Automatic symmetry detection for model checking using computational group theory

A.F. Donaldson[*] and A. Miller

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

Abstract. We present an automatic technique for the detection of structural symmetry in a model directly from its Promela specification. Our approach involves finding the *static channel diagram* of the model, a graphical representation of channel-based system communication; computing the group of symmetries of this diagram; and computing the largest possible subgroup of these symmetries which induce automorphisms of the underlying model. We describe a tool, SymmExtractor, which, for a given model and LTL property, uses our approach to find a group of symmetries of the model which preserve the property. This group can then be used for symmetry reduction during model checking using existing quotient-based methods. Unlike previous approaches, our method can detect arbitrary structural symmetries arising from the communication structure of the model.

Keywords

Promela/SPIN; symmetry reduction; model checking; communicating processes; distributed systems; formal modelling; GAP; concurrency

1 Introduction

Model checking [5] is an increasingly popular technique for the formal verification of concurrent systems. The application of model checking is limited due to the state-space explosion problem—as the number of components represented by a model increases, the size of the associated state-space grows exponentially. As such, models of realistic systems are often too large to feasibly check. Symmetry reduction techniques [3, 7, 15] can be used to combat this problem for models of systems with many replicated components. Symmetry in a system can result in portions of the state-space of a model of the system being *equivalent* up to rearrangement of component ids. If symmetry is known to be present in a model then model checking of certain properties can be performed over a quotient state-space, which is generally smaller than the full state-space of the model. Most

[*] Supported by the Carnegie Trust for the Universities of Scotland.

work on exploiting symmetry during model checking assumes that symmetries of a model are either known *a priori* [7], or are coded into the model through the use of special keywords [3, 15]. Both approaches require the modeller to provide information on the presence of symmetry in a model. This is potentially error prone, and compromises the automation of model checking, which is one of its main strengths as a verification technique. The challenge of automatic symmetry detection is to infer symmetries of the state-space underlying a model *without* explicitly constructing the state-space. The inferred symmetries must be guaranteed to be valid, otherwise the results of symmetry-reduced model checking are untrustworthy.

In this paper we present a method for the automatic detection of symmetry directly from the source code of a model, requiring no additional input from the user. Our approach applies to models written using the Promela specification language (used as input to the SPIN model checker [14]). Given a Promela model, generators for a group of *candidate* symmetries are found by analysing the *static channel diagram* of the model. These generators are checked individually against the model to see if they induce valid automorphisms of the underlying state-graph. Starting with the set of candidate generators which are valid, the largest possible subgroup of candidate symmetries which are all valid is computed. Unlike previous approaches to specifying symmetry using *scalarsets* [3, 15], our method can detect *arbitrary* structural symmetries arising from the communication structure of a model. A scalarset can only be used to specify full symmetry between a set of components of a model. The symmetry group computed using our approach is, by construction, an invariance group for a specified linear temporal logic (*LTL*) formula (contained within the Promela model). As such, the group can be used safely for symmetry reduction during model checking. Static channel diagrams were introduced in previous work [11]. The significant additional contributions of this paper include some detailed theoretical results to determine *valid* automorphisms, and the implementation of our approach via a tool, SymmExtractor, which makes use of the computational group theory package GAP [13]. We provide experimental results for a variety of models, and discuss how our approach can be extended. We conclude by briefly discussing some of the issues which will be involved in future work, implementing symmetry reduction techniques into SPIN based on our approach to symmetry detection.

2 Preliminaries

Model checking involves checking the correctness of a temporal logic formula ϕ over a Kripke structure $\mathcal{M} = (S, R, L, s_0)$ and a set of atomic propositions AP, where S is a finite set of states, $R \subseteq S \times S$ is a total transition relation, $L : S \to 2^{AP}$ labels each state with the propositions that are true at the state, and $s_0 \in S$ is an initial state. The Kripke structure $\mathcal{M}$ represents a model of a concurrent system. In practice $\mathcal{M}$ is obtained from a high level specification $\mathcal{P}$ written in a language such as Promela [14].

2.1 Promela

Promela (**Pro**cess **me**ta **la**nguage) is a high level specification language for modelling concurrent, distributed systems, and Promela programs are used as input to the Spin model checker [14]. A Promela program consists of a series of *proctype* definitions, global variable and channel declarations, an *init* process (used to initialise the model), and (optionally) a *never claim* process (used to verify a *LTL* formula). A *proctype* defines a parameterised process type, of which multiple copies can be instantiated by the *init* process. A *proctype* definition has the form `proctype name(param_list) {body}`. The body of a *proctype* consists of local variable declarations, as well as expressions and statements over local and global variables, and channels. A statement of the form

```
if :: seq_1
   :: seq_2
      ...
   :: seq_m
fi
```

is used to model nondeterministic branching (branching in which any executable sequence `seq_i` may be chosen). Similarly, a `do...od` statement is used to model repeated nondeterministic branching.

Global variables, and variables local to a *proctype* can be declared of type *bit*, *byte*, *short*, *int*, *pid*, *chan*, or *mtype*. A variable of type *chan* refers to a system channel, which has the form `[x] of {field_1,field_2,...,field_m}`, where $x \geq 0$ is the capacity of the channel, $m > 0$ is the number of fields which a message must contain to be sent on the channel, and for $1 \leq i \leq m$, $field\_i \in \{bit, byte, short, int, pid, chan, mtype\}$ specifies the type of the ith field of a message. A *send* operation on channel c is denoted $c!msg$, where msg is a list of values or variables, one for each field of the message. Similarly, a *receive* operation on channel c is denoted $c?msg$. Variables of type *pid* should only be assigned values that correspond to the instantiation number (process id) of an executing process. Each process has a predefined, read-only, local variable *_pid* which stores its instantiation number The value 0 may be used as a default value for variables of type *pid*. This is the instantiation number of the *init* process.

In this paper we consider models where all processes are instantiated simultaneously by the *init* process, and where processes do not themselves instantiate child processes (we discuss the implications of this in Section 5.5). In such models the *init* process has the form

```
init { atomic { run proctypename_1(params_1);
            ... run proctypename_m(params_m) } }
```

The keyword *atomic* ensures that the statements enclosed in the pair of braces immediately following the keyword are executed in sequence as a *single* transition of the system (provided that the statements do not block). In a Promela model, the *init* process is assigned process id 0 by default, and the other processes are assigned process ids in order, starting from 1. Two processes have the same

```
      chan box_1 = [1] of {pid,pid}; chan box_2 = [1] of {pid,pid};
      chan box_3 = [1] of {pid,pid}; chan box_4 = [1] of {pid,pid};
      chan box_5 = [1] of {pid,pid}; chan network = [5] of {pid,pid};
      pid received_from

      proctype mailer(chan in) {
        pid source, dest;
(1)     pid blocked_client = 3;
        chan out;
        do :: in?source,dest;
              if :: source==blocked_client -> skip
                 :: else ->
                    if :: dest==1 -> out = box_1 :: dest==2 -> out = box_2
(2)                    :: dest==3 -> out = box_3 :: dest==4 -> out = box_4
                       :: dest==5 -> out = box_5
                    fi;
                    out!source,dest
             fi
        od
      }

      proctype client(chan in) {
        pid source, dest;
        do :: in?source,dest; assert(dest==_pid); received_from = source
           :: atomic { nfull(network) -> source = _pid;
(3)            if :: dest = 1 :: dest = 2 :: dest = 3 :: dest = 4 :: dest = 5 fi;
               network!source,dest }
        od
      }

      init {
        atomic {
(4)       run client(box_1); run client(box_2); run client(box_3);
          run client(box_4); run client(box_5); run mailer(network)
        }
      }

      never {    /* !([]  (received_from!=3)) */
      T0_init:
(5)     if :: (! (received_from!=3)) -> goto accept_all
           :: (1) -> goto T0_init
        fi;
      accept_all: skip }
```

Fig. 1. Promela model of an email system.

process type if they are instantiations of the same *proctype*. To verify an *LTL* property, SPIN converts the *negation* of the property into a Büchi automaton, expressed as a *never-claim*. A never-claim is an additional process in the Promela model, specifying system behaviour that should *never* occur [14], i.e. behaviour which violates the given property.

Figure 1 shows Promela code for a model of an email system, adapted from [4]. The system consists of 5 instantiations of a parameterised *client* process, running in parallel with a *mailer* process. The *client* processes can send messages to each other via the *mailer* process, but all messages sent by the process with id 3 are blocked by the *mailer* process. Labels (1)—(5) have been added to the code for explanatory reasons and should otherwise be ignored. An example *LTL* property of interest for this model is:

Property 1 $[](received\_from \neq 3)$

which states that no *client* ever receives a message from *client* 3. Note that $received\_from$ is a global variable which is reset by a *client* process each time a message is received. The never-claim for Property 1 has been included at the end of the Promela code shown in Figure 1. As properties are included within the model in this way, the automorphism groups computed by our approach to symmetry detection are, by construction, property preserving (see Section 2.2).

Let $\mathcal{P}$ be a Promela program. Let Loc be the set of local variables, $Glob$ the set of global variables, and $Chan$ the set of channels of $\mathcal{P}$. Let D be the set of data values for the program. To denote a local variable of a process with process id i we write x_i where x is the name of the variable. For example, in the email example, $source_i$ denotes the local variable *source* of a *client* process with process id i. If x_i is a local variable of process i, and if processes i and j have the same process type, then x_j is the corresponding local variable of process j.

We now define the set AP of atomic propositions for a Promela program. Let $AP_{local} = \{(x_i = val) : x_i \in Loc, val \in D\}$, the set of propositions relating to local variables, and define AP_{global} and $AP_{channel}$, the set of propositions relating to global variables and channels respectively, similarly. Then $AP = AP_{local} \cup AP_{global} \cup AP_{channel}$. The underlying Kripke structure $\mathcal{M}$ over AP for the program $\mathcal{P}$ is generated by exploring all possible behaviours of $\mathcal{P}$. States of $\mathcal{M}$ are uniquely identified by a labelling of atomic propositions, and transitions between states are derived from the statements of the program. Note that each process in $\mathcal{P}$ has its own *program counter* variable which indicates the statements which may be executed in the next transition. Thus two states, for which all other variables are assigned identical values, may be distinguished due to assignments of the associated program counters.

We say that two programs $\mathcal{P}_1$ and $\mathcal{P}_2$ are equivalent, and write $\mathcal{P}_1 \equiv \mathcal{P}_2$, if they are the same up to rearrangement of: options in `if...fi` and `do..od` statements; operands to commutative operators; and `run` statements within the `init{atomic{...}}` block. Equivalent programs have identical behaviour, and thus the underlying Kripke structures for equivalent programs are the same.

2.2 Group theory and symmetry in model checking

Let G be a group, and let $\alpha_1, \alpha_2, \ldots, \alpha_n \in G$. The smallest subgroup of G containing the elements $\alpha_1, \ldots, \alpha_n$ is denoted $\langle \alpha_1, \alpha_2, \ldots, \alpha_n \rangle$, and is called the subgroup *generated* by $\alpha_1, \alpha_2, \ldots, \alpha_n$. The elements α_i $(1 \leq i \leq n)$ are called *generators* for this subgroup. Let $X = \{\alpha_1, \ldots, \alpha_n\}$ be a finite subset of G. Then we use $\langle X \rangle$ to denote $\langle \alpha_1, \ldots, \alpha_n \rangle$, the subgroup generated by X.

Let H be a subgroup of G, and let $\alpha \in G$. The set $H\alpha = \{\beta\alpha : \beta \in H\}$ is called a *right coset* of H in G. The set of all right cosets of H in G partitions G into disjoint equivalence classes. In particular, for $\alpha \in H$, we have $H\alpha = H$ [16].

Let $\mathcal{M} = (S, R, L, s_0)$ be a Kripke structure. An *automorphism* of $\mathcal{M}$ is a bijection $\alpha : S \to S$ which satisfies the following conditions:

- $\forall s, t \in S, \;\; (s,t) \in R \Rightarrow (\alpha(s), \alpha(t)) \in R$,

- $\alpha(s_0) = s_0$

In a model of a concurrent system with many replicated processes, Kripke structure automorphisms usually involve the permutation of process identifiers of identical processes throughout all states of a model. The set of all automorphisms of the Kripke structure $\mathcal{M}$ forms a group under composition of mappings. This group is denoted $Aut(\mathcal{M})$. A subgroup G of $Aut(\mathcal{M})$ induces an equivalence relation $\equiv_G$ on the states of $\mathcal{M}$ thus: $s \equiv_G t \Leftrightarrow s = \alpha(t)$ 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 1. *The quotient Kripke structure $\mathcal{M}_G$ of $\mathcal{M}$ with respect to G is a tuple $\mathcal{M}_G = (S_G, R_G, L_G, [s_0])$ where:*

- $S_G = \{[s] : s \in S\}$ *(the set of orbits of S under the action of G),*
- $R_G = \{([s], [t]) : (s, t) \in R\}$,
- $L_G([s]) = L(rep([s]))$ *(where $rep([s])$ is a unique representative of $[s]$),*
- $[s_0] \in S_G$ *(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). For a proof of the following theorem, together with details of the temporal logic CTL^*, see [5].

Theorem 1. *Let $\mathcal{M}$ be a Kripke structure, G a subgroup of $Aut(\mathcal{M})$ and ϕ a CTL^* formula. If ϕ is invariant under the group G then*

$$\mathcal{M}, s \models \phi \Leftrightarrow \mathcal{M}_G, [s] \models \phi$$

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 [3, 7]. Consider Property 1 for our email example. The property explicitly refers to the id of *client* 3, so an invariance group for this property is any subgroup of $Aut(\mathcal{M})$ which fixes *client* 3.

If automorphisms of a Kripke structure can be identified in advance, then a quotient structure can be incrementally constructed using an algorithm given in [15]. This means that it may be possible to construct the quotient structure even if the original structure is intractable. In the next section we show that symmetries of the Kripke structure associated with a Promela program can be detected by analysing the *static channel diagram* of the program.

3 Finding automorphisms via static channel diagrams

In this section we define the static channel diagram $\mathcal{C}(\mathcal{P})$ associated with a Promela program $\mathcal{P}$, and show how automorphisms of the corresponding Kripke structure $\mathcal{M}$ can be obtained by finding the automorphisms of $\mathcal{C}(\mathcal{P})$.

3.1 Static channel diagrams

Let $\mathcal{P}$ be a Promela program. A *static channel* of $\mathcal{P}$ is a channel which is declared globally, out of the scope of any *proctype* definition. Let V_P be the set of process identifiers for $\mathcal{P}$, and let V_C be the set of names of static channels of $\mathcal{P}$. For $i \in V_P$ let *proctype*(i) be the name of type *proctype* of which process i is an instantiation, and for $i \in V_C$ let *chantype*(i) denote the type of channel i (see Section 2.1).

Definition 2. *The static channel diagram of $\mathcal{P}$ is a coloured, bipartite digraph $\mathcal{C}(\mathcal{P}) = (V, E, C)$, where:*

- *$V = V_P \cup V_C$;*
- *For $i \in V_P$, $j \in V_C$, $(i, j) \in E$ iff process i has a send statement $j!msg$, $(j, i) \in E$ iff process i has a receive statement $j?msg$;*
- *For $x \in V$, $C(x) =$ proctype(x) if $x \in V_P$, and $C(x) =$ chantype(x) if $x \in V_C$.*

The static channel diagram of a program represents the potential communication links that can be detected from the program by considering only static channels. Figure 2 illustrates the static channel diagram for the email model of Figure 1. Processes are represented by ovals and channels by rectangles. The type of a process is indicated by the name preceding its process id in the diagram. The type of a channel is indicated by the shading of the rectangle. Note that although the *mailer* process sends messages to the client processes, it does so using its local channel, *out*, which is not a static channel. Thus this communication is not indicated in the static channel diagram.

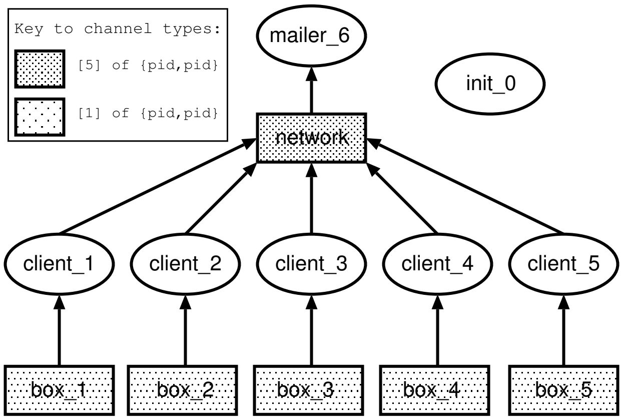

Fig. 2. Channel diagram of the message passing model.

An automorphism of the static channel diagram $\mathcal{C}(\mathcal{P})$ is a bijection $\alpha : V \to V$ which satisfies the following conditions:

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

Note that the second condition ensures that channels can only be mapped on to one another if they have the same capacity. It can be shown that the set of automorphisms of a static channel diagram $\mathcal{C}(\mathcal{P})$ forms a group under composition of mappings. We denote this group $Aut(\mathcal{C}(\mathcal{P}))$. Although our technique exploits the static communication declared within a Promela model, dynamic communication (in which messages are passed on channels whose names are received by other processes during program execution) which *cannot* be determined statically, is still permissible. This is because, if processes i and j are otherwise shown to be symmetrically equivalent (via static analysis), any dynamic communication involving process i will be reflected by corresponding dynamic communication involving process j.

Consider the static channel diagram for our email example, shown in Figure 2. Let $\alpha = (1\ 2)(box\_1\ box\_2)$, the mapping which swaps *client* 1 with *client* 2, and simultaneously swaps *box_*1 with *box_*2 in the diagram. Clearly α is an automorphism of $\mathcal{C}(\mathcal{P})$. In fact any permutation of the *client* processes and their incoming channels which leaves *box_i* connected to *client i* is an automorphism of $\mathcal{C}(\mathcal{P})$. The group $Aut(\mathcal{C}(\mathcal{P}))$ can be generated by the set $\{(1\ 2)(box\_1\ box\_2), (2\ 3)(box\_2\ box\_3), (3\ 4)(box\_3\ box\_4), (4\ 5)(box\_4\ box\_5)\}$.

We now show how the elements of $Aut(\mathcal{C}(\mathcal{P}))$ act on the source text of the Promela program $\mathcal{P}$, and on the Kripke structure underlying the program.

3.2 Action of $Aut(\mathcal{C}(\mathcal{P}))$ on $\mathcal{P}$

Let $\mathcal{P}$ be a Promela program with static channel diagram $\mathcal{C}(\mathcal{P})$, let $n > 0$ be the number of processes instantiated by $\mathcal{P}$, and let $\alpha \in Aut(\mathcal{C}(\mathcal{P}))$. The program $\alpha(\mathcal{P})$ is the same as $\mathcal{P}$, except that every applied occurrence of a static channel name c is replaced by the static channel name $\alpha(c)$, and every assignment statement of the form $x = val$, boolean expression of the form $x == val$ or $val == x$, where $type(x) = pid$ and $val \in \{1, \ldots, n\}$, is replaced by $x = \alpha(val)$, $x == \alpha(val)$ or $\alpha(val) == x$ respectively.

Consider the element $\alpha = (1\ 2)(box\_1\ box\_2) \in Aut(\mathcal{C}(\mathcal{P}))$ where $\mathcal{P}$ is our email example. Applying α to the code given in Figure 1 results in an identical program, except for the ordering of the options at labels (2) and (3), and the ordering of the *run* statements at label (4). Therefore $\mathcal{P} \equiv \alpha(\mathcal{P})$. If we take the element $\alpha = (2\ 3)(box\_2\ box\_3)$ then the programs $\mathcal{P}$ and $\alpha(\mathcal{P})$ are *not* equivalent, since the statement $blocked\_client = 3$ in $\mathcal{P}$ shown at label (1) of Figure 1 is replaced by the statement $blocked\_client = 2$ in $\alpha(\mathcal{P})$. Neither statement appears in both programs. Similarly, applying α to the expression $(!(received\_from! = 3))$ shown at label (5) of Figure 1 results in the expression $(!(received\_from! = 2))$. This inconsistency between $\mathcal{P}$ and $\alpha(\mathcal{P})$ shows that the given LTL property is not invariant under α.

For an element $\alpha \in Aut(\mathcal{C}(\mathcal{P}))$, we say that α is *valid* (for $\mathcal{P}$) if $\alpha(\mathcal{P}) \equiv \mathcal{P}$. We say that a subgroup H of $Aut(\mathcal{C}(\mathcal{P}))$ is valid (for $\mathcal{P}$) if every $\alpha \in H$ is valid.

3.3 Action of $Aut(\mathcal{C}(\mathcal{P}))$ on $\mathcal{M}$

For an element $\alpha \in Aut(\mathcal{C}(\mathcal{P}))$ we define a corresponding mapping α^* which is a permutation of the Kripke structure $\mathcal{M}$ underlying $\mathcal{P}$. For any $s \in S$, let $L(\alpha^*(s)) = \{\alpha(p) : p \in L(s)\}$. For a proposition $p \in AP$, the proposition $\alpha(p)$ is defined as follows:

If $p = (x_i = val) \in AP_{local}$ for some $x_i \in Loc$, and $type(x_i) \in \{pid, chan\}$ then $\alpha(p) = (x_{\alpha(i)} = \alpha(val))$, otherwise $\alpha(p) = (x_{\alpha(i)} = val)$. If $p = (x = val) \in AP_{global}$ for some $x \in Glob$, and $type(x) \in \{pid, chan\}$ then $\alpha(p) = (x = \alpha(val))$, otherwise $\alpha(p) = p$. If $p = (c[i] = msg) \in AP_{channel}$ for some $c \in Chan$, i.e. msg is at position i on channel c, then $\alpha(p) = (\alpha(c)[i] = \alpha(msg))$. Here α acts on msg by permuting the value of each field of msg which has type pid or $chan$, and leaving all other fields unchanged.

The following theorem shows that in certain cases the permutation α^* of $\mathcal{M}$ defined by an element $\alpha \in Aut(\mathcal{C}(\mathcal{P}))$ is an automorphism of $\mathcal{M}$.

Theorem 2. *Let $\mathcal{P}$ be a Promela program with static channel diagram $\mathcal{C}(\mathcal{P})$ and associated Kripke structure $\mathcal{M}$. Let $\alpha \in Aut(\mathcal{C}(\mathcal{P}))$. If α is valid for $\mathcal{P}$ then $\alpha^* \in Aut(\mathcal{M})$.*

For a proof of this theorem see [11]. The theorem shows that automorphisms of the Kripke structure underlying a Promela program can be obtained by finding symmetries of the static channel diagram for the program. Note that for any LTL property ϕ under investigation, the never claim for ϕ is included with a model (see Section 2.1). It follows that ϕ is invariant under all valid automorphisms constructed in this way. Thus, by Theorem 1, the set of valid automorphisms is suitable for checking the property ϕ over a quotient structure.

The static channel diagram of a program is typically a small graph which can be easily extracted from the program. Additionally, checking for an element α of $Aut(\mathcal{C}(\mathcal{P}))$ whether or not $\alpha(\mathcal{P}) \equiv \mathcal{P}$, can be implemented efficiently (see Section 5.2). Thus, using Theorem 2, it is possible to quickly obtain a group of Kripke structure automorphisms, generated by the set $\{\alpha^* : \alpha \in S,\ \alpha(\mathcal{P}) \equiv \mathcal{P}\}$, where S is the set of generators for $Aut(\mathcal{C}(\mathcal{P}))$. However, this group may not be as large as possible. Consider the generating set for $Aut(\mathcal{C}(\mathcal{P}))$ given in Section 3.1, where $\mathcal{P}$ is the Promela description of the email example. The generators $(2\ 3)(box\_2\ box\_3)$ and $(3\ 4)(box\_3\ box\_4)$ are clearly not valid for $\mathcal{P}$. Let G be the group generated by the remaining generators. Thus $G = \langle(1\ 2)(box\_1\ box\_2), (4\ 5)(box\_4\ box\_5)\rangle$. Consider the group $G' = \langle(1\ 2)(box\_1\ box\_2), (2\ 4)(box\_2\ box\_4), (4\ 5)(box\_4\ box\_5)\rangle$. Each generator of G' is valid for $\mathcal{P}$, and $G \subset G'$ since $(2\ 4)(box\_2\ box\_4) \notin G$. Thus G is not the largest valid subgroup of $Aut(\mathcal{C}(\mathcal{P}))$.

4 Finding the largest valid subgroup of $Aut(\mathcal{C}(\mathcal{P}))$

In this section we establish that, for a Promela program $\mathcal{P}$, there is a unique, largest valid subgroup of $Aut(\mathcal{C}(\mathcal{P}))$. We then present an algorithm to find this subgroup. First we state some preliminary results, omitting the (very straightforward) proofs for space reasons.

Lemma 1. *Let $\alpha, \beta \in Aut(\mathcal{C}(\mathcal{P}))$. Suppose α and β are both valid for $\mathcal{P}$. Then $\alpha\beta$ is valid for $\mathcal{P}$.*

Corollary 1. *Let S be a set of generators for $Aut(\mathcal{C}(\mathcal{P}))$. Let $S' = \{\alpha \in S : \alpha$ is valid for $\mathcal{P}\}$. Then $\langle S' \rangle$ is valid for $\mathcal{P}$.*

Corollary 2. *Suppose $H \leq Aut(\mathcal{C}(\mathcal{P}))$ is valid for $\mathcal{P}$. Let $\alpha \in Aut(\mathcal{C}(\mathcal{P})), \alpha \notin H$ be valid for $\mathcal{P}$. Then $\langle H \cup \{\alpha\} \rangle$ is valid for $\mathcal{P}$.*

Using Lemma 1 we can prove that there is a unique largest valid subgroup of $Aut(\mathcal{C}(\mathcal{P}))$.

Theorem 3. *There is a group $K \leq Aut(\mathcal{C}(\mathcal{P}))$ such that K is valid for $\mathcal{P}$, and for any $H \leq Aut(\mathcal{C}(\mathcal{P}))$ which is also valid for $\mathcal{P}$, $H \leq K$.*

Proof. Let $\mathcal{X}$ be the set of all valid subgroups of $Aut(\mathcal{C}(\mathcal{P}))$. Since $Aut(\mathcal{C}(\mathcal{P}))$ is finite, $Aut(\mathcal{C}(\mathcal{P}))$ has a finite number of subgroups, therefore $\mathcal{X}$ is finite. Let $K = \langle \bigcup_{H \in \mathcal{X}} H \rangle$. Since every generator of K is valid for $\mathcal{P}$, it follows by Lemma 1 that K is valid for $\mathcal{P}$. Clearly $H \leq K$ for every $H \in \mathcal{X}$, i.e. $H \leq K$ for every valid subgroup H of $Aut(\mathcal{C}(\mathcal{P}))$.

Our algorithm for finding the largest valid subgroup of $Aut(\mathcal{C}(\mathcal{P}))$ involves starting with a known valid subgroup H of $Aut(\mathcal{C}(\mathcal{P}))$, and adding valid coset representatives to the generators of H to obtain successively larger valid subgroups. The following lemma is used to determine when the largest possible valid subgroup has been found.

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

Proof. Let K be the unique, largest valid subgroup of $Aut(\mathcal{C}(\mathcal{P}))$. By Theorem 3, $H \leq K$. Suppose $H \subset K$. Then there exists $\alpha \in K$ with $\alpha \notin H$. So $H\alpha$ is a right coset of H in $Aut(\mathcal{C}(\mathcal{P}))$, $H\alpha \neq H$, and $\alpha_i \in H\alpha$ for some $2 \leq i \leq k$. By hypothesis, α_i is not valid for $\mathcal{P}$. However, $H\alpha \subseteq K$ and $\alpha_i \in H\alpha$, so we have $\alpha_i \in K$. This is a contradiction since K is valid for $\mathcal{P}$. Hence $H = K$.

Algorithm 1 shows how the unique largest valid subgroup of $Aut(\mathcal{C}(\mathcal{P}))$ can be computed.

Theorem 4. *Algorithm 1 computes the largest valid subgroup of $Aut(\mathcal{C}(\mathcal{P}))$.*

Proof. By Corollaries 1 and 2, the group H computed by Algorithm 1 is valid for $\mathcal{P}$. The group H is the largest subgroup of $Aut(\mathcal{C}(\mathcal{P}))$ which is valid for $\mathcal{P}$ by Lemma 2.

We discuss the implementation and efficiency of Algorithm 1 in Section 5.3.

Algorithm 1 Algorithm to find the largest valid subgroup of $Aut(\mathcal{C}(\mathcal{P}))$

```
$S :=$ generators of $Aut(\mathcal{C}(\mathcal{P}))$
$H := \langle \{\alpha \in S : \alpha$ is valid for $\mathcal{P}\}\rangle$
$C :=$ representatives of right cosets of $H$ in $Aut(\mathcal{C}(\mathcal{P}))$ except $H$
while $C \neq \emptyset$ do
  $C := C \setminus \{\alpha\}$
  if $\alpha$ is valid for $\mathcal{P}$ then
    $H := \langle H \cup \{\alpha\}\rangle$
    if $|Aut(\mathcal{C}(\mathcal{P}))|/|H| < |C|$ then
      $C :=$ representatives of right cosets of $H$ in $Aut(\mathcal{C}(\mathcal{P}))$ except $H$
    end if
  end if
end while
```

5 The SymmExtractor tool

Given a Promela program $\mathcal{P}$, the SymmExtractor tool finds the largest subgroup of $Aut(\mathcal{C}(\mathcal{P}))$ which is valid for $\mathcal{P}$. By Theorem 2 this group induces a group of automorphisms of the underlying Kripke structure which can be used for symmetry reduction while model checking.

Our tool parses a Promela model and stores its abstract syntax tree using a set of Java classes generated by the SableCC compiler generation tool [12]. The grammar for Promela given in [14] was used as input to SableCC, and the SPIN source distribution was used to resolve ambiguities in this grammar.

SymmExtractor operates in four stages. In the first stage the given program is type-checked to ensure that variables of type *pid* and *chan* are used appropriately: for example that *pid* variables should only be assigned to, or compared for equality with, other *pid* variables or values. These restrictions are similar to those applied to variables of type *scalarset* in previous work on symmetry [15]. In the second stage the static channel diagram $\mathcal{C}(\mathcal{P})$ is constructed. In the third stage, the *saucy* program [9] is used to compute a set of generators for $Aut(\mathcal{C}(\mathcal{P}))$. Finally each generator α is checked for validity. Using Algorithm 1, the largest valid subgroup of $Aut(\mathcal{C}(\mathcal{P}))$ is computed.

5.1 Obtaining static channel diagram automorphisms from a Promela program

Extracting the static channel diagram from a Promela program is straightforward, and involves one pass over the abstract syntax tree. Each time a *proctype* definition appears in the program, the formal names of outgoing and incoming channels for that *proctype* are recorded. A new channel node is added to the static channel diagram for each static channel in the program. For each *run* statement, a new process node is added to the static channel diagram. The formal parameters of the *proctype* for the new process are substituted for the actual parameters provided in the *run* statement, and edges between processes and channels are added to the channel diagram according to the substituted outgoing and incoming channel names for the *proctype*.

Generators for $Aut(\mathcal{C}(\mathcal{P}))$ are computed using *saucy* [9]. The *saucy* program has been specifically designed for finding automorphisms of sparse graphs which correspond to instances of satisfiability problems. Since static channel diagrams are relatively sparse, the performance of *saucy* is generally very good.

5.2 Checking the validity of an element of $Aut(\mathcal{C}(\mathcal{P}))$

Applying a channel diagram automorphism α to $\mathcal{P}$ as described in Section 3.2 is trivial. Determining whether or not $\mathcal{P} \equiv \alpha(\mathcal{P})$ requires the use of a normalisation function. Recall that programs $\mathcal{P}_1$ and $\mathcal{P}_2$ are equivalent if they are identical up to rearrangement of: options in choice statements; operands to commutative operators; and `run` statements within the `init{atomic{...}}` block. The function *normalise* sorts the options in a choice statement, the operands of a commutative operator, and the sequence of `run` statements of the *init* process, using the natural ordering on strings. It is clear that if two programs are *equal* after normalisation then they are *equivalent*. The notions of equivalence and normalisation which we use here are basic but practical. It is easy to construct an example of an obscure program $\mathcal{P}$ such that an element $\alpha \in Aut(\mathcal{C}(\mathcal{P}))$ would not be deemed valid for the program, but would actually induce a valid automorphism of the Kripke structure for the program. However, for all Promela programs our lightweight approach to checking symmetries is safe and very fast, and is sufficient for sensibly written programs.

Checking the validity of an element α against $\mathcal{P}$ involves two passes over the abstract syntax tree for the program: one to apply the permutation, and one to normalise the program after the symmetry has been applied. The original program only needs to be normalised once when checking a set of generators.

5.3 Using GAP to compute the largest valid subgroup

The computational group theory package GAP is used to implement Algorithm 1. The Java and GAP components of the tool communicate using redirected standard input and output. Given a group G and a subgroup H of G, GAP provides a function to efficiently compute right coset representatives of H in G. The number of generators of $Aut(\mathcal{C}(\mathcal{P}))$ is typically small, and so initial generators for the valid group H are found quickly by checking each generator of $Aut(\mathcal{C}(\mathcal{P}))$ for validity against the program $\mathcal{P}$.

The algorithm performs badly if the initial group H is small, and $Aut(\mathcal{C}(\mathcal{P}))$ is very large. In such cases the number of right coset representatives to consider is, in the worst case, $|Aut(\mathcal{C}(\mathcal{P}))|/|H|$. Our implementation includes a heuristic which can be applied to try to combat this problem. If the size of the initial valid subgroup H can be increased, fewer coset representatives need to be considered. An initial approach involved taking a set X of random elements of $Aut(\mathcal{C}(\mathcal{P}))\setminus H$ and checking the validity of each element of X against $\mathcal{P}$, adding the valid ones to the generators of H. However, when $Aut(\mathcal{C}(\mathcal{P}))$ is large, the probability of a random element being valid for $\mathcal{P}$ may be small. In this case a better approach is, for each $\beta \in X$ and each generator α of H, to check the validity of the element $\beta^{-1}\alpha\beta$ (the *conjugate* of α by β), adding each valid element $\beta^{-1}\alpha\beta$ to the generators of H. Adding random conjugates to the generators of H works well in practice, because discarding invalid generators of $\mathit{Aut}(\mathcal{C}(\mathcal{P}))$ may result in a group which can permute disjoint sets of processes and channels, but cannot permute processes/channels which are in different sets. For example, if $\mathcal{P}$ is the email model, we found in Section 3.3 that the valid generators of $\mathit{Aut}(\mathcal{C}(\mathcal{P}))$ are (1 2)(*box_1 box_2*) and (4 5)(*box_4 box_5*). The group generated by these elements can swap processes 1 and 2, and processes 4 and 5 (similarly channels *box_1*, *box_2* and *box_4*, *box_5*), but *cannot* swap e.g. process 2 with process 4 and *box_2* with *box_4*, even though this permutation is valid for $\mathcal{P}$. The element (2 4)(*box_2 box_4*) is a valid element of $\mathit{Aut}(\mathcal{C}(\mathcal{P}))$ which bridges the gap between processes 1, 2 and 4, 5 (and their associated channels). While a random element drawn from $\mathit{Aut}(\mathcal{C}(\mathcal{P}))$ is unlikely to bridge this gap, a random conjugate of (1 2)(*box_1 box_2*) (for example) is more likely to do so, since a conjugate of an element which exchanges two processes (and associated channels) will also exchange two processes (and associated channels).

5.4 Applying SymmExtractor to the email example

Running SymmExtractor with our email example as input yields the following output:

```
>symmextractor email.pml
   Program is well typed.
   Finding the static channel diagram C(P).
   Computing the group Aut(C(P)) using saucy.
     Aut(C(P)) = <(box2 box4)(2 4),(5 4)(box4 box5),(box2 box3)(3 2),(3 1)(box3 box1)>
   H = <(box2 box1)(2 1),(5 4)(box4 box5),(box2 box4)(2 4)>
     is a valid group for symmetry reduction.
```

The generators of $\mathit{Aut}(\mathcal{C}(\mathcal{P}))$ found by SymmExtractor agree with our discussion in Section 3.1. Observe that the generator (2 3)(*box_2 box_3*) which we identified to be unsuitable for symmetry reduction in Section 3.2 does not belong to the group H of valid symmetries.

5.5 Extending SymmExtractor

As discussed in Section 2.1, our current approach only applies to models where all processes are instantiated by the *init* process: processes do not themselves instantiate child processes. Large classes of distributed systems can be modelled without dynamic process creation, so this restriction is not too limiting. However, the modelling of multi-threaded software applications often requires dynamic processes to model dynamic thread creation. Extending our approach to handle dynamic process creation will be challenging since the definition of a static channel diagram assumes a constant set of running processes. (Note that our approach *can* handle systems with dynamic *communication structures*, see Section 3.1.)

SymmExtractor cannot detect *data symmetries*, which arise as a result of indistinguishable data values in a protocol. We do not see this as a practical limitation. It is common practice when designing a verification model to abstract away from data [8] and to model only control messages. Indeed, a verification model which allows a range of data values to be communicated between processes, but for which the behaviour of processes is *independent* of the data values communicated, is usually a badly designed model [14].

Structural symmetries arising due to channel-based communication are detected by SymmExtractor. Promela also allows communication by shared variables. To capture symmetry between shared variables, the definition of a static channel diagram can be extended to include additional nodes for shared variables. In this case we add an edge from a process node to a variable node for each process that may write to the variable; and an edge from a variable node to a process node for each process that may read from the variable. The group $Aut(\mathcal{C}(\mathcal{P}))$ then indicates permutations of processes, channels, and shared variables which preserve the communication structure of the program $\mathcal{P}$. The check for validity described in Section 5.2 can be extended to deal with shared variables in a straightforward manner, and the group-theoretic approach of Section 4 can be used to find the largest valid subgroup of $Aut(\mathcal{C}(\mathcal{P}))$.

6 Experimental results

We have tested SymmExtractor on a variety of Promela models of distributed systems in addition to the email example described earlier. These models include: a token ring network [18]; a client-server system with load balancing [2]; control flow in a three-tiered architecture [18]; and a resource allocation system with two priority levels [17]. Table 1 shows the time taken for symmetry detection in each model, and the size of the resulting symmetry group. Experiments were performed on a PC with a 2.4GHz Intel Xenon processor, 3Gb of available main memory, running Linux (2.4.18), with GAP version 4.3. In all cases, the time taken for symmetry detection would be an acceptable overhead before search. All models have non-trivial symmetry groups of significant order. The theoretical maximum factor of reduction which may be obtained through symmetry reduction with a group G is $|G|$, since the orbit of a state s under G may have at most $|G|$ elements. The results of Table 1 show that the theoretical maximum factors of reduction for the models we have tested are large, though the results do not indicate the factors of reduction which will be achieved in practice.

There is no clear relationship between the time taken for symmetry detection and the number of symmetries detected. This is because the approach to symmetry detection depends on the number of generators of $Aut(\mathcal{C}(\mathcal{P}))$ rather than on the size of $Aut(\mathcal{C}(\mathcal{P}))$, and there is no direct relationship between the size of a group and the size of its generating set. Symmetry detection for the resource allocator example took longer than for the other models due to asymmetry in the model resulting from priority levels. Applying our "random conjugates" heuristic

model	time (s)	$\|G\|$
Token ring	2.52	10
Load balancer	2.70	432
Three-tiered	3.56	144
Resource allocator	7.44	576

Table 1. Symmetry detection results for some example models.

for this model reduced the time for symmetry detection to 4.4s. The overhead of launching Gap in each experiment was less than 1s.

7 Related work

The SymmSpin package [3] implements symmetry reduction techniques for Spin based on an approach using *scalarsets* [15]. Symmetry reduction techniques for Spin have also been implemented by adding extra keywords to the Promela language [10]. Neither approach to symmetry reduction can automatically detect symmetries of a model—the user needs to identify symmetry and annotate the model to indicate what symmetry is present. Our approach to symmetry detection is fully automatic. Deriving symmetry from the communication structure of a shared variable concurrent program is proposed, but not automated, in [6]. The idea of detecting symmetries by finding graph automorphisms has also been applied to boolean satisfiability problems [1].

In certain cases, *partial* symmetries of a model can be safely exploited to combat state-explosion during model checking [17]. Our tool cannot currently detect these partial symmetries.

8 Conclusions and future work

We have described an approach for detection of structural symmetry in Promela models, and presented a tool, SymmExtractor, to detect these symmetries automatically. Although our approach is specific to models specified in Promela, it can clearly be generalised to any graph-based modelling method.

Future work includes the implementation of symmetry reduction techniques based on these structural symmetries for the Spin model checker. Although symmetry reduction packages for Spin exist [3, 10], they are limited with respect to the kinds of symmetry they can exploit. Since our detection method can handle systems with arbitrary structural symmetries, it will be necessary to write a new symmetry reduction package for Spin. In particular, techniques for efficiently computing orbit representatives during model checking will be required for arbitrary symmetry groups. The SymmSpin tool [3] makes use of various heuristics in systems where there is full symmetry between components. We plan to use a computational group theory package such as Gap to classify the symmetry group of a model so that a suitable heuristic for symmetry reduction can be chosen.

Acknowledgments

The authors would like to thank Simon Gay, Warwick Harvey and Colva Roney-Dougal for their useful comments on this work.

References

1. F. Aloul, A. Ramani, I. Markov, and K. Sakallah. Solving difficult SAT instances in the presence of symmetry. *IEEE Transactions on Computer Aided Design*, 22(9):1117–1137, 2003.
2. J. Balasubramanian, D. Schmidt, L. Dowdy, and O. Othman. Evaluating the performance of middleware load balancing strategies. In *EDOC'01*, pages 135–146. IEEE Computer Society Press, 2004.
3. D. Bosnacki, D. Dams, and L. Holenderski. Symmetric Spin. *International Journal on Software Tools for Technology Transfer*, 4(1):65–80, 2002.
4. M. Calder and A. Miller. Generalising feature interactions in email. In *Feature Interactions in Telecommunications and Software Systems VII*, pages 187–205. IOS Press, 2003.
5. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* The MIT Press, 1999.
6. E. Clarke, E. Emerson, S. Jha, and A. Sistla. Symmetry reductions in model-checking. In *CAV'98*, LNCS 1427, pages 147–158. Springer-Verlag, 1998.
7. E. Clarke, R. Enders, T. Filkhorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1–2):77–104, 1996.
8. E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. In *POPL'92*, pages 343–354. ACM Press, 1992.
9. P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov. Exploiting structure in symmetry detection for CNF. In *DAC'04*, pages 530–534. ACM Press, 2004.
10. F. Derepas and P. Gastin. Model checking systems of replicated processes with Spin. In *SPIN'01*, LNCS 2057, pages 235–251. Springer-Verlag, 2001.
11. A. Donaldson, A. Miller, and M. Calder. Finding symmetry in models of concurrent systems by static channel diagram analysis. In *AVoCS'04*, ENTCS 128(6). Elsevier Science Publishers B.V, 2005 (to appear).
12. E. Gagnon and L. J. Hendren. SableCC, an object-oriented compiler framework. In *TOOLS'98*, pages 140–154. IEEE Computer Society Press, 1998.
13. The Gap Group. *GAP–Groups Algorithms and Programming, Version 4.2.* Aachen, St. Andrews, 1999. http://www-gap.dcs.st-and.ac.uk/~gap.
14. G. J. Holzmann. *The SPIN model checker: primer and reference manual.* Addison Wesley, 2003.
15. C. Ip and D. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9:41–75, 1996.
16. J. Rose. *A Course in Group Theory.* Dover Publications, 1964.
17. A. P. Sistla and P. Godefroid. Symmetry and reduced symmetry in model checking. *ACM Transactions on Programming Languages and Systems*, 25(4):702–734, 2004.
18. A. S. Tanenbaum and M. van Steen. *Distributed Systems Principles and Paradigms.* Prentice Hall, 2002.