

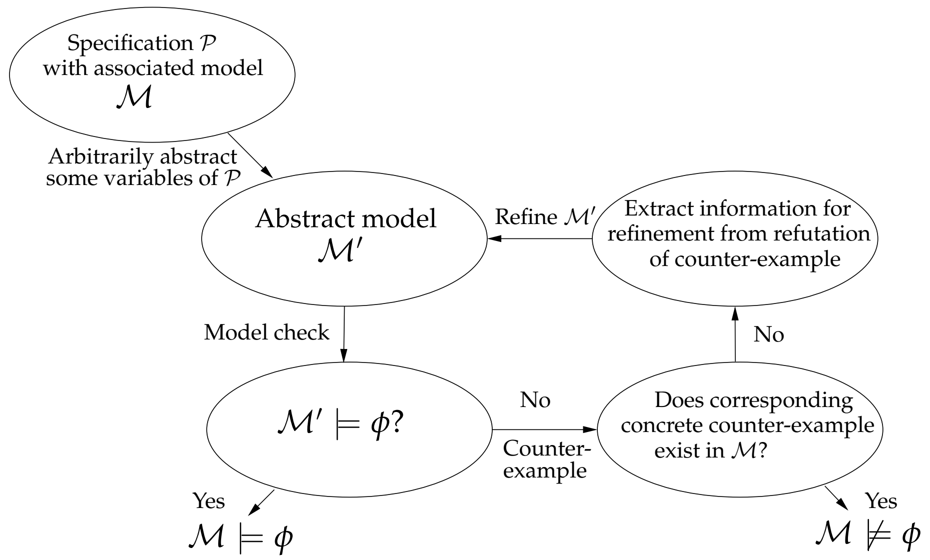


Figure 2.12: The CEGAR process.

$$\frac{\langle True \rangle \mathcal{M}_1 \langle \psi \rangle \qquad \langle \psi \rangle \mathcal{M}_2 \langle \phi \rangle}{\langle True \rangle \mathcal{M}_1 \parallel \mathcal{M}_2 \langle \phi \rangle}$$

Figure 2.13: Proof strategy for compositional verification.

inference rule of Figure 2.13 [32, 73].

Practical compositional verification involves first decomposing a system into components $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_n$ ($n > 0$), and finding an environment assumption A_i for each component $\mathcal{M}_i$. Each environment assumption must capture enough of the behaviour of $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_n$ so that proving $\mathcal{M}_i \parallel A_i \models \phi$ for each i is sufficient to show that $\mathcal{M}_1 \parallel \mathcal{M}_2 \parallel \cdots \parallel \mathcal{M}_n \models \phi$. The challenge in automating these techniques is the derivation of adequate environment assumptions. This can be achieved using methods for regular language learning [134].

Summary

Model checking is an automated technique which can be used to reason about temporal properties of finite state concurrent systems by constructing a model representing all system states. One of the major problems associated with model checking is state-space explosion. The main approaches to overcoming state-space explosion involve construction of an efficient high level specification (using design abstraction and source code or communication structure optimisation), a reduction in state representation size (e.g. symbolic representation or state compression), or a reduction in the number of states or paths explored (e.g. symmetry reduction, partial order reduction and data abstraction).

We have provided a formal definition of *CTL** model checking, outlined basic model checking algorithms for *CTL* and *LTL*, and surveyed a selection of model checking tools and state-space reduction techniques. In particular, we have provided a detailed overview of the SPIN model checker and its input language, Promela.

Chapter 3

Symmetry Reduction

Concurrent systems often contain many replicated components and, as a consequence, model checking may involve making a redundant search over equivalent areas of the state-space. For example, consider the Kripke structure shown in Figure 2.3, associated with the Promela mutual exclusion specification of Figure 2.6 (restricted to two processes). Though simple, this example clearly demonstrates the existence of symmetry within a Kripke structure. In terms of the mutual exclusion property $\mathbf{AG}(\neg(C_1 \wedge C_2))$ (Property 1), any pair of states (A, B) and (B, A), where A and B belong to $\{N, T, C\}$, are *equivalent* (state (A, B)) will satisfy the mutual exclusion property if and only if (B, A) does). Most symmetry reduction techniques exploit this type of symmetry by restricting state-space search to equivalence class representatives, and often result in significant savings in memory and verification time [14, 31, 55, 102].

The earliest use of symmetry reduction in automatic verification was in the context of high-level (coloured) Petri nets [94] where reduction by equivalent markings was used to construct finite reachability trees. These ideas were later extended for deadlock detection and the checking of liveness properties in place/transition nets [167].

3.1 Group Theory

Symmetries of a Kripke structure (see Section 3.2) form a *group*, thus our description of symmetry reduction techniques in this chapter, and the symmetry reduction techniques which we develop throughout the thesis, require some definitions and results from group theory. For more details, see e.g. [22, 81, 148].

3.1.1 Groups, subgroups and homomorphisms

Definition 4 *A group is a non-empty set G together with a binary operation $\circ : G \times G \to G$ which satisfies:*

- *For all $\alpha, \beta, \gamma \in G$, $\alpha \circ (\beta \circ \gamma) = (\alpha \circ \beta) \circ \gamma$*

- *There is an element $id \in G$ such that, for all $\alpha \in G$, $\alpha = id \circ \alpha = \alpha \circ id$. The element id is called the identity of G*
- *For all $\alpha \in G$ there is an element $\beta \in G$ such that $\alpha \circ \beta = \beta \circ \alpha = id$. The element β is called the inverse of α, denoted α^{-1}.*

In practice, the binary operation $\circ$ is usually composition of mappings, so we omit it, writing $\alpha\beta$ for $\alpha \circ \beta$.

Let G be a group and let $H \subseteq G$. If $\alpha\beta \in H$ for all $\alpha, \beta \in H$ (i.e. H is *closed* under the binary operation) then H is also a group, and we say that H is a *subgroup* of G, denoted $H \leq G$. If $H \subset G$ then H is a *proper* subgroup of G, denoted $H < G$.

Definition 5 *Let $X \subseteq G$. Then $\langle X \rangle$ denotes the smallest subgroup of G which contains X, and is called the subgroup generated by X. If $\alpha_1, \alpha_2, \ldots, \alpha_k \in G$ then we use $\langle \alpha_1, \alpha_2, dots, \alpha_k \rangle$ to denote $\langle \{\alpha_1, \alpha_2, \ldots, \alpha_k\} \rangle$.*

For any group G, if $X \subset G$ has the property that $G = \langle X \rangle$ then X is called a set of *generators* for G. It can be shown that if G is a finite group, there exists a generating set X for G with $|X| \leq \log_2 |G|$. As a result, it is often convenient to work with a small generating set for a large group.

Definition 6 *Let H be a subgroup of G, and let $\alpha \in G$. Then the set $H\alpha = \{\beta\alpha : \beta \in H\}$ is a (right) coset of H in G.*

A similar definition can be given for *left* cosets of G in G. We will henceforth use *coset* to mean *right coset*. It can be shown that the set of cosets of H in G is a partition of G. A set of *coset representatives* for H in G is a subset of G which consists of exactly one element from each coset of H in G.

Definition 7 *Let $\alpha, \beta \in G$. The element $\beta^{-1}\alpha\beta \in G$ is called the conjugate of α by β, and is denoted α^{β}. Let $H \leq G$, and suppose that for all $\alpha \in H$ and $\beta \in G$, $\alpha^{\beta} \in H$ (i.e. H is closed under conjugation). Then H is a normal subgroup of G, and we write $H \trianglelefteq G$.*

A mapping between two groups which preserves products of elements is called a *homomorphism*:

Definition 8 *Let $(G_1, \circ)$, $(G_2, \star)$ be groups. A homomorphism from G_1 to G_2 is a mapping $\theta : G_1 \rightarrow G_2$ which satisfies, for all $\alpha, \beta \in G_1$,*

$$\theta(\alpha \circ \beta) = \theta(\alpha) \star \theta(\beta).$$

If θ is injective then θ is a monomorphism from G_1 to G_2. If θ is bijective then θ is an isomorphism from G_1 to G_2, and G_1 and G_2 are said to be isomorphic, denoted $G_1 \cong G_2$.

Isomorphic groups are algebraically indistinguishable, and in some sense can be thought of as equal – they differ only in that their elements may be labelled differently [81]. However, two isomorphic groups may have distinct actions on a set (see Section 3.1.3), so for the purposes of this work it is important to regard groups which are isomorphic but whose elements are not presented in the same form, as distinct.

The following standard theorem shows that if there is a monomorphism from a group G_1 to a group G_2 then G_1 is isomorphic to a subgroup of G_2:

Theorem 2 *Let G_1, G_2 be groups and $\theta : G_1 \to G_2$ a monomorphism. Then $G_1 \cong \theta(G_1) \leq G_2$, where $\theta(G_1) = \{\theta(\alpha) : \alpha \in G_1\}$.*

3.1.2 Permutation groups

Let X be a non-empty set. A *permutation* of X is a bijection $\alpha : X \to X$. The set of all permutations of X forms a group under composition of mappings, denoted $Sym(X)$. Given a group $H \leq Sym(X)$, we use $moved(H)$ to denote the subset of X which is affected by H: $moved(H) = \{x \in X : \alpha(x) \neq x \text{ for some } x \in H\}$. For $\alpha \in Sym(X)$ we use $moved(\alpha)$ to denote the set $moved(\{x\})$. The *degree* of a permutation group G is defined to be $|moved(G)|$.

If X is finite then it can be shown that $|Sym(X)| = |X|!$, and an element of $Sym(X)$ can be conveniently expressed using *disjoint cycle form*. Let $\alpha \in Sym(X)$. If $\alpha = id$ then we write id for α as usual. Otherwise, we can write α as a product of cycles as follows:

$$\alpha = (a_{1,1}\ a_{1,2}\ \ldots\ a_{1,s_1})(a_{2,1}\ a_{2,2}\ \ldots\ a_{2,s_2})\ldots(a_{t,1}\ a_{t,2}\ \ldots\ a_{t,s_t})$$

where $t > 0$, $2 \leq s_i \leq |X|$ $(1 \leq i \leq t)$, $a_{i,j} \in X$ $(1 \leq i \leq t, 1 \leq j \leq s_t)$, and the $a_{i,j}$ are all distinct. In this form, for $x \in X$, if $x = a_{i,j}$ for some i and j then $\alpha(x) = a_{i,j+1}$ if $j < s_i$ and $\alpha(x) = a_{i,j'}$ where $j' = j+1$ if $j < s_i$ and $j' = 1$ if $j = s_1$; otherwise $\alpha(x) = x$.

Definition 9 *Let X be a non-empty set, $G \leq Sym(X)$, $x \in X$, $Y \subseteq X$, and $\mathcal{X}$ be a partition of X.*

- *The stabiliser of x in G is the set $stab_G(x) = \{\alpha \in G : \alpha(x) = x\}$.*
- *The pointwise statbiliser of Y in G is the set $stab^*_G(Y) = \{\alpha \in G : \alpha(x) = x \ \forall\ x \in Y\} = \bigcap_{x \in Y} stab_G(x)$.*
- *For $\alpha \in G$, define $\alpha(Y) = \{\alpha(x) : x \in Y\} \subseteq X$. The setwise stabiliser of Y in G*

is the set $stab_G(Y) = \{\alpha \in G : \alpha(Y) = Y\}$*.*

- *The partition stabiliser of* $\mathcal{X}$ *in* G *is the set* $stab_G(\mathcal{X}) = \{\alpha \in G : \alpha(Y) = Y \ \forall \ Y \in \mathcal{X}\} = \bigcap_{Y \in \mathcal{X}} stab_G(Y)$*.*

It is straightforward to show that $stab_G(x)$, $stab^*_G(Y)$, $stab_G(Y)$ *and* $stab_G(\mathcal{X})$ *are all subgroups of* G*.*

Definition 10 *Let* $G \leq Sym(X)$ *where* X *is a non-empty set. The group* G *induces an equivalence relation* $\equiv_G$ *on* X *thus:* $x \equiv_G y \Leftrightarrow x = \alpha(y)$ *for some* $\alpha \in G$*. The equivalence class under* $\equiv_G$ *of an element* $x \in X$*, denoted* $[x]_G$*, is called the* orbit *of* x *under* G*. The group* G *is* transitive *if there is a single orbit,* X*. An orbit* $O \subseteq X$ *is non-trivial if* $|O| > 1$*. When considering actions of* G *on two distinct sets* X *and* Y*, it is sometimes convenient to write* $[x]_G$ *for the orbit of* $x \in X$ *under* G*, and* $orb_G(y)$ *for the orbit of* $y \in Y$ *under* G*.*

Two important classes of permutation groups are symmetric groups and cyclic groups:

Definition 11 *For* $n > 0$*, the group* $Sym(\{1, 2, \ldots, n\})$ *is called the* symmetric group of degree n*, denoted* S_n*. From the above, we have* $|S_n| = n!$*.* S_n *is often referred to as the* full *symmetry group.*

Definition 12 *The* cyclic group of degree n*, denoted* C_n*, is the subgroup of* S_n *generated by the cycle* $(1\ 2\ \ldots\ n)$*.*

3.1.3 Group actions on sets

Fundamental to most applications of symmetry reduction in model checking is the idea that a group of permutations of a given set *induces* a group of permutations on another (usually larger) set. For example, a group of process identifier permutations naturally induces a group of permutations of the set of states associated with a specification. We describe this idea formally using *group actions*. The following definition and theorem are adapted from [148].

Definition 13 *We say that 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* $\alpha(x) \in X$ *and that, for all* $x \in X$ *and* $\alpha, \beta \in G$*,*

- $(\alpha\beta)(x) = \alpha(\beta(x))$
- $id(x) = x$*.*

Theorem 3 *Let G act on X. Then to each $\alpha \in G$ there corresponds an element $\rho_\alpha \in Sym(X)$ defined by $\rho_\alpha : x \mapsto \alpha(x)$, and the map $\rho : G \to Sym(X)$ defined by $\rho : \alpha \mapsto \rho_\alpha$ is a homomorphism.*

We call the homomorphism ρ the *permutation representation* of G corresponding to the group action.

3.1.4 Products of groups

Certain groups can be described as products of their subgroups. Four important kinds of product are direct, disjoint, wreath and semi-direct products, which we introduce in Definitions 14, 15, 16–17 and 18 respectively.

Let G be a group, and $H_1, H_2, \ldots, H_k$ subgroups of G ($1 \leq i \leq k$, $k > 1$). If $G = H_1 H_2 \ldots H_k = \{\alpha_1 \alpha_2 \ldots \alpha_k : \alpha \in H_i \ (1 \leq i \leq k)\}$ then G is the *product* of the H_i.

Definition 14 *Let G be a group, and let $H_1, H_2, \ldots, H_k$ be subgroups of G. Then G is the (internal) direct product of the H_i, written $G = H_1 \times H_2 \times \cdots \times H_k$, if $H_i \trianglelefteq G$ ($1 \leq i \leq k$), G is the product of the H_i, and $H_i \cap H_j = \{id\}$ for all $i \neq j$ $(1 \leq i, j \leq k)$.*

Definition 15 *Let $G \leq Sym(X)$, where X is a non-empty set. Suppose that G is the product of subgroups $H_1, H_2, \ldots, H_k$, and that $moved(H_i) \cap moved(H_j) = \emptyset$ for all $1 \leq i \neq j \leq k$. Then G is denoted $H_1 \bullet H_2 \bullet \cdots \bullet H_k$, and called the disjoint product of the H_i, and the H_i a disjoint product decomposition for G. The disjoint product is said to be non-trivial if $G \neq H_i \neq \{id\}$ for all $1 \leq i \leq k$.*

Note that if G has two disjoint product decompositions such that the constituent subgroups of the second product are all subgroups of constituent subgroups of the first, then we say that the second decomposition is *finer* than the first.

It is easy to show that if G is the disjoint product of $H_1, H_2, \ldots, H_k$ then G is the direct product of these subgroups, thus disjoint products are a special case of direct products for permutation groups with a specific action.

The following definition of the *wreath product* of two permutation groups, which we call the *outer* wreath product, is adapted from a definition given in [105] and allows us to construct a new permutation group from two arbitrary permutation groups.

Definition 16 *Let $H \leq S_m$ and $K \leq S_d$ for some $m, d > 0$. Let $X = \{1, 2, \ldots, md\}$, and let $\{X_1, X_2, \ldots, X_d\}$ be a partition of X into equal-sized subsets, with $X_i =$*

$\{(i-1)m+1,(i-1)m+2,\dots,(i-1)m+m\}$ $(1 \le i \le d)$. We define an action for K, and d distinct actions for H, on X.

For $\beta \in K$ and $x \in X$, suppose $x \in X_i$ for some $1 \le i \le d$, so that $x = (i-1)m+t$ for some $1 \le t \le m$. Define $\beta(x) = (\beta(i)-1)m+t$. Let σ be the permutation representation corresponding to this action of K on X.

For $\alpha \in H$, $x \in X$ and $1 \le i \le d$, suppose $x \in X_j$ for some $1 \le j \le d$, so that $x = (j-1)m+t$ for some $1 \le t \le m$. Define $\alpha(x) = x$ if $i \ne j$ and $\alpha(x) = (j-1)m+\alpha(t)$ otherwise. Let σ_i be the permutation representation corresponding to this action of H on X.

The outer wreath product of H and K is the group $H \wr K \le Sym(X)$ defined as follows: $H \wr K = \{\sigma(\beta)\sigma_1(\alpha_1)\sigma_2(\alpha_2)\dots\sigma_d(\alpha_d) : \beta \in K, \alpha_i \in H \ (1 \le i \le d)\}$.

Note that in the above definition, each of the σ_i permutes a different set of the partition and σ permutes the partition.

The next definition, which we call the *inner* wreath product, allows us to identify an existing group as a wreath product of subgroups. It is similar to, but more general than Definition 16: the requirement that X must be the set $\{1,2,\dots,md\}$ partitioned into contiguous subsets is lifted.

Definition 17 *Let $H \le S_m$ and $K \le S_d$ for some $m, d > 0$. Let X be a set with $|X| = md$, and $\{X_1, X_2, \dots, X_d\}$ a partition of X into equal-sized subsets, where X_i has the form $X_i = \{x_{i,1}, x_{i,2}, \dots, x_{i,m}\}$ for some $x_{i,j} \in X$ $(1 \le i \le d, 1 \le j \le m)$. We define an action for K, and d distinct actions for H, on X.*

For $\beta \in K$ and $x \in X$, suppose $x \in X_i$ for some $1 \le i \le d$, so that $x = x_{i,t}$ for some $1 \le t \le m$. Define $\beta(x) = x_{\beta(i),t}$. Let σ be the permutation representation corresponding to this action of K on X.

For $\alpha \in H$, $x \in X$ and $1 \le i \le d$, suppose $x \in X_j$ for some $1 \le j \le d$, so that $x = x_{j,t}$ for some $1 \le t \le m$. Define $\alpha(x) = x$ if $i \ne j$ and $\alpha(x) = x_{j,\alpha(t)}$ otherwise. Let σ_i be the permutation representation corresponding to this action of H on X.

If $G = \{\sigma(\beta)\sigma_1(\alpha_1)\sigma_2(\alpha_2)\dots\sigma_d(\alpha_d) : \beta \in K, \alpha_i \in H \ (1 \le i \le d)\}$ then G is the inner wreath product of H and K, also denoted $H \wr K$.

For neatness, we do not use notation to distinguish inner and outer wreath products, and explicitly state the type of product referred to when necessary. Note that we can unambiguously refer to the outer wreath product of two permutation groups, but must specify the partition $\mathcal{X} = \{X_1, X_2, \dots, X_d\}$ when reasoning about an inner wreath product. We refer to the triple $(H, K, \mathcal{X})$ as a wreath product *decomposition* for G, and say that the decomposition is *non-trivial* if both H and K are non-trivial.

It is clear that any inner wreath product $H \wr K$ (with associated partition) is identical, up to renaming of points, to the outer wreath product $H \wr K$. The order of

$H \wr K$ depends on the orders of H and K, and the degree of K:

Theorem 4 *If G is an inner or outer wreath product $H \wr K$ then $|G| = |H|^d \times |K|$, where d is the degree of K.*

Definitions 16 and 17 describe wreath products with the *imprimitive action* [22]. There are other definitions of wreath products with other kinds of action, and it is important to note that the results on wreath products which we present in Section 9.4 are specific to the imprimitive action.

Direct products and wreath products are both generalised by *semi-direct* products:

Definition 18 *Let G be a group, N a normal subgroup of G and H a subgroup of G. Suppose $G = NH$ and $N \cap H = \{id\}$. Then G is a* semi-direct product *of N and H, denoted $N \rtimes H$.*

3.1.5 Graphs and automorphisms

An *undirected/directed graph* (referred to as a *graph/digraph*) is a pair (V, E) where V is a set of vertices and E a set of edges – unordered/ordered pairs of vertices. An edge of a graph is written as a set $\{u, v\}$ whereas a digraph edge is written as a pair (u, v) $(u, v \in V)$. A *hypergraph* is a pair (V, E) where V is a set of vertices and $E \subseteq 2^V$ a set of *hyper*-edges.

A digraph (V, E) is *bipartite* if $V = V_1 \cup V_2$, where $V_1 \subset V$ and $V_2 \subset V$ are disjoint non-empty sets and, for $(u, v) \in E$, $u \in V_1$ and $v \in V_2$, or $u \in V_2$ and $v \in V_1$.

A *colouring* of (di/hyper)graph (V, E) is a mapping $C : V \to K$, where K is a finite set of *colours*. A coloured (di/hyper)graph is a triple (V, E, C) such that (V, E) is a (di/hyper)graph and C a colouring of (V, E).

If $\Gamma = (V, E)$ is graph/hypergraph and α a permutation of V, then for any $e = \{v_1, v_2, \ldots, v_m\} \in E$, α acts on e thus:

$$\alpha(e) = \{\alpha(v_1), \alpha(v_2), \ldots, \alpha(v_m)\}.$$

The action of α on the edges of a digraph is defined similarly, and the ordering must be preserved in this case.

Definition 19 *Let $\Gamma = (V, E, C)$ be a coloured (di/hyper)graph and α a permutation of V. Then α is an* automorphism *of Γ if the following conditions are satisfied:*

- *For all $e \in E$, $\alpha(e) \in E$*
- *For all $v \in V$, $C(v) = C(\alpha(v))$.*

An automorphism of an un-coloured (di/hyper)graph is a permutation of V which satisfies the first condition in the above definition. The set of all automorphisms of a (di/hyper)graph Γ forms a group under composition of mappings, denoted $Aut(\Gamma)$.

3.1.6 GAP and GRAPE

GAP (groups, algorithms and programming) [63] is a computational algebra system which provides data structures and algorithms for working with a variety of algebraic structures. In particular, GAP includes a large library of permutation group algorithms. Given generators (specified in disjoint cycle form) for a permutation group G acting on the set $\{1, 2, \ldots, n\}$, GAP functions can be used to compute, for example, subgroups of G with particular properties (such as point- and set-stabilisers); the orbits of G on $\{1, 2, \ldots, n\}$; coset representatives for a subgroup H of G; and homomorphisms from G to another group. The fundamental permutation group algorithms which GAP uses are described in [19, 159].

On its own, GAP provides little support for graph-theoretic computation. GRAPE (graph algorithms using permutation groups) [165] consists of a set of functions which can be imported into GAP, including a function to compute the automorphism group of a directed, coloured graph. This function interfaces with the *nauty* (no automorphisms, yes) program [124], which uses the most efficient algorithm currently known for finding the automorphism group of a graph [123].

For further details of the techniques used by GRAPE and *nauty* see [166]. In subsequent chapters we make use of the following functions:

- `AutGroupGraph(`Γ $[,C]$`)`: GRAPE function which returns generators for the automorphism group of the directed graph Γ (see Definition 19). The optional argument C allows a colouring on the vertices of Γ to be specified, in which case generators for the subgroup of automorphisms which preserve this colouring will be computed
- `IsomorphismGroups(`G,H`)`: GAP function which computes an isomorphism between the groups G and H if they are isomorphic (see Definition 8), and returns *fail* otherwise
- `IsomorphicSubgroups(`G,H`)`: GAP function which computes all monomorphisms (see Definition 8) from H into G up to conjugacy of the image groups. (Subgroups H and K of G conjugate if $H = \alpha^{-1}K\alpha$ for some $\alpha \in G$.)

3.2 Symmetry Reduction Using Quotient Structures

Definition 20 *Let $\mathcal{M} = (S, S_0, R)$ be a Kripke structure over D. An automorphism of $\mathcal{M}$ is a permutation $\alpha : S \to S$ which preserves the transition relation and set of initial states. That is α satisfies the following conditions:*

1. *For all* $s, t \in S, \quad (s,t) \in R \Rightarrow (\alpha(s), \alpha(t)) \in R$
2. $\alpha(S_0) = S_0$.

The set of all automorphisms of a Kripke structure $\mathcal{M}$ forms a group under composition of mappings, denoted $Aut(\mathcal{M})$.

In a model of a concurrent system with many replicated processes, Kripke structure automorphisms may involve the permutation of process identifiers or data values throughout all states of the model. On the other hand, a model may include a data structure which has geometrical symmetry [88], in which case Kripke structure automorphisms involve applying the geometrical symmetries throughout all states of the model. In each of these cases there is group G which permutes a (typically small) set of process identifiers, data values or nodes of a data structure, and an action of G on S (see Definition 13). Let ρ be the permutation representation corresponding to this action (see Theorem 3). The group of automorphisms of $\mathcal{M}$ *induced* by G is $\rho(G)$, the image of G under the permutation representation. Given $\alpha \in G$, rather than referring to the automorphism ρ_α of $\mathcal{M}$ we sometimes say simply that α is an automorphism of $\mathcal{M}$.

Given a subgroup G of $Aut(\mathcal{M})$, the orbits of S under G (see Definition 10) can be used to construct a *quotient* Kripke structure $\mathcal{M}_G$ as follows:

Definition 21 *The quotient Kripke structure $\mathcal{M}_G$ of $\mathcal{M}$ with respect to G is a tuple $\mathcal{M}_G = (S_G, S_G^0, R_G)$ where:*
- $S_G = \{rep_G(s) : s \in S\}$ *(where $rep_G(s)$ is a unique representative of $[s]_G$)*
- $S_G^0 = \{rep_G(s) : s \in S_0\}$
- $R_G = \{(rep_G(s), rep_G(t)) : (s,t) \in R\}$.

If G is non-trivial then the quotient structure $\mathcal{M}_G$ is smaller than $\mathcal{M}$. For any $s \in S$, the size of $[s]_G$ is bounded by $|G|$, and so the theoretical minimum size of S_G is $|S|/|G|$. Since for highly symmetric systems we may have $|G| = n!$, where n is the number of components, symmetry reduction potentially offers a considerable reduction in memory requirements.

To give an example of a quotient structure, for the mutual exclusion example shown in Figure 2.3, observe that swapping the process indices 1 and 2 throughout all states is an automorphism of the structure. If α denotes this automorphism then for this example $Aut(\mathcal{M}) = \{\alpha, id\}$, where id is the identity mapping. Choosing a unique representative from each orbit we obtain the quotient Kripke structure $\mathcal{M}_{Aut(\mathcal{M})}$ illustrated by Figure 3.1.

It can be shown [31, 55] (see Theorem 5 below) that a model and its quotient model satisfy the same *symmetric CTL** formulas. A *CTL** formula ϕ is symmetric, or invariant, with respect to G if for every maximal propositional sub-formula f appearing in ϕ (see Section 2.2.1), and for every $\alpha \in G$, $\mathcal{M}, s \models f \Leftrightarrow \mathcal{M}, \alpha(s) \models f$.

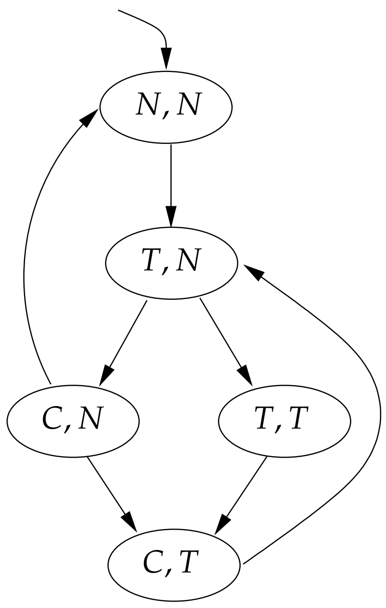


Figure 3.1: Quotient Kripke structure for two-process mutual exclusion.

Theorem 5 *If $\mathcal{M}$ and $\mathcal{M}_G$ denote a model and its quotient model with respect to a group G respectively, then $\mathcal{M}, s \models \phi \Leftrightarrow \mathcal{M}_G, rep_G(s) \models \phi$, for every symmetric $CTL^\star$ formula ϕ.*

Theorem 5 is proved by establishing a correspondence between the paths of $\mathcal{M}$ and $\mathcal{M}_G$, and using induction on the structure of $CTL^\star$ formulas [31, 55]. An analogous result holds for symmetric μ-calculus formulas [55].

Since, by Condition 2 of Definition 20, the initial states of $\mathcal{M}$ are preserved by G, we have the following corollary:

Corollary 1 *With the same conditions as Theorem 5, $\mathcal{M} \models \phi \Leftrightarrow \mathcal{M}_G \models \phi$.*

Corollary 1 is similar to Theorem 1 (see Section 2.6.3) which shows that properties of a Kripke structure can be inferred by proving properties of a structure which has been reduced by data abstraction. There are two key differences. Theorem 1 is restricted to $ACTL^\star$ formulas, while Corollary 1 applies to any $CTL^\star$ formula which is symmetric. In addition, unlike Theorem 1, Corollary 1 can be used to find errors as well as prove properties since, in this case, the implication is two-way. Consider the two-process mutual exclusion property $\mathbf{AG}(\neg(C_1 \wedge C_2))$ (Property 1 of Section 2.2.1). Let us call this property ϕ_1. Clearly ϕ_1 is symmetric with respect to the automorphism group $G = \{\alpha, id\}$, where α is defined as above. Thus the Kripke structure $\mathcal{M}$ (represented by Figure 2.3) satisfies ϕ_1 if and only if the quotient structure $\mathcal{M}_G$ (represented by Figure 3.1) does. Therefore, to check the mutual exclusion property, it is sufficient to check the quotient model only. Note that $\mathcal{M}_G$ also sat-

Algorithm 1 Algorithm to construct a quotient Kripke structure.

$S_G := \{rep_G(s) : s \in S_0\}$
$unexplored := \{rep_G(s) : s \in S_0\}$
$R_G := \emptyset$
while $unexplored \neq \emptyset$ **do**
  remove a state s from *unexplored*
  for all successor states t of s **do**
    add $s \rightarrow rep_G(t)$ to R_G
    if $rep_G(t) \notin S_G$ **then**
      add $rep_G(t)$ to S_G
      add $rep_G(t)$ to *unexplored*
    end if
  end for
end while

isfies the property ϕ_2 defined as $\mathbf{AG}(\neg C_2)$. However, as ϕ_2 is not symmetric with respect to the automorphism group, we cannot infer the truth (or otherwise) of ϕ_2 for $\mathcal{M}$. (Indeed, clearly $\mathcal{M} \not\models \phi_2$.) The *progress* property $\mathbf{AG}(T_1 \Rightarrow (\mathbf{F}C_1))$ (Property 2 of Section 2.2.1) is also not symmetric – it refers to process 1 in an asymmetric way). Consider Property 3 below, a weaker version of the progress property:

Property 3 $\mathbf{AG}((T_1 \vee T_2) \Rightarrow (\mathbf{F}(C_1 \vee C_2)))$.

This asserts that if *some* process is in the trying region then eventually *some* (but not necessarily the same) process will reach the critical section. This property is symmetric with respect to $Aut(\mathcal{M})$, and it is easy to check that it holds for both $\mathcal{M}$ and $\mathcal{M}_G$.

Algorithm 1 (adapted from [55, 102]), shows how a quotient structure can be constructed incrementally if symmetries of the Kripke structure can be identified before search. The successors of a given state are determined by the transition rules of a high level specification. Note that to determine a unique element $rep_G(s)$ for each orbit $[s]_G$, we require a *canonicalisation* function. We discuss the problem of constructing such a canonicalisation function in Section 3.4. Using Algorithm 1 it may be possible to build the quotient structure even though the original structure is intractably large.

An approach that is suggested for symmetry reduction during automata-based model checking involves the construction of an *annotated quotient structure* (AQS) [55, 161]. In this case there is a labelled edge between representative states $rep_G(s)$ and $rep_G(t)$ for every edge that exists (in $\mathcal{M}$) from $rep_G(s)$ to an element of $[t]_G$. If $(rep_G(s), t')$ were such an edge (in $\mathcal{M}$) then the edge (in the annotated quotient structure) would be labelled with a permutation α such that $\alpha(rep_G(t)) = t'$. Not only is it possible to *unwind* the original structure $\mathcal{M}$ from the (annotated) quotient structure, but it is also possible to check properties expressed in *indexed* $CTL^\star$ – an extension to $CTL^\star$ in which properties include the indexed quantifiers *for all processes* or *for some process*. In addition, properties to be checked are not required to be symmetric with respect to the group G. We discuss the use of $AQSs$ to verify properties under fairness assumptions in Section 3.6.2.

3.3 Identifying Symmetry

The first step which must be accomplished by any method which exploits symmetry is the identification of symmetries in a model. Let $\mathcal{M}$ be a Kripke structure. An obvious approach to solving this problem would be to construct $\mathcal{M}$, and then to find a symmetry group G of $\mathcal{M}$ using a standard algorithm (e.g. *nauty* [123]). These symmetries could be used to reduce $\mathcal{M}$ to a quotient model, $\mathcal{M}_G$.

This approach is flawed in two ways. Firstly, finding automorphisms of a Kripke structure is equivalent to checking for state-space isomorphism, which for large state-spaces is a hard problem (no polynomial time algorithm is known [123]). Secondly, if enough resources were available to construct $\mathcal{M}$ then symmetry reduction would be unlikely to be of much benefit. Indeed, the power of reduction techniques is that they allow a reduced model to be checked even when the unreduced model is intractable.

Thus the problem is to find symmetries of $\mathcal{M}$ *without* building $\mathcal{M}$ explicitly. We now discuss four approaches to symmetry identification: explicit specification of symmetry group generators, specification of symmetries via a purpose-built *scalarset* data type, restriction of the specification language to guarantee symmetry between components, and inference of symmetry by communication structure analysis.

3.3.1 Manual specification of a symmetry group

The problem of symmetry detection can be avoided altogether by requiring the user to manually specify generators for a symmetry group [27, 69, 133]. This approach requires expert user knowledge of symmetry reduction theory, and is prone to error. Nevertheless, if used with care, providing the option to specify symmetry manually allows symmetry reduction to be applied when the user can identify symmetries in a specification which are not recognised by an automated approach.

3.3.2 Scalarsets

A popular approach to symmetry detection involves annotation of the system description via a purpose-built data type [102]. The data type is called a *scalarset*, and acts as documentation that certain symmetries are present in a specification expressed in the Murϕ description language [40].

Definition 22 *A scalarset is an integer sub-range with restricted operations as follows:*

1. *An array with a scalarset index type can only be indexed by a scalarset variable of exactly the same type*
2. *A term of scalarset type must be a variable reference. (A scalarset may not appear as an operand to + or any other operator in a term)*
3. *Scalarset variables may only be compared using =, and in such cases, must be of exactly the same type*
4. *For all assignments $d := t$, if d is a scalarset variable, t must be a term of exactly the same scalarset type*
5. *If a scalarset variable is used as an index of a* **for** *statement, the body of the statement is restricted so that the result of the execution is independent of the order of the iterations.*

The restrictions are sufficient to ensure that consistent permutation of scalarset variables in all states corresponds to an automorphism of the state-space. Furthermore, violations of the restrictions can be detected in polynomial time [102]. As the above conditions refer to general language features, they can clearly be adapted to apply to other specification formalisms.

Given a specification $\mathcal{P}$ containing a scalarset $\mathcal{S}$ which represents the integer sub-range $\{0, 1, \ldots, n-1\}$ (for some $n > 0$), any permutation of $\{0, 1, \ldots, n-1\}$ naturally induces a permutation of the associated state-space. This is best illustrated by an example. Let x and y be variables of $\mathcal{P}$ with scalarset and non scalarset type respectively, and A an array with scalarset index type and element type. Let t be a state of the model associated with $\mathcal{P}$, and let $t.x$, $t.y$ and $t.A[i]$ denote the values of x, y and element i of A ($0 \leq i \leq n-1$) at state t respectively. Let $\alpha_{\mathcal{S}}$ be any permutation of $\{0, 1, \ldots, n-1\}$. Then $\alpha_{\mathcal{S}}$ is defined to act on state t such that: $\alpha_{\mathcal{S}}(t).x = \alpha_{\mathcal{S}}(t.x)$; $\alpha_{\mathcal{S}}(t).y = t.y$, and $\alpha_{\mathcal{S}}(t).A[i] = \alpha_{\mathcal{S}}(t.A[\alpha_{\mathcal{S}}^{-1}(i)])$. For a precise, recursive definition of this action, and a proof of the following theorem, see [102].

Theorem 6 *Given a specification containing a scalarset $\mathcal{S}$, every permutation $\alpha_{\mathcal{S}}$ on the states of the state-space $\mathcal{M}$ derived from the specification is an automorphism of $\mathcal{M}$.*

Corollary 2 *If a specification $\mathcal{P}$ has scalarsets $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_n$, there are symmetries in the state graph $\mathcal{M}$ and we can use the symmetry-reduced state graph $\mathcal{M}_G$ to perform verification, where G is the group of all permutations of the states with respect to $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_n$.*

```
atomic {
  j = 0;
  do :: j==A -> break
     :: j<A -> body; j++
  od
}
```

Figure 3.2: Synthesising a `for` loop with scalarset index variable in Promela.

If $|\mathcal{S}_i|$ is the size of scalarset $\mathcal{S}_i$ ($1 \leq i \leq n$), then the symmetry group G has order $|S_1|! \times |S_2|! \times \cdots \times |S_n|!$.

To illustrate the need for Condition 5 of Definition 22, let $\mathcal{S}$ be a scalarset type with range $\{0, 1, \ldots, n-1\}$. Consider a loop with counter variable j of type $\mathcal{S}$, and let k be a global variable of the same scalarset type. Including an assignment k := j at the end of the loop body means that the final value of k will be $n-1$. This distinguishes the value $n-1$ from the other values in the range of $\mathcal{S}$, thus the range cannot be safely permuted.

An example of the use of scalarsets with Murϕ is in the verification of the Needham-Shroeder public key protocol [131]. The protocol involves a set of *Initiator* processes and a set of *Receptor* processes. Each *Initiator* process is identified by the variable *Initiatorid* which is used to index an array storing the state of each Initiator process. The *Initiatorid* variable is also used as an index within a **for** loop containing the rules determining the behaviour of each Initiator process. As the *Initiator* processes behave symmetrically, by declaring the *Initiatorid* variable with a scalarset type, symmetry reduction can be automatically performed. Similarly, a scalarset type can be used to identify symmetry between the *Receptor* processes.

Scalarsets have been used to implement symmetry reduction techniques for the SPIN model checker via the SymmSpin tool [14] (see Section 3.9.1). Unlike Murϕ, the Promela language does not include a `for` construct. However, a `for` loop which uses a scalarset index variable can be synthesised using the Promela `do..od` construct within an `atomic` block, as shown in Figure 3.2. In the figure, j is a scalarset variable which has range $\{0, 1, \ldots, A-1\}$ for some $A > 0$. The loop body consists of a sequence of statements between the condition `j<A` and the increment `j++`, which must satisfy Condition 5 of Definition 22. In particular, this condition implies that the body must not include any potentially blocking statements (e.g. send/receive statements). Although the assignment `j=0` violates Condition 4 of Definition 22, since the loop is enclosed in an `atomic` block the value of j is not visible until after execution of the loop. At this point j is assigned to A, which SymmSpin uses as a default value for a scalarset variable within this range. Thus symmetry between scalarset values is not broken by the assignment `j=0`.

Figure 3.3 illustrates how scalarsets can be used to specify symmetry in the Promela mutual exclusion example of Figure 2.6.[1] In this specification, a constant

1. Note that in order to avoid modification of the Promela parser, SymmSpin requires information about scalarsets to be supplied in a separate file [14] (see Section 3.9.1). For simplicity and readability,

```
mtype {N,T,C}

const A = 5;
scalar PID[A];

mtype st[PID] = N;

proctype user(PID i) {
  PID j = A;
  bool critical_empty = false;

  do
    :: d_step { st[i]==N -> st[i]=T }
    :: d_step { st[i]==T ->
  critical_empty = true;
j = 0;
do
  :: j < A -> critical_empty = critical_empty && st[j]!=C;
     j++
  :: else -> break;
od;
j = A;
        if
  :: critical_empty -> st[i]=C
  :: else -> skip;
fi;
critical_empty = false;
       }
    :: d_step { st[i]==C -> st[i]=N }
  od
}

init {
  PID i = A;
  atomic {
    i = 0;
    do
      :: i<A -> run user(i); i++
      :: else -> break;
    od
  }
}
```

Figure 3.3: Identifying symmetry in a mutual exclusion example using scalarsets.

A is defined to represent the number of *user* processes in the specification, and a scalarset of size A named `PID` is introduced via the declaration `scalar PID[A]`. Since A is set to 5, the `PID` type is a scalarset with range $\{0, 1, 2, 3, 4\}$. Rather than using the built in `_pid` constant, the *user* proctype now takes a parameter i, which has type `PID`. The array *st* of `mtype` values is indexed by the `PID` type, indicated by the occurrence of `PID` in the declaration of *st*.

It is necessary to modify the syntax of the original Promela specification to satisfy the conditions of Definition 22. In particular, the boolean expression of Figure 2.6 (with re-indexing) `(st[0]!=C && st[1]!=C && st[2]!=C && st[3]!=C && st[4]!=C)` does not obey Condition 1 – literal values cannot be used to index an array of scalarset type. The modified specification computes the expression using a `do..od` loop (as described above), storing the result in boolean

we have included this information in the specification.

variable, `critical_empty`. Only if this variable evaluates to *true* can the process enter the critical state. Note that re-modelling the mutual exclusion protocol this way changes the semantics of the underlying model slightly. We discuss this in Section 4.2.

The use of scalarsets described above exploits component symmetry (symmetry between the processes themselves). The scalarset approach can also be used to exploit *data symmetry*. A scalarset that is used to denote data symmetry is referred to as a *data scalarset*.

Definition 23 *A scalarset $\mathcal{S}$ is a* data scalarset *in a specification $\mathcal{P}$ if $\mathcal{S}$ is not used as an array index or* **for** *statement index.*

If a protocol uses a data scalarset, then it is said to be *data independent* [180]. In this case, symmetry reduction can be used to reduce an infinite state space (in which data is unbounded) to a finite state space (with bounded data) thus:

Theorem 7 *If $\mathcal{P}$ is a specification, $\mathcal{S}$ the name of a data scalarset in $\mathcal{P}$ and $\mathcal{P}_1$ and $\mathcal{P}_2$ are specifications identical to $\mathcal{P}$ except that $\mathcal{S}$ is declared to be of size N_1 in $\mathcal{P}_1$ and N_2 in $\mathcal{P}_2$, then there exists $N_{\mathcal{S}} > 0$ such that the symmetry-reduced state graphs of $\mathcal{P}_1$ and $\mathcal{P}_2$ are isomorphic whenever $N_1 \geq N_{\mathcal{S}}$ and $N_2 \geq N_{\mathcal{S}}$.*

However this application of scalarsets is seldom required as data abstraction (see Section 2.6.3) can be used to eliminate redundant data values [30]. Data symmetry reduction will be discussed again in Section 3.8.

The original scalarset approach [102] only considered the verification of simple safety properties of the form $\mathbf{AG}(\neg error)$. However, scalarsets have been successfully used to exploit symmetry during the verification of more general *LTL* formulas [14]. A major drawback to scalarsets is that they only allow the specification of *total* (or *full*) symmetries (where all processes of a given type are interchangeable), so could be applied to a system of processes connected as a clique, say, but not, for example, as a ring or tree. An alternative data type, called *circularset* [100] and additional extensions to the scalarset data type [49] have been proposed to handle systems with ring structure and more general systems respectively. However, these alternatives share with the scalarset approach the problem that the user must identify symmetry in the model and select an appropriate data type to specify the presence of this symmetry. This means that symmetry reduction using scalarsets is not a "push button" reduction technique.

We say that symmetries of a model are *specified* using scalarsets, since given a specification with scalarset annotations it is trivial to determine a symmetry group for the underlying model.

```
Program

Module user = 5;

st[user] = 0;

u of user: {
    st[u] == 0 -> st[u] = 1;
    st[u] == 1 & ALL(v of user: st[v] != 2) -> st[u] = 2;
    st[u] == 2 -> st[u] = 0;
   }
```

Figure 3.4: An SMC specification of mutual exclusion with five processes.

3.3.3 Input language restriction

The problem of detecting structural symmetry between components can be made trivial by restricting the input specification language in such a way that full symmetry is guaranteed between processes which have the same type.

This is the approach to symmetry detection used by the SMC tool [163] (see Section 3.9.1). Figure 3.4 shows the mutual exclusion example with five processes, expressed in the SMC language. The values 0, 1 and 2 are used to represent the local states N, T and C respectively. Note that the boolean expression to check that the critical section is empty is expressed succinctly as `ALL(v of client: st[v] != 2)`. This expression preserves the semantics of the original Promela specification.

The model associated with an SMC specification with m module types and k_i instantiations of module type i $(1 \leq i \leq m)$ is guaranteed to have $k_1! \times k_2! \times \cdots \times k_m!$ component symmetries. The corresponding group of symmetries is the disjoint product (see Section 3.1.4) of groups S_{k_i} $(1 \leq i \leq m)$, each of which permutes one set of module indices. The modules of SMC are essentially analogous to the scalarsets of Murϕ [163], and *index* variables (variables which take module indices as values) must satisfy similar conditions to those of Definition 22. Only full symmetry between components of the same type can be identified using the SMC language.

We say that symmetries of a model are *specified* using a restricted input language since, by declaring multiple instances of a given module type, the user indicates the presence of symmetry and it is then trivial to determine the corresponding group of permutations.

Capturing symmetry by input language restriction is crucial to the approach of exploiting symmetry using *generic representatives* [45, 57, 58, 60] (see Section 3.5.2).

3.3.4 Communication structure analysis

Let $\mathcal{I} = \{1, 2, \ldots, n\}$ be a set of process identifiers, for some $n > 0$. For simple concurrent specifications consisting of a finite number of isomorphic (identical up to renaming) processes executing in parallel, communicating via shared variables, a subgroup of the automorphism group of $\mathcal{M}$ can be determined from the *communication relation* of the specification [55]. The *communication relation* CR of the specification $\mathcal{P} =||_{i\in\mathcal{I}}\ p_i$ is defined as the undirected graph $CR = (\mathcal{I}, E)$, where $\{i, j\} \in E$ iff processes p_i and p_j share a variable ($i, j \in \mathcal{I}$).

Theorem 8 *If $\mathcal{M}$ is the Kripke structure associated with $\mathcal{P} = ||_{i\in\mathcal{I}} p_i$ where all p_i are normal and isomorphic (see [55]) then $Aut(CR) \leq Aut(\mathcal{M})$.*

The group $Aut(CR)$ may be automatically computed since CR is typically small compared to $\mathcal{M}$, or may simply be known in advance.

Theorem 8 applies to systems in which all variables are shared between at most two processes, and all processes are of the same type. This result is generalised [27] to remove this restriction via the introduction of the *coloured hypergraph* $HG(\mathcal{P})$ of a shared variable specification $\mathcal{P}$ (see Section 3.1.5). The node set of the hypergraph $HG(\mathcal{P})$ is $\mathcal{I}$ and there is a hyper edge $w \subseteq \mathcal{I}$ if the specification $\mathcal{P}$ has a variable shared by all process p_i, $i \in w$. Each node is assigned a colour, so that two processes p_i and p_j are *isomorphic* iff nodes i and j have the same colour in the coloured hypergraph. Two processes are isomorphic in this case if they are of the same process type, and have equivalent sets of transitions.

Theorem 9 *If $\mathcal{M}$ is the Kripke structure associated with a specification $\mathcal{P} = ||_{i\in\mathcal{I}} p_i$ then $Aut(HG(\mathcal{P})) \leq Aut(\mathcal{M})$.*

In Chapter 7 we prove a similar result (Theorem 13) which shows that Kripke structure automorphisms can be derived by computing automorphisms of the *static channel diagram* of a specification which assumes a message passing model of computation.

3.3.5 A note on the complexity of automatic symmetry detection

As noted at the start of Section 3.3, for symmetry reduction to be useful it must be possible to determine symmetries of the model $\mathcal{M}$ associated with a specification without actually constructing $\mathcal{M}$. Ideally we would like an automatic symmetry detection technique to compute all symmetries of $\mathcal{M}$ by static analysis of a specification $\mathcal{P}$ without placing restrictions on the form of $\mathcal{P}$. However, in order to avoid complexity equivalent to that of checking a safety property of the form **AG** p (for some proposition p) it is *necessary* to restrict the form of specifications, or to reject certain potential symmetries which cannot be efficiently checked as belonging to $Aut(\mathcal{M})$. We illustrate why this is the case using a simple Promela example.

Let $\mathcal{P}$ be a Promela specification consisting of n instantiations of a *user* proctype, for some $n > 1$. Suppose that each *user* has a local variable, *special*, which is

```
breaksym:
       if
         :: _pid==1 && <expr> ->
            special = 1
         :: else -> skip
       fi;
```

Figure 3.5: Promela example to illustrate the general complexity of automatic symmetry detection.

set to 0 on declaration. We use $special_i$ to denote the local variable *special* of process i. Suppose that the body of every *user* process contains the conditional statement shown in Figure 3.5, where $\langle expr \rangle$ is an arbitrary boolean expression (which may refer to global variables and channels of the specification). Suppose the statement `special = 1` is the only assignment to `special` (after declaration), that appears in the definition of a *user*, and `_pid==1` the only guard which treats process identifiers asymmetrically. Clearly this statement cannot be executed by a *user* with `_pid` not equal to 1. Assume that the rest of the specification does not differentiate individual processes in any way.

If it is impossible for *user* 1 to reach `breaksym` with $\langle expr \rangle$ evaluating to true, then clearly any permutation of process identifiers will induce an automorphism on the associated model $\mathcal{M}$. Otherwise, *user* 1 will be able to execute the statement `special = 1`, leading to a state s with $s \models (special_1 = 1)$. By the above discussion it is clear that, for any $i > 1$ and any reachable state t in $\mathcal{M}$, $t \not\models (special_i = 1)$. Therefore any process permutation which does not fix 1 cannot induce an automorphism on $\mathcal{M}$. If α is such a permutation then determining whether α induces an automorphism of $\mathcal{M}$ is equivalent to checking the temporal property $\mathbf{AG}(\neg(user[1]@breaksym \wedge \langle expr \rangle))$.

Thus an effective symmetry detection technique for Promela must either restrict the use of the specification language so that this asymmetric use of process identifiers is not allowed, or conservatively assume that certain process permutations do not induce automorphisms of $\mathcal{M}$. In Chapters 7 and 8 we develop an efficient automatic symmetry detection technique for Promela which aims to place as few restrictions as possible on the form of a specification, and to detect a large subgroup of $Aut(\mathcal{M})$ in practice.

3.4 Exploiting Symmetry with a Simple Model of Computation

The crux of exploiting symmetry when model checking is that during search, when a state t is reached, it is necessary to test whether a state u has already been reached such that $t \equiv_G u$ (i.e. $t = \alpha(u)$ for some $\alpha \in G$). This is known as the *orbit problem* [31], and is central to all model checking methods that exploit symmetry. Techniques must be used to either solve the orbit problem efficiently, or to find some kind of approximate solution.

On encountering a state t, Algorithm 1 checks whether there is some $\alpha \in G$ such that $\alpha(t) \in S_G$ by checking whether $rep_G(t) \in S_G$, where rep is a function which computes a unique representative of $[t]_G$. Since the algorithm only stores representative states, if some state u with $u \equiv_G t$ has been encountered then $rep_G(u) \in S_G$. Since $rep_G(u) = rep_G(t)$ (which follows from $[u]_G = [t]_G$), the test $rep_G(t) \in S_G$ returns true and search can backtrack.

For simple concurrent systems it is common to reason about states using a single integer variable for each component, representing the valuation of the local variables of the component [27, 57, 59] (we justify this formally in Section 9.1). Using the notation preceding Definition 1 (Section 2.2), we have a set $V = \{v_1, v_2, \ldots, v_n\}$ of variables, where for each i the domain D_i of v_i is a finite set $L \subset \mathbb{Z}$. A state s is then a vector in L^n. An element $\alpha \in S_n$ acts naturally on a state $s = (x_1, x_2, \ldots, x_n) \in L^n$ as follows: $\alpha(s) = (x_{\alpha^{-1}(1)}, x_{\alpha^{-1}(2)}, \ldots, x_{\alpha^{-1}(n)})$.[2]

Let $\leq$ denote the usual lexicographic ordering on vectors in L^n: for $s, t \in L^n$ where $s = (x_1, x_2, \ldots, x_n)$, $t = (y_1, y_2, \ldots, y_n)$, $s \leq t$ if $s = t$ or there is some $1 \leq i \leq n$ such that $x_j = y_j$ for each $1 \leq j < i$, and $x_i < y_i$. When attempting to exploit symmetry with this model of computation, it is convenient to use the lexicographically least element in the orbit as a representative.

Definition 24 The Constructive Orbit Problem (COP) [27, 105] *Given a group $G \leq S_n$ and a state $s \in L^n$, find the lexicographically least element in the orbit of s.*

In other words, the COP is the problem of computing $min_{\leq}[s]_G$.

Theorem 10 [27, 105] *The COP is NP-hard.*

Despite this discouraging result, it has been shown that the COP can be solved efficiently for certain classes of symmetry group. Furthermore, it may be possible to efficiently compute an approximate solution to the COP, resulting in a quotient model which uses multiple representatives from each orbit.

3.4.1 "Easy" classes of groups

For the following classes of automorphism group G (acting on a model of a system of n processes), the constructive orbit problem (COP) can be solved in polynomial time [27, 105]:

- G has order polynomial in n, for example a cyclic or dihedral group, or the group associated with an $n \times n$ torus
- G is the full symmetric group S_n

2. This action makes sense provided the local state of a component does not include variables which take component identifiers as values. In Chapter 10 we consider this more complex case.

- G is a disjoint product or wreath product of groups for which the COP is polynomial time solvable
- G is generated by transpositions.

Note that, when G is the full symmetric group S_n, the lexicographically least (lex-least) element of the orbit of a state can be obtained by sorting the state-vector. When G has order polynomial in n the COP can be solved by enumerating the orbit of a state. In the other cases, the lex-least element is found by sorting segments of the state-vector individually. This is discussed further in Chapter 9.

3.4.2 Multiple representatives

Requiring that every element of a given orbit $[s]_G$ is mapped to the same representative of $[s]_G$ ensures that symmetry reduction is optimal in terms of space requirements. However, if unique representatives cannot be efficiently computed then the time requirements for model checking with symmetry may be prohibitive.

Suppose we relax the uniqueness condition so that, for an orbit $[s]_G$, any state in $[s]_G$ is mapped to one of a small set of representatives:

$$rep_G([s]_G) = \{rep_G(t) : t \in [s]_G\} \subseteq [s]_G.$$

This ensures that a quotient structure includes *at least one* state from each orbit $[s]_G$, and symmetry reduction is sound (i.e. Theorem 5 still holds). Provided the sets of representatives are sufficiently small compared to the orbits themselves, symmetry reduction can still be effective.

Clearly we can no longer select the minimal element of $[s]_G$ as $rep_G(s)$. However, we can often compute representatives of individual states very efficiently by choosing $rep_G(s) = f(s)$ where $f : S \to S$ is a *normalisation function* which maps all states to states no larger than themselves. A good normalisation function is one for which, for all s, $f(s)$ is "almost" the minimum state in $[s]_G$. Using a normalisation function in this way provides an *approximate* solution to the COP.

Symmetry reduction options which use multiple orbit representatives are provided by Murϕ and SymmSpin, as discussed in the next section. We propose a general approximate solution to the COP based on local search in Chapter 9.

3.4.3 Using orbit representatives in practice

The scalarset method [102] assumes the existence of a canonicalisation function (in which states are replaced by a *unique* equivalence class representative) or a normalisation function (in which a subset of states are used as *multiple* representative states). For symmetry reduction in Murϕ a suitable canonicalisation function [101] applies all permutations to a state s and returns the lexicographically smallest image. An approach using a normalisation function is also suggested, in which the state-vector is split into two parts. For a given state, a permutation α is selected that produces the lexicographically smallest image of the first (most significant) part of the associated state vector. The representative state chosen is the concatenation of the image of the two parts of the state vector (under α).

The use of normalisation and canonicalisation functions with scalarsets is extended [14] using heuristics to choose the order in which variables are positioned in the state-vector. This ordering determines, for example, which variables are most significant and appear in the first (leftmost) part of the split state-vector. One approach, the *sorted strategy*, involves the identification of an array indexed by a scalarset type (the *main array*) and placing it in the leftmost position of the state-vector. In another approach, the *segmented* strategy, the lexicographically smallest image of the second part of the state-vector, with respect to all of the permutations that canonicalise the first part, is used in the representative state. There is trade off between reduction in memory requirements and faster verification for the *sorted* and *segmented* strategies. The segmented strategy yields canonical representatives, but is more computationally expensive than the *sorted* strategy. On the other hand, use of the *sorted* strategy may result in several states from the same equivalence class being explored. In Chapter 9 we generalise the *segmented* strategy to apply to arbitrary symmetry groups.

Two other approaches, *pc-sorted* and *pc-segmented*, are suggested for systems in which no suitable main array exists but the process identities are of type scalarset. In this case a main array is constructed, containing the program counters. A prototype implementation of this approach is implemented in the SymmSpin package [14], which we discuss in Section 3.9.1.

A canonicalisation function is suggested, again within the context of SPIN [133], for systems with any (user-specified) symmetry. Though less restrictive than the scalar-set approach (full symmetry is not required and more general operations on permutable variables are permitted), a unique canonicalisation function must be constructed manually by the modeller for every individual model, thereby limiting the applicability of the method.

3.5 Combining Symmetry Reduction with Symbolic Representation

If BDDs are used to represent the state-space of a model then exploiting symmetry using the approach described in Section 3.2 becomes more complex, as the *orbit relation* of a symmetry group must be represented as a BDD. The orbit relation of a group G is the set of pairs $\{(s,t) : s,t \in S,\ s \equiv_G t\}$. It can be shown that the minimum size of a BDD representing the orbit relation induced by a transitive group (a group acting on a set such that every element of the set is moved by some element of the group) is exponential in the minimum of the number of components in a system and the number of states in one component [31]. The result is extended to the class of *separable* groups, which subsumes the class of transitive groups [105]. Since transitive groups occur commonly in models of concurrent systems, the combination of standard symmetry reduction techniques with symbolic model checking is limited. We now discuss some methods which avoid the construction of the orbit relation for symbolic model checking.

3.5.1 Multiple representatives and symbolic model checking

By using multiple representatives from each orbit, the problem of computing the orbit relation can be avoided to some extent [27, 31]. The idea is similar to the idea of using multiple representatives discussed in Section 3.4.2, but depends on a specific subset of automorphisms. If G is a set of Kripke structure automorphisms, a subset C of G is chosen which is closed under inverses and contains the identity element. The set of representatives *Rep* is selected so that each orbit (of the set of states S under G) has at least one element in *Rep* and, for every $s \in S$, there is some $\alpha \in C$ such that $\alpha(s) \in Rep$. The size of *Rep* (and consequently the size of the resulting quotient model) depends heavily on the choice of C. Hence C must be chosen carefully according to the structure of the system being verified [27]. The state-space of the quotient model is not reduced (with respect to the original model) as much as with unique representatives. However, the use of multiple representatives reduces the size of the BDDs required to store the state-space, and thus are more effective when symbolic representation of states is used.

In practice, BDDs reduced through multiple representatives may still be intractably large. Approaches using generic representatives or computing representatives dynamically, which we discuss in Sections 3.5.2 and 3.5.3 respectively, have been shown to outperform the multiple representatives approach [58, 59].

3.5.2 Generic representatives

For symbolic model checking of fully symmetric systems using BDDs, a method which uses *generic* representatives avoids both the orbit problem and construction of the orbit relation [57]. This method involves translating the specification for a model into a reduced specification, which can be explored using standard symbolic model checking algorithms. The idea of generic representatives is best explained using an example. For a basic model of mutual exclusion with three processes, the states $(N_1N_2T_3), (T_1N_2N_3)$ and $(N_1T_2N_3)$ are all equivalent. This is because there are two processes in the *neutral* local state and one in the *trying* local state in each of the three global states. The generic representative of these states is $(2N\,1T)$. A generic representative indicates *how many* processes are in each local state, but does not refer to individual processes. Thus the reduced specification abstracts from processes to *counters*, with one counter for each local state, indicating the number of processes currently in that state.

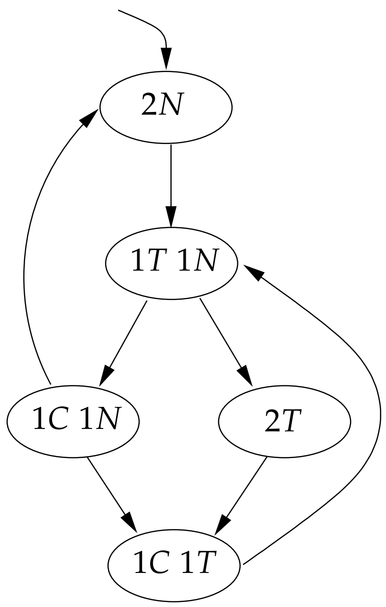


Figure 3.6: Symmetry-reduced model for two-process mutual exclusion using generic representatives.

```
byte no_N=5;
byte no_T=0;
byte no_C=0;

init {
  do
    :: d_step { no_N>0 -> no_N--; no_T++ }
    :: d_step { no_T>0 && no_C==0 -> no_T--; no_C++ }
    :: d_step { no_C>0 -> no_C--; no_N++ }
  od
}
```

Figure 3.7: Generic form of five-process mutual exclusion.

The translation rules defined for fully symmetric specifications (where symmetry is guaranteed by input language restriction) ensure that the state-space of the translated specification is isomorphic to the quotient structure associated with the original specification. Figure 3.6 shows the Kripke structure for two-process mutual exclusion using generic representatives. Notice that the structure is identical to the quotient structure of Figure 3.1, except for the change of variables.

Figure 3.7 shows the translated version of the five-process mutual exclusion specification of Figure 2.6. The associated Kripke structure, shown in Figure 3.8, has 11 states, whereas the original specification has a state-space of size 112.

The generic representatives approach is extended [58] to include systems with global shared variables. The translation of a specification into reduced form is polynomial in the length of the specification and the approach compares well to those using unique or multiple representatives. However, benefits of this approach can be negated due to the *local state explosion* problem, where the number of potential local states of a process is exponential in the number of local variables.

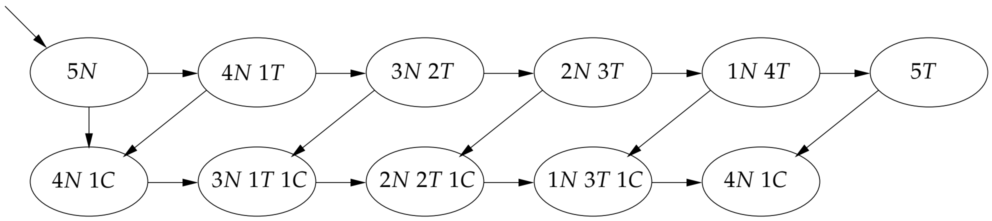


Figure 3.8: Kripke structure associated with the specification of Figure 3.7.

Since the reduced specification requires one counter per local state, BDD representations which require bits to be reserved for each counter become infeasible. Techniques have been proposed to limit local state explosion based on *live variable analysis* (similar to the data flow optimisations provided by SPIN [92]) and *local reachability analysis* [60]. The generic representatives approach is also limited as it only applies to fully symmetric systems which are simple enough to be amenable to counter abstraction [59].

The generic representatives approach has been applied to provide symmetry reduction for probabilistic symbolic model checking [45] (see Section 3.9.3).

3.5.3 Dynamic computation of representatives

Another approach to combining symmetry reduction techniques with symbolic representation (for *CTL* model checking) involves determining orbit representatives dynamically during fixpoint iterations [59]. Instead of building the orbit relation for a model, this approach works by computing transition images with respect to the unreduced structure, then mapping the new states to their respective representatives. This approach is not restricted to fully symmetric systems, and can handle data symmetry (see Section 3.8) as well as process symmetry. A potential bottleneck here is the operation of swapping bits in the BDD representation of the model, which must be performed repeatedly during representative computation. The complexity of such swaps depends exponentially on the distance, in the BDD variable ordering, between the variables to be swapped. To avoid this problem, permutations are expressed as a product of transpositions of *adjacent* elements. Experimental results show that this approach outperforms the use of multiple and generic representatives (see Sections 3.5.1 and 3.5.2 respectively) when applied to a queueing lock algorithm and a buggy version of a cache coherence protocol.

3.5.4 Under-approximation

Model checking algorithms that use depth-first search (DFS) can be adapted so that the first element of an orbit encountered during the search is chosen as the orbit representative [163]. However, this approach is not suitable for symbolic model checking techniques as DFS is very inefficient in the context of BDD state representation.

On-the-fly orbit representative selection is possible during symbolic reachability analysis by combining symmetry reduction with a technique known as *under-approximation* [7], where only a subset of reachable states is considered at each step of symbolic verification. This approach to symmetry reduction can be used for *falsification* of temporal properties, but cannot generally provide verification.

3.6 Combining Symmetry Reduction with Other Techniques

Basic symmetry reduction does not take into account the more sophisticated techniques associated with model checking. In this section we discuss the combination of symmetry and partial order reduction, and the modification of symmetry reduction techniques to successfully handle fairness.

3.6.1 Symmetry and partial order reduction

Partial order reduction (see Section 2.6.3) and symmetry reduction are orthogonal reduction techniques. They can therefore be successfully used in conjunction, resulting in larger savings in memory and verification time.

The combination of the two techniques was first suggested in the context of Petri nets [172]. This approach applies to the stubborn sets method of partial order reduction and is restricted to deadlock detection.

The idea of combining two reductions simultaneously is extended to verifying next-time free *LTL* properties via model checking [53]. Indeed, an algorithm is given combining partial-order reduction and any bisimulation preserving equivalence. When the equivalence is the automorphism relation, the algorithm proceeds as follows: from any state s an ample set of transitions is calculated. The orbit representatives of any states reachable via these transitions are then explored. A similar algorithm, combining the persistent sets method of partial order reduction with symmetry reduction is used within the stateless search technique implemented in VeriSoft [69] (see Section 3.9.4).

3.6.2 Exploiting symmetry under fairness assumptions

Fairness is vital for proving liveness properties, as it reflects the basic requirement that processes are executing at an indefinite but positive speed [56]. Two important kinds of fairness are *weak* fairness and *strong* fairness. Given a Kripke structure $\mathcal{M}$, an infinite path π of $\mathcal{M}$ is *strongly fair* if each process that is enabled infinitely often is executed infinitely often. A path π is *weakly fair* if any process that is continuously enabled is executed infinitely often.

Fairness is generally incompatible with basic symmetry reduction methods because the progress of an individual process along a path of the quotient structure cannot be tracked in the usual way. For example, consider the transition $(C, T) \rightarrow$

(N, T) in the two-process mutex Kripke structure (Figure 2.3) which results from process 1 leaving the critical section. The transition is represented in the quotient Kripke structure (Figure 3.1) by $(C, T) \rightarrow (T, N)$. The quotient transition indicates that one of the processes leaves the critical section, but there is no information as to *which* process this is.

Each state of the quotient structure is labelled according to the representative of an equivalence class of states in the original structure, and for a transition $s \rightarrow t$ in the original model, it may not be the case that transition $rep_G(s) \rightarrow rep_G(t)$ also occurs in the quotient model.

These fundamental problems are overcome when the automata theoretic approach using annotated quotient structures is used, in the context of (inherently symmetric) fair indexed $CTL^\star$ properties [56, 161]. An annotated quotient structure $\mathcal{M}_G$ is used together with an automaton $\mathcal{A}$ which accepts only fair computations. An efficient algorithm, based on finding maximal strongly connected components (MSCCs) [169] (see Section 2.3.2) is presented for model-checking fair indexed $CTL^\star$ formulas under the assumption of strong and (by implication) weak fairness. Correctness results (including liveness properties) are verified for a resource controller example using a prototype (fair) model checker. Comparison with an unreduced model indicates an exponential reduction in the number of stored states.

This approach to symmetry reduced model checking has been extended to the on-the-fly case [74] in which $\mathcal{M}_G \times \mathcal{A}$ is checked during construction. The approach also exploits *state symmetries* [55]. A state symmetry of a state s is a permutation $\alpha \in Aut(\mathcal{M})$ on process indices such that $\alpha(s) = s$. If processes i and j have the same local state in global state s, and if $\alpha(i) = j$, then only the transitions made from state s by process i need to be considered, saving space and computation time. The resulting algorithm is exploited in SMC [163], which we discuss in Section 3.9.1.

A parallel approach to model checking with symmetry reduction and weak fairness [13] combines the weak fairness algorithm implemented in SPIN [92] (based on the Choeka flag algorithm [24]) with a symmetry reduction algorithm [12] based on the nested depth first search (NDFS) approach to model checking [93]. As well as exploiting the usual advantages over the MSCC algorithms, the NDFS approach is compatible with approximate verification techniques, such as the hash-compact method and supertrace verification (see section 2.6.2).

3.7 Exploiting Symmetry in Less Symmetric Systems

Many systems which occur commonly in practice are comprised of several *similar*, but not all identical processes. An example is the readers-writers problem [57], where *m reader* processes and *n writer* processes access a shared resource, for some $m, n > 0$.

A writer always has priority over a reader when both are trying to access the shared resource. If $\mathcal{M}$ is a model of this system, then $\mathcal{M}$ is not fully symmetric. In fact $Aut(\mathcal{M})$ is $S_{\{1,2,\ldots,m\}} \times S_{\{m+1,m+2,\ldots,m+n\}}$ – readers can be permuted, writers can be permuted, but readers cannot be interchanged with writers.[3] 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.

It is possible to exploit this kind of *almost symmetry* during model checking. Indeed, by defining different classes of symmetry, such as *near* or *rough symmetry* [57], or *virtual symmetry* [52], it is still possible to infer temporal logic properties of the system by model checking a suitable quotient graph using the entire group S_{m+n} as the automorphism group.

3.7.1 Near and rough symmetry

Suppose $\mathcal{M}$ is a model of a system, and $\mathcal{I}$ the set of process identifiers associated with $\mathcal{M}$. Then a permutation $\alpha \in Sym\ \mathcal{I}$ is said to be a *near* automorphism of $\mathcal{M}$ if, for every transition $s \rightarrow t$ of $\mathcal{M}$, either $\alpha(s) \rightarrow \alpha(t)$ is a transition of $\mathcal{M}$ or s is totally symmetric with respect to $Aut(\mathcal{M})$. (That is, s is invariant under $Aut(\mathcal{M})$.) The model $\mathcal{M}$ is said to be *nearly symmetric* if it has a suitable group of near automorphisms G_n.

If, on the other hand, G_r is a subgroup of $Sym\ \mathcal{I}$, then $\mathcal{M}$ is *roughly* symmetric with respect to G_r if for every pair of states s and s' where $s \equiv_{G_r} s'$, any transition from s is matched by a transition from s' provided the associated local transition (from s') would involve a process with the highest priority. If $\mathcal{M}$ is a *nearly* (*roughly*) symmetric model with respect to group G_n (G_r) then, despite the lack of complete symmetry, it can be shown that symmetry reduction with respect to G_n (G_r) preserves all symmetric $CTL^{\star}$ properties [57].

3.7.2 Virtual symmetry

Both near and rough symmetry are subsumed by the notions of *virtual* and *strong virtual* symmetry [52]. As well as systems with static priorities (which can already be described via *rough* symmetry) virtual symmetry applies to systems where resources are asymmetrically shared according to dynamic priorities.

The symmetrisation R^G of a transition relation R by a group G is defined by:

$$R^G = \{\alpha(s) \rightarrow \alpha(t) : \alpha \in G \text{ and } s \rightarrow t \in R\}.$$

Intuitively, symmetrising a transition relation can be thought of as the process of adding transitions which are missing due to asymmetry in the system.

A structure $\mathcal{M}$ is 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 $s \rightarrow \alpha(t) \in R$. In

3. Assuming there are no symmetries other than those which permute process ids.

addition, if for any $s \to t \in R^{G_v}$, there exists α in $\text{Fix}(s, G_v)$ (the largest subgroup of G_v which fixes s) such that $s \to \alpha(t) \in R$, then $\mathcal{M}$ is said to be *strongly virtually symmetric* with respect to G_v. If a Kripke structure $\mathcal{M}$ is (strongly) virtually symmetric with respect to a group G_v, model checking of symmetric properties can be performed over $\mathcal{M}_{G_v}$ [52]. A procedure is given to identify the case where a Kripke structure is strongly virtually symmetric with respect to a group G_v. This procedure involves local counting of transitions which are present in R^{G_v} but absent in R. Virtual symmetry has been successfully combined with the generic representatives approach (see Section 3.5.2) for the case where processes are fully interchangeable with respect to virtual symmetry [179]. This allows symmetry-reduced symbolic model checking of partially symmetric systems, using the NuSMV model checker [25] (see Section 2.5).

3.7.3 Automata theoretic approaches

A method involving the symmetry reduction of models with little or no symmetry uses guarded annotated quotient structures (GQSs) [161, 162]. These structures are an extension to the annotated quotient structures [55, 56, 74], discussed in Section 3.2. Suppose $\mathcal{M}$ is the Kripke structure of a system, and $\mathcal{M}' \supseteq \mathcal{M}$ is obtained from $\mathcal{M}$ by adding transitions (in a similar manner to the process of symmetrisation described above), so that $\mathcal{M}'$ has more symmetry than $\mathcal{M}$. A guarded annotated quotient structure for $\mathcal{M}$ can be viewed as an annotated quotient structure for $\mathcal{M}'$, with edges labelled to indicate which processes can make the transition (in $\mathcal{M}$). Thus the original edges of $\mathcal{M}$ can be recovered from the representation of $\mathcal{M}'$. A temporal property ϕ can be checked over the guarded annotated quotient structure by unwinding the structure, even if ϕ is not symmetric with respect to the automorphisms used for reduction. This approach potentially allows large factors of reduction to be obtained since a larger group of automorphisms is used than would be possible using standard quotient structure reduction. Indeed, experimental results, using the SMC model checker [163], show how the GQS method is applied effectively to a system of prioritised processes.

A recent extension to the GQS approach [164] involves (symmetry reduced) model checking of *extended CTL* (*CCTL*) properties (which involve an additional construct, $COUNT$, for specifying the number of components in a given state). This extended logic is more expressive than indexed CTL (see Section 3.2).

Properties are again not restricted to being fully symmetric in an alternative automata theoretic approach [2], but must be *partially symmetric*. For example, consider the following property: "if some process is waiting for a resource then it will get it, provided none of the processes with higher identity will require the resource in the future". To check the satisfaction of a formula ϕ for a model $\mathcal{M}$, with set of states S, a set of equivalence relations are first computed between states of $\mathcal{B}$, the Büchi automaton representing ϕ. If G is a symmetry group of $\mathcal{M}$, one equivalence relation is defined for every element of G. Two states $b_1, b_2 \in B$ are equivalent with respect to $\alpha \in G$ if and only if the predecessors and successors of b_1 are mapped to the predecessors of b_2 and the successors of b_2 respectively (and vice versa). The quotient graph is then constructed by applying the equivalence relations to the pairs of states $(s, b) \in S \times \mathcal{B}$. The approach is extended [75] to partially symmetric *models* by representing the model itself as the synchronised product of a symmetric model and an asymmetric Büchi automaton. The method is illustrated using well-formed Petri nets.

3.8 Exploiting Data Symmetry

Most of the symmetry reduction methods described in this paper relate to structural symmetry. However, as discussed in Section 3.3, another form of symmetry, namely data symmetry, can be exploited to increase the effectiveness of model checking. In Section 3.3 we discussed the application of scalarsets to exploit data symmetries.

As software specifications often involve large data structures with vast numbers of values, it may be impossible to check that properties hold for every feasible assignment of values to the data set. That is, it may not be possible to check the properties for every interpretation of the model. It is therefore desirable to only check representative models for each equivalence class of interpretations. This use of data equivalence is exploited for software analysis using the Nitpick specification tool [103].

3.9 Implementations of Symmetry Reduction

In this section we list the major tools for which symmetry reduction has been implemented. This is not intended as an exhaustive exposition, but as a selective illustration.

3.9.1 Explicit state methods

Murϕ

The Murϕ specification language is the first language to have been augmented with the scalarset data type (see Section 3.3). As a result, the Murϕ verification system [40] is the first to implement symmetry reduction using scalarsets [102] and has inspired many of the other implementations discussed in this section.

An automorphism group for the state-space is determined statically from the Murϕ specification and consists of all permutations of scalarset values. The lexicographically smallest member of each orbit is used as the orbit representative and a suitable canonicalisation function (see Section 3.4.3) is used to map every state to its orbit representative.

Murϕ has been used to verify a number of highly symmetric algorithms (for example Peterson's n-process mutual exclusion algorithm [138] (see Section 4.3)) and a lock implementation for the Stanford DASH multiprocessor [116].

A prototype extension of Murϕ includes two alternative classes of algorithm for representative computation [106]. The first class of algorithms transforms each state encountered during search to a *characteristic graph*, and derives a canonical state representative from the canonical form of this graph. The *nauty* graph isomorphism tool [123] is used to perform canonicalisation operations. The other class of algorithms uses ordered partitions on states, and during canonicalisation considers only permutations which are compatible with the partitioning of a given state. This approach mimics the partitioning approach commonly used by graph isomorphism algorithms [123].

SMC

The Symmetry based Model Checker (SMC) [161, 163] is an explicit state model checker which has been specifically designed for the verification of highly symmetric systems. Exploiting both *process* symmetry and *state* symmetry, in addition to proving safety properties, SMC is the only model checker that can be used to effectively verify liveness properties under both strong and weak fairness assumptions. Symmetry is detected via input language restriction (see Section 3.3.3). Variables in the simple SMC language are either global variables accessed identically by processes of the same type, or arrays indexed by process identifiers and manipulated via universal or existential quantification (see the `ALL` statement of Figure 3.4).

Model checking is performed using a technique [74] involving annotated quotient structures (AQSs) (see Sections 3.2 and 3.6.2). The AQS can be constructed either in advance or on-the-fly. For on-the-fly construction, it is also possible to store the edges of the AQS during construction. If the edges are not stored considerable space savings can be made. However, verification time is increased dramatically.

The AQS is constructed incrementally, and the first state of an orbit encountered during search is used as the representative for that orbit. State symmetries of a state s are detected by partitioning processes within each module into equivalence classes. A *leader* process is chosen from each equivalence class, and only transitions from s made by one of the leader processes are explored. Reached states are stored in a hash-table, and a hashing function is used which *always* hashes equivalent states to the same location, and *desirably* hashes in-equivalent states to different locations. For a state s, the hashing function returns $Checksum(s) \ mod \ b$, where b is the hash-table size. The checksum for a state is computed from the values of variables in that state. Each time a state is to be stored at a position in the hash-table, a check is made to see if the state is equivalent to any other state in that position in the table. Two states with differing checksums cannot be equivalent, so SMC performs the pre-test of comparing checksums before checking the equivalence of two states. In many cases this quickly shows the inequivalence of states.

To check whether two states with equal checksums are equivalent, a polynomial time bounded, randomised algorithm is used which runs in quadratic time. This algorithm sometimes falsely reports that two equivalent states are not equivalent, which may result in the construction of a larger-than-optimal AQS (but is not unsafe – see Section 3.4.2).

SMC has been used to check the correctness of the link layer part of the IEEE standard 1394 *Firewire* protocol [95], and also a resource controller example. The resource controller example shows that exploiting state symmetry can speed up verification considerably when the number of processes is high. Recent extensions of SMC [162, 164] enable partially symmetric systems with priorities to be verified over a GQS, and properties to be expressed in *CCTL* (see Section 3.7.3).

SymmSpin

Symmetric SPIN (SymmSpin) [14] is a symmetry reduction package for the SPIN model checker [92]. To allow process symmetry of a system to be specified, the *scalarset* data type [102] is used. As noted in Section 3.3.2, to avoid modifying the Promela parser, rather than directly extending the Promela language with the scalarset data type, all of the symmetry information is provided (by the user) in a separate file. This is referred to as the *system description file*, and identifies which variables have the scalarset type.

SymmSpin uses a script [84] to modify the verifier generated by SPIN for a given specification (see Section 2.4.2), adding symmetry reduction via a *representative function* which, for a given state, computes an orbit representative for the state. For a given orbit the representative is the least element with respect to a specified canonicalisation function or one of the minimal elements computed via a normalisation function (see Section 3.4.3). During search SymmSpin stores the original states on the stack and representative states on the heap (see Section 2.3.2). This means that counter-example traces generated by SymmSpin correspond to real counter-example traces through the model, rather than the representatives of a counter-example trace.

Experiments using SymmSpin show that for certain models the factor of reduction gained is close to the theoretical limit [14]. These experiments also show that the combination of symmetry and partial order reduction can be effective. A prototype extension of SymmSpin for symmetry reduced model checking under weak fairness [13] has recently been developed. This is discussed in Section 3.6.2.

In Chapter 11 we present a symmetry reduction package which follows the SymmSpin approach of adding symmetry reduction algorithms to the verifier generated by SPIN.

Other SPIN*-based implementations*

An extension to SPIN is proposed [39] to allow symmetry reduction of models of systems of replicated processes. The specification language Promela is augmented with two additional keywords, *ref* and *public* which identify reference variables and local variables with public scope respectively. Unlike scalarsets, these variables may hold the addresses of other processes for communication purposes or represent process ids. Orbit representatives are computed by a process called *pseudo sorting* in which the parts of the state-vector corresponding to the individual processes are sorted lexicographically. As the original state-vector ordering depends on the order in which variables are declared, the efficiency of the sorting algorithm depends on the initial declaration ordering.

3.9.2 Symbolic methods

SMV

As a symbolic model checker, SMV [126] does not lend itself to symmetry reduction of the state-space. This is because the symbolic representation of the orbit relation as a BDD is prohibitively large (see Section 3.4). However, symmetry reduction on the *cases* associated with a property to be proved for a system is achieved via the use of scalarsets [125]. In order to exploit abstraction techniques available with SMV, a method called *temporal case splitting* is used to break a given property down into a parameterised set of assertions. This addresses state explosion, but may result in an unwanted side-effect, namely *case explosion*. Declaring variables as scalarsets enables SMV to sort the assertions into equivalence classes. Specifically, if we have two assertions ϕ_1 and ϕ_2 where ϕ_2 is obtained from ϕ_1 by some permutation of scalarset values, then ϕ_1 holds if and only if ϕ_2 holds. Thus for a given parameterised set of assertions, it is only necessary to check a representative subset of assertions. This representative subset is chosen in such a way that every assertion in the original parameterised set can be mapped to a representative assertion via permutation of scalarset values.

SYMM

One purpose-built symbolic model checker that exploits symmetry reduction methods for the verification of *CTL* specifications is SYMM [27]. SYMM uses a simple input language based on a shared variable model of computation and allows the user to give symmetries of the system to be verified.

To avoid computing the orbit relation, symmetry reduction is implemented using the *multiple orbit representatives* approach (see Section 3.5.1). SYMM has been used to verify the IEEE Futurebus arbiter protocol [96] which controls a number of prioritised components competing for a resource. Each individual process is described via a *module*. Modules with the same priority can be permuted.

Other symbolic implementations

The RuleBase model checker [8] has been experimentally extended with symmetry reduction techniques for *under-approximation* [7] (see Section 3.5.4). Generators for a symmetry group of the verified system are supplied by the user. The generators which are genuine symmetries of the system, and under which the checked property is invariant, are retained by the model checker for exploitation during search. Experimental results show that RuleBase performs significantly better for the checking of liveness properties when symmetry reduction is applied. However, no improvement in performance has been shown for safety properties.

An experimental model checking system, UTOOL [60], has been developed for the investigation of techniques to combine symmetry reduction with symbolic representation. This tool uses the input language of Murϕ and performs symbolic verification, exploiting symmetry wherever possible. UTOOL avoids constructing the orbit relation through the use of generic representatives, or through dynamic representative computation (see Sections 3.5.2 and 3.5.3 respectively). Though less efficient, for the purposes of comparison UTOOL also implements symmetry reduction using pre-computed multiple representatives (see Section 3.5.1).

3.9.3 Real-time and probabilistic methods

UPPAAL

The real time model checking tool, UPPAAL, has been extended to exploit symmetry [78], using scalarsets [102]. As the main purpose of UPPAAL is to perform reachability analysis, symmetry reduction using scalarsets is an obvious choice: the original scalarset theory was developed in the context of reachability analysis rather than the checking of temporal logic properties. However, the soundness of symmetry reduction does not follow directly, since the UPPAAL language is very different from that of Murϕ. Hence soundness is proved separately for UPPAAL.

The implementation of symmetry reduction in UPPAAL involves the development of an efficient algorithm for the computation of a canonical representative for a state. This is particularly challenging since UPPAAL represents sets of clock valuations symbolically using a difference bounded matrix (DBM).

The scalarsets for a given model define a set of *state swaps* for the model. Each state swap is an automorphism of the model, and the set of all state swaps can be used to compute a canonical state representative. In order to simplify the computation of representatives, two assumptions are made. The first is that an array indexed by scalarsets does not contain elements of scalarset type. The second is that a timed automaton in a UPPAAL model may only reset its clock to the value zero. This assumption ensures that individual clocks can always be ordered using the order in which they were reset; this is called the *diagonal property* and leads to a total ordering on states. Note that the diagonal property is important as, for a given total ordering, minimisation using state swaps of a general DBM is at least as hard as testing isomorphism for strongly regular graphs [78]. A state is minimised using the state swaps defined by scalarsets in the model, together with this total ordering. This minimised state is a canonical representative for the original state.

Experimental results for Fischer's mutual exclusion protocol show that exponential savings can be gained by exploiting symmetry. Further experiments for an audio/video protocol and for a distributed agreement algorithm are also encouraging. Since symmetry reduction in UPPAAL makes use of scalarsets, only total symmetries can be exploited.

RED

Another (symbolic) real time model checker to support symmetry reduction is RED [178]. The symmetry reduction algorithm uses relations between pointers to define an ordering among processes. This ordering is then used to compute a representative by sorting the associated orbits. Every permutation is constructed via successive composition of transpositions. This can lead to an over approximation of the reachable state-space (the "anomaly of image false reachability"). For this reason using RED with symmetry reduction is only useful for checking that a state is *not* reachable. The performance of RED (with symmetry reduction) is compared to that of Murϕ [40] (with symmetry reduction) and SMC [163] for three benchmark systems [178]. Since it manages to successfully combine symbolic techniques with symmetry reduction, as the number of processes increases, RED dramatically outperforms the other model checkers.

PRISM

As with standard symbolic model checking, a symmetry reduction technique for probabilistic symbolic model checking must avoid construction of the orbit relation (see Section 3.5). A prototype extension to the PRISM model checker uses an approach based on dynamic representative computation [59] (see Section 3.5.3) to build symmetry-reduced probabilistic models [114]. This approach requires initial construction of the unreduced model as a multi-terminal BDD (MTBDD) which is then reduced using an algorithm based on the bubble sort algorithm.

Although this approach cannot handle models with intractably large MTBDD representations, it is useful in the case where it is possible to build but not verify properties of an unreduced model. This situation occurs due to the additional space overhead associated with probabilistic verification algorithms over standard model checking algorithms. Experimental results for four case studies [114] show that this symmetry reduction technique can speed up model checking and facilitate verification of larger systems. Surprisingly, in certain cases the MTBDD for the quotient model is *larger* than the MTBDD for the unreduced model, due to loss of regularity as a result of permuting rows of the probabilistic transition matrix.

An alternative approach to exploiting symmetry in probabilistic model checking uses generic representatives (see Section 3.5.2) [45]. A fully symmetric PRISM specification is automatically translated into generic form using the GRIP (generic representatives in PRISM) tool. The resulting specification can then be directly checked using PRISM. This method avoids constructing an MTBDD for the original model, so can be applied to larger examples than the techniques of [114].

3.9.4 Direct model checking

*d*SPIN

An on-the-fly state-space exploration algorithm exploiting both process and heap object symmetry in Java programs has been implemented in the dSPIN model checking tool [98]. For dynamic systems modelled using dSPIN, the number of state components may grow along an execution path. Therefore, rather than applying symmetry reduction with respect to a fixed permutation group, a family of groups is considered. A suitable group is selected at each execution step. Orbit representatives are calculated using a similar set of heuristics to those used by SymmSpin.

Bogor

A symmetry reduction technique has been developed for the Bogor model checking framework [146], which is used to model check Java programs. The symmetry reduction methods used in Bogor [147] are based on those implemented in dSPIN, but use more efficient heuristics [99] for state-vector sorting.

States contain both thread and heap information and these different parts of the state (the thread and the heap state) are sorted separately. Threads are sorted by comparing associated program counters which does not always produce a unique ordering, but heap states can be sorted in a canonical way. For every heap state s, there is an associated set of memory locations, $l_{1,s}, l_{2,s}, \ldots, l_{r,s}$ say. It is possible to sort the indices of the memory locations (for a given s) by ordering the *traces* associated with each pair $(s, l_{i,s})$, $1 \leq i \leq r$. The trace for pair $(s, l_{i,s})$ is the smallest of all of the incoming *chains* (pairs of thread identifiers and variable sequences) which can themselves be ordered in a natural way. The sorting of the location indices produces a strictly ordered list of integers. If G is a symmetry group acting on the heap elements, then the ordered list associated with state s is identical to the corresponding list for any s' in the same orbit of G as s. Thus the index sorting function is a canonicalisation function (see Section 3.4.3).

Verisoft

The VeriSoft model checker [68] verifies *C*-code directly via a stateless search. As such, the symmetry reduction methods implemented in VeriSoft [69] rely on equivalences between sequences of *transitions* rather than between states.

In order for equivalent transitions to be identified, labels are added to transitions, so that the model is represented by a labelled transition system. Two transitions are equivalent with respect to a given symmetry group G if their respective labels are equivalent with respect to G. This concept can be easily extended to sequences of transitions. Symmetry reduction is used to prune transitions on-the-fly. If, for some state s and $\alpha \in G$, transitions a and $\alpha(a)$ are enabled at s and α fixes s, then only one of a or $\alpha(a)$ need be explored. This is similar to the notion of *state* symmetry described in Section 3.6.2. Given that s is not stored explicitly, it is not straightforward to check that α fixes s. However, assuming that α fixes the initial state s_0, if w is the sequence of transitions leading from s_0 to s, then it can be shown that $\alpha(s) = s$ if and only if w and $\alpha(w)$ are equivalent with respect to a partial ordering of transitions. Thus, by combining symmetry reduction with partial order reduction techniques (see Section 2.6.3) the problem of checking that $\alpha(s) = s$ is overcome.

Other direct model checking implementations

A limited form of symmetry reduction is applied [117] within the second generation Java PathFinder tool (JPF2) [176] (see Section 2.5) which model checks Java bytecode directly. Like dSPIN, JPF2 is capable of handling dynamic structures (although, unlike dSPIN, data is not allocated dynamically). States are composed of a static area, a dynamic area and a thread area, each of which is represented as an array. Two states are considered to be equivalent if a permutation applied to the static and dynamic area arrays of the first state, gives the corresponding arrays of the second. A canonicalisation function is used which imposes a simple ordering (calculated during model checking) on the static and dynamic areas of the states.

Summary

Symmetry reduction techniques involve avoiding the exploration of areas of the state-space which are symmetrically equivalent to those already visited. We have given an overview of symmetry reduction techniques for model checking and how they relate to other reduction approaches. We have also surveyed implementations of these techniques, both for existing model checkers (e.g. SPIN and Murϕ) and purpose-built checkers (e.g. SMC). This survey identifies three clear areas for research, which we address in the remainder of the thesis.

The *identification* of symmetries involves finding symmetries of a model without building the model explicitly. Detection of full symmetry between identical components can be achieved by annotating a specification with the scalarset data type, or by appropriately restricting the input language. On the other hand, with a shared variable model of computation, component symmetries may be derived via analysis of the communication structure of a high level specification. In Chapters 7 and 8 we develop fully automatic techniques for the detection of arbitrary component symmetries under a message passing model of computation, based on communication structure analysis.

The crux of exploiting symmetry in explicit state model checking is the (constructive) *orbit problem*: it must be solved efficiently, or a good approximate solution must be available. In Chapters 9 – 11 we present exact and approximate strategies for solving the COP for an arbitrary group of component symmetries.

Chapter 4

Analysing Symmetry in Simple Concurrent Systems

In this chapter we introduce a software tool, SPIN-to-GRAPE, which we have developed to allow comprehensive analysis of small state spaces using the computational graph theoretic package GRAPE (see Section 3.1.6). We apply SPIN-to-GRAPE to five example specifications, in each case highlighting the disadvantages of specifying symmetry using scalarsets (with SymmSpin) or via input language restriction (with the SMC language). For the first three examples we emphasise the manual effort which may be required at the specification level to specify symmetry. The subsequent examples exhibit fairly complex symmetry groups which cannot be specified by these techniques. We also use the examples to illustrate the change in symmetry resulting from modifications to the specifications.

These examples motivate the development of *automatic* symmetry detection techniques in Chapters 7 and 8, which can handle arbitrary types of structural symmetry and do not require annotation at the specification level by the user.

4.1 SPIN-to-GRAPE

The SPIN-to-GRAPE tool uses SPIN to construct the state-space associated with a Promela specification and produces a directed graph representation which can be input to the graph theoretic package GRAPE. GRAPE can then be used to compute the automorphism group of the state-space. We briefly describe the algorithm used by SPIN-to-GRAPE.

Among the options which SPIN provides for running verifications on Promela specifications is the *verbose* compile-time directive. This option writes every step of a verification run to standard output. Running a verification to search for invalid end-states on a deadlock-free specification with the *verbose* option, and no partial order reduction (or other options which change the structure of the state-space) selected, results in a textual description of the Kripke structure associated with the specification. In order to manipulate the Kripke structure as a directed graph using GRAPE we have designed a tool, SPIN-to-GRAPE, which takes relevant *verbose* output

```
7: Down - program non-accepting [pids 5-0]
New state 3
Pr: 5 Tr: 5
Pr: 5 Tr: 6
Pr: 5 Tr: 7
7: sv_save
  7: proc 5 exec 7, 10 to 10, D_STEP   non-accepting [tau=0]
8: Down - program non-accepting [pids 5-0]
Stack state 1
8: Up - program
sv_restor
  8: proc 5 reverses 7, 10 to 10, D_STEP [abit=0,adepth=0,tau=0,0]
Pr: 4 Tr: 5
7: sv_save
  7: proc 4 exec 5, 10 to 10, D_STEP   non-accepting [tau=0]
8: Down - program non-accepting [pids 5-0]
New state 4
Pr: 5 Tr: 5
Pr: 5 Tr: 6
Pr: 5 Tr: 7
8: sv_save
  8: proc 5 exec 7, 10 to 10, D_STEP   non-accepting [tau=0]
9: Down - program non-accepting [pids 5-0]
New state 5
```

Figure 4.1: Example of *verbose* output produced by SPIN.

and produces a description of a graph for GRAPE. Figure 4.1 shows a fragment of *verbose* output corresponding to the mutual exclusion specification of Figure 2.6.

The SPIN-to-GRAPE tool is a PERL [177] program based on Algorithm 2, which traces the steps taken by SPIN when performing the state-space search. The algorithm uses a separate stack for each process in the model. Every time a line in the input file (created from the verbose output) indicates that a process has executed a statement (for example the line `7: proc 5 exec 7` ... in Figure 4.1), the current state number is added to the stack of that process. When a line of input indicates that a process has reversed (when search backtracks, e.g. the line `8: proc 5 reverses 7` ... in Figure 4.1), a value is popped from the stack of that process, and the current state number is set to this value. Every time a line of input is found which specifies that a new or old state has been reached (indicated by `New state` x or `Stack state` x respectively, where x is a state number), a line of GRAPE code is generated specifying that a transition should be added to the graph. The file produced as output from SPIN-to-GRAPE can be loaded into GAP, and the `AutGroupGraph()` function of GRAPE used to find the automorphism group of the state-graph. Note that the graph representation produced by SPIN-to-GRAPE does *not* include information about the values of variables at each state: each node of the graph is represented by an integer.

Though SPIN-to-GRAPE is useful for working with Promela specifications that exhibit small state-spaces, the complexity of the *nauty* algorithm means that it is not generally feasible to analyse models with more than around 15,000 states (though the performance of *nauty* in practice depends intimately on the structure of the input graph [123]).

Algorithm 2 Algorithm used by SPIN-to-GRAPE to construct the state-space of a model from a SPIN output file

```
open input and output files
current := 1
for each line in input file do
  if line signifies new state s then
    output edge current → s
    current := s
  else if line signifies old state s then
    output edge current → s
  else if line indicates process p executes then
    push current on stack for p
  else if line indicates process p reverses then
    current := pop from stack for p
  end if
end for
close input and output files
```

To accompany SPIN-to-GRAPE we have written a GAP function for computing quotient state-spaces – `QuotientKripke(`Γ, G`)`. This function takes a directed graph Γ representing a Kripke structure $\mathcal{M}$, together with a subgroup G of $Aut(\mathcal{M})$, and returns a directed graph representing the quotient Kripke structure $\mathcal{M}_G$. As discussed in Section 3.2, the theoretical minimum size of $|S_G|$ (the number of states in the quotient structure) is $|S|/|G|$ (where $|S|$ is the number of states in the original structure). The `QuotientKripke()` function allows us to determine, for small models, the factor of reduction available by exploiting symmetry in practice.

4.2 Simple Mutual Exclusion Example

Recall the simple mutual exclusion example, used for illustration in Chapters 2 and 3. The Promela specification for mutual exclusion with five processes is shown in Figure 2.6. A modified specification, annotated with scalarsets for use with SymmSpin, is given in Figure 3.3 and discussed in Section 3.3.2. A version of the specification in the SMC language is given in Figure 3.4 and discussed in Section 3.3.3.

4.2.1 Comparing the original specification with the SymmSpin version

In the initial specification (Figure 2.6), a process in its trying state is either blocked (if some process is in the critical state), or can move to the critical state. In the modified specification (Figure 3.3), a trying process can always make a transition to check if the critical section is free, moving to the critical state if it is, remaining in the trying state otherwise. Figure 4.2 shows the model for the modified mutual

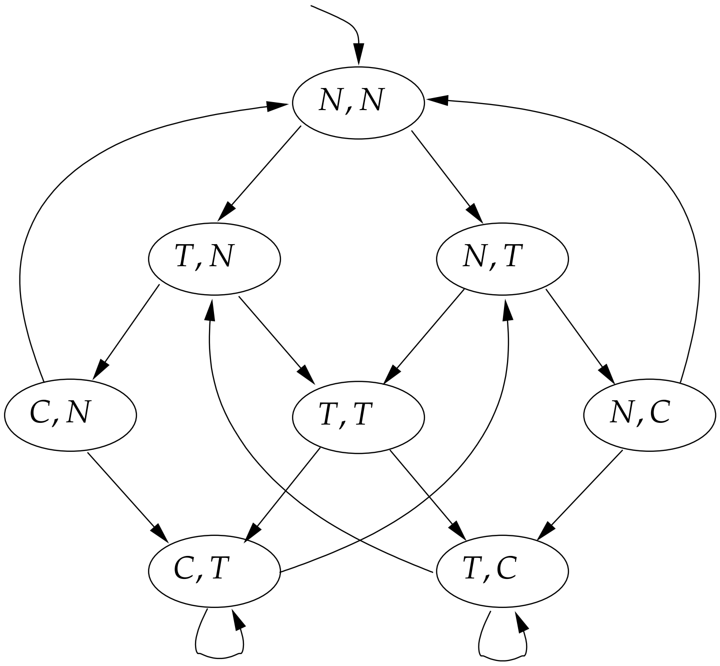


Figure 4.2: Mutual exclusion Kripke structure associated with the SymmSpin specification.

exclusion protocol when restricted to two processes.[1] The semantic difference is illustrated by the self-loop transitions in Figure 4.2 which are not present in Figure 2.3, the Kripke structure for the original specification. The re-modelling of the protocol has resulted in a more abstract underlying model which *simulates* the original (it adds behaviour). To verify that the symmetry reduction process preserves this behaviour, observe the self-loop transition in Figure 4.3, the quotient Kripke structure for the modified protocol under symmetry. It is difficult to see how this semantic difference could be avoided when using scalarsets to specify symmetry.

The SymmSpin specification is also more complex than the original. There are several additional variables: for each *user* process there is a flag to test whether the critical section is empty, a loop counter and an id parameter. The `init` process also uses a loop counter. This added complexity makes the specification more difficult to understand and increases the size of the state-vector for the associated model from 36 to 56 bytes. Through careful use of hidden variables (see Section 2.4.1) it is possible to reduce the state-vector to the original size, but this requires significant additional manual effort and expert knowledge of Promela. Furthermore, hidden variables can easily be misused, as SPIN does not check for cases where a hidden variable actually contains relevant state information [92].

1. The local variables i, j and *critical_empty* are not included in the figure as i is a constant process identifier, and j and *critical_empty* are only manipulated within a `d_step` block, being reset to default values before the end of this block.

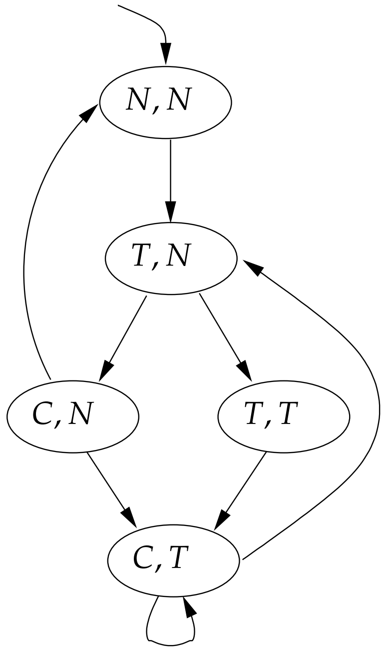


Figure 4.3: Quotient structure associated with the SymmSpin specification.

4.2.2 SMC specification

The SMC language is well suited to specifying this example (see Figure 3.4). The condition which guards the transition of a process to the critical state is expressed using the `ALL` quantifier, which asserts that a boolean expression must hold for every process of a given module type.

4.3 Peterson's Mutual Exclusion Protocol

Although useful for illustration, the mutual exclusion example discussed above is very abstract, and does not specify how exclusive access to the critical section by contending processes is guaranteed. We now discuss a more realistic protocol.

In Peterson's n-process mutual exclusion protocol [138], entry to the critical section is gained by a single process via a series of $n-1$ competitions. For each competition there is at least one loser, thus the mutual exclusion condition is satisfied since at most one process can win the final competition.

This protocol is used as an example for symmetry-reduced verification with the SymmSpin tool [14]. We discuss the SymmSpin specification, then present an alternative, semantically equivalent Promela specification which uses a smaller state-vector and is easier to understand. Though still symmetric, scalarsets cannot be used to directly specify symmetry in this enhanced specification. However, an equivalent SMC specification can be written.

We then present a more realistic specification of the protocol, and use SPIN-to-GRAPE to verify that the underlying model of this specification is symmetric. We show that neither scalarsets nor the SMC language can specity the inherent (full) symmetry for this example.

4.3.1 SymmSpin specification

We have obtained the Promela specification and accompanying system description file which were used for experiments with SymmSpin (personal communication, D. Bosnacki, 2003). Adapted versions of these files are given in Appendix A.1.1[2] The SymmSpin specification is based on a presentation of the algorithm in [122].

The system description file defines a `PID` scalarset type of size 3. SymmSpin regards global declarations as being part of a virtual proctype called `:system:`. For this example the `:system:` proctype includes two declarations which involve scalarsets. The *flag* array is indexed by variables of type `PID`, and its elements are bytes. This array is used to track the status of each process in the competition. Specifying that the index type of this array is `PID` indicates that the position of its elements (but not their values) should be affected by any permutation of the `PID` range. Additionally there is a global array *turn* which is indexed by the *byte* type, and contains `PID` values. Thus a `PID` permutation should affect the values of elements of this array, but not alter their positions. The system description file also states that the *user* proctype has two local `PID` variables.

Using SPIN-to-GRAPE to compute the symmetry group G of the state-space associated with this specification confirms that this use of scalarsets identifies all symmetries of the model. The symmetry group here is isomorphic to S_3, the symmetric group on three points (see Definition 11). Furthermore, GRAPE can be used to show that the quotient state-space constructed by SymmSpin is identical to that computed using the `QuotientKripke()` function.

4.3.2 A simpler, equivalent specification

If we do not use scalarsets to annotate the Peterson specification, and therefore are not concerned with the restrictions of Definition 22 (see Section 3.3.2), we can write a simpler Promela specification of the protocol, as shown in Appendix A.1.2.

In this specification the *user* proctype uses the built-in `_pid` variable (see Section 2.4.1) rather than being parameterised with an identifier. Although this means the *flag* array must be declared with size 4 rather than 3 (since values of the `_pid` variable start at 1) and so one value of this array is wasted, use of the built-in identifier instead of a parameter reduces the state-vector size by one byte for each process. Entry to the critical section is now guarded by a single boolean expression rather than a loop. This is easier to read, and avoids the inclusion of loop counter variables in the state-vector. The `init` process is also simplified. Consequently, the

2. We have applied some source code optimisation techniques (see Section 2.6.1) in order to compare the example fairly with an SMC specification and an alternative Promela specification.

simpler specification uses a 32 byte state-vector, whereas the original state-vector requires 44 bytes. We were not able to obtain a reduction in the state-vector for the original specification using hidden variables.

To check that the original and simplified specifications have the same underlying model we used SPIN-to-GRAPE to generate both state spaces as directed graphs, and GRAPE to confirm that these graphs are isomorphic. For this comparison we disabled the data-flow and statement merging optimisations provided by SPIN (see Section 2.4.3), in which case both models have 11,318 states.

It is interesting to note that applying data-flow optimisation and statement merging to the original specification results in a reduction to 6,143 states, whereas applying them to the simplified specification results in a model of just 2,636 states. Thus the simplification simultaneously reduces the state-vector size, and increases the factor of reduction obtained using the SPIN optimisations *before* symmetry reduction is even applied.

4.3.3 SMC specification

An SMC specification of the protocol is given as Appendix A.1.3. The specification is designed to be semantically equivalent to the Promela examples, and SMC verifies that the associated model also has 11,318 states. As in our enhanced Promela specification, the need for a loop to compute the predicate that guards entry to the critical section is avoided via the SMC `ALL` quantifier.

Although the SMC specification is difficult to read, this is due to the guarded command syntax of the language, rather than the method by which symmetry is handled.

4.3.4 A more realistic specification

The authors of [14] note that their specification uses atomicity in a somewhat unrealistic manner:

> *In our implementation the global predicate that guards the entry in the critical section is checked atomically. As this guard ranges over all process indices, the atomicity was necessary due to the restrictions on statements that can be used such that the state-space symmetry is preserved.*

Indeed, if the loop in the SymmSpin specification was not executed within an atomic statement, the boolean variable *ok* would be updated sequentially with respect to the process flags in a fixed order. This order would destroy symmetry between the processes.

In Appendix A.1.4 we give a Promela specification of the protocol where the predicate *ok* is computed non-atomically, and the process flags are considered in an *arbitrary* order. This is achieved by each process using a local array, *checked*, indexed by process identifiers, to track whether *flag[j]* has been considered for each $1 \leq j \leq n$. Making the order of this computation arbitrary means that the model checker will consider every possible ordering. This makes no assumptions about the order which an implementation of the protocol would use, making the specification very general. Additionally, not imposing an execution ordering preserves symmetry in the underlying model. The state-space associated with a two-process version of this specification is small enough that we can use SPIN-to-GRAPE to identify a symmetry group of order 2.

As noted in [14], the scalarset restrictions do not allow us to specify symmetry in this more realistic specification. Also, we cannot write an equivalent SMC specification since the language does not allow an update to refer to a specific process of a given module type. This example shows that even if there is *full* symmetry between identical processes, it may not be possible to specify this symmetry using scalarsets or a restricted input language.

4.4 A Prioritised Resource-Allocator

We now model a system which consists of *n client* processes, each of which requires access to a resource, and a *resource allocator* process which takes requests from the clients wishing to use the resource, granting access to one client at a time. Each client has a fixed *priority* level, and when faced with multiple requests the resource allocator grants access to the requesting client with the highest priority. When several requests are made with the same priority the resource allocator chooses non-deterministically which to satisfy. The system has a star topology, where the *resource allocator* process is the central node and all of the *client* processes communicate with this process only. The model is similar to an example used for symmetry reduction in partially symmetric systems using GQSs [162] (see Section 3.7).

Communication between a client and the resource allocator is controlled by a basic protocol. A client sends a *request* message to the allocator. When the allocator decides to allow this client access to the resource it sends back a *confirmation* message. Once the client finishes using the resource it sends a *finished* message to the allocator. There is one (asynchronous) communication channel between the resource allocator and each client. In order to allow a comprehensive investigation of the state-space of the associated specification using SPIN-to-GRAPE, it is important that the number of states is kept to a minimum. To reduce the number of states resulting from the interleaving of events internal to each process, atomic statements are used so that each execution step taken by each process includes a send or receive event (this state-space reduction is also suggested in [69]). An additional channel, *nullchan*, is used as a default value for local channel variables. This channel is declared with capacity 0 in order to minimise the state-vector.

A Promela specification of the system with seven clients and three levels of priority is given in Appendix A.2.1. There are two, three and two clients with priority levels 0, 1 and 2 respectively.

4.4.1 Analysis of symmetry in the resource allocator specification

The model $\mathcal{M}$ associated with the Promela specification $\mathcal{P}$ with seven clients, as described above, has of 1,921 states. Using our combination of SPIN, SPIN-to-GRAPE, GAP and GRAPE we find that $|Aut(\mathcal{M})| = 24$. It is clear that these automorphisms arise from the interchangeability of clients with the same priority level, and we can use GAP to show that $Aut(\mathcal{M})$ is isomorphic to a direct product (see Definition 14) of symmetric groups.

Proposition 1 *Let $\mathcal{M}$ be the model associated with the resource allocator specification described above. Then $Aut(\mathcal{M}) \cong S_2 \times S_3 \times S_2$.*

We used GAP to construct a group $G = S_2 \times S_3 \times S_2$ (GAP provides functionality to compute the direct product of permutation groups). The `IsomorphismGroups()` function was used to show that $Aut(\mathcal{M}) \cong G$. Each symmetric group consists of automorphisms which permute the identifiers of one set of similarly prioritised processes. The `QuotientKripke()` function shows that $|\mathcal{M}_{Aut(\mathcal{M})}| = 337$.

In general, for a resource allocator specification with n processes and k priority levels ($k, n > 0$), if m_i denotes the number of clients which have priority level i ($0 \leq i < k$, $\sum_{i=0}^{k-1} m_i = n$) then it is clear that $Aut(\mathcal{M})$ will be isomorphic to the group $\prod_{\substack{0 \leq i < k \\ m_i > 1}} S_{m_i}$, where $\prod$ denotes the direct product.

4.4.2 Re-modelling for SymmSpin and SMC

Symmetry in this example can be handled using scalarsets or input language restriction by separating the client processes into three distinct process types, *client0*, *client1* and *client2*, according to their priority level. To specify the example using scalarsets for use with SymmSpin, three separate scalarset types can be declared. The client proctype declarations will be essentially the same, and the resource allocator process will also involve duplicated code.

We have specified the example using SMC in Appendix A.2.2. Due to the major differences between the Promela and SMC languages, the SMC specification does not generate exactly the same state-space as the Promela specification (the SMC state-space is larger). However, both specifications model the same essential behaviour. Note that the three client modules in the SMC specification are almost identical, and that statements of the resource allocator module are separated into three similar blocks, one for each client module type.

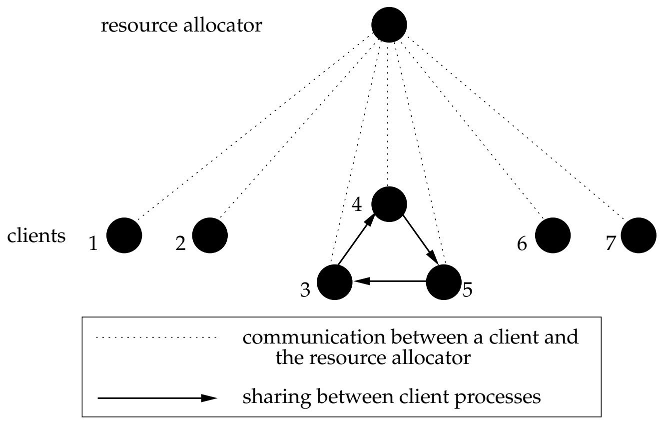


Figure 4.4: Inter-process sharing in the resource allocator specification.

4.4.3 Sharing between client processes

No communication between client processes takes place in our model of the resource allocator. Consider an alternative specification where client processes can *share* the resource. If client i is configured to share with client j then on receiving a *confirmation* message (i.e. gaining access to the resource), client i uses the resource then checks to see if client j has sent a request to the resource allocator. If this is the case, client i intercepts the request and gives client j access to the resource. When client j finishes using the resource it sends a *finished* message as usual. Client i intercepts this *finished* message, and sends its own *finished* message back to the resource allocator. Appendix A.2.3 shows a Promela specification of the resource allocator system where client 3 shares with client 4, client 4 with client 5, and client 5 with client 3.

Let $\mathcal{M}'$ denote the model associated with this specification. Our automated setup shows that $|Aut(\mathcal{M}')| = 12$, thus the degree of symmetry in the Kripke structure is reduced by enabling sharing between certain processes. This is because there is now a cyclic relationship between clients 3, 4 and 5 due to the configuration of the additional sharing functionality. This cyclic relationship is illustrated by Figure 4.4. We can use GAP to show that $Aut(\mathcal{M}') \cong S_2 \times C_3 \times S_2$, where C_3 is the cyclic group of order 3 (see Definition 12).

It is not possible to specify this product of symmetric and cyclic groups using SymmSpin or SMC as cyclic symmetries cannot be handled by either technique.

The resource allocator example illustrates that while priority information can be conveniently embedded within a specification, in order to specify symmetry between components of the same priority level using SymmSpin or SMC it is necessary to explicitly partition distinctly prioritised process into separate process types. This results in duplicated code, and makes the task of adding or removing priority levels laborious. Additionally, altering the communication structure to allow inter-process sharing changes the nature of symmetry in the associated model.

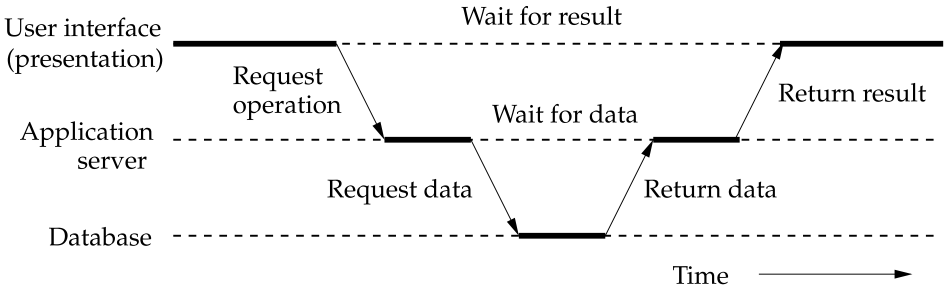


Figure 4.5: Flow of control in a three-tiered architecture.

The resulting symmetry group cannot be specified using either technique.

4.5 A Three-Tiered Architecture

A common software engineering design pattern for distributed systems is the *three-tiered architecture*. Components in such an architecture are separated into three layers, a layer of clients, a layer of servers and a layer of data storage systems. The typical flow of messages for such a system is shown in Figure 4.5 (adapted from [168]). This pattern is common in the e-business domain, where customers buy products or make bookings over the Internet. A set of servers at various geographical locations deal with customer (client) requests and communicate with a central (possibly replicated) database.

Our next specification is of a simple three-tiered system consisting of three process types: *client*, *server* and *database*. Each *client* process is parameterised by an input channel name, and a channel name associated with a *server* process. The *server* processes are parameterised by two channel names. The first of these channels is used to receive requests from *client* processes, and the second to send queries to the *database*. A *client* process loops continuously, sending a *request* message, and a reference to its incoming channel, to the *server* to which it is connected, and waiting until a *result* message is received on its incoming channel. Similarly each *server* process continuously repeats the actions of receiving a *request* and channel reference from a *client*, sending a *query* to the *database* and receiving *data*, then sending a *result* back to the *client* on the given channel. The *database* process continuously receives queries from the servers and returns data. All the channels in the specification are synchronous (to minimise the state-space sufficiently to allow us to use SPIN-to-GRAPE for analysis).

The configuration we consider consists of a database, three servers, and eight clients. There are three blocks of clients, two of size three, one of size two. Each block is associated with a distinct server. The Promela specification is given as Appendix A.3, and the topology is illustrated by Figure 4.6.

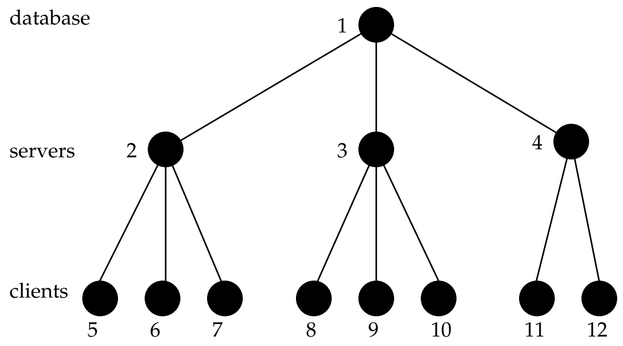


Figure 4.6: Topology of the three-tiered architecture specification.

4.5.1 Analysis of symmetry in the three-tiered specification

Let $\mathcal{P}$ denote the three-tiered specification of Appendix A.3, and $\mathcal{M}$ the associated model. The model is small enough to allow comprehensive analysis using our automated setup. This configuration of processes is interesting as there are multiple servers as well as multiple clients, and the tree of processes is not perfectly balanced, a feature which is reflected in the symmetry of the underlying model.

We have used our automated setup to prove the following result:

Proposition 2 *Let $\mathcal{M}$ be the model associated with the three-tiered specification described above. Then $Aut(\mathcal{M}) \cong (S_3 \wr S_2) \times S_2$.*

Here $S_3 \wr S_2$ denotes the outer wreath product of S_3 and S_2 (see Definition 16). As for direct products, GAP provides functionality for computing the wreath product of two permutation groups.

The original Kripke structure has 2,021 states. Using the *QuotientKripke* function, we find that the quotient structure with respect to $Aut(\mathcal{M})$ has 107 states. This is a significant factor of reduction which, for realistic sizes of model, could prove extremely effective in combatting state-space explosion. However, the kind of symmetry exhibited by this specification cannot be specified using scalarsets or input language restriction.

Intuitively, the reason that $Aut(\mathcal{M}) \cong (S_3 \wr S_2) \times S_2$ is that there are two blocks of three identical client processes (giving rise to the subgroup $S_3 \wr S_2$), and a single block of two client processes (giving rise to the subgroup S_2). Consider the model $\mathcal{M}$ associated with an arbitrary configuration of this three-tiered system. Let k be the maximum number of clients connected to any server in the configuration, and let m_i denote the number of servers which are connected to i clients for each $1 \leq i \leq k$. Since, for any $i > 0$, $S_i = S_i \wr S_1$, the above discussion and result clearly generalises to give:

$$Aut(\mathcal{M}) \cong \Big(\prod_{\substack{1 \leq i \leq k \\ m_i \neq 0}} (S_i \wr S_{m_i}) \Big).$$

In [105], the automorphism group of an arbitrary rooted tree is described, which could be used to generalise the above argument to systems with more than three tiers.

4.5.2 Mixed modes of communication in the three-tiered specification

As noted above, all communication in our three-tiered architecture specification is modelled using synchronous channels, so that messages are passed via a handshake between sender and recipient, with no buffering. Consider a modified version of the specification where the channel which $client_8$, $client_9$ and $client_{10}$ use to send requests to $server_3$ (channel `cl_se_2`) is changed to be an asynchronous buffer with size 1.

For the Kripke structure $\mathcal{M}$ associated with the original specification we have $Aut(\mathcal{M}) \cong (S_3 \wr S_2) \times S_2$. Let $\mathcal{M}'$ be the Kripke structure associated with the altered specification. Analysis using SPIN-to-GRAPE reveals that $Aut(\mathcal{M}') \cong S_3 \times S_3 \times S_2$, which is smaller than $(S_3 \wr S_2) \times S_2$. Clearly this is because the modified channel means that it is no longer possible to permute server processes 2 and 3, and their associated channels.

The three-tiered architecture specification is an example where a model has a non-trivial group of automorphisms which *cannot* be specified using existing techniques. The factor of reduction for this small example is still significant, showing that this type of symmetry is worth exploiting. Furthermore, modifying the example so that mixed modes of communication are used illustrates that the symmetry inherent in a model depends on the communication structure of the high level specification.

4.6 Message Routing in a Hypercube Network

A popular topology used in the implementation of switch-based multi-computers is the *hypercube* [168]. The following definition is adapted from [170]:

Definition 25 *The n-dimensional hypercube (where $n \geq 1$) is a graph $G = (V, E)$ where*

- $V = \{0,1\}^n$
- $E = \{\{\mathbf{x}, \mathbf{y}\} : \mathbf{x}, \mathbf{y} \in V \text{ differ in one bit}\}$.

A 4-dimensional hypercube can be displayed graphically as two cubes, as shown in Figure 4.7. In a switch-based multi-computer using a hypercube topology, messages are *routed* between the processors. Algorithm 3 is a simplified version of a routing algorithm described in [61]. For $x_i \in \{0,1\}$, we use $\overline{x_i}$ to denote $1 - x_i$ (the complement of x_i). Each node has an n-bit process identifier. On receiving a

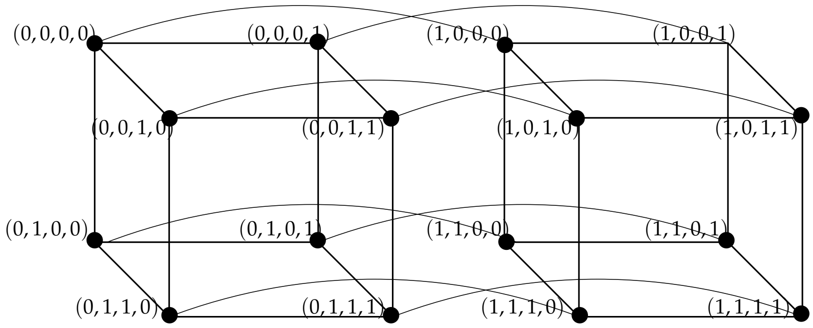


Figure 4.7: A 4-dimensional hypercube.

Algorithm 3 Basic algorithm for message routing in a hypercube network.

Behaviour of node $\mathbf{x} = (x_1, x_2, \ldots, x_n)$:
while *true* **do**
  receive message destined for node $\mathbf{y} = (y_1, y_2, \ldots, y_n)$
  if $\mathbf{x} = \mathbf{y}$ **then**
    process message
    choose a new destination node $\mathbf{y}$
  end if
  choose $i \in \{1, 2, \ldots, n\}$ such that $x_i \neq y_i$
  forward message to neighbour $(x_1, \ldots, \overline{x_i}, \ldots, x_n)$
end while

message, a node in the hypercube checks the id of the intended recipient. If this is the same as its own id then it processes the message, and chooses a new destination. Otherwise (or after a new destination has been chosen) the node forwards the message to a neighbour whose id has one more bit in common with the id of the intended recipient.

In our final example we model a system where messages are routed through a hypercube network using Algorithm 3. Appendix A.4.1 gives the Promela specification for message passing in a 3-dimensional hypercube. The processes are defined via a *node* proctype, parameterised by an input channel and n output channels (where n is the dimension of the hypercube), each of which is the input channel for a distinct neighbour. Global variables record the destination and current position of the message. Communication is achieved via a channel for each *node* in the hypercube, and the `init` process sends the first message to a non-deterministically chosen *node*. To ensure that the state-space of the model is small enough to analyse, only one message is passed through the network at a time.

In our specification the identifier of a *node* is an integer i in the range $1, 2, \ldots, n$. This represents the n-bit vector $i-1$ (viewed as a binary number). The subtraction is necessary since SPIN assigns process ids starting from 1 rather than 0. Given a message destined for *node* process k, *node* process i computes the bitwise exclusive-or of $k-1$ and $i-1$. If there is a 1 in position m of the result then the message can be forwarded to the neighbour of i with id j such that $i-1$ and $j-1$ (viewed as binary numbers) differ only bit m. The *node* process non-deterministically chooses one such suitable neighbour.

4.6.1 Analysis of symmetry in the hypercube specification

The automorphism group of an n-dimensional hypercube is well understood, and is derived in [76]. For any permutation $\alpha \in S_n$, we define the action of α on $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ by $\alpha(\mathbf{x}) = (x_{\alpha(1)}, x_{\alpha(2)}, \ldots, x_{\alpha(n)})$. For each $1 \leq i \leq n$, define the ith *complementation permutation* γ_i by $\gamma_i(\mathbf{x}) = (x_1, \ldots, \overline{x_i}, \ldots, x_n)$. Let $K_n = \langle \gamma_1, \gamma_2, \ldots, \gamma_n \rangle$, the group generated by all combinations of the γ_i. The automorphism group of the n-dimensional hypercube is the *semi-direct product* of S_n and K_n, denoted $K_n \rtimes S_n$ (see Definition 18, Section 3.1.4). It can be shown that $|K_n \rtimes S_n| = |K_n| \times |S_n| = 2^n \times n!$.

When analysing the nature of the symmetry in our hypercube specification we would have liked to have used a configuration with at least four dimensions as a case study. However, the state-space of even the 4-dimensional configuration proved too large to analyse using our setup (1.6×10^7 states) so we restrict ourselves to the 3-dimensional configuration (a cube). This problem demonstrates the rapid explosion of a state-space, and hence the need for techniques such as symmetry reduction.

Proposition 3 *Let $\mathcal{M}$ be the model associated with the 3-dimensional hypercube specification described above. Then $Aut(\mathcal{M}) \cong K_3 \rtimes S_3$.*

Again we have proved this result using our automated setup. The original Kripke structure has 15,409 states. Using `QuotientKripke()` we find that the resulting quotient structure has 411 states. Again the factor of reduction is encouraging. As with the three-tiered architecture example, the kind of symmetry exhibited by this specification cannot be specified using existing techniques.

Our specification of message routing in a hypercube involves arithmetic operations on variables which have *pid* type. Process ids are used as operands in bit-wise exclusive-or operations in order to determine how the packet should be routed. Approaches to exploiting symmetry usually prohibit these kind arithmetic operations, e.g. Condition 2 of Definition 22 prohibits this use of scalarset variables. This example shows that restrictions on the use of process identifiers in arithmetic operations are not always necessary for the preservation of symmetry. In Section 7.6.2 we discuss the problem of automatically identifying cases where process ids can be used as operands to arithmetic expressions without breaking symmetry.

Let $\mathcal{M}$ be the model associated with a configuration of the hypercube specification with n dimensions for some $n \geq 1$. It would seem likely, from the previous discussion, that the above result generalises to give $Aut(\mathcal{M}) \cong K_n \rtimes S_n$.

4.6.2 Message routing in a hypercube with a fixed initiator

Recall that in the hypercube specification the packet is first sent non-deterministically by the `init` process on one of the channels in the system. Such non-determinism in a model can often lead to a blow up of states, and a common approach to improve efficiency would be to remove such non-determinism. Indeed, altering the specification so that the `init` process always sends the packet on the channel associated with *node* process 1 results in a model with 8,866 states, compared with 15,409 states in the original model.[3]

SPIN-to-GRAPE shows that the resulting automorphism group of the altered model $\mathcal{M}'$ is isomorphic to a *subgroup* of the automorphism group of a cube. More specifically, $Aut(\mathcal{M}') \cong stab_{K_3 \rtimes S_3}(\mathbf{0})$, where $stab_{K_3 \rtimes S_3}(\mathbf{0})$ is the *stabiliser* of *node* $\mathbf{0} = (0,0,0)$.

Interestingly, the `QuotientKripke()` function shows that the quotient structure corresponding to $\mathcal{M}'$ has 1,669 states, whereas that corresponding to $\mathcal{M}$ has size 411. For this example, although removing non-determinism from the specification results in a reduction of size in the Kripke structure, the corresponding reduction in symmetry means that the quotient structure of the altered model is actually *larger* than the quotient structure of the model with no alterations.

The hypercube specification again illustrates that a model may have a group of automorphisms which lead to a significant state-space reduction, but which cannot be specified using existing techniques. Again, altering the communication structure of the specification (in this case so that there is a fixed initiating process) changes the nature of symmetry in the associated model.

Summary

We have introduced SPIN-to-GRAPE, a software tool which allows automorphisms of small state-graphs associated with Promela specifications to be explicitly computed. We have used SPIN-to-GRAPE to study five Promela examples. The first three examples highlight disadvantages of using scalarsets or input language restriction (via the SMC language) to specify symmetry. Our modified specification of Peterson's mutual exclusion protocol is an example for which there is full symmetry between components, and yet neither SymmSpin nor SMC can be used to express it. The final two examples exhibit fairly complex symmetry groups which decompose as wreath or semi-direct products of subgroups. This type of symmetry cannot be specified using scalarsets or the SMC language.

By making modifications to the example specifications and analysing the corresponding changes in symmetry in the underlying models, we have observed

3. Applying this modification to the 4-dimensional specification results in a reduction from 1.6×10^7 to 8.9×10^6 states. However, the smaller state-space is still too large for analysis using SPIN-to-GRAPE.

that the automorphism group of a model depends on the communication structure of its high level specification. In addition, we have shown that modifying a specification may reduce its state space but result in a loss of symmetry, so that the corresponding quotient model is larger than the quotient model associated with the original specification.

Chapter 5

Channel Diagrams

In Chapter 4 we identified a relationship between the communication structure of a Promela specification and the automorphisms of its associated model. In the resource allocator specification, allowing client processes to communicate with each other in order to share the resource reduces symmetry in the underlying model; making one of the communication links asynchronous in the three-tiered architecture specification destroys some of the original symmetry, and fixing the initiating process in the 3-dimensional hypercube specification results in a corresponding reduction in symmetry.

One formal notion of the communication structure of a Promela specification is its *channel diagram* [155]. In this chapter we show for each of the example specifications discussed in Chapter 4 that there is a correspondence between automorphisms of the channel diagram and automorphisms of the Kripke structure associated with a Promela specification. This correspondence is the motivation for the automatic symmetry detection techniques developed in Chapters 7 and 8, based on *static channel diagram* analysis.

5.1 Channel Diagram Associated with a Promela Specification

The channel diagram [155] associated with a Promela specification is a graphical representation of its channel-based communication structure. The definition we present here is adapted from the original presented in [155].

Given a Promela channel declaration `chan` c `= [`a`] of` $\{T_1, T_2, \ldots, T_k\}$, a is the *capacity* and $\{T_1, T_2, \ldots, T_k\}$ the *message type* of c. Note that $\{T_1, T_2, \ldots, T_k\}$ denotes an ordered list of types rather than a set. We use the set-based notation throughout for consistency with Promela. The *signature* of c, denoted *signature*(c) is the pair $(a, \{T_1, T_2, \ldots, T_k\})$. For example, if a channel `A` is declared as follows: `chan A = [3] of {mtype,byte}` then *signature*$(\mathtt{A}) = (3, \{\mathit{mtype}, \mathit{byte}\})$.

Let $\mathcal{P}$ be a Promela specification in which all process are instantiated atomically by the `init` process, and all channels are globally instantiated (see Section 2.4.1). Let V_P denote the set of process identifiers and V_C the set of global channel names in $\mathcal{P}$. For $i \in V_P$ let $proctype(i)$ be the name of the proctype of which process i is an instantiation.

Definition 26 *If $\mathcal{M} = (S, s_0, R)$ is the Kripke structure associated with $\mathcal{P}$ then the channel diagram of $\mathcal{P}$ is a coloured, bipartite digraph $\mathcal{CD}(\mathcal{P}) = (V, E, C)$ where:*

- *$V = V_P \cup V_C$ is the set of process identifiers and channel names in $\mathcal{P}$*
- *For $i \in V_P$ and $c \in V_C$,*
  - *$(i, c) \in E$ iff there is a reachable transition $(s, t) \in R$ which involves process i sending a message on channel c*
  - *$(c, i) \in E$ iff there is a reachable transition $(s, t) \in R$ which involves process i receiving a message on channel c*
- *C is a colouring function defined by $C(v) = proctype(v)$ if $v \in V_P$, and $C(v) = signature(v)$ if $v \in V_C$.*

Note that while it may not be possible to determine the operations involved in a transition (s, t) by examination of s and t alone (e.g. if the transition results from execution of an `atomic` block), this information can always be obtained from examination of s and t in the context of the specification $\mathcal{P}$.

Examples of channel diagrams are given throughout Sections 5.2 and 5.3. When displaying a channel diagram as a figure we use ovals and rectangles to represent processes and channels respectively.[1] The type of a process is given by its proctype name, and channel signatures are indicated using a key.

5.1.1 Deriving channel diagrams

Since Definition 26 depends on the transition relation R, construction of $\mathcal{CD}(\mathcal{P})$ in general requires exploration of the reachable states of $\mathcal{M}$. Thus we can only derive the channel diagram for a specification if its associated model is tractable. The channel diagrams used for illustration in this chapter have been manually derived from their associated Promela specifications via simulation with SPIN. This process could be automated by adding code to log the use of channels during verification to the `pan.c` file produced by SPIN (see Section 2.4.2).

In Chapter 7 we define the *static* channel diagram of a specification, which can be efficiently constructed by syntactic inspection of $\mathcal{P}$.

5.1.2 Channel diagram automorphisms

An automorphism of the channel diagram $\mathcal{CD}(\mathcal{P}) = (V, E, C)$ is an automorphism of the directed, coloured graph (V, E, C) (see Definition 19, Section 3.1.5).

1. In the original presentation of channel diagrams, ovals were used for channels and rectangles for processes [155]. The notation was changed by mistake in [42, 48, 49]. To be consistent with work published from this thesis we use the modified notation.

```
chan one_two = [1] of {int}; chan two_one = [1] of {int};

proctype node(chan in; chan out) {
  pid x;
  if
    :: in?  ...
    :: out! ...
  ...
}

init {
  atomic {
    run node(one_two,two_one);
    run node(two_one,one_two);
  }
}
```

Figure 5.1: A fragment of a Promela specification.

The group of all automorphisms of $\mathcal{CD}(\mathcal{P})$ is denoted $Aut(\mathcal{CD}(\mathcal{P}))$. We can compute $Aut(\mathcal{CD}(\mathcal{P}))$ by inputting $\mathcal{CD}(\mathcal{P})$ to GAP. The vertices of V_P are directly represented using the integers $\{1,2,\ldots,n\}$; the vertices of V_C are represented by $\{n+1, n+2, \ldots, n+m\}$, where $|V_C| = m$. The group $Aut(\mathcal{CD}(\mathcal{P}))$ is computed by the GRAPE function `AutGroupGraph()` (see Section 3.1.6). The colouring C is specified as an argument to this function.

An element $\alpha \in Aut(\mathcal{CD}(\mathcal{P}))$ has a natural action on $\mathcal{M}$, the model associated with $\mathcal{P}$, which we illustrate using an example. Let $\mathcal{P}$ be a Promela specification, part of which is shown in Figure 5.1, with associated model $\mathcal{M}$. Figure 5.2 shows the channel diagram for $\mathcal{P}$. It is easy to check that $Aut(\mathcal{CD}(\mathcal{P})) = \{id, (1\ 2)(one\_two\ two\_one)\}$. A state s of $\mathcal{M}$ has the form:

$$\begin{aligned} s \quad = \quad & (\text{contents of } one\_two,\ \text{contents of } two\_one,\ in_1,\ out_1,\ x_1,\ pc_1, \\ & \quad in_2,\ out_2,\ x_2,\ pc_2). \end{aligned}$$

where y_i denotes the value of variable y of *node* process i. The internal program counter variable for process i is denoted pc_i. For $\alpha \in Aut(\mathcal{CD}(\mathcal{P}))$, the state $\alpha(s)$ has the form:

$$\begin{aligned} \alpha(s) \quad = \quad & (\text{contents of } \alpha(one\_two),\ \text{contents of } \alpha(two\_one), \alpha(in_{\alpha(1)}),\ \alpha(out_{\alpha(1)}), \\ & \quad \alpha(x_{\alpha(1)}),\ pc_{\alpha(1)},\ \alpha(in_{\alpha(2)}),\ \alpha(out_{\alpha(2)}),\ \alpha(x_{\alpha(2)}),\ pc_{\alpha(2)}). \end{aligned}$$

The values of the variables of *node* i at s are initially those of *node* $\alpha(i)$ at $\alpha(s)$, then α is applied to the values of the channel and process id variables x_i, in_i and out_i. Similarly, the contents of channel c at s is that of channel $\alpha(c)$ at $\alpha(s)$. If a process id variable has value $y_i = 0$ then we define $\alpha(y_i) = 0$.

Concretely, suppose s = ([ ], [5], *two_one*, *one_two*, 0, 10, *one_two*, *two_one*, 1, 8) and $\alpha = (1\ 2)(one\_two\ two\_one)$. Then $\alpha(s)$ = ([5], [ ], *two_one*, *one_two*, 2, 8, *one_two*, *two_one*, 0, 10).

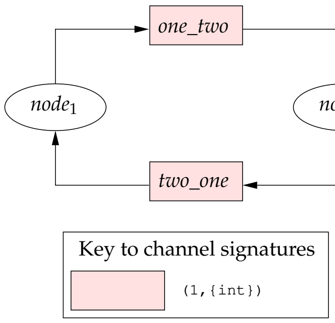


Figure 5.2: Channel diagram associated with the Promela code fragment of Figure 5.1.

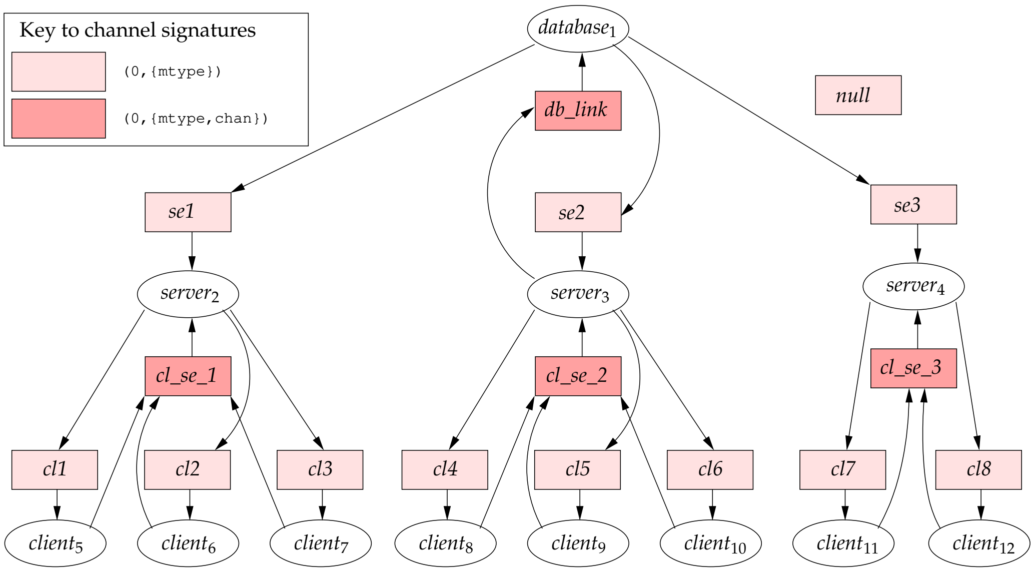


Figure 5.3: Channel diagram for three-tiered architecture specification.

We show that in some cases elements of $Aut(\mathcal{CD}(\mathcal{P}))$ induce *automorphisms* of $\mathcal{M}$ with this natural action.

5.2 Channel Diagrams for the Channel-based Specifications

We first consider the channel diagrams associated with the three-tiered architecture, hypercube and resource allocator specifications (see Sections 4.5, 4.6 and 4.4 respectively), since in these specifications processes communicate using channels rather than variables.

5.2.1 Three-tiered architecture channel diagram

Recall from Section 4.5 the three-tiered architecture example, the Promela specification of which is given in Appendix A.3. Figure 5.3 shows the channel diagram associated with this specification.

We use GRAPE to compute the following generating set for the group $Aut(\mathcal{CD}(\mathcal{P}))$:

$$\begin{aligned} Aut(\mathcal{CD}(\mathcal{P})) = \langle & (5\ 6)(cl1\ cl2),\ (6\ 7)(cl2\ cl3),\ (8\ 9)(cl4\ cl5), \\ & (9\ 10)(cl5\ cl6),\ (11\ 12)(cl7\ cl8), \\ & (5\ 8)(cl1\ cl4)(6\ 9)(cl2\ cl5)(7\ 10)(cl3\ cl6) \\ & \quad (2\ 3)(se1\ se2)(cl\_se\_1\ cl\_se\_2)\rangle. \end{aligned}$$

We can see from Figure 5.3 that the first generator of this group, the permutation $(5\ 6)(cl1\ cl2)$ is an automorphism of $\mathcal{CD}(\mathcal{P})$ since swapping clients 5 and 6 and simultaneously swapping the associated channels *cl1* and *cl2* leaves the structure and colouring of the channel diagram unchanged. The other generators can similarly be verified to be automorphisms of $\mathcal{CD}(\mathcal{P})$.

Our automated setup shows that the groups $Aut(\mathcal{M})$ and $Aut(\mathcal{CD}(\mathcal{P}))$ are isomorphic, thus there is a direct correspondence between channel diagram and Kripke structure automorphisms for this example.

Now consider the three-tiered specification with mixed modes of communication, discussed in Section 4.5.2. The difference between this specification and the original is that the signature of channel *cl_se_2* (the channel which client processes 8, 9 and 10 use to send requests to server process 3) is changed from (0,{`mtype`,`chan`}) to (1,{`mtype`,`chan`}). Let $\mathcal{P}'$ denote the modified specification, with associated model $\mathcal{M}'$. We observed in Section 4.5.2 that $Aut(\mathcal{M}')$ is a smaller group than $Aut(\mathcal{M})$, since changing this channel signature destroys symmetry between servers 2 and 3. Since channel nodes are coloured according to their signature, this change in symmetry is reflected in the automorphisms of the channel diagram associated with the specification: we find that $Aut(\mathcal{M}') \cong Aut(\mathcal{CD}(\mathcal{P}'))$.

5.2.2 Channel diagram for the hypercube specification

The channel diagram for the 3-dimensional hypercube specification (see Section 4.6 and Appendix A.4.1) is shown in Figure 5.4. Recall that the `init` process initially sends the packet to a non-deterministically chosen node, thus there are edges from the node representing the `init` process (with identifier 0) to every channel in the diagram. For neatness this is simplified in Figure 5.4.

The channel diagram $\mathcal{CD}(\mathcal{P})$ is essentially a cube. Since the *node* processes in $\mathcal{P}$ are all identical, we expect any automorphism of the channel diagram to correspond to an automorphism of the underlying Kripke structure, and indeed this is the case. As with the three-tiered architecture example, GRAPE shows that the groups $Aut(\mathcal{M})$ and $Aut(\mathcal{CD}(\mathcal{P}))$ are isomorphic.

In Section 4.6.2 we considered a modified specification where the `init` process always sends the packet initially to *node* process 1. Let $\mathcal{P}'$ denote this modified

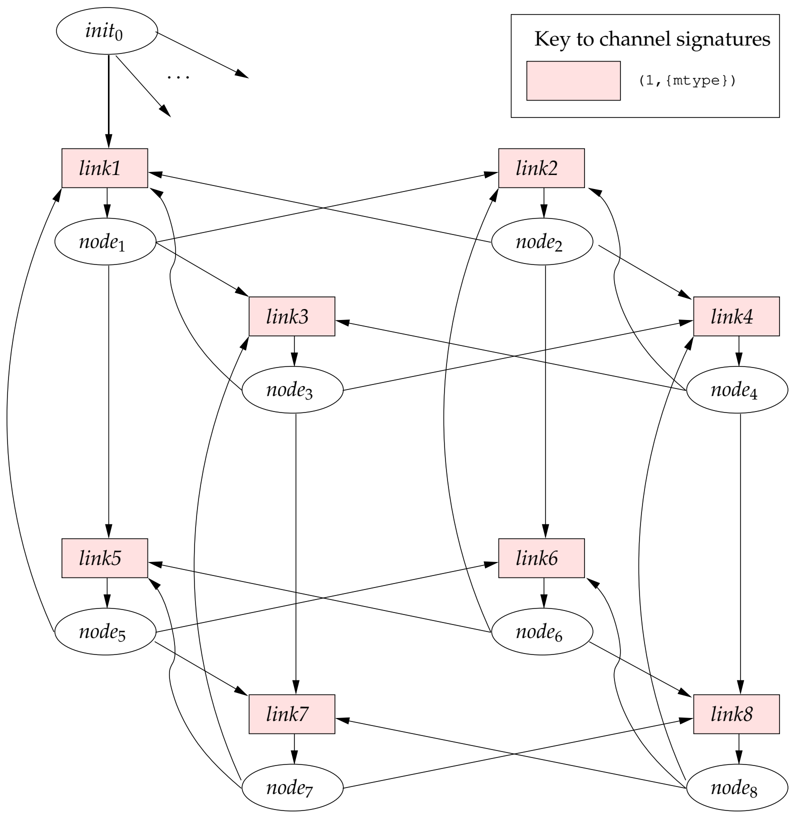


Figure 5.4: Channel diagram for 3d hypercube specification.

specification and $\mathcal{M}'$ its associated model. We found that this modification resulted in a corresponding loss in symmetry, since node process 1 is no longer equivalent to the other nodes. The channel diagram $\mathcal{CD}(\mathcal{P}')$ is identical to $\mathcal{CD}(\mathcal{P})$ except that the only edge from the node representing the `init` process is that to the channel node labelled *link1*. Removal of the other edges results in a loss of symmetry in the channel diagram, and the relationship between symmetries of the channel diagram and symmetries of the Kripke structure is maintained. Using our automated setup we find that $Aut(\mathcal{M}') \cong Aut(\mathcal{CD}(\mathcal{P}'))$.

5.2.3 Channel diagram for the prioritised resource allocator

Figure 5.5 shows the channel diagram for the prioritised resource allocator specification discussed in Section 4.4. The specification is given in Appendix A.2.1. The priority level of each client is also indicated in Figure 5.5, though this information is not part of the channel diagram.

Let $\mathcal{P}$ denote the resource allocator specification and $\mathcal{M}$ its associated model. Recall from Section 4.4.1 that $|Aut(\mathcal{M})| = 24$. Inputting the channel diagram $\mathcal{CD}(\mathcal{P})$ to GRAPE and computing its automorphism group reveals that:

$$Aut(\mathcal{CD}(\mathcal{P})) = \langle (1\ 2)(link1\ link2), (2\ 3)(link2\ link3), \ldots, (6\ 7)(link6\ link7) \rangle,$$

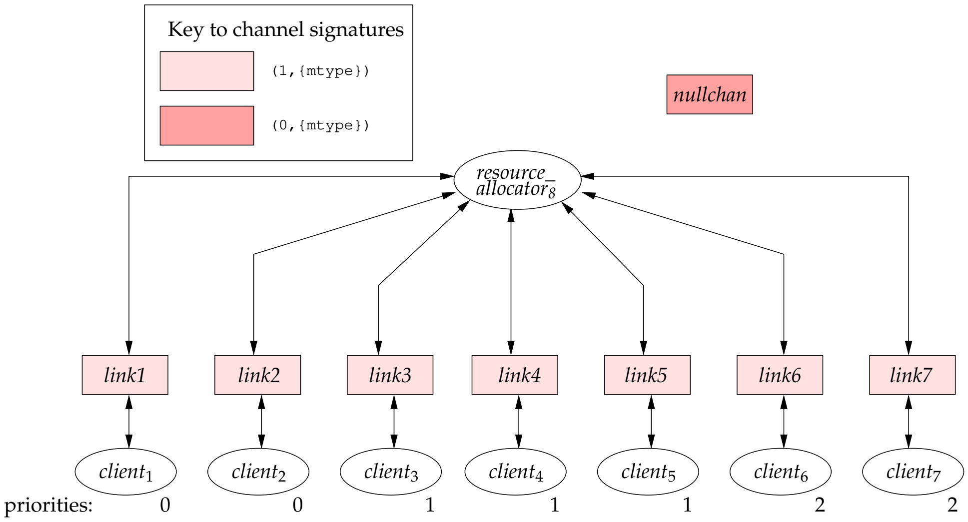


Figure 5.5: Channel diagram for resource allocator specification.

and $|Aut(\mathcal{CD}(\mathcal{P}))| = 5,040$. Since $|Aut(\mathcal{M})| \neq |Aut(\mathcal{CD}(\mathcal{P}))|$, there is not a direct correspondence between Kripke structure and channel diagram automorphisms. This is because priority levels, which induce asymmetry between components, are not encoded in the channel diagram.

However, we can use the function `IsomorphicSubgroups(`$Aut(\mathcal{CD}(\mathcal{P}))$, $Aut(\mathcal{M})$`)` to show that there is a monomorphism (see Definition 8, Section 3.1.1) which maps $Aut(\mathcal{M})$ to a subgroup G of $Aut(\mathcal{CD}(\mathcal{P}))$. By Theorem 2 (Section 3.1.1), $G \cong Aut(\mathcal{M})$. G is clearly the subgroup of $Aut(\mathcal{CD}(\mathcal{P}))$ which preserves the priority information indicated in Figure 5.5.

Let $\mathcal{P}'$ denote the resource allocator specification where certain clients share the resource (see Section 4.4.3), with associated model $\mathcal{M}'$. The corresponding channel diagram, $\mathcal{CD}(\mathcal{P}')$, is shown in Figure 5.6. The cyclic relationship between clients 3, 4 and 5 resulting from the configuration of process sharing (illustrated by Figure 4.4) is captured by the additional edges in Figure 5.6 compared with Figure 5.5.

We showed in Section 4.4.3 that $Aut(\mathcal{M}')$ is smaller than $Aut(\mathcal{M})$: introducing sharing reduces the symmetry inherent in the model. This reduction in symmetry is reflected in the channel diagram: we have $|Aut(\mathcal{CD}(\mathcal{P}'))| = 144$. Again there is a monomorphism from $Aut(\mathcal{M}')$ to a subgroup of $Aut(\mathcal{CD}(\mathcal{P}'))$.

5.3 Channel Diagrams for the Mutual Exclusion Examples

The Promela specifications of the simple mutual exclusion protocol (see Sections 2.2 and 4.2) and Peterson's mutual exclusion protocol (see Section 4.3) do not involve channels. Instead, processes communicate via global arrays. However, each of these examples still has a well-defined associated channel diagram consisting of just a

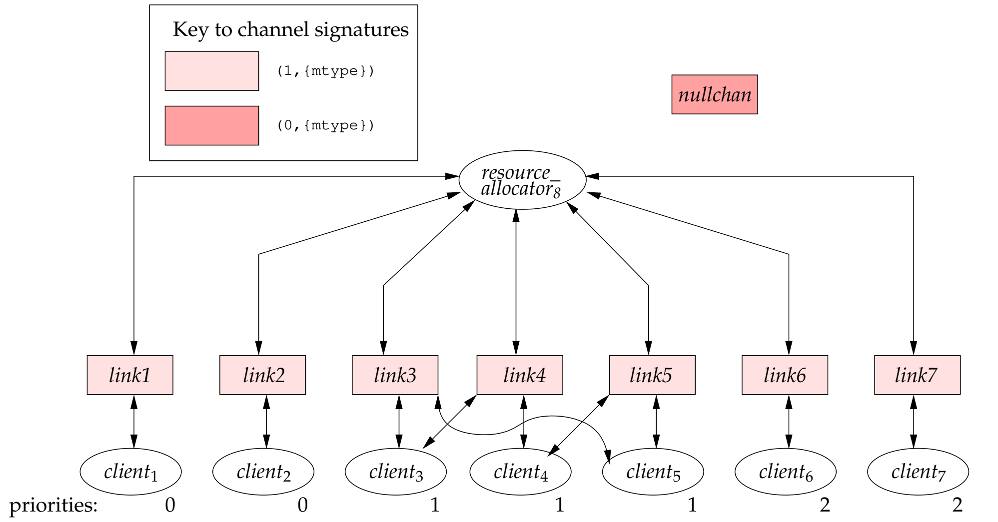


Figure 5.6: Channel diagram for resource allocator specification with resource sharing enabled.

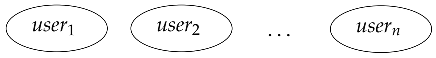


Figure 5.7: Form of channel diagram for the mutual exclusion examples.

set of process nodes – both the sets V_C and E of Definition 26 are empty. Figure 5.7 shows the general form of the channel diagram associated with an n-process mutual exclusion specification (either the simple example, or Peterson's protocol).

Although the channel diagram of Figure 5.7 is trivial, its automorphism group is the group S_n since all *user* processes are interchangeable. This group is isomorphic to the group of Kripke structure automorphisms for a mutual exclusion protocol with n processes.

5.4 Approximating Channel Diagrams

As discussed in Section 5.1.1, construction of $\mathcal{CD}(\mathcal{P})$ requires exploration of the reachable states of $\mathcal{M}$, which is precisely what model checking with symmetry reduction aims to avoid.

In Chapter 6 we introduce Promela-Lite, a specification language based on Promela. In Chapter 7 we define the *static* channel diagram $\mathcal{SCD}(\mathcal{P})$ associated with a Promela-Lite specification $\mathcal{P}$ and show that $\mathcal{SCD}(\mathcal{P})$ can be efficiently computed by static analysis of $\mathcal{P}$. The static channel diagram is an approximation of the channel diagram associated with a specification. We then show that there is a general correspondence between automorphisms of $\mathcal{SCD}(\mathcal{P})$ and automorphisms of $\mathcal{M}$, the model associated with $\mathcal{P}$.

Summary

We have defined the channel diagram of a Promela specification (first introduced in [155]), and used our automated setup to show for the examples of Chapter 4 that there is a correspondence between automorphisms of the channel diagram $\mathcal{CD}(\mathcal{P})$ and automorphisms of the Kripke structure $\mathcal{M}$ associated with a Promela specification $\mathcal{P}$. We have discussed the limitations associated with channel diagrams and motivated the use of the *static* channel diagram, an approximation of the channel diagram that can be efficiently computed via static analysis of $\mathcal{P}$.

Chapter 6

Promela-Lite

The examples of Chapter 4, and the correspondence between channel diagram and Kripke structure automorphisms observed in Chapter 5, motivate us to develop automatic symmetry detection techniques for Promela, which are *not* restricted to full symmetry, based on analysis of a structure similar to the channel diagram. This is the topic of Chapters 7 and 8. In order to support our techniques with a formal proof, we first present *Promela-Lite*, a specification language which captures the essential features of Promela. The Promela language includes a large set of keywords and language features which facilitate the specification of complex communication protocols. The downside of this is that proving properties about Promela specifications is laborious, requiring many case-by-case arguments. Rigorous proofs are also hindered by the lack of a formal definition of the semantics of Promela as implemented by SPIN.

Promela-Lite is a smaller specification language includes core Promela features such as parameterised processes, first-class channels and global variables, but omits many language features such as enumerated types, record types, arrays and rendez-vous channels. We are able to present a full grammar and type system for this smaller language, and define precise Kripke structure semantics for Promela-Lite specifications. In Chapter 7 we use the semantics to rigourously prove the correctness of our symmetry detection techniques for a Promela-like language. Promela-Lite and Promela are similar enough that it is not too great a leap of faith to accept that our results can be applied to Promela, for which a rigorous proof is not practical (as discussed above). In addition, omitting certain ornate features of Promela from Promela-Lite makes our proof easier to understand, and thus easier to transfer to other specification formalisms.

It is important to stress that we do not intend to implement a Promela-Lite model checker, or for users to write Promela-Lite specifications in practice (though we do illustrate the language with an example specification). While it may seem that the restricted syntax of Promela-Lite does not meet our aim of reducing the restrictions placed on the form of a specification, the restricted syntax is only for ease of presentation of our results. Our Promela implementation (see Chapter 8) lifts most of these restrictions.

The name *Promela-Lite* was inspired by *Featherweight Java*, a calculus which captures the core object oriented features of Java (classes, methods and inheritance), but omits most features of the full language [97].

We define the syntax and type system of Promela-Lite in Sections 6.1 and 6.2 respectively. In Section 6.3 we present Kripke structure semantics for the language, and prove that a well-typed Promela-Lite specification has a well-defined associated Kripke structure.

6.1 Syntax

6.1.1 A note on BNF

We use the standard Backus-Naur form (BNF, see e.g. [1]) to specify the syntax of Promela-Lite. BNF notation can be used to specify the grammar of a language via a sequence of *production rules* (also called non-terminals). A production rule $\langle prod \rangle$ has the form:

$$\begin{array}{ll} \langle prod \rangle ::= & A_{1,1} \quad A_{1,2} \quad \ldots \quad A_{1,s_1} \\ \mid & A_{2,1} \quad A_{2,2} \quad \ldots \quad A_{2,s_2} \\ \mid & \ldots \\ \mid & A_{k,1} \quad A_{k,2} \quad \ldots \quad A_{k,s_k} \end{array}$$

where each $A_{i,j}$ is either a production rule, or a terminal symbol. The terminal symbols include language keywords such as `do`, operators such as `::`; variable names, and literal values. A BNF grammar must have a designated initial production rule. A *sentence* in the language is a sequence of terminal symbols which conforms to the structure of the initial rule.

Let $\langle prod \rangle$ be a BNF production rule. We use the following shorthand notation to refer to occurrences of $\langle prod \rangle$ on the right hand side of other production rules:

- $\langle prod \rangle^?$ denotes an optional occurrence of $\langle prod \rangle$
- $\langle prod \rangle^*$ denotes a sequence of zero or more occurrences of $\langle prod \rangle$
- $\langle prod \rangle^+$ denotes a sequence of one or more occurrences of $\langle prod \rangle$
- $\langle prod\text{-}list, `\circ' \rangle$ denotes a $\circ$-separated list of one or more occurrences of $\langle prod \rangle$, i.e.
  $\langle prod\text{-}list, `\circ' \rangle ::= \langle prod \rangle$
  $\quad \mid \quad \langle prod \rangle \circ \langle prod\text{-}list, `\circ' \rangle$

6.1.2 Syntax of types

The syntax of Promela-Lite data types is summarised in Figure 6.1 (see Figure 6.3 for details of the $\langle name \rangle$ production rule). The initial production rule for this grammar is $\langle type \rangle$, and we refer to a sentence in the language of types as a *type*. The

$\langle type \rangle$ ::= `int`
  | `pid`
  | $\langle chantype \rangle$
  | $\langle typevar \rangle$

$\langle chantype \rangle$ ::= $\langle recursive \rangle^?$ `chan` { $\langle type\text{-}list, \text{`}\texttt{,}\text{'} \rangle$ }

$\langle recursive \rangle$ ::= `rec` $\langle typevar \rangle$ `.`

$\langle typevar \rangle$ ::= $\langle name \rangle$

Figure 6.1: Promela-Lite type syntax.

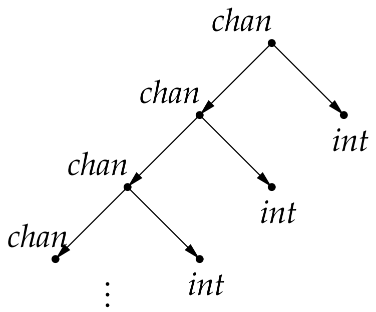


Figure 6.2: Infinite tree representing the recursive type $rec\, X \,.\, chan\{X, int\}$.

language includes two primitive data types, *int* and *pid*, representing integer values and process id values respectively. Basic channel types have the form $chan\{\overline{T}\}$, where $\overline{T}$ denotes a comma-separated list of types. The types which comprise $\overline{T}$ may themselves be channel types, thus Promela-Lite support first-class channels.

It can be useful for a channel of type T to accept a channel of type T as one of its arguments. In this case, T is a *recursive type* – its form is self-referential. Accordingly, Promela-Lite includes syntax for *recursive* channel types (the $\langle recursive \rangle$ rule of Figure 6.1). For example, consider a type T of the form $rec\, X \,.\, chan\{X, int\}$. Then T denotes a channel which accepts messages consisting of two fields: a channel of type T, and an integer. This recursive type can be *unfolded* by removing the initial '$rec\, X\,.$' and substituting 'X' for the original expression, resulting in the type expression $chan\{rec\, X \,.\, chan\{X, int\}, int\}$ (which can in turn be unfolding). The resulting types are the same, and intuitively they represent the type illustrated as an infinite tree in Figure 6.2. We discuss the implicit use of recursive types in Promela in Section 8.2.2. We use $chan\{\overline{T}\}$ to refer to an arbitrary channel type, since a channel type of the form $rec\, X \,.\, chan\{\dots\}$ can always be unfolded into this form. In Section 8.2.2 we discuss an algorithm for minimising recursive types by converting them to a canonical form. We say that two recursive types are equal if they are identical after minimisation.

The name 'X' used in the above example is a *type variable*, and is said to be *bound*, as it is introduced by the prefix '$rec\, X\,.$' and then occurs within the scope of this prefix. A type variable which is not bound is said to be *free*. A *well-formed* type is one for which there are no free type variables. The types *int*, $chan\{int\}$ and $rec\ X\,.\,chan\{X, int\}$ are all well-formed; the type $chan\{X\}$ is not.

6.1.3 Syntax of the language

A Promela-Lite specification consists of a series of channel and global variable declarations, one or more proctypes, and an `init` process. The syntax is given in Figure 6.3. The initial production rule is $\langle spec \rangle$, and we have simplified the presentation of the rules $\langle name \rangle$ and $\langle number \rangle$. We refer to a valid Promela-Lite sentence as a *specification*. In the $\langle guard \rangle$ production rule, $\bowtie$ denotes an operator taken from the set $\{==, !=, <, <=, >, >=\}$. For simplicity, we have not included the division operator in Promela-Lite. This is to avoid the need for detailed semantics for division-by-zero errors in Section 6.3, an issue which is orthogonal to the symmetry reduction techniques which we present in Chapter 7.

A channel declaration `chan` c `= [`a`] of {`$\overline{T}$`}` (according to the $\langle channel \rangle$ rule of Figure 6.3) defines a buffered channel c with type $chan\{\overline{T}\}$ and length a. This is similar to a *globally instantiated* channel in Promela (see page 27). We define the signature of c by $signature(c) = (\{\overline{T}\}, a)$. This is similar to the notion of channel signatures for Promela specifications defined in Section 5.1. We refer to channels declared in this way as *static* channels. The name of a static channel cannot be re-assigned (either by appearing on the left hand side of an assignment, or as an argument to a channel receive operation). If $signature(c) = (\{\overline{T}\}, a)$ we use $cap(c)$ to denote the capacity of c (which is equal to a).

A global variable declaration $T\ x = a$ associates a name x with a type $T \in \{int, pid\}$ and an initial value a.

A Promela-Lite proctype is a parameterised process definition. A proctype has a list of parameters, and a set of statements contained in a `do`...`od` loop. For simplicity we do not allow proctypes to declare local variables. In Promela, parameters to a proctype and local variables are treated identically, thus any local variables can be equivalently declared as parameter, with an initial value supplied as a `run` statement argument. For this reason we use the terms *parameter* and *local variable* interchangeably throughout this chapter and Chapter 7.

Each statement has the form `atomic` { $\langle guard \rangle$ `->` $\langle$*update-list*, '`;`'$\rangle$ }. Executability of the statement is decided by $\langle guard \rangle$, a boolean expression over variables and channels of the specification. The effect of a statement is determined by $\langle$*update-list*, '`;`'$\rangle$, which is a sequence of updates to variables and channels. The `atomic` block which surrounds the guard and updates indicates that executing the statement results in a single transition of the system. Keywords to determine the length, fullness and emptiness of channels are provided by the language.

The `init` process consists of a set of `run` statements. Each `run` statement instantiates a process of a given proctype, assigning initial values to all of its local variables. The `atomic` block surrounding the `run` statements indicates that all

⟨*spec*⟩ ::= ⟨*channel*⟩* ⟨*global*⟩* ⟨*proctype*⟩$^+$ ⟨*init*⟩

⟨*channel*⟩ ::= ⟨*name*⟩ `=` `[` ⟨*number*⟩ `]` `of` { ⟨*type-list*, ‘`,`’⟩ } `;`

⟨*global*⟩ ::= ⟨*type*⟩ ⟨*name*⟩ `=` ⟨*number*⟩ `;`

⟨*proctype*⟩ ::= ⟨*name*⟩ `(` ⟨*param-list*, ‘`;`’⟩$^?$ `)` `do` ⟨*statement-list*, ‘`::`’⟩ `od`

⟨*param*⟩ ::= ⟨*type*⟩ ⟨*name*⟩

⟨*statement*⟩ ::= `atomic` { ⟨*guard*⟩ `->` ⟨*update-list*, ‘`;`’⟩ }

⟨*guard*⟩ ::= ⟨*expr*⟩ $\bowtie$ ⟨*expr*⟩
 | `nfull` `(` ⟨*name*⟩ `)`
 | `nempty` `(` ⟨*name*⟩ `)`
 | `!` ⟨*guard*⟩
 | ⟨*guard*⟩ `&&` ⟨*guard*⟩
 | ⟨*guard*⟩ || ⟨*guard*⟩
 | `(` ⟨*guard*⟩ `)`

⟨*update*⟩ ::= `skip`
 | ⟨*name*⟩ `=` ⟨*expr*⟩
 | ⟨*name*⟩ `?` ⟨*name-list*, ‘`,`’⟩
 | ⟨*name*⟩ `!` ⟨*expr-list*, ‘`,`’⟩

⟨*init*⟩ ::= `init atomic` ⟨*run-list*, ‘`;`’⟩

⟨*run*⟩ ::= `run` ⟨*name*⟩ `(` ⟨*arg-list*, ‘`,`’⟩$^?$ `)` `;`

⟨*arg*⟩ ::= ⟨*name*⟩
 | ⟨*number*⟩
 | `null`

⟨*expr*⟩ ::= ⟨*name*⟩
 | ⟨*number*⟩
 | `_pid`
 | `null`
 | `len` `(` ⟨*name*⟩ `)`
 | `(` ⟨*expr*⟩ `)`
 | ⟨*expr*⟩ $\circ$ ⟨*expr*⟩ (where $\circ \in \{+, -, *\}$)

⟨*name*⟩ ::= an alpha-numeric string, which may include ‘_’, and must start with a letter or with ‘_’

⟨*number*⟩ ::= an integer

Figure 6.3: Syntax of Promela-Lite.

Judgements

$\Gamma \vdash \diamond$	Γ is a well-formed type environment
$\Gamma \vdash T$	T is a well-formed type in Γ
$\Gamma \vdash \overline{T}$	$T_1, T_2, \ldots, T_k$ are well-formed types in Γ
$\Gamma \vdash e : T$	e is a well-formed expression of type T in Γ
$\Gamma \vdash \overline{e} : \overline{T}$	$e_1, e_2, \ldots, e_k$ are well-formed expressions of types $T_1, T_2, \ldots, T_k$ respectively in Γ
$\Gamma \vdash f$ `OK`	f is a well-formed Promela-Lite fragment in Γ
$\Gamma \vdash f_i$ `OK` $(1 \leq i \leq l)$	$f_1, f_2, \ldots, f_l$ are well-formed Promela-Lite fragments in Γ

General form of a type rule

$$\frac{\Gamma_1 \vdash \mathcal{J}_1 \qquad \Gamma_2 \vdash \mathcal{J}_2 \qquad \ldots \qquad \Gamma_l \vdash \mathcal{J}_l \qquad \text{(other conditions)}}{\Gamma \vdash \mathcal{J}}\text{(rule name)}$$

Figure 6.4: Notation for type rules.

processes in the specification are instantiated simultaneously. If process i is an instantiation of proctype p, we write $proctype(i) = p$.

There is a special channel literal, `null`, which denotes an undefined channel reference, intended for use as a default value. The typing rules of Section 6.2 prevent use of `null` for communication. The value 0 can be used as a default value for variables with *pid* type. Like Promela, each Promela-Lite process has a built in constant, `_pid`, which records its run-time instantiation number. This is defined as the position of its `run` statement in the `init` process.

An example Promela-Lite specification is given in Figure 6.8 and discussed in Section 6.5.

6.2 Type System

We present a type system for Promela-Lite, using the notation of [23], adapted with shorthand notation from [97]. In Section 6.3 we present Kripke structure semantics for Promela-Lite specifications, and show that if $\mathcal{P}$ is a well-typed Promela-Lite specification then it has a well-defined associated Kripke structure (Theorem 11).

A *typing environment* Γ is an ordered list of distinct variables and their types, and has the form $x_1 : T_1, x_2 : T_2, \ldots, x_k : T_k$. Here $x : T$ reads "*x has type T*". The empty typing environment is denoted $\emptyset$, and the set of variables declared in typing environment Γ is denoted $dom(\Gamma)$. We associate with Γ a set $sc(\Gamma)$ consisting of the names of all static channels declared in Γ. We define $sc(\emptyset) = \emptyset$ (the first $\emptyset$ denotes the empty typing environment, the second an empty set), and use $\Gamma, sc(c)$ to denote a new environment which is the same as Γ except that c is added to $sc(\Gamma)$.

Figure 6.4 summarises the forms of type judgement which we use, together with the general form of a typing rule. The judgements can be used to assert that an environment Γ is well-formed, a type T is well-formed in Γ (see Section 6.1.2), an expression is well-formed and has type T in Γ, and a fragment of a specification (e.g. a proctype declaration or a statement) is well-formed in Γ. Intuitively, a type, expression, or fragment is well-formed if it can be attributed clear semantics (so, for example, the expression '5+*true*' is not well-formed, whereas '5+6' is a well-formed expression with type *int*), and an environment is well-formed if it is comprised of sensible variable declarations. Formally, asserting that an environment, type, expression or fragment is well-formed just means that it is regarded as legal by the type system. Given a language together with a type system and formal semantics, the intuitive and formal notions of well-formedness coincide if we can prove a theorem showing that well-formed sentences in the language are well behaved according to the semantics.

For brevity, we use $\Gamma \vdash \overline{T}$ to assert that types $T_1, T_2, \ldots, T_k$ are well-formed in Γ, and $\Gamma \vdash \overline{e} : \overline{T}$ to assert that for $1 \leq i \leq k$, expression e_i is well-formed and has type T_i in Γ. Similarly, $\Gamma \vdash f_i$ $\texttt{OK}$ $(1 \leq i \leq l)$ asserts that Promela-Lite fragments $f_1, f_2, \ldots, f_l$ are all well-formed in Γ. A typing rule consists of a horizontal line, with a list of judgements and other conditions above the line, and a single judgement below. If the judgements and conditions above the line all hold then the truth of the judgement below the line can be inferred.

Figure 6.5 gives a complete set of typing rules for Promela-Lite. The value n referred to by rule T-PID-LITERAL is the number of processes in the specification, and is determined by the number of `run` statements in the `init` process. For presentation of the type system we introduce a *tuple* type, to represent the form of arguments for a proctype. A proctype which accepts an ordered list of arguments of types $T_1, T_2, \ldots, T_k$ has type $(T_1, T_2, \ldots, T_k)$. The symbols a and $\overline{a}$ refer to literal values; e, e_i and $\overline{e}$ to expressions; c to a static channel; p to a proctype name; x and $\overline{x}$ to local/global variable names (x could also be a proctype name in T-VAR); u_i to updates; g_i to guards; r_i to `run` statements. In rule T-PROCTYPE we use $p\,(\overline{T}\ \overline{x})$ as shorthand for a proctype name together with a list of formal parameters $x_1, x_2, \ldots, x_k$, where x_i has type T_i $(1 \leq i \leq k)$. We use $\mathit{alldiff}\,(x_1, x_2, \ldots, x_k)$ to assert that $x_i \neq x_j$ if $i \neq j$ (i.e. the x_i are all different).

The rules T-SEND and T-RECV require that the fullness/emptiness of a channel is checked before it can be used for communication. Note that in both rules the guard g and/or updates $u_2, \ldots, u_l$ can be omitted (for conciseness this is not indicated in Figure 6.5).

A literal value in the range $\{0, 1, \ldots, n\}$ has both type *pid* and *int* according to the type system. Such a literal occurs in a *pid* context if it is assigned to a *pid* variable, sent as a *pid* argument on a channel, passed as a *pid* argument in a `run` statement, or compared with a *pid* variable using == or `!=`. We say that an occurrence of a has type *pid* if it occurs in a *pid* context, otherwise it has type *int*.

Environment

$$\frac{}{\emptyset \vdash \diamond}\ (\textsc{T-Env-}\emptyset) \qquad \frac{\Gamma \vdash \diamond \quad \Gamma \vdash T \quad x \notin dom(\Gamma)}{\Gamma, x : T \vdash \diamond}\ (\textsc{T-Env-}x)$$

Expressions

$$\frac{\Gamma \vdash \diamond \quad a \in \mathbb{Z}}{\Gamma \vdash a : int}\ (\textsc{T-Int-Literal}) \qquad \frac{\Gamma \vdash \diamond \quad a \in \{0, 1, \ldots, n\}}{\Gamma \vdash a : pid}\ (\textsc{T-Pid-Literal})$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{\_pid} : pid}\ (\textsc{T-\_pid}) \qquad \frac{\Gamma \vdash e : T}{\Gamma \vdash \texttt{(}e\texttt{)} : T}\ (\textsc{T-Parenthesis-}e)$$

$$\frac{\Gamma \vdash \diamond \quad \Gamma \vdash \overline{T}}{\Gamma \vdash \texttt{null} : chan\{\overline{T}\}}\ (\textsc{T-Null}) \qquad \frac{\Gamma, x : T, \Gamma' \vdash \diamond}{\Gamma, x : T, \Gamma' \vdash x : T}\ (\textsc{T-Var})$$

$$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int \quad \circ \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}}{\Gamma \vdash e_1 \circ e_2 : int}\ (\textsc{T-Arith}) \qquad \frac{\Gamma \vdash c : chan\{\overline{T}\}}{\Gamma \vdash \texttt{len(}c\texttt{)} : int}\ (\textsc{T-Len})$$

Guards

$$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int \quad \bowtie \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}}{\Gamma \vdash e_1 \bowtie e_2 \ \text{OK}}\ (\textsc{T-Rel})$$

$$\frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T \quad \bowtie \in \{\texttt{==}, \texttt{!=}\} \quad T \neq (T_1, T_2, \ldots, T_k)}{\Gamma \vdash e_1 \bowtie e_2 \ \text{OK}}\ (\textsc{T-Eq})$$

$$\frac{\Gamma \vdash g \ \text{OK}}{\Gamma \vdash \texttt{(}g\texttt{)} \ \text{OK}}\ (\textsc{T-Parenthesis-}g) \qquad \frac{\Gamma \vdash g \ \text{OK}}{\Gamma \vdash \texttt{!}g \ \text{OK}}\ (\textsc{T-Not})$$

$$\frac{\Gamma \vdash g_1 \ \text{OK} \quad \Gamma \vdash g_2 \ \text{OK}}{\Gamma \vdash g_1 \ \texttt{\&\&} \ g_2 \ \text{OK}}\ (\textsc{T-And}) \qquad \frac{\Gamma \vdash g_1 \ \text{OK} \quad \Gamma \vdash g_2 \ \text{OK}}{\Gamma \vdash g_1 \ \texttt{||} \ g_2 \ \text{OK}}\ (\textsc{T-Or})$$

Basic updates

$$\frac{\Gamma \vdash x : T \quad \Gamma \vdash e : T \quad x \notin sc(\Gamma) \quad T \neq (T_1, T_2, \ldots, T_k)}{\Gamma \vdash x = e \ \text{OK}}\ (\textsc{T-Assign}) \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{skip} \ \text{OK}}\ (\textsc{T-Skip})$$

Statements

$$\frac{\Gamma \vdash g \ \text{OK} \quad \Gamma \vdash u_i \ \text{OK} \ (1 \leq i \leq l)}{\Gamma \vdash \texttt{atomic \{ } g \ \texttt{->} \ u_1\texttt{;} \ u_2\texttt{;} \ \ldots\texttt{;} \ u_l \ \texttt{\}} \ \text{OK}}\ (\textsc{T-Update})$$

$$\frac{\Gamma \vdash g \ \text{OK} \quad \Gamma \vdash x : chan\{\overline{T}\} \quad \Gamma \vdash \overline{e} : \overline{T} \quad \Gamma \vdash u_i \ \text{OK} \ (2 \leq i \leq l)}{\Gamma \vdash \texttt{atomic \{ (}g\texttt{) \&\& nfull(}x\texttt{) ->} \ c\texttt{!}\overline{e}\texttt{;} \ u_2\texttt{;} \ \ldots\texttt{;} \ u_l \ \texttt{\}} \ \text{OK}}\ (\textsc{T-Send})$$

$$\frac{\begin{array}{c}\Gamma \vdash g \ \text{OK} \quad \Gamma \vdash x : chan\{\overline{T}\} \quad \Gamma \vdash \overline{x} : \overline{T} \\ \Gamma \vdash u_i \ \text{OK} \ (2 \leq i \leq l) \quad alldiff(\overline{x}) \quad \{\overline{x}\} \cap sc(\Gamma) = \emptyset\end{array}}{\Gamma \vdash \texttt{atomic \{ (}g\texttt{) \&\& nempty(}x\texttt{) ->} \ c\texttt{?}\overline{x}\texttt{;} \ u_2\texttt{;} \ \ldots\texttt{;} \ u_l \ \texttt{\}} \ \text{OK}}\ (\textsc{T-Recv})$$

Declarations

$$\frac{\begin{array}{c}\Gamma, x : T \vdash \langle global\rangle^* \ \langle proctype\rangle^+ \ \langle init\rangle \ \text{OK} \\ \Gamma \vdash a : T \quad T \in \{int, pid\} \quad x \notin dom(\Gamma)\end{array}}{\Gamma \vdash T \ x \ \texttt{=} \ a\texttt{;} \ \langle global\rangle^* \langle proctype\rangle^+ \langle init\rangle \ \text{OK}}\ (\textsc{T-Global})$$

$$\frac{\begin{array}{c}\Gamma, c : chan\{\overline{T}\}, sc(c) \vdash \langle channel\rangle^* \ \langle global\rangle^* \ \langle proctype\rangle^+ \ \langle init\rangle \ \text{OK} \\ \Gamma \vdash \overline{T} \quad c \notin dom(\Gamma) \quad a > 0\end{array}}{\Gamma \vdash \texttt{chan} \ c \ \texttt{= [}a\texttt{] of \{}\overline{T}\texttt{\};} \langle channel\rangle^* \langle global\rangle^* \langle proctype\rangle^+ \langle init\rangle \ \text{OK}}\ (\textsc{T-Sc})$$

Processes

$$\frac{\begin{array}{c}\Gamma \vdash \overline{T} \quad \Gamma, \overline{x} : \overline{T} \vdash s_i \ \text{OK} \ (1 \leq i \leq l) \\ \Gamma, p : (\overline{T}) \vdash \langle proctype\rangle^* \ \langle init\rangle \ \text{OK} \quad \{p, \overline{x}\} \cap dom(\Gamma) = \emptyset \quad alldiff(p, \overline{x})\end{array}}{\Gamma \vdash \texttt{proctype} \ p\texttt{(}\overline{T} \ \overline{x}\texttt{) \{ do ::} \ s_1 \ \texttt{::} \ s_2 \ \texttt{::} \ \ldots \ \texttt{::} \ s_l \ \texttt{od \}} \ \langle proctype\rangle^* \ \langle init\rangle \ \text{OK}}\ (\textsc{T-Proctype})$$

$$\frac{\Gamma \vdash p : (\overline{T}) \quad \Gamma \vdash \overline{a} : \overline{T} \quad \overline{a} \subseteq \mathbb{Z} \cup \{\texttt{null}\} \cup sc(\Gamma)}{\Gamma \vdash \texttt{run} \ p\texttt{(}\overline{a}\texttt{)} \ \text{OK}}\ (\textsc{T-Run})$$

$$\frac{\Gamma \vdash r_i \ \text{OK} \ (1 \leq i \leq k)}{\Gamma \vdash \texttt{init \{ atomic \{} \ r_1\texttt{;} \ r_2\texttt{;} \ \ldots\texttt{;} \ r_k \ \texttt{\} \}} \ \text{OK}}\ (\textsc{T-Init})$$

Figure 6.5: Type system for Promela-Lite.

A Promela-Lite specification $\mathcal{P}$ has one of the three forms:

1. `chan` c `= [`a`] of {`$\overline{T}$`};` $\langle channel\rangle^*\langle global\rangle^*\langle proctype\rangle^+\langle init\rangle$
2. $T\ x$ `=` a`;` $\langle global\rangle^*\langle proctype\rangle^+\langle init\rangle$
3. `proctype` p`(`$\overline{T}\ \overline{x}$`) { do ::` s_1 `::` s_2 `:: ... ::` s_l `od }` $\langle proctype\rangle^*\ \langle init\rangle$

depending whether or not there are any channel or global variable declarations in $\mathcal{P}$. Depending on which form $\mathcal{P}$ takes, one of the typing rules T-SC, T-GLOBAL or T-PROCTYPE is applicable. We say that $\mathcal{P}$ is *well-typed* if $\emptyset \vdash \mathcal{P}$ OK. We now present Kripke structure semantics for Promela-Lite, and show that these semantics unambiguously define the model associated with a well-typed Promela-Lite specification.

6.3 Kripke Structure Semantics

Let $\mathcal{P}$ be a Promela-Lite specification with n processes for some $n > 0$ (i.e. there are n `run` statements in the `init { atomic {`$\ldots$`} }` block). We now detail the semantics of $\mathcal{P}$ as a Kripke structure $\mathcal{M}$. We show that if $\mathcal{P}$ is well-typed according to the type system of Section 6.2 then the Kripke structure $\mathcal{M}$ is well-defined.

For a well-formed type T, let $lit(T)$ denote the set of all possible literal values which can have type T in the specification $\mathcal{P}$. Thus $lit(int) = \mathbb{Z}$[1], $lit(pid) = \{0, 1, \ldots, n\}$ and $lit(chan\{\overline{T}\}) = \{c : c$ is the name of a static channel with $c : chan\{\overline{T}\}\} \cup \{$`null`$\}$. Note that typing rule T-NULL ensures that `null` is a literal value for any well-formed channel type.

We define the *domain* of a variable or static channel as follows. If x is a global or local variable of type T then the domain of x is $lit(T)$. If c is a static channel with $signature(c) = (\{T_1, T_2, \ldots, T_k\}, l)$ (for some $k, l > 0$) then the domain of c is the set:

$$\{[(a_{1,1}, a_{1,2}, \ldots, a_{1,k}), (a_{2,1}, a_{2,2}, \ldots, a_{2,k}), \ldots, (a_{m,1}, a_{m,2}, \ldots, a_{m,k})]$$
$$: 0 \leq m \leq l,\ a_{i,j} \in lit(T_j)\ (1 \leq i \leq m,\ 1 \leq j \leq k)\}.$$

This set consists of all possible sequences of messages for the channel, including the empty sequence [ ].

Let p be a proctype in $\mathcal{P}$, and x a parameter of $\mathcal{P}$. Suppose that $proctype(i) = p$ for some i ($1 \leq i \leq n$). We use $p[i].x$ to denote the local variable x for this process. If c is a channel with type $chan\{T_1, T_2, \ldots, T_k\}$, we use $\vec{a}$ as a shorthand for a message $(a_1, a_2, \ldots, a_k)$ on c (where $a_i : T_i$, $1 \leq i \leq k$).

6.3.1 States of a specification

A state of a Promela-Lite specification $\mathcal{P}$ can be expressed as an ordered tuple consisting of a value for each variable in the specification, using the notation preceding

1. In practice, $lit(int)$ is a finite range of integers which can be represented using a fixed word size.

```
chan A = [1] of {pid,chan{int}}; chan B = [2] of {int}; chan C = [2]
of {int}; pid leader = 0;

proctype user(chan{int} in; chan{int} out; int x) {
  ...
}

init {
  atomic {
     run user(B,C,0);
     run user(C,B,0);
  }
}
```

Figure 6.6: Part of a simple Promela-Lite specification.

Definition 1 (Section 2.2), where the domain of each variable is as described above. However, it is more convenient to reason about a state as a set of propositions.

If s is a set consisting of exactly one proposition of the form $(x = a)$ for each variable x in $\mathcal{P}$ (where a is a value in the domain of x), then s can be converted into a state by writing the value of each variable and static channel as an appropriately ordered tuple. Thus we can equivalently (and more conveniently) reason about a state as a set of assignments to variables.

Figure 6.6 shows part of a simple Promela-Lite specification with three static channels, `A`, `B` and `C`, a global variable `leader` and two instantiations of a *user* proctype. If we order the static channels and global variables as they appear in the specification, and order the local variables of *user* 1 before those of *user* 2, then an example state of the associated model is:

$$s = ([(1, B)], [4, 5], [\,], 1, B, C, 0, C, B, 0).$$

Using the equivalent set-based notation we have:

$$\begin{aligned} s \;\; = \;\; & \{(A = [(1, B)]), (B = [4, 5]), (C = [\,]), (leader = 0), \\ & (user[1].in = B), (user[1].out = C), (user[1].x = 0), \\ & (user[2].in = C), (user[2].out = B), (user[2].x = 0)\}. \end{aligned}$$

We will use the latter notation in the rest of this chapter, and in Chapter 7. The set S of (potential) states of $\mathcal{M}$ consists of every possible assignment to variables and channels of $\mathcal{P}$. As discussed in Footnote 1 (page 118), the range of allowed integer values is finite, thus S is a finite set.

6.3.2 Initial state

The values with which global variables are assigned on declaration, together with the parameter values which are passed to proctypes in *run* statements, determine the initial state of $\mathcal{M}$.

For a global variable x with $x : T$, let $init(x)$ denote the value in $lit(T)$ to which x is assigned at its declaration. For a local variable $p[i].x$ with $p[i].x : T$, let $init(p[i].x)$ denote the initial value in $lit(T)$ to which x is assigned in the ith `run` statement. $\mathcal{M}$ has a single initial state s_0, defined thus:

$$
\begin{aligned}
s_o \quad = \quad & \{(c = [\,]) : c \text{ is a static channel name in } \mathcal{P}\} \cup \\
& \{(x = init(x)) : x \text{ is a global variable of } \mathcal{P}\} \cup \\
& \{(p[i].x = init(p[i].x)) : x \text{ is a parameter of proctype } p \\
& \;\text{instantiated by the } i\text{th } run \text{ statement } (1 \leq i \leq n)\}
\end{aligned}
$$

6.3.3 Expression evaluation

We define a function $eval_{p,i}$ which takes a state $s \in S$ and an expression e of the form $\langle expr \rangle$ (see Figure 6.3), and returns the value of e when evaluated at s in the context of process i with $proctype(i) = p$. Let $s \in S$ be a state of $\mathcal{M}$. Then:

- $eval_{p,i}(s, x) = a$ if $(x = a) \in s$ (i.e. x is a global variable)
- $eval_{p,i}(s, x) = a$ if $(p[i].x = a) \in s$ (i.e. x is a local variable of p)
- $eval_{p,i}(s, c) = c$ if c is a static channel name or `null`
- $eval_{p,i}(s, a) = a$ if $a \in \mathbb{Z}$
- $eval_{p,i}(s, \texttt{\_pid}) = i$
- $eval_{p,i}(s, \texttt{len(}c\texttt{)}) = k$ if c is a static channel and $(c = [\vec{a_1}, \vec{a_2}, \ldots, \vec{a_m}]) \in s$ $(0 \leq k \leq cap(c))$
- $eval_{p,i}(s, \texttt{len(null)}) = 0$
- $eval_{p,i}(s, \texttt{len(}x\texttt{)}) = eval_{p,i}(s, len(c))$ if $(p[i].x = c) \in s$
- $eval_{p,i}(s, \texttt{(}e\texttt{)}) = eval_{p,i}(s, e)$
- $eval_{p,i}(s, e_1 \circ e_2) = eval_{p,i}(s, e_1) \circ eval_{p,i}(s, e_2)$ (where $\circ \in \{+, -, *\}$).

As discussed in Footnote 1 (page 118), $lit(int)$ is a finite range of integers in practice. Let $min(int)$ and $max(int)$ denote the minimum and maximum values in this range, and assume $min(int) < 0$. If the result $eval_{p,i}(s, e_1) \circ eval_{p,i}(s, e_2)$ falls outwith the allowed range, we define $eval_{p,i}(s, e_1 \circ e_2) = ((eval_{p,i}(s, e_1) \circ eval_{p,i}(s, e_2) + |min|) \bmod (max - min)) - |min|$. This definition means that the result of such a calculation is *truncated* so that e.g. $max(int) + 1 = min(int)$. This follows the approach used by SPIN to deal with out-of-range operations in Promela specifications [92].

6.3.4 Satisfaction of guards

We use the $eval_{p,i}$ function to define a relation $\models_{p,i}$ between states and guards which determines whether a guard holds at a given state. For a guard g of the form $\langle guard \rangle$ (see Figure 6.3) and a state $s \in S$, with p and i as above, $s \models_{p,i} g$ means that the state s satisfies the guard g in the context of p and i. The relation $\models_{p,i}$ is defined as follows:

- $s \models_{p,i} e_1 \bowtie e_2$ iff $eval_{p,i}(s, e_1) \bowtie eval_{p,i}(s, e_2)$ (where $\bowtie \in \{$==, !=, <, <=, >, >=$\}$)[2]
- $s \models_{p,i}$ $\texttt{nfull}(c)$ iff $(c = [\vec{a_1}, \vec{a_2}, \ldots, \vec{a_m}]) \in s$ and $cap(c) > m$, where c is a static channel
- $s \models_{p,i}$ $\texttt{nempty}(c)$ iff $(c = [\vec{a_1}, \vec{a_2}, \ldots, \vec{a_m}]) \in s$ and $m > 0$, where c is a static channel
- $s \models_{p,i}$ $\texttt{nfull}(x)/\texttt{nempty}(x)$ iff $(p[i].x = c) \in s$ and $s \models_{p,i}$ $\texttt{nfull}(c)/\texttt{nempty}(c)$, where x is a local declared channel of p
- $s \models_{p,i} \texttt{!}g$ iff $s \not\models_{p,i} g$
- $s \models_{p,i} g_1 \,\texttt{\&\&}\, g_2$ iff $s \models_{p,i} g_1$ and $s \models_{p,i} g_2$
- $s \models_{p,i} g_1 \,\texttt{||}\, g_2$ iff $s \models_{p,i} g_1$ or $s \models_{p,i} g_2$
- $s \models_{p,i} \texttt{(}g\texttt{)}$ iff $s \models_{p,i} g$.

6.3.5 Effect of updates

For a variable x, define $var(x) = \begin{cases} x & \text{if } x \text{ is a global variable} \\ p[i].x & \text{if } x \text{ is a local variable} \end{cases}$

For each update u described by the $\langle update \rangle$ rule in Figure 6.3, the effect of u on a state s (in the context of a process i with proctype p) is given in Figure 6.7. In each case we define the update u, the conditions under which u applies, and the result of applying u to s (denoted $exec_{p,i}(s, u)$). Given a sequence of updates $u_1, u_2, \ldots, u_k$ and a state s the rules of Figure 6.7 can be applied repeatedly to define the state reached by executing the u_i in sequence, starting in state s. The resulting state is denoted $exec_{p,i}(s, u_1; u_2; \ldots; u_k)$, where $exec_{p,i}(s, u_1; u_2; \ldots; u_k) = exec_{p,i}(\ldots exec_{p,i}(exec_{p,i}(s, u_1), u_2) \ldots, u_k)$.

Note that for certain updates it may be the case that *none* of the rules of Figure 6.7 are applicable. For example, suppose $(c = [a_1, a_2, \ldots, a_m]) \in s$, where $m = cap(c)$, i.e. the static channel c is full in state s. In this case there is no rule which defines the effect of executing '$c\,\texttt{!}\,e_1, e_2, \ldots, e_k$', since a condition of the rule for sending on static channels is that the channel must not be full. We say that $exec_{p,i}(s, u)$ is undefined if no rule of Figure 6.7 is applicable.

A state s is well-defined if it can be equivalently expressed as a tuple. This is the only the case if it contains exactly one proposition for each variable of $\mathcal{P}$. Thus for the resulting state t to be well-defined it must be the case that the rule corresponding to an update removes propositions about a distinct set of variables, then adds one proposition for each variable. For an arbitrary Promela-Lite specification this is not necessarily the case. Consider an update '$c\texttt{?}x, x$', where c is a static channel and x is a global variable. Suppose $(x = a) \in s$, $(c = [(a_1, a_2)]) \in s$ and $a_1 \neq a_2$. The rule for executing receive updates constructs state $exec_{p,i}(s, `c\texttt{?}x, x')$ by

2. Strictly, $\bowtie$ on the right hand side of 'iff' is $=, \neq, \leq$ or $\geq$ if $\bowtie$ on the left hand side is ==, != ,<= or >= respectively.

u	**Conditions on** s	**Resulting state** $exec_{p,i}(s,u)$
‘`skip`’	none	s
‘x `=` e’	$(var(x)=a)\in s$	$(s\setminus\{(var(x)=a)\})\cup\{(var(x)=eval_{p,i}(s,e))\}$
‘c`!`$e_1,e_2,\ldots,e_k$’	$(c=[\vec{a_1},\vec{a_2},\ldots,\vec{a_m}])\in s$ $s\models_{p,i}$ `nfull`(c)	$(s\setminus\{(c=[\vec{a_1},\vec{a_2},\ldots,\vec{a_m}])\})\cup$ $\{(c=[\vec{a_1},\vec{a_2},\ldots,\vec{a_m},(eval_{p,i}(s,e_1),eval_{p,i}(s,e_2),\ldots,$ $eval_{p,i}(s,e_k))])\}$
‘c`?`$x_1,x_2,\ldots,x_k$‘	$(c=[(a_{1,1},a_{1,2},\ldots,a_{1,k}),$ $\vec{a_2},\ldots,\vec{a_m}])\in s$ $s\models_{p,i}$ `nempty`(c) $(var(x_j)=b_j)\in s\ (1\leq j\leq k)$	$(s\setminus\{(c=[(a_{1,1},a_{1,2},\ldots,a_{1,k}),\vec{a_2},\ldots,\vec{a_m}]),$ $(var(x_1)=b_1),(var(x_2)=b_2),\ldots,(var(x_k)=b_k)\})\cup$ $\{(c=[\vec{a_2},\ldots,\vec{a_m}]),(var(x_1)=a_{1,1}),(var(x_2)=a_{1,2}),$ $\ldots,(var(x_k)=a_{1,k})\}$
‘x`!`$e_1,e_2,\ldots,e_k$’	$(p[i].x=c)\in s$	$exec_{p,i}(s,$‘c`!`$e_1,e_2,\ldots,e_k'$) (if well-defined)
‘x`?`$x_1,x_2,\ldots,x_k$’	$(p[i].x=c)\in s$	$exec_{p,i}(s,$‘c`?`$x_1,x_2,\ldots,x_k'$) (if well-defined)

Figure 6.7: Update execution rules. Each rule is interpreted in the context of process i which is an instantiation of proctype p.

removing $(x=a)$ from s, then adding the propositions $(x=a_1)$ and $(x=a_2)$. Thus $exec_{p,i}(s,$‘c`?`x,x') is not well-defined.

The following theorem states that, for a well-typed Promela-Lite specification $\mathcal{P}$, if the guard associated with a statement of $\mathcal{P}$ is satisfied at state $s\in\mathcal{M}$, then the rules of Figure 6.7 lead to a well-defined next-state t. In other words, the theorem shows that execution of a well-typed specification at a given state can always progress if some process has a guard which is true at the state. The proof is presented in Appendix B.1.

Theorem 11 *(Progress theorem) Let $\mathcal{P}$ be a well-typed Promela-Lite specification with associated model $\mathcal{M}$, s a state of $\mathcal{M}$,* `atomic` $\{$ g `->` u_1`;` u_2`;` $\ldots u_l$ $\}$ *a statement of proctype p, and i the identifier of an instantiation of p. Suppose $s\models_{p,i} g$. Then $exec_{p,i}(s,u_1;u_2;\ldots;u_l)$ is well-defined.*

From now on, when we refer to a Promela-Lite specification $\mathcal{P}$ we assume that $\mathcal{P}$ is well-typed.

6.3.6 Deriving a Kripke structure

Let $\mathcal{P}$ be a Promela-Lite specification. The states S and initial state s_0 of $\mathcal{M}$ are as defined above. The transition relation R is defined as follows. Let $s\in S$ and let `atomic` $\{$ g `->` u_1`;` u_2`;` $\ldots$`;` u_k $\}$ be a statement of proctype p in $\mathcal{P}$. Suppose process i is an instantiation of p. If $s\models_{p,i} g$ then $(s,exec_{p,i}(s,u_1;u_2;\ldots;u_k))\in R$. By Theorem 11, $exec_{p,i}(s,u_1;u_2;\ldots;u_k)$ is well-defined.

6.4 Promela-Lite → Promela

Promela-Lite is not a subset of Promela since it includes extended notation for channel types, and the built-in `null` constant.

Let $\mathcal{P}$ be a Promela-Lite specification. Then $\mathcal{P}$ can be converted into a Promela specification as follows. Firstly, unfold all recursive type expressions in $\mathcal{P}$ so that they have the form $chan\{\overline{T}\}$ (where the types comprising $\overline{T}$ may be recursive). Secondly, replace every type expression of the form $chan\{T\}$ with *chan*. Finally, add the declaration `chan null = [0] of` $\{T\}$ to the beginning of the specification, where T is any Promela type (e.g. `bit`).

The Promela-Lite semantics described in Section 6.3 are based on: the semantics for Promela described informally in [92], four years of SPIN use, and the SPIN source code. The semantics presented in Section 6.3 have been designed so that if $\mathcal{P}$ is a well-typed Promela-Lite specification and $\mathcal{P}'$ the corresponding Promela specification then $\mathcal{P}$ and $\mathcal{P}'$ have the same associated model.

In Appendix C.1 we discuss, in detail, the Promela features which Promela-Lite omits.

6.5 Example: Load-balancing

To illustrate Promela-Lite we now discuss an example specification of a message passing system, given in Figure 6.8. The specification consists of three *server* processes, six *client* processes and three *load-balancer* processes. A particular *client* has been blocked by the system, indicated by the global *pid* variable *blocked_client*.

A *load-balancer* process continuously receives requests sent by *client* processes. A request consists of two parts: the identity of a *client* (derived from its `_pid` variable), and the input channel of the *client*. If the message is from the blocked *client* then the *load-balancer* sends back the value 0, indicating that the request has been denied. Otherwise the *load-balancer* forwards the name of the input channel of the given *client* to the server with the shortest queue of incoming messages (choosing non-deterministically between servers which share the shortest queue length). On receiving a *client* channel name, a *server* uses it to send the value 1 to the *client*, which abstractly represents the result of the request.

The specification has a dynamic communication structure since channel references are passed between processes.

Summary

In order to allow the rigorous development of automatic symmetry detection techniques for Promela, we have presented the syntax, type system and Kripke structure semantics for Promela-Lite, a specification language which captures the essential features of Promela, but is easier to work with in practice. We have illustrated Promela-Lite using a specification of a load-balancing system.

```
chan se1 = [3] of {chan{int}};
chan se2 = [3] of {chan{int}};
chan se3 = [3] of {chan{int}};

chan lb1 = [1] of {pid,chan{int}};
chan lb2 = [1] of {pid,chan{int}};
chan lb3 = [1] of {pid,chan{int}};

chan cl1 = [1] of {int}; chan cl2 = [1] of {int};
chan cl3 = [1] of {int}; chan cl4 = [1] of {int};
chan cl5 = [1] of {int}; chan cl6 = [1] of {int};

pid blocked_client = 9;

proctype loadbalancer(chan{pid,chan{int}} in;
           chan{int} client_link; pid client_id; int pc) {
  do
    :: atomic { pc==1 && nempty(in) -> in?client_id,client_link;
                 pc = 2 }
    :: atomic { pc==2 && client_id!=blocked_client -> pc = 3 }
    :: atomic { pc==2 && client_id==blocked_client &&
                 nfull(client_link) -> client_link!0; pc = 4 }
    :: atomic { pc==3 && len(se1)<=len(se2) && len(se1)<=len(se3)
                 && nfull(se1) -> se1!client_link; pc = 4 }
    :: atomic { pc==3 && len(se2)<=len(se1) && len(se2)<=len(se3)
                 && nfull(se2) -> se2!client_link; pc = 4 }
    :: atomic { pc==3 && len(se3)<=len(se1) && len(se3)<=len(se2)
                 && nfull(se3) -> se3!client_link; pc = 4 }
    :: atomic { pc==4 -> client_id = 0; client_link = null; pc = 1 }
  od
}

proctype server(chan{chan{int}} in; chan{int} client_link; int pc) {
  do
    :: atomic { pc==1 && nempty(in) -> in?client_link; pc = 2 }
    :: atomic { pc==2 && nfull(client_link) -> client_link!1;
                 pc = 3 }
    :: atomic { pc==3 -> client_link = null; pc = 1 }
  od
}

proctype client(chan{int} in; chan{pid,chan{int}} lb;
           int response; int pc) {
  do
    :: atomic { pc==1 && nfull(lb) -> lb!_pid,in; pc = 2 }
    :: atomic { pc==2 && nempty(in) -> in?response; pc = 3 }
    :: atomic { pc==3 -> response = -1; pc = 1 }
  od
}

init {
  atomic {
    run server(se1,null,1); run server(se2,null,2);
    run server(se3,null,3); run loadbalancer(lb1,null,0,1);
    run loadbalancer(lb2,null,0,1); run loadbalancer(lb3,null,0,1);
    run client(cl1,lb1,-1,1); run client(cl2,lb1,-1,1);
    run client(cl3,lb2,-1,1); run client(cl4,lb2,-1,1);
    run client(cl5,lb3,-1,1); run client(cl6,lb3,-1,1);
  }
}
```

Figure 6.8: Promela-Lite specification of a load-balancing system.

Chapter 7

Finding Symmetry by Static Channel Diagram Analysis

The examples in Chapter 4 have led us to identify some problems with existing symmetry detection techniques using scalarsets and input language restriction. These approaches cannot handle certain kinds of symmetry which arise from the communication structure of a system, and they place undue restrictions on the form of specifications; in particular the way in which process identifiers may be used. In Chapter 5 we established a correspondence between channel diagram automorphisms and Kripke structure automorphisms for these example specifications.

In this chapter we introduce the *static* channel diagram of a Promela-Lite specification. This diagram type is similar to the channel diagram, but can be extracted by syntactic inspection of a specification even if the associated model is intractably large. We formally establish a general correspondence between automorphisms of the static channel diagram and automorphisms of the Kripke structure associated with a Promela-Lite specification.

We present a symmetry detection technique based on this correspondence, which can be summarised as follows: generators for a group of *candidate* symmetries for a Promela-Lite specification are found by analysing the static channel diagram of the specification. 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. These symmetries can then be used for reduced model checking.

Unlike previous approaches to symmetry detection, our approach can detect *arbitrary* component symmetries arising from the communication structure of a specification. The approach can be fully automated (as we demonstrate in Chapter 8), and requires no additional information from the user. The only requirement is that the specification satisfies certain restrictions which we formally describe using a type system. The restrictions can be automatically checked, and are less strict than those imposed by the scalarset data type or the SMC input language.

At the end of this chapter we discuss various ways in which the technique could be extended to further reduce restrictions on the form of a specification, and to capture symmetry between global variables. We emphasise that the static channel diagram is merely used as a heuristic for finding a good group of candidate symmetries, and discuss other possible diagram types.

7.1 Static Channel Diagrams

Let $\mathcal{P}$ be a Promela-Lite specification with n processes. Let $V_P = \{1, 2, \ldots, n\}$ be the set of process identifiers, and V_C the set of static channel names for $\mathcal{P}$. Recall from rules T-SEND and T-RECV (Figure 6.5, Section 6.2) that a Promela-Lite statement involves at most one send or receive update, and this update must appear at the beginning of the sequence of updates for the statement.

Definition 27 *The* static channel diagram *associated with $\mathcal{P}$ is a coloured, bipartite digraph $\mathcal{SCD}(\mathcal{P}) = (V, E, C)$ where:*

- *$V = V_P \cup V_C$ is the set of process identifiers and static channel names in $\mathcal{P}$*
- *For $i \in V_P$, $c \in V_C$ and $proctype(i) = p$,*
  - *$(i, c) \in E$ iff p has a statement of the form '`atomic` $\{$ g `->` $\langle name \rangle$`!`e_1`,`e_2`,` $\ldots$`,`e_k`;` u_2`;` $\ldots$`;` u_l $\}$' where $\langle name \rangle$ is c, or $\langle name \rangle$ is a parameter of p initialised with value c*
  - *$(c, i) \in E$ iff p has a statement of the form '`atomic` $\{$ g `->` $\langle name \rangle$`?`x_1`,`x_2`,` $\ldots$`,`x_k`;` u_2`;` $\ldots$`;` u_l $\}$' where $\langle name \rangle$ is c, or $\langle name \rangle$ is a parameter of p initialised with value c*
- *C is a colouring function defined by $C(v) = proctype(v)$ if $v \in V_P$, and $C(v) = signature(v)$ if $v \in V_C$.*

The difference between the static channel diagram of a Promela-Lite specification and the channel diagram of a Promela specification (Definition 26, Section 5.1) is that the channel diagram records all possible channel-based communication, whereas the static channel diagram records *potential* communication on certain channels. The static channel diagram of a specification can be seen as a static approximation of the communication structure for the specification. It does not capture communication arising from dynamic passing of channel references, and edges of the diagram may result from send/receive updates which in practice cannot be executed in any reachable state of $\mathcal{M}$.

7.1.1 Deriving static channel diagrams

Given a Promela-Lite specification $\mathcal{P}$, $\mathcal{SCD}(\mathcal{P})$ can be efficiently derived via a single pass of $\mathcal{P}$. The node set and colouring can be deduced immediately from the declaration of static channels and the `run` statements.

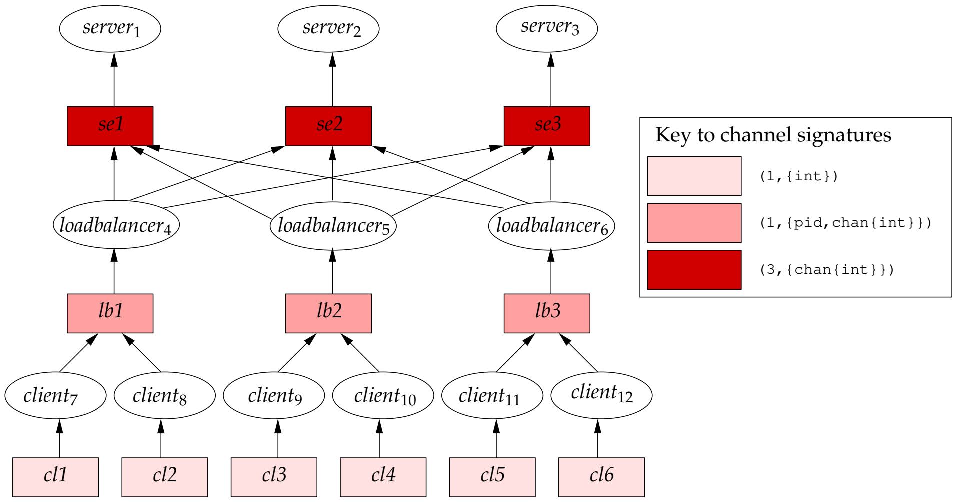


Figure 7.1: Static channel diagram associated with the load-balancer specification (Figure 6.8).

If a proctype p involves an explicit send/receive on static channel c then an edge $(i, c)/(c, i)$ is added to the diagram for each $i \in V_{\mathcal{P}}$ such that $proctype(i) = p$. Each channel parameter x of p is marked as a send parameter and/or a receive parameter if p contains an update of the form $x!e_1, e_2, \ldots, e_k$ and/or $x?x_1, x_2, \ldots, x_k$. For each $i \in V_{\mathcal{P}}$ with $proctype(i) = p$, suppose the actual value for x in the ith `run` statement is c (where c is a static channel name). If x is marked as a send/receive parameter then an edge $(i, c)/(c, i)$ is added to the diagram.

The next result follows from the above discussion:

Proposition 4 *Let $\mathcal{P}$ be a Promela-Lite specification. The complexity of deriving $\mathcal{SCD}(\mathcal{P})$ from $\mathcal{P}$ is linear in the size of $\mathcal{P}$.*

Therefore, unlike deriving the channel diagram of a Promela specification (Section 5.1.1), it is possible to derive $\mathcal{SCD}(\mathcal{P})$ from a Promela-Lite specification $\mathcal{P}$ even if $\mathcal{M}$ is intractably large.

Figure 7.1 shows the static channel diagram for the Promela-Lite specification of the load-balancer system, given in Figure 6.8. The graphical notation is similar to that for channel diagrams introduced in Section 5.1. Note that there are no outgoing edges from the *server* processes to the *client* input channels. This is because communication from a *server* process to a *client* channel is achieved *dynamically*, using the channel reference passed to a *server* by one of the *load-balancer* processes.

The state-space associated with the associated Promela version of the specification (derived using the method described in Section 6.4) is intractably large, thus we cannot compute its associated channel diagram.

7.2 Static Channel Diagram Automorphisms

An automorphism of the static channel diagram $\mathcal{SCD}(\mathcal{P}) = (V, E, C)$ is an automorphism of the directed, coloured graph (V, E, C) (see Definition 19, Section 3.1.5). The group of all automorphisms of $\mathcal{SCD}(\mathcal{P})$ is denoted $Aut(\mathcal{SCD}(\mathcal{P}))$. This is analogous to the notion of a channel diagram automorphism (see Section 5.1.2). For $\alpha \in Aut(\mathcal{SCD}(\mathcal{P}))$ we define $\alpha(0) = 0$ and $\alpha(\texttt{null}) = \texttt{null}$, where 0 and `null` are the default values used by variables of type *pid* and *chan* respectively.

Since $\mathcal{SCD}(\mathcal{P})$ is a small graph (its size is proportional to the size of $\mathcal{P}$), the group $Aut(\mathcal{SCD}(\mathcal{P}))$ can be efficiently computed directly using a standard algorithm such as *nauty* [123], or via GRAPE as described in Section 5.1.2 (for channel diagrams). Let $\mathcal{P}$ denote the load-balancer specification of Figure 6.8. The static channel diagram $\mathcal{SCD}(\mathcal{P})$ is shown in Figure 7.1. Using GRAPE we find:

$$\begin{aligned} Aut(\mathcal{SCD}(\mathcal{P})) \quad = \quad & \langle (7\ 8)(cl1\ cl2), (9\ 10)(cl3\ cl4), (11\ 12)(cl5\ cl6), \\ & (4\ 5)(lb1\ lb2)(7\ 9)(cl1\ cl3)(8\ 10)(cl2\ cl4), \\ & (5\ 6)(lb2\ lb3)(9\ 11)(cl3\ cl5)(10\ 12)(cl4\ cl6), \\ & (1\ 2)(se1\ se2), (2\ 3)(se1\ se2)\rangle. \end{aligned}$$

Recall that i is the pid of the ith proctype inistantated in the `init` process.

It is straightforward to check that each generator of this group is indeed an automorphism of $\mathcal{SCD}(\mathcal{P})$. We have used GAP to show that $Aut(\mathcal{SCD}(\mathcal{P})) \cong S_3 \times (S_2 \wr S_3)$. Intuitively, the group $S_2 \wr S_3$ arises due to symmetry within each of the three blocks of *clients* (the group S_2), combined with symmetry between the three blocks (the group S_3). The group S_3 on the left hand side of the direct product corresponds to permutation of the *server* processes (and their associated channels).

We now define the image of $\mathcal{P}$ under an element of $Aut(\mathcal{SCD}(\mathcal{P}))$, and an action of $Aut(\mathcal{SCD}(\mathcal{P}))$ on the states of $\mathcal{M}$.

7.2.1 Image of $\mathcal{P}$ under $\alpha \in Aut(\mathcal{SCD}(\mathcal{P}))$

Let $\mathcal{P}$ be a Promela-Lite specification and $\alpha \in Aut(\mathcal{SCD}(\mathcal{P}))$. The specification $\alpha(\mathcal{P})$ is obtained from $\mathcal{P}$ by replacing every applied occurrence of a static channel name c with $\alpha(c)$; every occurrence of a value $a \in \{1, 2, \ldots, n\}$ in a *pid* context (see Section 6.2) with $\alpha(a)$, and permuting the order of `run` statements so that `run` statement i appears in position $\alpha(i)$ in $\alpha(\mathcal{P})$ $(1 \leq i \leq n)$.

Similarly, given an expression e, guard g, update u or statement s of $\mathcal{P}$, the expression $\alpha(e)$, guard $\alpha(g)$, update $\alpha(u)$ or statement $\alpha(s)$ is obtained by replacing every static channel name c and *pid* literal a with $\alpha(c)$ and $\alpha(a)$ respectively.

7.2.2 Action of $Aut(\mathcal{SCD}(\mathcal{P}))$ on the states of $\mathcal{M}$

Let $\alpha \in Aut(\mathcal{SCD}(\mathcal{P}))$. We first define the effect of α on propositions which refer to variables and static channels of $\mathcal{P}$:

- Let $(x = a)$ be a proposition referring to a global variable x with $x : T$ and $a \in lit(T)$. If $T = pid$ then $\alpha((x = a)) = (x = \alpha(a))$, otherwise $\alpha((x = a)) = (x = a)$.
- Let $(p[i].x = a)$ be a proposition referring to a local variable x of process i, with $x : T$ and $a \in lit(T)$. If $T = pid$ or $T = chan\{\overline{T}\}$ then $\alpha((p[i].x = a)) = (p[\alpha(i)] = \alpha(a))$. Otherwise $\alpha((p[i].x = a)) = (p[\alpha(i)] = a)$. Since α preserves the colouring of processes according to their proctype, process $\alpha(i)$ is also an instantiation of proctype p and therefore the local variable $p[\alpha(i)]$ exists. Thus the action of α is well-defined.
- Let $(c = [\vec{a_1}, \vec{a_2}, \ldots, \vec{a_m}])$ be a proposition referring to a static channel c with signature $(\{\overline{T}\}, l)$ where $0 \leq m \leq l$. Then $\alpha((c = [\vec{a_1}, \vec{a_2}, \ldots, \vec{a_m}])) = (\alpha(c) = [\vec{a_1}^{\alpha}, \vec{a_2}^{\alpha}, \ldots, \vec{a_m}^{\alpha}])$. If $\vec{a_i} = (a_1, a_2, \ldots, a_k)$ then $\vec{a_i}^{\alpha} = (b_1, b_2, \ldots, b_k)$ where $b_i = \alpha(a_i)$ if $T_i = pid$ or $T_i = chan\{\overline{U}\}$ and $b_i = a_i$ otherwise. The action of α is well-defined as α preserves the signature of static channels.

Let $\mathcal{M} = (S, s_0, R)$ be the model associated with $\mathcal{P}$. Recall that a state $s \in S$ is a set of propositions, one for each variable and static channel of $\mathcal{P}$. The state $\alpha(s)$ is defined as follows: $\alpha(s) = \{\alpha(z) : z \in s\}$.

For all $s \in S$ and $\alpha, \beta \in Aut(\mathcal{SCD}(\mathcal{P}))$, it is clear that $(\alpha\beta)(s) = \alpha(\beta(s))$ and $id(s) = s$, therefore the definition of $\alpha(s)$ is an *action* of $Aut(\mathcal{SCD}(\mathcal{P}))$ on S (see Definition 13, Section 3.1.3).

7.3 Correspondence Result

Let ρ be the permutation representation of $Aut(\mathcal{SCD}(\mathcal{P}))$ corresponding to its action on S. By Theorem 3 (Section 3.1.3), $\rho(Aut(\mathcal{SCD}(\mathcal{P}))) \leq Sym(S)$. Now $Aut(\mathcal{M}) \leq Sym(S)$, but we cannot, in general, say anything about the relationship between $\rho(Aut(\mathcal{SCD}(\mathcal{P})))$ and $Aut(\mathcal{M})$ with respect to the subgroup relation.

In this section we define what it means for an element of $Aut(\mathcal{SCD}(\mathcal{P}))$ to be *valid* for $\mathcal{P}$, and show that the set of all valid elements of $Aut(\mathcal{SCD}(\mathcal{P}))$ form a subgroup $G \leq Aut(\mathcal{SCD}(\mathcal{P}))$. We prove that if $\alpha \in Aut(\mathcal{SCD}(\mathcal{P}))$ is valid for $\mathcal{P}$ then $\rho(\alpha) \in Aut(\mathcal{M})$. Thus $\rho(G) \leq Aut(\mathcal{M})$. The relationship between the various groups is illustrated in Figure 7.2.

7.3.1 Valid elements of $Aut(\mathcal{SCD}(\mathcal{P}))$

We say that two Promela-Lite specifications $\mathcal{P}_1$ and $\mathcal{P}_2$ are equivalent, and write $\mathcal{P}_1 \equiv \mathcal{P}_2$, if they are identical up to re-arrangement of statements in the `do...od`

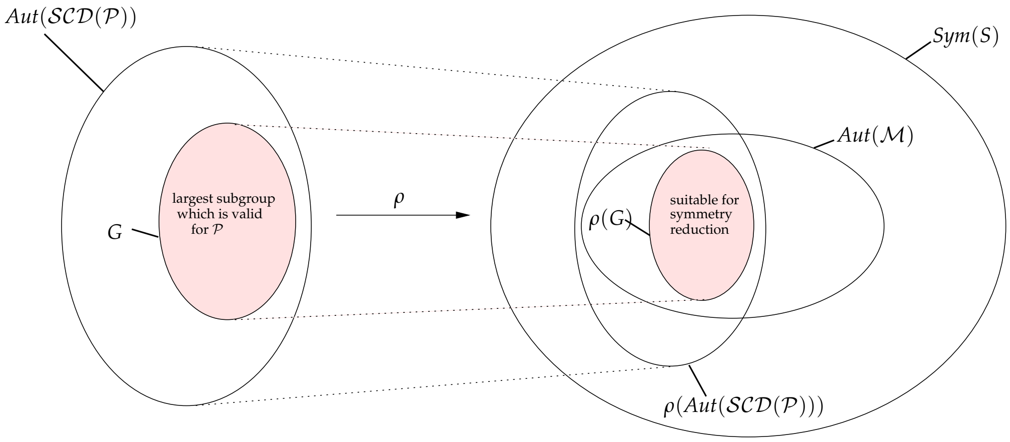


Figure 7.2: Relationship between valid automorphisms of $\mathcal{SCD}(\mathcal{P})$ and automorphisms of $\mathcal{M}$.

construct, and operands to the commutative operators `+`, `*`, `&&` and `||`. For brevity, we then say that $\mathcal{P}_1$ and $\mathcal{P}_2$ are identical "up to re-arrangement".

An element $\alpha \in Aut(\mathcal{SCD}(\mathcal{P}))$ is *valid* for $\mathcal{P}$ if $\alpha(\mathcal{P}) \equiv \mathcal{P}$.

Theorem 12 *Let* $G = \{\alpha \in Aut(\mathcal{SCD}(\mathcal{P})) : \alpha$ *is valid for* $\mathcal{P}\}$. *Then* $G \leq Aut(\mathcal{SCD}(\mathcal{P}))$.

Proof Since $id(\mathcal{P}) = \mathcal{P}$, clearly $id(\mathcal{P}) \equiv \mathcal{P}$, thus $id \in G$. Associativity is inherited from $Aut(\mathcal{SCD}(\mathcal{P}))$. Let $\alpha, \beta \in G$. Then $\alpha(\mathcal{P})$ and $\beta(\mathcal{P})$ are identical to $\mathcal{P}$ up to re-arrangement. It follows that $\alpha\beta(\mathcal{P}) \equiv \mathcal{P}$ (by successively applying the rearrangements of α to those of β), i.e. $\alpha\beta \in G$. Since $Aut(\mathcal{SCD}(\mathcal{P}))$ is finite, $\alpha^{-1} = \alpha^k$ for some $k > 0$, thus $\alpha^{-1} \in G$ by the above argument. The result follows. ■

If H is a subgroup of $Aut(\mathcal{SCD}(\mathcal{P}))$ such that every element of H is valid for $\mathcal{P}$ we say that H is valid for $\mathcal{P}$. The group G of Theorem 12 is the largest valid subgroup of $Aut(\mathcal{SCD}(\mathcal{P}))$.

To check whether $\mathcal{P} \equiv \alpha(\mathcal{P})$ for $\alpha \in Aut(\mathcal{SCD}(\mathcal{P}))$, we use a function *normalise*. The specification $normalise(\mathcal{P})$ is obtained from $\mathcal{P}$ by sorting the statements in the `do...od` loop of a proctype and the operands of commutative operators, using the natural ordering on strings. It is clear that if two specifications are *equal* after normalisation then they are *equivalent*. Thus $\alpha \in Aut(\mathcal{SCD}(\mathcal{P}))$ is valid for $\mathcal{P}$ if $normalise(\mathcal{P}) = normalise(\alpha(\mathcal{P}))$. This provides an efficient, conservative test of validity for elements of $Aut(\mathcal{SCD}(\mathcal{P}))$. Since the complexity of sorting a list of length k is $O(k \log(k))$, we have:

Proposition 5 *The complexity of checking whether* $\mathcal{P} \equiv \alpha(\mathcal{P})$ *is* $O(|\mathcal{P}| \log(|\mathcal{P}|))$.

7.3.2 Main result

In this section we prove the following theorem:

Theorem 13 *Let $\mathcal{P}$ be a Promela-Lite specification, and $\alpha \in Aut(\mathcal{SCD}(\mathcal{P}))$. If α is valid for $\mathcal{P}$ then $\rho(\alpha) \in Aut(\mathcal{M})$.*

The proof of Theorem 13 uses two technical lemmas, proofs of which are given in Appendix B.2.

Lemma 1 *If $\alpha \in Aut(\mathcal{SCD}(\mathcal{P}))$ and g is a guard in $\mathcal{P}$ then $s \models_{p,i} g \Leftrightarrow \alpha(s) \models_{p,\alpha(i)} \alpha(g)$.*

Lemma 2 *Let $u_1, u_2, \ldots, u_k$ be updates of $\mathcal{P}$, $\alpha \in Aut(\mathcal{SCD}(\mathcal{P}))$ and s a state such that $exec_{p,i}(s, u_1; u_2; \ldots; u_k)$ is well-defined. Then $exec_{p,\alpha(i)}(\alpha(s), \alpha(u_1); \alpha(u_2); \ldots; \alpha(u_k)) = \alpha(exec_{p,i}(s, u_1; u_2; \ldots; u_k))$.*

Proof of Theorem 13 By Definition 20 (Section 3.2), we must show that (i) if $(s,t) \in R$ then $(\alpha(s), \alpha(t)) \in R$, and (ii) $\alpha(s_0) = s_0$.

If $(s,t) \in R$ then there is a process with pid i such that $proctype(i) = p$ (for some proctype p), and a statement z in p such that the guard of z holds for process i at s, and execution of the updates of z by process i at s leads to state t. Since $\alpha(\mathcal{P})$ is identical to $\mathcal{P}$ up to re-arrangement, the statement $\alpha(z)$ (possibly re-arranged) also appears in proctype p. By Lemma 1, the guard of $\alpha(z)$ holds for process $\alpha(i)$ at $\alpha(s)$, and by Lemma 2, execution of the updates of $\alpha(z)$ by process $\alpha(i)$ at $\alpha(s)$ leads to state $\alpha(t)$. Therefore $(\alpha(s), \alpha(t) \in R$.

We must show that for any proposition $(v = d)$ in s_0, $\alpha((v = d)) \in s_0$ also. In s_0, all static channels are empty, so for any static channel c, the propositions $(c = [\,])$ and $\alpha((c = [\,])) = (\alpha(c) = [\,])$ both belong to s_0. For each global variable x, $(x = x_0) \in s_0$, where x_0 is the initial value for x (specified at declaration). If $x : int$ then $\alpha((x = x_0)) = (x = x_0) \in s_0$. If $x : pid$ then we must have $\alpha(x_0) = x_0$ (otherwise we would not have $\alpha(\mathcal{P}) \equiv \mathcal{P}$), so $\alpha((x = x_0)) = (x = \alpha(x_0)) = (x = x_0) \in s_0$. Let x be a global variable name with $x : int$, and let $init(x)$ be the value in $lit(int)$ to which x is assigned on declaration. Then $(x = init(x)) \in s_0$, and $\alpha((x = init(x)) = (x = init(x)) \in s_0$. Suppose on the other hand $x : pid$, and let $init(x)$ be the value in $lit(int)$ to which x is assigned on declaration. Then $(x = init(x)) \in s_0$, and $\alpha((x = init(x)) = (x = \alpha(init(x)))$. Since $\alpha(\mathcal{P}) \equiv \mathcal{P}$, the declaration of x in $\mathcal{P}$ must exactly match the declaration of x in $\alpha(\mathcal{P})$, so we must have $\alpha(init(x)) = init(x)$. Thus $\alpha((x = init(x))) = (x = init(x)) \in s_0$. ■

7.4 Finding the Largest Valid Subgroup of $Aut(\mathcal{SCD}(\mathcal{P}))$

We showed in Section 7.3.1 (Theorem 12) that the set G consisting of all elements of $Aut(\mathcal{SCD}(\mathcal{P}))$ which are valid for $\mathcal{P}$ is a subgroup of $Aut(\mathcal{SCD}(\mathcal{P}))$. G is thus the largest subgroup of $Aut(\mathcal{SCD}(\mathcal{P}))$ which is valid for $\mathcal{P}$.

In this section we present an algorithm to find this subgroup. First we establish some preliminary results. For the relevant group theoretic definitions, see Section 3.1.

Lemma 3 *Let X be a set of generators for $Aut(\mathcal{SCD}(\mathcal{P}))$. Let $X' = \{\alpha \in X : \alpha \text{ is valid for } \mathcal{P}\}$. Then $\langle X' \rangle$ is valid for $\mathcal{P}$.*

Proof By definition of G, $X' \subseteq G$. Therefore $\langle X' \rangle \leq G$, and the result follows. ■

Lemma 4 *Suppose $H \leq Aut(\mathcal{SCD}(\mathcal{P}))$ is valid for $\mathcal{P}$ and $\alpha \in Aut(\mathcal{SCD}(\mathcal{P}))$ is valid for $\mathcal{P}$. Then $\langle H \cup \{\alpha\} \rangle$ is valid for $\mathcal{P}$.*

Proof Since H is valid for $\mathcal{P}$, $H \leq G$. Similarly, since α is valid for $\mathcal{P}$, $\alpha \in G$. Thus $H \cup \{\alpha\} \subseteq G$. It follows from Lemma 3 that $\langle H \cup \{\alpha\} \rangle \leq G$, and the result follows. ■

Our algorithm for finding G starts with a known valid subgroup H of $Aut(\mathcal{SCD}(\mathcal{P}))$, and adds valid coset representatives (see Definition 6, Section 3.1.1) to the generators of H to obtain successively larger valid subgroups. The following lemma is used to determine when G has been found.

Lemma 5 *Suppose $H \leq Aut(\mathcal{SCD}(\mathcal{P}))$ and H is valid for $\mathcal{P}$. Let $\{\alpha_1, \alpha_2 \ldots, \alpha_k\}$ be a set of coset representatives for H in $Aut(\mathcal{SCD}(\mathcal{P}))$, where $\alpha_1 \in H$, $\alpha_i \in Aut(\mathcal{SCD}(\mathcal{P})) \setminus H$ for $2 \leq i \leq k$ and $k = |Aut(\mathcal{SCD}(\mathcal{P}))|/|H|$. Suppose $\alpha_2, \ldots, \alpha_k$ are not valid for $\mathcal{P}$. Then H is the unique largest valid subgroup of $Aut(\mathcal{SCD}(\mathcal{P}))$ (i.e. $H = G$).*

Proof Since H is valid for $\mathcal{P}$, $H \leq G$. Suppose $H \subset G$. Then there exists $\alpha \in G$ with $\alpha \notin H$. So $H\alpha$ is a right coset of H in $Aut(\mathcal{SCD}(\mathcal{P}))$, and $H\alpha = H\alpha_i$ for some $2 \leq i \leq k$. Since $\alpha \in G$, $H\alpha \subseteq G$, so $H\alpha_i \subseteq G$ and thus $\alpha_i \in G$. This is a contradiction since G is valid for $\mathcal{P}$. Hence $H = G$. ■

Algorithm 4 can be used to compute the largest valid subgroup G of $Aut(\mathcal{SCD}(\mathcal{P}))$.

Theorem 14 *Algorithm 4 computes the largest valid subgroup of $Aut(\mathcal{SCD}(\mathcal{P}))$.*

Proof By Lemmas 3 and 4, the group H computed by Algorithm 4 is valid for

Algorithm 4 Algorithm to find the largest valid subgroup of $Aut(\mathcal{SCD}(\mathcal{P}))$

$X :=$ generators of $Aut(\mathcal{SCD}(\mathcal{P}))$
$H := \langle\{\alpha \in X : \alpha(\mathcal{P}) \equiv \mathcal{P}\}\rangle$
$U :=$ representatives of right cosets of H in $Aut(\mathcal{SCD}(\mathcal{P}))$ except H
while $U \neq \emptyset$ **do**
  $U := U \setminus \{\alpha\}$
  if $\alpha(\mathcal{P}) \equiv \mathcal{P}$ **then**
    $H := \langle H \cup \{\alpha\}\rangle$
    if $|Aut(\mathcal{SCD}(\mathcal{P}))|/|H| < |U|$ **then**
      $U :=$ representatives of right cosets of H in $Aut(\mathcal{SCD}(\mathcal{P}))$ except H
    end if
  end if
end while

$\mathcal{P}$. The group H is the largest subgroup of $Aut(\mathcal{SCD}(\mathcal{P}))$ which is valid for $\mathcal{P}$ by Lemma 5. ■

We illustrate Algorithm 4 using the load-balancer example. Let $\mathcal{P}$ be the specification of Figure 6.8. Generators for $Aut(\mathcal{SCD}(\mathcal{P}))$ computed by GRAPE are given in Section 7.2. The generators which do not fix the process identifier 9 are *not* valid for $\mathcal{P}$ since, if α is one of these generators, the declaration `pid blocked_client = 9;` in $\mathcal{P}$ is replaced with `pid blocked_client =` $\alpha(9)$, and $\alpha(9) \neq 9$, thus $\alpha(\mathcal{P}) \not\equiv \mathcal{P}$. The other generators are valid for $\mathcal{P}$, therefore:

$$\begin{aligned} H \;=\; & \langle (7\ 8)(cl1\ cl2), (11\ 12)(cl5\ cl6), \\ & (1\ 2)(se1\ se2), (2\ 3)(se1\ se2)\rangle \end{aligned}$$

is valid for $\mathcal{P}$. GAP tells us that $|Aut(\mathcal{SCD}(\mathcal{P}))| = 288$ and $|H| = 24$, so there are $|Aut(\mathcal{SCD}(\mathcal{P}))|/|H| = 12$ cosets of H in $Aut(\mathcal{SCD}(\mathcal{P}))$. We can use GAP to compute representatives $\alpha_1, \alpha_2, \ldots, \alpha_{11}$ for the 11 cosets of H in $Aut(\mathcal{SCD}(\mathcal{P}))$ which are distinct from H. We find that the first nine of these are not valid for $\mathcal{P}$, but $\alpha_{10} = (4\ 6)(lb1\ lb3)(7\ 11)(cl1\ cl5)(8\ 12)(cl2\ cl6)$ is valid for $\mathcal{P}$. This element is added to the generators of H, and we find $|H| = 48$, so there are now $|Aut(\mathcal{SCD}(\mathcal{P}))|/|H| = 6$ cosets of H in $Aut(\mathcal{SCD}(\mathcal{P}))$. However, it is more efficient to check the final original coset representative α_{11} than to compute and check a new set of coset representatives. This is the purpose of the innermost conditional statement in Algorithm 4. We find that α_{11} is not valid for $\mathcal{P}$, thus:

$$\begin{aligned} H \;=\; & \langle (7\ 8)(cl1\ cl2), (11\ 12)(cl5\ cl6), \\ & (1\ 2)(se1\ se2), (2\ 3)(se1\ se2) \\ & (4\ 6)(lb1\ lb3)(7\ 11)(cl1\ cl5)(8\ 12)(cl2\ cl6)\rangle \end{aligned}$$

is the largest subgroup of $Aut(\mathcal{SCD}(\mathcal{P}))$ which is valid for $\mathcal{P}$.

Algorithm 4 performs badly if the initial group H is small, and $Aut(\mathcal{SCD}(\mathcal{P}))$ very large. If H is the largest valid subgroup then $(|Aut(\mathcal{SCD}(\mathcal{P}))|/|H|) - 1$ coset representatives must be checked. We discuss the implementation of Algorithm 4, together with a group theoretic optimisation, in Section 8.3.3.

7.5 Generalising Static Channel Diagram Automorphisms

Let $\mathcal{P}$ be a Promela-Lite specification with associated model $\mathcal{M}$. We have shown that automorphisms of $\mathcal{M}$ can be derived from the group $Aut(\mathcal{SCD}(\mathcal{P}))$. We now define a group $Aut(\Psi(\mathcal{P}))$ such that $Aut(\mathcal{SCD}(\mathcal{P})) \leq Aut(\Psi(\mathcal{P}))$, and show that our techniques can be generalised so that automorphisms of $\mathcal{M}$ can be derived from $Aut(\Psi(\mathcal{P}))$.

Definition 28 *$\Psi(\mathcal{P})$ is a coloured graph $\Psi(\mathcal{P}) = (V, \emptyset, C)$ where:*

- *$V = V_P \cup V_C$ is the set of process identifiers and static channel names in $\mathcal{P}$*
- *C is a colouring function defined by $C(v) = proctype(v)$ if $v \in V_P$, and $C(v) = signature(v)$ if $v \in V_C$.*

The graph $\Psi(\mathcal{P})$ could be obtained from $\mathcal{SCD}(\mathcal{P})$ by removing all of the edges of $\mathcal{SCD}(\mathcal{P})$, although it is trivial to obtain Ψ from $\mathcal{P}$. The group $Aut(\Psi(\mathcal{P}))$ is the subgroup of $Sym(V)$ which preserves the colouring C, i.e. $Aut(\Psi(\mathcal{P})) = \{\alpha \in Sym(V) : C(v) = C(\alpha(v)) \ \forall \ v \in V\}$.

The techniques presented in this chapter were motivated by the correspondence between channel diagram and Kripke structure automorphisms observed in Chapter 4. However, if $Aut(\mathcal{SCD}(\mathcal{P}))$ is replaced with $Aut(\Psi(\mathcal{P}))$ consistently throughout Sections 7.3 and 7.4, the correspondence result still holds, and Algorithm 4 can be used to find the largest valid subgroup of $Aut(\Psi(\mathcal{P}))$.

We show that the largest valid subgroup of $Aut(\Psi(\mathcal{P}))$ is the same as the largest valid subgroup of $Aut(\mathcal{SCD}(\mathcal{P}))$:

Theorem 15 *Let $\mathcal{P}$ be a Promela-Lite specification, and G the largest subgroup of $Aut(\Psi(\mathcal{P}))$ which is valid for $\mathcal{P}$. Then $G \leq Aut(\mathcal{SCD}(\mathcal{P}))$ and G is the largest subgroup of $Aut(\mathcal{SCD}(\mathcal{P}))$ which is valid for $\mathcal{P}$.*

Proof $\Psi(\mathcal{P}) = (V, \emptyset, C)$ and $\mathcal{SCD}(\mathcal{P}) = (V, E, C)$, where $V = V_P \cup V_C$. Let $\alpha \in G$. Suppose $\alpha \notin Aut(\mathcal{SCD}(\mathcal{P}))$. Since $\alpha \in Aut(\Psi(\mathcal{P}))$, α preserves the colouring C, therefore there must be an edge $(u, v) \in E$ such that $(\alpha(u), \alpha(v)) \notin E$. By definition of $\mathcal{SCD}(\mathcal{P})$, we have $(u, v) = (i, c)$ or $(u, v) = (c, i)$ for some $i \in V_P$ and $c \in V_C$.

Suppose $(u, v) = (i, c)$. By Definition 27 (Section 7.1) there is a proctype p in $\mathcal{P}$ such that $proctype(i) = p$ and p contains a statement z which involves a write on a static channel c, or a local variable of p initialised with value c in `run` statement i. Since $\alpha(\mathcal{P}) \equiv \mathcal{P}$, the statement $\alpha(z)$ (possibly re-arranged) also appears in proctype p. If z involves a write on static channel c then $\alpha(z)$ involves a write on static channel $\alpha(c)$. If z involves a write on a local variable of p initialised with value c in `run` statement i then $\alpha(z)$ involves a write on the same variable which, since $\alpha(\mathcal{P}) \equiv \mathcal{P}$, is initialised with value $\alpha(c)$ in `run` statement $\alpha(i)$. In both cases, since $proctype(\alpha(i)) = p$, $(\alpha(i), \alpha(c)) \in E$. If $(u, v) = (c, i)$ then, by a similar argument, $(\alpha(c), \alpha(i)) \in E$.

This is a contradiction, so we must have $(u, v) \in E \Rightarrow (\alpha(u), \alpha(v)) \in E$, i.e. $\alpha \in Aut(\mathcal{SCD}(\mathcal{P}))$. The result follows. ■

We have $G \leq Aut(\mathcal{SCD}(\mathcal{P})) \leq Aut(\Psi(\mathcal{P}))$, where G is the largest valid subgroup of $Aut(\Psi(\mathcal{P}))$. Since $Aut(\mathcal{SCD}(\mathcal{P}))$ is usually smaller than $Aut(\Psi(\mathcal{P}))$ it is more practical to search for G in $Aut(\mathcal{SCD}(\mathcal{P}))$ than $Aut(\Psi(\mathcal{P}))$. For example, let $\mathcal{P}$ be the load-balancing specification (see Figure 6.8). Then $\Psi(\mathcal{P})$ is obtained by removing all the edges from $\mathcal{SCD}(\mathcal{P})$ shown in Figure 7.1. Any permutation which maps a process/channel node coloured with a given proctype name or channel signature to a similarly coloured process/channel node is an automorphism of $\Psi(\mathcal{P})$. Using GRAPE we find:

$$\begin{aligned} Aut(\Psi(\mathcal{P})) = \langle \quad & (1\,2), (2\,3), (4\,5), (5\,6), (7\,8), \\ & (8\,9), (9\,10), (10\,11), (11\,12), \\ & (cl5\ cl6), (cl4\ cl5), (cl3\ cl4), \\ & (cl2\ cl3), (cl1\ cl2), (lb2\ lb3), \\ & (lb1\ lb2), (se2\ se3), (se1\ se2) \quad \rangle \end{aligned}$$

and $|Aut(\Psi(\mathcal{P}))| = 671{,}846{,}400$. None of the generators of $Aut(\Psi(\mathcal{P}))$ are valid for $\mathcal{P}$. On the other hand, in Section 7.4 we showed that $|Aut(\mathcal{SCD}(\mathcal{P}))| = 288$, and that the initial valid subgroup generated by the valid generators of $Aut(\mathcal{SCD}(\mathcal{P}))$ has size 24. The largest valid subgroup of $Aut(\mathcal{SCD}(\mathcal{P}))$ (and thus of $Aut(\Psi(\mathcal{P}))$) was shown to have size 48. Computing within $Aut(\mathcal{SCD}(\mathcal{P}))$ rather than $Aut(\Psi(\mathcal{P}))$ reduces the problem of searching the whole of $Aut(\Psi(\mathcal{P}))$ for 48 elements to searching a small set of coset representatives in a much smaller group.

$Aut(\mathcal{SCD}(\mathcal{P}))$ can be thought of as a good upper bound for valid symmetries, from which the least upper bound G can be computed. An open research problem is to determine whether there is an alternative diagram to $\mathcal{SCD}(\mathcal{P})$, $\Gamma(\mathcal{P})$ say, such that for any Promela-Lite specification $\mathcal{P}$, $\Gamma(\mathcal{P})$ can be extracted from $\mathcal{P}$ in polynomial time and $Aut(\Gamma(\mathcal{P})) = G$, the largest valid subgroup of $Aut(\Psi(\mathcal{P}))$.

Guards (ctd.)

$$\frac{\Gamma \vdash e : pid \qquad \Gamma \vdash a \in \{0,1,\ldots,n\} \qquad \bowtie \in \{<,<=,>,>=\}}{\Gamma \vdash e \bowtie a\ OK} \ (\text{T-RELATIONAL-PID-LIT})$$

$$\frac{\Gamma \vdash e : pid \qquad \Gamma \vdash a \in \{0,1,\ldots,n\} \qquad \bowtie \in \{<,<=,>,>=\}}{\Gamma \vdash a \bowtie e\ OK} \ (\text{T-RELATIONAL-LIT-PID})$$

Figure 7.3: Typing rules to allow *pid* expressions to be compared with literal values using relational operators.

7.6 Extending the Techniques

The Promela-Lite syntax and type system place fewer restrictions on the use of *pid* literals and expressions than those associated with scalarset variables in Murϕ/SymmSpin (see Definition 22, Section 3.3.2), or index variables in SMC (see Section 3.3.3). In particular, literal *pid* values can be referred to explicitly in expressions and updates. However, relational and arithmetic operations involving *pid* expressions are still not allowed.

The technique presented in this chapter can handle arbitrary kinds of symmetry which arise from the static channel diagram of a specification. However, the technique may perform badly if $Aut(\mathcal{SCD}(\mathcal{P}))$ is large but has a small largest valid subgroup. In addition, symmetry between global variables cannot be detected.

In this section we outline some ways in which certain restrictions on the use of *pid* expressions can be relaxed, and sketch how the static channel diagram can be extended to include information regarding global variables, allowing symmetries between global variables to be captured. We also show how the technique can be generalised to apply to a more abstract diagram type of which the static channel diagram is a special case.

7.6.1 Allowing relational operators with *pid* arguments

Since $lit(pid)$ is a finite set of integers ($\{0,1,\ldots,n\}$), if e is an expression with $e : pid$ and $a \in \{0,1,\ldots,n\}$, the guard $e < a$ can be re-written as a disjunction: `(e==0||e==1||... e==`$a-1$`)` (where $a-1$ denotes a value rather than an arithmetic expression). The guards $e > a$, $e <= a$, $e >= a$, $a < e$, $a > e$, $a <= e$ and $a >= e$ can be expanded in a similar way.

Suppose we extend the type system to include the typing rules given in Figure 7.3. These rules allow expressions of *pid* type to be compared relationally with *pid* literals. Let $\mathcal{P}'$ be a Promela-Lite specification which is well-typed in this extended type system. Let $\mathcal{P}$ be the specification obtained from $\mathcal{P}'$ by expanding every guard $e \bowtie a$ or $a \bowtie e$ (where $e : pid$ is either a variable name or `_pid`, and $a \in \{0,1,\ldots,n\}$) using the method described above. Clearly $\mathcal{P}'$ is a Promela-Lite program which is well-typed with respect to the original type system, and $\mathcal{P}'$ and $\mathcal{P}$ have identical associated models. Thus our symmetry detection techniques can be applied to $\mathcal{P}$ to obtain a group of static channel diagram automorphisms suitable for symmetry reduction when model checking $\mathcal{P}'$.

This straightforward expansion technique makes Promela-Lite less restrictive and so allows our automatic symmetry detection techniques to apply to a wider range of specifications.

7.6.2 Symmetrically invariant operations

The Promela-Lite type system prohibits the use of process identifiers in arithmetic operations (rule T-ARITH). This restriction is typical of techniques for symmetry identification (see Sections 3.3.2 and 3.3.3). However, as discussed in Section 4.6.1 for the hypercube example, it is not necessarily the case that arithmetic operations of this kind destroy symmetry.

An arithmetic operation involving only literal integers and *pid* variables $x_1, x_2, \ldots, x_k$ can be thought of as a function $f(x_1, x_2, \ldots, x_k)$. Given an element $\alpha \in Aut(\mathcal{SCD}(\mathcal{P}))$, if k is small then it is feasible to check whether $\alpha(f(a_1, a_2, \ldots, a_k)) = f(\alpha(a_1), \alpha(a_2), \ldots, \alpha(a_k))$ for every combination of values $a_i \in \{1, 2, \ldots, n\}$. If this is true we say that f is *invariant* under α.

For example, in a uni-directional ring network with five processes, the update `next = (current%5)+1` could be used to find the neighbour or the process identified by *current*. Let $\alpha \in Aut(\mathcal{SCD}(\mathcal{P}))$ have the form $(1\ 2\ 3\ 4\ 5)\beta$, where β is a permutation of static channels. It is easy to check that, for any value of $current \in \{1, 2, 3, 4, 5\}$, $\alpha((current\%5) + 1) = (\alpha(current)\%5 + 1)$.

Suppose we extend the type system to allow an update of the form $x = f(x_1, x_2, \ldots, x_k)$ where $x, x_1, \ldots, x_k$ are *pid* variables, as long as the enclosing statement has a guard of the form g `&&` x_1`!=0 &&` x_2`!=0 && ... &&` x_k`!=0`. This ensures that the operation f is not applied to arguments which have the value 0, as 0 represents a default *pid* value.

If $\mathcal{P}$ is well-typed according to the extended type system, we can replace a statement of the form `atomic {` g `&&` x_1`!=0 &&` x_2`!=0 && ...&&` x_k`!=0 -> ...;` x `=` $f(x_1, x_2, \ldots, x_k)$`; ... }` (where $x, x_1, \ldots, x_k$ are variables with type *pid*) with n^k distinct statements, each of the form:

`atomic {` g `&&` x_1`==`a_1 `&&` x_2`==`a_2 `&& ... &&` x_k`==`a_k `->`
`...;` x `=` $f(a_1, a_2, \ldots, a_k)$`; ... }`

where $a_i \in \{1, 2, \ldots, n\}$ $(1 \leq i \leq n)$ and $f(a_1, a_2, \ldots, a_k)$ is the value of f for this input. Since f only involves the x_i and constant values, this value can be statically computed. If the resulting specification is well-typed according to the original type system (i.e. if each value $f(a_1, a_2, \ldots, a_k) \in lit(pid)$) then the standard technique can be applied. In this case the complexity of checking whether α is valid for $\mathcal{P}$ is still polynomial in the size of $\mathcal{P}$, but the size of $\mathcal{P}$ is now $O(n^k)$ where k is the highest arity of any arithmetic function involving *pid* variables, thus the two approaches are essentially equivalent. Using the above example, the statement:

`atomic {` g `&& current!=0 -> ...; next = (current%5)+1; ... }`

is replaced by five statements:

atomic { g `&& current==1 -> ...; next = 2; ...` }
atomic { g `&& current==2 -> ...; next = 3; ...` }
atomic { g `&& current==3 -> ...; next = 4; ...` }
atomic { g `&& current==4 -> ...; next = 5; ...` }
atomic { g `&& current==5 -> ...; next = 1; ...` }

Clearly this approach is not practical if k is large, in which case more sophisticated techniques are required.

7.6.3 Capturing symmetry between global variables

If global variables are used for communication in a specification $\mathcal{P}$ then automorphisms of the associated model $\mathcal{M}$ may arise due to permutations of the variables. The symmetry detection technique presented above cannot handle this kind of symmetry, since the static channel diagram $\mathcal{SCD}(\mathcal{P})$ does not capture the relationship between processes and global variables.

We now sketch an extension of our technique to deal with this kind of symmetry, based on the notion of an *extended* static channel diagram. For a Promela-Lite specification $\mathcal{P}$, let V_G denote the set of global variable names for $\mathcal{P}$.

Definition 29 *Let $\mathcal{SCD}(\mathcal{P}) = (V', E', C')$ be the static channel diagram associated with $\mathcal{P}$. The extended static channel diagram associated with $\mathcal{P}$ is a coloured, tripartite digraph $\mathcal{ESCD}(\mathcal{P}) = (V, E, C)$ where:*

- *$V = V_P \cup V_C \cup V_G$ is the set of process identifiers, static channel names and global variable names in $\mathcal{P}$*
- *If $e \in E'$ then $e \in E$*
- *For $i \in V_P$, $x \in V_G$ and $proctype(i) = p$,*
  - *$(i, x) \in E$ iff p has an update of the form $x = e$*
  - *$(x, i) \in E$ iff p has an update of the form $y = e$ where the expression e refers to global variable x*
- *C is a colouring function defined by $C(v) = C'(v)$ if $v \in V'$, and $C(v) = type(v)$ if $v \in V_G$.*

This definition is identical to Definition 27 (Section 7.1) except that $\mathcal{ESCD}(\mathcal{P})$ includes nodes for global variables, and edges between process identifiers and global variables. An edge from a process identifier to a global variable node is included if the process can potentially update the variable; an edge from a global variable node to a process identifier is included if the result of an update made by the process can potentially be affected by the value of the variable.

The group $Aut(\mathcal{ESCD}(\mathcal{P}))$ is the set of all automorphisms of the directed, coloured graph $\mathcal{ESCD}(\mathcal{P})$ (see Definition 19, Section 3.1.5). Given $\alpha \in$

$Aut(\mathcal{ESCD}(\mathcal{P}))$, the definition of $\alpha(\mathcal{P})$ is similar to the case where $\alpha \in \mathcal{SCD}(\mathcal{P})$, except that each applied occurrence of a global variable name x in $\mathcal{P}$ is replaced with $\alpha(x)$ in $\alpha(\mathcal{P})$, a declaration `int` $x = init(x)$ is replaced with `int` $x = init(\alpha(x))$, and a declaration `pid` $x = init(x)$ is replaced with `pid` $x = \alpha(init(\alpha(x)))$.

If $\mathcal{M} = (S, s_0, R)$ is the model associated with $\mathcal{P}$, the action of $Aut(\mathcal{ESCD}(\mathcal{P}))$ on S is similar to that of $Aut(\mathcal{SCD}(\mathcal{P}))$, except that if x is a global variable $\alpha((x = a)) = (\alpha(x) = a)$ if $x : int$, and $\alpha((x = a)) = (\alpha(x) = \alpha(a))$ if $x : pid$.

The statements and proofs of Lemmas 1 and 2 and Theorem 13 are readily adapted to show that, for $\alpha \in Aut(\mathcal{ESCD}(\mathcal{P}))$, if $\alpha(\mathcal{P}) \equiv \mathcal{P}$ then $\rho(\alpha) \in Aut(\mathcal{M})$. It is trivial to modify the computational group theoretic approach of Section 7.4 in order to compute the largest valid subgroup of $Aut(\mathcal{ESCD}(\mathcal{P}))$.

7.6.4 Extending the notion of validity

In the following example we show that our notion of validity may be unnecessarily restrictive. Let $\mathcal{P}$ be a Promela specification with associated model $\mathcal{M}$, p a proctype of $\mathcal{P}$, x, y, z local variables of p with $x : pid$ and $y, z : int$, and $\alpha \in Aut(\mathcal{SCD}(\mathcal{P}))$. Suppose α maps 2 to 1 and 1 to 2, and $\alpha(\mathcal{P}) \equiv \mathcal{P}$, so that $\rho(\alpha) \in Aut(\mathcal{M})$.

Assume that the body of p begins with the following two statements:

```
atomic { x==1 && y!=3 && z!=4 -> x=0; }
atomic { x==2 && y!=3 && z!=4 -> x=0; }
```

so that the body of p in $\alpha(p)$ begins with the same statements in a different order. Clearly we can re-write these statements as follows:

```
atomic { x==1 && y!=3 && z!=4 -> x=0; }
atomic { x==2 && (!(y!=3 || z!=4)) -> x=0; }
```

without changing $\mathcal{M}$. In this case, we still have $\rho(\alpha) \in Aut(\mathcal{M})$. However, the body of p in $\alpha(\mathcal{P})$ now begins with the statements:

```
atomic { x==2 && y!=3 && z!=4 -> x=0; }
atomic { x==1 && (!(y!=3 || z!=4)) -> x=0; }
```

Assuming that $\mathcal{P}$ does not happen to also include these statements, the bodies of p in $\mathcal{P}$ and $\alpha(p)$ are *not* the same up to re-arrangement, i.e. $\alpha(\mathcal{P}) \not\equiv (\mathcal{P})$.

Nevertheless, our approach to checking the validity of elements is safe and fast, and is sufficient for most sensibly written specifications. It would be possible to extend our techniques to employ a more sophisticated equivalence check, e.g. by using a theorem prover.

Summary

We have defined the *static channel diagram* $\mathcal{SCD}(\mathcal{P})$ associated with a Promela-Lite specification $\mathcal{P}$, and shown that it can be efficiently computed via a single pass of $\mathcal{P}$. After defining a group action of the automorphism group $Aut(\mathcal{SCD}(\mathcal{P}))$ on the states S of $\mathcal{M}$, the model associated with $\mathcal{P}$, we have proved that there is a largest *valid* subgroup $G \leq Aut(\mathcal{SCD}(\mathcal{P}))$ for which $\rho(G) \leq Aut(\mathcal{M})$, where ρ is the permutation representation of the group action. Furthermore, we have presented a computational group theoretic algorithm for computing the group G. This technique allows a subgroup of $Aut(\mathcal{M})$ to be efficiently derived from the specification $\mathcal{P}$.

We have shown that our technique can be generalised to apply to any subgroup of $Aut(\Psi(\mathcal{P}))$, but that $Aut(\mathcal{SCD}(\mathcal{P}))$ can be a good candidate group for efficient symmetry detection. We have discussed extensions to the approach which allow certain relational and arithmetic operations involving process identifier variables, and the detection of symmetry between global variables. In addition, we have suggested how the notion of valid automorphisms could be extended.

Chapter 8

SymmExtractor – an Automatic Symmetry Detection Tool for Promela

In this chapter we describe SymmExtractor, an automated symmetry detection tool for Promela which we have developed, based on the static channel diagram analysis techniques of Chapter 7. After providing an overview of the tool, we discuss the restrictions on the form of a Promela specification which must be satisfied before SymmExtractor can be applied. We then discuss two problems related to type-checking which SymmExtractor solves: how to deduce the type of an incompletely specified channel in order to check the validity of static channel diagram automorphisms, and how to convert recursive channel types to a canonical form to allow comparison when constructing a static channel diagram.

We discuss the way in which the GAP and *saucy* tools are used to compute the largest valid subgroup of $Aut(\mathcal{SCD}(\mathcal{P}))$, and provide experimental results showing how SymmExtractor performs on a variety of specifications based on the motivating examples of Chapter 4.

In order to assess the practical feasibility of the restrictions imposed by SymmExtractor we have carried out a user study, applying SymmExtractor to a set of Promela examples written as solutions to two student assessed exercises. We present the results of this evaluation, which highlight some mismatches between the restrictions imposed by SymmExtractor and the specification styles used in practical Promela examples.

8.1 An Overview of SymmExtractor

SymmExtractor is a Java program based on a Promela parser generated using the SableCC compiler generation framework [62]. The Promela grammar is adapted from a BNF grammar presented in [92], with the SPIN source code used to resolve ambiguity in the grammar specification.

The abstract syntax tree representation of the input specification is type-checked, and type reconstruction is used to obtain the full types of all channels in the specification. Reconstructed channel types which are *recursive* are then

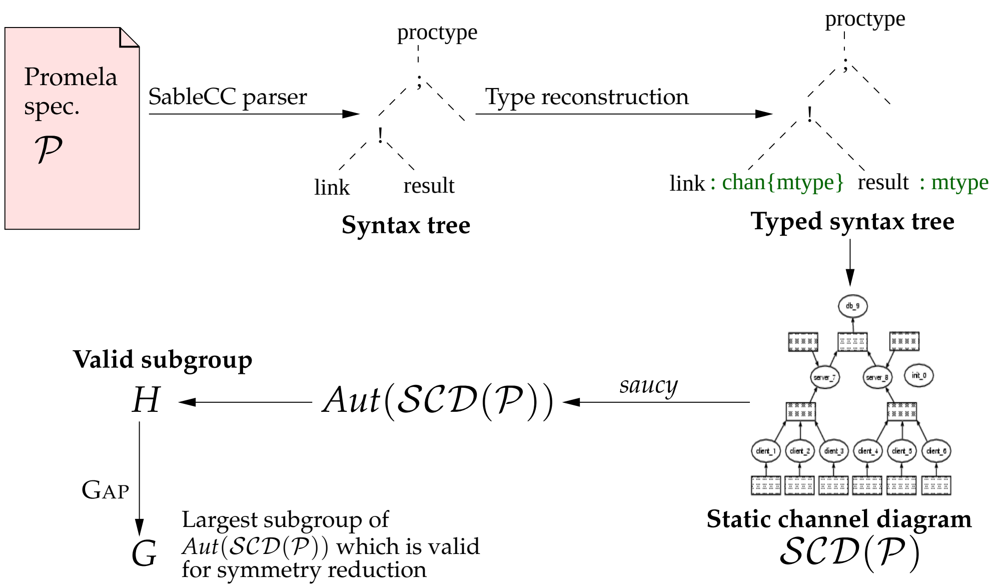


Figure 8.1: The automatic symmetry detection processes used by SymmExtractor.

converted to a minimised canonical form. The typed abstract syntax tree is then checked to see whether it satisfies certain restrictions imposed by the theory of Chapter 7. If these restrictions are satisfied then the static channel diagram $\mathcal{SCD}(\mathcal{P})$ for the specification $\mathcal{P}$ is derived, and its automorphisms are computed using the *saucy* program [37]. Algorithm 4 of Section 7.4 is then used to compute the largest subgroup of $Aut(\mathcal{SCD}(\mathcal{P}))$ which is valid for $\mathcal{P}$. Checking validity depends on the reconstructed type information obtained by SymmExtractor, and GAP is used to calculate sets of coset representatives.

The automatic symmetry detection process is summarised in Figure 8.1. The SymmExtractor implementation is embedded in our symmetry reduction package TopSPIN (see Chapter 11), and is available online. Instructions on how to use SymmExtractor are included in the TopSPIN manual, Appendix C.2. We discuss various aspects of our automatic symmetry detection process in the remainder of this chapter.

8.1.1 Summary of the restrictions imposed by SymmExtractor

As discussed in Chapter 6, Promela includes a number of language features which are not included in Promela-Lite. Most of these features could be handled by a straightforward extension of the results of Chapter 7, and are therefore supported by SymmExtractor. On the other hand, there are certain features of Promela for which the theory of Chapter 7 cannot obviously be extended. These features are not supported by SymmExtractor.

Appendix C.1 provides a detailed summary of non-Promela-Lite features which SymmExtractor does and does not support. We now summarise the restrictions on the form of a Promela specification which SymmExtractor requires, and automatically checks:

1. The `init` process must have the form

   $\texttt{init \{}$
   $\quad\texttt{atomic \{}$
   $\quad\quad\texttt{run } \langle \textit{name} \rangle_1\texttt{(...);}$
   $\quad\quad\texttt{run } \langle \textit{name} \rangle_2\texttt{(...);}$
   $\quad\quad\quad\vdots$
   $\quad\quad\texttt{run } \langle \textit{name} \rangle_n\texttt{(...);}$
   $\quad\quad\langle \textit{statement-list}, \text{`}\texttt{;}\text{'} \rangle$
   $\quad\texttt{\};}$
   $\quad\langle \textit{statement-list}, \text{`}\texttt{;}\text{'} \rangle$
   $\texttt{\}}$

   The list $\langle \textit{statement-list}, \text{`}\texttt{;}\text{'} \rangle$ of statements within the `atomic` block must be assignments of literal values to distinct variables. This is explained in Section 8.3.2. The $n$ `run` statements must be the only `run` statements occurring in the specification.
2. All global channel declarations must include a channel initialiser (see Section 2.4.1), and names of global channels must be treated as constants.
3. Variables of type *pid* must only be assigned to values in the range $\{0, 1, \ldots, n\}$, to other *pid* variables or to the `_pid` constant, and may not be used as operands to arithmetic operators.
4. An array must either be indexed by *pid* variables and literal values in the range $\{0, 1, \ldots, n\}$, or by *byte* variables and literal values in the range $\{0, 1, \ldots, 255\}$. The former case is only permissible if the array is declared with size $n + 1$.
5. The assignment $x = y$, where x and y are *chan* variables, is only permissible if x and y have the same channel type.

In Section 8.5 we investigate the implications of these restrictions in practice by studying a set of Promela specifications written as solutions to student assessed exercises.

8.2 Typechecking Promela

When designing Promela-Lite, to ease presentation of our theoretical results we included notation for fully specifying channel types, and for defining recursive types. In Promela, the type of a first class channel can only be partially specified using a channel initialiser, and while recursive types cannot be explicitly declared, they can be used implicitly as we demonstrate in Section 8.2.2.

```
chan A = [1] of {chan};
chan B = [1] of {pid,int};

proctype Q() {
  chan C;
  A?C;
  C!3,4;
  ...
}

proctype R() {
  A!B;
  ...
}
```

Figure 8.2: Promela example where type information is partially specified.

We now show that full type information for a Promela specification can be obtained using *type reconstruction*, and that resulting recursive types can be stored canonically via a minimisation process. This complete type information allows the theoretical results of Chapter 7 to be applied to Promela specifications.

8.2.1 Reconstructing channel types

As noted in Section 6.4, a key difference between Promela-Lite and Promela is that channel types in Promela-Lite are fully specified, whereas certain Promela channel types are only partially specified.

Consider a Promela specification $\mathcal{P}$ which includes the fragment of code shown in Figure 8.2. The local variable C of Q is declared to be a channel, but the type of messages it accepts is unspecified. Messages for channel A are references to channels, but the type of these channels is not specified in the initialiser for A. The type for channel B is fully specified – $B : chan\{pid, int\}$. A value for C is obtained via the statement $A?C$, by which C is assigned to some global channel name which has been sent on A by another process (an instantiation of proctype R, for example). Let $\alpha = (3\ 4)$. We cannot deduce the form of $\alpha(\mathcal{P})$ without knowing the complete type of C. If $C : chan\{pid, pid\}$ then the statement '$C!3,4$' is replaced in $\alpha(\mathcal{P})$ with '$C!\alpha(3),\alpha(4)$', i.e. '$C!4,3$'. On the other hand if $B : chan\{pid, int\}$ then the corresponding statement in $\alpha(\mathcal{P})$ is '$C!\alpha(3),4$', i.e. '$C!4,4$'.

Complete type information for A and therefore C can be obtained using *constraint-based type reconstruction* (also known as type inference) [139]. We explain this process using the Promela fragment of Figure 8.2.

The channel type information which *is* available from the specification is recorded, and is annotated with type variables X_i, $i \in \mathbb{N}$, which record missing type information, as shown in the top left panel of Figure 8.3.

Each time a channel name is used for communication in the specification, a *constraint* is posted. For example, the statement $A?C$ implies that channel A must accept single-field messages where the field has the same type as C. We know already that A accepts single type messages of type *chan* X_1,[1] and we know that $C : chan\ X_2$, so we can post the constraint $chan\ X_2 = chan\ X_1$. Constraints are posted similarly for the other communication statements. We use the notation *pid*/*int* to denote a type which is either *int* or *pid*, and use this notation to handle the literal values 3 and 4 which (out of context) can be assigned either type. The constraints for our example, together with the statements from which they arise, are shown in the top right panel of Figure 8.3.

The resulting system of constraints is then solved using a process known as *unification*. This process checks whether the system is consistent, and if so provides concrete values for type variable. If we attempt to unify the constraints $X_2 = X_1$, $X_2 = \{pid/int, pid/int\}$ and $X_1 = \{pid, int\}$, then since X_2 is a tuple of size 2, X_1 must also be a tuple of size 2. Furthermore, each entry of the tuple for X_2 must match the corresponding entry in X_1. This is the case since *pid*/*int* matches *pid* in the first case, and *pid*/*int* matches *int* in the second. Since $X_1 = \{pid, int\}$ is a stricter constraint than $X_2 = \{pid/int, pid/int\}$, $\{pid, int\}$ is taken as a concrete value for X_1 and X_2. The unification process is illustrated in the middle panel of Figure 8.3.

If the constraints shown in Figure 8.3 are the only constraints which arise from $\mathcal{P}$ relating to channels A, B and C, then the complete types for A, B and C are *reconstructed* as shown at the bottom of the figure, by substituting type variables for their concrete values. Armed with this additional type information, with $\alpha = (3\ 4)$ as above, we can unambiguously assert that the statement '`C!3,4`' should be replaced with '`C!4,4`' in $\alpha(\mathcal{P})$. This example shows that type reconstruction is critical to our automatic symmetry detection techniques.

Unification *fails* when the system of constraints is inconsistent: in this case the unification process should stop and report a type error. Unification of consistent constraints may not provide a concrete value for all type variables if, for example, a channel is never used. If we declare `chan` A `= [1] of {chan}`, but never use A, then A will be assigned the type $chan\{chan\ Y\}$ where Y is a type variable, but no constraints relating to Y will be posted. For the purposes of automatic symmetry detection we can simply assign $Y = \{int\}$ in this case.

For a more general description of constraint-based type reconstruction, see [139]. Our implementation is based on an algorithm described in [1].

8.2.2 Dealing with recursive types

Let $\mathcal{P}$ be a Promela specification. Recall from Definition 27, Section 7.1, that two channel nodes in the static channel diagram for $\mathcal{P}$ are coloured the same if they have the same channel signature. In order to construct the colouring function asso-

1. Note that the type expression *chan* X_1 denotes a channel with accepts a tuple of messages, where both the arity of the tuple and the type of each message field are unknown. This is different from the expression $chan\{X_1\}$ – a channel with this type accepts messages comprised of a *single* field of unknown type.

Initial type information, annotated with type variables	**Constraints posted based on channel usage**
$A \ : \ chan\{chan\ X_1\}$	$A?C \ \rightarrow \ chan\ X_2 = chan\ X_1$
$B \ : \ chan\{pid, int\}$	$C?3,4 \ \rightarrow \ X_2 = \{pid/int, pid/int\}$
$C \ : \ chan\ X_2$	$A\,!\,B \ \rightarrow \ chan\ X_1 = chan\{pid, int\}$

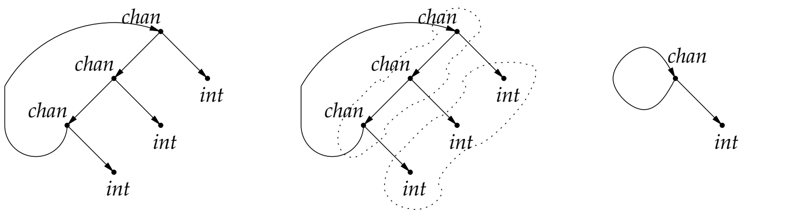
Solution to system of constraints

$$\begin{aligned} & chan\ X_2 = chan\ X_1 \ \wedge\ X_2 = \{pid/int, pid/int\} \ \wedge\ chan\ X_1 = chan\{pid, int\} \\ \Rightarrow\quad & X_2 = X_1 \ \wedge\ X_2 = \{pid/int, pid/int\} \ \wedge\ X_1 = \{pid, int\} \\ \Rightarrow\quad & X_1 = X_2 = \{pid, int\} \end{aligned}$$

Reconstructed channel types

$A : chan\{chan\{pid, int\}\}$ $\qquad B : chan\{pid, int\}$ $\qquad C : chan\{pid, int\}$

Figure 8.3: Type reconstruction for a simple example.



Graph for original type expression — Equivalence classes under bisimulation — Graph for minimised type expression

Figure 8.4: Minimisation of the recursive type $chan\{chan\{rec\ X\,.\,chan\{X, int\}, int\}, int\}$.

ciated with $\mathcal{SCD}(\mathcal{P})$ it is necessary to be able to compare channel types for equality.

This is straightforward, unless the types are recursive (see Section 6.1.2). Although Promela does not include syntax for specifying recursive channel types, they can be implied by channel usage. Consider the channel declaration `chan` A `= [1] of {chan}`, and the statement $A\,!\,A$. Using constraint-based type reconstruction, we record that $A : chan\{chan\ X\}$ from the declaration of A, where X is a type variable. We then post the constraint $X = \{chan\ X\}$ according to the statement $A\,!\,A$. Since X appears on both sides of this equation, X is defined *recursively*. We can assign to A the recursive type $rec\ X\ .\ chan\{X\}$. This kind of channel usage has been employed in realistic Promela specifications, e.g. a specification of a telephone system [20].

Due to the manner in which type reconstruction works, we may end up with the *same* recursive type appearing in many different forms. Suppose that a specification includes channels A, B and C, and that after applying type reconstruction we find $A : rec\ X\,.\,chan\{X, int\}$, $B : chan\{chan\{rec\ X\,.\,chan\{X, int\}, int\}, int\}$ and $C : rec\ X\,.\,chan\{chan\{chan\{X, int\}, int\}, int\}$. The types for A, B and C are all the same, and are intuitively represented by the infinite tree shown in Figure 6.2, Section 6.1.2.

In order to compare recursive types for equality, we first convert them to a minimal, canonical form. This is achieved using an algorithm for minimisation of deterministic finite automata [121]. The algorithm requires a type to be represented as a directed graph; this is illustrated on the left of Figure 8.4 for the type $chan\{chan\{rec\ X\ .\ chan\{X, int\}, int\}, int\}$. The largest *bisimulation* on this graph is then computed. This relation partitions the graph nodes into equivalence classes. The type graph for the minimised type expression is the quotient graph with respect to the bisimulation equivalence relation, shown on the right of Figure 8.4.

As we have explained, recursive type minimisation is necessary for static channel diagram extraction. In addition, when a Promela specification is *not* well-typed, minimising recursive type expressions can make type error messages easier to understand.

8.3 Obtaining Static Channel Diagram Automorphisms from a Promela Specification

Given a Promela specification $\mathcal{P}$, the static channel diagram $\mathcal{SCD}(\mathcal{P})$ is defined analogously to the static channel diagram for a Promela-Lite specification (Definition 27, Section 7.1). $\mathcal{SCD}(\mathcal{P})$ can be extracted from $\mathcal{P}$ in linear time, as discussed in Section 7.1.1. The only complication is checking equality of recursive type expressions, as discussed in Section 8.2.2.

8.3.1 Computing $Aut(\mathcal{SCD}(\mathcal{P}))$

For illustration in Chapter 5 we used the GRAPE package to compute static channel diagram automorphisms. GRAPE interfaces with the *nauty* graph automorphism package [124].

For efficiency and ease of implementation, SymmExtractor uses *saucy* [37], a graph automorphism program based on *nauty*, to compute $Aut(\mathcal{SCD}(\mathcal{P}))$. We chose *saucy* over *nauty* as we found it easier to program with. Additionally, *saucy* has been shown to perform better than *nauty* when applied to large, sparse graphs [37]. We have found that static channel diagrams are typically sparse (though they may not be large). Our implementation uses a prototype extension of *saucy* which can handle directed graphs (personal communication, P. Darga and I. L. Markov, 2007).

8.3.2 Checking the validity of an element of $Aut(\mathcal{SCD}(\mathcal{P}))$

As discussed in Section 8.2.1, application of an element $\alpha \in Aut(\mathcal{SCD}(\mathcal{P}))$ to $\mathcal{P}$ requires information about the types of message arguments to channel variables which may not be directly available from the specification. This information is available after type reconstruction has been applied to $\mathcal{P}$.

The notion of validity for Promela specifications is slightly different to that for Promela-Lite specifications. Promela specifications $\mathcal{P}_1$ and $\mathcal{P}_2$ are equivalent if they are identical up to re-arrangement of operators to commutative operands, options in `do...od` statements, options in `if...fi` statements, and statements which appear after the `run` statements in the `init` { `atomic` { `...` } } block. Each of these statements assigns a distinct variable to a literal value (see Section 8.1.1), and they are enclosed in an `atomic` block, so their order does not matter. The intended use of these statements is for initialising *pid*-indexed arrays, such as the array of priority levels in the resource allocator example (see Section 4.4).

Once the specification $\alpha(\mathcal{P})$ has been obtained, checking whether $\mathcal{P} \equiv \alpha(\mathcal{P})$ involves an in-order traversal of the abstract syntax tree for each specification, sorting the operands to commutative operators and the options of `do...od` and `if...fi` statements, and sorting the initilisation statements described above. If $\mathcal{P} \equiv \alpha(\mathcal{P})$ then the specifications should be identical after this normalisation process has been applied.

8.3.3 Using GAP to compute the largest valid subgroup

In order to compute the largest valid subgroup of $Aut(\mathcal{SCD}(\mathcal{P}))$, SymmExtractor uses a GAP implementation of Algorithm 4 (Section 7.4). The Java and GAP components of SymmExtractor communicate using redirected standard input and output. Given a group G and a subgroup H of G, GAP provides a function to efficiently compute right coset representatives of H in G. The number of generators of $Aut(\mathcal{SCD}(\mathcal{P}))$ is typically small, and so initial generators for the valid group H are found quickly by checking each generator of $Aut(\mathcal{SCD}(\mathcal{P}))$ for validity against the specification $\mathcal{P}$.

As discussed in Section 7.4, Algorithm 4 performs badly if the initial group H is small, and $Aut(\mathcal{SCD}(\mathcal{P}))$ is very large. Our implementation includes a heuristic which can be applied to try to combat this problem. If the size of the initial valid subgroup H can be increased, fewer coset representatives need to be considered. An initial approach for increasing the size of H involved taking a set A of random elements of $Aut(\mathcal{SCD}(\mathcal{P})) \setminus H$ and checking the validity of each element of A against $\mathcal{P}$, adding the valid ones to the generators of H. However, when $Aut(\mathcal{SCD}(\mathcal{P}))$ is large, the probability of a random element being valid for $\mathcal{P}$ may be small. In this case a better approach is, for each $\beta \in A$ and each generator α of H, to check the validity of the element $\beta^{-1}\alpha\beta$ (the *conjugate* of α by β, see Definition 7symmetry:grouptheory:basics), adding each valid element $\beta^{-1}\alpha\beta$ to the generators of H (if it is not already contained in H). Adding random conjugates to the generators of H can work well in practice: discarding invalid generators of $Aut(\mathcal{SCD}(\mathcal{P}))$ often results in a group which can permute disjoint sets of processes and channels; adding random conjugates to this group can provide mappings between these disjoint sets.

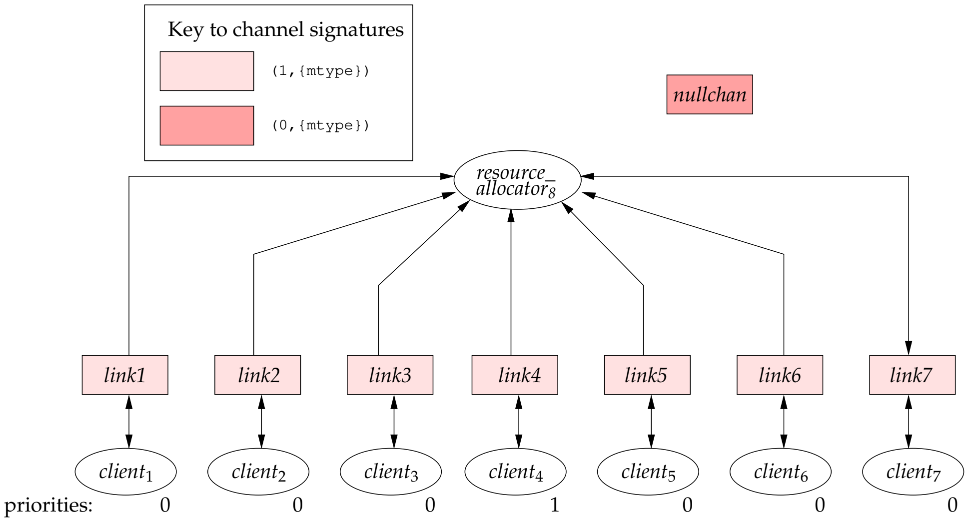


Figure 8.5: Static channel diagram for a prioritised resource allocator specification.

Recall the prioritised resource allocator specification of Section 4.4. Consider a version of the specification with seven *client* processes, where *client* 4 has priority level 1, and all other clients have priority level 0. The static channel diagram for such a specification is shown in Figure 8.5. Using *saucy* to compute a generating set for this group, and using $(i\ j)$ to denote the element $(i\ j)(linki\ linkj)$ we find that:

$$Aut(\mathcal{SCD}(\mathcal{P})) = \langle (1\ 2), (2\ 3), (3\ 4), (4\ 5), (5\ 6), (6\ 7) \rangle,$$

and $|Aut(\mathcal{SCD}(\mathcal{P}))| = 5040$. However, as *client* 4 is distinguished by its differing priority level, the generators $(3\ 4)$ and $(4\ 5)$ are not valid for $\mathcal{P}$. Removing these elements from the generating set has the effect of eliminating any permutations which map *client* processes in the set $\{1, 2, 3\}$ to the set $\{5, 6, 7\}$, and vice-versa. The result is a significantly smaller group H, with $|H| = 36$.

We can use GAP to pick three random elements of $Aut(\mathcal{SCD}(\mathcal{P}))$, say $\beta_1 = (1\ 6\ 5)(2\ 7\ 3)$, $\beta_2 = (1\ 3\ 7\ 4\ 2\ 5)$, $\beta_3 = (1\ 6\ 5\ 7\ 3\ 2\ 4)$. Figure 8.6 shows the elements obtained by conjugating each generator β of H by one of the β_i (i.e. computing $\beta_i^{-1}\alpha\beta_i$).

Observe that these random conjugates yield six elements of G which are valid for $\mathcal{P}$, but do not belong to H. Adding any one of these elements to the generators of H to give the group H' say, we find that $|H'| = 720$ whereas $|H| = 36$. Thus by considering 12 random conjugates, we have reduced the problem of checking $|Aut(\mathcal{SCD}(\mathcal{P}))|/|H| = 140$ coset representatives to checking $|Aut(\mathcal{SCD}(\mathcal{P}))|/|H'| = 7$.

We cannot say anything about how well this approach works in general, and it is likely to be problem-specific. However, it can be a useful optimisation when

Already in H	Not valid for $\mathcal{P}$	Valid for $\mathcal{P}$ and not in H
$(1\ 2)^{\beta_1} = (6\ 7)$	$(6\ 7)^{\beta_2} = (4\ 6)$	$(2\ 3)^{\beta_1} = (2\ 7)$
$(2\ 3)^{\beta_2} = (5\ 7)$	$(1\ 2)^{\beta_3} = (4\ 6)$	$(5\ 6)^{\beta_1} = (1\ 5)$
$(5\ 6)^{\beta_3} = (5\ 7)$	$(2\ 3)^{\beta_3} = (2\ 4)$	$(6\ 7)^{\beta_1} = (3\ 5)$
		$(1\ 2)^{\beta_2} = (3\ 5)$
		$(5\ 6)^{\beta_2} = (1\ 6)$
		$(6\ 7)^{\beta_3} = (3\ 5)$

Figure 8.6: Conjugation of the generators of H by elements $\beta_1 = (1\ 6\ 5)(2\ 7\ 3)$, $\beta_2 = (1\ 3\ 7\ 4\ 2\ 5)$ and , $\beta_3 = (1\ 6\ 5\ 7\ 3\ 2\ 4)$. For brevity, $(i\ \ j)$ is used to denote $(i\ \ j)(linki\ \ linkj)$.

$Aut(\mathcal{SCD}(\mathcal{P}))$ is large and H, the initial valid subgroup, is small but non-trivial.

The user can set a time-out period, after which the search for the largest valid subgroup will terminate, returning the valid subgroup computed so far. SymmExtractor provides feedback to the user by displaying the number of cosets which need to be checked, in the worst case. This feedback indicates whether it is worth waiting a little longer for a larger valid subgroup to be computed.

8.4 Experimental Results

We now present experimental results running SymmExtractor on a variety of Promela specifications, based on the examples described in Chapter 4. We divide the specifications into six *families*, and refer to an individual specification as a *configuration* of one of the families. For convenience, we introduce some shorthand notation for referring to configurations.

8.4.1 Specification families and configurations

The families of specifications we consider are: *simple mutex*, *Peterson*, *Peterson without atomicity*, *resource allocator*, *three-tiered architecture* and *hypercube*.

The *simple mutex*, *Peterson* and *Peterson without atomicity* families consist of versions of mutual exclusion protocols based on the examples presented in Sections 2.4.1, 4.3.2 and 4.3.4 respectively (with Promela examples given in Figure 2.6 and Appendices A.1.2 and A.1.4). A configuration of one of these families is identified via the number n of processes considered in the specification.

A configuration in the *resource allocator* family is a version of the resource allocator specification introduced in Section 4.4 and Appendix A.2.1. We consider two kinds of configuration. A configuration is identified by the signature a_0-a_1-$\ldots$-a_{k-1}, where $a_i > 0$ $(0 \leq i < k)$ if there are $k > 1$ distinct priority levels and *client* processes $1, 2, \ldots, a_0$ have priority level 0, $a_0 + 1, a_0 + 2, \ldots, a_1$ have priority level 1, etc. A configuration is referred to as *alternating* x, where $x > 0$ is even, if there are two priority levels, x *client* processes, and the priority levels alternates between 0 and 1 every three *client* processes. For example, *alternating 10* denotes a 10-*client* configuration where *client* processes 1, 2, 5, 6, 9 and 10 have priority level 0 and *client* processes 3, 4, 7 and 8 have priority level 1. The resource allocator specifications reveal an interesting problem. If an array of size $n+1$ is indexed using only literal values in the range 0–n then it is not possible to determine whether the index type of the array should be *pid* or *byte*. This is true of the `priorities` array in the resource allocator specification (see Appendix A.2.1). By default, SymmExtractor conservatively assigns the index type for such an array to be *byte*. This causes symmetry detection to return a trivial group for the resource allocator examples. We overcome this problem by incorporating an assertion of the form `assert(priorities[_pid]<`n`)` (where n is the number of *client* processes) in one of the `atomic` blocks in the body of the *client* proctype. Since `_pid` has type *pid*, this makes the index type of the array unambiguous.

A configuration in the *three-tiered architecture* family is a version of the three-tiered architecture specification introduced in Section 4.5 and Appendix A.3. A configuration with k *server* processes ($k > 0$) and $a_i > 0$ *client* processes connected to server i ($1 \leq i \leq k$) is identified via the signature a_1-a_2-...-a_k. For example, 5-5-5-5 denotes a configuration with 5 *servers* and 25 *clients*, 5 connected to each server.

Recall that the hypercube specification of Section 4.6 and Appendix A.4.1 involves arithmetic operations on variables which have *pid* type. This was discussed in Section 4.6.1. However, we used SPIN-to-GRAPE to check that the symmetry group associated with the hypercube specification is isomorphic to the group of automorphisms of a 3-dimensional cube, which is in turn isomorphic to the group of automorphisms of the channel diagram associated with the specification (see Section 5.2.2). SymmExtractor is based on the type system of Figure 6.5, Section 6.2, which does *not* allow *pid* variables to be operands in arithmetic expressions. In order to apply SymmExtractor to examples based on the hypercube specification, we have used techniques similar to those described in Section 7.6.2 to re-model the specification without these arithmetic expressions, in a semantics-preserving way. The re-modelled version of the 3-dimensional hypercube specification is given in Appendix A.4.2. As discussed in Section 7.6.2, this specification is much longer than the original. A configuration of the *hypercube* family is a version of the modified hypercube specification. A configuration is identified via the number n of dimensions of the hypercube. Note that configuration n is comprised of 2^n *node*.

8.4.2 Results and discussion

For various configurations of the families described in Section 8.4.1, Figure 8.7 reports the following figures:

- $|Aut(\mathcal{SCD}(\mathcal{P}))|$ – size of the automorphism group of the static channel diagram associated with configuration $\mathcal{P}$

Configuration $\mathcal{P}$	$\lvert Aut(\mathcal{SCD}(\mathcal{P}))\rvert$	$\lvert H\rvert$	$\lvert G\rvert$	*saucy* time	*find largest* time
simple mutex					
5	120	=	=	0.03	0.07
10	3.6×10^6	=	=	0.03	0.20
20	2.4×10^{18}	=	=	0.03	0.59
40	8.1×10^{37}	=	=	0.03	1.64
Peterson					
3	6	=	=	0.03	0.06
6	720	=	=	0.02	0.16
9	362880	=	=	0.04	0.30
12	4.8×10^8	=	=	0.03	0.56
Peterson without atomicity					
3	6	=	=	0.03	0.08
6	720	=	=	0.08	0.25
9	362880	=	=	0.03	0.52
12	4.8×10^8	=	=	0.03	0.89
resource allocator					
3/4	5040	144	144	0.03	1.52
2/2/3	5040	24	24	0.03	5.24
5/5	3.6×10^6	14400	14400	0.05	8.34
3/3/4	3.6×10^6	864	864	0.03	114.49
alternating 10	3.6×10^6	32	17280	0.03	18.12
alternating 12	4.7×10^8	64	518400	0.04	314.87
alternating 14	8.7×10^{10}	128	2.9×10^6	0.06	$>$ 12 hours
alternating 16	2.1×10^{13}	256	1.6×10^9	0.05	$>$ 12 hours
three-tiered architecture					
3/3/2	144	=	=	0.04	0.09
3/3/3	1296	=	=	0.05	0.14
4/4/3	6912	=	=	0.05	0.17
4/4/4	82944	=	=	0.05	0.26
5/5/5/5	5.0×10^9	=	=	0.07	0.55
hypercube					
2d	8	2	4	0.04	0.59
3d	48	2	8	0.03	2.50
4d	384	2	16	0.04	99.11
5d	3840	2	32	0.08	7171.57

Figure 8.7: Experimental results for automatic symmetry detection.

	2 conjugates		5 conjugates		10 conjugates	
Configuration	$\|H\|$	time	$\|H\|$	time	$\|H\|$	time
resource allocator						
alternating 10	192	9.86	2880	9.32	17280	8.50
alternating 12	34560	39.14	518400	32.66	518400	33.59
alternating 14	414720	172.3	2.9×10^6	115.45	2.9×10^6	116.39
alternating 16	1.6×10^9	539.65	1.6×10^9	541.86	1.6×10^9	543.87
hypercube						
2d	2	0.65	2	0.79	4	0.84
3d	4	1.85	8	1.60	8	1.98
4d	4	71.26	16	23.10	16	26.73
5d	8	3409.03	16	1786.33	32	957.79

Figure 8.8: Optimised symmetry detection using random conjugates.

- $|H|$ – size of the initial subgroup generated by the valid generators of $Aut(\mathcal{SCD}(\mathcal{P}))$
- $|G|$ – size of the largest valid subgroup of $Aut(\mathcal{SCD}(\mathcal{P}))$, computed using Algorithm 4 (Section 7.4)
- *saucy* time – time (in seconds) taken by *saucy* to compute generators for $Aut(\mathcal{SCD}(\mathcal{P}))$
- *find largest* time – time (in seconds) taken to compute $|G|$ given generators for $Aut(\mathcal{SCD}(\mathcal{P}))$.

When all generators of $Aut(\mathcal{SCD}(\mathcal{P}))$ are valid, $Aut(\mathcal{SCD}(\mathcal{P}))$, H and G are equal, so there is no need to use Algorithm 4. This is indicated by '$=$' in Figure 8.7. When $|G|$ could not be computed within 12 hours, the entry '$>$ 12 hours' appears in the table. In these cases, the configuration is given in italics, as the group G has been successfully computed by other means, which we discuss below. All experiments were performed on a PC with a 2.4 Intel Xeon processor and 3Gb of main memory.[2]

The '*saucy* time' column shows that, for all the configurations we tried, there is a minimal overhead associated with using *saucy* to compute $Aut(\mathcal{SCD}(\mathcal{P}))$, regardless of how large this group is.

Configurations from the three mutual exclusion families, as well as *three-tiered architecture* configurations, show that automatic symmetry detection is very efficient when all generators of $Aut(\mathcal{SCD}(\mathcal{P}))$ are valid. In this case the '*find largest* time' column reports the time taken to check validity of these generators against the input specification $\mathcal{P}$. The results for the *simple mutex* configuration with 40 processes, and configuration 5-5-5-5 in the *three-tiered architecture* family (which involves 25 processes) show that SymmExtractor is robust enough to handle large Promela specifications.

Results for the first four *resource allocator* configurations shown in Figure 8.7 illustrate cases where the initial valid subgroup H turns out to be the largest valid

2. All of the Promela specifications used for these experiments are available online in archived form (see Section 1.2).

subgroup of $Aut(\mathcal{SCD}(\mathcal{P}))$. In these cases, since $H \neq Aut(\mathcal{SCD}(\mathcal{P}))$, it is necessary to run Algorithm 4 to confirm that H is indeed the largest valid subgroup. This is time-consuming for the 3-3-4 configuration.

The *alternating resource allocator* and *hypercube* specifications illustrate a strict containment relationship: $\{id\} \subset H \subset G \subset Aut(\mathcal{SCD}(\mathcal{P}))$. Although the number of cosets of H in $Aut(\mathcal{SCD}(\mathcal{P}))$ may be large, if G is significantly larger than H then the number of coset representatives which need to be checked for validity diminishes rapidly as valid representatives are found. However, for the *alternating 14* and *alternating 16* configurations the number of cosets of H in $Aut(\mathcal{SCD}(\mathcal{P}))$ is so large that G could not be computed within 12 hours. Note that the size of G in these larger examples has been computed with the aid of random conjugates, as discussed below.

In Section 4.6.1 we computed the automorphism group of the Kripke structure associated with the 3-dimensional *hypercube* specification. (The results of Section 4.6.1 are the same if we re-run SPIN-to-GRAPE using the modified specification considered here.) We used GAP to show that this group is isomorphic to the automorphism group of a cube – the group $K_3 \rtimes S_3$, which has order 48. If $\mathcal{P}$ is the 3-dimensional *hypercube* specification, it is not surprising that $Aut(\mathcal{SCD}(\mathcal{P})) \cong K_3 \rtimes S_3$ (given the similar result for $Aut(\mathcal{CD}(\mathcal{P}))$ described in Section 5.2.2), and the results of Figure 8.7 confirm that $|Aut(\mathcal{SCD}(\mathcal{P}))| = 48$ also. It is surprising, therefore, that G, the largest valid subgroup of $Aut(\mathcal{SCD}(\mathcal{P}))$, has order 8. SymmExtractor describes $Aut(\mathcal{SCD}(\mathcal{P}))$ in terms of three generators as follows:

$$\begin{aligned} Aut(\mathcal{SCD}(\mathcal{P})) \quad = \quad & \langle (2\ 3)(link2\ link3)(6\ 7)(link6\ link7), \\ & (3\ 5)(link3\ link5)(4\ 6)(link4\ link6) \\ & (1\ 2)(linky1\ link2)(3\ 4)(link3\ link4)(7\ 8)(5\ 6)(link5\ link6) \\ & \quad (link7\ link8)\rangle. \end{aligned}$$

The first two of these generators are invalid. To see why this is the case, for the generator $\alpha = (2\ 3)(link2\ link3)(6\ 7)(link6\ link7)$, consider the first `run` statement of $\mathcal{P}$:

```
run node(link1,link2,link3,link5);
```

By the definition of $\alpha(\mathcal{P})$ in Section 7.2.1, the first `run` statement of $\alpha(\mathcal{P})$ is:

```
run node(link1,link3,link2,link5);
```

The fact that these `run` statements are not identical implies that $\mathcal{P} \not\equiv \alpha(\mathcal{P})$. Similarly, the second generator above is shown to be invalid for $\mathcal{P}$. The largest valid subgroup G can be shown to be isomorphic to the subgroup K_3 of $K_3 \rtimes S_3$. For each of the *hypercube* configurations we have analysed we see a similar result: $|Aut(\mathcal{SCD}(\mathcal{P}))| = |K_n \rtimes S_n| = 2^n \times n!$, $|H| = 2$, and $|G| = |K_n| = 2^n$ (where n is the dimension of the hypercube). SymmExtractor does not detect a symmetry group isomorphic to $K_n \rtimes S_n$ (which, for the 3-dimensional case, we have identified as a symmetry group for the associated model) due to the definition of *validity* given in Section 7.3.1. It may be possible to relax this notion (as discussed in Section 7.6.4) to automatically detect larger symmetry groups for the *hypercube* specifications.

While the elimination of arithmetic expressions from the specification is unrelated to this validity issue, re-modelling the specification to avoid arithmetic on *pid* variables results in a much larger specification; this problem was noted in Section 7.6.2. The complexity of checking whether $\alpha \in Aut(\mathcal{SCD}(\mathcal{P}))$ is valid for $\mathcal{P}$ is proportional to the size of $\mathcal{P}$, and is therefore time-consuming for the 4- and 5-dimensional *hypercube* specifications. This explains the lengthy computation of G for the 5-dimensional configuration, despite the fact that the ratio $|Aut(\mathcal{SCD}(\mathcal{P}))|/|H|$ is not large (compared to that for e.g. the *alternating 12* resource allocator configuration, for which the time to compute G is much less).

Results using the random conjugates optimisation

The *alternating* resource allocator configurations and the *hypercube* configurations are examples where the initial valid subgroup H is non-trivial, but the search for G involves checking a significant number of coset representatives. To alleviate this problem we tried increasing the size of H using random conjugates as described in Section 8.3.3.

For each of the relevant specifications, Figure 8.8 shows the size of H, enlarged using two, five and ten random conjugates, and the resulting time to find G. The sizes of $Aut(\mathcal{SCD}(\mathcal{P}))$ and G are as in Figure 8.7. The results show that this optimisation can be useful in practice: symmetry detection using conjugates is faster in all cases. The speed-up is particularly noticeable for the *alternating* 12 *resource allocator* and 5-dimensional *hypercube* configurations, and it was possible to detect symmetry for the *alternating* 14 and 16 *resource allocator* configurations, which were previously intractable.

Applying SymmExtractor to modified versions of the specifications

In Sections 4.4.3, 4.5.2 and 4.6.2 we considered modifications to the resource allocator, three-tiered architecture and hypercube specifications respectively, and used SPIN-to-GRAPE to see the effect of these modifications on the automorphism groups associated with these specifications.

Figure 8.9 shows the results of applying SymmExtractor to these modified specifications. Symmetry detection is effective for the *three-tiered architecture* specification with mixed modes of communication, and the *resource allocator* specification which features sharing between *client* processes, and the symmetry groups obtained in each case conform to the groups computed using SPIN-to-GRAPE in Sections 4.4.3 and 4.5.2 respectively.

Configuration $\mathcal{P}$	$\lvert Aut(\mathcal{SCD}(\mathcal{P}))\rvert$	$\lvert H\rvert$	$\lvert G\rvert$	*saucy* time	*find largest* time
resource allocator sharing	864	4	12	0.03	4.47
three-tiered mixed	72	=	=	0.05	0.08
hypercube fixed	6	1	1	0.03	1.08

Figure 8.9: Applying SymmExtractor to the modified *resource allocator*, *three-tiered architecture* and *hypercube* specifications.

For the hypercube specification with a fixed initiator SymmExtractor does not detect any non-trivial symmetry, and must enumerate $Aut(\mathcal{SCD}(\mathcal{P}))$ to determine this. Recall from Section 4.6.2 that the symmetry group associated with this modified specification is isomorphic to $stab_{K_3 \rtimes S_3}(\mathbf{0})$. Since the symmetry group which SymmExtractor computes for the original specification is isomorphic to K_3 rather than $K_3 \rtimes S_3$, it not surprising that the group computed for the modified specification corresponds $stab_{K_3}(\mathbf{0}) = \{id\}$.

8.5 Using Two Student Assessed Exercises to Evaluate SymmExtractor

The SymmExtractor tool can handle more general kinds of symmetry than SymmSpin or SMC, places fewer restrictions on the form of specifications, and does not require annotation of a specification with additional data types.[3] However, SymmExtractor still places some restrictions on the form of specifications, as summarised in Section 8.1.1. Due to the fundamental difficulty of automatic symmetry detection (see Section 3.3.5), these restrictions are not unreasonable. Nevertheless, it is important to assess the impact of the restrictions on the practical use of SymmExtractor.

We present an evaluation of SymmExtractor based on a set of example solutions to two assessed exercises from the *Modelling Reactive Systems* final year course at the University of Glasgow. We discuss the ethical issues involved in using student programs for research, present the design of our evaluation, and propose some changes to SymmExtractor, and directions for future work, based on the evaluation results. As well as providing insight into the challenges of automatic symmetry detection, the chapter is a novel case study in formal methods evaluation.

8.5.1 The Modelling Reactive Systems course

Modelling Reactive Systems (MRS) is a final year 20-lecture formal methods course at the University of Glasgow. The primary focus of the course is on the theory and practice of model checking, and students use SPIN in practical sessions. The main prerequisite for MRS is a discrete mathematics course for computing science,

3. Arguably, making a distinction between the *pid* and *byte* data types, which SPIN regards as interchangeable, means that the use of the *pid* type is a form of symmetry-related annotation. However, this primitive type is already part of the specification language, and is simple to use.

which covers the basics of set theory, predicate logic, relational algebra and methods of proof. In addition, students are required to have passed first year mathematics courses on calculus and algebra, as well as multiple computing science courses on programming, data structures and algorithms. Almost 20% of the assessment for MRS is via a practical exercise which involves specifying a reactive system using Promela, then reasoning about the specification with SPIN. We now describe the practical assignments which were set for sessions 2004/2005 and 2005/2006.

Telephone Exchange

The MRS practical exercise for 2004/2005 involved producing three versions of a specification for a two user telephone system. Version one was a naive model, version two a model in which acknowledgment were included, and version three a model in which call clear-down was made to be asymmetric (only the initiator of a call could terminate the call). Intuitively, a Promela specification of a two-user telephone exchange should exhibit one non-trivial symmetry which switches the local states of the users (and their associated channels) throughout all global states. Thus solutions to this modelling task provide a good set of Promela examples with which to evaluate the restrictions imposed by SymmExtractor. Furthermore, the associated state-spaces are small enough for SPIN-to-GRAPE (see Section 4.1) to compute all state-space symmetries present in a given specification, which can be compared with those detected by SymmExtractor.

Railway Signalling System

The practical exercise for 2005/2006 involved designing a specification of a railway system consisting of two *train* processes, eight *gate* processes and a *controller* process. The trains were each to travel around one of two circular tracks which intersected along a section, as illustrated in Figure 8.10. The trains were to communicate with the *controller* process to indicate their approach to and departure from the gates, and the *controller* process in turn was to communicate with the gates to instruct them to raise and lower, as appropriate. The communication protocol was to be designed in such a way as to avoid the two trains having access to the section of shared track simultaneously. The diagonal grey line of Figure 8.10 illustrates symmetry in the structure of the system. We would expect a model of such a system to exhibit one non-trivial automorphism corresponding to simultaneously swapping the local states of gates 0 and 4, 1 and 5, 2 and 6, 3 and 7, and trains 1 and 2.

8.5.2 Ethical approval

Before we describe how we have used solutions to the practical exercises for MRS to evaluate SymmExtractor, we outline the ethics procedure we have followed.

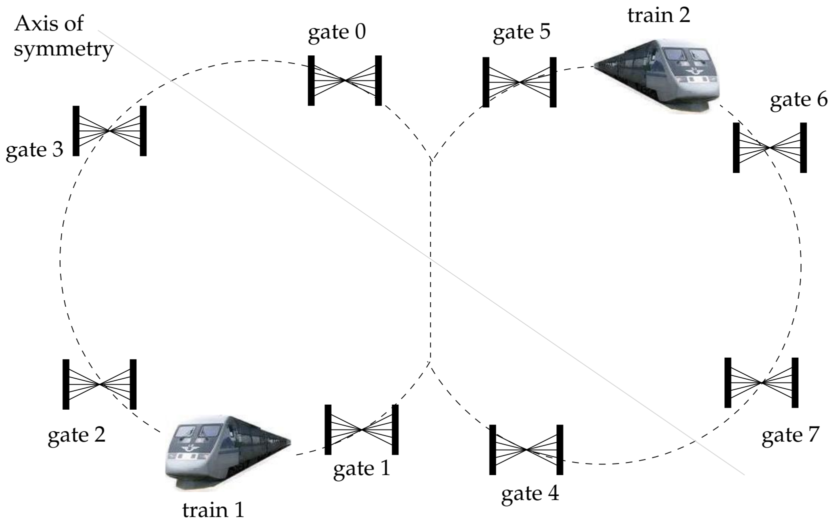


Figure 8.10: Layout of railway signalling system.

To ensure that our user study is ethical, we have followed the *Glasgow Ethics Code* check-list [142]. This is a 12-point check-list distilled from the ethical standard of the British Psychological Society [16], and focuses on the issues which are most relevant to computing science projects. Compliance with most of the points on the check-list was straightforward. The following points required some care:

- **All participants explicitly stated that they agreed to take part.** Students who allowed us to use their solutions in the study were provided with an information sheet detailing the aims of the study, and asked to sign a consent form. The information sheet and consent form given to students in session 2005/2006 are included as Appendix D, and are adapted from a standard example [141]. The intended usage of students' solutions is detailed in Section 8.5.3.
- **The researcher conducting the experiment is not in a position of authority or influence over any of the participants.** As the solutions formed part of the course assessment, it was important that the consent of students was not sought until after solutions had been assessed and returned. This assured students that their decision to take part in the study could have no effect on their score for the exercise, and encouraged them to answer the assessed questions in exactly the same way as they would have otherwise.

A further ethical concern is that the assessed exercises should be designed to meet the intended learning outcomes of the course and not to meet research aims (unless these overlap). In addition, since assessment has been shown to narrow students' focus [15], care must be taken to ensure that an assessment biased towards the research interests of the course director does not restrict breadth of learning. In our case the exercises had been set to meet the course aims before we designed our evaluation.

The study was approved by the ethics committee of the Faculty of Information and Mathematical Sciences at the University of Glasgow (ref. *FIMS00203*). We obtained signed consent forms from 17 students from session 2004/2005, and 12 students from session 2005/2006. The average class size for these years was 35.

8.5.3 Methods

For each specification in the sample set we gathered the following data by a combination of automatic and manual analysis:

1. Size of the unreduced state-space (computed using SPIN)
2. State-space symmetries computed by SPIN-to-GRAPE, and size of the resulting quotient state-space (if feasible)
3. Symmetry breaking features of the specification, and modifications required to restore symmetry (documented by experimenter)
4. Violations of restrictions imposed by SymmExtractor (as reported by the tool) and modifications required to satisfy restrictions (documented by experimenter)
5. Symmetries detected by SymmExtractor.

We also used our symmetry reduction package TopSPIN, which is described in Chapter 11, to check that our symmetry reduction results agree with the quotient state-spaces produced independently by SPIN-to-GRAPE.

Symmetry breaking features are aspects of the specification which destroy the intuitive symmetry discussed in Section 8.5.1. When SPIN-to-GRAPE showed absence of this expected symmetry in a given specification, the experimenter manually examined the specification to identify symmetry breaking features. In the cases where it was not feasible to use SPIN-to-GRAPE for state-space analysis, the experimenter looked for certain commonly occurring symmetry breaking features.

We classify the modifications of 4 above as *minor* if they could be avoided by a straightforward extension of SymmExtractor, *medium* if they would be unnecessary if SymmExtractor could capture symmetry between global variables (as discussed in Section 7.6.3), or *major* if they could only be avoided by significant development of the theory of Chapter 7 on which SymmExtractor is based.

8.5.4 Results

Telephone exchange

We refer to the individual components of a three part solution as specifications. Of the 51 specifications analysed, just over half did not exhibit the expected symmetry due to symmetry breaking features. In most cases this was because `run` statements were not surrounded by an `atomic` block; for other examples the telephone users were initialised asymmetrically (e.g. *handset* variables for users 1 and 2 were set to *up* and *down* respectively, destroying symmetry between users). In all cases it was

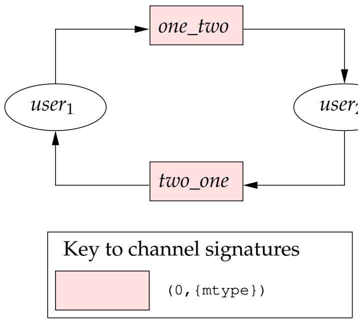


Figure 8.11: Typical static channel diagram for the telephone examples.

possible to restore symmetry by trivial modifications, with a negligible effect on the global state-space. With these modifications, SymmExtractor was able to detect symmetry immediately from 23 of the resulting specifications. A further 13 required modifications which we classified as *minor*; these included replacing locally instantiated channels with globally instantiated channels, and removing channel instantiation statements from record declarations. Another seven specifications required *medium* modifications (as described above). The final eight specifications required *major* modifications. These modifications have identified a problem with the usability of SymmExtractor, which involves the way that arrays indexed by process identifiers are accessed. This is discussed further in Section 8.5.5. An example of a typical specification which required major modifications, together with our re-modelled version, can be found in Appendix A.5.

After necessary modifications, SymmExtractor was able to efficiently detect symmetries for all specifications. Figure 8.11 shows the typical static channel diagram structure associated with the example solutions.

Railway signalling system

The 12 Promela solutions to the railway signalling exercise all resulted in large state-spaces. Thus for these examples it was not possible to use SPIN-to-GRAPE to compute symmetries of the global state-space. However, in six of the solutions the experimenter was able to identify the common symmetry breaking feature of `run` statements occurring outwith an `atomic` block.

SymmExtractor proved to be ineffective in detecting symmetry for this set of examples (after fixing symmetry breaking features): in all cases, specifications required *major* modifications. We illustrate the kind of re-modelling required for SymmExtractor to be applicable to these solutions using an example.

Figure 8.12 shows part of Promela specification which is typical of the set of solutions for this exercise. Specifically, the figure includes a *train* proctype and the `init` process, but omits proctypes for *gate*, `shared_gate` and *controller* processes. Figure 8.13 shows the portion of the example after re-modelling. Full versions of

```
mtype = {approaches, leaves, lower, raise, atgate, faraway, up, down};

chan control_link = [0] of {mtype, byte};
chan gate_link [8] = [0] of {mtype};

mtype bar[8] = down;
bool on_shared_track[2] = false;
bool shared_track_open = false

proctype train(byte current_gate, id) {
  mtype position = atgate;
  control_link!approaches,current_gate;
  do :: atomic { position==faraway ->
          if :: current_gate==3 -> current_gate = 0; assert(id == 0)    (*)
             :: current_gate==7 -> current_gate = 4; assert(id == 1)
             :: else -> current_gate++;
          fi;
          control_link!approaches,current_gate; position=atgate}
     :: atomic { (bar[current_gate]==up && position==atgate)->
          if :: (current_gate == (id * 4)) -> on_shared_track[id] = true
             :: else -> skip
          fi;
          position = faraway; control_link!leaves,current_gate;
          if :: (current_gate == (id * 4 + 1)) ->                        (**)
                on_shared_track[id] = false
             :: else -> skip
          fi
        }
  od
}

init {
  atomic {
    run controller(); run shared_gate(0); run gate(1); run gate(2);
    run gate(3); run shared_gate(4); run gate(5); run gate(6);
    run gate(7); run train(2, 0); run train(6, 1);
  }
}
```

Figure 8.12: Typical example of a solution to the railway signalling problem.

both Promela specifications are given in Appendix A.6.

In the original specification, a *train* process is instantiated with an id which is either 0 or 1. Similarly, the eight *gate* processes are instantiated with an id in the range 0–7. A *train* is also instantiated with the identifier of the gate at which it starts. Recall from Section 8.1.1 that SymmExtractor requires processes to use their built-in `_pid` variable, rather than being passed an id as a parameter, and that processes instantiated by the `init` process are assigned identifiers in order, starting from 1. Comparing lines marked `(*)` in the original and modified specifications illustrates this. Instead of referring to gates 3 and 0, in the re-modelled specification we refer to gates 5 and 2. This is because the `_pid` values for *gate* and `shared_gate` processes are in the range 2–9. Similarly, instead of asserting that `id==0`, we now assert that `_pid==10`, since the `_pid` value for the first train process in the re-modelled specification is 10.

The major disadvantage of this modification is that *train* processes must now index into the `on_shared_track` array using their `_pid` variable, which is in the range 10–11, and thus the array must be declared with size 12. The first ten positions of this array are wasted, but form part of the state-vector.

Another significant re-modelling step is illustrated by the lines marked `(**)`. Recall from Section 8.1.1 that SymmExtractor does not allow variables of type *pid* to be used in arithmetic operations. Thus the expression `current_gate ==(id*4+1)` must be re-modelled. In the original specification this expression can be re-written as a disjunction: `(id==0&&current_gate==1)||(id==1&& current_gate==5)`, since id is either 0 or 1. In the re-modelled specification this translates to:

```
(_pid==10&&current_gate==3)||(_pid==11&&current_gate==7).
```

In the re-modelled specification, to avoid the use of a global array of channels (another restriction of SymmExtractor), *gate* and `shared_gate` proctypes are parameterised by a channel. For details of this re-modelling, see the online specifications.

After this heavy-weight re-modelling, SymmExtractor can be used to find valid static channel diagram automorphisms for the specification of Figure 8.13. However, the process of symmetry detection is slow for this example. The static channel diagram for the specification is shown in Figure 8.14. SymmExtractor computes generators for $Aut(\mathcal{SCD}(\mathcal{P}))$ as follows: $Aut(\mathcal{SCD}(\mathcal{P})) =$

$$\begin{aligned}\langle\ \ & (3\ 4)(gate\_link\_3\ gate\_link\_4), (4\ 5)(gate\_link\_4\ gate\_link\_5),\\ & (5\ 7)(gate\_link\_5\ gate\_link\_7), (7\ 8)(gate\_link\_7\ gate\_link\_8),\\ & (8\ 9)(gate\_link\_8\ gate\_link\_9), (2\ 6)(gate\_link\_2\ gate\_link\_6),\\ & (10\ 11)\ \ \rangle.\end{aligned}$$

Here $|Aut(\mathcal{SCD}(\mathcal{P}))| = 2880$. However, the largest possible subgroup computed by SymmExtractor has order 2, and consists of the identity and the element:

$$\begin{aligned}&(2\ 6)(gate\_link\_2\ gate\_link\_6)(3\ 7)(gate\_link\_3\ gate\_link\_7)\\ &(4\ 8)(gate\_link\_4\ gate\_link\_8)(5\ 9)(gate\_link\_5\ gate\_link\_9)(10\ 11).\end{aligned}$$

Thus SymmExtractor must search the group $Aut(\mathcal{SCD}(\mathcal{P}))$ to find this single non-trivial valid symmetry. Clearly if there were more processes in the specification $Aut(\mathcal{SCD}(\mathcal{P}))$ would be would be larger, and this search might not be feasible. For this example, the random conjugates optimisation presented in Section 8.3.3 does not help, as the initial valid subgroup is trivial.

8.5.5 Outcomes of the evaluation

The evaluation has led to some straightforward changes to the documentation and design of SymmExtractor, as well as some interesting open research questions. We

```
mtype = {approaches, leaves, lower, raise, atgate, faraway, up, down};

chan control_link = [0] of {mtype, pid};
chan gate_link_2 = [0] of {mtype}; ...; chan gate_link_9 = [0] of {mtype};

mtype bar[12] = down;
bool on_shared_track[12] = false;
bool shared_track_open = false

proctype train(pid current_gate) {
  mtype position = atgate;
  control_link!approaches,current_gate;
  do :: atomic { position==faraway ->
          if :: current_gate==2-> current_gate=3
             :: current_gate==3-> current_gate=4
             :: current_gate==4-> current_gate=5
             :: current_gate==5 -> current_gate = 2; assert(_pid == 10)   (*)
             :: current_gate==6 -> current_gate=7
             :: current_gate==7 -> current_gate=8
             :: current_gate==8 -> current_gate=9
             :: current_gate==9 -> current_gate=6; assert(_pid == 11)
          fi;
          control_link!approaches,current_gate; position=atgate}
     :: atomic { (bar[current_gate]==up && position==atgate) ->
          if :: ((_pid==10 && current_gate == 2)||
                 (_pid==11 && current_gate == 6)) ->                     (**)
                on_shared_track[_pid] = true
             :: else -> skip
          fi;
          position = faraway; control_link!leaves,current_gate;
          if :: ((_pid==10 && current_gate == 3)||
                 (_pid==11 && current_gate==7)) ->
                on_shared_track[_pid] = false
             :: else -> skip
          fi
        }
  od
}

init {
  atomic {
    run controller(); run shared_gate(gate_link_2);
    run gate(gate_link_3); run gate(gate_link_4);
    run gate(gate_link_5); run shared_gate(gate_link_6);
    run gate(gate_link_7); run gate(gate_link_8);
    run gate(gate_link_9); run train(4); run train(8);
  }
}
```

Figure 8.13: Re-modelled version of the railway signalling specification.

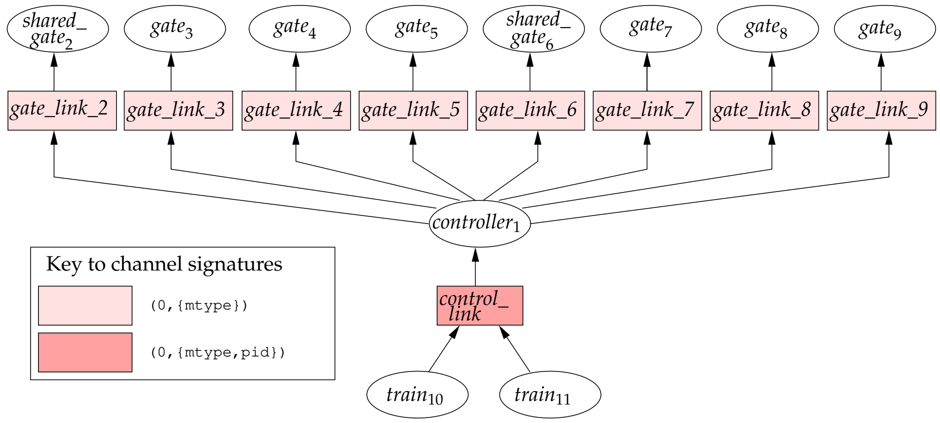


Figure 8.14: Static channel diagram for the modified railway signalling specification.

have extended the SymmExtractor documentation with a short set of modelling guidelines based on the problems encountered when applying the tool to this set of examples (see Appendix C.2.3). It is clear that the tool would be more useful if it could handle symmetry between global variables, as discussed in Section 7.6.3.

The line marked `(**)` in Figure 8.13 illustrates a case where using a process identifier in an arithmetic operation *does not* destroy symmetry in the underlying model. Thus an open research question is: under what conditions is it possible to relax this restriction of SymmExtractor? We outlined a possible solution to this problem in Section 7.6.2.

As discussed in Section 8.5.4, for the re-modelled railway signalling example, $Aut(\mathcal{SCD}(\mathcal{P}))$ is much larger than the valid subgroup which SymmExtractor eventually computed. Thus for this example, the static channel diagram does not reflect the communication structure of the system very well. As discussed in Section 7.5, it would be interesting to investigate other graphical representations derived from Promela specifications which may better reflect the symmetry present in examples such as this one.

A major challenge which the evaluation results have presented is to find techniques to automatically determine the relationship between numeric identifiers passed as parameters to processes by the user (and used to access arrays), and the run-time id values which SPIN assigns to processes. This was a problem associated with some of the telephone specifications, and with all of the railway signalling solutions.

The problem is that the built-in `_pid` variable indicates the instantiation number of a process with respect to *all* processes, not with respect to a given proctype. Therefore the first *train* process in Figure 8.13 has `_pid` value 10, even though it is the first *train* process to be instantiated. This means that if the `_pid` variable is used as an index for an array of values relevant to processes of a given proctype, the array must be as large as the highest id of any instantiation of this proctype, and this may significantly increase the space requirement for each state of the model. An elegant solution to this problem would greatly improve the usability of SymmExtractor.

Summary

We have described SymmExtractor, an automatic symmetry detection tool for Promela based on the techniques presented in Chapter 7. In particular, we have discussed the way in which SymmExtractor uses type reconstruction and bisimulation minimisation to handle incomplete channel types and recursive types respectively. We have described the way in which the *saucy* and GAP tools are used to find the largest valid group of static channel diagram automorphisms for a Promela specification.

Experimental results show that the overhead of applying SymmExtractor to detect Kripke structure automorphisms before search is minimal, except when $Aut(\mathcal{SCD}(\mathcal{P}))$ is large and the largest valid subgroup of $Aut(\mathcal{SCD}(\mathcal{P}))$ is small. We have suggested an optimisation based on random conjugates to help overcome this problem, and shown that this optimisation can help with symmetry detection for practical examples.

We have presented the methods and results of a user study to asses the feasibility of the restrictions on the form of a Promela specification which SymmExtractor imposes. This evaluation has identified some cases where SymmExtractor is over-restrictive and cannot detect symmetries which SPIN-to-GRAPE has shown to exist. The evaluation results have identified some challenging open research problems in the area of automatic symmetry detection.

Chapter 9

Exact and Approximate Strategies for Symmetry Reduction

In Chapters 7 and 8 we have been concerned with the problem of *detecting* automorphisms of the Kripke structure associated with a Promela specification. We now turn our attention to the problem of efficiently exploiting symmetry in order to verify large systems using model checking.

As discussed in Section 3.4, given a symmetry group G, a common approach to ensuring that equivalent states are recognised during search is to convert each newly encountered state s into $min_{\preceq}[s]_G$, the *smallest* state in its orbit (under a suitable total ordering $\preceq$), before it is stored. However, the problem of computing $min_{\leq}[s]_G$ for an arbitrary group (where $\leq$ is the lexicographic ordering on vectors), known as the *constructive orbit problem* (COP), is NP-hard [27] (see Definition 24 and Theorem 10, Section 3.4).

Existing symmetry reduction packages, such as SymmSpin [14] and SMC [163], are limited as they can only exploit full symmetry between identical components of a system. This eases the problem of identification of symmetry, and the COP can be solved efficiently for this special case. However, as illustrated in Chapter 4, the automorphism group associated with a Kripke structure may be more complex. Since the automatic symmetry detection techniques of Chapters 7 and 8 can detect *arbitrary* kinds of symmetry arising from the static channel diagram of a specification, it is important to have techniques to solve the COP efficiently using information about the structure of G, or to provide an efficient, *approximate* solution to the COP (see Sections 3.4.1 and 3.4.2 respectively) when no such information is available.

In this chapter we generalise existing techniques for efficiently exploiting symmetry under a simple model of computation, and give an approximate strategy for use with symmetry groups for which fast, exact strategies cannot be found. We use computational group theory to automatically determine the structure of a group as a disjoint/wreath product of subgroups before search so that an appropriate symmetry reduction strategy can be chosen.

9.1 A Model of Computation Without References

We refer to a process, global variable or buffered channel in a concurrent system as a *component*. We now justify the claim made in Section 3.4 that we can reason about states of a concurrent system using a single integer variable for each component.

Let $V = \{v_1, v_2, \ldots, v_l\}$ be the set of variables associated with a concurrent system, and let D_i be the finite domain of v_i ($1 \leq i \leq l$). Let $m = max\{|D_i| : 1 \leq i \leq l\}$. Then we can enlarge each D_i so that $|D_i| = m$, and assume without loss of generality that $D_i = \{1, 2, \ldots, m\}$ ($1 \leq i \leq l$). Let $\mathcal{M} = (S, s_0, R)$ be a Kripke structure defined in terms of $D = D_1 \times D_2 \times \cdots \times D_l$ (see Definition 1, Section 2.2).

Suppose V is partitioned into n subsets, $V_1, V_2, \ldots, V_n$ for some $n > 0$, where each set V_i consists of the variables associated with a single component i of the system. Then, for $1 \leq i \leq n$, $V_i = \{v_{i,1}, v_{i,2}, \ldots, v_{i,l_i}\}$, for some $v_{i,j} \in V$ and $l_i > 0$, such that $\sum_{i=1}^{n} l_i = l$. If component i is a global variable, V_i consists of a single variable. If component i is a buffered channel with capacity t then V_i consists of t variables, one for each place in the buffer. On the other hand, if i is a process, V_i is the set of local variables for that process. Then $D = \prod_{i=1}^{n} \prod_{j=1}^{l_i} \widetilde{D}_{i,j}$ where $\widetilde{D}_{i,j}$ 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 V_i. For $1 \leq i \leq n$, define a map $\theta_i : D \to \{1, 2, \ldots, m^f\}$ as follows: for any state $s = (d_{1,1}, d_{1,2}, \ldots, d_{1,l_i}, \ldots, d_{n,1}, d_{n,1}, \ldots, d_{n,l_n}) \in D$, $\theta_i(d) = \sum_{j=1}^{l_i} d_{i,j} m^{j-1}$. Define $\theta(s) = (\theta_1(s), \theta_2(s), \ldots, \theta_n(s))$. It is straightforward to check that θ is injective. We can then define a Kripke structure $\mathcal{M}' = (S', s_0', R')$ thus:

- $S' = \theta(D)$
- $s_0' = \theta(s_0)$
- $R' = \{(\theta(s), \theta(t)) : (s, t) \in R\}$.

The structure $\mathcal{M}'$ is obtained from $\mathcal{M}$ by representing all variables for a given component by a single variable with a larger domain. Clearly $\mathcal{M}'$ is essentially *the same* as $\mathcal{M}$: since θ is injective and is (trivially) a surjection from D to $\theta(D)$, we can always translate a state of $\mathcal{M}'$ back to a unique state of $\mathcal{M}$.

The above argument justifies the assumption made throughout this chapter that a Kripke structure representing a system of n components can be defined in terms of a set of n variables, each with finite domain $L \subset \mathbb{Z}$, so that a state is a vector in L^n. We compare states using $\leq$, the natural lexicographic ordering on vectors (see Section 3.4). Throughout, let $I = \{1, 2, \ldots, n\}$.

9.1.1 Action of S_n on states

Let $G \leq S_n$ and $s = (x_1, x_2, \ldots, x_n) \in L^n$. Assuming that components in the system do not hold references to one another, we can define the state $\alpha(s)$ as follows:

$$\alpha(s) = (x_{\alpha^{-1}(1)}, x_{\alpha^{-1}(2)}, \ldots, x_{\alpha^{-1}(n)}).$$

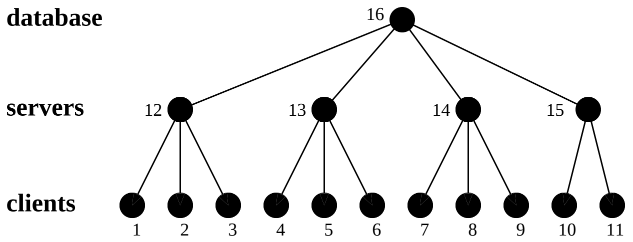


Figure 9.1: Communication structure for a three-tiered architecture.

In Chapter 10 we consider a more realistic model of computation where components may hold references to one another, and present a corresponding action of S_n. Throughout this chapter we use $min[s]_G$ to denote $min_{\leq}[s]_G$, the $\leq$-minimum state in the orbit of s under G.

9.1.2 Symmetry detection

We assume that generators for a symmetry group G have been computed via the communication structure associated with a high level specification, or have been provided explicitly. To make our results general, we do not assume that symmetry has necessarily been detected using the techniques of Chapter 7.

For illustration, throughout this chapter we consider a system with a three-tiered architecture based on the example in Section 4.5. Figure 9.1 shows a possible communication graph for this system, which we assume has been extracted from a specification of the system by some symmetry detection tool. Let $\mathcal{M}_{3T}$ be a model of the system. Using the GRAPE program, we compute G_{3T}, the automorphism group of the communication graph in terms of generators:

$$\begin{aligned} G_{3T} &= \langle (1\ 2), (2\ 3), (4\ 5), (5\ 6), (7\ 8), (8\ 9), (10\ 11), \\ &\quad (12\ 13)(1\ 4)(2\ 5)(3\ 6), (13\ 14)(4\ 7)(5\ 8)(6\ 9) \rangle. \end{aligned}$$

Note that the last two elements of the generating set of G_{3T} are products of transpositions. We assume that $\rho(G_{3T}) \leq Aut(\mathcal{M}_{3T})$ (where ρ is the permutation representation of G_{3T} on the states of $\mathcal{M}_{3T}$), and will use this group and its subgroups as examples to illustrate some of our techniques.

9.2 Exploiting Basic Symmetry Groups

9.2.1 Efficient application of permutations

Before we discuss symmetry reduction strategies, we consider the problem of applying a permutation α to a state s (i.e. computing $\alpha(s)$).

Direct application of a permutation α to a state $s = (x_1, x_2, \ldots, x_n)$ clearly requires exactly n operations: we must compute $x_{\alpha^{-1}(i)}$ for each i. On the other hand, applying a transposition $(i\ j)$ to s is a constant time operation – the local states x_i and x_j are simply exchanged.

Lemma 6 *Let $\alpha \in S_n$. Then α can be expressed as a product of at most $n-1$ transpositions.*

Proof If α is a cycle $(a_1\ a_2\ \ldots\ a_m)$ for some $m \leq n$ then α can be expressed as a product of $m-1$ transpositions: $\alpha = (a_1\ a_2)(a_1\ a_3)\ldots(a_1\ a_m)$ (where $1 \leq a_i \leq n$ for each $1 \leq i \leq m$) [81]. Suppose α is instead a product of l disjoint cycles, $\alpha_1, \alpha_2, \ldots, \alpha_l$, for some $l > 0$, where cycle α_i has length m_i $(1 \leq i \leq l)$. We have $\sum_{i=1}^{l} m_i \leq n$. Since each α_i can be written as a product of $m_i - 1$ transpositions, α can be written as a product of $\sum_{i=1}^{l}(m_i - 1) \leq n-1$ transpositions. ■

In Section 11.3 we provide experimental evidence that representing a permutation α as a list of transpositions, and computing $\alpha(s)$ by successively applying these transpositions, speeds up symmetry reduction by a significant constant factor.

9.2.2 Enumerating small groups

The most obvious strategy for computing $min[s]_G$ is to consider each state in $[s]_G$, and return the smallest. This can be achieved by *enumerating* the elements $\alpha(s)$, $\alpha \in G$. If G is small then this strategy is feasible in practice, and provides an exact symmetry reduction strategy. The SymmSpin package provides an enumeration strategy for full symmetry groups, which is optimised by generating permutations incrementally by composing successive transpositions.

We generalise this optimisation for arbitrary groups using *stabiliser chains*. A stabiliser chain for G is a series of subgroups of the form $G = G^{(1)} \geq G^{(2)} \geq \cdots \geq G^{(k)} = \{id\}$, for some $k > 1$, where $G^{(i)} = stab_{G^{(i-1)}}(x)$ for some $x \in moved(G^{(i-1)})$ $(2 \leq i \leq k)$. If $U^{(i)}$ is a set of representatives for the cosets of $G^{(i)}$ in $G^{(i-1)}$ $(2 \leq i \leq k)$, then each element of G can be uniquely expressed as a product $u_{k-1}u_{k-2}\ldots u_1$, where $u_i \in U^{(i)}$ $(1 \leq i < k)$ [19]. Permutations can be generated incrementally using elements from the coset representatives, and the set of images of a state s under G computed using a sequence of partial images (see Algorithm 5). To ensure efficient application of permutations, the coset representatives are stored as a list of transpositions, applied in succession, as described above.

GAP provides functionality to efficiently compute a stabiliser chain and associated coset representatives for an arbitrary permutation group. Although this approach still involves enumerating the elements $\alpha(s)$ for every $\alpha \in G$ (and is thus infeasible for large groups), calculating each $\alpha(s)$ is faster. The experimental results of Section 11.3 show an improvement over basic enumeration. Additionally, it is only necessary to store coset representatives, rather than all elements of G.

Stabiliser chains are used extensively in computational group theory [19, 63],

Algorithm 5 Computing $min[s]_G$ using a stabiliser chain.

```
min[s]_G := s
for all u_1 ∈ U_1 do
  s_1 := u_1(s)
  for all u_2 ∈ U_2 do
    s_2 := u_2(s_1)
    ⋮
    for all u_k ∈ U_k do
      s_k := u_k(s_{k-1})
      if s_k < min[s]_G then
        min[s]_G := s_k
      end if
    end for
    ⋮
  end for
end for
```

and have been utilised in symmetry breaking approaches for constraint programming [64]. We are, to our knowledge, the first to apply these techniques to model checking.

9.2.3 Minimising sets for G if $G \cong S_m$ ($m \leq n$)

For systems where there is full symmetry between components, the smallest state in the orbit of $s = (x_1, x_2, \ldots, x_n)$ can be computed by *sorting* the tuple s lexicographically. [14, 27]. For example, 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. Since sorting can be performed in polynomial time, this provides an efficient solution for the COP when $G = S_n$.

Recall the group G_{3T} of automorphisms of the communication graph of Figure 9.1. Consider the subgroup:

$$H = \langle (12\ 13)(1\ 4)(2\ 5)(3\ 6), (13\ 14)(4\ 7)(5\ 8)(6\ 9) \rangle.$$

This group permutes *server* components 12, 13 and 14, with their associated blocks of *client* components. It is clear that H is isomorphic to S_3, the symmetric group on 3 objects. However, we cannot compute $min[s]_H$ by sorting s in the usual way, since this is equivalent to applying an element $\alpha \in S_{16}$ to s, which may not belong to H.

We can deal with a group G acting in this way using a *minimising set* for G. Using terminology from [59], G is said to be *nice* if there is a small set $X \subseteq G$ such that, for any $s \in S$, $s = min[s] \Leftrightarrow s \leq \alpha(s)\ \forall\ \alpha \in X$. In this case we call X a *minimising set* for G. If a small minimising set X can be found for a large group G, then computing the representative of a state involves iterating over the small set X, minimising the state until a fix-point is reached. At this point, no element of

Algorithm 6 State minimisation using a minimising set X.

```
min := s
repeat
  min' := min
  for α ∈ X do
    if α(min') < min then
      min := α(min')
    end if
  end for
until min' = min
```

the minimising set maps the state to a smaller image, thus the minimal element has been found. This approach is described by Algorithm 6.

We show that for a large class of groups which are isomorphic to S_m for some $m \leq n$, a minimising set with size polynomial in m can be efficiently found. This minimising set is derived from the swap permutations used in a selection sort algorithm. As discussed in Definition 10 (Section 3.1.2), we use $orb_G(i)$ rather than $[i]_G$ to refer to the orbit of $i \in I$ under G. This is to avoid confusion between orbits of states and orbits of component identifiers.

Theorem 16 *Suppose that, for each $i \in I$ such that $orb_G(i)$ is non-trivial (see Definition 10), $stab_G(i)$ fixes exactly one element from each non-trivial orbit of G acting on I, and that $G \cong S_m$, where $m = |orb_G(i)| > 1$ for some $i \in I$. Then there is an isomorphism $\theta : S_m \to G$ such that $\{(i\ j)^\theta : 1 \leq i < j \leq m\}$ is a minimising set for G.*

Proof Assume without loss of generality that all orbits of I under G are non-trivial, and let $\Omega = \{x_1, x_2, \ldots, x_m\}$ be one of the orbits. For $1 \leq i \leq m$, let $C_i = \{x \in I : \alpha(x) = x \text{ for all } \alpha \in stab_G(x_i)\}$. By our hypothesis, C_i consists of one element from each orbit $1 \leq i \leq m$, and it is easy to show that the C_i are disjoint. We call each C_i a *column*. There is an isomorphism θ from G' (the action of G on the columns) to G acting on I. Since $G \cong S_m$, G' contains all column transpositions $(i\ j)$, so $(i\ j)^\theta \in G$ for any $1 \leq i < j \leq m$. The element $(i\ j)^\theta$ maps all elements of column i to elements of column j.

Now consider states s and s', where $s' = \alpha(s)$ for some $\alpha \in G$. Let i be the smallest index for which $s(i) \neq s'(i)$. Let j be the index such that $j = \alpha^{-1}(i)$. All of the elements in the column containing j (column j' say) are mapped via α to the column containing i (column i' say). Then $s' < s$ iff $(i'\ j')^\theta s < s$. Hence s is minimal in its orbit iff $(i\ j)^\theta(s) \geq s$ for all $i < j$. So the set $\{(i\ j)^\theta : 1 \leq i < j \leq m\}$ is a minimising set for G. ■

Note that the minimising set is much smaller than G, and the conditions of Theorem 16 can be easily checked using GAP. Although testing two arbitrary groups for isomorphism can be very inefficient, if a set of m candidate columns is found, testing whether the action of G on the columns is isomorphic to S_m can be done efficiently using the GAP function `IsNaturalSymmetricGroup(G)`.

It may seem that the conditions of Theorem 16 are unnecessary, and that, given any isomorphism $\theta : S_m \to G$, the set $\{(i\ j)^\theta : 1 \leq i < j \leq m\}$ is a minimising set for G. However, consider the group G below, which is a subgroup of the symmetry group of a hypercube (see Section 4.6):

$$G = \langle (1\ 2)(5\ 6)(9\ 10)(13\ 14), (1\ 2\ 4\ 8)(3\ 6\ 12\ 9)(5\ 10)(7\ 14\ 13\ 11) \rangle \leq S_{14}$$

G is isomorphic to S_4, with an isomorphism $\theta : S_4 \to G$ is defined on generators by $(1\ 2\ 3\ 4)^\theta = (1\ 2\ 4\ 8)(3\ 6\ 12\ 9)(5\ 10)(7\ 14\ 13\ 11)$, $(1\ 2)^\theta = (4\ 8)(5\ 9)(6\ 10)(7\ 11)$. The state $s = (6, 10, 3, 6, 3, 5, 7, 10, 4, 8, 2, 1, 9, 3) \in \{1, 2, \ldots, 10\}^{14}$ *can not* be minimised using the set $\{(i\ j)^\theta : 1 \leq i < j \leq 4\}$.

Theorem 17 *If G satisfies the conditions of Theorem 16 and $X = \{(i\ j)^\theta : 1 \leq i < j \leq m\}$ then $min[s]_G$ can be computed in $O(m^3)$ time for any $s \in L^n$, using Algorithm 6.*

Proof Clearly $|X| = |\{(i\ j)^\theta : 1 \leq i < j \leq m\}| = m(m-1)/2$.

A *column entry* for a state s with respect to a column C_i is a tuple of local states of s whose indices (in s) belong to C_i. Clearly we can order column entries lexicographically. An element $(i\ j)^\theta$ of X has the effect of transposing two column entries for a given state s. We say that the column entry for s with respect to C_i has index i. Now suppose that the smallest column entry for s has index j. Then clearly $min\{\alpha(s) : \alpha \in X\}\ (i\ j)^\theta s$. Hence, after the first iteration of the outer loop of Algorithm 6, the state min' has its first (left-most) column entry as small as possible. Similarly, after the second iteration min' has (at least) its first and second column entries as small as possible, and after m iterations all column entries are ordered in such a way that $min' = min$, in which case the outer loop terminates.

We have shown that, in the worst case, Algorithm 6 involves iterating m times over a set of size $O(m^2)$. The result follows. ■

Each iteration of the outer loop of Algorithm 6 applies every element of X to min', the minimum state found by the previous iteration, and updates min' to the smallest image under X. Algorithm 7 works similarly, but updates the current minimum *every time* an element of X is found which yields a smaller image. We have found that this works better in practice.

9.2.4 Local search for unclassifiable groups

If G is large group then computing $min[s]_G$ by enumeration of the elements of G may be infeasible, even with the group-theoretic optimisations discussed in Sec-

Algorithm 7 Optimised state minimisation using a minimising set X.

$min := s$
repeat
  $min' := min$
  for $\alpha \in X$ **do**
    if $\alpha(min) < min$ **then**
      $min := \alpha(min)$
    end if
  end for
until $min' = min$

tions 9.2.1 and 9.2.2. If no minimising set is available for G, and G cannot be classified as a composite symmetry group (see Sections 9.3 and 9.4) then we must exploit G via an approximate symmetry reduction strategy.

We propose an approximate strategy based on *gradient-descent local search*,[1] which has proved successful for a variety of search problems in artificial intelligence [150]. In this case the function *min* works by performing a local search of $[s]_G$ starting at s, using the generators of G as operations from which to compute a successor state. The search starts by setting $t = s$, and proceeds iteratively. On each iteration, $\alpha(t)$ is computed for each generator α of G. If $t \leq \alpha(t)$ for all α then a local minimum has been reached, and t is returned as a representative for $[s]_G$. Otherwise, t is set to the smallest image $\alpha(t)$, and the search continues. In Section 11.3 we show that this local search algorithm is effective when exploring the state spaces of various configurations of message routing in a hypercube network.

There are various local search techniques which could be employed to attempt to improve the accuracy of this strategy. *Random-restart* hillclimbing [150] involves the selection of several random elements of $[s]_G$ in addition to s, and performing local search from each of them, returning the smallest result. In our case we could apply such a technique by finding the image of a state s under distinct, random elements of G (GAP provides functionality for generating random group elements). Another potential improvement would be to use *simulated annealing* [108] to escape local minima.

9.3 Exploiting Disjoint Products

Certain kinds of symmetry groups can be decomposed as a product of subgroups. In this case it may be possible to solve the COP separately for each subgroup, providing a solution to the COP for the whole group. In particular, if a symmetry group permutes disjoint sets of components independently then the group can be described as the *disjoint product* of the groups acting on these disjoint sets (see Definition 15, Section 3.1.4).

1. This is referred to in [43, 44] as *hillclimbing* local search.

Disjoint products occur frequently in model checking problems. For example, the symmetry group associated with the resource allocator specification of Section 4.4 is a disjoint product of two groups, which independently permute components with priority levels 0 and 1 respectively. In our three-tiered architecture example (see Section 9.1.2), the group G_{3T} can be shown to decompose as a disjoint product $G_{3T} = H_1 \bullet H_2$ where:

$$\begin{aligned} H_1 &= \langle (1\,2), (2\,3), (4\,5), (5\,6), (7\,8), (8\,9), \\ &\quad (12\,13)(1\,4)(2\,5)(3\,6), (13\,14)(4\,7)(5\,8)(6\,9) \rangle \\ H_2 &= \langle (10\,11) \rangle. \end{aligned}$$

If G is a disjoint product of subgroups $H_1, H_2, \ldots, H_k$ then $min[s]_G = min[\ldots min[min[s]_{H_1}]_{H_2} \ldots]_{H_k}$ [27], so the COP for G can be solved by considering each subgroup H_i in turn. Even if it is necessary to enumerate over the elements of each H_i, it is more efficient to enumerate over the resulting $\sum_{i=1}^{k} |H_i|$ elements than the $\prod_{i=1}^{k} |H_i|$ elements of G. Furthermore, it may be that some or all of the H_i can be handled using minimising sets, or wreath product decompositions (see Section 9.4).

However, the above result is only useful when designing a fully automatic symmetry reduction package if it is possible to automatically and efficiently determine, before search, whether or not G decomposes as a disjoint product of subgroups.

We present two solutions to this problem: a sound, incomplete approach which runs in polynomial time, and a sound, complete approach which in the worst case runs in exponential time. We show that the second approach can be optimised using computational group theory to run efficiently for the kind of symmetry groups which arise in model checking problems.

9.3.1 Efficient, sound, incomplete approach

Let $G = \langle X \rangle$ for some $X \subseteq G$ with $id \notin X$. Define a binary relation $B \subseteq X^2$ as follows: for all $\alpha, \beta \in X$, $(\alpha, \beta) \in B \Leftrightarrow moved(\alpha) \cap moved(\beta) \neq \emptyset$. Clearly B is symmetric, and since for any $\alpha \in G$ with $\alpha \neq id$, $moved(\alpha) \neq \emptyset$, B is reflexive. It follows that the transitive closure of B, denoted B^*, is an equivalence relation on X. We now show that if B^* has multiple equivalence classes then each class generates a subgroup of G which is a non-trivial factor for a disjoint product decomposition of G.

Lemma 7 *Suppose that $\alpha, \beta \in X$, and that $(\alpha, \beta) \notin B^*$. Then $moved(\alpha) \cap moved(\beta) = \emptyset$ and α and β commute.*

Proof If $moved(\alpha) \cap moved(\beta) \neq \emptyset$ then $(\alpha, \beta) \in B \subseteq B^*$, a contradiction, thus $moved(\alpha) \cap moved(\beta) = \emptyset$. Therefore if α_1 and β_1 are cycles in the disjoint cycle forms of α and β respectively then α_1 and β_1 are disjoint and therefore commute. By repeatedly swapping disjoint cycles, it follows that $\alpha\beta = \beta\alpha$. ■

Theorem 18 *Suppose $C_1, C_2, \ldots, C_k$ are the equivalence classes of X under B^* where $k \geq 2$. For $1 \leq i \leq k$ let $H_i = \langle C_i \rangle$. Then $G = H_1 \bullet H_2 \bullet \cdots \bullet H_k$, and $H_i \neq \{id\}$ $(1 \leq i \leq k)$.*

Proof Clearly $H_1 H_2 \ldots H_k \subseteq G$. If $\alpha \in G$ then $\alpha = \alpha_1\alpha_2\ldots\alpha_d$ for some $\alpha_1, \alpha_2, \ldots, \alpha_d \in X$, $d > 0$. By Lemma 7 we can arrange the α_l so that elements of C_i appear before those of C_j whenever $i < j$. It follows that $G = H_1 H_2 \ldots H_k$. By Lemma 7, $moved(H_i) \cap moved(H_j) = \emptyset$ for $1 \leq i \neq j \leq k$ and so $G = H_1 \bullet H_2 \bullet \cdots \bullet H_k$, where (since $id \notin X$) the H_i are non-trivial. ■

Consider the group G_{3T} which is generated by the set $X = \{(1\ 2), (2\ 3), (4\ 5), (5\ 6), (7\ 8), (8\ 9), (12\ 13)(1\ 4)(2\ 5)(3\ 6), (13\ 14)(4\ 7)(5\ 8)(6\ 9)(10\ 11)\}$. It is straightforward to check that the equivalence classes under B^* for this example are as follows:

$$\begin{aligned} C_1 = \{&(1\ 2), (2\ 3), (4\ 5), (5\ 6), (7\ 8), (8\ 9), (12\ 13)(1\ 4)(2\ 5)(3\ 6), \\ &(13\ 14)(4\ 7)(5\ 8)(6\ 9)\} \\ C_2 = \{&(10\ 11)\}, \end{aligned}$$

which generate the groups H_1 and H_2 respectively, described at the start of Section 9.3. This is the finest disjoint product decomposition of G_{3T}.

The approach is incomplete as it does not guarantee the finest decomposition of an arbitrary group G as a disjoint product. To see this, suppose that the element $(1\ 2)(10\ 11)$ is added to the generating set for the group G_{3T}. This causes the equivalence classes C_1 and C_2 to merge, and a non-trivial disjoint decomposition for G_{3T} is not obtained.

However, in practice we have not found a case in which the finest decomposition is not detected when generators have been computed by a graph automorphism program. The approach is very efficient as it works purely with the generators of G, of which there are typically few.

9.3.2 Sound and complete approach

We now present an algorithm for computing the finest non-trivial decomposition of G as a disjoint product of subgroups. The algorithm runs in exponential time in the worst case, but for many groups which arise in model checking problems we can obtain polynomial run-time via a computational group theoretic optimisation. We present three lemmas, the proofs of which can be found in Appendix B.3. Throughout this section we use (variations of) Ω and $\mathcal{O}$ to refer to orbits and sets of orbits respectively.

Algorithm 8 *disjoint_decomposition*$(G, \mathcal{O})$ – G is a group and $\mathcal{O}$ its non-trivial orbits.

for all partitions $\{\mathcal{O}_1, \mathcal{O}_2\}$ of $\mathcal{O}$ **do**
  if $G^{\mathcal{O}_1} \leq G$ and $G^{\mathcal{O}_2} \leq G$ **then**
    return *disjoint_decomposition*$(G^{\mathcal{O}_1}, \mathcal{O}_1) \bullet$ *disjoint_decomposition*$(G^{\mathcal{O}_2}, \mathcal{O}_2)$
  end if
end for
return G

Let $G \leq S_n$, and $\mathcal{O}$ the set of all non-trivial orbits of G. For $\mathcal{O}' \subseteq \mathcal{O}$, any $\alpha \in G$ can be written as $\alpha = \alpha_1\alpha_2 \ldots \alpha_s\beta_1\beta_2 \ldots \beta_t$, where $moved(\alpha_i) \subseteq \Omega \in \mathcal{O}'$ $(1 \leq i \leq s)$ and $moved(\beta_i) \subseteq \Omega \in (\mathcal{O} \setminus \mathcal{O}')$ $(1 \leq i \leq t)$. With α in this form, the *restriction* of α to $\mathcal{O}'$ is the permutation $\alpha^{\mathcal{O}'} = \alpha_1\alpha_2 \ldots \alpha_s$. In general, $\alpha^{\mathcal{O}'} \notin G$. For $H \leq G$, the *restriction* of H to $\mathcal{O}'$ is the group $H^{\mathcal{O}'} = \{\alpha^{\mathcal{O}'} : \alpha \in H\}$. In general, $H^{\mathcal{O}'} \not\leq G$.

Lemma 8 *Suppose $G = H_1 \bullet H_2$ where $H_1 \neq \{id\}$ and $H_2 \neq \{id\}$. Then there are sets $\mathcal{O}_1$, $\mathcal{O}_2$ of non-trivial orbits of G such that $\{\mathcal{O}_1, \mathcal{O}_2\}$ is a partition of $\mathcal{O}$ and for $i \in \{1, 2\}$, $H_i = G^{\mathcal{O}_i}$.*

Algorithm 8 is a recursive algorithm for computing a disjoint decomposition of G. If G can be decomposed, then by Lemma 8 there is some partition $\{\mathcal{O}_1, \mathcal{O}_2\}$ of $\mathcal{O}$ such that $G = G^{\mathcal{O}_1} \bullet G^{\mathcal{O}_2}$. The algorithm detects when a partition with this property has been found, based on the following lemma:

Lemma 9 *If $\{\mathcal{O}_1, \mathcal{O}_2\}$ is a partition of $\mathcal{O}$ and $G^{\mathcal{O}_i} \leq G$ for $i \in \{1, 2\}$ then $G = G^{\mathcal{O}_1} \bullet G^{\mathcal{O}_2}$.*

Once a decomposition of the form $G = G^{\mathcal{O}_1} \bullet G^{\mathcal{O}_2}$ has been found, the groups $G^{\mathcal{O}_1}$ and $G^{\mathcal{O}_2}$ are recursively decomposed. This guarantees the finest decomposition of G as a disjoint product, thus Algorithm 8 is complete.

Computing $G^{\mathcal{O}_i}$ by restricting each generator of G to $\mathcal{O}_i$ is trivial. Testing whether $G^{\mathcal{O}_i} \leq G$ can be done in low-degree polynomial time using standard computational group theoretic data structures [19]. Thus the complexity of Algorithm 8 is dominated by the number of partitions of $\mathcal{O}$ which must be considered in the worst case. If G *does not* decompose as a disjoint product then every partition of $\mathcal{O}$ of size two must be considered. The number of such partitions is $S(|\mathcal{O}|, 2)$, a Stirling number of the second kind [70], and it can be shown that $S(|\mathcal{O}|, 2) = 2^{|\mathcal{O}|-1} - 1$. In the worst case, $|\mathcal{O}|$ may be $n/2$, thus the complexity of Algorithm 8 is $O(2^n)$.

A computational group theoretic optimisation

We can optimise the performance of Algorithm 8 for many commonly occurring symmetry groups by introducing the notion of *dependent orbits*:

Definition 30 *Let $\Omega_1, \Omega_2 \in \mathcal{O}$. We say that Ω_1 is dependent on Ω_2 if $|stab^*_G(\Omega_2)^{\Omega_1}| < |G^{\Omega_1}|$.*

Intuitively, Ω_1 is dependent on Ω_2 if fixing each point in Ω_2 has an effect on the action of G on Ω_1. It is easy to show that Ω_1 is dependent on Ω_2 iff Ω_2 is dependent on Ω_1, so we say that two orbits are *dependent* if one is dependent on the other. We now show that dependent orbits must belong to the same element of the partition of Lemma 8:

Lemma 10 *Let $\{\mathcal{O}_1, \mathcal{O}_2\}$ be a partition of $\mathcal{O}$ such that $G = G^{\mathcal{O}_1} \bullet G^{\mathcal{O}_2}$ (as in Lemma 8). Let $\Omega_i, \Omega_j \in \mathcal{O}$ be dependent. Then $\{\Omega_i, \Omega_j\} \subseteq \mathcal{O}_1$ or $\{\Omega_i, \Omega_j\} \subseteq \mathcal{O}_2$.*

Define a binary relation $B \subseteq \mathcal{O} \times \mathcal{O}$ as follows: $(\Omega_1, \Omega_2) \in B$ if Ω_1 and Ω_2 are dependent. We have already established that B is symmetric, and it is obviously reflexive. We have not determined whether B is, in general, transitive, so we use B^* to denote the transitive closure of B. Suppose $\{\mathcal{O}_1, \mathcal{O}_2\}$ is a partition of $\mathcal{O}$ with $G = G^{\mathcal{O}_1} \bullet G^{\mathcal{O}_2}$ (as in Lemma 8). If C is an equivalence class of B^*, called a *dependency class*, then by Lemma 10 and induction, $C \subset \mathcal{O}_i$ for some i.

Since *Algorithm* 8 depends critically on the size of the set $\mathcal{O}$, we can potentially improve performance by taking $\mathcal{O}$ to be the set of all dependency classes, rather than the set of all orbits, if there are fewer dependency classes. Computing the dependency classes involves computing pointwise stabilisers, which is a polynomial time operation [19].

Examples

We illustrate the sound and complete approach using a group for which the optimisation above reduces the problem so that there is only one potential partition $\{\mathcal{O}_1, \mathcal{O}_2\}$ to consider. We also give a pathological example for which our optimisation does not help at all.

Let G be the following group:

$$\begin{aligned} G = \langle & (1\ 2\ 3)(4\ 5\ 6)(7\ 8\ 9)(10\ 11\ 12)(14\ 15)(17\ 18)(20\ 21), \\ & (2\ 3)(5\ 6)(8\ 9)(11\ 12)(13\ 14\ 15)(16\ 17\ 18)(19\ 20\ 21)\rangle. \end{aligned}$$

Due to the manner in which the generators of G have been presented, applying the sound and complete approach of Section 9.3.1 does not yield a disjoint product decomposition. Using GAP, we find that G has seven non-trivial orbits:

$$\mathcal{O} = \{\{1,2,3\},\{4,5,6\},\{7,8,9\},\{10,11,12\},\{13,14,15\},\{16,17,18\},\{19,20,21\}\}$$

and there are $S(7,2) = 63$ partitions of these orbits. However, analysing these orbits for dependency, we find that the orbits $\mathcal{O}_1 = \{\{1,2,3\},\{4,5,6\},\{7,8,9\},\{10,11,12\}\}$ are all dependent, and $\mathcal{O}_2 = \{\{13,14,15\},\{16,17,18\},\{19,20,21\}\}$ are all dependent. There is only one partition of $\mathcal{O}$ which preserves these dependencies – the partition $\{\mathcal{O}_1, \mathcal{O}_2\}$. It is straightforward to check that

$$\begin{aligned} G &= G^{\mathcal{O}_1} \bullet G^{\mathcal{O}_2} \\ &= \langle(1,2,3)(4,5,6)(7,8,9)(10,11,12),(1,2)(4,5)(7,8)(10,11)\rangle \\ &\quad \bullet\langle(13,14,15)(16,17,18)(19,20,21),(13,14)(16,17)(19,20)\rangle. \end{aligned}$$

This is an example for which the computational group theoretic optimisation is very effective.

Now consider, for any $n > 2$, the following group:

$$\begin{aligned} G_n = \langle & (1\ 2)(3\ 4) \\ & (3\ 4)(5\ 6) \\ & \vdots \\ & (n-6\ \ n-5)(n-4\ \ n-3) \\ & (n-3\ \ n-2)(n-1\ \ n)\rangle. \end{aligned}$$

It is clear that G_n has $n/2$ non-trivial orbits: $\mathcal{O} = \{\{1,2\},\{3,4\},\{5,6\},\ldots,\{n-1,n\}\}$. It is not so obvious, but easy to check, that no two orbits are dependent. Hence the computational group theoretic optimisation does not reduce the number of partitions of $\mathcal{O}$ which must be checked to determine whether G_n decomposes as a disjoint product. The number of partitions is $S(\frac{n}{2},2) = 2^{n/2} - 1$, and all of these must be checked, since G_n *does not* decompose as a non-trivial disjoint product for any n (this can be proved by induction). However, this is not a group which we have encountered in association with a model checking problem.

An open problem in this area is to determine whether there is a polynomial time algorithm for finding the finest disjoint product decomposition of an arbitrary group G. A possible approach is to find a stronger notion of dependent orbits, with the property that if $C_1, C_2, \ldots, C_h$ are the dependency classes of the orbits then $G = G^{C_1} \bullet G^{C_2} \bullet \cdots \bullet G^{C_h}$.

This problem is of computational group theoretic interest. From a model checking perspective, the sound and incomplete approach of Section 9.3.1 returns the finest disjoint product decomposition of groups whose generators have been automatically computed. The sound, complete approach, with our computational group theoretic optimisation, can efficiently handle all the types of symmetry group which we have observed in connection with model checking problems, regardless of the way their generators are presented.

9.4 Exploiting Wreath Products

Suppose that a symmetry group partitions the components of a system into subsets such that there is analogous symmetry within each subset, and symmetry between the subsets. Then the group can be described as the *inner wreath product* of the group which acts on the subsets, and the group which permutes the subsets (see Definition 17, Section 3.1.4).

Wreath products occur in model checking problems when systems are modelled using a tree structure. In Section 4.5 we established that the symmetry group associated with the three-tiered architecture specification exhibits wreath product symmetry. Recall the group G_{3T} introduced in Section 9.1.2. In Section 9.3, we showed that G_{3T} decomposes as a disjoint product $H_1 \bullet H_2$. We now show that the factor H_1 of this product decomposes as an inner wreath product.

We have $H_1 \leq Sym(X)$ where $X = \{1, 2, \ldots, 9, 12, 13, 14\}$. Consider the following partition $\{X_1, X_2, X_3\}$ of X, where we describe each X_i as an ordered set of elements $X_i = \{x_{i,1}, x_{i,2}, \ldots, x_{i,4}\}$ (as in Definition 17, Section 3.1.4):

$$\begin{aligned} X_1 &= \{1, 2, 3, 12\} \\ X_2 &= \{4, 5, 6, 13\} \\ X_3 &= \{7, 8, 9, 14\} \end{aligned}$$

Taking $K = S_3$ and $H = S_3$,[2] let σ be the permutation representation corresponding to the action of K on X and σ_1, σ_2 and σ_3 those for H on X as in Definition 17. Then:

$$\begin{aligned} \sigma(K) &= \langle (1\,4)(2\,5)(3\,6)(12\,13), (4\,7)(5\,8)(6\,9)(13\,14) \rangle \\ \sigma_1(H) &= \langle (1\,2), (2\,3) \rangle \\ \sigma_2(H) &= \langle (4\,5), (5\,6) \rangle \\ \sigma_3(H) &= \langle (7\,8), (8\,9) \rangle \end{aligned}$$

The group $\sigma(K)$ permutes the partition $\{X_1, X_2, X_3\}$, whereas each group $\sigma_i(H)$ permutes the set X_i. One can verify (using GAP) that $H_1 = \sigma(K)\sigma_1(H)\sigma_2(H)\sigma_3(H)$, i.e. $H_1 = H \wr K$.

2. The group H can be thought of as the subgroup of $Sym(\{1, 2, 3, 4\})$ which fixes the point 4.

Suppose that $G \leq S_n$ has a non-trivial (inner) wreath product decomposition $(H, K, \mathcal{X})$ with associated permutation representations $\sigma, \sigma_1, \sigma_2, \ldots, \sigma_d$ for the actions of K and H on $\{1, 2, \ldots, n\}$ (where $d = |\mathcal{X}|$). For a state $s \in L^n$ it can be shown that $min[s]_G = min[min[\ldots min[min[s]_{\sigma_1(H)}]_{\sigma_2(H)} \cdots]_{\sigma_d(H)}]_{\sigma(K)}$ [27]. This means that the COP for G can be solved by considering each subgroup $\sigma_i(H)$ in turn, followed by the subgroup $\sigma(K)$. Even if we have to deal with these groups using enumeration, it is more efficient to enumerate over the resulting $d \times |H| + |K|$ elements than all $|H|^d|K|$ elements of G. Furthermore, it may be possible to deal with the groups $\sigma(K)$ and $\sigma_i(H)$ $(1 \leq i \leq d)$ efficiently using minimising sets or further disjoint/wreath decompositions.

As with the similar result for disjoint products presented in Section 9.3, the result for wreath products is only useful for automatic symmetry reduction if we can automatically determine, before search, whether an arbitrary permutation group is a wreath product. We present an algorithm to determine whether a group G decomposes as a wreath product for the case when G is *transitive* (see Definition 10, Section 3.1.2). We then propose an extension of our approach to the case where G may not be transitive.

9.4.1 Wreath product decomposition for transitive groups

If G is a transitive permutation group then we can determine whether G has wreath product structure by considering the *block systems* of G. We introduce some standard definitions and results on block systems. See [85, 148] for details.

Definition 31 *Let $G \leq Sym(X)$ and $Y \subseteq X$, where X is a non-empty set. Then Y is a* block *for G iff, for all $\alpha \in G$, $\alpha(Y) = Y$ or $\alpha(Y) \cap Y = \emptyset$.*

Essentially a block is a subset of X which is either fixed by an element of G, or moved *completely* by the element. The sets X, $\{x\}$ (for any $x \in X$), and $\emptyset$ are always blocks for G, and are called *trivial* blocks. Given a non-empty block Y, it can be shown that the set $\{\alpha(Y) : \alpha \in G\}$ is a partition of G, each set in this partition is a block, and all the blocks have the same size. Such a partition is called a *block system* for G, *generated* by Y. In general, rather than singling out a specific block, we say that a partition $\mathcal{X} = \{X_1, X_2, \ldots, X_d\}$ of X is a block system for G if each X_i is a block for G, and the blocks are all images of each other under G. A *non-trivial* block system is one for which the blocks are non-trivial.

Definition 32 *Let $\{X_1, X_2, \ldots, X_d\}$ be a block system for $G \leq Sym(X)$. For $1 \leq i \leq d$, the group $(stab_G(X_i))^{X_i}$ is called the* block stabiliser *for X_i.*

This is the restriction of the group $stab_G(X_i)$ to the block X_i, and is analogous to the restriction of a group to a union of orbits in Section 9.3.2. This restriction is well-defined since X_i is clearly an orbit of $stab_G(X_i)$. It can be shown that for any blocks X_i, X_j, $(stab_G(X_i))^{X_i}$ and $(stab_G(X_j))^{X_j}$ are identical up to renaming of the points on which they act. If $|X_i| = m$ we can identify all of the groups $(stab_G(X_i))^{X_i}$ with a group $H \leq S_m$ by renaming points in the obvious way. We call H the *block stabilizer* for the system.

The block stabiliser for X_i shows the effect of G on the points contained in X_i. The effect of G on the blocks, regarded as "black boxes", is characterised by the *block permuter*:

Definition 33 *Let $\mathcal{X} = \{X_1, X_2, \ldots, X_d\}$ be a block system for $G \leq Sym(X)$. For $\alpha \in G$, define $\alpha(X) = \{\alpha(x) : x \in X\}$ in the usual way. It is easy to check that this is an action of G on $\mathcal{X}$ (see Definition 13, Section 3.1.3). Let σ be the permutation representation of this action so that $\sigma(G) \leq Sym(\mathcal{X})$. We can identify $Sym(\mathcal{X})$ with S_d by renaming X_i as i ($1 \leq i \leq d$). The group obtained by regarding $\sigma(G)$ as a subgroup of S_d is called the* block permuter *for $\mathcal{X}$.*

The following important theorem in wreath product theory (see, for example, [127] for a proof) shows that if G is a transitive permutation group which admits a non-trivial block system then G is contained in an (inner) wreath product. The theorem is followed by a straightforward lemma.

Theorem 19 *Let $G \leq Sym(X)$ be transitive and $\mathcal{X}$ a non-trivial block system for G. Let H and K be the block stabiliser and block permuter for $\mathcal{X}$ respectively. Then H and K are non-trivial and G is contained in the (non-trivial) inner wreath product of H and K with associated partition $\mathcal{X}$, i.e. $G \leq H \wr K$.*

Lemma 11 *Let $H \wr K$ be the inner wreath product of Theorem 19, with associated block system $\mathcal{X}$. Let $\sigma_1, \sigma_2, \ldots, \sigma_d$ be the actions of H on X described in Definition 17. Then $\sigma_i(H) = (stab_G(X_i))^{X_i}$, the stabiliser of block X_i ($1 \leq i \leq d$).*

Conversely to Theorem 19, we show that any inner wreath product exhibits a block system.

Lemma 12 *Let $G \leq Sym(X)$ and suppose G is an inner wreath product $H \wr K$ with associated partition $\mathcal{X} = \{X_1, X_2, \ldots, X_d\}$. Then $\mathcal{X}$ is a block system for G.*

Proof Let σ and $\sigma_1, \sigma_2, \ldots, \sigma_d$ be the permutation representations of the actions of K and H on X. Any element $\delta \in G$ has the form $\delta = \sigma(\beta)\sigma_1(\alpha_1)\sigma_2(\alpha_2)\ldots\sigma_d(\alpha_d)$ for some $\beta \in K$, $\alpha_1, \alpha_2, \ldots, \alpha_d \in H$. For any $X_i \in \mathcal{X}$, clearly $\sigma_j(\alpha_j)(X_i) = X_i$, therefore $\delta(X_i) = \sigma(\beta)(X_i)$. By definition of σ, either $\sigma(\beta)(X_i) = X_i$, or $\sigma(\beta)(X_i) \cap X_i = \emptyset$.

Algorithm 9 Computing a wreath product decomposition for a transitive permutation group G.

for all non-trivial block systems $\mathcal{X} = \{X_1, X_2, \ldots, X_d\}$ for G **do**
  $K :=$ block permuter for $\mathcal{X}$
  $\theta : G \to K :=$ permutation representation of action of G on $\mathcal{X}$
  $\sigma_1(H) := (stab_G(X_1))^{X_1}$
  if $|G| = |\sigma_1(H)|^d |K|$ **then**
    for all $i \in \{2, \ldots, d\}$ **do**
      $\sigma_i(H) := (stab_G(X_i))^{X_i}$
    end for
    for all monomorphisms $\sigma' : K \to G$ **do**
      if $K = \theta(\sigma'(K))$ **then**
        $\sigma := \sigma'$
        break
      end if
    end for
    return $\sigma(K), \sigma_1(H), \ldots, \sigma_d(H)$
  end if
end for
return fail

The result follows. ■

The next theorem is a direct consequence of Theorem 19, Lemma 12 and Theorem 4 (Section 3.1.4).

Theorem 20 *Let $G \leq Sym(X)$ be transitive. Then G can be decomposed as a non-trivial inner wreath product $H \wr K$, with associated partition $\mathcal{X}$, iff $\mathcal{X}$ is a non-trivial block system for G, K and H are the block permuter and block stabiliser for $\mathcal{X}$ respectively, and $|G| = |H|^{|\mathcal{X}|}|K|$.*

The consequence of Theorem 20 is that our search for a non-trivial inner wreath product decomposition of an arbitrary transitive permutation group G boils down to searching the non-trivial block systems for G. Given a block system, we know that G is contained in the inner wreath product associated with the block system, and can determine whether G *is* this wreath product by checking the order of G.

Algorithm 9.4.1 (the correctness of which follows from Theorem 20) can be used to find a non-trivial wreath product decomposition for a transitive group G, if one exists. Rather than returning a decomposition in the form $(H, K, \mathcal{X})$, the algorithm returns the groups $\sigma(K)$ and $\sigma_i(H)$ $(1 \leq i \leq d)$, which are all that we require to solve the constructive orbit problem efficiently.

For each non-trivial block system $\mathcal{X}$, the block permuter K and a single block stabiliser $(stab_G(X_1))^{X_1}$ are computed. Since $(stab_G(X_1))^{X_1}$ is isomorphic to the block stabiliser for $\mathcal{X}$ it is sufficient to compare $|G|$ with $|(stab_G(X_1))^{X_1}|^d|K|$ to determine (by Theorem 20) whether the current block system corresponds to a wreath product decomposition. In the case where equality of orders holds, by Lemma 11 the groups $\sigma_i(H)$ can be computed as block stabilisers. The challenge is to compute σ, the permutation representation of the action of K. We know that σ maps K to an isomorphic subgroup of G, therefore σ must be a monomorphism (see Theorem 2, Section 3.1.1). Furthermore, the restriction of $\sigma(K)$ to act on the blocks, i.e. the group $\theta(\sigma(K))$, must be equal to K. Therefore σ can be computed by considering (in the worst case) all monomorphisms from K to G.

Efficiency

We can compute θ, K and an individual block system for G, and determine the orders of K and $(stab_G(X_1)^{X_1})$ in polynomial time using algorithms presented in [85]. Although polynomial time algorithms are not available for computation of arbitrary setwise stabilisers, a block stabiliser $stab_G(X_i)$ *can* be computed in polynomial time [85], after which computing the restricted group $(stab_G(X_i))^{X_i}$ is straightforward. The potential bottlenecks of Algorithm 9.4.1 are: the number of block systems which may need to be considered, and the computation of all monomorphisms from K to G.

It can be shown (by counting chains of blocks) that an upper bound for the number of distinct block systems for a permutation group G is $n^{\log_2 n}$, where n is the degree of G (personal communication, P. J. Cameron, 2007). This upper bound is not too large for the sizes of n which occur in model checking problems.

Computing all monomorphisms from K to G can be achieved via the GAP function `IsomorphicSubgroups(`G`,`K`)` (see Section 3.1.6). The complexity of this algorithm is not documented, but it is not a polynomial-time algorithm (personal communication, S. Linton, 2007). An alternative algorithm for computing $\sigma(K)$ is presented as part of a constructive proof [109, Lemma 2.4], though this algorithm does not appear to be more efficient than `IsomorphicSubgroups`. Note that it is only necessary to compute the monomorphism σ if G does indeed decompose as an inner wreath product. The benefits which can result from having a wreath product decomposition for G may therefore justify this computation.

We have observed that in many practical examples σ is the mapping defined by: $\sigma(\beta)(x_{i,j}) = x_{\beta(i),j}$, where each block X_i has the form $\{x_1, x_2, \ldots, x_m\}$ with $x_i \leq x_j$ whenever $i < j$. Our implementation of Algorithm 9.4.1 tries this simple pre-test for σ before resorting to monomorphism computation.

9.4.2 Extending the approach to intransitive permutation groups

The results of Section 9.4.1 provide a solution to the wreath product decomposition problem for transitive groups. However, wreath product groups which occur in model checking problems are not necessarily transitive. Consider the subgroup H_1 of G_{3T} (see Sections 9.3 and 9.1.2 respectively). H_1 has two orbits, $\{1, 2, \ldots, 9\}$ and $\{12, 13, 14\}$. More generally, the symmetry group associated with a rooted tree is an intransitive wreath product [105]: nodes at differing depths in the tree, or nodes at the same depth which occur in non-isomorphic sub-trees, must be in separate orbits. Unfortunately, there is very little literature on intransitive wreath products. Even works which are dedicated to the topic of wreath products either assume transitivity throughout [109], or only briefly discuss the intransitive case [127].

Transitivity is imposed in Section 9.4.1 due to Theorem 19. The need for transitivity in the proof of Theorem 19 (see [127]) is unclear: it appears that transitivity is required simply because the theorem appears in the context of *imprimitive* permutation groups, which are transitive by definition [148]. We conjecture that Theorem 19 holds when the transitivity condition is omitted.

Assuming this conjecture, there is a further problem: techniques for computing block systems are restricted to transitive groups [85]. We use an algorithm to work around this problem as follows: if G has $f > 1$ distinct orbits then for each orbit Ω we find a (possibly trivial) block system for G^{Ω}. We then attempt to construct a block for G which is the union of f blocks, one from each block system.

Formally, assume that the orbits of G are $\Omega_1, \Omega_2, \ldots, \Omega_f$, and assume without loss of generality that these orbits are non-trivial. For each Ω_i, let $blocks(\Omega_i)$ be the set of all block systems for G^{Ω_i}, excluding $\{\Omega_i\}$ but including the trivial system $\{\{x\} : x \in \Omega_i\}$. For each $\mathcal{X}_1 \in blocks(\Omega_1)$, consider every set of block systems $\{\mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_f\}$ such that $\mathcal{X}_i \in blocks(\Omega_i)$, $|\mathcal{X}_i| = |\mathcal{X}_1|$ for all $i > 1$, and at least one $\mathcal{X}_i$ is non-trivial. We attempt to construct a block from the $\mathcal{X}_i$ as follows: Set $B = X_1$ where X_1 is any block in $\mathcal{X}_1$. Find a block $X_2 \in \mathcal{X}_2$ such that $B \cup X_2$ is a block for G, and set $B = X_1 \cup X_2$. Continue this process until no suitable X_i exists, or $B = X_1 \cup X_2 \cup \cdots \cup X_f$ is a block for G ($X_i \in \mathcal{X}_i$, $1 \leq i \leq f$). In the latter case, store the block system generated by B.

Algorithm 9.4.1 can be applied to the set of block systems for G obtained via this process, to obtain an inner wreath product decomposition.

The symmetry reduction package TopSpin, described in Chapter 11, uses the techniques described above to compute wreath product decompositions for arbitrary groups. If our conjecture above proves to be incorrect, it is possible that our implementation may compute an erroneous wreath product decomposition for a group G. The worst case scenario then is that representative computation for G might result in multiple orbit representatives. This compromises the optimality, but not the soundness, of symmetry reduction.

9.5 Direct and Semi-direct Products

As noted in Section 3.1.4, disjoint products are a special case of *direct* product, and both direct and wreath products are examples of *semi-direct* products. It is natural to ask whether the idea of solving the COP efficiently by decomposing G as a disjoint/wreath product can be extended to apply to direct/semi-direct products. We use the symmetry group of a 3-dimensional hypercube to provide counter-examples which show that a direct extension of the techniques is *not* possible.

Recall the group $K_3 \rtimes S_3$, the symmetry group of a 3-dimensional hypercube (i.e. a cube), introduced in Section 4.6.1. For $1 \leq i \leq 8$ we use the integer i to represent the node in a cube corresponding to the boolean vector for $i-1$.[3] Then the groups K_3 and S_3 can be expressed using generators as follows:

$$\begin{aligned} K_3 &= \langle (1\,2)(3\,4)(5\,6)(7\,8), (1\,3)(2\,4)(5\,7)(6\,8), (1\,5)(2\,6)(3\,7)(4\,8) \rangle \\ S_3 &= \langle (2\,3\,5)(4\,7\,6), (2\,3)(6\,7) \rangle. \end{aligned}$$

Conjecture 1 *If $G \leq S_n$ and $G = H_1 \rtimes H_2$ then, for $s \in L^n$, $min[s]_G = min[min[s]_{H_1}]_{H_2}$,* or *$min[s]_G = min[min[s]_{H_2}]_{H_1}$.*

Counterexample Consider the group $G = K_3 \rtimes S_3 \leq S_8$, and the state $s = (4,3,2,4,2,1,1,3)$. Using GAP, we can compute $min[s]_G$ by enumeration of G, and we find that $min[s]_G = (1,2,2,4,3,1,4,3)$. Again using GAP, we find that $min[s]_{K_3} = (1,2,3,1,3,4,4,2) = t$ say, and that $min[t]_{S_3} = t \neq min[s]_G$. Similarly, we find that $min[min[s]_{S_3}]_{K_3} = (1,2,2,4,3,4,1,3) \neq min[s]_G$. ■

Conjecture 2 *If $G = H_1 \times H_2 \times \cdots \times H_k \leq S_n$ then, for $s \in L^n$, $min[s]_G = min[\ldots min[min[s]_{H_1}]_{H_2} \ldots]_{H_k}$.*

Counterexample It is easy to show that the group K_3 decomposes as a direct product – $K_3 = H_1 \times H_2 \times H_3$, where:

$$\begin{aligned} H_1 &= \langle (1\,2)(3\,4)(5\,6)(7\,8) \rangle \\ H_2 &= \langle (1\,3)(2\,4)(5\,7)(6\,8) \rangle \\ H_3 &= \langle (1\,5)(2\,6)(3\,7)(4\,8) \rangle. \end{aligned}$$

Consider the state $s = (3,4,4,2,5,4,1,5)$. Using enumeration we find that $min[s]_{K_3} = (1,5,5,4,4,2,3,4)$. However, $min[min[min[s]_{H_i}]_{H_j}]_{H_k} = (3,4,4,2,5,4,1,5)$

3. We use the integers $\{1,2,\ldots,8\}$ rather than $\{0,1,\ldots,7\}$ so that we can present examples computed using GAP.

for any distinct $i, j, k \in \{1, 2, 3\}$. This shows that s cannot be minimised by considering H_1, H_2 and H_3 independently, no matter which order they are applied in. ■

Note that Conjectures 1 and 2 show simply that the COP for semi-direct and direct products cannot, in general, be solved by straightforward application of techniques which work for disjoint and wreath products. Of course this does not mean that there is *no* way to efficiently solve the COP by exploiting this product structure.

9.6 Choosing a Strategy for G

The strategies which we have presented for minimising a state with respect to basic and composite groups can be combined to yield a symmetry reduction strategy for the arbitrary group G by classifying the group using a top-down recursive algorithm.

The algorithm starts by searching for a minimising set for G of the form prescribed in Theorem 16, so that $min[s]_G$ can be computed as described in Section 9.2.3. If no such minimising set can be found, a decomposition of G as a disjoint/wreath product is sought. In this case the algorithm is applied recursively to obtain a minimisation strategy for each factor of the product so that $min[s]_G$ can be computed using these strategies as described in Sections 9.3 and 9.4 respectively. If G remains unclassified and $|G|$ is sufficiently small, enumeration is used, otherwise local search (see Section 9.2.4) is selected.

Summary

In this chapter we have developed techniques for solving the Constructive Orbit Problem, which is key to exploiting symmetry in explicit-state model checking. We have described a method for efficiently applying a permutation to a state, an operation which is fundamental to symmetry reduction. We have also shown that a basic symmetry reduction strategy based on enumeration can be optimised by representing a symmetry group using sets of coset representatives for a stabiliser chain.

Previous approaches to symmetry reduction have exploited full symmetry groups by sorting states. We have generalised this idea using the concept of a minimising set, and have shown how a minimising set for many commonly occurring groups which are isomorphic to fully symmetric groups can be computed.

It has been established that the COP can be solved compositionally if a group can be decomposed as a disjoint/wreath product of subgroups [27]. However, these results are only useful for automated model checking if the decomposition process can be automated. We have proposed two approaches to decomposing a group as a disjoint product of subgroups. The first is sound, very efficient, but incomplete. However, we have found it to work well in practice when applied to groups which have been automatically computed using graph automorphism software. The second approach is sound and complete, but runs in exponential time. We have proposed a computational group theoretic optimisation for this approach which works well for commonly occurring groups. We have shown how a wreath product decomposition for a transitive group can be found by examining non-trivial block systems for the group. Using the concept of a pseudo block system, we have extended this decomposition approach to apply to arbitrary imprimitive wreath products, and discussed the efficiency of the decomposition algorithm.

We have shown that, in general, the COP cannot be solved compositionally for groups which decompose as direct or semi-direct products of subgroups by straightforward extension of the techniques for disjoint and wreath products. To show this, we have used counter-examples based on the group $G = K_3 \rtimes S_3$, the symmetry group of a 3-dimensional cube.

Chapter 10

Extending Symmetry Reduction Strategies to a Realistic Model of Computation

When components do not hold references to other components, the simple model of computation and the action of a permutation on a state described in Section 9.1 are sufficient to reason about concurrent systems. The model is common to numerous works on symmetry reductions for model checking (e.g. [27, 57, 59]), and is adequate for reasoning about input languages where components do not individually hold references to other components, e.g. the input languages of SMC [163] and `SYMM` [27], or where components are specified using *synchronisation skeletons* [57].

However, if components *can* hold references to one another then any permutation that moves component i will also affect the local state of any components which refer to component i. Sophisticated specification languages, such as Promela, include data-types to represent process and channel identifiers. Both the results presented in [27] on solving the COP for groups which decompose as disjoint/wreath products, and our results on minimising sets for fully symmetric groups (see Section 9.2.3) do not hold in general for this extended model of computation. We illustrate this using an example in Section 10.1.2.

Thus for Promela specifications where local variables refer to process and channel identifiers, the efficient symmetry reduction strategies presented in Chapter 9 are not always exact; in some cases they may yield an *approximate* implementation of the function *min*, as discussed in Section 3.4.2. This does not compromise the safety of symmetry reduced model checking, but means that symmetry reduction is not memory-optimal.

For the simple case of full symmetry between identical components, the SymmSpin package deals with local variables which are references to component identifiers by dividing the local state of each component into two portions, one which does not refer to other components (the *insensitive* portion), and another which consists entirely of such references (the *sensitive* portion). A state is minimised by first sorting it with respect to the insensitive portion. Then, for each subset of components with identical insensitive portions, every permutation of the subset is considered, and the permutation which leads to the smallest image is applied. This is known as the *segmented* strategy. In this chapter we show that the segmented strategy can be generalised so that the exact strategies presented in Chapter 9 yield exact strategies under a more realistic model of computation.

We present the Constructive Orbit Problem *with References* (COPR), and show that polynomial time strategies for the COP under the simple model of computation of Chapter 9 do not *directly* solve the COPR. We then present a computational group theoretic approach based on the segmented strategy, which extends any strategy for solving the COP to a solution for the COPR. Our extension results in exact symmetry reduction, at the expense of polynomial time. However, experimental results, which we present in Section 11.3, demonstrate that in practice our approach is significantly more efficient than symmetry reduction by enumeration (see Section 9.2.2). We show that the COPR is polynomial-time equivalent to COP, and discuss the relationship between these problems and the computational group theoretic problem of finding the smallest image of a set under a group [120].

10.1 A Model of Computation With References

As in Chapter 9, let $I = \{1, 2, \ldots, n\}$ be the set of component identifiers for a concurrent system. Suppose that the local state of a component is comprised of two parts, its *control* state and its *reference* state.

The control state of a component is determined by the values of all local variables of that component which are *not* references to other components, e.g. a program counter or boolean flag. Without loss of generality (see Section 9.1), we can represent a local control state abstractly as an integer taken from a finite set $L_c \subset \mathbb{Z}$.

The reference state of a component is determined by the values of all local variables which *are* references to other components. For example, components in a leader election protocol may require a reference variable to (eventually) hold the identity of the leader; a user in a model of telephony may hold a reference to its current partner. Thus a reference state is a tuple in the set $L_r = (I \cup \{0\})^m$ for some $m \geq 0$. Here m is the number of references held by a component, and 0 is used as a default value (e.g. to represent that the leader is unknown). Without loss of generality we can assume that all components have exactly $m \geq 0$ reference local variables.

Thus a global state $s \in (L_c \times L_r)^n$ has the form:

$$s = (l_1, (r_{1,1}, r_{1,2}, \ldots, r_{1,m}), l_2, (r_{2,1}, r_{2,2}, \ldots, r_{2,m}), \ldots, l_n, (r_{n,1}, r_{n,2}, \ldots, r_{n,m})),$$

where $l_i \in L_c$ represents the control state of component i, and $r_{i,j} \in I \cup \{0\}$ is the value of the jth reference variable of component i ($i \in I, 1 \leq j \leq m$).

In the special case where $m = 0$, i.e. when components do not hold references to one another, L_r consists of a 0-tuple, and can thus be ignored. A state $s \in L_c^n$ then has the form $s = (l_1, l_2, \ldots, l_n)$ described in Section 9.1. We refer to models of computation where $m > 0$ and $m = 0$ as a model of computation *with* and *without* references, respectively.

10.1.1 The Constructive Orbit Problem with References

With the extended model of computation, in order to define a total ordering $\preceq$ on $S \subseteq (L_c \times L_r)^n$, we define two projection mappings, *ctrl* and *ref*, projecting a state on to its control and reference parts respectively. For a state $s = (l_1, (r_{1,1}, r_{1,2}, \ldots, r_{1,m}), l_2, (r_{2,1}, r_{2,2}, \ldots, r_{2,m}), \ldots, l_n, (r_{n,1}, r_{n,2}, \ldots, r_{n,m})), ctrl(s) = (l_1, l_2, \ldots, l_n)$ and $ref(s) = (r_{1,1}, r_{1,2}, \ldots, r_{1,m}, \ldots\ldots, r_{n,m})$.

Definition 34 *For $s, t \in S$, $s \preceq t$ if either $s = t$; $ctrl(s) < ctrl(t)$; or $ctrl(s) = ctrl(t)$ and $ref(s) < ref(t)$. Here $ref(s)$ and $ref(t)$ are compared using the usual lexicographic ordering on vectors (similarly $ctrl(s)$ and $ctrl(t)$).*

It is clear that $\preceq$ is a total ordering on states. We write $s \prec t$ if $s \preceq t$ and $s \neq t$. We now extend the COP to the model of computation with references:

Definition 35 The COP with References (COPR) *Given a group $G \leq S_n$ and a state $s \in (L_c \times L_r)^n$, find $min_{\preceq}[s]_G$, the $\preceq$-least element in the orbit of s under G.*

It is clear that the COPR is a generalisation of the COP: in the special case where $m = 0$ the COP and the COPR are identical. Since the COP is *NP*-hard (Theorem 10), the COPR is *NP*-hard by restriction. In fact, the two problems can be shown to be polynomial-time equivalent. An instance of the COP is trivially an instance of the COPR, and an instance of the COPR can be converted, in quadratic time, to an instance of the COP. The latter is achieved by replacing each component id reference $r_{i,j}$ by a vector of n binary values, which are all 0 unless $r_{i,j} = l > 0$, in which case the binary value l places from the right is 1. For example, if $n = 8$ and $r_{i,j} = 5$, the value of $r_{i,j}$ is converted to the binary sequence $\underbrace{0,0,0}_{n-5},\underbrace{1,0,0,0,0}_{5}$. The variables introduced to hold these values are modelled as components with binary valued local state. If *convert* denotes a function which performs this conversion, then placing the value 1 l places to the right ensures that, for states s and t, $s \preceq t$ iff $convert(s) \leq convert(t)$. Elements of the symmetry group G must also be transformed appropriately, so that if s is a state and $\alpha \in G$, the transformed element α' must satisfy $convert(\alpha(s)) = \alpha'(convert(s))$.

Algorithm 10 A COP strategy for S_n based on selection sort.

```
α := id
for all i ∈ [1, ..., n − 1] do
  β := id
  for all j ∈ [i + 1, ..., n] do
    if (i j)α(s) < βα(s) then
      β := (i j)
    end if
  end for
  α := βα
end for
return α
```

10.1.2 Problems with references

Recall the polynomial time strategies for the COP described in Chapter 9. Clearly the strategy based on enumeration extends immediately to a model of computation with references, if $|G|$ is polynomial in n. However, the other strategies are not immediately applicable. We show this for the COP strategy where $G = S_n$ and representatives are computed by sorting. Similar arguments can be applied for the other strategies.

The proof that the COP for $G = S_n$ can be solved by sorting a state s is based on the following lemma:

Lemma 13 *In the simple model of computation, there are no $i_1, j_1, i_2, j_2 \in I$ where $i_1 < j_1$, $i_2 < j_2$, $(i_2\ j_2)(s) < s$ and $(i_1\ j_1)(s) \geq s$, but $(i_2\ j_2)(i_1\ j_1)(s) < (i_2\ j_2)(s)$.*

However, this result does not hold in the presence of references.

Lemma 14 *Lemma 13 does not hold for the model of computation with references where the ordering $\leq$ is replaced with $\preceq$.*

Proof We prove Lemma 14 by counter-example. Suppose $n = 3$, and consider $s = (\mathbf{1}, 0, \mathbf{0}, 2, \mathbf{0}, 2)$. Take $i_1 = 2$, $j_1 = 3$, $i_2 = 1$ and $j_2 = 3$. Then we have $(i_2\ j_2)(s) = (\mathbf{0}, 2, \mathbf{0}, 2, \mathbf{1}, 0) \prec s$, $(i_1\ j_1)(s) = (\mathbf{1}, 0, \mathbf{0}, 3, \mathbf{0}, 3) \succ s$. But $(i_2\ j_2)(i_1\ j_1) = (1\ 3\ 2)$, and $(1\ 3\ 2)(s) = (\mathbf{0}, 1, \mathbf{0}, 1, \mathbf{1}, 0) \prec (i_2\ j_2)(s)$. ■

This counter-example for the case $n = 3$ can be extended to give a counter-example for any $n \geq 3$ – consider i_1, j_1, i_2 and j_2 as above, and $s = (\mathbf{1}, 0, \mathbf{0}, 2, \mathbf{0}, 2, \mathbf{0}, 0, \ldots, \mathbf{0}, 0)$.

Applying Algorithm 10 with $\leq$ replaced by $\preceq$ to $s = (\mathbf{1}, 0, \mathbf{0}, 2, \mathbf{0}, 2)$ gives the element $(1\ 3)$ which does not minimise s, whereas enumeration of S_3 gives $(1\ 3\ 2)$, which does. Thus this adaptation of Algorithm 10 does not yield an exact COPR strategy.

Suppose $G \leq S_n$ is a symmetry group and $G' \leq S_{n'}$ is the group isomorphic to G obtained by the conversion of a COPR instance to a COP instance discussed in Section 10.1.1 (here $n' \geq n$ is the number of components used in the representation of converted states). A polynomial time COP strategy for G does not in general yield a corresponding COPR strategy for G', as the action of G' on $\{1, 2, \ldots, n'\}$ may be fundamentally different to that of G on $\{1, 2, \ldots, n\}$. For example, if G is the disjoint product of subgroups H and K then G' is the *direct* product of subgroups H' and K', but it may not be the case that H' and K' act disjointly on $\{1, 2, \ldots, n'\}$.

We now show how a polynomial time exact COP strategy can be extended to an exact COPR strategy. The result is not a polynomial time strategy, but may be significantly more efficient than the enumeration strategy if G is large.

10.2 Segmentation: Extending Strategies to a Model of Computation With References

Our approach to extending a strategy for COP to one for COPR works by constructing a *partition* of I from a given state, and enumerating the *stabiliser* of this partition.

10.2.1 Segmenting a state

We define a subset of $[s]_G$ whose elements have minimal control states.

Definition 36 *Let* $small_G(s) = \{t \in [s]_G : ctrl(t) \leq ctrl(u) \ \forall\ u \in [s]_G\}$.

Clearly $min_{\preceq}[s]_G \in small_G(s)$. Given a state s, the vector $ctrl(s)$ can be viewed as a state under a model of computation without references. The following result is a consequence of this observation and Definition 36:

Lemma 15 *For* $s \in S$, $t \in small_G(s) \Leftrightarrow ctrl(t) = min_{\leq}[ctrl(s)]_G$.

For $0 \leq i \leq k$, let $s^{(i)} = \{j \in I : l_j = i\}$, i.e. the set of indices of components which have control state i in s. Define the function seg acting on states by:

$$seg(s) = \{s^{(0)}, s^{(1)}, \ldots, s^{(k)}\}.$$

Then clearly, for any state s, $seg(s)$ is a partition of I.

10.2.2 Symmetry reduction via segmentation

For a group H acting on a set X, recall the definition of the stabiliser of a partition $\mathcal{X}$ of X (Definition 9, Section 3.1.2), denoted $stab_H(\mathcal{X})$.

Algorithm 11 Extending an exact COP strategy f for a group G to an exact COPR strategy.

$\beta := f(ctrl(s))$
$H := stab_G(seg(\beta(s)))$
$\alpha = id$
for all $\delta \in H$ **do**
  if $\delta\beta(s) \prec \alpha\beta(s)$ **then**
    $\alpha := \delta$
  end if
end for
return $\alpha\beta$

Lemma 16 *If $t \in small_G(s)$ and $\alpha(t) < t$ for some $\alpha \in G$ then $\alpha \in stab_G(seg(t))$.*

Proof Since $t \in small_G(s)$ and $\alpha(t) < t$, by Definition 36 clearly $ctrl(t) = ctrl(\alpha(t))$ and $ref(t) > ref(\alpha(t))$. Since $ctrl(t) = ctrl(\alpha(t))$, $t^{(i)} = \alpha(t)^{(i)}$ for $1 \leq i \leq k$, i.e. $seg(t) = seg(\alpha(t))$. Thus α preserves $seg(t)$, i.e. $\alpha \in stab_G(seg(t))$. ■

Thus, if a state $t \in small_G(s)$ is not the smallest element in $[s]_G$ under $\preceq$ then search for a minimising element of G can be restricted to $stab_G(seg(t))$. Note that if component indices $i, j \in X \in seg(t)$, it is still necessary to consider elements of G which map i to j. Thus we cannot treat the elements of $seg(t)$ as sequences and compute their pointwise stabiliser (which would be computationally easier).

Suppose that we have an exact COP strategy f for G. Let $\beta = f(ctrl(s))$, so that $\beta(ctrl(s)) = min_{\leq}[ctrl(s)]_G$. Clearly $\beta(ctrl(s)) = ctrl(\beta(s))$, and therefore by Lemma 15, $\beta(s) \in small_G(s)$. By Lemma 16, the group $H = stab_G(seg(\beta(s)))$ can now be enumerated to find an element α such that $\alpha\beta(s) \preceq \delta\beta(s)$ for all $\delta \in H$. Thus we have proved the following:

Theorem 21 *Let $s \in S$, $G \leq Aut(\mathcal{M})$, and let f be an exact COP strategy for G. Then Algorithm 11 is an exact COPR strategy for G.*

Figure 10.1 illustrates graphically the relationship between $[s]_G$ (represented by the outer ellipse) and its subset $small_G(s)$ (represented by the inner ellipse), and the process of computing an element of G which minimises s. We illustrate the approach further with an example.

Let n, m, L_c and L_r and G be as in the example in Section 10.1.2. Let $s = (\mathbf{1}, 2, \mathbf{0}, 1, \mathbf{0}, 1, \mathbf{2}, 1)$. Then $ctrl(s) = (1, 0, 0, 2)$, and applying Algorithm 10, we find that $\beta = (1\ 3)$ satisfies $\beta(ctrl(s)) = min_{\leq}[ctrl(s)]_G$. Applying β to s gives $t = (\mathbf{0}, 3, \mathbf{0}, 3, \mathbf{1}, 2, \mathbf{2}, 3)$, and $seg(t) = \{\{1, 2\}, \{2\}, \{3\}\}$. It is easy to check that $stab_G(seg(t)) = \langle(1\ 2)\rangle$, a group of order 2, and that applying $(1\ 2)$ to t gives $min_{\preceq}[s]_G = (\mathbf{0}, 3, \mathbf{0}, 3, \mathbf{1}, 1, \mathbf{2}, 3)$. For this example, the application of 6 group ele-

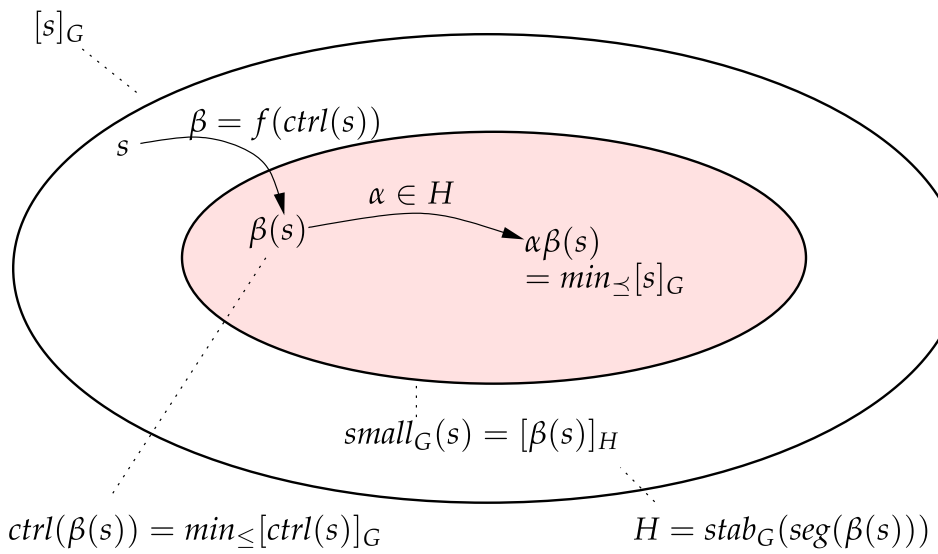


Figure 10.1: Symmetry reduction by segmentation.

ments is required by Algorithm 10, followed by enumeration of a group of order 2. Computing $min_{\preceq}[s]_G$ by basic enumeration would have required the application of all 24 elements of G to s.

10.3 Efficiency

Assuming that f can be computed in polynomial time (using strategies described in [27] and Chapter 9), the efficiency of Algorithm 11 is dominated by computation of and iteration over H.

Computing $H = stab_G(seg(s))$ is equivalent to computing the stabiliser of a set in a group. The most efficient algorithms available for computing set stabilisers involve backtrack search of the group using a base and strong generating set [19]. Typically this search can be heavily pruned using both problem-independent heuristics, and heuristics based on properties of set stabilisers. Thus, despite the fact that no polynomial time algorithm is known for computing set stabilisers, the associated overhead is not large. Furthermore, as the experimental results of Section 11.3 show, the set $\{seg(s) : s \in S\}$ of all partitions of I which must be considered during search, is often much smaller than the number of possible partitions of I.[1] Thus, re-computation of partition stabilisers can be avoided by caching partition-stabiliser pairs.

In the worst case, H may have size $|G|$ (e.g. when $|seg(s)| = 1$), and $|G|$ may be as large as $n!$ (in the case where $G = S_n$). However, if the number of distinct

1. The number of such partitions is B_n, the nth Bell number, which is defined recursively by $B_0 = 1$ and $B_n = \sum_{k=0}^{n-1} \binom{n}{k} B_k$ for $n > 0$ [149].

component control states is reasonably large, many states s will have the property that $|seg(s)| = n$, in which case $stab_G(seg(s))$ is the trivial group.

Summary

In order for our symmetry reduction techniques to be applicable to the Promela specification language, we have defined a realistic model of computation where components may hold references to one another. We have defined the COP *with references*, and shown that although the COP and COPR are polynomial-time equivalent, polynomial time strategies for solving the COP for specific groups do not, in general, directly extend to solve the COPR.

We have presented a technique for extending any COP strategy to solve the COPR by generalising the *segmented* symmetry reduction strategy used by the SymmSpin tool [14]. The extended strategy involves applying the initial strategy, followed by an enumeration process to compute the minimum state in the set of states regarded as minimal by the initial strategy. Although the extended strategy does not run in polynomial time, it is more sophisticated than basic enumeration. For many states, solving the COPR involves applying a polynomial time COP strategy, then enumerating over a small (even trivial) permutation group.

Chapter 11

TopSpin – a Computational Group Theoretic Symmetry Reduction Package for Spin

In this chapter we describe TopSpin, a symmetry reduction package which we have developed for the Spin model checker. TopSpin uses SymmExtractor (see Chapter 8) for automatic symmetry detection, and the strategies presented in Chapters 9 and 10 to exploit symmetry efficiently.

We provide an overview of TopSpin in Section 11.1, and present some examples of the source code generated by TopSpin for a selection of symmetry reduction strategies in Section 11.2. We present experimental results which demonstrate the effectiveness of our symmetry reduction techniques for a variety of Promela specifications Section 11.3, and discuss some possible extensions to TopSpin in Section 11.4.

11.1 An Overview of TopSpin

As described in Section 2.4.2 and illustrated by Figure 2.9, to check properties of a Promela specification Spin converts the specification into a C source file, `pan.c`. This *verifier* is then compiled and executed, and the state space thus generated is searched, resulting in a counter-example, an exhaustive search with an absence of counter-examples, or an incomplete search due to memory restrictions.

TopSpin follows the approach used by the SymmSpin symmetry reduction package [14] (see Section 3.9.1), where `pan.c` is generated as usual by Spin, and then converted to a new file, `sympan.c`, which includes algorithms for symmetry reduction. With TopSpin, because we allow for arbitrary system topologies, symmetry must be detected before `sympan.c` can be generated. The process involved in generating `sympan.c` is summarised in Figure 11.1, which combines the process of Figure 2.9 with automatic symmetry detection and classification.

The SymmExtractor tool (see Chapter 8) is used to extract the static channel diagram $\mathcal{SCD}(\mathcal{P})$ of the Promela specification $\mathcal{P}$, and to compute the largest valid subgroup $G \leq Aut(\mathcal{SCD}(\mathcal{P}))$ with respect to $\mathcal{P}$. The symmetry detection process is

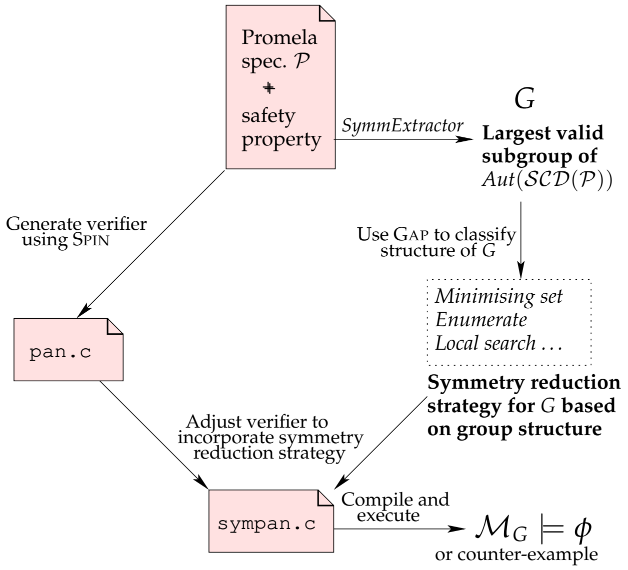


Figure 11.1: The symmetry reduction process.

illustrated in Figure 8.1, Section 8.1, and involves the construction of a syntax tree representation of $\mathcal{P}$, annotated with type information.

Based on the symmetry reduction strategy specified by the user (see below), or using the default *fast* strategy, TopSPIN generates C algorithms for symmetry reduction. The overall structure of this code is strategy-dependent, but the fine details (in particular the function which applies a permutation to a state) are specification-specific, and depend critically on the type information provided by SymmExtractor. These algorithms are merged with `pan.c` to form `sympan.c`, which can be compiled and executed as usual. TopSPIN is currently limited to the verification of safety properties, which can be expressed using assertions.

TopSPIN provides four symmetry reduction strategies: *enumeration*, *hillclimbing*, *fast*, and *segmented*. The *enumeration* and *hillclimbing* strategies use the general representative computation techniques presented in Sections 9.2.2 and 9.2.4 respectively. The *enumeration* strategy provides exact symmetry reduction, thus is memory optimal. However, it may be very slow if G is large. The *hillclimbing* strategy is approximate (it does not guarantee computation of a unique representative from each equivalence class) but computationally inexpensive. With both the *fast* and *segmented* strategies, TopSPIN analyses the structure of G using a GAP implementation of the algorithms presented in Chapter 9, and generates routines for representative computation based on this structural information. If the *fast* strategy is selected then these routines may or may not provide exact symmetry reduction, depending on whether processes in the input specification hold references to one another. If the *segmented* strategy is used then the representative computation routines are followed by a *segmentation* phase (as described in Chapter 10) which guarantees unique representatives. Note that TopSPIN implements the sound, incomplete approach of Section 9.3.1 for decomposing a group as a disjoint product.

To allow the user to manually specify symmetry (e.g. when SymmExtractor is not capable of detecting it automatically), and to allow TopSPIN to be linked with alternative automatic symmetry detection tools, TopSPIN also accepts generators for a group of process and channel automorphisms specified (in disjoint cycle form) in an input file. The resulting group can still be automatically classified if the *fast* or *segmented* strategy is used.

In Sections 9.2.1 and 9.2.2 we discussed optimisations for efficient application of permutations, and efficient enumeration of a group respectively. TopSPIN uses these optimisations by default, but they can be disabled for purposes of comparison.

11.2 Computing Representatives

A key component of the `pan.c` verifier is the `store` function. Given a single argument s (a SPIN state-vector) the `store` function determines whether s already belongs to the set of previously stored states, adding it to the set if it is not. In summary, TopSPIN adds a function `rep` to `pan.c`, and replaces each call of the form `store(s)` with a call `store(rep(s))`. If `rep` returns a unique representative of $[s]_G$ it is clear that by modifying every call to `store` in this way we ensure only a single state from each equivalence class is ever added to the state-space, resulting in optimal symmetry reduction. Alternatively, `rep` may provide sub-optimal symmetry reduction by mapping $[s]_G$ on to a small set of representatives.

We now give some examples to illustrate the C code which TopSPIN generates for the function `rep`, for a variety of symmetry reduction strategies. The examples use the standard C functions `memcpy(a,b,c)`, which copies c bytes from the memory region pointed to by b to the memory region pointed to by a, and `memcmp(a,b,c)`, which compares these memory regions, returning 0 if they are equal, a positive value if b is larger than a (viewed as a binary vector), and a negative value otherwise. The `State` type denotes a SPIN state-vector, and `State*` denotes a pointer to a state vector. Each version of the `rep` function relies on a subsidiary function `applyPermToState`. This function computes the image of a state under a given permutation. Its implementation is specification-specific, dependent on the location of *pid* and *chan* variables and *pid*-indexed arrays. The type information gathered from the input specification during symmetry detection is critical to the generation of this function. If a symmetry group is specified manually then it is necessary for TopSPIN to type-check the input specification in order to successfully generate the `applyPermToState` function. We do not give a code example for this function; for details see the TopSPIN source code (the location of which is given in Section 1.2), or examine the `sympan.c` file which TopSPIN generates for a given

```
State* rep(State* s) {
  int i0, i1, i2, i3, i4;
  State partialImages[5];

  memcpy(min, s, vsize);

  for(i4=0; i4<2; i4++) {
    memcpy(&partialImages[4],s,vsize);
    applyPermToState(&partialImages[4],coset_reps[4][i4]);

    for(i3=0; i3<3; i3++) {
      memcpy(&partialImages[3],&partialImages[4],vsize);
      applyPermToState(&partialImages[3],coset_reps[3][i3]);

      for(i2=0; i2<4; i2++) {
        memcpy(&partialImages[2],&partialImages[3],vsize);
        applyPermToState(&partialImages[2],coset_reps[2][i2]);

        for(i1=0; i1<5; i1++) {
          memcpy(&partialImages[1],&partialImages[2],vsize);
          applyPermToState(&partialImages[1],coset_reps[1][i1]);

          for(i0=0; i0<6; i0++) {
            memcpy(&partialImages[0],&partialImages[1],vsize);
            applyPermToState(&partialImages[0],coset_reps[0][i0]);

            if(memcmp(&partialImages[0],min,vsize)<0) {
              memcpy(min,&partialImages[0],vsize);
            }
          }
        }
      }
    }
  }
  return min;
}
```

Figure 11.2: Representative computation for 6 process Peterson mutual exclusion protocol via enumeration, using a stabiliser chain.

specification. For readability, we have tidied up the code examples in the following sections to some extent.

11.2.1 Enumeration

Figure 11.2 shows the `rep` function which TopSPIN generates given a specification of a 6-process version of Peterson's mutual exclusion protocol (see Section 4.3 and Appendix A.1.1). Note that TopSPIN would, by default, choose a more efficient representative computation strategy for this example, as we show in Section 11.2.2.

Recall the process of enumeration using a stabiliser chain, described in Section 9.2.2. The symmetry group for G in this example is S_6, and GAP has been used to construct a stabiliser chain for G. The chain has length six, so there are five sets of coset representatives. The coset representatives are stored using a 2-dimensional array, `coset_reps`. An array, `partial_images`, is used to store images of the state s under consecutive coset representatives. The element `partial_images[4]` is the image of s under an element of `coset_reps[4]`, and for $0 \leq i < 4$, `partial_images[`i`]` is the image of `partial_images[`$i+1$`]` under an element of `coset_reps[`i`]`. The final image of s under an element of G is stored in

```
State* rep(State* s) {
  int j;

  memcpy(min, s, vsize);
  do {
    memcpy(last_min,min,vsize);

    for(j=0; j<15; j++) {
      memcpy(tmp,min,vsize);
      applyPermToState(tmp,minimising_set[j]);
      if(memcmp(tmp,min,vsize)<0)
        memcpy(min,tmp,vsize);
    }
  } while(memcmp(min,last_min,vsize)!=0);
  return min;
}
```

Figure 11.3: Representative computation for 6 process Peterson mutual exclusion protocol, using a minimising set.

`partial_images[0]`, and is compared with the smallest state in the orbit so far, `min` (a global variable), using the C function `memcmp`.[1] If `partial_images[0]` is found to be *smaller* than `min` then the value of `min` is overwritten with this new minimum, using the `memcpy` function.

The code shown in Figure 11.2 is essentially an implementation of Algorithm 5, Section 9.2.2, for the 6-process Peterson mutual exclusion example.

11.2.2 Minimising sets

As noted above, TopSPIN would not use enumeration for the Peterson mutual exclusion example by default. Rather, a minimising set for G would be computed using the techniques described in Section 9.2.3. Figure 11.3 shows the code for `rep` which is generated in this case. The function is essentially an implementation of Algorithm 7, Section 9.2.3. The (global) variables `min`, `last_min` and `tmp` are SPIN state vectors. The minimising set is stored as an array, `minimising_set`, and the algorithm proceeds by iterating over this array and minimising the state `min`, until `min` does not change. For this example the group is S_6 and the minimising set has size 15, as predicted by the formula for minimising set size given in the proof of Theorem 17 (Section 9.2.3).

11.2.3 Hillclimbing local search

The code generated when the hillclimbing local search strategy (see Section 9.2.4) is chosen, either automatically by TopSPIN or manually for experimental purposes, is similar to that generated when a minimising set is used. Figure 11.4 shows the code generated for the `rep` function when hillclimbing local search is applied to the Peterson mutual exclusion example with six processes. The key differences between Figure 11.3 and Figure 11.4 are that a generating set `gens` for G is used, rather than a minimising set, and that on each iteration of the inner loop a permutation

1. The argument `vsize` to `memcmp` is a global variable denoting the length of the state vector.

```
State* rep(State* s) {
  int j;

  memcpy(min, s, vsize);
  do {
    memcpy(last_min,min,vsize);

    for(j=0; j<5; j++) {
      memcpy(tmp,last_min,vsize);
      applyPermToState(tmp,gens[j]);
      if(memcmp(tmp,min,vsize)<0)
        memcpy(min,tmp,vsize);
    }
  } while(memcmp(min,last_min,vsize)!=0);
  return min;
}
```

Figure 11.4: Representative computation using hillclimbing local search.

is applied to `last_min`, rather than `min`. This ensures that the inner loop computes the smallest image of `last_min` under the generators of G. If this image is smaller than `last_min` then local search continues. Otherwise this local minimum is returned as a representative.

11.2.4 Applying a composite strategy

If TopSPIN computes a decomposition for G as a disjoint or wreath product of subgroups (using the techniques of Sections 9.3 and 9.4 and the recursive classification algorithm described in Section 9.6) then the `rep` function consists of multiple sections of code, one for each factor of the product.

Consider a configuration of the resource allocator specification (see Section 4.4 and Appendix A.2) consisting of three processes with priority level 0, and four with priority level 1. This is the specification denoted '3-4' in Section 8.4. The symmetry group associated with this example decomposes a disjoint product $H_1 \bullet H_2$ where H_1 and H_2 are isomorphic to S_3 and S_4 respectively. These groups can be handled using minimising sets of size 3 and 6. Figure 11.5 shows the code for `rep` generated by TopSPIN.

11.2.5 The *segmented* strategy

If the *segmented* strategy is chosen then the function `rep` is generated as for the *fast* strategy, but the `return` statement is prefixed by a function call of the form `segment(min)`. The code which is common to the *fast* strategy corresponds to the line $\beta := f(ctrl(s))$ in Algorithm 11, Section 10.2.2. The `segment` call corresponds to the remainder of Algorithm 11. We give a high-level explanation of how the `segment` function is implemented.

Before search, a variable of a proctype is classed as *sensitive* if it has type *pid* or *chan*, or if it is an array indexed by values of type *pid*, otherwise it is classed as *insensitive*. In order to handle user-defined (possibly nested) records, and arbitrary

```
State* rep(State* s) {
  int j;

  memcpy(min, s, vsize);

  do {
    memcpy(last_min,min,vsize);
    for(j=0; j<3; j++) {
      memcpy(tmp,min,vsize);
      applyPermToState(tmp,minimising_set_1[j]);
      if(memcmp(tmp,min,vsize)<0)
        memcpy(min,tmp,vsize);
    }
  } while(memcmp(min,last_min,vsize)!=0);

  do {
    memcpy(last_min,min,vsize);
    for(j=0; j<6; j++) {
      memcpy(tmp,min,vsize);
      applyPermToState(tmp,minimising_set_2[j]);
      if(memcmp(tmp,min,vsize)<0)
        memcpy(min,tmp,vsize);
    }
  } while(memcmp(min,last_min,vsize)!=0);

  return min;
}
```

Figure 11.5: Representative computation for a resource allocator specification which has an associated disjoint product group $H_1 \bullet H_2$, using two minimising sets.

arrays, distinct fields of a record variable are regarded as separate variables. Similarly, distinct elements of an array which is *not* indexed by values of type *pid* are treated as separate variables. A field of a buffered channel is classed as sensitive or insensitive in an analogous way.

The `segment` function has a single state-vector parameter s. A partition $seg(s)$ of process identifiers and static channel names is constructed from s as follows. Process identifiers i and j are in the same partition if $proctype(i) = proctype(j)$, and the *insensitive* variables of i and j are equal at s. Buffered channels c and d are in the same partition if $signature(c) = signature(d)$, c and d have the same length at s, and the insensitive fields of c and d are equal at s. Synchronous channels have no state, and thus need not be included in the partition.

GAP is used to compute the subgroup $H = stab_G(seg(s))$, and returns a set of coset representatives for efficient enumeration of H (see Section 11.2.1). The minimum image of s under H is computed using these coset representatives using a routine similar to that shown in Figure 11.2, Section 11.2.1. As mentioned in Section 10.3, we optimise the performance of the `segment` function by caching partition-stabiliser pairs. When a partition is computed, before calling on GAP to compute the associated stabiliser, a lookup is made to a table of stabilisers, indexed by partitions. If the partition has been encountered before then there is no need to re-compute the stabiliser. Experimental results in Section 11.3 show that exploration of a large state-space may result in only a few distinct partitions.

The *segmented* strategy is the only strategy which requires communication with GAP during search. Due to the technical difficulty of calling external processes from a C program, the current implementation of TopSPIN provides a GAP function, `Verify()`, which starts the `sympan` executable as a slave process. Unfortunately, it is not possible (to our knowledge) to pass command-line arguments to `sympan` this way, so parameters such as the maximum search depth must be changed manually in `sympan.c`, which requires some expert knowledge. Communication between `sympan` and GAP takes place via a text stream, using a simple bespoke protocol.

11.3 Experimental Results

We demonstrate the effectiveness of our symmetry reduction techniques by applying TopSPIN to a selection of Promela specifications. We categorise these specifications into families in Section 11.3.1. In Section 11.3.2 we discuss the type of symmetry associated with each family, which determines the strategy chosen by TopSPIN when the *fast* option is selected. In Section 11.3.3 we discuss the experimental results.

11.3.1 Specification families and configurations

We consider each of the specification families used for experiments with SymmExtractor in Section 8.4, and use the notation introduced in Section 8.4.1 to denote configurations of these families. In order to fully illustrate TopSPIN we use two additional families of Promela specifications: an email system, and a loadbalancer which forwards requests from a pool of clients to a pool of servers in a fair manner.

The email example is adapted from [21]. A configuration of the system consists of *n client* processes, which communicate by sending messages to a *mailer* process via a *network* channel component. The client components are instantiations of the same parameterised process and thus behave identically, so there is full symmetry between clients. Components in a Promela specification of the system use reference variables to keep track of the sender and recipient of a given message. An *email* configuration with n clients is denoted n.

Components of a configuration in the *loadbalancer* family are a set of *m server* and *n client* processes with associated communication channels, and a *loadbalancer* process (with a dedicated input channel). The *load* of a server is the number of messages queued on its input channel. Client processes send requests to the loadbalancer, and if any of the *server* channels are not full, the loadbalancer forwards a request nondeterministically to one of the least loaded *server* queues. Each request contains a reference to the input channel of its associated client process, and the server designated by the loadbalancer uses this channel to service the request. A *loadbalancer* configuration with *m server* and *n client* processes is denoted m-n.

For purposes of comparison, we have slightly modified some specifications in order to be able to verify reasonably large examples *without* symmetry reduction.

11.3.2 Symmetry groups associated with each family

The *simple mutex*, *Peterson*, *Peterson without atomicity* and *email* specifications all exhibit groups which are isomorphic to S_n, where n is the configuration size. For the mutual exclusion examples, the group actually is S_n – there is full symmetry between the competing processes, and no channels. The symmetry group associated with an *email n* specification consists of all permutations of the *n client* processes which simultaneously permute their corresponding input channels. For configurations in each of these families, TopSPIN automatically classifies the associated symmetry group, and computes a minimising set (see Section 9.2.3). The resulting code for representative computation is similar to the example given in Figure 11.3, Section 11.2.2.

Given a *resource allocator* configuration denoted a_0-a_1-...-a_{k-1}, the corresponding symmetry group is a disjoint product $H_1 \bullet H_2 \bullet \cdots \bullet H_k$, where $H_i \cong S_{a_{i-1}}$ for each $0 \leq i \leq k$. The group H_i consists of all permutations of *client* processes with priority level a_{i-1} which simultaneously permute the *client* communication channels. TopSPIN automatically computes this disjoint product decomposition and identifies a minimising set for each factor of the product. Code for representative computation is produced in a similar manner to the example given in Figure 11.5, Section 11.2.4.

The symmetry group associated with an *m-n loadbalancer* configuration is also a disjoint product. The group has the form $H_1 \bullet H_2$, where $H_1 \cong S_m$ and $H_2 \cong S_n$ permute the *server* and *client* components respectively (simultaneously permuting their corresponding input channels). Once again, TopSPIN outputs code for representative computation by computing a minimising set for each factor of this product.

We consider *three-tiered architecture* specifications which are balanced – that is, there are *m server* components, and a block of *n client* components connected to each *server* component (for some $m, n > 0$). Given a configuration $\underbrace{n\text{-}n\text{-}\ldots\text{-}n}_{m}$ in the three-tiered family, the associated symmetry group is an inner wreath product $H \wr K$, where $H \cong S_n$ and $K \cong S_m$. The wreath product contains m copies of H, each of which permutes *client* processes and channels within one of the blocks. The group K permutes the *m server* components. An element of K which maps *server i* to *server j* also maps the block of clients connected to *server i* to the block of clients connected to *server j*. TopSPIN uses the techniques of Section 9.4 to automatically compute this wreath product decomposition, and then computes distinct minimising sets for each copy of H and a minimising set for K. Code for the resulting composite strategy is again similar to the example given in Figure 11.5, Section 11.2.4.

For configurations in the *hypercube* family, we manually specify generators for a group of symmetries. This is due to the inability of SymmExtractor to automatically detect the complete group of symmetries for these examples, as discussed in Section 8.4.2. The hypercube examples exhibit fairly large groups, which *cannot* be decomposed as disjoint/wreath product, and for which no minimising set can be found by our methods. Using the *fast* strategy, TopSPIN selects hillclimbing local search (see Section 9.2.4) to handle this type of symmetry, and outputs code for representation similar to the example given in Figure 11.4, Section 11.2.3.

11.3.3 Results and discussion

Figure 11.6 contains experimental results for various configurations of the above families. For each configuration, we give the number of model states without symmetry reduction (**orig**), with memory optimal symmetry reduction using the *enumeration* strategy (**red**), and with symmetry reduction using the *fast* strategy (**fast**). When the number of model states is the same using the *enumeration* and *fast* strategies, '=' appears in the **fast** column. State-space sizes which are larger than 10^6 are given to the nearest hundred-thousand, with the exception of the *Peterson without atomicity* 4 configuration (as discussed below). The use of state compression (see Section 2.6.2) is indicated by the number of states in italics. This option was selected for three configurations to allow verification without symmetry reduction more efficiently, with an associated time overhead.

Verification times (in seconds) are given for the *enumeration* strategy with and without the group-theoretic optimisations of Section 9.2.2 (**basic** and **enum** respectively), for the *fast* (**fast**) option, as well as for the case where symmetry reduction is not applied (**orig**). The size of the symmetry group ($|G|$) and the time, in seconds, taken by GAP to classify this group (**classify time**) are also given.

Verification attempts which exceed available resources, or do not terminate within 15 hours, are indicated by '-'. All experiments are performed on a PC with a 2.4GHz Intel Xeon processor, 3Gb of available main memory, running SPIN version 4.2.3.[2] For the *email 7* and *loadbalancer* 3-7, 4-6 and 5-7 configurations, the size of the reduced state-space was computed using the *segmented* strategy, for which timing information is given in Figure 11.6 and discussed below.

For all specification families except the *hypercube* family, the application of symmetry reduction allows the verification of larger configurations – even using state compression, memory requirements were quickly exceeded when symmetry reduction was not applied. In all cases, the enumeration strategy without optimisations is significantly slower than the optimised enumeration strategy, which is in turn slower than the strategy chosen by TopSPIN.

Processes in the *simple mutex* configurations do not hold references to one another, so the *fast* strategy provides exact symmetry reduction, as expected. In contrast, the difference between the *fast* and *enumeration* strategies is especially marked for the *simple mutex* 10 configuration, where the symmetry group is much

2. An archive of the Promela specifications used for the experiments is available online (see Section 1.2).

Config	states orig	time orig	$\|G\|$	states red	time basic	time enum	classify time	states fast	time fast
simple mutex									
5	113	0.09	120	12	0.10	0.10	0.19	=	0.06
10	6145	0.11	3.6×10^6	22	-	1088	0.14	=	0.05
15	278529	5	1.3×10^{12}	-	-	-	0.17	32	0.08
20	1.2×10^7	561	2.4×10^{18}	-	-	-	0.23	42	0.12
Peterson									
3	2636	0.35	6	494	0.08	0.02	0.13	=	0.35
4	60577	0.60	24	3106	0.04	0.20	0.13	=	0.41
5	1.56×10^6	11	120	17321	16	7	0.13	=	1
6	4.48×10^7	2666	720	89850	722	304	0.13	=	7
7	-	-	5040	442481	30458	13885	0.13	=	56
8	-	-	40320	-	-	-	0.14	2.09×10^6	412
9	-	-	362880	-	-	-	0.14	9.62×10^6	3034
Peterson without atomicity									
2	291	0.36	2	148	0.34	0.34	0.33	=	0.34
3	75356	1	6	12706	0.83	0.68	0.14	=	0.62
4	-	-	24	3.6332×10^6	3426	972	0.13	3.6335×10^6	427
resource allocator									
3-3	16768	0.2	36	1501	0.9	0.3	0.17	=	0.1
4-4	199018	2	576	3826	57	19	0.19	=	0.4
5-5	2.2×10^6	42	14400	8212	4358	1234	0.16	=	2
4-4-4	2.39×10^7	1587	13824	84377	-	12029	0.19	=	17
5-5-5	-	-	1728000	-	-	-	0.17	254091	115
three-tiered architecture									
3-3	103105	5	72	2656	7	4	0.41	=	2
4-4	1.1×10^6	37	1152	5012	276	108	0.44	=	2
3-3-3	2.54×10^7	4156	1296	50396	4228	1689	0.41	=	19
4-4-4	-	-	82944	-	-	-	0.51	130348	104
email									
3	23256	0.1	6	3902	0.9	0.8	0.16	3908	0.2
4	852641	9	24	36255	13	6	0.16	38560	2
5	$\mathit{3.04{\times}10^7}$	3576	120	265315	679	253	0.13	315323	40
6	-	-	720	1.7×10^6	-	13523	0.14	2.3×10^6	576
7	-	-	5040	9.3×10^6	-	-	0.14	1.53×10^7	6573
loadbalancer									
2-6	$\mathit{2.37{\times}10^7}$	1585	1440	23474	656	265	0.32	31066	5
2-7	-	-	10080	44137	10314	4376	0.32	61245	16
3-6	-	-	4320	125126	13468	5024	0.25	256204	57
3-7	-	-	30240	293657	-	-	0.28	685167	213
4-6	-	-	17280	527548	-	-	0.29	1.7×10^6	487
4-7	-	-	120960	1.2×10^6	-	-	0.30	3.7×10^6	1583
hypercube									
3d	13181	0.3	48	308	0.6	0.3	0.07	468	0.2
4d	380537	18	384	1240	58	34	0.07	6986	13
5d	$\mathit{9.6{\times}10^6}$	2965	3840	3907	7442	5241	0.10	90442	946

Figure 11.6: Experimental results for symmetry reduction with TopSpin.

larger than even the unreduced state-space. Configurations in all the other families consist of processes which *do* hold references to one another, in which case the *fast* strategy does not promise exact symmetry reduction even when the associated symmetry group can be classified appropriately (see Section 10.1.2). However, for the *Peterson*, *resource allocator* and *three-tiered architecture* specifications, exact symmetry reduction is obtained using the *exact* strategy (at least for the configurations to which we could feasibly apply the *enumeration* strategy).

Exact symmetry reduction using the *fast* strategy is not obtained for the *email* or *loadbalancer* configurations, or for the *Peterson without atomicity* 4 configuration. Nevertheless, a large factor of reduction is gained by exploiting symmetry in this way, and verification is fast. The difference in model sizes using the *fast* and *enumeration* strategies for the *Peterson without atomicity* configuration is small.

As discussed above, TopSpin uses hillclimbing local search when the *fast* strategy is applied to the *hypercube* specifications. This requires storage of more states than the enumeration strategy, but is considerably faster and still results in a greatly reduced state-space.

Figure 11.7 shows the time taken for symmetry reduction using the *segmented* strategy, applied to the *email* and *loadbalancer* configurations. The (reduced) model sizes are given in the **states red** column of Figure 11.6. The *email* and *loadbalancer* configurations are suitable for the *segmented* strategy since the *fast* strategy does not provide optimal symmetry reduction, and the symmetry group associated with each configuration can be classified using the techniques of Chapter 9. Using the polynomial time COP strategy obtained via this classification, together with the techniques of Chapter 10, we obtain exact symmetry reduction more efficiently than via enumeration. Indeed, for larger configurations in each family, the *segmented* strategy allows us to feasibly construct a memory-optimal reduced state-space, which was not possible using straightforward enumeration. Recall that the *segmented* strategy works by applying a polynomial time COP strategy to a state, computing a partition associated with the resulting state, and enumerating the stabiliser of this partition to find the unique representative. Since many different states may exhibit the same partition, TopSpin stores a table of stabiliser subgroups, indexed by partitions, as discussed in Section 11.2.5. Figure 11.7 records the number of distinct partitions which were computed for each *email* and *loadbalancer* configuration. Note that for all configurations this number is much smaller than the number of reduced model states.

11.4 Extending TopSpin

The main limiting feature of TopSpin is that it does not allow symmetry-reduced verification of *LTL* properties. The symmetry group derived by SymmExtractor is,

Configuration	time segmented	no. partitions
email		
3	0.2	5
4	4	7
5	71	9
6	1600	11
7	50970	13
loadbalancer		
2-6	28	94
2-7	266	259
3-6	271	330
3-7	2722	451
4-6	2378	884
4-7	29779	1296

Figure 11.7: Results for the *segmented* strategy applied to *email* and *loadbalancer* configurations.

by construction, an invariance group for a temporal property embedded in a specification as a *never* claim. This is a result of the fact that the *never* claim is just a Promela process. If α is a static channel diagram automorphism under which a given property ϕ is *not* invariant then α will not preserve the structure of the never claim for ϕ, and so will be (correctly) judged as invalid by SymmExtractor.

A solution to the problem of combining symmetry reduction with *LTL* model checking in SPIN is presented in [13]. The next step in the development of TopSPIN is to implement this solution for the general kinds of symmetry supported by TopSPIN. This will involve adapting the nested depth-first search algorithm which SPIN uses to check *LTL* properties, so that only representative states are considered.

In Section 11.2.5 we explained that use of the segmented strategy requires communication between `sympan` and GAP during search, and currently relies on GAP being the *master* process, starting `sympan` as a slave process. It should be possible to change this so that `sympan` starts GAP, which would be more natural, and would avoid the current problem of passing command-line arguments to `sympan`.

There is some overhead associated with passing data between GAP and `sympan` using a text stream during a verification run. We could remove this overhead by implementing C versions of the small number of GAP functions which are used by the *segmented* strategy, in particular the algorithms associated with computing setwise stabilisers. However, the GAP implementation has been refined over a number of years by experts in computational group theory, and reportedly includes many optimisations (some based on randomisation) which are not documented in the literature. For this reason, it is likely that any savings in communication overhead would be lost to a reduction in efficiency. As a proof-of-concept, we implemented the *Schreier-Sims* algorithm (a fundamental algorithm on which most techniques for computing with permutation groups are based) in C, from a description given in [19]. While our implementation produced correct results, it was significantly slower than GAP.

The *segmented* strategy introduces the challenge of managing communication between `sympan` and GAP via a simple protocol. It would be interesting to describe the control aspects of this protocol using Promela, and improve our confidence in TopSPIN by eliminating any potential deadlocks in the protocol which SPIN may find.

Summary

We have described TopSPIN, a computational group-theoretic symmetry reduction package for the SPIN model checker. TopSPIN provides automatic symmetry detection using SymmExtractor, and also allows the user to manually specify symmetry. For efficient symmetry reduction (and to allow experimental comparison with naïve approaches), TopSPIN provides a variety of strategies for representative computation, based on the techniques of Chapters 9 and 10.

We have given an overview of the tool, and discussed each of the symmetry reduction strategies in some detail, providing examples of the C code which TopSPIN generates for representative computation. We have provided experimental results for a variety of Promela specifications which illustrate the practical effectiveness of our computational group-theoretic symmetry reduction methods; both over verification without symmetry, and symmetry-reduced verification using basic enumeration. In addition, we have discussed some directions for future development of TopSPIN, the most important of which is to provide support for *LTL* model checking.

Chapter 12

Conclusions and Open Problems

The original contribution of this thesis can be divided into two parts: techniques for automatic symmetry detection in model checking, and methods for efficiently exploiting arbitrary symmetry groups in explicit-state model checking. The goal of these methods is to combat the state-space explosion problem, which limits the application of model checking to relatively small systems. We have presented in-depth theoretical results in each area, and backed up our theory with robust software tools and convincing experimental results.

Having provided an overview of model checking and a detailed survey of symmetry reduction techniques in Chapters 2 and 3 respectively, we illustrated some problems with existing symmetry detection and reduction techniques in Chapter 4 via a selection of example Promela specifications. In order to analyse symmetry in the models associated with these specifications we introduced the SPIN-to-GRAPE tool.

We identified two major problems with existing techniques for identifying symmetry, namely the necessity for the user to annotate a specification with symmetry-related keywords (or to use an appropriately restricted specification language), and the limitation of being able to identify only full symmetry groups. To overcome these restrictions we proposed a method for automatic symmetry detection based on *static channel diagram* analysis, in Chapter 7. This method was motivated by a correspondence between Kripke structure automorphisms and *channel diagram* automorphisms, which we investigated for specific examples in Chapter 5. To present our results rigourously without obscuring them in the complexity of Promela we introduced a smaller language, Promela-Lite, in Chapter 6. We have used the techniques of Chapter 7 to develop SymmExtractor, an automatic symmetry detection tool for Promela, described in Chapter 8. Experimental results show that SymmExtractor is mostly efficient, using group-theoretic optimisations to deal with certain difficult input specifications. We have assessed the usability of SymmExtractor by applying the tool to a set of example specifications written as solutions to a student assessed exercise. The study reveals some ways in which the tool could be improved, and provides a case study in formal methods evaluation.

Our symmetry detection methods allow identification of groups which are more complex than full symmetry groups. This leads to the *constructive orbit problem* (COP), which involves computing orbit representatives with respect to *arbitrary* symmetry groups. In Chapter 9 we presented an optimised method for enumerating small groups, and a generalisation of techniques for dealing with full symmetry groups by sorting, based on *minimising sets*. The minimising sets approach allows us to automatically and efficiently handle a large class of commonly occurring groups which are isomorphic to symmetric groups. We have extended the application of techniques for handling disjoint/wreath product groups by presenting algorithms to automatically determine disjoint/wreath product decompositions for arbitrary groups. To deal with large groups which cannot be classified using minimising sets or decomposed as products we have proposed an approximate technique for computing representatives based on hillclimbing local search. The results of Chapter 9 are based on a simple model of computation where components do not hold references to one another. For many practical systems this is not the case. In Chapter 10 we introduced the constructive orbit problem *with references*, and showed that any algorithm for solving the COP can be extended to solve the COPR. However, this extension comes at the expense of polynomial time complexity. In Chapter 11 we presented TopSPIN, a symmetry reduction package for the SPIN model checker which uses SymmExtractor for automatic symmetry detection, and provides a variety of symmetry reduction strategies based on techniques from Chapters 9 and 10. Using several families of Promela specifications, we have presented experimental results which show the effectiveness of our techniques.

Throughout the thesis we have suggested improvements to each of our software tools, and have identified a number of areas for further theoretical investigation. We now summarise these implementation and research issues in the hope that they may lead to further research and development on symmetry reduction techniques for model checking.

12.1 Outstanding Implementation Issues

While we have emphasised the importance of implementing our ideas, there are several features and optimisations which we have not had time to incorporate in our tool set.

In Sections 7.6.1, 7.6.2 and 7.6.3 we proposed straightforward extensions to our symmetry detection techniques for: allowing certain relational operations with *pid* arguments; supporting arithmetic expressions which involve *pid* variables, and capturing symmetry between global variables respectively. The user study of Section 8.5 showed that these extensions (particularly support for global variable symmetry and arithmetic expressions over *pid* variables) would improve the usability of SymmExtractor. The additional functionality should be relatively straightforward to implement.

As discussed in Section 11.4, our symmetry reduction package TopSPIN does not currently support the verification of *LTL* properties. This is not a research issue: the problem of combining symmetry reduction with *LTL* verification has already been investigated [13]. Nevertheless, for TopSPIN to be of interest to the SPIN community as a whole, facilities for *LTL* model checking under symmetry should be implemented.

The *segmented* strategy (see Section 11.2.5) could also be improved. Currently if this strategy is chosen it is necessary to launch the `sympan` executable from within the GAP system. This is somewhat counter-intuitive, and poses problems with passing command-line arguments (e.g. to set the maximum search depth) to `sympan`. The performance of this strategy could potentially be improved by linking `sympan` with compiled GAP code, rather than requiring `sympan` to communicate with GAP using a text stream during a model checking run.

12.2 Research Problems Arising from the Thesis

In Section 7.5 we observed that although the symmetry detection techniques of Chapter 7 are motivated by the concept of a static channel diagram – in turn motivated by the *channel diagram* concept, presented in Chapter 5 and inspired by [155] – there is nothing fundamentally important about the static channel diagram definition. By introducing the structure $\Psi(\mathcal{P})$, we showed that the main results of the chapter (contained in Sections 7.3 and 7.4) hold when $Aut(\mathcal{SCD}(\mathcal{P}))$ is replaced with any subgroup G of $Aut(\Psi(\mathcal{P}))$. In defence of static channel diagrams, we showed that they do provide an upper bound for the largest valid subgroup of $Aut(\Psi(\mathcal{P}))$ (see Theorem 15, Section 7.5). An interesting question for future research is whether there exists a structure $\Gamma(\mathcal{P})$ (say) which can be extracted from $\mathcal{P}$ in polynomial time, such that $Aut(\Gamma(\mathcal{P}))$ is exactly the largest valid subgroup of $Aut(\mathcal{SCD}(\mathcal{P}))$. Such a structure would eliminate the need for Algorithm 4 in Section 7.4, which computes the largest valid subgroup of $Aut(\mathcal{SCD}(\mathcal{P}))$. Experimental results with SymmExtractor (see Section 8.4) show that this algorithm is the bottleneck for our automatic symmetry detection methods. Although the *random conjugates* optimisation described in Section 8.3.3 can, in some cases, reduce this bottleneck, the desired structure $\Gamma(\mathcal{P})$ would remove it completely.

We showed in Section 7.6.4 (using a contrived example) that the notion of validity used by SymmExtractor could potentially be relaxed. The current test for deciding whether a static channel diagram automorphism is valid for an input specification is somewhat conservative, but is correspondingly efficient. While we have found that our notion of validity is usually acceptable in practice, applying SymmExtractor to the hypercube example (see Sections 4.6 and 8.4.2) identified a situation where the group of automorphisms computed by SymmExtractor is significantly smaller than the group of automorphisms computed using SPIN-to-GRAPE. This practical example suggests that a less restrictive notion of validity is worth investigation.

The main research challenge identified by the user study of Section 8.5 is to find techniques to automatically determine the relationship between numeric identifiers passed as parameters to processes by the user, and the run-time `_pid` values which SPIN assigns to processes. This is a Promela-specific issue, arising due to the need to index into arrays using process identifiers, but an elegant solution would greatly improve the practical usability of SymmExtractor.

In Section 9.3 we presented two approaches to computing disjoint product decompositions for arbitrary permutation groups, motivated by the fact that, given such a decomposition, we can compute equivalence class representatives under the whole group by applying the factors of this decomposition separately. The sound and incomplete approach described in Section 9.3.1 works extremely well for groups which have been computed automatically using a graph automorphism program, so this approach is implemented in TopSPIN. We illustrated the fact that the approach is not complete using a simple example. From a group-theoretic perspective, it is desirable to have a sound and complete solution to this problem. We presented one such solution in Section 9.3.2, but showed that, in the worst case, it has exponential time complexity. An area for future research would be to further analyse the complexity of this problem, aiming to find a polynomial time algorithm. Since the sound, incomplete approach used by TopSPIN in solving this problem works well in practice, this open problem is of more interest to the computational group theory community than to the model checking community. Similarly, a topic for future research includes determining the exact complexity of the wreath product decomposition problem which we investigated in Section 9.4.

We provided counter-examples in Section 9.5 showing that the compositional approach to computing orbit representatives with respect to disjoint/wreath products does *not* directly extend to direct/semi-direct products. As noted in Section 9.5 these counter-examples do not mean that the structure of direct/semi-direct product groups cannot be exploited in some other way to compute representatives efficiently, and this would be a potential direction for future work. The family of hypercube automorphism groups provide motivation: the group $K_n \rtimes S_n$ is a semi-direct product, and K_n in turn is a direct product of n groups of order 2.

A recent approach to symmetry breaking in constraint programming requires a solution to a problem related to COP [104]. During search, symmetry breaking is performed by backtracking when the partial assignment of variables at a given node is determined *not* to be lexicographically least in its orbit under a symmetry group G. The approach relies on a variant of an algorithm for finding the smallest image of a set under a permutation group [120]. This problem can be shown to be polynomial time equivalent to COP, and the *smallest image* algorithm can be used to solve arbitrary COP instances. Though the algorithm is not polynomial time, it exploits the structure of G in order to perform better than basic enumeration. The smallest image algorithm is general, and does not rely on specific features like minimising sets or disjoint decompositions as ours does. Therefore we expect our techniques will be more efficient for these special cases (and indeed preliminary experiments using GAP confirm this). Nevertheless, a potentially beneficial area for future work would be to replace basic enumeration with the *smallest image* algorithm. In particular the approach to symmetry reduction using *segmentation*, presented in Chapter 10 to deal with systems where components hold references to one another, relies on an enumeration phase during representative computation. It may be possible to significantly speed up this technique by using the *smallest image* algorithm instead of this enumeration step.

12.3 The Future

We conclude with three open problems which have proved to be beyond the scope of this thesis, but which we believe are important to the applicability and effectiveness of symmetry reduction techniques for practical model checking. These are: exploiting *partial* symmetries, *over-exploiting* symmetry, and using parallel processing technology for efficient representative computation.

Partial symmetry reduction

The importance of techniques for the exploitation of partial symmetry (in the context of hardware verification) is neatly summarised in [51]:

> *In conversations we have had with industrial hardware engineers, it comes out that while symmetry reduction is often applicable due to the presence of many similar subcomponents, there are also many instances where it is not – quite – applicable. That is, the systems are not genuinely symmetric but "approximately" symmetric, for example, because of one different component or slight differences among all components. This limits the scope of utility of symmetry reduction techniques.*

Our experience concurs with this statement. Although we have shown the effectiveness of our symmetry reduction techniques using a number of convincing examples, we have had to eliminate many more promising-looking case-studies which turned out to be almost, but not quite, symmetric.

We surveyed several approaches to handling partial symmetry in Section 3.7. The main drawback of the notion of *virtual* symmetry presented in [52] is the problem of deriving, at the specification level, a group of virtual symmetries for a Kripke structure. In addition, virtual symmetry reduction techniques still require the property under consideration to be invariant under the group of virtual symmetries. Methods for exploiting partial symmetry using guarded annotated quotient structures [162] are potentially more promising, being able to handle asymmetric properties as well as asymmetric models. Still, there is little indication of how general partial symmetries can be detected at the source level. General, efficient techniques for automatic partial symmetry detection would greatly increase the applicability of symmetry reduction in model checking.

Over-exploiting symmetry

When model checking a very large state-space, it may be acceptable to use an *unsound* reduction technique which efficiently covers a large portion of reachable states, but does not provide 100% coverage. This is exemplified by the *supertrace* method provided by SPIN, which reduces the storage requirement for a state to a single bit at the expense of complete verification [87]. This kind of reduction technique is useful when we are interested in finding errors in a system, rather than proving absence of errors.

In a symmetry reduction context, it may be possible to provide more efficient verification either by exploiting a *super*-group of Kripke structure automorphisms, or by computing representatives in such a way that several orbits are represented by a single state. Suppose that we have a group $G \leq Aut(\mathcal{M})$, but cannot find an efficient reduction strategy for G. If we can find a group G', with an efficient reduction strategy, such that $G \subset G'$, then performing symmetry reduction with respect to G' will give *at least* the factor of reduction obtained using G. However, if $Aut(\mathcal{M}) \subset G'$ this approach exploits more symmetry than actually exists in the model. Nevertheless, the approach may quickly discover counter-examples to the property being checked. This is valuable if unreduced verification exhausts available resources before finding a counter-example, but sound symmetry reduction is too time-consuming to be feasible. The idea of using a representative computation function which maps several states to the same representative follows a similar philosophy.

Parallel symmetry reduction

If we want to achieve sound, complete symmetry reduction under a model of computation with references then, with current techniques, we may have to resort to enumeration of large groups. Recall that the *segmented* approach to representative computation, presented in Chapter 10, improves basic enumeration by exploiting symmetry group structure, but still requires enumeration of partition stabilisers.

Given a group G and state s, computing $min[s]_G$ by enumerating G is inherently parallelisable. If we have n processing units then we can split G into n equally-sized disjoint subsets $X_1, X_2, \ldots, X_n$. Processing unit i can be used to independently compute $s_i = min\{\alpha(s) : \alpha \in X_i\}$ $(1 \leq i \leq n)$, thus the s_i can be computed in parallel. It is clear that $min[s]_G = min\{s_1, s_2, \ldots, s_n\}$. It seems natural to extend the *enumeration* and *segmented* strategies in TopSpin to run on multiple processing units, using a parallel programming system for C such as *Sieve* [119, 145]. Further research effort would be required to improve more sophisticated representative computation algorithms via parallel technology. Note that parallel approaches to model checking (see e.g. [144]) do not remove the need for symmetry reduction techniques. Distributing a task over a number of processing units promises, in the best case, a linear reduction in verification time and a linear increase in available memory. On the other hand, symmetry reduction using a large group may offer an exponential state-space reduction, so it is sensible to utilise parallel technology for efficient symmetry reduction.

12.4 Summary

We have summarised the results of the thesis, and outlined areas for further research and development of the thesis topics. In addition, we have proposed three areas for future research into symmetry reduction for model checking.

The techniques we have developed in this work are useful for the verification of genuinely symmetric systems with large state-spaces, using standard computing platforms. We hope that the next generation of symmetry reduction techniques will be able to use parallel processing technology to over-exploit partially symmetric systems with *very* large state-spaces.

Appendix A

Example Specifications

This appendix is comprised of Promela and SMC specifications, together with system description files for SymmSpin specifications, used as examples throughout the thesis. These examples are also available online (see Section 1.2).

A.1 Peterson's Mutual Exclusion Protocol

We give various specifications of Peterson's mutual exclusion protocol. See Section 4.3 for a discussion of these specifications.

A.1.1 SymmSpin specification

System description

This description indicates to SymmSpin where scalarset types appear in the Promela specification below.

```
const N 3

scalar PID[N]

proctype :system: {
   bytes flag[PID];
   PID turn[byte];
}

proctype user[PID] {
   PID i;
   PID j;
}
```

Promela specification

```
#define N 3
#define PID byte

byte flag[N];
PID turn[N];
byte inCR = 0;
```

```
proctype user (PID i) {

   PID j = N;
   byte k;
   bool ok;

   do :: k = 1;
         do :: k < N ->
               flag[i] = k;
               turn[k] = i;
again:         atomic {
                  ok = true;
                  j = 0;
                  do :: j < N ->
                        if :: j != i ->
                              ok = ok && (flag[j] < k)
                           :: else -> skip
                        fi;
                        j++
                     :: else -> break
                  od;

                  if :: ok || turn[k] != i
                     :: else -> j = N; goto again
                  fi;
                  j = N;
               };
               k++
            :: else -> break
         od;

         atomic { inCR++;  assert(inCR == 1) };  inCR--;
         flag[i] = 0;
   od;
}

/* initialize flags and start the processes */

init {
   atomic{
      byte i = 0;
      do :: i < N -> flag[i] = 0;
            turn[i] = N; run user(i);
            i++
         :: else break
      od;
   }
}
```

A.1.2 Simpler, equivalent Promela specification

```
byte flag[4] = 0;
pid turn[3] = 4;
byte inCR = 0

proctype user () {
```

```
   byte k;
   bool ok;

   do :: k = 1;
         do :: k < 3 ->
               flag[_pid] = k;
               turn[k] = _pid;
again:         atomic {
                  ok = ((_pid==1)||(_pid!=1 && flag[1]<k))&&
                       ((_pid==2)||(_pid!=2 && flag[2]<k))&&
                       ((_pid==3)||(_pid!=3 && flag[3]<k));

                  if :: ok || turn[k] != _pid
                     :: else -> goto again
                  fi
               };
               k++
            :: else -> break
         od;

         atomic { inCR++; assert(inCR == 1) };  inCR--;
         flag[_pid] = 0;
   od;
}

/* start the processes */

init {
   atomic{
      run user();
      run user();
      run user();
   }
}
```

A.1.3 SMC specification

```
Program

Module process = 5;

flag[process]=0;
k[process]=0;
pc[process]=1;
inCR[process]=0;
turn1[process]=0;
turn2[process]=0;
turn3[process]=0;

p of process;

p: {

  pc[p]==1 -> k[p]=1, pc[p]=2;
  pc[p]==2 & k[p] < 3 -> pc[p]=3;
  pc[p]==3 -> flag[p]=k[p], pc[p]=4;
  pc[p]==4 & k[p]==1 -> ALL(q of process: turn1[q]=0),
```

```
                           turn1[p]=1, pc[p]=5;
  pc[p]==4 & k[p]==2 -> ALL(q of process: turn2[q]=0),
                           turn2[p]=1, pc[p]=5;
  pc[p]==4 & k[p]==3 -> ALL(q of process: turn3[q]=0),
                           turn3[p]=1, pc[p]=5;
  pc[p]==5 & k[p]==1 & (ALL(q of process: p==q | (p!=q & flag[q]<k[p])) |
                           turn1[p]==0) -> pc[p]=6;
  pc[p]==5 & k[p]==2 & (ALL(q of process: p==q | (p!=q & flag[q]<k[p])) |
                           turn2[p]==0) -> pc[p]=6;
  pc[p]==5 & k[p]==3 & (ALL(q of process: p==q | (p!=q & flag[q]<k[p])) |
                           turn3[p]==0) -> pc[p]=6;
  pc[p]==5 & k[p]==1 & (!(ALL(q of process: p==q | (p!=q & flag[q]<k[p]))
                           | turn1[p]==0)) -> pc[p]=5;
  pc[p]==5 & k[p]==2 & (!(ALL(q of process: p==q | (p!=q & flag[q]<k[p]))
                           | turn2[p]==0)) -> pc[p]=5;
  pc[p]==5 & k[p]==3 & (!(ALL(q of process: p==q | (p!=q & flag[q]<k[p]))
                           | turn3[p]==0)) -> pc[p]=5;
  pc[p]==6 -> k[p]=k[p]+1, pc[p]=2;
  pc[p]==2 & (!(k[p]<3)) -> pc[p]=7;
  pc[p]==7 -> ALL(q of process: inCR[q]=inCR[q] + 1), pc[p]=8;
  pc[p]==8 -> ALL(q of process: inCR[q]=inCR[q] - 1), pc[p]=9;
  pc[p]==9 -> flag[p]=0, pc[p]=1;

}
```

A.1.4 More realistic Promela specification

```
byte flag[4] = 0;
pid turn[3] = 0;
byte inCR = 0

proctype user() {

   byte k;
   bool checked[4] = false;
   bool ok = false;

   do :: k = 1;
         do :: k < 3 ->
         flag[_pid] = k;
               turn[k] = _pid;
again:         atomic {
                  ok = true; checked[_pid]=true
               };
               do :: (!ok || checked[1]&&checked[2]&&checked[3]) ->
                     atomic {
                        do :: checked[1] -> checked[1] = false;
                           :: checked[2] -> checked[2] = false;
                           :: checked[3] -> checked[3] = false;
                           :: else -> break;
                        od;
                        break
                     }
                  :: d_step {
                        !checked[1] -> ok = ok && flag[1]<k;
                                       checked[1]=true
                     }
```

```
                :: d_step {
                      !checked[2] -> ok = ok && flag[2]<k;
                                     checked[2]=true
                   }
                :: d_step {
                      !checked[3] -> ok = ok && flag[3]<k;
                                     checked[3]=true
                   }
             od;
             if :: atomic { ok || turn[k] != _pid -> ok = false }
                :: atomic { else -> ok = false; goto again }
             fi;
             k++
          :: else -> break
       od;

       atomic { inCR++;  assert(inCR == 1) };  inCR--;
       flag[_pid] = 0;
   od;
}

/* start the processes */

init {
   atomic{
      run user();
      run user();
      run user();
   }
}
```

A.2 Resource Allocator

This section includes Promela and SMC specifications of a resource allocator system, which is described in Section 4.4.

A.2.1 Promela specification

```
mtype = {request,confirmation,finished};
chan link1 = [1] of {mtype};
chan link2 = [1] of {mtype};
chan link3 = [1] of {mtype};
chan link4 = [1] of {mtype};
chan link5 = [1] of {mtype};
chan link6 = [1] of {mtype};
chan link7 = [1] of {mtype};
chan nullchan = [0] of {mtype};
pid resource_user = 0;
byte priorities[8];

hidden byte priority_level;

proctype client(chan link) {
   do :: link!request;
```

```
        atomic { link?confirmation; resource_user = _pid };
        atomic { resource_user = 0; link!finished }
    od
}

proctype resource_allocator() {

    chan client_chan = nullchan;

    do :: atomic {
             (link1?[request]||link2?[request]||link3?[request]||
              link4?[request]||link5?[request]||link6?[request]||
              link7?[request]);
             priority_level = 2;
             do :: priorities[1]==priority_level && link1?[request] ->
                      client_chan = link1; break
                :: priorities[2]==priority_level && link2?[request] ->
                      client_chan = link2; break
                :: priorities[3]==priority_level && link3?[request] ->
                      client_chan = link3; break
                :: priorities[4]==priority_level && link4?[request] ->
                      client_chan = link4; break
                :: priorities[5]==priority_level && link5?[request] ->
                      client_chan = link5; break
                :: priorities[6]==priority_level && link6?[request] ->
                      client_chan = link6; break
                :: priorities[7]==priority_level && link7?[request] ->
                      client_chan = link7; break
                :: else -> priority_level--
             od;
             client_chan?request;
          }
          client_chan!confirmation;
          d_step { client_chan?finished; client_chan = nullchan }
    od
}

init {
    atomic {
       run client(link1);
       run client(link2);
       run client(link3);
       run client(link4);
       run client(link5);
       run client(link6);
       run client(link7);
       run resource_allocator();

       priorities[1] = 0;
       priorities[2] = 0;
       priorities[3] = 1;
       priorities[4] = 1;
       priorities[5] = 1;
       priorities[6] = 2;
       priorities[7] = 2;
    }
}
```

A.2.2 SMC specification

```
Program

Module client0 = 2;
Module client1 = 3;
Module client2 = 2;
Module resourceallocator = 1;

link0[client0] = 0;
link1[client1] = 0;
link2[client2] = 0;

resourceuser[] = 0;

d0 of client0;
d1 of client1;
d2 of client2;

c0 of client0: {
  link0[c0] == 0 -> link0[c0] = 1;
  link0[c0] == 2 & resourceuser[] == 0 ->
    resourceuser[] = resourceuser[] + 1;
  link0[c0] == 2 & resourceuser[] == 1 ->
    resourceuser[] = resourceuser[]-1, link0[c0] = 3;
}

c1 of client1: {
  link1[c1] == 0 -> link1[c1] = 1;
  link1[c1] == 2 & resourceuser[] == 0 ->
    resourceuser[] = resourceuser[]+1;
  link1[c1] == 2 & resourceuser[] == 1 ->
    resourceuser[] = resourceuser[]-1, link1[c1] = 3;
}

c2 of client2: {
  link2[c2] == 0 -> link2[c2] = 1;
  link2[c2] == 2 & resourceuser[] == 0 ->
    resourceuser[] = resourceuser[]+1;
  link2[c2] == 2 & resourceuser[] == 1 ->
    resourceuser[] = resourceuser[]-1, link2[c2] = 3;
}

r of resourceallocator: {

  link2[c2] == 1 -> link2[c2] = 2;
  link2[c2] == 3 -> link2[c2] = 0;

  link1[c1] == 1 & ALL(d2: link2[d2]==0) -> link1[c1] = 2;
  link1[c1] == 3 -> link1[c1] = 0;

  link0[c0] == 1 & ALL(d2: link2[d2]==0) & ALL(d1: link1[d1]==0) ->
    link0[c0] = 2;
  link0[c0] == 3 -> link0[c0] = 0;
}
```

A.2.3 Promela specification with sharing

```
mtype = {request,confirmation,finished};
chan link1 = [1] of {mtype};
chan link2 = [1] of {mtype};
chan link3 = [1] of {mtype};
chan link4 = [1] of {mtype};
chan link5 = [1] of {mtype};
chan link6 = [1] of {mtype};
chan link7 = [1] of {mtype};
chan nullchan = [0] of {mtype};
pid resource_user = 0;
byte priorities[9];

hidden byte priority_level

proctype client(chan link) {
   do :: link!request;
         atomic { link?confirmation; resource_user = _pid };
         atomic { resource_user = 0;
            if :: _pid==3 && link4?[request] -> link4?request;
                                                 link4!confirmation;
                                                 link4?finished
               :: _pid==4 && link5?[request] -> link5?request;
                                                 link5!confirmation;
                                                 link5?finished
               :: _pid==5 && link3?[request] -> link3?request;
                                                 link3!confirmation;
                                                 link3?finished
               :: else -> skip
            fi;
            link!finished
         }
   od
}

proctype resource_allocator() {

   chan client_chan = nullchan;

   do :: atomic {
            (link1?[request]||link2?[request]||link3?[request]||
             link4?[request]||link5?[request]||link6?[request]||
             link7?[request]);
            priority_level = 2;
            do :: priorities[1]==priority_level && link1?[request] ->
                     client_chan = link1; break
               :: priorities[2]==priority_level && link2?[request] ->
                     client_chan = link2; break
               :: priorities[3]==priority_level && link3?[request] ->
                     client_chan = link3; break
               :: priorities[4]==priority_level && link4?[request] ->
                     client_chan = link4; break
               :: priorities[5]==priority_level && link5?[request] ->
                     client_chan = link5; break
               :: priorities[6]==priority_level && link6?[request] ->
                     client_chan = link6; break
```

```
            :: priorities[7]==priority_level && link7?[request] ->
               client_chan = link7; break
            :: else -> priority_level--
         od;
         client_chan?request;
      };
      client_chan!confirmation;
      d_step { client_chan?finished; client_chan = nullchan }
   od
}

init {
   atomic {
      run client(link1);
      run client(link2);
      run client(link3);
      run client(link4);
      run client(link5);
      run client(link6);
      run client(link7);
      run resource_allocator();

      priorities[1] = 0;
      priorities[2] = 0;
      priorities[3] = 1;
      priorities[4] = 1;
      priorities[5] = 1;
      priorities[6] = 2;
      priorities[7] = 2;
   }
}
```

A.3 Three-tiered Architecture

The following specification models a system with a three-tiered architecture, which is discussed in Section 4.5.

```
mtype = {request,response,query,result};

chan db_link = [0] of {mtype,chan};
chan cl_se_1 = [0] of {mtype,chan};
chan cl_se_2 = [0] of {mtype,chan};
chan cl_se_3 = [0] of {mtype,chan};
chan cl1 = [0] of {mtype};
chan cl2 = [0] of {mtype};
chan cl3 = [0] of {mtype};
chan cl4 = [0] of {mtype};
chan cl5 = [0] of {mtype};
chan cl6 = [0] of {mtype};
chan cl7 = [0] of {mtype};
chan cl8 = [0] of {mtype};
chan se1 = [0] of {mtype};
chan se2 = [0] of {mtype};
chan se3 = [0] of {mtype};

chan null = [0] of {mtype}
```

```
proctype client(chan in; chan link) {
   do :: link!request,in;
         in?response
   od
}

proctype server(chan in; chan c_link) {
   chan current_client=null;
   do :: c_link?request,current_client;
         db_link!query,in;
         in?result;
         current_client!response;
         current_client=null
   od
}

proctype database(chan link) {
   chan current_server=null;
   do :: link?query,current_server;
         current_server!result;
         current_server=null
   od
}

init {
   atomic {
      run database(db_link);
      run server(se1,cl_se_1);
      run server(se2,cl_se_2);
      run server(se3,cl_se_3);
      run client(cl1,cl_se_1);
      run client(cl2,cl_se_1);
      run client(cl3,cl_se_1);
      run client(cl4,cl_se_2);
      run client(cl5,cl_se_2);
      run client(cl6,cl_se_2);
      run client(cl7,cl_se_3);
      run client(cl8,cl_se_3)
   }
}
```

A.4 Message Routing in a Hypercube

The specifications below model message passing in a hypercube network, and are discussed in Sections 4.6 and 8.4.1 respectively.

A.4.1 Original Promela specification

```
/* Determines whether position i of byte bv is set to 1 */
#define IS_1(bv,i) (bv&(1<<i))

mtype = {packet};
```

```
chan link1 = [1] of {mtype};
chan link2 = [1] of {mtype};
chan link3 = [1] of {mtype};
chan link4 = [1] of {mtype};
chan link5 = [1] of {mtype};
chan link6 = [1] of {mtype};
chan link7 = [1] of {mtype};
chan link8 = [1] of {mtype};

pid dest = 0;
pid current = 0;

inline choose_destination() {
   if
      :: _pid!=1 -> dest = 1
      :: _pid!=2 -> dest = 2
      :: _pid!=3 -> dest = 3
      :: _pid!=4 -> dest = 4
      :: _pid!=5 -> dest = 5
      :: _pid!=6 -> dest = 6
      :: _pid!=7 -> dest = 7
      :: _pid!=8 -> dest = 8
   fi
}

inline choose_next_dimension() {
   if
      :: IS_1(((_pid-1)^(dest-1)),0) -> chosen_dimension = 0
      :: IS_1(((_pid-1)^(dest-1)),1) -> chosen_dimension = 1
      :: IS_1(((_pid-1)^(dest-1)),2) -> chosen_dimension = 2
   fi;
   assert(chosen_dimension<3);
}

proctype node(chan in; chan out0; chan out1; chan out2) {

   byte chosen_dimension = 4;

loop:
   atomic {
      in?packet; current = _pid;
      if :: dest==_pid -> choose_destination()
         :: else -> skip
      fi
   }
   atomic { choose_next_dimension();
      if :: chosen_dimension == 0 -> out0!packet
         :: chosen_dimension == 1 -> out1!packet
         :: chosen_dimension == 2 -> out2!packet
      fi;
      chosen_dimension = 4;
      current = 0;
   }
   goto loop
}

init {
   atomic {
```

```
      run node(link1,link2,link3,link5);
      run node(link2,link1,link4,link6);
      run node(link3,link4,link1,link7);
      run node(link4,link3,link2,link8);
      run node(link5,link6,link7,link1);
      run node(link6,link5,link8,link2);
      run node(link7,link8,link5,link3);
      run node(link8,link7,link6,link4);
      if
         :: link1!packet; dest = 1
         :: link2!packet; dest = 2
         :: link3!packet; dest = 3
         :: link4!packet; dest = 4
         :: link5!packet; dest = 5
         :: link6!packet; dest = 6
         :: link7!packet; dest = 7
         :: link8!packet; dest = 8
      fi
   }
}
```

A.4.2 Re-modelled specification which does not involve arithmetic on *pid* variables

```
mtype = {packet};

chan link1 = [1] of {mtype};
chan link2 = [1] of {mtype};
chan link3 = [1] of {mtype};
chan link4 = [1] of {mtype};
chan link5 = [1] of {mtype};
chan link6 = [1] of {mtype};
chan link7 = [1] of {mtype};
chan link8 = [1] of {mtype};

pid dest = 0;
pid current = 0

inline choose_destination() {
   if :: _pid!=1 -> dest = 1
      :: _pid!=2 -> dest = 2
      :: _pid!=3 -> dest = 3
      :: _pid!=4 -> dest = 4
      :: _pid!=5 -> dest = 5
      :: _pid!=6 -> dest = 6
      :: _pid!=7 -> dest = 7
      :: _pid!=8 -> dest = 8
   fi
}

inline choose_next_dimension() {
   if :: _pid==1 && dest==2 -> if :: chosen_dimension = 0 fi
      :: _pid==1 && dest==3 -> if :: chosen_dimension = 1 fi
      :: _pid==1 && dest==4 ->
         if :: chosen_dimension = 0 :: chosen_dimension = 1 fi
      :: _pid==1 && dest==5 -> if :: chosen_dimension = 2 fi
      :: _pid==1 && dest==6 ->
```

```
   if :: chosen_dimension = 0 :: chosen_dimension = 2 fi
:: _pid==1 && dest==7 ->
   if :: chosen_dimension = 1 :: chosen_dimension = 2 fi
:: _pid==1 && dest==8 ->
   if :: chosen_dimension = 0
      :: chosen_dimension = 1
      :: chosen_dimension = 2
   fi
:: _pid==2 && dest==1 -> if :: chosen_dimension = 0 fi
:: _pid==2 && dest==3 ->
   if :: chosen_dimension = 0 :: chosen_dimension = 1 fi
:: _pid==2 && dest==4 -> if :: chosen_dimension = 1 fi
:: _pid==2 && dest==5 ->
   if :: chosen_dimension = 0 :: chosen_dimension = 2 fi
:: _pid==2 && dest==6 -> if :: chosen_dimension = 2 fi
:: _pid==2 && dest==7 ->
   if :: chosen_dimension = 0
      :: chosen_dimension = 1
      :: chosen_dimension = 2
   fi
:: _pid==2 && dest==8 ->
   if :: chosen_dimension = 1 :: chosen_dimension = 2 fi
:: _pid==3 && dest==1 -> if :: chosen_dimension = 1 fi
:: _pid==3 && dest==2 ->
   if :: chosen_dimension = 0 :: chosen_dimension = 1 fi
:: _pid==3 && dest==4 -> if :: chosen_dimension = 0 fi
:: _pid==3 && dest==5 ->
   if :: chosen_dimension = 1 :: chosen_dimension = 2 fi
:: _pid==3 && dest==6 ->
   if :: chosen_dimension = 0
      :: chosen_dimension = 1
      :: chosen_dimension = 2
   fi
:: _pid==3 && dest==7 -> if :: chosen_dimension = 2 fi
:: _pid==3 && dest==8 ->
   if :: chosen_dimension = 0 :: chosen_dimension = 2 fi
:: _pid==4 && dest==1 ->
   if :: chosen_dimension = 0 :: chosen_dimension = 1 fi
:: _pid==4 && dest==2 -> if :: chosen_dimension = 1 fi
:: _pid==4 && dest==3 -> if :: chosen_dimension = 0 fi
:: _pid==4 && dest==5 ->
   if :: chosen_dimension = 0
      :: chosen_dimension = 1
      :: chosen_dimension = 2
   fi
:: _pid==4 && dest==6 ->
   if :: chosen_dimension = 1 :: chosen_dimension = 2 fi
:: _pid==4 && dest==7 ->
   if :: chosen_dimension = 0 :: chosen_dimension = 2 fi
:: _pid==4 && dest==8 -> if :: chosen_dimension = 2 fi
:: _pid==5 && dest==1 -> if :: chosen_dimension = 2 fi
:: _pid==5 && dest==2 ->
   if :: chosen_dimension = 0 :: chosen_dimension = 2 fi
:: _pid==5 && dest==3 ->
   if :: chosen_dimension = 1 :: chosen_dimension = 2 fi
:: _pid==5 && dest==4 ->
   if :: chosen_dimension = 0
      :: chosen_dimension = 1
```

```
            :: chosen_dimension = 2
         fi
      :: _pid==5 && dest==6 -> if :: chosen_dimension = 0 fi
      :: _pid==5 && dest==7 -> if :: chosen_dimension = 1 fi
      :: _pid==5 && dest==8 ->
         if :: chosen_dimension = 0 :: chosen_dimension = 1 fi
      :: _pid==6 && dest==1 ->
         if :: chosen_dimension = 0 :: chosen_dimension = 2 fi
      :: _pid==6 && dest==2 -> if :: chosen_dimension = 2 fi
      :: _pid==6 && dest==3 ->
         if :: chosen_dimension = 0
            :: chosen_dimension = 1
            :: chosen_dimension = 2
         fi
      :: _pid==6 && dest==4 ->
         if :: chosen_dimension = 1 :: chosen_dimension = 2 fi
      :: _pid==6 && dest==5 -> if :: chosen_dimension = 0 fi
      :: _pid==6 && dest==7 ->
         if :: chosen_dimension = 0 :: chosen_dimension = 1 fi
      :: _pid==6 && dest==8 -> if :: chosen_dimension = 1 fi
      :: _pid==7 && dest==1 ->
         if :: chosen_dimension = 1 :: chosen_dimension = 2 fi
      :: _pid==7 && dest==2 ->
         if :: chosen_dimension = 0
            :: chosen_dimension = 1
            :: chosen_dimension = 2
         fi
      :: _pid==7 && dest==3 -> if :: chosen_dimension = 2 fi
      :: _pid==7 && dest==4 ->
         if :: chosen_dimension = 0 :: chosen_dimension = 2 fi
      :: _pid==7 && dest==5 -> if :: chosen_dimension = 1 fi
      :: _pid==7 && dest==6 ->
         if :: chosen_dimension = 0 :: chosen_dimension = 1 fi
      :: _pid==7 && dest==8 -> if :: chosen_dimension = 0 fi
      :: _pid==8 && dest==1 ->
         if :: chosen_dimension = 0
            :: chosen_dimension = 1
            :: chosen_dimension = 2
         fi
      :: _pid==8 && dest==2 ->
         if :: chosen_dimension = 1 :: chosen_dimension = 2 fi
      :: _pid==8 && dest==3 ->
         if :: chosen_dimension = 0 :: chosen_dimension = 2 fi
      :: _pid==8 && dest==4 -> if :: chosen_dimension = 2 fi
      :: _pid==8 && dest==5 ->
         if :: chosen_dimension = 0 :: chosen_dimension = 1 fi
      :: _pid==8 && dest==6 -> if :: chosen_dimension = 1 fi
      :: _pid==8 && dest==7 -> if :: chosen_dimension = 0 fi
   fi;
   assert(chosen_dimension<3)
}

proctype node(chan in; chan out0; chan out1; chan out2) {

   byte chosen_dimension = 4;

loop:
   atomic {
```

```
        in?packet;
        current = _pid;
        if :: dest==_pid -> choose_destination()
           :: else -> skip
        fi
     };
     atomic {
        choose_next_dimension();
        if :: chosen_dimension == 0 -> out0!packet
           :: chosen_dimension == 1 -> out1!packet
           :: chosen_dimension == 2 -> out2!packet
        fi;
        chosen_dimension = 4;
        current = 0
     };
     goto loop
  }

  init {
     atomic {
        run node(link1,link2,link3,link5);
        run node(link2,link1,link4,link6);
        run node(link3,link4,link1,link7);
        run node(link4,link3,link2,link8);
        run node(link5,link6,link7,link1);
        run node(link6,link5,link8,link2);
        run node(link7,link8,link5,link3);
        run node(link8,link7,link6,link4);
        if :: link1!packet; dest = 1
           :: link2!packet; dest = 2
           :: link3!packet; dest = 3
           :: link4!packet; dest = 4
           :: link5!packet; dest = 5
           :: link6!packet; dest = 6
           :: link7!packet; dest = 7
           :: link8!packet; dest = 8
        fi
     }
  }
```

A.5 Telephony

The following Promela code provides part of an example telephone specification which cannot be handled by SymmExtractor, and a re-modelled version which can. They are discussed in some detail in Section 8.5.4.

A.5.1 Original telephone specification

```
mtype = { alert, answer, cutoff, ack };

chan link12 = [0] of { mtype };
chan link21 = [0] of { mtype };

bool idle_st[2] = true;
```

```
bool dial_st[2];
bool calling_st[2];
bool ringing_st[2];
bool talking_st[2];
bool finish_st[2];

proctype user(chan in, out; byte id) {

   mtype response;
   bit is_caller;

idle:
   assert(idle_st[id] && !dial_st[id] && !calling_st[id] &&
          !ringing_st[id] && !talking_st[id]);
   is_caller = 0;
   do :: atomic {
            idle_st[id] = 0;
            dial_st[id] = 1
         };
         goto dial
      :: in?alert ->
         out!ack;
         atomic {
            idle_st[id] = 0;
            ringing_st[id] = 1
         };
         goto ringing
   od;

dial:
   assert(!idle_st[id] && dial_st[id] && !calling_st[id] &&
          !ringing_st[id] && !talking_st[id]);
   do :: out!alert;
         in?response;
         if :: response == ack ->
               atomic {
                  dial_st[id] = 0;
                  calling_st[id] = 1
               };
               is_caller = 1;
               goto calling
            :: response == alert ->
               atomic {
                  dial_st[id] = 0;
                  talking_st[id] = 1
               };
               goto talk
         fi
      :: atomic {
            dial_st[id] = 0;
            finish_st[id] = 1
         };
         goto finish
   od;

   ... etc.

}
```

```
init {
   atomic {
      run user(link21,link12,0);
      run user(link12,link21,1);
   }
}
```

A.5.2 Telephone specification after re-modelling

```
mtype = { alert, answer, cutoff, ack };

chan link12 = [0] of { mtype };
chan link21 = [0] of { mtype };

bool idle_st[3] = true;
bool dial_st[3];
bool calling_st[3];
bool ringing_st[3];
bool talking_st[3];
bool finish_st[3];

proctype user(chan in, out) {

   mtype response;
   bit is_caller;

idle:
   assert(idle_st[_pid] && !dial_st[_pid] && !calling_st[_pid] &&
          !ringing_st[_pid] && !talking_st[_pid]);
   is_caller = 0;
   do :: atomic {
            idle_st[_pid] = 0;
            dial_st[_pid] = 1
         };
         goto dial
      :: in?alert ->
         out!ack;
         atomic {
            idle_st[_pid] = 0;
            ringing_st[_pid] = 1
         };
         goto ringing
   od;

dial:
   assert(!idle_st[_pid] && dial_st[_pid] && !calling_st[_pid] &&
          !ringing_st[_pid] && !talking_st[_pid]);
   do :: out!alert;
         in?response;
         if :: response == ack ->
               atomic {
                  dial_st[_pid] = 0;
                  calling_st[_pid] = 1
               };
               is_caller = 1;
               goto calling
```

```
            :: response == alert ->
               atomic {
                  dial_st[_pid] = 0;
                  talking_st[_pid] = 1
               };
               goto talk
         fi
      :: atomic {
            dial_st[_pid] = 0;
            finish_st[_pid] = 1
         };
         goto finish;
   od;

   ... etc.

}

init {
   atomic {
      run user(link21,link12);
      run user(link12,link21);
   }
}
```

A.6 Railway Signalling System

The Promela code below provides full versions of a specification railway signalling system which is discussed in Section 8.5.4.

A.6.1 Original railway signalling system

```
mtype = {approaches, leaves, lower, raise, atgate, faraway, up, down};

chan control_link = [0] of {mtype, byte};
chan gate_link [8] = [0] of {mtype};

mtype bar[8] = down;
bool on_shared_track[2] = false;
bool shared_track_open = false

proctype train(byte current_gate, id) {

   mtype position = atgate;

   control_link!approaches,current_gate;
   do :: atomic {
            position==faraway ->
if :: current_gate==3 -> current_gate = 0; assert(id==0)
               :: current_gate==7 -> current_gate = 4; assert(id==1)
               :: else -> current_gate++;
            fi;
            control_link!approaches,current_gate; position = atgate}
      :: atomic {
```

```
            (bar[current_gate]==up && position==atgate) ->
            if :: (current_gate==(id*4)) -> on_shared_track[id] = true
               :: else -> skip
            fi;
            position = faraway; control_link!leaves,current_gate;
            if :: (current_gate==(id*4+1)) -> on_shared_track[id] = false
               :: else -> skip
            fi
         }
   od
}

proctype controller() {

   mtype message;
   byte current_gate;

   do :: control_link?message,current_gate ->
         if :: atomic {
                  message==approaches ->
                  gate_link[current_gate]!raise
               }
            :: atomic {
                  message==leaves ->
                  gate_link[current_gate]!lower
               }
         fi
   od
}

proctype gate(byte id) {

   mtype message;

   do :: gate_link[id]?message ->
         if :: atomic {
                  message==lower ->
                  bar[id] = down
               }
            :: atomic {
                  message==raise -> bar[id] = up
               }
         fi
   od
}

proctype shared_gate(byte id) {

   mtype message;

   do :: gate_link[id]?message ->
         if :: atomic {
                  message==lower ->
                  bar[id] = down;
                  assert(shared_track_open);
                  shared_track_open = false
               }
            :: message==raise ->
```

```
lock:          if :: atomic {
                         ((!on_shared_track[0]) && (!on_shared_track[1]) &&
                          (!shared_track_open)) ->
                         shared_track_open = true; bar[id] = up
                      }
                  :: else -> goto lock
               fi
         fi
   od
}

init {
   atomic {
      run controller();
      run shared_gate(0); run gate(1); run gate(2); run gate(3);
      run shared_gate(4); run gate(5); run gate(6); run gate(7);
      run train(2,0); run train(6,1);
   }
}
```

A.6.2 Railway signalling system after re-modelling

```
mtype = {approaches, leaves, lower, raise, atgate, faraway, up, down};

chan control_link = [0] of {mtype, pid};
chan gate_link_2 = [0] of {mtype};
chan gate_link_3 = [0] of {mtype};
chan gate_link_4 = [0] of {mtype};
chan gate_link_5 = [0] of {mtype};
chan gate_link_6 = [0] of {mtype};
chan gate_link_7 = [0] of {mtype};
chan gate_link_8 = [0] of {mtype};
chan gate_link_9 = [0] of {mtype};

mtype bar[12] = down;
bool on_shared_track[12] = false;
bool shared_track_open = false

proctype train(pid current_gate) {

   mtype position = atgate;

   control_link!approaches,current_gate;
   do :: atomic {
            position==faraway ->
            if :: current_gate==2-> current_gate = 3
               :: current_gate==3-> current_gate = 4
               :: current_gate==4-> current_gate = 5
               :: current_gate==5 -> current_gate = 2; assert(_pid==10)
               :: current_gate==6 -> current_gate = 7
               :: current_gate==7 -> current_gate = 8
               :: current_gate==8 -> current_gate = 9
               :: current_gate==9 -> current_gate = 6; assert(_pid==11)
            fi;
            control_link!approaches,current_gate; position = atgate
         }
      :: atomic {
```

```
            (bar[current_gate]==up && position==atgate) ->
            if :: ((_pid==10 && current_gate==2)||
                   (_pid==11 && current_gate==6)) ->
                  on_shared_track[_pid] = true
               :: else -> skip
            fi;
            position = faraway; control_link!leaves,current_gate;
            if :: ((_pid==10 && current_gate==3)||
                   (_pid==11 && current_gate==7)) ->
                  on_shared_track[_pid] = false
               :: else -> skip
            fi
         }
   od
}

inline send(id,msg) {
   if
      :: id==2 -> gate_link_2!msg
      :: id==3 -> gate_link_3!msg
      :: id==4 -> gate_link_4!msg
      :: id==5 -> gate_link_5!msg
      :: id==6 -> gate_link_6!msg
      :: id==7 -> gate_link_7!msg
      :: id==8 -> gate_link_8!msg
      :: id==9 -> gate_link_9!msg
   fi
}

proctype controller() {

   mtype message;
   pid current_gate;

   do :: control_link?message,current_gate ->
         if :: atomic {
                  message==approaches ->
                  send(current_gate,raise)
               }
            :: atomic {
                  message==leaves ->
                  send(current_gate,lower)
               }
         fi
   od
}

proctype gate(chan link) {

   mtype message;

   do :: link?message ->
         if :: atomic { message==lower -> bar[_pid] = down }
            :: atomic { message==raise -> bar[_pid] = up }
         fi
   od
}
```

```
proctype shared_gate(chan link) {

   mtype message;

   do :: link?message ->
         if :: atomic {
                  message==lower -> bar[_pid] = down;
                  assert(shared_track_open); shared_track_open = false
               }
            :: message==raise ->
lock:          if :: atomic {
                        ((!on_shared_track[10]) && (!on_shared_track[11])
                         && (!shared_track_open)) ->
                        shared_track_open = true; bar[_pid] = up
                     }
                  :: else -> goto lock
               fi
         fi
   od
}

init {
   atomic {
      run controller(); run shared_gate(gate_link_2);
      run gate(gate_link_3); run gate(gate_link_4);
      run gate(gate_link_5); run shared_gate(gate_link_6);
      run gate(gate_link_7); run gate(gate_link_8);
      run gate(gate_link_9); run train(4); run train(8);
   }
}
```

Appendix B

Proofs Omitted from the Text

B.1 Proof of the Promela-Lite Progress Theorem (Theorem 11, Section 6.3.5)

The proof of Theorem 11 relies on the following lemma:

Lemma 17 *Let $\mathcal{P}$, $\mathcal{M}$ and s be as above. Let u be an update appearing in a statement of proctype p, and suppose $proctype(i) = p$. If u is '`skip`' or 'x = e' then $exec_{p,i}(s, u)$ is well-defined.*

Proof If u is '`skip`' then the definition of $exec_{p,i}(s, u)$ places no conditions on s, and $exec_{p,i}(s, u) = s$.

Let Γ be the typing environment comprised of entries for the global variables and static channels of $\mathcal{P}$, proctypes appearing before p in $\mathcal{P}$, and the local variables of p. If u has the form 'x = e', where x is an identifier and e an expression then, since $\Gamma \vdash u$ OK, x is not a static channel name, and both x and e have type T where T is a well-formed type which is not the type of a proctype (rule T-ASSIGN). Thus x is the name of a global variable or a local variable of p.

If x is the name of a global variable then we must have $(x = a) \in s$ for some $a \in lit(T)$. Therefore, according to Figure 6.5, $exec_{p,i}(s, u) = \big(s \setminus \{(x = a)\}\big) \cup \{(x = eval_{p,i}(e))\}$, which is clearly well-defined.

On the other hand if x is the name of a local variable then $(p[i].x = a) \in s$ for some $a \in lit(T)$, and we have $exec_{p,i}(s, u) = \big(s \setminus \{(p[i].x = a)\}\big) \cup \{(p[i].x = eval_{p,i}(e)\}$. Again, this is a well-defined state. The result follows. ■

Proof of Theorem 11 Let Γ be the typing environment as defined in the proof of Lemma 17, and let $\langle stmnt \rangle$ denote the Promela-Lite statement `atomic` { g `->` u_1; u_2; $\ldots u_l$ }.

Suppose u_1 has the form `skip` or x = e. Then by Lemma 17, $exec_{p,i}(u_1)$ is well-defined.

Suppose u_1 has the form $x!\overline{e}$. Then x has type $chan\{\overline{T}\}$ in Γ, so x is either a local variable of p, or a static channel name. There is no typing rule from which

$\Gamma \vdash u_1$ OK can be inferred, thus rule T-UPDATE cannot be used to infer that $\Gamma \vdash \langle stmnt \rangle$ OK. Thus $\Gamma \vdash \langle stmt \rangle$ OK must follow from rule T-SEND. Therefore the guard g must have the form `(`h`) && nfull(`x`)`, or just `nfull(`x`)` (see Section 6.2). Since, by hypothesis, $s \models_{p,i} g$, we must have $s \models_{p,i}$ `nfull(`x`)`. Suppose x is a static channel name, so that $(x = [\vec{a_1}, \vec{a_2}, \ldots, \vec{a_m}]) \in s$, where $0 \leq m < cap(x)$. The conditions on s required by the rule for $exec_{p,i}(s, u_1)$ are satisfied. It is easy to see that the resulting state is well-formed. If x is a local variable of p then $(p[i].x = c) \in s$, where c is a static channel name or `null`. However, $s \models_{p,i}$ `nfull(`x`)` $\Leftrightarrow s \models_{p,i}$ `nfull(null)`, and we cannot have $s \models_{p,i}$ `nfull(null)` (see page 121). Thus c is a static channel name, and $exec_{p,i}(s, u_1) = exec_{p,i}(s, c\,!\,\overline{e})$, which is well-defined by the above argument.

Suppose u_1 has the form $x?\overline{x}$. Then by a similar argument (using the fact that the x_i must be distinct, and that no x_i is a static channel name), $exec_{p,i}(s, u_1)$ is well-defined.

We have shown that $exec_{p,i}(s, u_1)$ is well-defined. Suppose that $exec_{p,i}(s, u_j)$ is well-defined for some $1 \leq j < l$. The type rules for statements (T-UPDATE, T-SEND and T-RECV) ensure that u_{j+1} has the form `skip` or x = a. By Lemma 17, $exec_{p,i}(exec_{p,i}(s, u_j), u_{j+1})$ is well-defined.

Since $exec_{p,i}(s, u_1)$ is well-defined, it follows that $exec_{p,i}(\ldots exec_{p,i}(exec_{p,i}(s, u_1), u_2) \ldots, u_l) = exec_{p,i}(s, u_1; u_2; \ldots; u_l)$ is well-defined. ■

B.2 Proof of Lemmas 1 and 2 (Section 7.3.2)

The proof of Lemma 1 depends on the following two sub-lemmas:

Lemma 18 *Let $\alpha \in Aut(\mathcal{SCD}(\mathcal{P}))$ and let e be an expression in $\mathcal{P}$ with $e : int$. Then $eval_{p,i}(s, e) = eval_{p,\alpha(i)}(\alpha(s), \alpha(e))$.*

Proof The Promela-Lite syntax (Figure 6.3) and type system (Figure 6.5) restrict the form of expressions with type *int* to simple expressions of the form:

1. a, where $a \in \mathbb{Z}$
2. x, where x is a local or global variable of type *int*
3. `len(null)`
4. `len(`c`)`, where c is a static channel name
5. `len(`x`)`, where x is a local variable of type *chan*

or an arithmetic combination of the above. Since α only acts on static channel names and values of type *pid*, if e is a simple expression of one of the first three forms above, clearly $\alpha(e) = e$ and $eval_{p,i}(s, e) = eval_{p,\alpha(i)}(\alpha(s), \alpha(e)) = eval_{p,\alpha(i)}(\alpha(s), e)$.

If e has the form `len(`c`)` where c is a static channel, and $(c = [\vec{a_1}, \vec{a_2}, \ldots, \vec{a_m}]) \in s$ then $\alpha(e)$ has the form `len(`$\alpha(c)$`)`, $(\alpha(c) = [\vec{a_1}^{\alpha}, \vec{a_2}^{\alpha}, \ldots, \vec{a_m}^{\alpha}]) \in \alpha(s)$, and

$eval_{p,i}(s,e) = eval_{p,\alpha(i)}(\alpha(s),\alpha(e)) = m$. If e has the form `len(x)` where x is a local variable of $\mathcal{P}$ and $(x = c) \in s$, with c a static channel name or `null`, then $eval_{p,i}(s,e) = eval_{p,i}(s,\texttt{len(c)})$. By the above argument, $eval_{p,i}(s,\texttt{len}(c)) = eval_{p,\alpha(i)}(s,\texttt{len}(\alpha(c))) = eval_{p,\alpha(i)}(\alpha(s),\alpha(e))$.

If e is an arithmetic combination of simple expressions, then clearly by induction the result holds. ■

Lemma 19 *Let e be an expression with $e : pid$ or $e : chan\{\overline{T}\}$. Then*

$$eval_{p,\alpha(i)}(\alpha(s),\alpha(e)) = \alpha(eval_{p,i}(s,e)).$$

Proof The form of expressions of type pid are restricted to: a where $a \in lit(pid)$ and a occurs in a pid context, `_pid`, or x where x is a global/local variable with type pid.

Suppose e has the form a where $a \in lit(pid)$ and a occurs in a pid context. Then $\alpha(e) = \alpha(a)$. We have $eval_{p,\alpha(i)}(\alpha(s),\alpha(e)) = eval_{p,\alpha(i)}(\alpha(s),\alpha(a)) = \alpha(a) = \alpha(eval_{p,i}(s,a)) = \alpha(eval_{p,i}(s,e))$.

If e has the form `_pid` then $eval_{p,\alpha(i)}(\alpha(s),\alpha(e)) = eval_{p,\alpha(i)}(\alpha(s),\texttt{\_pid}) = \alpha(i) = \alpha(eval_{p,i}(s,\texttt{\_pid})) = \alpha(eval_{p,i}(s,e))$.

Now suppose e has the form x where x is a global variable with $x : pid$. Suppose that $(x = a) \in s$, so that $(x = \alpha(s)) \in \alpha(s)$. Then $eval_{p,\alpha(i)}(\alpha(s),\alpha(e)) = eval_{p,\alpha(i)}(\alpha(s),x) = \alpha(a) = \alpha(eval_{p,i}(s,x)) = \alpha(eval_{p,i}(s,e))$. The cases where x is a local variable with $x : pid$ is similar.

The form of expressions of type $chan\{\overline{T}\}$ is restricted to: c where c is a static channel name; `null`; x where x is a local variable with type $chan\{\overline{T}\}$.

If e has the form `null` then $eval_{p,\alpha(i)}(\alpha(s),\alpha(e)) = eval_{p,\alpha(i)}(\alpha(s),\texttt{null}) = \texttt{null} = \alpha(\texttt{null}) = \alpha(eval_{p,i}(s,e))$.

The arguments for the cases where e is a static channel name, or e is a local/global variable with type $chan\{\overline{T}\}$, are analogous to those where e is a pid literal, or e is a local/global variable with type pid. ■

Proof of Lemma 1

Base cases: Suppose g has the form e_1==e_2. By type rule T-EQ we must have $e_1 : T$ and $e_2 : T$ for some type T.

- If $T = int$ then by Lemma 18 $eval_{p,i}(s,e_j) = eval_{p,\alpha(i)}(\alpha(s),\alpha(e_j))$ for $j \in \{1,2\}$. We have $s \models_{p,i} e_1\texttt{==}e_2 \Leftrightarrow eval_{p,i}(s,e_1) = eval_{p,i}(s,e_2) \Leftrightarrow eval_{p,\alpha(i)}(\alpha(s),\alpha(e_1)) = eval_{p,\alpha(i)}(\alpha(s),\alpha(e_2)) \Leftrightarrow \alpha(s) \models_{p,\alpha(i)} \alpha(e_1)\texttt{==}\alpha(e_2)$.
- If $T = pid$ then by Lemma 19 $eval_{p,\alpha(i)}(\alpha(s),\alpha(e_j)) = \alpha(eval_{p,i}(s,e_j)$ for $j \in \{1,2\}$. We have $s \models_{p,i} e_1\texttt{==}e_2 \Leftrightarrow eval_{p,i}(s,e_1) = eval_{p,i}(s,e_2) \Leftrightarrow \alpha(eval_{p,i}(s,e_1)) = \alpha(eval_{p,i}(s,e_2)) \Leftrightarrow eval_{p,\alpha(i)}(\alpha(s),\alpha(e_1)) = eval_{p,\alpha(i)}(\alpha(s),$

$\alpha(e_2)) \Leftrightarrow \alpha(s) \models_{s,\alpha(i)} \alpha(e_1)\texttt{==}\alpha(e_2)$.

- If $T = chan\{\overline{T}\}$ the result follows similarly using Lemma 19.

This completes the argument for the case where g has the form $e_1\texttt{==}e_2$, and the case where g has the form $e_1\,\texttt{!=}e_2$ is similar.

If g has the form $e_1 \texttt{<} e_2$ then the type system requires that $e_1 : int$ and $e_2 : int$ (rule T-REL). We have $s \models_{p,i} e_1 \texttt{<} e_2 \Leftrightarrow eval_{p,i}(s, e_1) < eval_{p,i}(s, e_1)$, and $\alpha(s) \models_{p,\alpha(i)} \alpha(e_1) \texttt{<} \alpha(e_2) \Leftrightarrow eval_{p,\alpha(i)}(\alpha(s), \alpha(e_1)) < eval_{p,\alpha(i)}(\alpha(s), \alpha(e_2))$. By Lemma 18, $eval_{p,i}(s, e_1) = eval_{p,\alpha(i)}(\alpha(s), \alpha(e_1))$ and $eval_{p,i}(s, e_2) = eval_{p,\alpha(i)}(\alpha(s), \alpha(e_2))$. Therefore $eval_{p,i}(s, e_1) < eval_{p,i}(s, e_1) \Leftrightarrow eval_{p,\alpha(i)}(\alpha(s), \alpha(e_1)) < eval_{p,\alpha(i)} (\alpha(s), \alpha(e_2))$, i.e. $s \models_{p,i} e_1 \texttt{<} e_2 \Leftrightarrow \alpha(s) \models_{p,\alpha(i)} \alpha(e_1) \texttt{<} \alpha(e_2)$. The cases $e_1 \texttt{<=} e_2$, $e_1 \texttt{>} e_2$ and $e_1 \texttt{>=} e_2$ are similar.

Suppose g has the form $\texttt{nfull}(c)$ where c is a static channel name. Suppose $(c = [\vec{a_1}, \vec{a_2}, \ldots, \vec{a_k}]) \in s$ for some $0 \leq k \leq cap(c)$. Then $(\alpha(c) = [\vec{a_1}^{\alpha}, \vec{a_2}^{\alpha}, \ldots, \vec{a_k}^{\alpha}]) \in \alpha(s)$. Then $s \models_{p,i} \texttt{nfull}(c) \Leftrightarrow cap(c) > k \Leftrightarrow \alpha(s) \models_{p,\alpha(i)} \texttt{nfull}(\alpha(c))$.

If g has the form $\texttt{nfull}(x)$ where x is a local variable of p with $x : chan\{\overline{T}\}$ then suppose $(p[i].x = \texttt{null}) \in s$. Then $(p[\alpha(i)].x = \texttt{null}) \in \alpha(s)$, and we have $s \not\models_{p,i} \texttt{nfull}(x)$ and $\alpha(s) \not\models_{p,\alpha(i)} \texttt{nfull}(x)$. Suppose instead $(p[i].x = c) \in s$ where c is a static channel name. Then $(p[\alpha(i)].x = \alpha(c)) \in \alpha(s)$. We have $s \models_{p,i} \texttt{nfull}(x) \Leftrightarrow s \models_{p,i} \texttt{nfull}(c) \Leftrightarrow \alpha(s) \models_{p,\alpha(i)} \texttt{nfull}(\alpha(c))$ (by the above argument for static channels) $\Leftrightarrow \alpha(s) \models_{p,\alpha(i)} \texttt{nfull}(x)$.

The cases $\texttt{nempty}(c)$ and $\texttt{nempty}(x)$ where c is a static channel name and x a local variable with $x : chan\{\overline{T}\}$ are similar.

Inductive step:

Suppose the result holds for all guards of length less than m for some $m > 1$. Let g_1, g_2 be guards with length less than m.

If g has the form $\texttt{!}g_1$ then $s \models_{p,i} g \Leftrightarrow s \not\models_{p,i} g_1 \Leftrightarrow \alpha(s) \not\models_{p,\alpha(i)} \alpha(g_1)$ (by inductive hypothesis) $\Leftrightarrow \alpha(s) \models_{p,\alpha(i)} \texttt{!}\alpha(g_1) \Leftrightarrow \alpha(s) \models_{p,\alpha(i)} \alpha(g)$. If g has the form $\texttt{(}g_1\texttt{)}$ the result follows similarly.

If g has the form $g_1 \;\texttt{\&\&}\; g_2$ then $s \models_{p,i} g \Leftrightarrow s \models_{p,i} g_1$ and $s \models_{p,i} g_2$, $\Leftrightarrow$ $\alpha(s) \models_{p,\alpha(i)} \alpha(g_1)$ and $\alpha(s) \models_{p,\alpha(i)} \alpha(g_2)$ (by inductive hypothesis) $\Leftrightarrow \alpha(s) \models_{p,\alpha(i)} \alpha(g_1) \;\texttt{\&\&}\; \alpha(g_2) \Leftrightarrow \alpha(s) \models_{p,\alpha(i)} \alpha(g)$. If g has the form $g_1 \;\texttt{||}\; g_2$ the result follows similarly. ■

The proof of Lemma 2 uses the following sub-lemma:

Lemma 20 *Let u be an update of $\mathcal{P}$, $\alpha \in Aut(\mathcal{SCD}(\mathcal{P}))$ and s a state such that $exec_{p,i}(s, u)$ is well-defined. Then $exec_{p,\alpha(i)}(\alpha(s), \alpha(u)) = \alpha(exec_{p,i}(s, u))$.*

Proof If u is `skip` the result is immediate.

Suppose u has the form $x \texttt{=} e$, and let $var(x)$ be defined as in Figure 6.7. Define $\alpha(var(x)) = x$ if $var(x) = x$, and $\alpha(var(x)) = p[\alpha(i)].x$ if $var(x) = p[i].x$.

If $x : int$ then suppose $(var(x) = a) \in s$. Then $\alpha((var(x) = a)) = (var(x) = a) \in \alpha(s)$ also. Suppose $eval_{p,i}(s, e) = b$. Then $eval_{p,\alpha(i)}(\alpha(s), \alpha(e)) = b$ by Lemma 18. We have $\alpha((var(x) = b)) = (var(x) = b)$, and so

$$\begin{aligned} exec_{p,\alpha(i)}(\alpha(s), `x = \alpha(e)') &= \big(\alpha(s) \setminus \{(var(x) = a)\}\big) \cup \{(var(x) = b)\} \\ &= \alpha\big((s \setminus \{(var(x) = a)\}\big) \cup \{(var(x) = b)\}\big) \\ &= \alpha(exec_{p,i}(s, `x = e'). \end{aligned}$$

If $x : pid$, then suppose $(var(x) = a) \in s$. Then $\alpha((x = a)) = (\alpha(var(x)) = \alpha(a)) \in \alpha(s)$ also. Suppose $eval_{p,i}(s, e) = b$. Then $eval_{p,\alpha(i)}(\alpha(s), \alpha(e)) = \alpha(b)$ by Lemma 18. We have $\alpha((var(x) = b)) = (\alpha(var(x)) = \alpha(b))$, and so

$$\begin{aligned} exec_{p,\alpha(i)}(\alpha(s), `x = \alpha(e)') &= \big(\alpha(s) \setminus \{(\alpha(var(x)) = \alpha(a))\}\big) \cup \\ &\qquad \{(\alpha(var(x)) = \alpha(b))\} \\ &= \alpha\big((s \setminus \{(var(x) = a)\}\big) \cup \{(var(x) = b)\}\big) \\ &= \alpha(exec_{p,i}(s, `x = e'). \end{aligned}$$

The argument is similar if $x : chan\{\overline{T}\}$.

Suppose u has the form $x\,!\,e_1, e_2, \ldots, e_k$, and suppose x is a static channel name, with $x : chan\{T_1, T_2, \ldots, T_k\}$ so that $e_j : T_j$ ($1 \leq j \leq k$). Suppose $(x = [\vec{a_1}, \vec{a_2}, \ldots, \vec{a_m}]) \in s$ for some $m < cap(x)$. Then $\alpha((x = [\vec{a_1}, \vec{a_2}, \ldots, \vec{a_m}])) = (\alpha(x) = [\vec{a_1}^{\alpha}, \vec{a_2}^{\alpha}, \ldots, \vec{a_m}^{\alpha}]) \in \alpha(s)$. For $1 \leq j \leq k$, let a_j denote $eval_{p,i}(s, e_j)$, and let $b_j = a_j$ if $T_j = int$, and $b_j = \alpha(a_j)$ otherwise. Using Lemmas 18 and 19, we have $b_j = eval_{p,\alpha(i)}(\alpha(s), \alpha(e_j))$. Thus $(b_1, b_2, \ldots, b_k) = (a_1, a_2, \ldots, a_k)^{\alpha}$ (using the notation of Section 7.2.2). Then $exec_{p,\alpha(i)}(\alpha(s), `\alpha(x)\,!\,\alpha(e_1), \alpha(e_2), \ldots, \alpha(e_k)') =$

$$\begin{aligned} &= \big(\alpha(s) \setminus \{(\alpha(x) = [\vec{a_1}^{\alpha}, \vec{a_2}^{\alpha}, \ldots, \vec{a_m}^{\alpha}])\}\big) \cup \\ &\qquad \{(\alpha(x) = [\vec{a_1}^{\alpha}, \vec{a_2}^{\alpha}, \ldots, \vec{a_m}^{\alpha}, (b_1, b_2, \ldots, b_k)])\} \\ &= \big(\alpha(s) \setminus \{(\alpha(x) = [\vec{a_1}^{\alpha}, \vec{a_2}^{\alpha}, \ldots, \vec{a_m}^{\alpha}])\}\big) \cup \\ &\qquad \{(\alpha(x) = [\vec{a_1}^{\alpha}, \vec{a_2}^{\alpha}, \ldots, \vec{a_m}^{\alpha}, (a_1, a_2, \ldots, a_k)^{\alpha}])\} \\ &= \alpha\big((s \setminus \{(x = [\vec{a_1}, \vec{a_2}, \ldots, \vec{a_m}])\}\big) \cup \\ &\qquad \{(x = [\vec{a_1}, \vec{a_2}, \ldots, \vec{a_m}, (a_1, a_2, \ldots, a_k)])\}\big) \\ &= \alpha(exec_{p,i}(s, `x\,!\,e_1, e_2, \ldots, e_k')). \end{aligned}$$

If x is a local variable of p then suppose $(x = c) \in s$, where c is a static channel name. Therefore $(x = \alpha(c)) \in \alpha(s)$, and $exec_{p,\alpha(i)}(\alpha(s), `x\,!\,\alpha(e_1), \alpha(e_2), \ldots, \alpha(e_k)') = exec_{p,\alpha(i)}(\alpha(s), `\alpha(c)\,!\,\alpha(e_1), \alpha(e_2), \ldots, \alpha(e_k)') = \alpha(exec_{p,i}(s, `c\,!\,e_1, e_2, \ldots, e_k'))$ (by the above argument) $= \alpha(exec_{p,i}(s, `x\,!\,e_1, e_2, \ldots, e_k'))$.

Suppose u has the form $x?x_1, x_2, \ldots, x_k$, and suppose x is a static channel name, with $x : chan\{T_1, T_2, \ldots, T_k\}$ so that $x_j : T_j$ ($1 \leq j \leq k$). Suppose

$(x = [(a_{1,1}, a_{1,2}, \ldots, a_{1,k}), \vec{a_2}, \ldots, \vec{a_m}]) \in s$ for some $m < cap(x)$, and $(var(x_j) = b_j) \in s$ $(1 \leq j \leq k)$. Define $d_{1,j} = a_{1,j}$ if $T_j = int$, and $d_{1,j} = \alpha(a_{1,j})$ otherwise $(1 \leq j \leq k)$. Then $(d_{1,1}, d_{1,2}, \ldots, d_{1,k}) = (a_{1,1}, a_{1,2}, \ldots, a_{1,k})^\alpha$ (using the notation of Section 7.2.2), and $\alpha((var(x_j) = a_{1,j})) = (\alpha(var(x_j) = d_{1,j})$ $(1 \leq j \leq k)$. Similarly, define $d_j = b_j$ if $x_j : int$, and $d_j = \alpha(b_j)$ otherwise. Then $\alpha((var(x_j) = b_j)) = (\alpha(var(x_j)) = d_j)$. We have $\alpha((x = [(a_{1,1}, a_{1,2}, \ldots, a_{1,k}), \vec{a_2}, \ldots, \vec{a_m}])) = (\alpha(x) = [(d_{1,1}, d_{1,2}, \ldots, d_{1,k}), \vec{a_2}^\alpha, \ldots, \vec{a_m}^\alpha]) \in \alpha(s)$, and $\alpha((var(x_j) = b_j)) = (\alpha(var(x_j)) = d_j) \in \alpha(s)$ $(1 \leq j \leq k)$. Then $exec_{p,\alpha(i)}(\alpha(s), `\alpha(x)?x_1, x_2, \ldots, x'_k) =$

$$
\begin{aligned}
= \quad & \big(\alpha(s) \setminus \{(\alpha(x) = [(d_{1,1}, d_{1,2}, \ldots, d_{1,k}), \vec{a_2}^\alpha, \ldots, \vec{a_m}^\alpha]), \\
& (\alpha(var(x_1)) = d_1), (\alpha(var(x_2)) = d_2), \ldots, (\alpha(var(x_k)) = d_k)\}\big) \cup \\
& \{(\alpha(x) = [\vec{a_2}^\alpha, \ldots, \vec{a_m}^\alpha]), (\alpha(var(x_1)) = d_{1,1}), (\alpha(var(x_2)) = d_{1,2}), \\
& \ldots, (\alpha(var(x_k)) = d_{1,k})\} \\
= \quad & \big(\alpha(s) \setminus \{(\alpha(x) = [(a_{1,1}, a_{1,2}, \ldots, a_{1,k})^\alpha, \vec{a_2}^\alpha, \ldots, \vec{a_m}^\alpha]), \\
& \alpha((var(x_1) = b_1)), \alpha((var(x_2) = b_2)), \ldots, \alpha((var(x_k) = b_k))\}\big) \cup \\
& \{(\alpha(x) = [\vec{a_2}^\alpha, \ldots, \vec{a_m}^\alpha]), \alpha((var(x_1) = a_{1,1})), \alpha((var(x_2) = a_{1,2})), \\
& \ldots, \alpha((var(x_k) = a_{1,k}))\} \\
= \quad & \alpha\big((s \setminus \{(x = [(a_{1,1}, a_{1,2}, \ldots, a_{1,k}), \vec{a_2}, \ldots, \vec{a_m}]), \\
& (var(x_1) = b_1), (var(x_2) = b_2), \ldots, (var(x_k) = b_k)\}\big) \cup \\
& \{(x = [\vec{a_2}, \ldots, \vec{a_m}]), (var(x_1) = a_{1,1}), (var(x_2) = a_{1,2}), \ldots, \\
& (var(x_k) = a_{1,k})\}\big) \\
= \quad & \alpha(exec_{p,i}(s, `x?x_1, x_2, \ldots, x'_k)).
\end{aligned}
$$

If x is a local variable of p then suppose $(x = c) \in s$, where c is a static channel name. Therefore $(x = \alpha(c)) \in \alpha(s)$, and $exec_{p,\alpha(i)}(\alpha(s), `x?x_1, x_2, \ldots, x'_k) = exec_{p,\alpha(i)}(\alpha(s), `\alpha(c)?x_1, x_2, \ldots, x'_k) = \alpha(exec_{p,i}(s, `c?x_1, x_2, \ldots, x'_k))$ (by the above argument) = $\alpha(exec_{p,i}(s, `x?x_1, x_2, \ldots, x'_k))$. ■

Proof of Lemma 2 As defined on page 121, $exec_{p,\alpha(i)}(\alpha(s), \alpha(u_1); \alpha(u_2); \ldots; \alpha(u_k))$

$$
\begin{aligned}
= \quad & exec_{p,\alpha(i)}(\ldots exec_{p,\alpha(i)}(exec_{p,\alpha(i)}(\alpha(s), \alpha(u_1)), \alpha(u_2)) \ldots, \alpha(u_k)) \\
= \quad & exec_{p,\alpha(i)}(\ldots exec_{p,\alpha(i)}(\alpha(exec_{p,i}(s, u_1)), \alpha(u_2)) \ldots, \alpha(u_k)) \\
& \quad \text{(by Lemma 20)} \\
= \quad & exec_{p,\alpha(i)}(\ldots \alpha(exec_{p,i}(exec_{p,i}(s, u_1), u_2)) \ldots, \alpha(u_k)) \\
& \quad \text{(by Lemma 20)} \\
\vdots \\
= \quad & \alpha(exec_{p,i}(\ldots exec_{p,i}(exec_{p,i}(s, u_1), u_2) \ldots, u_k)) \\
& \quad \text{(by repeated application of Lemma 20)} \\
= \quad & \alpha(exec_{p,i}(s, u_1; u_2; \ldots; u_k)). \ ■
\end{aligned}
$$

B.3 Proofs of Lemmas 8 – 10 (Section 9.3.2)

Proof of Lemma 8 Let $\mathcal{O}_i$ be the set of orbits of H_i, for $i \in \{1,2\}$. Since $moved(H_1) \cap moved(H_2) = \emptyset$, $\{\mathcal{O}_1, \mathcal{O}_2\}$ is a partition of $\mathcal{O}$. For $i \in \{1,2\}$, every $x \in moved(H_i)$ must belongs to a non-trivial orbit of H_i, thus $moved(H_i) \subseteq \cup\mathcal{O}_i$. But clearly if x belongs to a non-trivial orbit of H_i, i.e. $x \in \mathcal{O}_i$, then $x \in moved(H_i)$, so $moved(H_i) = \cup\mathcal{O}_i$. Let $\alpha = \alpha_1\alpha_2 \in G$, where $\alpha_i \in H_i$ ($i \in \{1,2\}$). Then $\alpha^{moved(H_i)} = \alpha_i$, so $G^{moved(H_i)} \subseteq H_i$. Clearly $H_i \subseteq G^{moved(H_i)} = G^{\mathcal{O}_i}$. The result follows. ■

Proof of Lemma 9 Since $\cup\mathcal{O}_1$ and $\cup\mathcal{O}_2$ are disjoint, we have $moved(G^{\mathcal{O}_1}) \cap moved(G^{\mathcal{O}_2}) = \emptyset$. Let $\alpha \in G$. Then α can be written as a product of disjoint, mutually commutative permutations, each acting on a distinct orbit of G. Therefore $\alpha = \alpha_1\alpha_2$, where α_i acts on the orbits of $\mathcal{O}_i$, i.e. $\alpha_i \in G^{\mathcal{O}_i}$ for $i \in \{1,2\}$. We have shown that $G = G^{\mathcal{O}_1}G^{\mathcal{O}_2}$. The result follows ■

Proof of Lemma 10 Suppose, without loss of generality, that $\Omega_i \in \mathcal{O}_1$. If $\Omega_j \notin \mathcal{O}_1$ then we must have $\Omega_j \in \mathcal{O}_2$. Since Ω_i and Ω_j are dependent, $stab^*_G(\Omega_j)^{\Omega_i} \subset G^{\Omega_i}$, so there exists $\alpha \in G$ such that $\alpha^{\Omega_i} \neq id$, $\alpha^{\Omega_j} \neq id$ (since α does not fix every point in Ω_j, and $\alpha^{\Omega_i} \notin (stab^*_G(\Omega_j))^{\Omega_i}$.

The permutation α can be expressed in as a product $\alpha_i\beta_1\alpha_j\beta_2$, where α_i only acts on Ω_i, β_1 acts on $\mathcal{O}_1 \setminus \{\Omega_i\}$, α_j only acts on Ω_j, and β_2 acts on $\mathcal{O}_2 \setminus \{\Omega_j\}$. Now $G = G^{\mathcal{O}_1} \bullet G^{\mathcal{O}_2}$, so every element γ of G can be expressed uniquely as a product $\gamma = \gamma_1\gamma_2$ where $\gamma_1 \in G^{\mathcal{O}_1}$, $\gamma_2 \in G^{\mathcal{O}_2}$, and $\gamma_1, \gamma_2 \in G$. For the element α, we have $\gamma_1 = \alpha_i\beta_1$ and $\gamma_2 = \alpha_j\beta_2$. It follows that $\alpha_j\beta_2 \in G$. Therefore $(\alpha_i\beta_1\alpha_j\beta_2)(\alpha_j\beta_2)^{-1} \in G$, i.e. $(\alpha_i\beta_1)(\alpha_j\alpha_j^{-1}\beta_2\beta_2^{-1}) \in G$ (α_j and β_2 commute, since they act on disjoint points), i.e. $\alpha_i\beta_1 = \delta$, say, belongs to G. Clearly $\delta^{\Omega_i} = \alpha^{\Omega_i}$, but $\delta \in stab^*_G(\Omega_j)$. It follows that $\alpha^{\Omega_i} \in stab^*_G(\Omega_j)^{\Omega_i}$. This is a contradiction. It follows that $\Omega_j \in \mathcal{O}_1$. ■

Appendix C

SymmExtractor and TopSpin

In Appendix C.1 we survey the features of Promela which are not part of Promela-Lite, discussing whether or not they are supported by SymmExtractor. We provide a brief guide to the installation and use of TopSpin (which incorporates SymmExtractor) in Appendix C.2. In Appendix C.2.3 we present a set of modelling guidelines to aid the construction of Promela specifications for use with SymmExtractor and TopSpin. These guidelines are based on findings of the user study of Section 8.5.

C.1 Promela vs. Promela-Lite in the Context of SymmExtractor

Promela features which are not part of Promela-Lite but are supported by SymmExtractor are discussed in Appendix C.1.1. In Appendix C.1.2 we discuss features of Promela which are not currently supported by SymmExtractor but could be handled relatively easily. In Appendix C.1.3 we list Promela features which cannot be handled by simple extensions to the theory of Chapter 7, and would require additional research effort to be supported by our implementation.

C.1.1 Supported omissions

All of the Promela features discussed in the following categories *are* supported by SymmExtractor despite not being part of Promela-Lite. In most cases it is obvious that the techniques presented in Chapters 6 and 7 could be extended in a trivial (if laborious) manner to handle the features. We provide a brief justification for certain more complex cases and note some features which are supported by SymmExtractor but not TopSpin.

Types and variables

SymmExtractor supports the following Promela/non-Promela-Lite features which relate to types and variables:

- Primitive data types *bit*, *bool*, *mtype*, *byte* and *short*
- Arrays indexed using the *byte* type

- User-defined record types
- Boolean literals `true` and `false`
- Local variables (in addition to parameters)
- The built-in '_' variable for the receipt of 'don't care' (scratch) message fields.

Arrays which are indexed using the *pid* data type are also supported. These are slightly more complex: a static channel diagram automorphism acting on a state of a specification should permute the positions of elements of a *pid*-indexed array in the obvious way. Also, an expression of the form $A[d]$ in a specification $\mathcal{P}$, where A is a *pid*-indexed array and d a literal *pid* value, should be replaced with the expression $A[\alpha(d)]$ in $\alpha(\mathcal{P})$, where $\alpha \in Aut(\mathcal{SCD}(\mathcal{P}))$. It is clear that the results of Chapter 7 could be extended to handle arrays with *pid* index type. As discussed in Section 8.1.1, an array should be indexed using either *byte* or *pid*, but not both.

Promela allows the declaration of synchronous channels, which are not part of Promela-Lite. Formally extending the Promela-Lite semantics and the results of Chapter 7 to take into account synchronous channels would be straightforward, but laborious. They are supported by SymmExtractor.

Control structures and expressions

SymmExtractor supports the full range of Promela control structures, together with some forms of expression which are not included in Promela-Lite:

- Separation of statements using `;` or `->`
- Conditional `if..fi` constructs
- Nested `do..od` constructs (Promela-Lite specifications include a single, mandatory, top-level `do..od` construct)
- Label definitions, and statements of the form `goto` ⟨*label*⟩
- `break`, `else`, `unless`, `provided` and `timeout`
- Condition expressions of the form `(`⟨*boolean-expr*⟩ `->` ⟨*expr*⟩ `:` ⟨*expr* ⟩`)`
- Expressions as statements
- Receive poll expressions.

A Promela specification which uses these language features can be translated into a less elegant but equivalent Promela-Lite specification, via the introduction of an explicit program counter variable.

To manage the complexity of a specification, Promela allows the inclusion of inline macros, similar to procedures in an imperative language. Macro invocations are expanded by SPIN using textual replacement before verification. SymmExtractor deals with inline macros similarly.

Promela-Lite includes the `atomic` keyword, but the type system of Section 6.2 ensures that the statements within an atomic statement cannot block. In Promela it is possible, and permissible, for blocking to occur within an `atomic` sequence. The semantics for this are rather complex, but are clearly orthogonal to symmetry-related issues. Therefore, unrestricted `atomic` blocks are supported by SymmExtractor.

Handling `d_step` blocks is more complex. Recall from Section 2.4.1 that a `d_step` block must not involve non-determinism. This cannot be statically checked, so the verifier generated by SPIN for a given specification checks for non-determinism within `d_step` blocks during search. If non-deterministic choice is possible in a `d_step` block then the first executable choice is taken by the verifier, and a warning generated. This means that options to `if..fi` and `do..od` statements are not, in general, commutative within a `d_step` block. For this reason, when checking whether $\mathcal{P} \equiv \alpha(\mathcal{P})$ as described in Section 8.3.2, SymmExtractor does not sort the options of `if..fi` and `do..od` statements which occur within `d_step` blocks.

Operators

The following Promela operators are not part of Promela-Lite, but are supported by SymmExtractor:

- `empty` and `full`
- Non-destructive channel read operator
- Bitwise, modulo and division operators
- `eval` operator (and receipt of messages corresponding to literal values).

Simulation features

The Promela keywords `printf`, `STDIN`, `show` and `priority` can be used to aid simulation of a specification, but have no effect on verification. SymmExtractor ignores the use of these keywords in a Promela specification.

Reasoning mechanisms

SymmExtractor supports property specification using `assert` statements, never claims, `accept`/`progress` labels, and trace/notrace constructs (see [92] for details of these).

Since never claims and trace/notrace constructs are Promela processes they can be handled by the existing theory of Chapter 7. Furthermore, a group of valid static channel diagram automorphisms is, by default, an invariance group for the property represented by a never claim or trace/notrace construct (see Section 11.4).

Note that the TopSPIN symmetry reduction package is currently limited to the verification of simple safety properties expressed via assertions, as discussed in Section 11.4. It does *not* support never claims, `accept`/`progress` labels or trace/notrace constructs.

Miscellaneous

Unlike Promela, Promela-Lite does not include syntax for comments. SymmExtractor allows specification to include Promela style comments, which obviously has no effect on symmetry.

The `hidden` keyword can be used to tell SPIN to exclude a global variable from the state vector (see Section 2.4.1). The value of a hidden variable x at a given state s during search depends only on the previously visited state, not on the state s itself. Thus, in general, no assumptions can be made about the value of x at s, unless x is known to be a constant, and hidden variables are intended to be used as "scratch" variables within atomic statements [92]. SymmExtractor supports use of the `hidden` keyword (by ignoring its occurrence) and, like SPIN, places the responsibility of its correct usage on the user.

A global variable can be prefixed with the `local` keyword to tell the SPIN partial order reduction algorithm that the variable is accessed by a single process as if it were local to that process. Since this keyword has no relation to symmetry it is allowed, and ignored, by SymmExtractor.

Promela includes keywords `xr` and `xs`, which stand for *exclusive receive* and *exclusive send* respectively. A process can included a declaration `xr` $\langle$*name*$\rangle$, where $\langle$*name*$\rangle$ is the name of a previously declared channel, to indicate that only this process can receive messages on the channel. The `xs` keyword is used similarly. Providing SPIN with this information can lead to more efficient partial-order reduction. It is not possible to check, statically, whether `xs` and `xr` are used correctly, but incorrect uses are flagged by SPIN during verification. These keywords do not affect the presence of symmetry in the model associated with a specification, so are supported by SymmExtractor. However, there is a problem with exploiting `xs/xr` information in conjunction with symmetry reduction. Let $\mathcal{P}$ be a Promela specification with associated model $\mathcal{M}$, and c a channel in $\mathcal{P}$. Suppose c is marked `xs` by process 1, and there is some valid $\alpha \in Aut(\mathcal{SCD}(\mathcal{P}))$ with $\alpha(1) = 2$ and $\alpha(c) = c$. Assume that there is exactly one transition (s,t) in $\mathcal{M}$ which involves process 1 sending on c. Then the transition $(\alpha(s), \alpha(t))$ involves process 2 sending on c, violating the `xs` assertion on c. Clearly this is the only such transition. When model checking without symmetry reduction, both (s,t) and $(\alpha(s), \alpha(t))$ will be considered, and the `xs` violation detected. However, with symmetry reduction only one of these pairs of transition will be considered, so this violation of the `xs` assertion will not be detected. An analogous argument can be given for `xr`. Therefore TopSPIN should not be applied to specifications which use `xs` and `xr` assertions.

C.1.2 Omissions which could be supported

Our implementation requires Promela processes to be instantiated using `run` statements within an `init` process. However, Promela also allows multiple copies of a given proctype to be instantiated by prefixing the `proctype` keyword with `active [`k`]`, where $k > 0$ is the number of processes of the proctype to be instantiated. Use of this keyword changes the way in which run-time process identifiers are assigned by SPIN, thus changes the way a static channel diagram is constructed. With some effort, our implementation could be modified to accommodate this method of instantiating processes.

Promela supports an *unsigned* numeric type. A declaration of the form `unsigned` x `:` y declares an integer variable x which takes non-negative values which can be represented using y bits. Clearly the use of this data type will have no effect on our symmetry detection/reduction techniques. However, SymmExtractor is integrated with an enhanced Promela type checker (see Section 8.2), which does not currently support the *unsigned* data type. A temporary fix for this omission is to replace each occurrence of the *unsigned* keyword with one of the other numeric types during symmetry detection.

Though not strictly part of the Promela language, SPIN supports Promela specifications which include C-style `#define` macros. SymmExtractor could easily be extended to handle this kind of macro by applying the C pre-processor to a specification before parsing.

SymmExtractor does not allow channel initialisers to be associated with channels which are declared locally to a proctype. This is to simplify the identification of channels for inclusion in the static channel diagram. In a specification where processes are created dynamically, local channel initialisers result in dynamic instantiation of channels, which does not fit comfortably with the static channel diagram concept. However, since SymmExtractor requires a fixed number of running processes, it would be possible to extend SymmExtractor to allow locally initialised channels.

For simplicity, SymmExtractor does not currently support arrays of channels. Further implementation work could remove this restriction.

C.1.3 Omissions which cannot currently be supported

As noted above, it is hard to see how a specification where processes are created dynamically fits in with the *static channel diagram* concept on which SymmExtractor is based. This is not to say that specifications with dynamic process creation do not exhibit symmetry: indeed, SPIN-to-GRAPE can be used to check that the "Agents and Servers" specification of [92], which involves dynamic process creation, exhibits a non-trivial automorphism group. Extending our techniques to identify symmetry with a dynamically changing pool of processes will require further theoretical work, perhaps building on techniques for this problem developed for the dSPIN model checker [98] (see Section 3.9.4). For the time being, specifications which involve dynamic process creation can be re-modelled using the approach described in Appendix C.2.3.

Package	URL	Version
Java runtime environment	`http://java.sun.com/`	1.5.0_06
JUnit library (`junit.jar`)	`http://junit.org/`	3.8.1
GAP system	`http://gap-system.org/`	4.4.6
SPIN model checker	`http://spinroot.com/`	4.2.6
GNU C Compiler (gcc)	`http://gcc.gnu.org/`	3.3.4
PERL	`http://www.perl.org/`	5.8.4

Figure C.1: TopSPIN prerequisites.

Promela provides alternative channel operators '`!!`' (sorted send) and '`??`' (random receive). Sending data on a buffered channel using '`!!`' causes messages to be queued on the channel in sorted order. Messages can be retrieved from the buffer in a random order using '`??`'. These operators provide a useful alternative to FIFO channel semantics. They also aid state-space reduction: storing channel contents in a sorted manner can be seen as a form of state canonicalisation. However, storing *pid* messages in a sorted queue imposes an ordering on the set $lit(pid)$. It is not immediately clear whether this ordering has an effect on symmetry, so for the time being SymmExtractor does not support the '`!!`' and '`??`' operators.

Recent versions of SPIN allow C code to be embedded in a Promela specification, and certain variables from the C part of the specification to be included in the SPIN state vector. Automatic symmetry detection for this mix of C and Promela is beyond the scope of this thesis, but is certainly an interesting area for further research.

C.2 TopSPIN Installation and User Guide

We provide instructions on how to obtain and configure TopSPIN, together with details of the third-party packages required by the tool, in Appendix C.2.1. In Appendix C.2.2 we provide a brief guide to the use of TopSPIN. Some modelling guidelines are given in Appendix C.2.3.

C.2.1 Installing and configuring TopSPIN

Prerequisites

TopSPIN is written in Java, PERL and GAP, interfaces with the GAP and SPIN packages, and produces C code which must then be compiled. The Java implementation requires the JUnit library. Figure C.1 summarises the packages which must be installed before TopSPIN can be used, providing URLs for each. The version of each package which we have used for development of TopSPIN is also given.

In addition, TopSPIN uses a prototype extension of *saucy* which has been extended to handle directed graphs. This functionality will eventually be available from the *saucy* website (`http://vlsicad.eecs.umich.edu/BK/SAUCY/`). For the time being, a source distribution of *saucy* with the required extended functionality is provided with TopSPIN.[1]

Downloading the software

The TopSPIN release distribution is available online as an archive from the *Software* page at the following URL:

`http://www.dcs.gla.ac.uk/people/personal/ally/thesis/`

The files are compressed using the Linux utilities `gzip` and `tar`, and should be extracted using standard tools. After extraction, move the `TopSPIN` folder and its contents to a suitable location (e.g. `C:\Program Files\TopSPIN` under Windows), and navigate to this folder. The folder should contain `TopSPIN.jar`, together with the sub-folders `lib`, `saucy`, `Common` and `TempFiles`. Copy the `junit.jar` file into the `lib` folder.

Setting up a GAP *workspace*

In order to start GAP efficiently, TopSPIN requires a GAP *workspace* to be set up. Full details of GAP workspaces are available online [63]. Essentially, a workspace is an image of a GAP session with a selection of libraries and files already loaded and ready to be executed. In our case, the workspace consists of the GAP files used for automatic symmetry detection and classification.

Navigate into the `Common` folder. Start GAP and type:

```
Read("WorkspaceGenerator.gap");
```

followed by:

```
SaveWorkspace("gapworkspace");
```

Ensure that these commands are typed *exactly* as shown. Entering the second command should result in `true` being printed to the console. Exit GAP by typing `quit;` (ensuring that the semi-colon is included in this command).

Compiling saucy

Navigate into the `saucy` folder, and type `make`. Assuming that `gcc` is correctly installed, this is all that should be required to compile the *saucy* program.

Setting up a configuration file

TopSPIN uses a textual configuration file, `config.txt` to locate GAP, *saucy*, various common files and a folder for temporary files, during execution. Symmetry detection and reduction options are also specified in this file.

The structure of `config.txt` is summarised in Figure C.2. Example configuration files for Windows and Linux systems are given in Figures C.3 and C.4

1. Permission for including the *saucy* distribution with TopSPIN has been granted by Paul Darga, lead developer of *saucy*.

Line	Description	Default
`gap`	path to GAP	n/a
`saucy`	path to *saucy*	n/a
`tempfiles`	path to TopSPIN`TempFiles` folder	n/a
`common`	path to TopSPIN `Common` folder	n/a
`timebound`	bound, in seconds, for largest valid subgroup computation	no bound
`conjugates`	number of random conjugates to be used	0
`transpositions`	boolean indicating whether permutations should be applied as transpositions	true
`stabiliserchain`	boolean indicating whether to use a stabiliser chain for enumeration	true
`strategy`	symmetry reduction strategy	fast
`symmetryfile`	path to file containing symmetry group generators	n/a

Figure C.2: Structure of a TopSPIN configuration file.

```
gap=C:\gap4r4\bin\gap.bat
saucy=C:\Documents and Settings\Ally D\TopSPIN\saucy\saucy.exe
tempfiles=C:\Documents and Settings\Ally D\TopSPIN\TempFiles\
common=C:\Documents and Settings\Ally D\TopSPIN\Common\
timebound=0
conjugates=0
transpositions=true
stabiliserchain=true
strategy=fast
```

Figure C.3: A TopSPIN configuration file for Windows.

respectively. Users should create their own configuration file based on their specific setup and symmetry reduction needs. The configuration options related to symmetry detection and reduction are described in Appendix C.2.2.

C.2.2 Using SymmExtractor and TopSPIN

The TopSPIN jar file can be executed to: typecheck a specification to see if it is suitable for symmetry reduction; detect symmetries of a specification (i.e. run SymmExtractor), or add symmetry reduction algorithms to the C code generated by SPIN for a given specification. Note that in all cases the `config.txt` file must be in the current directory. We use `TOPSPINPATH` to denote the TopSPIN folder (e.g. `C:\Program Files\TopSPIN`) and `SPECIFICATION` the Promela specification to which TopSPIN is being applied (e.g. `loadbalancer.p`).

Typechecking a specification

To typecheck a specification, type:

```
java -jar TOPSPINPATH/TopSPIN.jar check SPECIFICATION
```

Detecting symmetry

To apply SymmExtractor to find symmetries associated with a specification, type:

```
gap=/users/grad/ally/Scripts/gap
saucy=/users/grad/ally/TopSPIN/saucy/saucy
tempfiles=/users/grad/ally/TopSPIN/TempFiles/
common=/users/grad/ally/TopSPIN/Common/
timebound=10
conjugates=4
transpositions=true
stabiliserchain=true
strategy=enumerate
```

Figure C.4: A TopSPIN configuration file for Linux.

`java -jar TOPSPINPATH/TopSPIN.jar detect SPECIFICATION`

For certain specifications, the search for the largest valid subgroup of symmetries for a given specification may be time-consuming. A bound of x seconds for this search can be specified by adding the line:

`timebound=`x

to `config.txt`. If no bound is required then add the line `timebound=0` to the file. To specify that $x \geq 0$ random conjugates should be used for symmetry detection (see Section 8.3.3), add the line:

`conjugates=`x

Using the TopSPIN strategies

Assuming that a specification exhibits a non-trivial group of static channel diagram automorphisms, TopSPIN can be used to generate a verifier with symmetry reduction algorithms by typing:

`java -jar TOPSPINPATH/TopSPIN.jar SPECIFICATION`

All being well, this should generate files called `sympan.c` and `group.o`. The `sympan.c` file can then be compiled to an executable using `gcc`:

`gcc -o sympan -DSAFETY -DNOFAIR sympan.c group.o`

Other SPIN compile-time options can be included as usual: the `-DSAFETY` and `-DNOFAIR` options are merely examples. Except when the *segmented* strategy is used, verification using the resulting `sympan` executable is performed as with the `pan` executable produced normally using SPIN. The special case of the *segmented* strategy is described below.

To specify which of the *enumeration*, *hillclimbing*, *fast* or *segmented* strategies should be used, adjust the `strategy` line of `config.txt` accordingly. The `usetranspositions` and `usestabiliserchain` options can be set to *true* or *false* depending on whether efficient application of transpositions and efficient enumeration using a stabiliser chain, respectively, is desired.

Symmetry can be specified manually via a line of the form:

`symmetryfile=FILENAME`

where `FILENAME` is the name of a file containing generators for a group which acts on processes identifiers and static channel names. Examples of such files are available online (see Section 1.2) in the archive of files used for experiments with

TopSPIN.

If the *segmented* strategy is selected then it is necessary to execute `sympan` from within GAP. To do this, copy the `sympan` executable to the TopSPIN `Common` directory; navigate to this directory; start GAP, and type:

```
Read("Verify.gap");
```

followed by:

```
Verify("sympan");
```

C.2.3 Modelling guidelines

The user study of Section 8.5 has identified some common specification features which can render a model asymmetric, as well as some limitations of SymmExtractor which require further research and implementation work. We present some modelling guidelines to help users avoid unnecessary loss of symmetry, and work around the existing limitations of SymmExtractor and TopSPIN.

Avoiding symmetry breaking features

TopSPIN is capable of exploiting *total* symmetries associated with Promela specifications. For the tool to work effectively it is important to ensure that symmetry is not destroyed by an unnecessarily asymmetric specification style.

Ensure that processes in a specification are started simultaneously. TopSPIN requires that all `run` statements are enclosed in an `atomic` block, within the `init` process. This ensures that all processes are instantiated together. Without the `atomic` block the processes would be instantiated in a fixed order, which would destroy any symmetry between processes.

Do not configure processes asymmetrically, unless faithful modelling depends on this. For example, when modelling a telephone network where individual *user* processes transition between local states *idle*, *dial*, *calling*, *ringing* and *talk* (say), ensure that all *user* processes start in the same local state, unless there is a good reason for doing otherwise. An asymmetric initial configuration may significantly reduce the size of the symmetry group associated with a specification, leading to less effective symmetry reduction.

Working within the limits of the tools

TopSPIN and SymmExtractor aim to cope with as much of the Promela language as possible. However, there are currently various features of the language which are not supported. In our experience, it is generally possible to re-model a specification so that it falls into the set of specifications accepted by the tools. We provide here a few re-modelling guidelines regarding: the use of statement separators; dynamic process creation, and the use of built-in process identifiers over user-defined identifiers.

```
proctype P() {

  /* body */

}

proctype Q() {
   ...

   run P();

   ...
}
```

Figure C.5: Skeleton Promela specification with dynamic process creation.

Due to limitations with the automatic parser generator used in the development of TopSPIN, the tool follows strictly the use of statement and declaration separators defined in the Promela grammar [92]. The grammar states that separators should be used as such, rather than as statement/declaration terminators. SPIN relaxes this restriction, allowing separators to be used optionally as terminators. When using an existing Promela specification with TopSPIN it is usual to have to modify the way in which semi-colons are used, to some extent. In particular, a semi-colon *must* follow the closing brace of an `atomic` block if the block forms part of a list of statements.

Dynamic process creation is not supported by TopSPIN. If a specification relies on dynamic process creation then it may be possible to re-model the processes as shown in Figures C.5 and C.6. Figure C.5 shows a specification which instantiates copies of proctype P dynamically. Assuming that 3 is an upper bound for the number of instances of P which should be running at any time, Figure C.6 shows an alternative way of expressing the specification. The proctype P now includes a channel parameter, and an instance of P waits until it can receive on this channel before executing its body. Its body is identical to the original, except that it includes a final `goto` statement after which it returns to its initial configuration.[2] The `init` process instantiates three copies of P, each with a distinct synchronous channel. Instead of instantiating a copy of P, the proctype Q now offers the literal value 1 to all channels on which instances of P may be listening. The example of Figures C.5 and C.6 can be adapted to handle multiple process types, with any fixed upper bound for each process type.

For symmetry to be detected, it is important for proctypes to use their built-in `_pid` variable rather than a user-defined process identifier. This is illustrated in Figures C.7 and C.8. Processes in Figure C.7 are parameterised by a *byte* identifier, which they use to index the `st` array. SymmExtractor is not yet sophisticated enough to work out the correspondence between the `id` parameter and the built-in identifier for each process. However, the specification can be converted into a form

2. This `goto` statement should really be part of an atomic block which also resets any local variables of the proctype to their initial values.

```
chan wakeup_P_1 = [0] of {bit};
chan wakeup_P_2 = [0] of {bit};
chan wakeup_P_3 = [0] of {bit}

proctype P(chan start) {

sleep:
   start?1;

   /* body */

   goto sleep
}

proctype Q() {
   ...

   if :: wakeup_P_1!1
      :: wakeup_P_2!1
      :: wakeup_P_3!1
     run P();

   ...
}

init {
   atomic {
      /* original 'run' statements */

      run P(wakeup_P_1);
      run P(wakeup_P_2);
      run P(wakeup_P_3)
   }
}
```

Figure C.6: Re-modelled Promela specification without dynamic process creation.

which SymmExtractor can handle, as shown in Figure C.8. The disadvantage here is that position 0 of the array `st` is un-used, meaning that an array of size three rather than two is required, increasing the state-vector size by one byte. On the other hand, eliminating the `id` variables reduces the state-vector by two bytes, so the re-modelling works well for this example.

```
mtype = {N,T,C}
mtype st[2]=N

proctype user(byte id) {
  do
    :: d_step { st[id]==N -> st[id]=T }
    :: d_step { st[id]==T && st[0]!=C && st[1]!=C -> st[id]=C }
    :: d_step { st[id]==C -> st[id]=N }
  od
}

init {
  atomic {
    run user(0);
    run user(1);
  }
}
```

Figure C.7: Promela specification which uses user-defined process identifiers.

```
mtype = {N,T,C} mtype st[3]=N;

proctype user() {
  do
    :: d_step { st[_pid]==N -> st[_pid]=T }
    :: d_step { st[_pid]==T && st[1]!=C && st[2]!=C -> st[_pid]=C }
    :: d_step { st[_pid]==C -> st[_pid]=N }
  od
}

init {
  atomic {
    run user();
    run user();
  }
}
```

Figure C.8: Re-modelled specification which uses the `_pid` variable.

Appendix D

Ethics Consent Form and Information Sheet

The following two pages contain copies of the consent form and information sheet for the user study described in Section 8.5.2. The forms included here are those given to students from session 2005/2006, and are adapted from a standard example [141]. The forms given to students from session 2004/2005 are very similar.

CONSENT FORM

INFORMATION SHEET

Acronyms

- **BNF** Bakus-Naur form
- **COP** Constructive orbit problem
- **COPR** Constructive orbit problem with references
- *CTL* Computation tree logic
- *CTL** Extended computation tree logic
- **ETCH** Enhanced type checker
- **GAP** Groups, algorithms and programming
- **GRAPE** Graph algorithms using permutation groups
- *LTL* Linear temporal logic
- **MRS** Modelling reactive systems course
- **nauty** No automorphisms, yes
- **SMC** Symmetry-based model checker
- **SymmSpin** Symmetric SPIN
- **SPIN** Simple Promela interpreter
- **SPIN-to-GRAPE** A tool for analysing symmetry in Promela specifications
- **TopSPIN** A symmetry reduction package for SPIN

Mathematical Notation

- $H \bullet K$ Disjoint product of H and K
- $H \wr K$ Wreath product of H and K
- $H \times K$ Direct product of H and K
- $H \rtimes K$ Semi-direct product of H and K
- $H \leq G$ H is a subgroup of G
- $H \trianglelefteq G$ H is a normal subgroup of G
- $moved(H)$ Set of points permuted by H
- $moved(\alpha)$ Set of points permuted by element α
- α^β Conjugate $\beta^{-1}\alpha\beta$ of α by β
- C_n Cyclic group of order n
- S_n Symmetric group of degree n, or isomorphic subgroup of the group associated with an n-dimensional hypercube
- $stab_G(x)$ Stabiliser of the point x in G
- $stab_G(X)$ Setwise stabiliser of X in G
- $stab^*_G(X)$ Pointwise stabiliser of X in G
- $stab_G(\mathcal{X})$ Stabiliser of partition $\mathcal{X}$ in G
- $[s]_G$ Orbit of state s under G
- $orb_G(i)$ Orbit of component identifier i under G
- Ω An orbit
- $\mathcal{O}$ A set of orbits
- G^Ω Restriction of G to act on orbit Ω
- $G^\mathcal{O}$ Restriction of G to act on the union of $\mathcal{O}$
- $\mathcal{M}$ Kripke structure
- $\mathcal{P}$ High level specification (e.g. in Promela, Promela-Lite or SMC)
- $\mathcal{CD}(\mathcal{P})$ Channel diagram associated with $\mathcal{P}$
- $\mathcal{SCD}(\mathcal{P})$ Static channel diagram associated with $\mathcal{P}$
- $Aut(\mathcal{CD}(\mathcal{P}))$ Group of channel diagram automorphisms
- $Aut(\mathcal{SCD}(\mathcal{P}))$ Group of static channel diagram automorphisms

Bibliography

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers – Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] K. Ajami, S. Haddad, and J.-M. Ilié. Exploiting symmetry in linear time temporal logic model checking: One step beyond. In B. Steffen, editor, *Proceedings of TACAS 1998*, volume 1384 of *LNCS*, pages 52–67. Springer, 1998.

[3] R. Alur and T. A. Henzinger. Logics and models of real time: A survey. In J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors, *Proceedings of REX Workshop 1991*, volume 600 of *LNCS*, pages 74–106. Springer, 1992.

[4] R. Alur and R. P. Kurshan. Timing analysis in COSPAN. In R. Alur, T. A. Henzinger, and E. D. Sontag, editors, *Proceedings of Hybrid Systems III*, volume 1066 of *LNCS*, pages 220–231. Springer, 1996.

[5] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In E. A. Boiten, J. Derrick, and G. Smith, editors, *Proceedings of IFM 2004*, volume 2999 of *LNCS*, pages 1–20. Springer, 2004.

[6] T. Ball and S. K. Rajamani, editors. *Proceedings of SPIN 2003*, volume 2648 of *LNCS*. Springer, 2003.

[7] S. Barner and O. Grumberg. Combining symmetry reduction and under-approximation for symbolic model checking. *Formal Methods in System Design*, 27(1-2):29–66, 2005.

[8] I. Beer, S. Ben-David, C. Eisner, and A. Landver. RuleBase: An industry-oriented formal verification tool. In *Proceedings of DAC 1996*, pages 655–660. ACM, 1996.

[9] S. Ben-David, T. Heyman, O. Grumberg, and A. Schuster. Scalable distributed on-the-fly symbolic model checking. In W. A. H. Jr. and S. D. Johnson, editors, *Proceedings of FMCAD 2000*, volume 1954 of *LNCS*, pages 390–404. Springer, 2000.

[10] G. Bhat, R. Cleaveland, and O. Grumberg. Efficient on-the-fly model checking for ctl^*. In *Proceedings of LICS 1995*, pages 388–397. IEEE Computer Society, 1995.

[11] B. Bollig and I. Wegener. Improving the variable ordering of obdds is np-complete. *IEEE Trans. Computers*, 45(9):993–1002, 1996.

[12] D. Bosnacki. A nested depth first search algorithm for model checking with symmetry reduction. In Peled and Vardi [137], pages 65–80.

[13] D. Bosnacki. A light-weight algorithm for model checking with symmetry reduction and weak fairness. In Ball and Rajamani [6], pages 89–103.

[14] D. Bosnacki, D. Dams, and L. Holenderski. Symmetric spin. *STTT*, 4(1):92–106, 2002.

[15] J. Bowden, G. Masters, and P. Ramsden. Influence of assessment demands on first year students' approaches to learning. In *Research and Development in Higher Education: A Forgotten Species?*, pages 397–407. Higher Education Research and Development Society of Australia, 1987.

[16] British Psychological Society. *Code of Conduct*. `http://www.bps.org.uk/`.

[17] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[18] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[19] G. Butler. *Fundamental Algorithms for Permutation Groups*, volume 559 of *LNCS*. Springer, 1991.

[20] M. Calder and A. Miller. Using SPIN for feature interaction analysis - a case study. In Dwyer [50], pages 143–162.

[21] M. Calder and A. Miller. Generalising feature interactions in email. In D. Amyot and L. Logrippo, editors, *Proceedings of FIW 2003*, pages 187–204. IOS Press, 2003.

[22] P. Cameron. *Permutation Groups*. Cambridge University Press, 1999.

[23] L. Cardelli. Type systems. In A. B. Tucker, editor, *The Computer Science and Engineering Handbook*, pages 2208–2236. CRC Press, 1997.

[24] Y. Choueka. Theories of automata on ω-tapes: A simplified approach. *J. Comput. Syst. Sci.*, 8(2):117–141, 1974.

[25] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In E. Brinksma and K. G. Larsen, editors, *Proceedings of CAV 2002*, volume 2404 of *LNCS*, pages 359–364. Springer, 2002.

[26] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Proceedings of Logics of Programs Workshop*, volume 131 of *LNCS*, pages 52–71. Springer, 1982.

[27] E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla. Symmetry reductions in model checking. In A. J. Hu and M. Y. Vardi, editors, *Proceedings of CAV 1998*, volume 1427 of *LNCS*, pages 147–158. Springer, 1998.

[28] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-

state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[29] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[30] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[31] E. M. Clarke, S. Jha, R. Enders, and T. Filkorn. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1/2):77–104, 1996.

[32] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.

[33] E. Clayberg and D. Rubel. *Eclipse: Building Commercial-Quality Plug-Ins*. Addison Wesley, 2004.

[34] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. Expressing checkable properties of dynamic systems: the bandera specification language. *STTT*, 4(1):34–56, 2002.

[35] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.

[36] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of POPL 1977*, pages 238–252. ACM, 1977.

[37] P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov. Exploiting structure in symmetry detection for CNF. In S. Malik, L. Fix, and A. B. Kahng, editors, *Proceedings of DAC 2004*, pages 530–534. ACM, 2004.

[38] C. Demartini, R. Iosif, and R. Sisto. dSPIN: A dynamic extension of SPIN. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Proceedings of SPIN 1999 Workshops*, volume 1680 of *LNCS*, pages 261–276. Springer, 1999.

[39] F. Derepas and P. Gastin. Model checking systems of replicated processes with spin. In Dwyer [50], pages 235–251.

[40] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *Proceedings of ICCD 1992*, pages 522–525. IEEE Computer Society, 1992.

[41] A. F. Donaldson and S. J. Gay. Etch: An enhanced type checking tool for promela. In P. Godefroid, editor, *Proceedings of SPIN 2005*, volume 3639 of *LNCS*, pages 266–271. Springer, 2005.

[42] A. F. Donaldson and A. Miller. Automatic symmetry detection for model checking using computational group theory. In J. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *Proceedings of FM 2005*, volume 3582 of *LNCS*, pages 481–496. Springer, 2005.

[43] A. F. Donaldson and A. Miller. A computational group theoretic symmetry reduction package for the SPIN model checker. In M. Johnson and V. Vene, editors, *Proceedings of AMAST 2006*, volume 4019 of *LNCS*, pages 374–380. Springer, 2006.

[44] A. F. Donaldson and A. Miller. Exact and approximate strategies for symmetry reduction in model checking. In J. Misra and T. Nipkow, editors, *Proceedings of FM 2006*, volume 4085 of *LNCS*, pages 541–556. Springer, 2006.

[45] A. F. Donaldson and A. Miller. Symmetry reduction for probabilistic model checking using generic representatives. In Graf and Zhang [72], pages 9–23.

[46] A. F. Donaldson and A. Miller. Extending symmetry reduction techniques to a realistic model of computation. *Electr. Notes Theor. Comput. Sci.*, 2007. To appear.

[47] A. F. Donaldson, A. Miller, and M. Calder. Comparing the use of symmetry in constraint processing and model checking. In W. Harvey and Z. Kiziltan, editors, *Proceedings of SymCon 2004*, pages 18–25, 2004.

[48] A. F. Donaldson, A. Miller, and M. Calder. Finding symmetry in models of concurrent systems by static channel diagram analysis. *Electr. Notes Theor. Comput. Sci.*, 128(6):161–177, 2005.

[49] A. F. Donaldson, A. Miller, and M. Calder. SPIN-to-GRAPE: A tool for analysing symmetry in Promela models. *Electr. Notes Theor. Comput. Sci.*, 139(1):3–23, 2005.

[50] M. B. Dwyer, editor. *Proceedings of SPIN 2001*, volume 2057 of *LNCS*. Springer, 2001.

[51] E. A. Emerson. Model checking: Theory into practice. In S. Kapoor and S. Prasad, editors, *Proceedings of FSTTCS 2000*, volume 1974 of *LNCS*, pages 1–10. Springer, 2000.

[52] E. A. Emerson, J. Havlicek, and R. J. Trefler. Virtual symmetry reduction. In *Proceedings of LICS 2000*, pages 121–131. IEEE Computer Society, 2000.

[53] E. A. Emerson, S. Jha, and D. Peled. Combining partial order and symmetry reductions. In E. Brinksma, editor, *Proceedings of TACAS 1997*, volume 1217 of *LNCS*, pages 19–34. Springer, 1997.

[54] E. A. Emerson and C.-L. Lei. Modalities for model checking: Branching time logic strikes back. *Sci. Comput. Program.*, 8(3):275–306, 1987.

[55] E. A. Emerson and A. P. Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9(1/2):105–131, 1996.

[56] E. A. Emerson and A. P. Sistla. Utilizing symmetry when model-checking under fairness assumptions: An automata-theoretic approach. *ACM Trans. Program. Lang. Syst.*, 19(4):617–638, 1997.

[57] E. A. Emerson and R. J. Trefler. From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. In L. Pierre and T. Kropf,

editors, *Proceedings of CHARME 1999*, volume 1703 of *LNCS*, pages 142–156. Springer, 1999.

[58] E. A. Emerson and T. Wahl. On combining symmetry reduction and symbolic representation for efficient model checking. In D. Geist and E. Tronci, editors, *Proceedings of CHARME 2003*, volume 2860 of *LNCS*, pages 216–230. Springer, 2003.

[59] E. A. Emerson and T. Wahl. Dynamic symmetry reduction. In N. Halbwachs and L. D. Zuck, editors, *Proceedings of TACAS 2005*, volume 3440 of *LNCS*, pages 382–396. Springer, 2005.

[60] E. A. Emerson and T. Wahl. Efficient reduction techniques for systems with many components. *Electr. Notes Theor. Comput. Sci.*, 130:379–399, 2005.

[61] A. Ferreira. Parallel and communication algorithms for hypercube multiprocessors. In A. Zomaya, editor, *Handbook of Parallel and Distributed Computing*, chapter 19, pages 568–589. McGraw-Hill, 1996.

[62] E. Gagnon and L. Hendren. SableCC, an object-oriented compiler framework. In *Proceedings of TOOLS USA 1998*, pages 140–154. IEEE Computer Society, 1998.

[63] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.4*, 2006. `http://www.gap-system.org/`.

[64] I. P. Gent, W. Harvey, and T. Kelsey. Groups and constraints: Symmetry breaking during search. In P. V. Hentenryck, editor, *Proceedings of CP 2002*, volume 2470 of *LNCS*, pages 415–430. Springer, 2002.

[65] R. Gerth. *Concise Promela Reference*, 1997. `http://www.spinroot.com/spin/Man/Quick.html`.

[66] P. Godefroid. On the costs and benefits of using partial-order methods for the verification of concurrent systems. In D. Peled, V. Pratt, and G. Holzmann, editors, *Proceedings of POMIV 1996*, volume 29 of *DIMACS series in Discrete Mathematics and Theoretical Computer Science*, pages 289–303. American Mathematical Society, 1996.

[67] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *LNCS*. Springer, 1996.

[68] P. Godefroid. Model checking for programming languages using verisoft. In *Proceedings of POPL 1997*, pages 174–186. ACM, 1997.

[69] P. Godefroid. Exploiting symmetry when model-checking software. In Wu et al. [182], pages 257–275.

[70] K. Goldberg, M. Newman, and E. Haynsworth. Combinatorial analysis. In M. Abramowitz and I. Stegun, editors, *Handbook of Mathematical Functions: with Formulas, Graphs and Mathematical Tables*. Dover Publications, 1972.

[71] S. Graf and L. Mounier, editors. *Proceedings of SPIN 2004*, volume 2989 of *LNCS*. Springer, 2004.

[72] S. Graf and W. Zhang, editors. *Proceedings of ATVA 2006*, volume 4218 of *LNCS*. Springer, 2006.

[73] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Trans. Program. Lang. Syst.*, 16(3):843–871, 1994.

[74] V. Gyuris and A. P. Sistla. On-the-fly model checking under fairness that exploits symmetry. *Formal Methods in System Design*, 15(3):217–238, 1999.

[75] S. Haddad, J.-M. Ilié, and K. Ajami. A model checking method for partially symmetric systems. In T. Bolognesi and D. Latella, editors, *Proceedings of FORTE/PSTV 2000*, volume 183 of *IFIP Conference Proceedings*, pages 121–136. Kluwer, 2000.

[76] F. Harary. The automorphism group of a hypercube. *J. UCS*, 6(1):136–138, 2000.

[77] K. Havelund and T. Pressburger. Model checking JAVA programs using JAVA PathFinder. *STTT*, 2(4):366–381, 2000.

[78] M. Hendriks, G. Behrmann, K. G. Larsen, P. Niebert, and F. W. Vaandrager. Adding symmetry reduction to Uppaal. In K. G. Larsen and P. Niebert, editors, *Proceedings of FORMATS 2003 (Revised Papers)*, volume 2791 of *LNCS*, pages 46–59. Springer, 2003.

[79] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *J. ACM*, 32(1):137–161, 1985.

[80] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In Ball and Rajamani [6], pages 235–239.

[81] I. Herstein. *Topics in Algebra*. John Wiley & Sons, 1975.

[82] J. Hillston. *A Compositional Approach to Performance Modeling.*, volume 12 of *Distinguished Dissertations in Computer Science*. Cambridge University Press, 1996.

[83] A. Hinton, M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In H. Hermanns and J. Palsberg, editors, *Proceedings of TACAS 2006*, volume 3920 of *LNCS*, pages 441–444. Springer, 2006.

[84] L. Holenderski. *The* `AdjustPan` *script*. Department of Computing Science, Eindhoven University of Technology, 2000. `http://citeseer.ist.psu.edu/holenderski00adjustpan.html`.

[85] D. F. Holt, B. Eick, and E. A. O'Brien. *Handbook of Computational Group Theory*. Chapman & Hall/CRC, 2005.

[86] G. J. Holzmann. State compression in SPIN. In R. Langerak, editor, *Proceedings of SPIN 1997*, pages 1–10, 1997.

[87] G. J. Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design*, 13(3):289–307, 1998.

[88] G. J. Holzmann and R. Joshi. Model-driven software verification. In Graf and Mounier [71], pages 76–91.

[89] G. J. Holzmann and D. Peled. An improvement in formal verification. In D. Hogrefe and S. Leue, editors, *Proceedings of FORTE 1994*, volume 6 of *IFIP Conference Proceedings*, pages 197–211. Chapman & Hall, 1995.

[90] G. J. Holzmann and A. Puri. A minimized automaton representation of reachable states. *STTT*, 2(3):270–278, 1999.

[91] G. J. Holzmann and M. H. Smith. Software model checking – extracting verification models from source code. In Wu et al. [182], pages 481–497.

[92] G. Holzmann. *The SPIN model checker: primer and reference manual*. Addison Wesley, 2004.

[93] G. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In J. Gregoire, G. Holzmann, and D. Peled, editors, *Proceedings of SPIN 1996*, volume 32 of *DIMACS series in Discrete Mathematics and Theoretical Computer Science*, pages 23–32. American Mathematical Society, 1996.

[94] P. Huber, A. M. Jensen, L. O. Jepsen, and K. Jensen. Towards reachability trees for high-level Petri nets. In G. Rozenberg, H. J. Genrich, and G. Roucairol, editors, *Proceedings of APN 1983-1984 (selected papers)*, volume 188 of *LNCS*, pages 215–233. Springer, 1985.

[95] IEEE. *IEEE Standard for a High Performance Serial Bus Std 1394-1995*, 1995.

[96] IEEE-896.1. *IEEE Standard for Futurebus+ – logical protocol specification Std 896.1-1991*. IEEE, 1992.

[97] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[98] R. Iosif. Symmetry reduction criteria for software model checking. In D. Bosnacki and S. Leue, editors, *Proceedings of SPIN 2002*, volume 2318 of *LNCS*, pages 22–41. Springer, 2002.

[99] R. Iosif. Symmetry reductions for model checking of concurrent dynamic software. *STTT*, 6(4):302–319, 2004.

[100] C. N. Ip. *State Reduction Methods for Automatic Formal Verification*. PhD thesis, Department of Computer Science, Stanford University, December 1996.

[101] C. N. Ip and D. L. Dill. Better verification through symmetry. In D. Agnew, L. J. M. Claesen, and R. Camposano, editors, *Proceedings of CHDL 1993*, volume A-32 of *IFIP Transactions*, pages 97–111. North-Holland, 1993.

[102] C. N. Ip and D. L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):41–75, 1996.

[103] D. Jackson, S. Jha, and C. Damon. Isomorph-free model enumeration: A new method for checking relational specifications. *ACM Trans. Program. Lang. Syst.*, 20(2):302–343, 1998.

[104] C. Jefferson, T. Kelsey, S. Linton, and K. Petrie. GAPLex: Generalized static

symmetry breaking. In F. Benhamou, N. Jussien, and B. O'Sullivan, editors, *Trends in Constraint Programming*, chapter 9, pages 191–205. ISTE, 2007.

[105] S. Jha. *Symmetry and Induction in Model Checking*. PhD thesis, School of Computer Science, Carnegie Mellon University, October 1996.

[106] T. A. Junttila. New orbit algorithms for data symmetries. In *Proceedings of ACSD 2004*, pages 175–184. IEEE Computer Society, 2004.

[107] B. W. Kernighan and D. Ritchie. *The C Programming Language, Second Edition*. Prentice Hall, 1988.

[108] K. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[109] L. G. Kovács. Wreath decompositions of finite permutation groups. *Bull. Austral. Math. Soc.*, 40(2):255–279, 1989.

[110] D. Kozen. Results on the propositional mu-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.

[111] R. P. Kurshan. Analysis of discrete event coordination. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Proceedings of REX Workshop 1989*, volume 430 of *LNCS*, pages 414–453. Springer, 1990.

[112] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton Series in Computer Science. Princeton University Press, 1995.

[113] M. Z. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. In J.-P. Katoen and P. Stevens, editors, *Proceedings of TACAS 2002*, volume 2280 of *LNCS*, pages 52–66. Springer, 2002.

[114] M. Z. Kwiatkowska, G. Norman, and D. Parker. Symmetry reduction for probabilistic model checking. In T. Ball and R. B. Jones, editors, *Proceedings of CAV 2006*, volume 4144 of *LNCS*, pages 234–248. Springer, 2006.

[115] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *STTT*, 1(1-2):134–152, 1997.

[116] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. L. Hennessy, M. Horowitz, and M. S. Lam. The Stanford dash multiprocessor. *IEEE Computer*, 25(3):63–79, 1992.

[117] F. Lerda and W. Visser. Addressing dynamic issues of program model checking. In Dwyer [50], pages 80–102.

[118] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of POPL 1985*. ACM, 1985.

[119] S. Lindley. Implementing deterministic declarative concurrency using sieves. In G. Blelloch, editor, *Proceedings of DAMP 2007*. ACM, 2007.

[120] S. Linton. Finding the smallest image of a set. In J. Gutierrez, editor, *Proceedings of ISSAC 2004*, pages 229–234. ACM, 2004.

[121] P. Linz. *An Introduction to Formal Languages and Automata*. Jones and Bartlett, 1986.

[122] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[123] B. D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[124] B. D. McKay. *nauty User's Guide (version 2.2)*, 2003.
`http://cs.anu.edu.au/~bdm/nauty/nug.pdf`.

[125] K. L. McMillan. A methodology for hardware verification using compositional model checking. *Sci. Comput. Program.*, 37(1-3):279–309, 2000.

[126] K. McMillan. *Symbolic Model Checking*. Kluwer, 1993.

[127] J. D. P. Meldrum. *Wreath Products of Groups and Semigroups*, volume 74 of *Pitman Monographs and Surveys in Pure and Applied Mathematics*. Longman, 1995.

[128] S. Merz. Model checking: A tutorial overview. In F. Cassez, C. Jard, B. Rozoy, and M. D. Ryan, editors, *Proceedings of MOVEP 2000*, volume 2067 of *LNCS*, pages 3–38. Springer, 2001.

[129] A. Miller, M. Calder, and A. F. Donaldson. A template-based approach for the generation of abstractable and reducible models of featured networks. *Computer Networks*, 51:439–455, 2007.

[130] A. Miller, A. F. Donaldson, and M. Calder. Symmetry in temporal logic model checking. *ACM Comput. Surv.*, 38(3), 2006. Article 8.

[131] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murϕ. In *Proceedings of S&P 1997*, pages 141–151. IEEE Computer Society, 1997.

[132] M. Müller-Olm, D. A. Schmidt, and B. Steffen. Model-checking: A tutorial introduction. In A. Cortesi and G. Filé, editors, *Proceedings of SAS 1999*, volume 1694 of *LNCS*, pages 330–354. Springer, 1999.

[133] R. Nalumasu and G. Gopalakrishnan. Explicit-enumeration based verification made memory-efficient. In S. D. Johnston, editor, *Proceedings of CHDL/ASP-DAC/VLSI 1995*, pages 617–622. IEEE Computer Society, 1995.

[134] W. Nam and R. Alur. Learning-based symbolic assume-guarantee reasoning with automatic decomposition. In Graf and Zhang [72], pages 170–185.

[135] D. Peled. Combining partial order reductions with on-the-fly model-checking. *Formal Methods in System Design*, 8(1):39–64, 1996.

[136] D. Peled. *Software Reliability Methods*. Springer, 2001.

[137] D. Peled and M. Y. Vardi, editors. *Proceedings of FORTE/PSTV 2002*, volume 2529 of *LNCS*. Springer, 2002.

[138] G. L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.

[139] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[140] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI Series. Series F: Computer and System Science*, pages 123–144. Springer, 1985.

[141] H. Purchase. *Example participant consent form.* `http://www.dcs.gla.ac.uk/~/ethics/`.

[142] H. Purchase. Student compliance with ethical guidelines: The Glasgow ethics approach. *ITALICS*, 5(2), 2006. Article 1.

[143] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Proceedings of the International Symposium on Programming*, volume 137 of *LNCS*, pages 337–351. Springer, 1982.

[144] M. Rangarajan, S. Dajani-Brown, K. Schloegel, and D. Cofer. Analysis of distributed Spin applied to industrial-scale models. In Graf and Mounier [71], pages 267–285.

[145] A. Richards. *The Codeplay Sieve C++ Parallel Programming System*, 2006. `http://www.codeplay.com`.

[146] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *Proceedings of ESEC/FSE 2003*, pages 267–276. ACM, 2003.

[147] Robby, M. B. Dwyer, J. Hatcliff, and R. Iosif. Space-reduction strategies for model checking dynamic software. *Electr. Notes Theor. Comput. Sci.*, 89(3):499–517, 2003.

[148] J. S. Rose. *A Course in Group Theory*. Dover Publications, 1994.

[149] G. Rota. The number of partitions of a set. *Amer. Math Monthly*, 71:498–504, 1964.

[150] S. Russel and P. Norvig. *Artificial Intelligence, a Modern Approach*. Prentice Hall, 1995.

[151] J. Rutten, M. Kwiatkowska, G. Norman, and D. Parker. *Mathematical Techniques for Analysing Concurrent and Probabilistic Systems*, volume 23 of *CRM Monograph Series*. American Mathematical Society, 2004.

[152] T. C. Ruys. Low-fat recipes for spin. In K. Havelund, J. Penix, and W. Visser, editors, *Proceedings of SPIN 2000*, volume 1885 of *LNCS*, pages 287–321. Springer, 2000.

[153] T. C. Ruys. *Towards Effective Model Checking*. PhD thesis, Department of Computer Science, Twente University, March 2001.

[154] T. C. Ruys and G. J. Holzmann. Advanced spin tutorial. In Graf and Mounier [71], pages 304–305. Diagrams appear in accompanying slides.

[155] P. Saffrey. *Optimising Communication Structure for Model Checking*. PhD thesis, Department of Computing Science, University of Glasgow, July 2003.

[156] P. Saffrey and M. Calder. Optimising communication structure for model checking. In M. Wermelinger and T. Margaria, editors, *Proceedings of FASE 2004*, volume 2984 of *LNCS*, pages 310–323. Springer, 2004.

[157] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[158] H. Schildt. *Java: The Complete Reference, J2SE 5 Edition*. McGraw-Hill, 2004.

[159] A. Seress. *Permutation Group Algorithms*. Cambridge University Press, 2003.

[160] S. Seshia. Lecture notes on computer-aided verification. Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 2006.

[161] A. P. Sistla. Employing symmetry reductions in model checking. *Computer Languages, Systems & Structures*, 30(3-4):99–137, 2004.

[162] A. P. Sistla and P. Godefroid. Symmetry and reduced symmetry in model checking. *ACM Trans. Program. Lang. Syst.*, 26(4):702–734, 2004.

[163] A. P. Sistla, V. Gyuris, and E. A. Emerson. SMC: a symmetry-based model checker for verification of safety and liveness properties. *ACM Trans. Softw. Eng. Methodol.*, 9(2):133–166, 2000.

[164] A. P. Sistla, X. Wang, and M. Zhou. Checking extended CTL properties using guarded quotient structures. In *Proceedings of SEFM 2004*, pages 87–94. IEEE Computer Society, 2004.

[165] L. H. Soicher. GRAPE: a system for computing with graphs and groups. In L. Finkelstein and W. Kantor, editors, *Groups and Computation*, volume 11 of *DIMACS series in Discrete Mathematics and Theoretical Computer Science*, pages 287–291. American Mathematical Society, 1993.

[166] L. H. Soicher. Computing with graphs and groups. In L. W. Beineke and R. J. Wilson, editors, *Topics in Algebraic Graph Theory*, pages 250–266. Cambridge University Press, 2004.

[167] P. H. Starke. Reachability analysis of Petri nets using symmetries. *Systems Analysis–Modelling–Simulation*, 8(4/5):293–303, 1991.

[168] A. S. Tanenbaum and M. van Steen. *Distributed Systems – Principles and Paradigms*. Prentice Hall, 2002.

[169] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[170] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 1994.

[171] F. Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3):121–189, 1995.

[172] A. Valmari. Stubborn sets for reduced state space generation. In G. Rozenberg, editor, *Proceedings of APN 1989*, volume 483 of *LNCS*, pages 491–515. Springer, 1991.

[173] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Proceedings of LICS 1986*, pages 332–344. IEEE Computer Society, 1986.

[174] M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Inf. Comput.*, 115(1):1–37, 1994.

[175] B. Vergauwen and J. Lewi. A linear local model checking algorithm for CTL. In E. Best, editor, *Proceedings of CONCUR 1993*, volume 715 of *LNCS*, pages 447–461. Springer, 1993.

[176] W. Visser, K. Havelund, G. P. Brat, and S. Park. Model checking programs. In P. Alexander and P. Flener, editors, *Proceedings of ASE 2000*, pages 3–12. IEEE Computer Society, 2000.

[177] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl.* O'Reilly, 2000.

[178] F. Wang and K. Schmidt. Symmetric symbolic safety-analysis of concurrent software with pointer data structures. In Peled and Vardi [137], pages 50–64.

[179] O. Wei, A. Gurfinkel, and M. Chechik. Identification and counter abstraction for full virtual symmetry. In D. Borrione and W. J. Paul, editors, *Proceedings of CHARME 2005*, volume 3725 of *LNCS*, pages 285–300. Springer, 2005.

[180] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proceedings of POPL 1986*, pages 184–193. ACM, 1986.

[181] P. Wolper and D. Leroy. Reliable hashing without collosion detection. In C. Courcoubetis, editor, *Proceedings of CAV 1993*, volume 697 of *LNCS*, pages 59–70. Springer, 1993.

[182] J. Wu, S. T. Chanson, and Q. Gao, editors. *Proceedings of FORTE 1999*, volume 156 of *IFIP Conference Proceedings*. Kluwer, 1999.

[183] S. Yovine. KRONOS: A verification tool for real-time systems. *STTT*, 1(1-2):123–133, 1997.