Ann Math Artif Intell (2009) 57:1–35
DOI 10.1007/s10472-009-9171-4

On the constructive orbit problem

Alastair F. Donaldson · Alice Miller

Published online: 29 December 2009
© Springer Science+Business Media B.V. 2009

Abstract Symmetry reduction techniques aim to combat the state-space explosion problem for model checking by restricting search to representative states from equivalence classes with respect to a group of symmetries. The standard approach to representative computation involves converting a state to its minimal image under a permutation group G, before storing the state. This is known as the *constructive orbit problem* (COP), and is *NP* hard. It may be possible to solve the COP efficiently if G is known to have certain structural properties: in particular if G is isomorphic to a full symmetry group, or G is a disjoint/wreath product of subgroups. We extend existing results on solving the COP efficiently for fully symmetric groups, and investigate the problem of automatically classifying an arbitrary permutation group as a disjoint/wreath product of subgroups. We also present an approximate COP strategy based on local search, and some computational group-theoretic optimisations to improve the basic approach of solving the COP by symmetry group enumeration. Experimental results using the TopSpin symmetry reduction package, which interfaces with the computational group-theoretic system Gap, illustrate the effectiveness of our techniques.

Keywords Symmetry reduction · Model checking · Orbit problem · Computational group theory · Wreath product

Mathematics Subject Classifications (2000) 68N30 · 68U07 · 20B25 · 20E22

Alastair F. Donaldson is supported by EPSRC grant EP/G051100. Alice Miller is supported by EPSRC grant EP/E032354.

A. F. Donaldson (✉)
Oxford University Computing Laboratory, Oxford, UK
e-mail: alastair.donaldson@comlab.ox.ac.uk

A. Miller
Department of Computing Science, University of Glasgow, Glasgow, UK

1 Introduction

A major problem in software design is to prove that a complex system will behave as expected. For example, errors due to the interleaving behaviour of concurrent processes of a system may not be detected until testing is performed on a prototype, by which time they are difficult—and expensive—to correct.

Model checking [8, 23] is a popular, automated technique which allows errors in a software-controlled system to be found early on in the design process. The modeller writes a description that captures the essential aspects of the system (*e.g.* communication) but ignores implementation detail. The system description is typically constructed via a specification language (such as Promela [23] or the Reactive Modules language [1]) and converted to a finite-state model, usually a Kripke structure, by a model checker such as SPIN [23] or SMV [31]. Basic properties such as lack of deadlock, or more complicated properties expressed in a temporal logic, can then be checked by searching the state-space associated with the model.

The main barrier to the widespread use of model checking is the *state-space explosion problem*: as the number of processes in a system increases, the size of the state-space associated with a model of the system grows combinatorially, becoming too large to search exhaustively. A significant area of model checking research focusses on techniques to ameliorate this problem, either by storing states in a compact form (*e.g.* symbolically [31]), or by analysing the system specification to determine tighter bounds on the number of states which must be checked to determine the truth of a given property.

One popular technique of the latter type is *symmetry reduction* (see *e.g.* [7, 16, 24], or [33] for a recent survey). Replication of processes in a concurrent system frequently induces replication, or *symmetry*, in the state-space associated with a model of the system. Symmetries between indistinguishable processes form a *group* which acts on the state-space, partitioning it into equivalence classes called *orbits*. To model check a particular class of temporal property it is sufficient to search one state per orbit. As long as it is possible to obtain information about symmetry from a specification without constructing the associated state-space, symmetry reduction can result in more efficient verification.

Given a group G, the standard approach to ensuring that equivalent states are recognised during search is to convert a newly encountered state s into $\min[s]_G$, the *smallest* state in the orbit of s (under a suitable total ordering) before it is stored. Thus the crux of exploiting symmetry is being able to efficiently compute $\min[s]_G$. This is known as the *constructive orbit problem* (COP), and has been shown to be *NP* hard [6]. However, it is possible to solve the COP in polynomial time for full symmetry groups, and groups which decompose as disjoint or wreath products of subgroups [6]. Alternatively, it is safe (though not memory-optimal) to *approximate* the COP by computing a small number of representatives from each equivalence class [2, 6, 7]. On the other hand, if G is not too large then $\min[s]_G$ can be computed by brute-force enumeration of G [2, 13].

After a discussion of related work (Section 2) and a summary of some necessary background on model checking, group theory and symmetry reduction (Section 3), we provide some optimisations based on standard computational group theory (CGT) for efficient enumeration of an arbitrary group (Sections 4.1 and 4.2). We then present a polynomial time strategy for solving the COP when G is isomorphic

to the symmetric group S_m for some $m > 0$ (Section 4.3). This generalises a result on symmetric groups presented in [6]. To handle large, arbitrary groups we propose COP strategy which uses *local search* to approximate $\min[s]_G$ (Section 4.4).

We then turn our attention to groups which decompose as disjoint or wreath products of subgroups. Techniques for exploiting these kinds of group are only useful for automatic model checking if it is possible to determine, before search, whether an arbitrary group does in fact decompose as an appropriate product of subgroups. We provide two techniques for computing disjoint product decompositions: an efficient, sound, incomplete approach which works directly with symmetry group generators, and a complete approach which, in the worst case, runs in exponential time (Section 5). We then present an algorithm for automatically determining whether an arbitrary transitive group decomposes as a wreath product, and describe an extension to this algorithm for intransitive wreath products (Section 5).

The paper concludes with a discussion of experimental results using the TopSpin symmetry reduction package for the Spin model checker, which show that solving the constructive orbit problem efficiently using our theoretical techniques can result in significant savings, both in memory and verification time (Section 8).

2 Related work

The COP was introduced in [6], together with basic techniques for solving the COP efficiently when G is a fully symmetric group, or a disjoint/wreath product of subgroups. This paper puts these results into practice by extending the class of fully symmetric groups for which the COP can be efficiently solved, and providing automatic techniques for *detecting* the case where G is a disjoint/wreath product of subgroups. The convention of taking $\min[s]_G$ as a representative for $[s]$ has been relatively widely adopted [2, 6, 12, 13, 29], but is not universal. The symmetry-based model checker (SMC) [37], for example, uses the first state encountered from a given orbit as the representative of that orbit.

The idea of easing the complexity of symmetry reduction by using multiple representatives from each orbit was introduced in the context of symbolic model checking [7]. Multiple representatives in explicit state model checking are used by the SymmSpin and SMC packages (see [2] and [37] respectively) for the special case of fully symmetric groups. The local search strategy presented here is more general, allowing symmetry reduction via multiple representatives for an arbitrary group.

The results of this paper are presented in the context of a simple model of computation where the local state of each process in a concurrent system is abstracted into an integer value, and processes do not hold references to one another. In [14] we have considered the problem of extending the model of computation to allow inter-process references.

An approach to symmetry breaking in constraint programming requires an efficient solution to a COP-related problem: determining whether a given partial assignment of variables in the search tree is lexicographically least in its orbit. The key difference between this problem and the COP is that the COP requires the minimal orbit representative to be explicitly computed; the approach of [25] merely requires a yes/no answer as to whether a given assignment is minimal. An efficient implementation of the techniques presented in [25] uses a computational

group-theoretic algorithm for finding the smallest image of a set of points under a permutation group acting on the points [30]. This problem can be shown to be equivalent to the COP, thus the algorithm of [30] provides a general COP solution. The main difference between our work and that of [30] is that our techniques allow polynomial-time solutions to the COP for certain types of group and an approximate solution to the COP for arbitrary groups, whereas the approach in [30] provides a non-polynomial time (but often efficient) exact solution to the COP for arbitrary groups.

Some of the results in this paper were presented, in a preliminary form, in [13].

3 Background

We provide an introduction to model checking (Section 3.1), a summary of some group-theoretic definitions and theorems (Section 3.2), and an introduction to symmetry reduction theory (Section 3.3).

3.1 Model checking

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) $CTL^\star$ [8].

We use a simple model to represent the computation of a system comprised of n communicating processes, interleaving concurrently [6, 17]. Let $I = \{1, 2, \ldots, n\}$ be the set of process identifiers (for some $n > 0$), and let $\emptyset \neq L \subset \mathbb{Z}$ denote a finite set of possible local states of the processes. A Kripke structure is defined in terms of L and n as follows:

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

1. $S = L^n$ 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 $i > 0$, $(s_{i-1}, s_i) \in R$. A state $s \in S$ is *reachable* if there is a path $s_0, s_1, \ldots, s, \ldots$ in $\mathcal{M}$ where $s_0 \in S_0$. When checking properties of a Kripke structure we are interested only in its reachable states.

Figure 1 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. For convenience we denote these local states N, T and C rather than using three integers. 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 use A_i to assert that process i is in local state A. 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

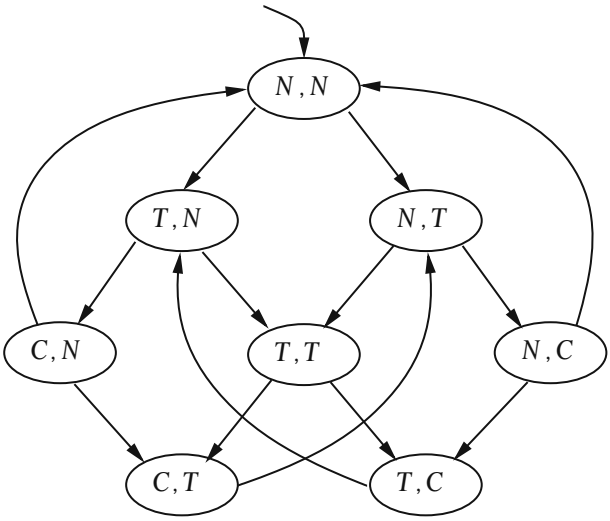

Fig. 1 Kripke structure for two-process mutual exclusion

(indicated by an incoming edge with no predecessor state in Fig. 1). In the initial state both processes are in the neutral location.

To express properties of Kripke structures it is typical to use the branching time temporal logic $CTL^\star$ (or one of its sub-logics). The set of $CTL^\star$ formulas are defined inductively over propositions A_i, where $A \in L$ and $1 \leq i \leq n$. The quantifiers **A** and **E** are used to denote *for all paths*, and *for some path* respectively. In addition, **X**, **U**, **F** and **G** represent the standard *next-time*, *strong until*, *eventually* and *always* operators. For a $CTL^\star$ property ϕ, we write $\mathcal{M}, s \models \phi$ if ϕ holds at state s of $\mathcal{M}$, and $\mathcal{M} \models \phi$ if ϕ holds for every initial state of $\mathcal{M}$. For the scope of this paper it is unnecessary to formally present the syntax and semantics of $CTL^\star$, which are detailed in [8]. We can express the mutual exclusion property in $CTL^\star$ as: $\mathbf{AG}(\neg(C_1 \wedge C_2))$. The Kripke structure of Fig. 1 clearly satisfies this property as (C, C) is not a reachable state.

3.1.1 Spin *and Promela*

In Section 8 we provide experimental results using the Spin model checker [23]. The Spin tool allows verification by model checking for specifications written in Promela, a high level language geared towards modelling communication protocols associated with concurrent, distributed systems. A Promela specification consists of a series of parameterised process definitions, a set of global variables and communication channels, an initialisation process which instantiates multiple copies of the parameterised processes.

The Spin tool explores the state-space associated with a Promela specification, which is generated by considering the interleaving behaviour of these concurrent processes. Spin provides model checking algorithms to verify deadlock freedom, absence of assertion violations, and more complex temporal properties expressed in LTL (a sub-logic of $CTL^\star$). To manage state-space explosion, Spin incorporates a variety of state compression and state-space reduction techniques. Partly due to the success of these techniques, Spin has been widely used in industry and academia for reasoning about communications protocols. Nevertheless, state-space explosion

remains a problem when attempting to verify complex specifications. The TopSpin tool [12], which incorporates the symmetry reduction techniques described in this paper, provides further state-space reduction by exploiting symmetry.

3.2 Basic group theory

We present some basic definitions and results from group theory, and introduce the computational group theoretic package Gap.

Definition 2 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 3 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 the notation $\langle \alpha_1, \alpha_2, \ldots, \alpha_k \rangle$ to denote the group $\langle \{\alpha_1, \alpha_2, \ldots, \alpha_k\} \rangle$.

For any group G, if $X \subseteq G$ 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 4 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 H 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.

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

Definition 5 Let $(G_1, \circ)$, $(G_2, \star)$ be groups. A *homomorphism* from G_1 to G_2 is a mapping $\theta : G_1 \to G_2$ 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 [21]. However, two isomorphic groups may have distinct actions on a set (see below).

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 1 *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 $\theta(G_1) = \{\theta(\alpha) : \alpha \in G_1\}$.*

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 set of permutations $P \subseteq Sym(X)$, we use $moved(P)$ to denote the subset of X which is affected by P: $moved(P) = \{x \in X : \alpha(x) \neq x \text{ for some } \alpha \in P\}$. For $\alpha \in Sym(X)$ we define $moved(\alpha) = moved(\{\alpha\})$. 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 $t > 0$, $2 \leq s_i \leq |X|$ $(1 \leq i \leq t)$, $a_{i,j} \in X$ $(1 \leq i \leq t,\ 1 \leq j \leq s_i)$, 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 $\alpha(x) = a_{i,j'}$ where $j' = j + 1$ if $j < s_i$ and $j' = 1$ if $j = s_i$; if $x \neq a_{i,j}$ for any i and j then $\alpha(x) = x$.

Definition 6 Let X be a non-empty set, $G \leq Sym(X)$, $x \in X$ and $Y \subseteq X$.

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

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

The stabiliser $stab_G(x)$ of a single point under G can be computed in low-degree polynomial time using standard CGT techniques, as can the pointwise stabiliser $stab^*_G(Y)$ for a set Y of points. With current methods, the setwise stabiliser $stab_G(Y)$ cannot be computed in polynomial time: this subgroup is computed via backtrack search using a base and strong generating set [3].

Definition 7 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 $\Omega \subseteq X$ is non-trivial if $|\Omega| > 1$.

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.

An important class of permutation groups are the symmetric groups:

Definition 8 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 on n points.

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 [35].

Definition 9 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 2 *Let G act on X. Then to each $\alpha \in G$ there corresponds an element $\rho_\alpha \in Sym(X)$ defined by $\rho_\alpha(x) = \alpha(x)$ for all $x \in X$, 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.

Certain groups can be described as products of their subgroups. Two important kinds of product are disjoint and wreath products, which we introduce in Definitions 10 and 11 respectively.

Definition 10 Let $G \leq Sym(X)$, where X is a non-empty set. Suppose $H_1, H_2, \ldots, H_k$ $(k > 1)$ are subgroups of G such that $G = \{\alpha_1\alpha_2 \ldots \alpha_k : \alpha_i \in H_i\ (1 \leq i \leq k)\}$ and $moved(H_i) \cap moved(H_j) = \emptyset$ 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 set of 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 $1 \leq i \leq k$.

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.

The following definition of the *wreath product* of two permutation groups is adapted from a definition given in [26], and allows us to identify an existing group as a wreath product of subgroups.

Definition 11 Let $H \leq S_m$ and $K \leq S_d$ for some $m, d > 0$. Let X be a set with $|X| = md$, and $\{X_1, X_2, \ldots, X_d\}$ a partition of X into equal-sized subsets, where X_i has the form $X_i = \{x_{i,1}, x_{i,2}, \ldots, x_{i,m}\}$ for some $x_{i,j} \in X$ $(1 \leq i \leq d, 1 \leq j \leq m)$. 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 $1 \leq i \leq d$, so that $x = x_{i,t}$ for some $1 \leq t \leq m$. 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 $1 \leq i \leq d$, suppose $x \in X_j$ for some $1 \leq j \leq d$, so that $x = x_{j,t}$ for some $1 \leq t \leq m$. Define $\alpha(x) = x$ if $i \neq j$ and $\alpha(x) = x_{j,\alpha(t)}$ otherwise. Let σ_i be the permutation representation corresponding to this action of H on X.

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

Note that we must specify the partition $\mathcal{X} = \{X_1, X_2, \ldots, X_d\}$ when reasoning about a 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. The order of $H \wr K$ depends on the orders of H and K, and the degree of K:

Theorem 3 *If G is a wreath product $H \wr K$ then $|G| = |H|^d \times |K|$, where d is the degree of K.*

Definition 11 describes a wreath product with the *imprimitive action* [5]. 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 6 are specific to the imprimitive action.

GAP (groups, algorithms and programming) [18] 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 detailed in [3, 22]. We use GAP to present examples throughout the paper, and our TopSPIN symmetry reduction package (see Section 8) interfaces with GAP to perform CGT calculations.

3.3 Symmetry reduction

Definition 12 Let $\mathcal{M} = (S, S_0, R)$ be a Kripke structure. 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 $s, t \in S$, $(s, t) \in R \Leftrightarrow (\alpha(s), \alpha(t)) \in R$
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 usually involve the permutation of process identifiers throughout all states of the model. In this case there is a group G which permutes a (typically small) set of process identifiers, and an action of G on S (see Definition 9). Let ρ be the permutation representation corresponding to this action (see Theorem 2). 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 7) can be used to construct a *quotient* Kripke structure $\mathcal{M}_G$ as follows:

Definition 13 The quotient Kripke structure $\mathcal{M}_G$ of $\mathcal{M}$ with respect to G is a tuple $\mathcal{M}_G = (S_G, S^0_G, R_G)$ where:

- $S_G = \{rep_G(s) : s \in S\}$ (where $rep_G(s)$ is a unique representative of $[s]_G$)
- $S^0_G = \{rep_G(s) : s \in S_0\}$
- $R_G = \{(rep_G(s), rep_G(t)) : (s, t) \in R\}$.

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 processes, 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 Fig. 1, 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 Fig. 2.

It can be shown that a model and its quotient model satisfy the same *symmetric* $CTL^\star$ formulas [7, 16]. A $CTL^\star$ formula ϕ is symmetric, or invariant, with respect to G if for every maximal propositional sub-formula f appearing in ϕ, and for every $\alpha \in G, \mathcal{M}, s \models f \Leftrightarrow \mathcal{M}, \alpha(s) \models f$.

Theorem 4 *If $\mathcal{M}$ and $\mathcal{M}_G$ denote a model and its quotient model with respect to a group G respectively, then $\mathcal{M} \models \phi \Leftrightarrow \mathcal{M}_G \models \phi$, for every symmetric $CTL^\star$ formula ϕ.*

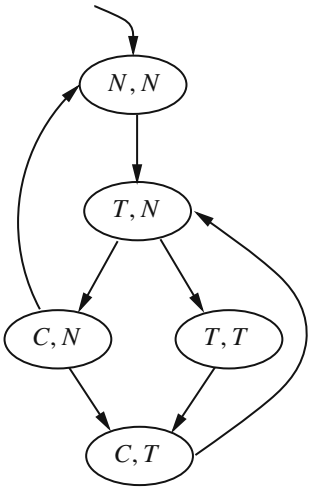

Fig. 2 Quotient Kripke structure for two-process mutual exclusion

Note that Theorem 4 can be used to find errors as well as prove properties since the implication is two-way. Consider the two-process mutual exclusion property $\mathbf{AG}(\neg(C_1 \wedge C_2))$. 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 Fig. 1) satisfies ϕ_1 if and only if the quotient structure $\mathcal{M}_G$ (represented by Fig. 2) does. Therefore, to check the mutual exclusion property, it is sufficient to check the quotient model only. Note that $\mathcal{M}_G$ also satisfies 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 $\mathcal{M} \not\models \phi_2$.)

Algorithm 1 (adapted from [16, 24]), 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. Using Algorithm 1 it may be possible to build a quotient

Algorithm 1 Algorithm to construct a quotient Kripke structure

```
S_G := {rep_G(s) : s ∈ S_0}
unexplored := {rep_G(s) : s ∈ S_0}
R_G := ∅
while unexplored ≠ ∅ do
    remove a state s from unexplored
    for all successor states t of s do
        add s → rep_G(t) to R_G
        if rep_G(t) ∉ S_G then
            add rep_G(t) to S_G
            add rep_G(t) to unexplored
        end if
    end for
end while
```

structure even though the associated unreduced structure is intractably large. To determine a unique element $rep_G(s)$ for each orbit $[s]_G$, we require a *canonicalisation* function. Methods for efficient canonicalisation functions are the focus of our paper, and are discussed in detail below.

3.3.1 Symmetry detection

In this paper, we are concerned with techniques for exploiting process symmetries during model checking, rather than detecting symmetry before search. Structural symmetries of a model $\mathcal{M}$ are typically inferred by extracting a *communication graph* from the initial specification. The vertex set of this graph is the set I, representing the processes of the system. Provided that the specification obeys certain restrictions so that processes of the same type are not explicitly distinguished, automorphisms of the communication graph induce automorphisms of $\mathcal{M}$. Since the communication graph is typically small, these automorphisms can be computed automatically using a package such as *saucy* [9]. Practical examples of communication graphs include the *static channel diagram* of a Promela specification [11], and the *coloured hypergraph* [6] of a shared variable concurrent program.

For illustration, throughout this paper we consider a system with a three-tiered architecture, illustrated by the communication graph of Fig. 3. Let $\mathcal{M}_{3T}$ be a model of the system. Using the *saucy* program, we compute G_{3T}, the automorphism group of the communication graph in terms of generators:

$$G_{3T} = \langle (1\ 2), (2\ 3), (4\ 5), (5\ 6), (7\ 8), (8\ 9), (10\ 11),$$
$$(12\ 13)(1\ 4)(2\ 5)(3\ 6), (13\ 14)(4\ 7)(5\ 8)(6\ 9) \rangle.$$

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.

3.3.2 The constructive orbit problem

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* [7], 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.

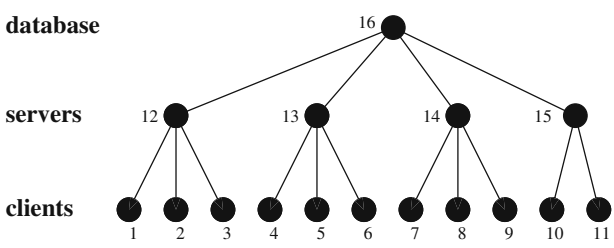

Fig. 3 Communication structure for a three-tiered architecture

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 $u \equiv_G t$), the test $rep_G(t) \in S_G$ returns true. As a consequence, there is no need for the successors of t to be added to the *unexplored* set: these successor states are all equivalent to successors of $rep_G(u)$, which have already been added to *unexplored*.

An element $\alpha \in S_n$ acts naturally on a state $s = (x_1, x_2, \ldots, x_n) \in L^n$ as follows: $\alpha(s) = (x_{\alpha^{-1}(1)}, x_{\alpha^{-1}(2)}, \ldots, x_{\alpha^{-1}(n)})$.[1] This action makes sense provided the local state of a process does not include variables which take process identifiers as values. This complex situation is treated in [14].

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 14 The constructive orbit problem (COP) [6, 26] 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 5 [6, 26] *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: fully symmetric groups, disjoint products and wreath products [6]. We investigate these kinds of symmetry group further in Sections 4.3, 5 and 6 respectively. Alternatively, 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. We propose one such approximate solution in Section 4.4.

In the worst case, we can solve the COP exactly by enumerating over the group G. While this is impractical for large groups, it can be a practical solution for smaller symmetry groups.

4 Exploiting basic symmetry groups

In this section we discuss two optimisations to the basic enumeration strategy which exploit the decomposition of permutations into transpositions and the use of stabiliser chains respectively. We then present a result which generalises the class of

[1] It may appear more straightforward to define $\alpha(s)$ in terms of elements $x_{\alpha(i)}$ rather than $x_{\alpha^{-1}(i)}$ ($1 \leq i \leq n$). However, our definition is intuitive in practice, *e.g.* if $s = (7, 5, 4)$ and $\alpha = (1\ 2\ 3)$ then $\alpha(s) = (4, 7, 5)$ as one would expect. Defining $\alpha(s)$ in terms of elements $x_{\alpha(i)}$ would yield $\alpha(s) = (5, 4, 7)$, which is less intuitive.

fully symmetric groups for which the COP can be efficiently solved, and propose an approximate COP strategy for arbitrary groups, based on local search.

4.1 Efficient application of permutations

Consider the problem of applying a permutation α to a state s, *i.e.*, computing $\alpha(s)$ where s has the form $(x_1, x_2, \ldots, x_n)$. Direct application of α to s 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 1 *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: $\alpha = (a_1\ a_2)(a_1\ a_3)\ldots(a_1\ a_m)$ [21]. 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. □

Note that expressing and applying permutation α as a list of up to $n-1$ transpositions does not change the complexity of applying α to a state, which remains an $O(n)$ operation. However, in practice many permutations can be expressed as a product of a small number of transpositions, in which case the transposition method is more efficient than direct application. In Section 8 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.

4.2 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 $u_k u_{k-1} \ldots u_2$, where $u_i \in U^{(i)}$ $(1 < i \leq k)$ [3]. 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 2). To ensure efficient application of permutations, the coset representatives are stored as a list of transpositions, applied in succession, as described in Section 4.1. Applying sequences

Algorithm 2 Computing $\min[s]_G$ using a stabiliser chain

```
min[s]_G := s
for all u_2 ∈ U_2 do
  s_2 := u_2(s)
  for all u_3 ∈ U_3 do
    s_3 := u_3(s_2)
    ⋮
    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
```

of transpositions works particularly well here since each coset representative can typically be represented as a product of a small number of transpositions.

GAP provides functionality to efficiently compute a stabiliser chain and associated coset representatives for an arbitrary permutation group (as discussed in Section 3.2, computing point stabilisers is a polynomial-time operation). 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 8 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 [3, 18], and have been utilised in symmetry breaking approaches for constraint programming [19]. We are, to our knowledge, the first to apply these techniques to model checking.

We have used stabiliser chains to efficiently enumerate elements of a group, which requires CGT calculations to be performed before search, to work out sets of coset representatives for use by Algorithm 2. Further gains in efficiency may be possible by using CGT methods *during* search. The orbit of a state s can be significantly smaller than associated symmetry group G, in which case it would be more efficient to compute the orbit of s directly via CGT methods, which use a base and strong generating set [3] instead of relying on enumeration. We intend to investigate this optimisation, which would require the integration of non-trivial CGT algorithms with the model checking process, as future work.

4.3 Minimising sets for G if $G \cong S_m$ $(m \leq n)$

For systems where there is full symmetry between processes, the smallest state in the orbit of $s = (x_1, x_2, \ldots, x_n)$ can be computed by *sorting* the tuple s lexicographically. [2, 6]. For example, for a system with four processes, 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 Fig. 3. 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* processes 12, 13 and 14, with their associated blocks of *client* processes. 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*. Using terminology from [17], 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 the minimising set maps the state to a smaller image, thus the minimal element has been found. This approach is formalised by Algorithm 3.

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 computed. This minimising set is derived from the swap permutations used in a selection-sort algorithm. As discussed in Definition 7, 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 process identifiers.

Theorem 6 *Suppose that, for each $x \in I$ such that $orb_G(x)$ is non-trivial (see Definition 7), $stab_G(x)$ fixes exactly one element from each non-trivial orbit of G acting on I, and that $G \cong S_m$, where $m = |orb_G(y)| > 1$ for some $y \in I$. Then there is an isomorphism $\theta : S_m \to G$ such that $\{\theta((i\ j)) : 1 \leq i < j \leq m\}$ 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 $\theta((i\ j)) \in G$

Algorithm 3 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
```

for any $1 \leq i < j \leq m$. The element $\theta((i\ j))$ 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 $\theta((i'\ j'))(s) < s$. Hence s is minimal in its orbit iff $\theta((i\ j))(s) \geq s$ for all $i < j$. So the set $\{\theta((i\ j)) : 1 \leq i < j \leq m\}$ is a minimising set for G. □

Note that the minimising set is much smaller than G, and the conditions of Theorem 6 can be easily checked using GAP. Although testing two arbitrary groups for isomorphism can be expensive, 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 performed efficiently using the GAP function `IsNaturalSymmetricGroup(G)`.

It may seem that the conditions of Theorem 6 are unnecessary, and that, given any isomorphism $\theta : S_m \to G$, the set $\{\theta((i\ j)) : 1 \leq i < j \leq m\}$ is a minimising set for G. However, consider the group G below, which is a subgroup of the symmetry group of a hypercube:

$$G = \langle (1\ 2)(5\ 6)(9\ 10)(13\ 14), (1\ 2\ 4\ 8)(3\ 6\ 12\ 9)(5\ 10)(7\ 14\ 13\ 11) \rangle \leq S_{14}$$

The group G is isomorphic to S_4: an isomorphism $\theta : S_4 \to G$ is defined on generators by $\theta((1\ 2\ 3\ 4)) = (1\ 2\ 4\ 8)(3\ 6\ 12\ 9)(5\ 10)(7\ 14\ 13\ 11)$, $\theta((1\ 2)) = (4\ 8)(5\ 9)(6\ 10)(7\ 11)$. However, 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 $\{\theta((i\ j)) : 1 \leq i < j \leq 4\}$.

Theorem 7 *If G satisfies the conditions of Theorem* 6 *and $X = \{\theta((i\ j)) : 1 \leq i < j \leq m\}$ then* $\min[s]_G$ *can be computed in $O(m^3)$ time for any $s \in L^n$, using Algorithm* 3.

Proof Clearly $|X| = |\{\theta((i\ j)) : 1 \leq i < j \leq m\}| = m(m-1)/2$.

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. Column entries can be ordered lexicographically. An element $\theta((i\ j))$ 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 $\min\{\alpha(s) : \alpha \in X\} = \theta((i\ j))(s)$. Hence, after the first iteration of the outer loop of Algorithm 3, the first (left-most) column entry for state $\min'$ is in the smallest possible position. 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, the outer loop of Algorithm 3 iterates m times. Since each iteration of this loop involves iterating over a set of size $m(m-1)/2$ the complexity of Algorithm 3 is $O(m^3)$ as required. □

Each iteration of the outer loop of Algorithm 3 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 4 works similarly, but updates the current minimum *every*

Algorithm 4 Optimised 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
```

time an element of X is found which yields a smaller image. We have found that this works better in practice.

4.4 Local search for unclassified groups

If G is a 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 Sections 4.1 and 4.2. If no minimising set is available for G, and G cannot be classified as a composite symmetry group (see Sections 5 and 6) then we must exploit G via an approximate symmetry reduction strategy.

We propose an approximate strategy based on *gradient-descent local search*,[2] which has proved successful for a variety of search problems in artificial intelligence [36]. 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.

The effectiveness of local search is dependent on the set of group generators used: adding elements to the generating set can potentially improve the accuracy of search, at the expense of speed. In the extreme, G could be used as a generating set, in which case the strategy would compute $\min[s]_G$ precisely, but less efficiently even than brute-force enumeration.

In Section 8 we show that the local search algorithm is effective when exploring the state-spaces of various configurations of message routing in a hypercube network. For these experiments, the generating set is taken to be the set of generators for the group of a hypercube computed by the *saucy* graph automorphism program.

There are various local search techniques which could be employed to attempt to improve the accuracy of this strategy. *Random-restart* local search [36] 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).

[2] This is referred to in [13] as *hillclimbing* local search.

Another potential improvement would be to use *simulated annealing* [27] to escape local minima.

5 Exploiting disjoint products

Certain kinds of symmetry group can be decomposed as products 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 processes independently then the group can be described as the *disjoint product* of the groups acting on these disjoint sets (see Definition 10).

Disjoint products occur frequently in model checking problems. For example, the symmetry group associated with a prioritised resource-allocation system, with $k > 0$ priority levels, is a disjoint product of k groups, each of which permutes processes with a specific priority level. In our three-tiered architecture example (see Fig. 3), the group G_{3T} can be shown to decompose as a disjoint product $G_{3T} = H_1 \bullet H_2$ where:

$$\begin{aligned} H_1 &= \langle (1\ 2), (2\ 3), (4\ 5), (5\ 6), (7\ 8), (8\ 9), \\ &\qquad (12\ 13)(1\ 4)(2\ 5)(3\ 6), (13\ 14)(4\ 7)(5\ 8)(6\ 9) \rangle \\ H_2 &= \langle (10\ 11) \rangle. \end{aligned}$$

If G is a disjoint product of subgroups $H_1, H_2, \ldots, H_k$ then we have $\min[s]_G = \min[\ldots \min[\min[s]_{H_1}]_{H_2} \ldots]_{H_k}$ [6], 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 (see Section 4.3) or wreath product decompositions (see Section 6).

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. From the above discussion it is clear that the finer the disjoint product decomposition the better, thus we would ideally like a *complete* automatic method: one which computes the finest non-trivial disjoint product decomposition for G, if such a decomposition exists.

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 CGT to run efficiently for the kinds of symmetry group which arise in model checking problems.

5.1 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 2 *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 8 *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 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 2 we can arrange the α_l so that elements of C_i appear before those of C_j whenever $i < j$. It follows that we can write α as $\beta_1\beta_2 \ldots \beta_k$ where β_i is the product of all the α_j belonging to C_i, with $\beta_i = id$ if there are no such α_j $(1 \leq i \leq k)$. In other words, $\alpha = \beta_1\beta_2 \ldots \beta_k$ where $\beta_i \in H_i$ $(1 \leq i \leq k)$. Again by Lemma 2, $moved(H_i) \cap moved(H_j) = \emptyset$ 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} 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:

$$C_1 = \{(1\ 2), (2\ 3), (4\ 5), (5\ 6), (7\ 8), (8\ 9), (12\ 13)(1\ 4)(2\ 5)(3\ 6), (13\ 14)(4\ 7)(5\ 8)(6\ 9)\}$$

$$C_2 = \{(10\ 11)\},$$

which generate the groups H_1 and H_2 respectively, described at the start of Section 5. 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.

5.2 Sound and complete approach

If generators for a symmetry group G have been specified manually, or returned as the result of a CGT-based calculation (*e.g.* if G is a subgroup of a wreath product decomposition, computed using the methods of Section 6) then these generators may

not be in a palatable form for the "generators-only" approach of Section 5.1, which may not provide the finest possible disjoint product decomposition. Recall that the finer the decomposition the more efficiently we can compute $\min[s]_G$.

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 CGT-based optimisation. We present three lemmas, the straightforward proofs of which we omit (they are presented in full in [10]). Throughout this section we use (variations of) Ω and $\mathcal{O}$ to refer to orbits and sets of orbits respectively.

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 $moved(\alpha_i) \subseteq \Omega_i' \in \mathcal{O}'$ $(1 \leq i \leq s)$ and $moved(\beta_j) \subseteq \Omega_j \in (\mathcal{O} \setminus \mathcal{O}')$ $(1 \leq j \leq t)$. 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, $H^{\mathcal{O}'} \not\leq G$. For a single orbit Ω, we use H^{Ω} to denote $H^{\{\Omega\}}$.

Lemma 3 *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 $i \in \{1, 2\}$, $H_i = G^{\mathcal{O}_i}$.*

Lemma 4 *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 $G = G^{\mathcal{O}_1} \bullet G^{\mathcal{O}_2}$.*

Algorithm 5 is a recursive algorithm for computing a disjoint decomposition of G. If G can be decomposed, then by Lemma 3 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 uses Lemma 4 to detect when a partition with this property has been found. 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 5 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 performed in low-degree polynomial time using standard CGT data structures [3]. Thus the complexity of Algorithm 5 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 [20]. 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 5 is $O(2^n)$.

Algorithm 5 *disjoint_decomposition*$(G, \mathcal{O})$—G is a group and $\mathcal{O}$ its non-trivial orbits

```
for all partitions {O1, O2} of O do
  if G^O1 ≤ G and G^O2 ≤ G then
    return disjoint_decomposition(G^O1, O1) • disjoint_decomposition(G^O2, O2)
  end if
end for
return G
```

5.2.1 A computational group theoretic optimisation

We can optimise the performance of Algorithm 5 for many commonly occurring symmetry groups using the notion of *dependent orbits*. Recall from Definition 6 that $stab^*_G(Y)$ denotes the pointwise stabiliser of Y in G, which can be computed in polynomial-time.

Definition 15 Let $\Omega_1, \Omega_2 \in \mathcal{O}$. We say that Ω_1 is *dependent* on Ω_2 if

$$\left|stab^*_G(\Omega_2)^{\Omega_1}\right| < \left|G^{\Omega_1}\right|.$$

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. It can be shown that dependent orbits must belong to the same element of the partition of Lemma 3:

Lemma 5 *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 3). Suppose $\Omega_i, \Omega_j \in \mathcal{O}$ are 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 3). If C is an equivalence class of B^*, called a *dependency class*, then by Lemma 5 and induction, $C \subseteq \mathcal{O}_i$ for some i.

Since *Algorithm* 5 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, as noted above, is a polynomial time operation [3].

5.2.2 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:

$$G = \langle (1\ 2\ 3)(4\ 5\ 6)(7\ 8\ 9)(10\ 11\ 12)(14\ 15)(17\ 18)(20\ 21),$$
$$(2\ 3)(5\ 6)(8\ 9)(11\ 12)(13\ 14\ 15)(16\ 17\ 18)(19\ 20\ 21)\rangle.$$

Due to the manner in which the generators of G have been presented, applying the sound and incomplete approach of Section 5.1 does not yield a disjoint product decomposition. Using GAP, we find that G has seven non-trivial orbits:

$$\mathcal{O} = \big\{\{1, 2, 3\}, \{4, 5, 6\}, \{7, 8, 9\}, \{10, 11, 12\},$$
$$\{13, 14, 15\}, \{16, 17, 18\}, \{19, 20, 21\}\big\}$$

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 CGT optimisation is very effective.

Now consider, for any even $n > 2$, the following group:

$$\begin{aligned} G_n = \langle &(1\ 2)(3\ 4), \\ &(3\ 4)(5\ 6), \\ &\vdots \\ &(n-5\ \ n-4)(n-3\ \ n-2), \\ &(n-3\ \ n-2)(n-1\ \ n)\rangle. \end{aligned}$$

i.e., $G_n = \langle \{(2i-1\ \ 2i)(2i+1\ \ 2i+2) : 1 \leq i < n/2\}\rangle$.

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 CGT 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). Note that this is a contrived example: we have not encountered this group 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 5.1 returns the finest disjoint product decomposition of groups whose generators have been automatically computed. The sound, complete approach, with our CGT 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.

6 Exploiting wreath products

Suppose that a symmetry group partitions the processes 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 *wreath product* of the group which acts on the subsets, and the group which permutes the subsets (see Definition 11).

Wreath products occur in model checking problems when systems are modelled using a tree structure. Recall the group G_{3T} introduced in Section 3.3. In Section 5, 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 a 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 11):

$$X_1 = \{1, 2, 3, 12\}$$
$$X_2 = \{4, 5, 6, 13\}$$
$$X_3 = \{7, 8, 9, 14\}$$

Taking $K = S_3$ and $H = S_3$,[3] 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 11. Then:

$$\sigma(K) = \langle (1\ 4)(2\ 5)(3\ 6)(12\ 13), (4\ 7)(5\ 8)(6\ 9)(13\ 14) \rangle$$
$$\sigma_1(H) = \langle (1\ 2), (2\ 3) \rangle$$
$$\sigma_2(H) = \langle (4\ 5), (5\ 6) \rangle$$
$$\sigma_3(H) = \langle (7\ 8), (8\ 9) \rangle$$

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 (*e.g.* using GAP) that $H_1 = \{\beta\, \alpha_1\, \alpha_2\, \alpha_3 : \beta \in \sigma(K), \alpha_1 \in \sigma_1(H), \alpha_2 \in \sigma_2(H), \alpha_3 \in \sigma_3(H)\}$, *i.e.*, $H_1 = H \wr K$.

Let $G \leq S_n$ act on a set X with partition $\mathcal{X}$. Suppose that G has a non-trivial wreath product decomposition of the form $(H, K, \mathcal{X})$, 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 $\min[s]_G = \min[\min[\ldots \min[\min[s]_{\sigma_1(H)}]_{\sigma_2(H)} \ldots]_{\sigma_d(H)}]_{\sigma(K)}$ [6]. 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 5, 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 7). We then propose an extension of our approach to the case where G may not be transitive.

[3] The group H can be thought of as the subgroup of $Sym(\{1, 2, 3, 4\})$ which fixes the point 4, *i.e.*, $H = stab_{S_4}(4) = S_3$.

6.1 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 [22, 35] for details.

Definition 16 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 17 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 5.2. This restriction is well-defined since X_i is clearly a union of orbits 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 the blocks have size m then, for all $1 \leq i \leq d$, we can identify group $(stab_G(X_i))^{X_i}$ 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 18 Let $\mathcal{X} = \{X_1, X_2, \ldots, X_d\}$ be a block system for G where $G \leq Sym(X)$. 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 9). 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, [32] for a proof) shows that if G is a transitive permutation group which admits a non-trivial block system then G is contained in a wreath product. The theorem is followed by two straightforward lemmas.

Theorem 9 *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) wreath product of H and K with associated partition $\mathcal{X}$,* i.e., *$G \leq H \wr K$.*

Lemma 6 *Let $H \wr K$ be the wreath product of Theorem* 9*, with associated block system $\mathcal{X}$. Let $\sigma_1, \sigma_2, \ldots, \sigma_d$ be the actions of H on X described in Definition* 11*. Then $\sigma_i(H) = (stab_G(X_i))^{X_i}$, the stabiliser of block X_i ($1 \leq i \leq d$).*

Conversely to Theorem 9, a wreath product naturally exhibits a block system:

Lemma 7 *Let $G \leq Sym(X)$ and suppose G is a 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.*

The next theorem is a direct consequence of Theorem 9, Lemma 7 and Theorem 3 (Section 3.2).

Theorem 10 *Let $G \leq Sym(X)$ be transitive. Then G can be decomposed as a non-trivial 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 10 is that our search for a non-trivial 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 wreath product associated with the block system, and can determine whether G *is* this wreath product by checking the order of G. Algorithm 6.1 (the correctness of which follows from Theorem 10) 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 10) whether the current block system corresponds to a wreath product decomposition. In the case where equality of orders holds, by Lemma 6 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 1, Section 3.2). 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.

6.1.1 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 [22]. Although (as discussed in Section 3.2) polynomial time algorithms are not

Algorithm 6 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**
        return $\sigma(K), \sigma_1(H), \ldots, \sigma_d(H)$
      end if
    end for
    {Unreachable: there will always be a suitable monomorphism}
  end if
end for
return fail

available for computation of arbitrary setwise stabilisers, a block stabiliser $stab_G(X_i)$ *can* be computed in polynomial time [22], after which computing the restricted group $(stab_G(X_i))^{X_i}$ is straightforward. The potential bottlenecks of Algorithm 6.1 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)`. 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 [28, Lemma 2.4], though this algorithm does not appear to be more efficient than the algorithm provided by GAP. Note that it is only necessary to compute the monomorphism σ if G does indeed decompose as a 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 as follows: $\sigma(\beta)(x_{i,j}) = x_{\beta(i),j}$, where each block X_i has the form $\{x_{i,1}, x_{i,2}, \ldots, x_{i,m}\}$ with $x_{i,j} < x_{i,k}$ whenever $j < k$. Our implementation of Algorithm 6.1 tries this simple pretest for σ before resorting to monomorphism computation.

6.2 Extending the approach to intransitive permutation groups

The results of Section 6.1 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 5 and 3.3 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 [26]: 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 dedicated to wreath products either assume transitivity throughout [28], or only briefly discuss the intransitive case [32].

Transitivity is imposed in Section 6.1 due to Theorem 9. The need for transitivity in the proof of Theorem 9 (see [32]) 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 [35]. We conjecture that Theorem 9 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 [22]. 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 orbit.

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 6.1 can be applied to the set of block systems for G obtained via this process, to obtain a wreath product decomposition.

The symmetry reduction package TopSpin (see [12] and Section 8) 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: the groups $\sigma(K), \sigma_1(H), \sigma_2(H), \ldots, \sigma_d(H)$ returned by Algorithm 6.1 are all subgroups of G. Even if the wreath product decomposition is erroneous, by setting $\min[s]_G = \min[\min[\ldots \min[\min\,[s]_{\sigma_1(H)}]_{\sigma_2(H)} \cdots]_{\sigma_d(H)}]_{\sigma(K)}$ we still have $\min[s]_G \in [s]_G$, which is sufficient to guarantee soundness of symmetry reduction.

7 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 an 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 6, so that $\min[s]_G$ can be computed as described in Section 4.3. 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 5 and 6 respectively. If G remains unclassified and $|G|$ is sufficiently small, enumeration is used (Section 4.2), otherwise local search is selected (Section 4.4).

8 Symmetry reductions in practice

8.1 Extending the model of computation

When processes do not hold references to one another, the simple model of computation and the action of a permutation on a state (described in Sections 3.1 and 3.3 respectively) are sufficient to reason about concurrent systems, since it is always possible to represent the local state of a process using an integer. However, if processes *can* hold references to one another then any permutation that moves process i will also affect the local state of any processes which refer to i.

Sophisticated specification languages such as Promela [23] include special data-types to represent process and channel identifiers. An extended model of computation for such languages is presented in [14]. Neither the results presented in [6] on solving the COP for groups which decompose as disjoint/wreath products, or our results on minimising sets for fully symmetric groups (see Section 4.3) hold in general for this extended model of computation: for Promela specifications where local variables refer to process and channel identifiers, the efficient symmetry reduction strategies presented above are not always exact—in some cases they yield an *approximate* implementation of the min function. As with the approximate local search strategy proposed in Section 4.4, this does not compromise the safety of symmetry reduced model checking. Indeed, for large models, there are many states for which the strategies *do* give exact representatives in an extended model of computation as the experimental results in Section 8.2 show. We present a solution to the problem of extending symmetry reduction techniques to a realistic model of computation in [14].

8.2 Experimental results

We have implemented the strategies discussed in Sections 4–6 as TopSpin [12], a fully automatic symmetry reduction package for the Spin model checker. We present experimental results applying our symmetry reduction techniques to a variety of Promela examples.

8.2.1 Specification families

We consider four specification families which exhibit full symmetry groups. Three are mutual exclusion protocols: a simple mutual exclusion protocol based on the Kripke structure of Fig. 1; a specification of Peterson's n-process mutual exclusion protocol [34] used for experiments with the SymmSpin symmetry reduction package [2], and a more realistic, less atomic version of this specification. The other family of fully symmetric specifications model an email system, adapted from [4], consisting of

n client processes, which communicate by sending messages to a *mailer* process via a *network* channel component. These four specification families 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 these are the only system components. The symmetry group associated with an *email* 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.

To illustrate disjoint product groups, we consider specifications of a *resource allocator*, where prioritised client processes compete for exclusive access to a shared resource. A *resource allocator* configuration with k priority levels in which there are a_i clients with priority level i ($1 \leq i \leq k$) is denoted a_1-a_2-...-a_k. The symmetry group associated with such a specification is a disjoint product $H_1 \bullet H_2 \bullet \cdots \bullet H_k$, where $H_i \cong S_{a_i}$ for each $1 \leq i \leq k$. The group H_i consists of all permutations of *client* processes with priority level a_i which simultaneously permute the *client* communication channels. TopSpin automatically computes this disjoint product decomposition (using the approach presented in Section 5.1), and identifies a minimising set for each factor of the product.

Wreath product groups are associated with configurations of a *three-tiered architecture* specification, introduced in [15], which exhibit network topologies similar to the communication graph of Fig. 3. We consider *three-tiered architecture* specifications which are balanced—that is, there are m *server* processes, and a set of n *client* processes connected to each *server* process (for some $m, n > 0$). Such a configuration is denoted $\underbrace{n\text{-}n\text{-}\ldots\text{-}n}_{m}$, and the associated symmetry group is a 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 sets. The group K permutes the m *server* processes, and an element of K which maps *server i* to *server j* also maps the set of clients connected to *server i* to the set of clients connected to *server j*. It is clear that these wreath product groups are intransitive. TopSpin uses the techniques of Section 6 to automatically compute wreath product decompositions, and then computes distinct minimising sets for each copy of H and a minimising set for K.

To illustrate the case where the symmetry group associated with a specification *cannot* be handled using a minimising set, or via a disjoint/wreath product decomposition, we use specifications of a *hypercube* system, also introduced in [15]. An n-dimensional *hypercube* specification consists of 2^n *node* processes which pass messages using a simple routing algorithm. The hypercube examples exhibit fairly large groups, for which TopSpin selects the *local search* strategy.

8.2.2 Discussion

Figure 4 shows experimental results for various configurations of the above families. For each configuration, we give the number of model states without symmetry reduction (**states orig**), with memory optimal symmetry reduction using the *enumeration* strategy (**states red**), and with symmetry reduction using the strategy chosen by TopSpin (**states fast**). When the number of model states is the same using the *enumeration* and *fast* strategies, '=' appears in the **states fast** column, indicating that

Config	states orig	time orig	states red	time basic	time enum	states fast	time fast
simple mutex							
5	113	0.09	12	0.10	0.10	=	0.06
10	6145	0.11	22	-	1088	=	0.05
15	278529	5	-	-	-	32	0.08
20	1.2×10^7	561	-	-	-	42	0.12
Peterson							
3	2636	0.35	494	0.08	0.02	=	0.35
4	60577	0.60	3106	0.04	0.20	=	0.41
5	1.6×10^6	11	17321	16	7	=	1
6	4.5×10^7	2666	89850	722	304	=	7
7	-	-	442481	30458	13885	=	56
8	-	-	-	-	-	2.1×10^6	412
9	-	-	-	-	-	9.6×10^6	3034
Peterson without atomicity							
2	291	0.36	148	0.34	0.34	=	0.34
3	75356	1	12706	0.83	0.68	=	0.62
4	-	-	3.6332×10^6	3426	972	3.6335×10^6	427
resource allocator							
3-3	16768	0.2	1501	0.9	0.3	=	0.1
4-4	199018	2	3826	57	19	=	0.4
5-5	2.2×10^6	42	8212	4358	1234	=	2
4-4-4	2.4×10^7	1587	84377	-	12029	=	17
5-5-5	-	-	-	-	-	254091	115
three-tiered architecture							
3-3	103105	5	2656	7	4	=	2
4-4	1.1×10^6	37	5012	276	108	=	2
3-3-3	2.5×10^7	4156	50396	4228	1689	=	19
4-4-4	-	-	-	-	-	130348	104
email							
3	23256	0.1	3902	0.9	0.8	3908	0.2
4	852641	9	36255	13	6	38560	2
5	3.04×10^7	3576	265315	679	253	315323	40
6	-	-	1.7×10^6	-	13523	2.3×10^6	576
7	-	-	-	-	-	1.5×10^7	6573
hypercube							
3d	13181	0.3	308	0.6	0.3	468	0.2
4d	380537	18	1240	58	34	6986	13
5d	9.6×10^6	2965	3907	7442	5241	90442	946

Fig. 4 Experimental results for symmetry reduction with TopSpin

the value for this column matches the value in the **states red** column. The use of state compression (a state-space reduction technique provided by Spin [23]) 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 4.2 (**time basic** and **time enum** respectively), for the *fast* (**time fast**) option, as well as for the case where symmetry reduction is not applied (**time orig**). 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.

Figure 5 provides details of the symmetry group associated with each specification configuration. The "$|G|$" column shows the size of each group. For groups which TopSpin was able to automatically classify, the "$G \cong$" column describes the structure of the group as a disjoint/wreath product of symmetric groups (to which the group is isomorphic). The time, in seconds, taken to run the classification algorithm for each configuration is also presented (**classify time**). This is less than a second in all cases.

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 (see columns **time basic**, **time enum** and **time fast** of Fig. 4 respectively).

Processes in the *simple mutex* configurations do not hold references to one another, so the *fast* strategy provides exact symmetry reduction, as expected. The difference in verification time between the *fast* and *enumeration* strategies (columns

Fig. 5 Details of symmetry groups associated with specifications, and time taken to classify each group

Config	jGj	G D	classify time
simple mutex			
5	120	S_5	0.19
10	3.6×10^6	S_{10}	0.14
15	1.3×10^{12}	S_{15}	0.17
20	2.4×10^{18}	S_{20}	0.23
Peterson			
3	6	S_3	0.13
4	24	S_4	0.13
5	120	S_5	0.13
6	720	S_6	0.13
7	5040	S_7	0.13
8	40320	S_8	0.14
9	362880	S_9	0.14
Peterson without atomicity			
2	2	S_2	0.33
3	6	S_3	0.14
4	24	S_4	0.13
resource allocator			
3-3	36	$S_3 \bullet S_3$	0.17
4-4	576	$S_4 \bullet S_4$	0.19
5-5	14400	$S_5 \bullet S_5$	0.16
4-4-4	13824	$S_4 \bullet S_4 \bullet S_4$	0.19
5-5-5	1728000	$S_5 \bullet S_5 \bullet S_5$	0.17
three-tiered architecture			
3-3	72	$S_3 \wr S_2$	0.41
4-4	1152	$S_4 \wr S_2$	0.44
3-3-3	1296	$S_3 \wr S_3$	0.41
4-4-4	82944	$S_4 \wr S_3$	0.51
email			
3	6	S_3	0.16
4	24	S_4	0.16
5	120	S_5	0.13
6	720	S_6	0.14
7	5040	S_7	0.14
hypercube			
3d	48	-	0.07
4d	384	-	0.07
5d	3840	-	0.10

time enum and **time fast** of Fig. 4) is especially marked for the *simple mutex* 10 configuration, where the symmetry group is much 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 8.1). However, for the *Peterson*, *resource allocator* and *three-tiered architecture* specifications, the occurrence of '=' in the **states fast** column of Fig. 4 indicates that exact symmetry reduction is obtained using the *exact* strategy (at least for the configurations to which we could apply the *enumeration* strategy).

Exact symmetry reduction using the *fast* strategy is not obtained for the *email* 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 local search when the *fast* strategy is applied to the *hypercube* specifications. The difference between the **states red** and **states fast** columns of Fig. 4 for these specifications shows that local search requires storage of more states than the enumeration strategy, but considerably fewer states than without symmetry reduction (*cf.* the **states orig** column). Comparing the **time orig**, **time enum** and **time fast** columns we see that symmetry reduced verification using the local search strategy is the fastest method for the hypercube specifications.

9 Conclusions and future work

We have presented a number of theoretical results and strategies for solving the constructive orbit problem (COP) for model checking. We have identified optimisations to the basic approach of solving the COP by symmetry group enumeration; a general local search strategy which can be used to approximate the COP, and a wide class of symmetric groups for which the COP can be solved in polynomial time. One of the main contributions of this paper has been to provide techniques to automatically determine whether an arbitrary group decomposes as a disjoint/wreath product of subgroups. Given a product decomposition for a group G, the COP can be solved efficiently for G by applying solutions for each factor of the product in turn. Our techniques for automatically detecting product decompositions allow efficient symmetry reduction techniques to be fully automated. Experimental results using the TopSpin symmetry reduction package show that our symmetry reduction techniques are effective in practice.

Future work includes further analysis of the complexity of computing the finest disjoint product decomposition for an arbitrary permutation group and determining the exact complexity of the wreath product decomposition problem. Since the methods used by TopSpin to tackle these problems work well in practice, this open problem is of more interest to the computational group theory community than to the model checking community. We also intend to investigate the use of CGT calculations during search for optimised equivalence class enumeration as discussed in Section 4.2.

From a model checking perspective, an important area for future work involves extending TopSpin to support the verification of temporal logic (LTL) properties—

currently the tool is restricted to checking basic safety properties expressed via assertions in a Promela specification.

As discussed in Section 2, our efficient methods for solving the COP in polynomial time for specific types of group contrasts with a generic, non-polynomial approach presented in [30]. In future work we plan to carry out an experimental comparison of these distinct approaches.

Acknowledgements We are grateful to the EPSRC-funded SymNet network of excellence for funding several events at which we were able to discuss our ideas with experts in the computational group theory community. Thanks to Colva Roney-Dougal, Steve Linton, Peter Cameron and members of the GAP forum for comments and insights related to this work, and to the anonymous reviewers for their suggestions on improving the paper.

References

1. Alur, R., Henzinger, T.A.: Reactive modules. Form. Methods Syst. Des. **15**(1), 7–48 (1999)
2. Bosnacki, D., Dams, D., Holenderski, L.: Symmetric spin. Softw. Tools Technol. Trans. **4**(1), 92–106 (2002)
3. Butler, G.: Fundamental algorithms for permutation groups. In: Lecture Notes in Computer Science, vol. 559. Springer, New York (1991)
4. Calder, M., Miller, A.: Generalising feature interactions in email. In: Amyot, D., Logrippo, L. (eds.) Feature Interactions in Telecommunications and Software Systems VII, 11–13 June 2003, Ottawa, Canada, pp. 187–204. IOS, Amsterdam (2003)
5. Cameron, P.: Permutation Groups. Cambridge University Press, Cambridge (1999)
6. Clarke, E.M., Emerson, E.A., Jha, S., Sistla, A.P.: Symmetry reductions in model checking. In: Hu, A.J., Vardi, M.Y. (eds.) Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, 28 June–2 July 1998, Proceedings. Lecture Notes in Computer Science, vol. 1427, pp. 147–158. Springer, New York (1998)
7. Clarke, E.M., Jha, S., Enders, R., Filkorn, T.: Exploiting symmetry in temporal logic model checking. Form. Methods Syst. Des. **9**(1/2), 77–104 (1996)
8. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT, Cambridge (1999)
9. Darga, P.T., Liffiton, M.H., Sakallah, K.A., Markov, I.L.: Exploiting structure in symmetry detection for CNF. In: Malik, S., Fix, L., Kahng, A.B. (eds.) Proceedings of the 41th Design Automation Conference, DAC 2004, San Diego, CA, USA, 7–11 June 2004, pp. 530–534. ACM, New York (2004)
10. Donaldson, A.F.: Automatic techniques for detecting and exploiting symmetry in model checking. Ph.D. thesis, Department of Computing Science, University of Glasgow (2007)
11. Donaldson, A.F., Miller, A.: Automatic symmetry detection for model checking using computational group theory. In: Fitzgerald, J., Hayes, I.J., Tarlecki, A. (eds.) FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Newcastle, UK, 18–22 July 2005, Proceedings. Lecture Notes in Computer Science, vol. 3582, pp. 481–496. Springer, New York (2005)
12. Donaldson, A.F., Miller, A.: A computational group theoretic symmetry reduction package for the SPIN model checker. In: Johnson, M., Vene, V. (eds.) Algebraic Methodology and Software Technology, 11th International Conference, AMAST 2006, Kuressaare, Estonia, 5–8 July 2006, Proceedings. Lecture Notes in Computer Science, vol. 4019, pp. 374–380. Springer, New York (2006)
13. Donaldson, A.F., Miller, A.: Exact and approximate strategies for symmetry reduction in model checking. In: Misra, J., Nipkow, T. (eds.) FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, 21–27 August 2006, Proceedings. Lecture Notes in Computer Science, vol. 4085, pp. 541–556. Springer, New York (2006)
14. Donaldson, A.F., Miller, A.: Extending symmetry reduction techniques to a realistic model of computation. Electr. Notes Theor. Comput. Sci. **185**, 63–76 (2007)
15. Donaldson, A.F., Miller, A., Calder, M.: SPIN-to-GRAPE: a tool for analysing symmetry in Promela models. Electr. Notes Theor. Comput. Sci. **139**(1), 3–23 (2005)

16. Emerson, E.A., Sistla, A.P.: Symmetry and model checking. Form. Methods Syst. Des. **9**(1/2), 105–131 (1996)
17. Emerson, E.A., Wahl, T.: Dynamic symmetry reduction. In: Halbwachs, N., Zuck, L.D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, 4–8 April 2005, Proceedings. Lecture Notes in Computer Science, vol. 3440, pp. 382–396. Springer, New York (2005)
18. The GAP Group: GAP—groups, algorithms, and programming, version 4.4. http://www.gap-system.org/ (2006)
19. Gent, I.P., Harvey, W., Kelsey, T.: Groups and constraints: symmetry breaking during search. In: Hentenryck, P.V. (ed.) Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, 9–13 September 2002, Proceedings. Lecture Notes in Computer Science, vol. 2470, pp. 415–430. Springer, New York (2002)
20. Goldberg, K., Newman, M., Haynsworth, E.: Combinatorial analysis. In: Abramowitz, M., Stegun, I. (eds.) Handbook of Mathematical Functions: With Formulas, Graphs and Mathematical Tables. Dover, New York (1972)
21. Herstein, I.: Topics in Algebra. Wiley, New York (1975)
22. Holt, D.F., Eick, B., O'Brien, E.A.: Handbook of Computational Group Theory. Chapman & Hall/CRC, London (2005)
23. Holzmann, G.: The SPIN Model Checker: Primer and Reference Manual. Addison Wesley, Reading (2004)
24. Ip, C.N., Dill, D.L.: Better verification through symmetry. Form. Methods Syst. Des. **9**(1/2), 41–75 (1996)
25. Jefferson, C., Kelsey, T., Linton, S., Petrie, K.: GAPLex: generalized static symmetry breaking. In: Benhamou, F., Jussien, N., O'Sullivan, B. (eds.) Trends in Constraint Programming, chap. 9, pp. 191–205. ISTE, Eugene (2007)
26. Jha, S.: Symmetry and induction in model checking. Ph.D. thesis, School of Computer Science, Carnegie Mellon University (1996)
27. Kirkpatrick, K., Gelatt, C., Vecchi, M.: Optimization by simulated annealing. Science **220**, 671–680 (1983)
28. Kovács, L.G.: Wreath decompositions of finite permutation groups. Bull. Aust. Math. Soc. **40**(2), 255–279 (1989)
29. Kwiatkowska, M.Z., Norman, G., Parker, D.: Symmetry reduction for probabilistic model checking. In: Ball, T., Jones, R.B. (eds.) Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, 17–20 August 2006, Proceedings. Lecture Notes in Computer Science, vol. 4144, pp. 234–248. Springer, New York (2006)
30. Linton, S.: Finding the smallest image of a set. In: Gutierrez, J. (ed.) Symbolic and Algebraic Computation, International Symposium ISSAC 2004, Santander, Spain, 4–7 July 2004, Proceedings, pp. 229–234. ACM, New York (2004)
31. McMillan, K.: Symbolic Model Checking. Kluwer, Deventer (1993)
32. Meldrum, J.D.P.: Wreath products of groups and semigroups. In: Pitman Monographs and Surveys in Pure and Applied Mathematics, vol. 74. Longman, New York (1995)
33. Miller, A., Donaldson, A.F., Calder, M.: Symmetry in temporal logic model checking. ACM Comput. Surv. **38**(3) (2006), Article 8
34. Peterson, G.L.: Myths about the mutual exclusion problem. Inf. Process. Lett. **12**(3), 115–116 (1981)
35. Rose, J.S.: A Course in Group Theory. Dover, New York (1994)
36. Russel, S., Norvig, P.: Artificial Intelligence, A Modern Approach. Prentice Hall, Englewood Cliffs (1995)
37. Sistla, A.P., Gyuris, V., Emerson, E.A.: SMC: a symmetry-based model checker for verification of safety and liveness properties. ACM Trans. Softw. Eng. Methodol. **9**(2), 133–166 (2000)