Spin-to-Grape: A tool for analysing symmetry in Promela models

Alastair Donaldson, Alice Miller, Muffy Calder

Department of Computing Science, University of Glasgow, Scotland
{ally,alice,muffy}@dcs.gla.ac.uk

Abstract

We provide two examples of Promela models of concurrent, distributed systems, whose associated Kripke structures have more complex symmetry groups than those of models commonly cited in the literature. We present a tool, Spin-to-Grape, which allows the state-graph of a Promela model to be manipulated using the group-theoretic package Gap and its graph-theoretic add on, Grape. Through studying these examples we show a correspondence between the symmetry group of the *channel diagram* of a system and the symmetry group of the Kripke structure associated with the system. We then identify some general classes of systems and describe the symmetry groups of the associated models. Finally we discuss ways in which symmetry reduction techniques incorporated within Spin, e.g. the SymmSpin package, could be extended to exploit symmetry in such models.

Key words: Reactive systems; concurrency; formal modelling and verification; symmetry reduction; distributed systems; Promela/Spin; Gap/Grape; nauty

1 Introduction

Model checking is a technique whereby properties of a system can be checked by building an abstract model of the system and, by exploring all possible execution paths, checking whether the model satisfies the properties. Often models exhibit high degrees of symmetry, and this can be exploited, to reduce the cost of the search. The Spin model checker [11] is an explicit state, on-the-fly model checker, designed primarily for the verification of communications protocols. Verification models for use with Spin are written using the specification language Promela (**Pro**cess **me**ta **la**nguage).

Recently attempts have been made to incorporate symmetry reduction techniques within the Spin model checker [2]. In particular, the SymmSpin package has been shown to give very large factors of reduction when applied to models with many replicated components which are fully symmetric. The

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

symmetry reduction techniques employed by SymmSpin are based on the approach of Ip and Dill using the notion of the *scalarset* [13]—a special data type with restricted, symmetry-preserving operations. In addition to Spin, scalarsets have also been used to add symmetry reduction to other verification systems such as Uppaal [10] and Murφ [6]. Scalarsets are an ideal means by which to exploit symmetry in models with many replicated components if there is full symmetry between these components. For example, scalarsets could be effectively used to gain symmetry reductions when verifying a model of a network of identical processes with the complete topology. Ip and Dill also note that the scalarset type could be modified to work with other kinds of symmetry, e.g. rotational symmetry in a system with a ring topology.

However, although symmetry reduction is currently a hot topic, few examples are available to (a) illustrate the fundamental concepts and (b) generalise the results to more realistic systems. In this paper we provide some concrete examples of models which have more complicated symmetry groups than those of examples commonly cited in the literature, symmetries which cannot be exploited currently by the scalarset approach. Our models are of distributed systems which exhibit both concurrency and non-determinism. We show that the symmetry groups of our models are isomorphic to the symmetry groups of the *channel diagrams* associated with the models, and describe the structure of these groups in terms of basic permutation groups. Through these examples we identify some more general classes of systems and predict the symmetry present in the associated models. We describe a technique by which the group theoretic package Gap [25] is used to investigate the automorphism groups of Promela models, and present a software tool, Spin-to-Grape, which allows the state-graph of a Promela model to be loaded into Gap and manipulated using Grape, a graph-theoretic package for Gap. Finally we propose some extensions to the scalarset data type which could be used to apply symmetry reduction techniques to a larger class of Promela models than is currently possible with SymmSpin.

2 Preliminaries

In section 2.1 we give some mathematical definitions which will be used throughout. The application of symmetry reduction to model checking is briefly discussed in section 2.2, and we give an outline of the Gap package in section 2.3.

2.1 Basic group theory and automorphisms of graphs

Definition 2.1 Let G be a non-empty set, and let $\circ : G \times G \to G$ be a binary operation. We say that $(G, \circ)$ is a *group* if G is closed under $\circ$; $\circ$ is associative; G has an identity element 1_G; and for each element $x \in G$ there is an inverse element $x^{-1} \in G$ such that $x \circ x^{-1} = x^{-1} \circ x = 1_G$.

We call the operation $\circ$ *multiplication* in G. When it is clear what the binary operation is, we simply refer to a group as G rather than $(G, \circ)$. Let H be a non-empty subset of a group G. If H is a group in its own right under the binary operation of G, i.e. it satisfies definition 2.1, then we call H a subgroup of G and write $H \leq G$.

Let G be a group, and let $g_1, g_2, \ldots, g_n \in G$. The set of elements of G obtained by multiplying together (in any order and allowing repetition) any of the elements $g_1, g_2, \ldots, g_n, g_1^{-1}, g_2^{-1}, \ldots, g_n^{-1}$ is denoted by $\langle g_1, g_2, \ldots, g_n \rangle$. This set is a subgroup of G, called the subgroup generated by $g_1, g_2, \ldots, g_n$.

Consider the set $[n] = \{1, 2, \ldots, n\}$. A *permutation* of $[n]$ is a bijection from $[n]$ to $[n]$. We use *disjoint cycle form* [18] to represent permutations of $[n]$. The set of all permutations of $[n]$ forms a group under composition of mappings. This group is called the *symmetric group on n points*, and is denoted S_n.

Definition 2.2 Let $(G, \circ)$ and $(H, *)$ be groups, and let $\theta : G \to H$ be a mapping. We say that θ is an *isomorphism* from G to H if it is bijective and satisfies the following condition:

$$\forall g, h \in G \,.\, \theta(g \circ h) = \theta(g) * \theta(h).$$

In this case the inverse mapping θ^{-1} is also an isomorphism from H to G, and we say that G and H are *isomorphic*, denoted $G \cong H$.

The relation $\cong$ is an equivalence relation on groups [18], so if groups G and H are isomorphic they are equivalent in group theoretic terms. In fact they are algebraically indistinguishable. Hence isomorphic groups are often regarded as equal since they have exactly the same structure. If a group H is *isomorphic to* a subgroup of a group G we just say that H *is* a subgroup of G.

For neatness, and the ability to compare structural properties of different groups, we often identify a group G as being isomorphic to a product of permutation groups H and K. For example as a *direct* product (denoted $H \times K$), a *semi-direct* product (denoted $H.K$), or a *wreath* product (denoted $H \wr K$). We do not provide definitions of these products here, for details see [18].

Definition 2.3 Let $\Gamma = (V, E)$ be a directed, uncoloured graph, where V is a non-empty set of vertices and $E \subseteq V \times V$ is a set of edges. An automorphism of Γ is a bijection α from V to V which satisfies the following condition:

$$\forall x, y \in V \,.\, (x, y) \in E \Leftrightarrow (\alpha(x), \alpha(y)) \in E.$$

If Γ is a coloured graph with a colouring C (C maps each element of V to exactly one colour taken from some non-empty set) then an automorphism of

Γ must satisfy the additional condition:

$$\forall x \in V \,.\, C(x) = C(\alpha(x)).$$

For a directed graph Γ (coloured or uncoloured), let $Aut(\Gamma)$ denote the set of all automorphisms of Γ. It can be shown that $Aut(\Gamma)$ forms a group under composition of mappings. We call $Aut(\Gamma)$ the *automorphism group* of Γ.

2.2 Symmetry and model checking

Throughout this section let $\mathcal{P} = p_1 \parallel p_2 \parallel \cdots \parallel p_n$ be a concurrent program, where $p_1, p_2, \ldots, p_n$ are processes running in parallel for some $n \geq 1$, and AP a set of atomic propositions for the program $\mathcal{P}$. The set of communication channels associated with $\mathcal{P}$ is denoted by $\{c_{n+1}, c_{n+2}, \ldots, c_{n+k}\}$ for some $k \geq 0$.

Definition 2.4 The *Kripke structure* $\mathcal{M}$ over AP associated with $\mathcal{P}$ is a quadruple $\mathcal{M} = (S, R, L, s_0)$ where:

(i) S is a non-empty, finite set of states

(ii) $R \subseteq S \times S$ is a *total* transition relation, that is for each $s \in S$ $\exists t \in S$ such that $(s, t) \in R$

(iii) $L : S \to 2^{AP}$ is a mapping that labels each state in S with the set of atomic propositions true in that state

(iv) $s_0 \in S$ is an initial state.

The Kripke structure $\mathcal{M}$ gives the formal semantics of the program $\mathcal{P}$. An *automorphism* of the Kripke structure $\mathcal{M}$ is an automorphism of the uncoloured, directed graph with vertex set S and edge set R (see definition 2.3). The group of automorphisms of a Kripke structure $\mathcal{M}$ is denoted $Aut(\mathcal{M})$. A subgroup G of $Aut(\mathcal{M})$ induces an equivalence relation $\equiv_G$ on the states of $\mathcal{M}$ by the rule $s \equiv_G t \Leftrightarrow s = \alpha(t)$ for some $\alpha \in G$. The equivalence class under $\equiv_G$ of a state $s \in S$, denoted $[s]$, is called the *orbit* of s under the action of G. The orbits can be used to construct a *quotient* Kripke structure $\mathcal{M}_G$ as follows:

Definition 2.5 The quotient Kripke structure $\mathcal{M}_G$ of $\mathcal{M}$ with respect to G is a quadruple $\mathcal{M}_G = (S_G, R_G, L_G, [s_0])$ where:

(i) $S_G = \{[s] : s \in S\}$ (the set of orbits of S under the action of G)

(ii) $R_G = \{([s], [t]) : (s, t) \in R\}$

(iii) $L_G([s]) = L(rep([s]))$ (where $rep([s])$ is a unique representative of $[s]$)

(iv) $[s_0] \in S_G$ (the orbit of the initial state $s_0 \in S$).

In general $\mathcal{M}_G$ is a smaller structure than $\mathcal{M}$, but $\mathcal{M}_G$ and $\mathcal{M}$ are equivalent in the sense that they satisfy the same set of logic properties which are *invariant* under the group G (that is, properties which are "symmetric" with

respect to G). Thus by choosing a suitable symmetry group G, model checking can be performed over $\mathcal{M}_G$ instead of $\mathcal{M}$, often resulting in considerable savings in memory and verification time. For more details of quotient structures and symmetry reduced model checking see for example [5].

It would be possible in principle to construct a quotient Kripke structure by constructing the original structure, finding its automorphism group, and identifying the orbits of the structure under this group. However, finding automorphisms of a graph is a hard problem, for which no polynomial time algorithm is known [16]. In addition, a quotient Kripke structure cannot be found using this method if the original structure is intractable. Thus any useful symmetry reduction method must allow us to find automorphisms of a Kripke structure without explicitly building the structure.

It is well known that automorphisms of a Kripke structure often arise as a result of symmetry in the architecture or network topology of the concurrent system being modelled [4]. We show in this paper that, in some cases, such symmetries can be detected by looking at the *channel diagram* [19] of the system.

Definition 2.6 The *channel diagram* corresponding to the concurrent program $\mathcal{P}$ is a directed, coloured graph $\mathcal{C}(\mathcal{P}) = (V, E, C)$ where: $V = V_P \cup V_C$ and $V_P = \{1, \ldots, n\}$, $V_C = \{n+1, \ldots, n+k\}$ are the set of indices of processes and channels in the system respectively; for $i, j \in V, (i, j) \in E$ if and only if $i \in V_P, j \in V_C$ and process p_i can send a message on channel c_j, or $i \in V_C, j \in V_P$ and process p_j can receive a message on channel c_i; the mapping C assigns each process or channel to a process type or channel type respectively (where process types and channel types are disjoint).

In Figure 2 we give an example of the channel diagram of a distributed system, where processes are represented by ovals, channels by rectangles, and types by textual labels. Indices are indicated beside each oval or rectangle.

Definition 2.6 is specific to the message passing computation paradigm. In [4], a similar definition is made for a shared variable computation paradigm—the *coloured hypergraph* corresponding to the concurrent program $\mathcal{P}$. Under certain restrictions, each automorphism of the coloured hypergraph corresponds to an automorphism of the underlying Kripke structure associated with the system. An automorphism α of the hypergraph is applied to a state of the Kripke structure by permuting the shared variables and local variables of processes according to the permutation of process indices by α.

An analogous result holds for the message passing paradigm. An automorphism of a channel diagram $\mathcal{C}(\mathcal{P}) = (V, E, C)$ is a graph automorphism (see definition 2.3) of $\mathcal{C}(\mathcal{P})$ considered as a directed, coloured graph. Elements of $Aut(\mathcal{C}(\mathcal{P}))$ permute both the set V_P of process indices and the set V_C of channel indices, and can therefore be applied to states of a Kripke structure as follows. Let us refer to the set of variables whose domains are process identifiers as I-variables. The labelling function L of the Kripke structure labels

each state s with a set of assignments to local variables and channels of $\mathcal{P}$. If v_i is a local variable of process p_i, then, since α maps the process index i to the index of a process with the same process type as p_i, there is a corresponding local variable $v_{\alpha(i)}$ of process $p_{\alpha(i)}$. The labelling of local variables in the state $\alpha(s)$ is therefore defined by the following rules:

$$(v_i = x) \in L(s) \Rightarrow (v_{\alpha(i)} = x) \in L(\alpha(s)) \text{ if } v_i \text{ is not an } I\text{-variable}$$

and

$$(v_i = x) \in L(s) \Rightarrow (v_{\alpha(i)} = \alpha(x)) \in L(\alpha(s)) \text{ if } v_i \text{ is an } I\text{-variable}$$

The automorphism α permutes the contents of channels in $\mathcal{P}$ in a similar way. Under certain restrictions, an automorphism $\alpha \in Aut(\mathcal{C}(\mathcal{P}))$ corresponds to an automorphism $\alpha' \in Aut(\mathcal{M})$, and we have:

Lemma 2.7 *If $\mathcal{P}$ is a concurrent program satisfying a set of restrictions R, $\mathcal{M}$ is the Kripke structure associated with $\mathcal{P}$, and $\mathcal{C}(\mathcal{P})$ is the channel diagram corresponding to $\mathcal{P}$, then $Aut(\mathcal{C}(\mathcal{P})) \leq Aut(\mathcal{M})$.*

The set of restrictions R are similar to those associated with the scalarset approach, described in section 7, in that they apply to the subset of the variables associated with $\mathcal{P}$ which are I-variables. In this case, statements in $\mathcal{P}$ involving an I-variable x are restricted to those of the form $x \Delta y$, where Δ is '$=$', '$==$' or '$\neq$', and y is an I-variable; $chan?x$ (read x from channel with name $chan$); or $chan!x$ (write x to channel with name $chan$).

Although straightforward, for space reasons we do not provide a proof of Lemma 2.7 here. However, we do prove the result for specific examples in sections 4 and 5. Lemma 2.7 enables us to find automorphisms of an intractably large Kripke structure by analysing the channel diagram of the system, which is typically a small graph.

2.3 Gap *and* Grape

Gap (**G**roups, **A**lgorithms and **P**rogramming) [25], is a computational group theory package which consists of an imperative programming language, a type system for working with algebraic structures, and an extensive library of functions for computing with these structures. Gap provides a rich set of functions for computing with groups, but on its own provides little support for graph theoretic computation. Grape (**GR**aph **A**lgorithms using **PE**rmutation groups) [22] consists of a set of functions which can be imported into Gap. Among the functions which Grape provides is a function to compute the automorphism group of a directed, coloured graph. This function interfaces to the *nauty* (**no** **aut**omorphisms, **y**es) program [17], which uses the most efficient algorithm currently known for finding the automorphism group of a graph [16].

Full details of Gap and Grape can be found on the Gap website [25]. In this paper we make use of the following two functions:

- AutGroupGraph(Γ[,C])—Returns the automorphism group of the directed graph Γ. The optional argument C allows a colouring on the vertices of Γ to be specified, and only automorphisms which preserve this colouring will be returned.
- IsomorphismGroups(G,H)—Computes an isomorphism between the groups G and H if they are isomorphic (see definition 2.2), and returns *fail* otherwise.

3 The Spin-to-Grape tool

Among the options which Spin provides for running verifications on Promela models is the `-DVERBOSE` compile-time directive. This option causes every step of a verification to be recorded, and once completed, saved to a file. Running a verification to search for invalid end-states on a deadlock-free model with the `-DVERBOSE` option, and no partial order reduction (an algorithm ensuring that only one representative from a set of equivalent paths is searched), effectively causes the whole of the corresponding Kripke structure to be output. In order to manipulate the states and transitions of the Kripke structure associated with a model using Gap and Grape we have designed a tool, Spin-to-Grape, which takes relevant *verbose* output and produces a graph which can be input to Gap.

The Spin-to-Grape tool consists of a PERL program based on Algorithm 1 below, which re-traces the steps taken by Spin when performing the state-space search. The algorithm uses a separate stack for each process in the model. Every time a line in the input file indicates that a process has executed a statement, the current state number is added to the stack of that process. When a line of input indicates that a process has reversed, a value is popped from the stack of that process, and the current state number is set to this value. This is how the algorithm keeps track of the current state number when processing the input file. Every time a line of input is found which specifies that a new or old state has been found, the algorithm writes a line of Grape code to the output file, specifying that a transition should be added to the state graph. The file produced as output from Spin-to-Grape can be loaded into Gap, and the *AutGroupGraph()* function of Grape can be called to find the automorphism group of the state graph, using the *nauty* algorithm. The PERL code listing is available on our website [3].

Though the Spin-to-Grape tool is extremely useful for working with simple Promela models, it has limitations. Use of the `-DVERBOSE` option greatly increases the time taken for a full state-space search, since Spin displays on-screen every step involved. It would be preferable if the tool were incorporated into Spin, so that as the state-space of a model is explored, Spin could output a file containing the Gap and Grape commands required to build the state-graph, rather than the entire verbose output.

Another problem with the tool is that the size of input file which Gap

Algorithm 1 Algorithm used by Spin-to-Grape to construct the state-space of a model from a Spin output file

```
open input and output files
current state := 1
find the number of states n of the model
output Gap lines to create a null graph K with n vertices
for each line l in the input file do
  if l signifies a new state s then
    output line to add to K an edge from current state to s
    current state := s
  else if l signifies an old state s then
    output line to add to K an edge from current state to s
  else if l indicates that process p executes then
    push current state on to process stack of process p
  else if l indicates that process p reverses then
    pop a state s from stack of process p
    current state := s
  end if
end for
close input and output files
```

can accept is limited due to a bug in Gap (version 4.3). The "current line" counter of an input file is held in a variable of type *short*, and this counter wraps back to 0 if the number of lines of an input file exceeds the capacity of a *short* variable. We have reported this bug and it will be fixed in the next release of Gap. Currently it is possible to modify and recompile Gap to solve the problem, or to split a large input into several smaller files. In addition, the *nauty* algorithm only works for graphs of size $\leq 2^{15}-3$ (although theoretically this bound can be raised via recompilation). More crucially, the complexity of the *nauty* algorithm means that it performs badly on large graphs, and its use is not generally feasible for graphs of more than around 15,000 vertices.

To accompany the Spin-to-Grape tool we have written a Gap function, *QuotientKripke()*. This function takes the states and transitions of a Kripke structure $\mathcal{M}$ (as a directed graph) together with a subgroup G of $Aut(\mathcal{M})$, and returns the directed graph consisting of the states and transitions of the quotient Kripke structure $\mathcal{M}_G$. This function allows us to determine the potential factors of reduction available through the use of symmetry for small models. The code for the *QuotientKripke()* function is also available on our website [3].

In the following sections we describe two Promela models, and show how the Spin-to-Grape tool allows us to analyse the symmetry present in both the state-graphs and the channel diagrams of these models.

4 Three-tiered architecture model

A common software engineering design pattern for distributed systems is the *three-tiered architecture*. Components in such an architecture are separated into three layers, a layer of clients, a layer of servers and a layer of data storage systems. The typical flow of messages for such a system is shown in figure 1 (adapted from [23]). This pattern is common in the e-business domain, where customers buy products or make bookings over the Internet. A set of servers at various geographical locations deal with customers' (clients') requests and communicate with a central (possibly replicated) database.

Our first model is of a simple three-tiered system consisting of three process types: *client*, *server* and *database*. Each *client* process is parameterised by its id and a channel name associated with a server process. The *server* processes are parameterised by two channel names. The first of these channels is used to receive requests from *client* processes, and the second to send queries to the database. A client process loops continuously, sending a *request* message to the server to which it is connected, and waiting until a *result* message is received on its incoming channel. Similarly each server process continuously repeats the actions of receiving a *request* from a client, sending a *query* to the database and receiving *data*, then sending a *result* back to the client. The database process continuously receives queries from the servers and returns data. For each server process there is an array of channels—one channel for each client associated with that server. The database also uses an array of channels, one for each of the servers. All the channels in the model are synchronous. The channel diagram of a system with a database, three servers and eight clients is illustrated in Figure 2. Each channel is annotated with the type of messages it accepts. A message type consists of one or more field types enclosed in curly braces. The field types used here are *byte*, and *mtype*, which is an enumerated type used to represent control instructions. The type of a channel also depends on the length of the channel, i.e. the number of messages it can hold. Since all channels in the model are synchronous (and so have length 0), lengths are not indicated on the diagram. The code for this configuration of the three-tiered model is available on our website [3]

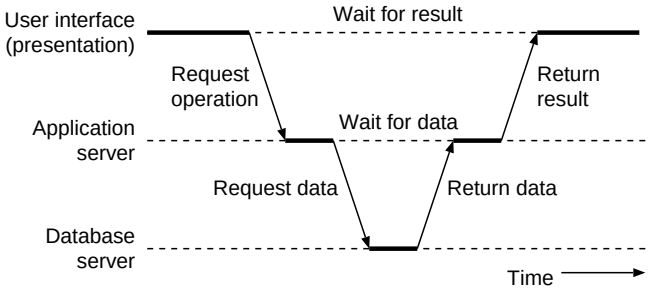


Fig. 1. Flow of control in a three-tiered architecture system

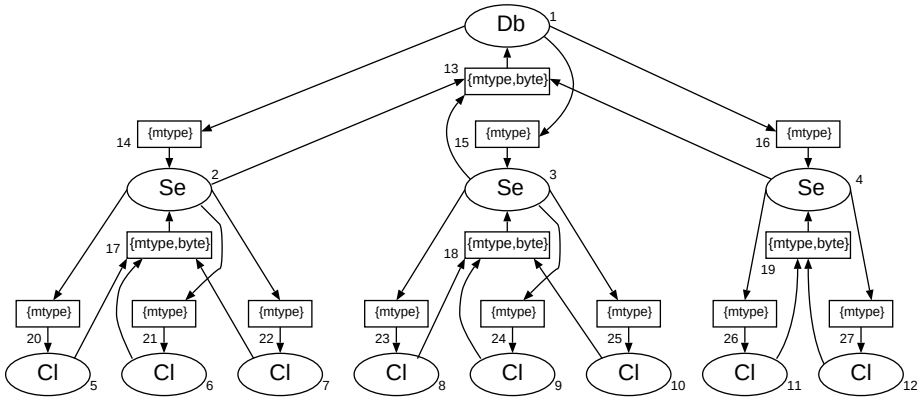


Fig. 2. Channel diagram of a three-tiered architecture model with one database (Db), three servers (Se) and eight clients (Cl).

Analysis of symmetry in the three-tiered model

In order to analyse the symmetry in the model of a three-tiered architecture we use the configuration shown in Figure 2 as a case-study. Let $\mathcal{P}$ denote a system with this configuration, with channel diagram $\mathcal{C}(\mathcal{P})$ and associated Kripke structure $\mathcal{M}$. The configuration is suitable as it has multiple servers as well as multiple clients, and the tree of processes is not perfectly balanced, a feature which we would expect to be reflected in the symmetry present in the Kripke structure. The resulting state-space of the configuration is small enough to allow comprehensive symmetry analysis using our automated setup.

Inputting the channel diagram of Figure 2 to Gap (specifying a colouring according to process types), and calling the Grape function *AutGroupGraph()* to find the group of channel diagram automorphisms shows that

$$Aut(\mathcal{C}(\mathcal{P})) = \langle (5\ 6)(20\ 21), (6\ 7)(21\ 22), (8\ 9)(23\ 24), (9\ 10)(24\ 25),$$
$$(5\ 8)(6\ 9)(7\ 10)(20\ 23)(21\ 24)(22\ 25)(2\ 3)(17\ 18)(14\ 15), (11\ 12)(26\ 27) \rangle$$

We can see from Figure 2 that the first generator of this group, the permutation (5 6)(20 21) is an automorphism of the channel diagram since, clearly, swapping clients 5 and 6, and simultaneously swapping channels 20 and 21, leaves the structure of the channel diagram unchanged. Since clients 5 and 6 have the same process type, and channels 20 and 21 have the same channel type, the permutation preserves the colouring of the channel diagram. The other generators act on the channel diagram similarly. The elements of $Aut(\mathcal{C}(\mathcal{P}))$ act on states of the Kripke structure $\mathcal{M}$ as we described in the discussion preceding Lemma 2.7 in section 2.2. Since channels in the model are synchronous, no swapping of channel contents is necessary in this case. The three-tiered model focusses on control flow rather than data flow, so we would expect all symmetries of the Kripke structure $\mathcal{M}$ to be structurally induced.

We prove that the automorphism groups of the Kripke structure and the

channel diagram are isomorphic, and identify them as being in turn isomorphic to a product of symmetric groups:

Lemma 4.1 *For the configuration of the three-tiered model described above,*

$$Aut(\mathcal{M}) \cong Aut(\mathcal{C}(\mathcal{P})) \cong (S_3 \wr S_2) \times S_2$$

We have used our combination of Spin, Spin-to-Grape, Gap and Grape to prove this lemma. The Spin-to-Grape tool was used to input to Gap the state-graph of the Kripke structure $\mathcal{M}$. The automorphism group of this state-graph was then found using the *AutGroupGraph()* function.

We used Gap to construct a group $G = (S_3 \wr S_2) \times S_2$ (Gap provides functions to compute the direct product and wreath product of two groups). The *IsomorphismGroups()* function was used to show that $Aut(\mathcal{C}(\mathcal{P})) \cong Aut(\mathcal{M})$, and again to show that $Aut(\mathcal{M}) \cong G$.

The original Kripke structure has 4,393 states. We have used our *QuotientKripke()* function to compute the quotient structure with respect to the group $Aut(\mathcal{M})$, finding it to have 281 states. This is a significant factor of reduction which, for realistic sizes of model, could prove extremely effective in combatting the problem of state-space explosion.

Generalisation: Intuitively, the group $Aut(\mathcal{C}(\mathcal{P}))$ is isomorphic to the group $(S_3 \wr S_2) \times S_2$ since there are two blocks of three *client* processes (giving rise to the subgroup $S_3 \wr S_2$), and a single block of two *client* processes (giving rise to the subgroup S_2). Consider an arbitrary configuration $\mathcal{P}$ of the three-tiered model. Let $\mathcal{M}$ be the Kripke structure associated with $\mathcal{P}$ and let K be the maximum number of clients that any server in the configuration is connected to. Let m_i denote the number of servers which are connected to i clients for each $i, 1 \leq i \leq K$. The above discussion and result clearly generalises to give:

$$Aut(\mathcal{M}) \cong Aut(\mathcal{C}(\mathcal{P})) \cong \prod_{\substack{1 \leq i \leq K \\ m_i \neq 0}} (S_i \wr S_{m_i}),$$

where $\prod$ denotes the direct product over i. In [14], Jha shows how the automorphism group of an arbitrary rooted tree can be found. His approach could be used to generalise the above argument to systems with more than three tiers.

5 Model of message routing in a hypercube

A popular topology used in the implementation of switch-based multicomputers is the *hypercube* [23]. The following definition is adapted from [24]:

Definition 5.1 The n-dimensional hypercube is a graph $G = (V, E)$ where

$$V = \{(b_1, \ldots, b_n) : b_i = 0, 1\}$$

and

$$E = \{((b_1, \ldots, b_n), (c_1, \ldots, c_n)) : b \text{ and } c \text{ differ in one bit}\}$$

The 4-dimensional hypercube can be represented in 3-dimensions using two cubes, as shown in figure 3. In a switch-based multicomputer using a hypercube topology, messages are *routed* between the processors. As our second example we model the routing of messages in a hypercube network using Algorithm 2 (below), which is a simplified version of an algorithm described in [8].

Our model defines a parameterised *node* process. To ensure that the state-space of the model is small enough to analyse we only consider one message being passed through the network at a time. Global variables record the destination and current position of the message. Communication is achieved through an array of channels, one for each node in the hypercube, and the *init* process (the process in which initial values of arrays, channels etc. are set) sends the first message to a non-deterministically chosen node.

To decide which neighbours are suitable to forward a given message on to, a node process first computes the bitwise exclusive-or of its own position and the message destination. Each bit of this result is then considered. If there is a 1 in position m of the result then the neighbour of the current node whose coordinates differ in only position m is closer to the message destination than the current node. The node process non-deterministically chooses one such suitable neighbour. Again, Promela code for the model is available on our website [3].

Analysis of symmetry in the hypercube model

The automorphism group of an n-dimensional hypercube is well understood, and is derived in [9]. The nodes of a hypercube can be represented by binary vectors of the form $(x_1, x_2, \ldots, x_n)$. For any permutation α in S_n, define the action of α on $(x_1, \ldots, x_n)$ by $\alpha((x_1, \ldots, x_n)) = (x_{\alpha(1)}, \ldots, x_{\alpha(n)})$. For each

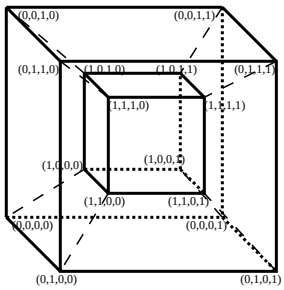


Fig. 3. Topology of a 4-dimensional hypercube

Algorithm 2 Basic algorithm for message routing in a hypercube network

```
while true do
  receive message destined for node x
  if id = x then
    choose a new destination x
  end if
  determine neighbours which are closer than current node to node x
  forward message to one of these neighbours
end while
```

i define the i-th *complementation permutation* c_i by $c_i((x_1, \ldots, x_i \ldots, x_n)) = (x_1, \ldots, \neg x_i, \ldots, x_n)$. Let $K_n = \langle c_1, \ldots, c_n \rangle$, the group generated by all combinations of the c_i. The automorphism group of the n-dimensional hypercube is the *semi-direct product* of S_n and K_n, denoted $S_n.K_n$.

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

Let $\mathcal{P}$ denote the 3-dimensional configuration of the hypercube message-routing system, with channel diagram $\mathcal{C}(\mathcal{P})$ and associated Kripke structure $\mathcal{M}$. The topology of the system is a cube, so we have $Aut(\mathcal{C}(\mathcal{P})) = S_3.K_3$. Since the *node* processes in $\mathcal{P}$ are all identical, we expect any automorphism of the cube topology to correspond to an automorphism of the underlying Kripke structure. Again, all symmetry is likely to be structurally induced.

Lemma 5.2 *For the 3-dimensional configuration of the hypercube model described above,*

$$Aut(\mathcal{M}) \cong Aut(\mathcal{C}(\mathcal{P})) = S_3.K_3$$

We have proved this lemma using Spin-to-Grape, Gap and Grape as we did for Lemma 4.1.

The original Kripke structure has 15,409 states. Using our *QuotientKripke()* function as for the three-tiered model, we find that the resulting quotient structure has 411 states. Again the factor of reduction gained is encouraging.

Lemma 5.2 is interesting since our model of message routing in a hypercube *does not* satisfy the restrictions we discussed following Lemma 2.7: process ids are used as operands in bitwise exclusive-or operations in order to determine how the packet should be routed. This shows that the restrictions associated with Lemma 2.7 are *sufficient*, but not *necessary* for the Lemma to hold. It would be interesting to try to identify general cases where process ids can be used as operands to arithmetic expressions without breaking symmetry.

Generalisation: Let $\mathcal{P}$ be a configuration of the hypercube model with n dimensions for some $n \geq 1$. Let $\mathcal{M}$ be the Kripke structure associated with $\mathcal{P}$, and let $\mathcal{C}(\mathcal{P})$ be the channel diagram corresponding to $\mathcal{P}$. It would seem likely, from the previous discussion, that the above result generalises to give

$$Aut(\mathcal{M}) \cong Aut(\mathcal{C}(\mathcal{P})) = S_n.K_n$$

We would intuitively expect this generalised result to hold for any model of a distributed algorithm on an n-dimensional hypercube of identical processes.

6 Applying Spin-to-Grape to less symmetric models

In our analysis so far we have used Spin-to-Grape, together with Gap and Grape, to prove for certain model configurations that the automorphism group of the Kripke structure associated with the model is isomorphic to the automorphism group of the channel diagram corresponding to the model. In this section we make slight alterations to each of the models to make them less symmetric, and use Spin-to-Grape to detect the changes in symmetry which these alterations cause.

6.1 Mixed modes of communication in the three-tiered model

In our model of the three-tiered architecture illustrated by Figure 2, all communication is modelled using synchronous channels, so that messages are passed via a handshake between sender and recipient, with no buffering. We modify our model by making one of the channels a one place buffer (i.e. an asynchronous channel with length 1)—the channel which clients with process ids 5, 6 and 7 use to send requests to the server with process id 2. The rest of the model is unchanged.

For the Kripke structure $\mathcal{M}$ of the original model we have $Aut(\mathcal{M}) \cong (S_3 \wr S_2) \times S_2$. Let $\mathcal{M}'$ be the Kripke structure of the altered model. Analysis using Spin-to-Grape reveals that $Aut(\mathcal{M}') \cong S_3 \times S_3 \times S_2$, which is a smaller group than $(S_3 \wr S_2) \times S_2$. Clearly this is because the altered communication channel means that it is no longer possible to permute the servers with process ids 2 and 3. This shows that it is not only the behaviour of processes and the presence of connections between processes that affects the structural symmetry of a model: the exact nature of the communication links is also crucial. Our approach to finding symmetry by looking at the channel diagram of a system handles this kind of asymmetry well, since changing the length of a channel changes the *type* of that channel. This is reflected in the symmetry of the channel diagram, and thus the corresponding change in the symmetry of the Kripke structure is also detected.

The Kripke structure $\mathcal{M}'$ has 14,995 states, and using the *QuotientKripke()* function we find that the resulting quotient structure has 1115 states. The factor of reduction is smaller than the factor of reduction before alterations to

the model were made. This is clearly due to the change in symmetry of the model.

6.2 Message routing in a hypercube with a fixed initiator

In the hypercube model described in section 5, the packet is first sent non-deterministically by the *init* process on one of the channels in the system. Such non-determinism in a model can often lead to a blow up of states, and a common approach when attempting to write an efficient verification model would be to remove such non-determinism. We alter the model so that the packet is always first sent on the channel associated with the node with an id of 0. A state-space search using Spin reveals that the altered model has 8,866 states, in comparison to 15,409 states of the original model.

Spin-to-Grape shows that the resulting automorphism group of the altered model is isomorphic to a *subgroup* of the automorphism group of a cube. Let $G = S_3.K_3$ be the automorphism group of a cube, and let $\mathcal{M}'$ be the Kripke structure for the altered model. Then we have $Aut(\mathcal{M}') \cong stab_G(0) = \{\alpha \in G : \alpha(0) = 0\}$, the subgroup of G which fixes node 0 (the *stabiliser* of node 0 in G). This shows that for a model consisting of identical processes connected as a given topology, symmetries of the underlying Kripke structure which correspond to symmetries of the topology will not necessarily be present if the *init* process does not behave *symmetrically* with respect to all processes. Again, our approach would detect such a change in symmetry: in the original model there would be an edge in the channel diagram from the *init* process to *every* channel in the system. In the altered model there would only be an edge from the *init* process to the channel associated with node 0. This would cause a change in symmetry in the channel diagram, and the corresponding change in symmetry of the underlying Kripke structure would be detected.

Interestingly, the *QuotientKripke()* function shows that the quotient structure corresponding to the altered model has size 1,669. The quotient structure corresponding to the model with no alterations has size 411. In this case, although removing non-determinism from the model results in a reduction of size in the Kripke structure, the corresponding reduction in symmetry means that the quotient structure of the altered model is actually *larger* than the quotient structure of the model with no alterations.

7 Extending symmetry reduction in Spin

In [13], Ip and Dill proposed a special *scalarset* data type which could be added to a system description language to make it easy to detect and exploit symmetries of the system. A scalarset is an integer subrange with restricted operations which ensures that consistent permutation of scalarset values throughout all states of a Kripke structure result in an automorphism of the Kripke structure. The restrictions prevent scalarset values being compared with each other

by ordering relations, or used as operands in arithmetic expressions. The ids of equivalent processes in a system could be modelled using scalarsets, indicating that applying a permutation of the ids throughout the Kripke structure would lead to a valid automorphism of the structure.

Though in many cases it has been shown to be effective [2,6,10], in general symmetry reduction using scalarsets is limited as it only allows the specification and exploitation of *total* symmetries. It can certainly not be applied to our examples. Our analysis of the three-tiered model shows that automorphisms of a Kripke structure may not always correspond to the permutation of the ids of just one process type: in a model with a tree-like structure they may be due to combining permutations of the ids of many process types. Similarly our analysis of the hypercube model shows that in networks of identical processes the presence of symmetry is dependent on the network topology. The application of scalarsets is currently limited to very simple topologies such as stars and cliques. We propose two ways in which the scalarset data type could be extended to the kinds of models that we have described. We extend the notation used in [13] in which scalarsets are declared thus:

$$\text{type } \langle name \rangle \ : \ \text{Scalarset}[size]$$

7.1 Associated scalarsets

An *associated scalarset* would allow the symmetry of tree-structures to be described in a model specification. An associated scalarset is declared as usual with a specified size N. The declaration also requires that a list of N scalarset types is specified. This list associates each of the N elements of the associated scalarset with another scalarset. It is safe to permute process ids i and j of an associated scalarset type if the scalarset types associated with both i and j are the same (and are permuted). For example:

$$\text{type } server_id \ : \ \text{AssocScalarset}(3, [\text{Scalarset}(3), \text{Scalarset}(3), \text{Scalarset}(2)]);$$

specifies an associated scalarset type of size 3, where elements 1 and 2 are associated with 'standard' scalarsets of size 3, and element 3 is associated with a 'standard' scalarset of size 2. This associated scalarset type could be applied to the configuration of the three-tiered system shown in Figure 2, assuming that the client processes were declared using appropriate scalarset ids. The idea of association could be applied recursively to describe tree-structures with greater depth.

7.2 Topology-based scalarsets

When specifying a network of identical processes, symmetry between processes may be exploited if the network topology is known. A *topology-based scalarset* is declared with a given size, and a specified topology which must also be

described in some standardised way. For example:

$$\begin{aligned}&\text{type } \mathit{node_id} \ : \ \text{TopologyScalarset}(8, \mathit{cube}) \ \mathit{where}\ \mathit{cube} \ := \\ &\quad 1 \to \{2,3,5\},\ 2 \to \{1,4,6\},\ 3 \to \{1,4,7\},\ 4 \to \{2,3,8\}, \\ &\quad 5 \to \{1,6,7\},\ 6 \to \{2,5,8\},\ 7 \to \{3,5,8\},\ 8 \to \{4,6,7\};\end{aligned}$$

specifies a topology-based scalarset type for nodes in a cube network, such as the one we considered in section 5. The definition of the cube topology indicates, for each node, which other nodes that node is connected to. Any automorphism of the specified topology could be safely applied to a topology-based scalarset, and Grape could be used to find the group of all such automorphisms.

The application of symmetry reduction to model checking using Spin could be enhanced by extending the SymmSpin package to support these new types. In order to do this it would be necessary to define *symmetry preserving* operations on elements of these types, i.e. operations which ensure that symmetry can safely be exploited. It would also be necessary to devise algorithms to make efficient use of the new scalar set type symmetries during model checking. The Spin-to-Grape tool and the automated setup presented here would prove useful during the development of these extensions, to test the validity of symmetry reductions on specific models.

8 Related work

The SymmSpin package [2] has been shown to give very large factors of reduction when applied to models with many replicated components which are fully symmetric. SymmSpin uses the scalarset type introduced by Ip and Dill in [13]; scalarsets have also been used recently to add symmetry reduction techniques to the real-time model checking tool Uppaal [10].

Emerson et al. have investigated the role of symmetry reduction in partially symmetric systems, including systems which involve priority levels which vary between otherwise identical processes [7]. The SMC tool [21], a symmetry-based model checker which allows safety and liveness properties to be verified, has been designed to handle fairness constraints, which cannot be handled by standard symmetry reduction techniques. Results on partial symmetry by Sistla and Gyuris [20] have also been implemented in the SMC tool.

In [15] the application of Gap to the problem of identifying symmetries in digital circuits is considered. No other paper on symmetry and model checking attempts to exploit Gap, or indeed any computational algebra package. Aloul et al. [1] use Gap, and in particular the *nauty* algorithm [17], to implement efficient symmetry breaking techniques for boolean satisfiability. In [14], the symmetries of various common architectures, including hypercubes and rooted trees are discussed. However, no concrete examples of models which use these architectures are provided and the problem of finding these automorphisms

automatically is not investigated.

9 Conclusions and Further work

We have presented a result indicating that automorphisms of a Kripke structure associated with a concurrent system $\mathcal{P}$ can be determined by finding the automorphisms of the associated channel diagram $\mathcal{C}(\mathcal{P})$. Additionally, we have presented a software tool, Spin-to-Grape, which allows us to analyse the symmetry present in simple Promela models of concurrent, distributed systems using the group theoretic package Gap and its graph theoretic add-on, Grape. We have described two Promela models, used Spin-to-Grape to analyse the symmetry present in the Kripke structures of certain configurations of these models, and suggested generalisations of our results to arbitrary configurations. Our analysis has shown that symmetry in the channel diagrams of our models corresponds to symmetry in the associated Kripke structures. Further, our analysis has identified kinds of symmetry which cannot currently be exploited by the Spin symmetry reduction package SymmSpin. We have proposed some extensions to the scalarset data type which could be used to extend the capabilities of SymmSpin.

We aim to develop the SymmSpin package by implementing the extensions to the scalarset data type which we have proposed. In particular, this will involve the identification of symmetry preserving operations for each new type, as well as algorithms to make efficient use of such symmetries during model checking. We are currently investigating how the Spin-to-Grape tool could be incorporated into Spin, to avoid having to generate verbose output in order to investigate the associated state-graph.

In this paper we have not searched for symmetries when a temporal logic property is present in the model, nor have we attempted any sort of symmetry reduction. Our purpose here was to create a tool that could identify symmetries in simple models and to investigate ways in which the SymmSpin approach could be extended. However, we intend to extend our approach to address both of the issues above.

References

[1] F.A. Aloul, A. Ramani, I.L. Markov, and K.A. Sakallah. Solving difficult SAT instances in the presence of symmetry. *IEEE Transactions on Computer Aided Design*, 22(9):1117–1137, September 2003.

[2] Dragan Bosnacki, Dennis Dams, and L. Holenderski. Symmetric Spin. In Klaus Havelund, John Penix, and Willem Visser, editors, *Proceedings of the 7th SPIN Workshop (SPIN 2000)*, volume 1885 of *Lecture Notes in Computer Science*, pages 1–19, Stanford, CA, USA, September 2000. Springer-Verlag.

[3] M. Calder and A. Miller. Veriscope publications website: `http://www.dcs.gla.ac.uk/research/veriscope/publications.html`.

[4] E. Clarke, E. Emerson, S. Jha, and A. Sistla. Symmetry reductions in model-checking. In Hu and Vardi [12], pages 147–158.

[5] E. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1–2):77–104, 1996.

[6] D. Dill, A. Drexler, A. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *Proceedings 1992 IEEE International Conference on Computer Design: VLSI in Computer & Processors (ICCD'92)*, pages 522–525, Cambridge, MA, USA, October 1992. IEEE Computer Society.

[7] E. Emerson, J. Havlicek, and R. Trefler. Virtual symmetry reduction. In *Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science*, pages 121–131, Santa Barbara, CA, USA, June 2000. IEEE Computer Society Press.

[8] A. Ferreira. Parallel and Communication Algorithms for Hypercube Multiprocessors. In A. Zomaya, editor, *Handbook of Parallel and Distributed Computing*, chapter 19, pages 568–589. McGraw-Hill, New York (USA), 1996.

[9] F. Harary. The automorphism group of a hypercube. *Journal of Universal Computer Science*, 6(1):136–138, 2000.

[10] M. Hendriks, G. Behrmann, K. Larsen, P. Niebert, and F. Vaandrager. Adding symmetry reduction to UPPAAL. In K.G. Larson and P. Niebert, editors, *Proceedings of the 1st International Workshop on Formal Modelling and Analysis of Timed Systems (FORMATS 2003)*, volume 2791 of *Lecture Notes in Computer Science*, pages 46–59, Merseille, France, September 2003. Springer-Verlag.

[11] G. Holzmann. *The SPIN model checker: primer and reference manual*. Addison Wesley, Boston, U.S.A., 2003.

[12] A. Hu and M. Vardi, editors. *Proceedings of the 10th International Conference on Computer-aided Verification (CAV '98)*, volume 1427 of *Lecture Notes in Computer Science*, Vancouver, British Columbia, Canada, June/July 1998. Springer-Verlag.

[13] C. Ip and D. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9:41–75, 1996.

[14] Somesh Jha. *Symmetry and Induction in Model Checking*. PhD thesis, School of Computer Science, Carnegie Mellon University, October 1996.

[15] G. Manku, R. Hojati, and R. Brayton. Structural symmetry and model checking. In Hu and Vardi [12], pages 159–171.

[16] B. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[17] B. McKay. *nauty* user's guide (version 1.5). Technical Report TR-CS-90-02, Australian National University, Computer Science Department, 1990.

[18] J. Rose. *A Course in Group Theory.* Dover Publications, New York, U.S.A., 1994.

[19] P. Saffrey. *Optimising Communication Structure for Model Checking.* PhD thesis, Department of Computing Science, University of Glasgow, July 2003.

[20] A. Sistla and P. Godefroid. Symmetry and reduced symmetry in model checking. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of the 13th International Conference on Computer-aided Verification (CAV 2001)*, volume 2102 of *Lecture Notes in Computer Science*, pages 91–103, Paris, France, July 2001. Springer-Verlag.

[21] A. Sistla, V. Gyuris, and E. Emerson. SMC: A symmetry-based model checker for verification of safety and liveness properties. *ACM Transactions on Software Engineering and Methodology*, 9:133–166, 2000.

[22] L.H. Soicher. Grape: a system for computing with graphs and groups. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 11:287–291, 1991.

[23] A. Tanenbaum and M. van Steen. *Distributed Systems Principles and Paradigms.* Prentice Hall, 2002.

[24] G. Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, 1994.

[25] The Gap Group. *GAPâĂŞ- Groups Algorithms and Programming, Version 4.2.* Aachen, St. Andrews, 1999. http://www-gap.dcs.st-and.ac.uk/∼gap.