Available online at www.sciencedirect.com
SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 128 (2005) 161–177
www.elsevier.com/locate/entcs

Finding Symmetry in Models of Concurrent Systems by Static Channel Diagram Analysis

Alastair F. Donaldson, Alice Miller, Muffy Calder

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

Abstract

Over the last decade there has been much interest in exploiting symmetry to combat the state explosion problem in model checking. Although symmetry in a model often arises as a result of symmetry in the topology of the system being modelled, most model checkers which exploit structural symmetry are limited to topologies which exhibit *total* symmetries, such as stars and cliques. We define the *static channel diagram* of a concurrent, message passing program, and show that under certain restrictions there is a correspondence between symmetries of the static channel diagram of a program and symmetries of the Kripke structure associated with the program. This allows the detection, and potential exploitation, of structural symmetry in systems with arbitrary topologies. Our method of symmetry detection can handle mobile systems where channel references are passed on channels, resulting in a dynamic communication structure. We illustrate our results with an example using the Promela modelling language.

Keywords: Model checking, symmetry, concurrency, distributed systems, formal verification, Promela/Spin.

1 Introduction

Model checking is an automated technique for the verification of concurrent systems [4]. To check whether or not a system satisfies a set of properties, an abstract, finite state model of the system is written using a specification language, and the properties are expressed as temporal logic formulae. A software tool called a *model checker* then searches the state space of the model, checking whether or not the properties hold at each state. If a violation of a property is found, the model checker returns a counter example path through the model which leads to the error. If the state space is exhaustively searched and no violations are found then the *model* satisfies the properties. As long

doi:10.1016/j.entcs.2005.04.010

as the model accurately specifies the behaviour of the system relevant to the properties, it can be concluded that the *system* satisfies the properties. Model checking is useful for finding bugs which have a rare probability of occurrence (and are therefore hard to detect), and a potentially catastrophic effect. As a result, model checking is particularly suitable for the verification of critical systems.

Model checking is hindered by the *state explosion problem*. This is where, as the number of components in a model increases, the state space of the model suffers combinatorial growth, quickly becoming too large to feasibly check. Much research in model checking concentrates on methods to tackle the state explosion problem. Such methods include symbolic representation of states, abstraction, partial order reduction, and induction (see [4] for example). Another approach exploits symmetry inherent in the system [1,2,5,6,11]. Concurrent systems often contain many replicated components and, as a consequence, model checking may involve making a redundant search over equivalent areas of the state space. Symmetry reduction techniques involve restricting the search to equivalence class representatives, and often result in significant savings in memory and verification time [1,5,6]. However, most model checkers which exploit structural symmetry are restricted in two ways. First, they are limited to topologies which exhibit *total* symmetries, such as stars and cliques. Second, they rely on the *user* to specify information about symmetry in the model. This is potentially error prone, and compromises the automation of model checking, which is one of its main strengths as a verification technique.

We present an approach to the detection of structural symmetries in models of message passing systems with *arbitrary* communication structures. Our approach involves analysing the *static channel diagram* of the system being modelled. Symmetry detection using this approach can be fully automated, and requires no additional information from the user—the only requirement is that models satisfy certain restrictions, which can be automatically checked. Future work will be to implement symmetry reduction using these structural symmetries.

1.1 Overview of results

We define the *static channel diagram* of a concurrent program, and show that, under certain restrictions, there is a correspondence between automorphisms of the static channel diagram of a program and automorphisms of the Kripke structure associated with the program. Thus automorphisms of an intractably large Kripke structure can be obtained from the static channel diagram of the program, which is typically a small graph. The static channel diagram can be automatically determined with complexity linear in the size of the program. The restrictions can be checked with complexity $O(k(|X| + N))$, where N is the size of the program, X is the set of variables in the program, and k is the number of generators of the automorphism group of the static channel diagram. Our approach can handle programs where channel references are passed on channels, leading to a dynamic communication structure.

2 Concurrent programs

Throughout the paper we use ':=' to denote the assignment operator, and '=' to denote the boolean operator which tests equality. Let D be a finite data domain, and let $\perp \in D$ denote an *undefined* value.

Definition 2.1 A concurrent program $\mathcal{P}$ consists of:

- a set of concurrently executing processes $\{p_i \mid 1 \leq i \leq m\}$ for some $m \geq 1$,
- a set of communication channels $\{c_i \mid m < i \leq m+n\}$ for some $n \geq 0$,
- a finite set X of local variables, which take values from D. A local variable of process p_i $(1 \leq i \leq m)$ is denoted by an identifier subscripted by i, e.g. x_i. The set of local variables of p_i is denoted X_i. Variables in X are divided into three types: *p-variables*, the values of which are process indices drawn from the set $\{\perp, 1, \ldots, m\}$; *c-variables*, the values of which are channel indices drawn from the set $\{\perp, m+1, \ldots, m+n\}$; and *standard variables* (variables which are not p-variables or c-variables).
- a finite set of program statements of the form $g_i \to u_i$, where $i \in \{1, \ldots, m\}$ is a process index, g_i is a boolean guard, and u_i is an update to variables of p_i and channels of the program.
- a mapping *type* which maps each process index to a *process type*, and each channel index to a *channel type*.
- an initialisation function $\mathit{init} : X \to D$ which assigns each variable in X to an initial value in D.

Variables may initially be assigned to the undefined value $\perp$. Two processes p_i and p_j $(1 \leq i, j \leq m)$ are of the same type (i.e. $\mathit{type}(i) = \mathit{type}(j)$) if they are instantiations of the same parameterised process definition. Two channels are of the same type if they have the same *capacity*, and hold values of the same data type (p-variables, c-variables, or standard variables). Channels in the program may only hold values of a single data type. Note that in allowing channels to hold c-variable values we can handle systems with dynamic communication structures.

Associated with $\mathcal{P}$ is a set AP of atomic propositions, which we now define.

For each variable $x \in X$, and for each $d \in D$, $(x = d) \in AP$ (these propositions refer to the values of variables in the program). For each channel c_i in $\mathcal{P}$, let $cap(c_i)$ denote the maximum number of messages which c_i can hold, $len(c_i)$ the number of messages that c_i holds at a given time, and $next(c_i)$ the next message to be read from c_i. Then for values $d_1, \ldots, d_k \in D$, $m < i \leq m + n$, $0 \leq k \leq cap(c_i)$, and for any k-tuple $[d_1, \ldots, d_k] \in D^k$, $(c_i = [d_1, \ldots, d_k]) \in AP$, $(next(c_i) = d_1) \in AP$ and $(len(c_i) = k) \in AP$ (these propositions refer to the contents and lengths of channels in the program).

Here $[d_1, \ldots, d_k]$ denotes a first-in-first-out buffer containing k elements (we use $[\,]$ to denote an empty buffer). When a message is written to a channel, it is added to the right of the buffer. When a message is read from a channel, it is removed from the left of the buffer, and all messages shift one place to the left.

The execution of the program $\mathcal{P}$ is determined by its set of statements. For a statement $g_i \rightarrow u_i$, in any system state where the guard g_i evaluates to true, the program may execute the update u_i, resulting in a transition to another system state. A guard g_i is a boolean combination of atomic propositions referring to variables of process p_i, or to the length or next value of a channel.

An update u_i is an assignment to local variables of process p_i, and to channels of the system. An update u_i can have the form *skip* (no values of variables or channels are updated), $x_i := d$ ($x_i \in X_i$, $d \in D$), or may involve a *static* channel update (read from or write to a fixed channel c_j), or a *dynamic* channel update. A dynamic channel update involves reading from or writing to channel c_{x_i}, where $x_i \in X_i$ is a c-variable. We use the notation $read(c_j)$ and $write(d, c_j)$ ($m < j \leq m+n, d \in D$) to denote a static read from and a write to channel c_j respectively (and similarly for a dynamic read/write). An update can also consist of a sequence of these updates, executed simultaneously.

Note that a model expressed as a *sequence* of statements (for example, a model written in the Promela specification language [10]) can still be thought of in these terms. If k is the program counter value associated with a statement of a Promela model executed by process p_i, then the guard associated with this statement contains the proposition $(pc_i = k)$, where pc_i is the local variable representing the program counter of process p_i.

2.1 Deriving a Kripke structure from a concurrent program

To reason about the formal semantics of a concurrent program $\mathcal{P}$ we use a Kripke structure [4].

Definition 2.2 Let $\mathcal{P}$ be a concurrent program with atomic propositions AP. The *Kripke structure* $\mathcal{M}$ over AP for $\mathcal{P}$ is a quadruple $\mathcal{M} = (S, R, L, s_0)$

where:

- S is a finite set of program states, consisting of all possible assignments to variables and channels.
- $R \subseteq S \times S$ is a transition relation. For a state $s \in S$ and a program statement $g_i \rightarrow u_i$ $(1 \leq i \leq n)$, if g_i holds in s then $(s,t) \in R$, where $t \in S$ is the state resulting from the update u_i.
- $L : S \rightarrow 2^{AP}$ is a mapping that labels each state in S with the values of variables, contents of channels and lengths of channels in that state.
- $s_0 \in S$ is the initial state of the program.

The initial state s_0 of the program is labelled as follows:

$$L(s_0) = \bigcup_{i=m+1}^{m+n} \{(c_i = []), (len(c_i) = 0), (next(c_i) = \perp)\} \cup \{(x = init(x) \mid x \in X\}$$

We now define the effect of statement execution on the Kripke structure for a program. Let $s \in S$, and $g_i \rightarrow u_i$ a statement of process p_i $(1 \leq i \leq m)$. Applying u_i to s results in a state $t \in S$. If u_i has the form $skip$ then $t = s$. Suppose u_i consists of a single assignment $x_i := d'$ $(x_i \in X_i, d' \in D)$. Then t is defined by:

$$L(t) = \big(L(s) \setminus \{(x_i = d)\}\big) \cup \{x_i = d'\},$$

where $d, d' \in D$ are the values of x_i before and after the update respectively. Suppose that, for channel c_j, $(c_j = [d_1, \ldots, d_k]) \in L(s)$, where $m < j \leq m+n$, $d_l \in D$ $(1 \leq l \leq k)$, and $0 \leq k \leq cap(c_j)$. If the update u_i consists of a static channel write of the form $write(d, c_j)$ $(d \in D)$ then t is defined by:

$$\begin{aligned} L(t) = \big(&L(s) \setminus \{(c_j = [d_1, \ldots, d_k]), (len(c_j) = k)\}\big) \cup \\ &\{(c_j = [d_1, \ldots, d_k, d]), (len(c_j) = k+1)\}. \end{aligned}$$

If u_i is a static channel read, namely $read(c_j)$, then t is defined by:

$$\begin{aligned} L(t) = \big(&L(s) \setminus \{(c_j = [d_1, \ldots, d_k]), (len(c_j) = k), (next(c_j) = d_1)\}\big) \\ &\cup \{(c_j = [d_2, \ldots, d_k]), (len(c_j) = k-1), (next(c_j) = d_2)\}. \end{aligned}$$

If the update is a dynamic channel update involving channel c_{x_i}, where $x_i \in X_i$ is a c-variable of process p_i, suppose $(x_i = j) \in L(s)$ for some $m < j \leq m+n$. Then the update involves channel c_j, and the state t is defined in the same way as for a static channel update. If u_i is a sequence of updates, executed simultaneously, then the state t is determined by applying each update in the sequence in order.

3 Symmetry and model checking

In this section we present some basic group theoretic definitions, and summarise the theory of symmetry reduction in model checking. For a thorough introduction to symmetry reduction in model checking see e.g. [6].

Definition 3.1 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 $x \circ x^{-1} = x^{-1} \circ x = 1_G$.

When it is clear what the binary operation $\circ$ 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 3.1, 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 $g_1, \ldots, g_n, g_1^{-1}, \ldots, g_n^{-1}$ is denoted $\langle g_1, g_2, \ldots, g_n \rangle$. This set is a subgroup of G, called the subgroup *generated* by $g_1, g_2, \ldots, g_n$.

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:[1]

- $\forall s, t \in S, \ \ (s, t) \in R \Rightarrow (\alpha(s), \alpha(t)) \in R,$
- $\alpha(s_0) = s_0$

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 $\mathcal{M}$ 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 3.2 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:

- $S_G = \{[s] : s \in S\}$ (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 equiv-

[1] These conditions define automorphisms of the underlying transition system only, as invariance under labellings isn't needed in our approach.

alent 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 [4].

Theorem 3.3 *Let $\mathcal{M}$ be a Kripke structure, G be a subgroup of of $Aut(\mathcal{M})$ and f be a CTL* formula. If f is invariant under the group G then*

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

where $\mathcal{M}_G$ is the quotient structure corresponding to $\mathcal{M}$.

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 [1,5,6].

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 [13]. 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. If automorphisms of a Kripke structure can be identified in advance, then a quotient structure can be incrementally constructed using Algorithm 1, even if the original structure is intractable.

Algorithm 1 Algorithm to construct a quotient Kripke structure

```
reached := {rep(s0)}
unexplored := {rep(s0)}
while unexplored ≠ ∅ do
  remove a state s from unexplored
  for all successor states q of s do
    if rep(q) is not in reached then
      append rep(q) to reached
      append rep(q) to unexplored
    end if
  end for
end while
```

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 [5]. In Sections 4 and 5 we define the *static channel* *diagram* of a concurrent program $\mathcal{P}$, and show that, under certain restrictions, automorphisms of the static channel diagram of $\mathcal{P}$ give rise to automorphisms of the Kripke structure associated with $\mathcal{P}$.

4 Static channel diagrams

Let $\mathcal{P}$ be a concurrent program as defined in Section 2.

Definition 4.1 The *static channel diagram* corresponding to the concurrent program $\mathcal{P}$ is a directed, coloured graph $\mathcal{C}(\mathcal{P}) = (V, E, C)$ where:

- $V = V_P \cup V_C$ is the set of indices of processes and channels in the program: $V_P = \{1, \ldots, m\}$ and $V_C = \{m+1, \ldots, m+n\}$.
- for $i \in V_P$ and $j \in V_C$, $(i, j) \in E$ if and only if there is a statement $g_i \rightarrow u_i$ in $\mathcal{P}$ where u_i involves a static channel write update on the channel c_j; for $i \in V_C$ and $j \in V_P$, $(i, j) \in E$ if and only if there is a statement $g_j \rightarrow u_j$ in $\mathcal{P}$ where u_j involves a static channel read update on the channel c_i,
- C is a colouring function defined by, for all $i \in V$, $C(i) = \mathit{type}(i)$ (see Definition 2.1).

In [7] we present a similar definition of the *channel diagram* of a concurrent program. In a channel diagram there is an edge between i and j if it is *possible* for process p_i to write to or read from channel c_j. The *static* channel diagram differs in two ways. First, it does not capture *dynamic* communication, which arises from dynamic channel updates of the form $write(d, c_{x_i})$ and $read(c_{x_i})$, where $x_i \in X_i$ is a c-variable. Second, suppose the program has a statement $g_i \rightarrow u_i$ such that u_i updates channel c_j, but that the guard g_i does not evaluate to true in *any* state of the system. Then the update u_i will give rise to an edge in the static channel diagram even though it is *impossible* for it to be executed. These differences mean that the static channel diagram can be found by straightforward analysis of the program $\mathcal{P}$—it is not necessary to establish the possible run time values for each c-variable, or to check for guards which will never be executable. The static channel diagram corresponding to a concurrent program $\mathcal{P}$ can be constructed using Algorithm 2, which has complexity linear in the size of $\mathcal{P}$.

Proposition 4.2 *Let $\mathcal{P}$ be a concurrent program, and let N be the number of statements in $\mathcal{P}$. Then the complexity of Algorithm 2 is $O(N)$.*

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

$$\forall i, j \in V,\ \ (i, j) \in E \Rightarrow (\alpha(i), \alpha(j)) \in E,\ \ \mathit{and}\ \ \forall i \in V,\ \ C(i) = C(\alpha(i)).$$

Algorithm 2 Algorithm for finding the static channel diagram $\mathcal{C}(\mathcal{P})$ of a concurrent program $\mathcal{P}$.

$V := \{1, 2, \ldots, m+n\}$, $E := \emptyset$, $C := type$
for all $(g_i \to u_i) \in \mathcal{P}$ **do**
  if u_i involves a channel write update on c_j **then**
    $E := E \cup \{(i, j)\}$
  else if u_i involves a channel read update on c_j **then**
    $E := E \cup \{(j, i)\}$
  end if
end for

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 $Aut(\mathcal{C}(\mathcal{P}))$.

5 Correspondence result

In this section we present the main theorem of the paper. For convenience, proofs have been collected in an appendix at the end of the paper. We show that an automorphism α of the static channel diagram corresponding to a program $\mathcal{P}$ can be used to define a permutation α^* of the state set S of the Kripke structure $\mathcal{M}$ associated with $\mathcal{P}$, and that under Restrictions 1 and 2, α^* is an automorphism of $\mathcal{M}$. Thus:

Restriction 1 *Let $g_i \to u_i$ be a statement in $\mathcal{P}$. Then for all $\alpha \in Aut(\mathcal{C}(\mathcal{P}))$, $g_{\alpha(i)} \to u_{\alpha(i)}$ must also be a statement in $\mathcal{P}$.*

Restriction 2 *Let $\alpha \in Aut(\mathcal{C}(\mathcal{P}))$. The init function which assigns initial values to the variables of $\mathcal{P}$ must be such that for each $x_i \in X$ $(1 \leq i \leq m)$, $init(x_{\alpha(i)}) = init(x_i)^\alpha$, where $init(x_i)^\alpha = init(x_i)$ if x_i is a standard variable, and $init(x_i)^\alpha = \alpha(init(x_i))$ otherwise.*

Restriction 1 assures that statements of the program are *closed* under the elements of $Aut(\mathcal{C}(\mathcal{P}))$, and Restriction 2 that variables of the system must be initialised symmetrically.

Theorem 5.1 *Let $\mathcal{P}$ be a concurrent program which satisfies Restrictions 1 and 2. Let $\mathcal{C}(\mathcal{P})$ be the static channel diagram of $\mathcal{P}$, and let $\mathcal{M}$ be the Kripke structure associated with $\mathcal{P}$. Let $G = \{\alpha^* : \alpha \in Aut(\mathcal{C}(\mathcal{P}))$. Then $G \leq Aut(\mathcal{M})$.*

This means that if a program $\mathcal{P}$ satisfies Restrictions 1 and 2, (symmetry reduced) model checking can be performed over the quotient structure $\mathcal{M}_G$. Since $\mathcal{C}(\mathcal{P})$ is typically a small graph, the group $Aut(\mathcal{C}(\mathcal{P}))$, and hence the group G, can be found quickly using a standard algorithm [12].

In order to define the permutation α^* acting on the states of $\mathcal{M}$, we first define the action of α on an atomic proposition $p \in AP$. Suppose $p = (x_i = d)$ for some $x_i \in X_i$, $d \in D$, $1 \leq i \leq m$. Then $\alpha(p) = (x_{\alpha(i)} = d^\alpha)$, where $d^\alpha = d$ if x_i is a standard variable, and $d^\alpha = \alpha(d)$ otherwise. If $p = (c_i = [d_1, \ldots, d_k])$ for some channel c_i ($m < i \leq m+n$), $d_l \in D$ ($1 \leq l \leq k$), and $0 \leq k \leq cap(c_i)$, then $\alpha(p) = (c_{\alpha(i)} = [d_1^\alpha, \ldots, d_k^\alpha])$, where $d_l^\alpha = d_l$ if the channel c_i holds standard variable values, and $d_l^\alpha = \alpha(d_l)$ otherwise ($1 \leq l \leq k$). If $p = (len(c_i) = k)$ for some channel c_i ($m < i \leq m+n$) and $0 \leq k \leq cap(c_i)$ then $\alpha(p) = (len(c_{\alpha(i)}) = k)$. If $p = (next(c_i) = d)$ for some channel c_i ($m < i \leq m+n$) and $d \in D$, then $\alpha(p) = (next(c_{\alpha(i)}) = d^\alpha)$, where $d^\alpha = d$ if c_j holds standard variable values, and $d^\alpha = \alpha(d)$ otherwise. We define $\alpha(\perp) = \perp$, hence $\alpha(x = \perp) = (\alpha(x) = \perp)$ for all $x \in X$.

We now define the permutation α^* on a state $s \in S$. Recall that a state is uniquely defined by a labelling in terms of atomic propositions. The state $\alpha^*(s)$ is defined as follows:

$$L(\alpha^*(s)) = \{\alpha(p) \mid p \in L(s)\}.$$

It is clear that α^* is indeed a permutation of the set S.

To prove Theorem 5.1, we must consider the action of $\alpha \in Aut(\mathcal{C}(\mathcal{P}))$ on the guards and updates of the program. The action of α on a guard g_i is defined inductively (on the atomic propositions contained in g_i).

Lemma 5.2 *Let $s \in S$ be a state of the program $\mathcal{P}$. Let $\alpha \in Aut(\mathcal{C}(\mathcal{P}))$, and let g_i be a guard. Then g_i holds at s iff $g_{\alpha(i)}$ holds at $\alpha^*(s)$.*

Proof. See appendix. □

Let u_i be an update in the program $\mathcal{P}$. Suppose u_i is a variable update of the form $x_i := d$ ($x_i \in X_i$, $d \in D$). Then the update $u_{\alpha(i)}$ has the form $x_{\alpha(i)} := d^\alpha$, where $d^\alpha = d$ if x_i is a standard variable, and $d^\alpha = \alpha(d)$ otherwise. If u_i is a static channel update $write(d, c_j)$ or $read(c_j)$, for some channel c_j ($m < j \leq m+n$) and $d \in D$, then $u_{\alpha(i)}$ is a static channel update $write(d^\alpha, c_{\alpha(j)})$ or $read(c_{\alpha(j)})$ respectively. Similarly, if u_i is a dynamic channel update $write(d, c_{x_i})$ or $read(c_{x_i})$, for some c-variable $x_i \in X_i$ and $d \in D$, then $u_{\alpha(i)}$ is a dynamic channel update $write(d^\alpha, c_{x_{\alpha(i)}})$ or $read(c_{x_{\alpha(i)}})$. If u_i has the form *skip* then $u_{\alpha(i)}$ also has the form *skip*. Finally, if u_i is a sequence of updates executed simultaneously, then $u_{\alpha(i)}$ is the sequence of corresponding updates obtained by applying the above rules. We now prove that if executing update u_i from state s leads to state t, then executing update $u_{\alpha(i)}$ from state $\alpha^*(s)$ leads to state $\alpha^*(t)$.

Algorithm 3 Algorithm to find a subgroup of $Aut(\mathcal{C}(\mathcal{P}))$ which satisfies Restrictions 1 and 2.

```
gens := generators of Aut(C(P))
for all x_i ∈ X do
  for all α ∈ gens do
    if init(x_α(i)) ≠ init(x_i)^α then
      gens := gens \ {α}
    end if
  end for
end for
for all (g_i → u_i) ∈ P do
  for all each α ∈ gens do
    if g_α(i) → u_α(i) ∉ P then
      gens := gens \ {α}
    end if
  end for
end for
```

Lemma 5.3 *Let $s \in S$ be a state of the program $\mathcal{P}$, and let $\alpha \in Aut(\mathcal{C}(\mathcal{P}))$. If $s \to t \in R$ is a transition associated with update u_i, and $\alpha^*(s) \to t'$ is the corresponding transition associated with update $u_{\alpha(i)}$, then $t' = \alpha^*(t)$.*

Proof. See appendix. □

Using Lemmas 5.2 and 5.3, Theorem 5.1 follows. The proof is given in the appendix. To see why Restriction 1 of Theorem 5.1 is necessary, suppose that $(x_1 = 2) \to (y_1 := 3)$ is a statement of a concurrent program $\mathcal{P}$, where x_1 and y_1 are p-variables (of process p_1). Suppose $\alpha = (1\ 2)(2\ 3) \in Aut(\mathcal{C}(\mathcal{P}))$. Then processes p_1 and p_2 must have the same process type, so are instantiations of the same parameterised process. Therefore x_2 and y_2 are p-variables of process p_2, and $(x_2 = 2) \to (y_2 := 3)$ is also a statement of the program. However, applying α to the statement $(x_1 = 2) \to (y_1 := 3)$ gives the statement $(x_{\alpha(1)} = \alpha(2)) \to (y_{\alpha(1)} := \alpha(3))$ (since x_1 and y_1 are p-variables), which is the statement $(x_2 = 1) \to (y_2 := 2)$. This statement may *not necessarily* belong to the program $\mathcal{P}$.

To check Restriction 1 it is sufficient to check, for each *generator* α of $Aut(\mathcal{C}(\mathcal{P}))$ and each statement $g_i \to u_i$ in $\mathcal{P}$, that $g_{\alpha(i)} \to u_{\alpha(i)}$ is also a statement in $\mathcal{P}$ (any element of $Aut(\mathcal{C}(\mathcal{P}))$ can be expressed as a product of generators). Similarly, Restriction 2 can be checked using only the generators of $Aut(\mathcal{C}(\mathcal{P}))$. Let H be the subgroup of $Aut(\mathcal{C}(\mathcal{P}))$ generated by the subset of generators of $Aut(\mathcal{C}(\mathcal{P}))$ which satisfy Restrictions 1 and 2. Let $K = \{\alpha^* \mid \alpha \in H\}$. It is clear from the proof of Theorem 5.1 that in this case $K \leq Aut(\mathcal{M})$.

Algorithm 3 shows how the largest subset of a given set of generators for $Aut(\mathcal{C}(\mathcal{P}))$ which satisfy Restrictions 1 and 2 can be found.

Proposition 5.4 *Let $\mathcal{P}$ be a concurrent program with variable set X and static channel diagram $\mathcal{C}(\mathcal{P})$. Suppose $\mathcal{P}$ has N statements, and $Aut(\mathcal{C}(\mathcal{P}))$ has k generators. Then the complexity of Algorithm 3 is $O(k(|X| + N))$.*

6 Load balancing example

To illustrate the theory we now give an example of a model of a concurrent message passing system. The model consists of three *server* processes, six *client* processes, and two *load balancer* processes. A load balancer process continuously receives messages sent by client processes, and forwards each message to the server process with the shortest queue of incoming messages. The message received by a load balancer process from a client is a reference to the incoming channel of the client. The load balancer passes this reference on to the chosen server, and the server sends a message back to the client using the channel reference. Thus the model has a dynamic communication structure. We have implemented this model using Promela, the input language to the SPIN model checker [10], and the code is available on our website [3].

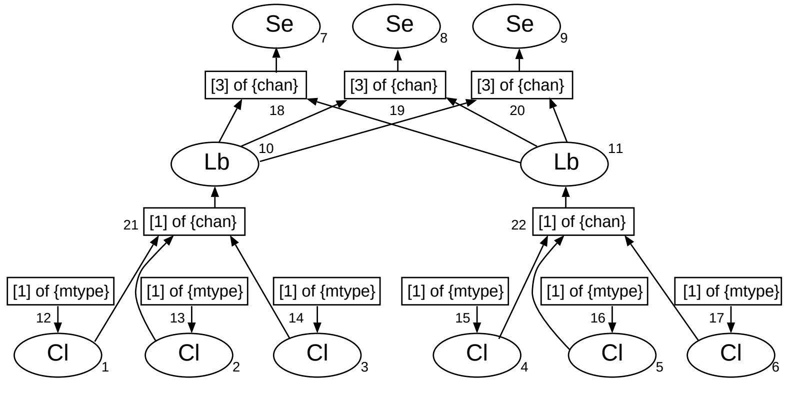

Fig. 1. Channel diagram of the load balancing example—clients are denoted Cl, servers Se and load balancers Lb.

We have written a tool which finds the static channel diagram of a Promela model using Algorithm 2, and uses the *nauty* algorithm [13] to compute the group of static channel diagram automorphisms. Figure 1 shows a graphical representation of the channel diagram found by our tool. Processes are represented by ovals, channels by rectangles, and types by textual labels. The label $\{chan\}$ indicates that a channel holds c-variable values, and the label

$\{mtype\}$ indicates that a channel holds values of an enumerated message type. Note that there are no outgoing edges from the server processes. This is because communication from a server process to a client channel is achieved *dynamically*, using the reference passed to the server by one of the load balancer processes. Using the *nauty* algorithm, our tool finds the generators of $Aut(\mathcal{C}(\mathcal{P}))$ as follows:

$$\begin{aligned} Aut(\mathcal{C}(\mathcal{P})) = \langle & (5\ 6)(16\ 17), (4\ 5)(15\ 16), (2\ 3)(13\ 14), \\ & (1\ 2)(12\ 13), (8\ 9)(19\ 20), (7\ 8)(18\ 19), \\ & (1\ 4)(2\ 5)(3\ 6)(10\ 11)(12\ 15)(13\ 16)(14\ 17)(21\ 22)\rangle \end{aligned}$$

To see how the first generator acts on the static channel diagram shown in Figure 1, observe that swapping (client) processes 5 and 6, and simultaneously swapping channels 16 and 17, preserves the structure of the graph. Inputting the generators to the group theoretic package GAP [9] reveals that, for this example, $Aut(\mathcal{C}(\mathcal{P}))$ has 864 elements.

We have not yet implemented Algorithm 3 to check Restrictions 1 and 2 of Theorem 5.1. However, in this example it is clear from the Promela code that they are satisfied. By Theorem 5.1, a subgroup of $Aut(\mathcal{M})$ is derivable from $Aut(\mathcal{C}(\mathcal{P}))$. This group of automorphisms could be exploited during model checking.

7 Related work

In [5], a result similar to Theorem 5.1 is presented for a shared variable model of communication—the automorphism group of the *coloured hypergraph* of a concurrent, shared variable program is shown to be a subgroup of the automorphism group of the underlying Kripke structure. In a sense the static channel diagram of a concurrent message passing program is analogous to the coloured hypergraph of a concurrent, shared variable program. The definition of a static channel diagram is adapted from the definition of a channel diagram originally presented in [14], and used in [7]. The SymmSpin package [1] adds symmetry reduction to the SPIN model checker using the scalarset approach of Ip and Dill [11]. However, scalarsets only allow the exploitation of *total* symmetries, and require the modeller to specify symmetries using the scalarset data type. The problem of exploiting symmetry reduction while model checking under fairness constraints is the focus of the SMC tool [15], and in [8], the problem of exploiting *partial* symmetries of systems is considered.

8 Conclusions and future work

We have defined the *static channel diagram* $\mathcal{C}(\mathcal{P})$ of a concurrent, message passing program $\mathcal{P}$, and have proved that, under certain restrictions, the group $Aut(\mathcal{C}(\mathcal{P}))$ of automorphisms of $\mathcal{C}(\mathcal{P})$ allows us to derive a subgroup of $Aut(\mathcal{M})$, the group of automorphisms of the Kripke structure $\mathcal{M}$ associated with $\mathcal{P}$. We have shown that the static channel diagram can be automatically extracted with complexity linear in the size of $\mathcal{P}$, and the restrictions can be checked automatically with complexity $O(k(|X|+N)$, where N is the size of $\mathcal{P}$, X the set of variables in $\mathcal{P}$, and k the number of generators of $Aut(\mathcal{C}(\mathcal{P}))$. The modeller does not need to provide information about symmetry in the model, so the approach does not require error prone, manual effort. We describe a tool which automatically finds automorphisms of the static channel diagram of a Promela model, and give an example model of a client-server system with load balancing. Our symmetry detection technique can handle systems which exhibit mobility, i.e. systems where channel references are passed on channels. To our knowledge, no other published work on symmetry detection can handle such systems. Our results show that symmetry reduction for message passing models can potentially be a "push button" reduction technique.

We intend to implement symmetry reduction in the SPIN model checker [10] using this approach to symmetry detection. Exploiting symmetry effectively during search is made difficult by the problem of computing representatives of states. For the exploitation of *total* symmetries, heuristics have been shown to be effective in solving this problem [1], and for certain kinds of symmetry groups, unique representatives can be computed with complexity polynomial in the number of processes [5]. Since our approach can detect symmetries of arbitrary communication structures, an important area of future work will be to try to identify heuristics which are more generally applicable. We will also try to extend our approach to detect *partial* symmetries of models, as investigated by Emerson et al. [8].

References

[1] D. Bosnacki, D. Dams, and L. Holenderski. Symmetric spin. *International Journal on Software Tools for Technology Transfer*, 4(1):65–80, 2002.

[2] M. Calder and A. Miller. Five ways to use induction and symmetry in the verification of networks of processes by model-checking. In *AVoCS '02*.

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

[4] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, Cambridge, Masachusetts, 1999.

[5] E.M. Clarke, E.A. Emerson, S. Jha, and A.P. Sistla. Symmetry reductions in model-checking. In *CAV '98*, pages 147–158, 1998.

[6] E.M. 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.

[7] A. Donaldson, A. Miller, and M. Calder. SPIN-to-GRAPE: a tool for analysing symmetry in promela models. In *ARTS '04*.

[8] E. Allen Emerson, John W. Havelick, and Richard J. Trefler. Virtual symmetry reduction. In *LICS '00*, pages 121–131, Santa Barbara, California, USA, 1995.

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

[10] Gerard J. Holzmann. *The SPIN model checker: primer and reference manual*. Addison Wesley, Boston, 2003.

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

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

[13] Brendan D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

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

[15] A. Prasad Sistla, Viktor Gyuris, and E. Allen 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.

Appendix—Proofs Omitted From the Text

Proof of Lemma 5.2. If $g_i = \mathit{true}$ the result holds trivially. If $g_i = p$ for some $p \in AP$, then $g_{\alpha(i)} = \alpha(p)$. Now $p \in L(s) \Leftrightarrow \alpha(p) \in L(\alpha^*(s))$ by definition of $\alpha^*(s)$, so the result holds. If $g_i = \neg h_i$ for some propositional h_i, $g_{\alpha(i)} = \neg\alpha(h_i)$. We have

$$\begin{aligned}\neg h_i \in L(s) &\Leftrightarrow p \notin L(s)\ \forall p \in h_i\\ &\Leftrightarrow \alpha(p) \notin L(\alpha^*(s))\ \forall p \in h_i\\ &\Leftrightarrow \neg\alpha(p) \in L(\alpha^*(s))\ \forall p \in h_i\\ &\Leftrightarrow \neg h_{\alpha(i)} \in L(\alpha^*(s))\end{aligned}$$

so the result holds. The cases where $g_i = h_i \wedge k_i$ and $g_i = h_i \vee k_i$ for propositional subformulae h_i and k_i follow using structural induction.

Proof of Lemma 5.3. Suppose u_i is a variable update $x_i := d$ ($x_i \in X_i, d \in D$), and suppose $(x_i = e) \in L(s)$ ($e \in D$). Then $u_{\alpha(i)}$ is a variable update $x_{\alpha(i)} := d^\alpha$, and $(x_{\alpha(i)} = e^\alpha) \in L(\alpha^*(s))$. We have $L(t') = (L(\alpha^*(s)) \setminus \{(x_{\alpha(i)} = e^\alpha)\}) \cup \{(x_{\alpha(i)} = d)\} = (L(\alpha^*(s)) \setminus \{\alpha(x_i = e)\}) \cup \{\alpha(x_i = d)\} = L(\alpha^*(t))$. Therefore $t' = \alpha^*(t)$.

Suppose u_i is a static channel write update $write(d, c_j)$ for some channel c_j $(m < j \leq m+n)$, $d \in D$, and suppose $(c_j = [d_1, \ldots, d_k]), (len(c_j) = k) \in L(s)$. Then $u_{\alpha(i)}$ is a channel write update $write(d^\alpha, c_{\alpha(j)})$, and $(c_{\alpha(j)} = [d_1^\alpha, \ldots, d_k^\alpha]), (len(c_{\alpha(j)}) = k) \in L(\alpha^*(s))$. We have

$$\begin{aligned} L(t') &= (L(\alpha^*(s)) \setminus \{(c_{\alpha(j)} = [d_1^\alpha, \ldots, d_k^\alpha]), (len(c_{\alpha(j)}) = k)\}) \\ &\qquad \cup \{(c_{\alpha(j)} = [d_1^\alpha, \ldots, d_k^\alpha, d^\alpha]), (len(c_{\alpha(j)}) = k+1)\} \\ &= (L(\alpha^*(s)) \setminus \{\alpha(c_j = [d_1, \ldots, d_k]), \alpha(len(c_j) = k)\}) \\ &\qquad \cup \{\alpha(c_j = [d_1, \ldots, d_k, d]), \alpha(len(c_j) = k+1)\} \\ &= L(\alpha^*(t)). \end{aligned}$$

Therefore $t' = \alpha^*(t)$. If u_i is a static channel read update then a similar argument applies.

Suppose u_i is a dynamic channel write update $write(d, c_{x_i})$, where $d \in D$, and $x_i \in X_i$ is a c-variable. If $(x_i = j) \in L(s)$ for some $m < j \leq m+n$, then the transition associated with u_i is the same as the transition that would be associated with a static channel write update of the form $write(d, c_j)$. The update $u_{\alpha(i)}$ is a dynamic channel write update $write(d^\alpha, c_{x_{\alpha(i)}})$, and $(x_{\alpha(i)} = \alpha(j)) \in L(s)$. Therefore the transition associated with $u_{\alpha(i)}$ is the same as the transition that would be associated with a static channel write update of the form $write(d^\alpha, c_{\alpha(j)})$. It follows, using the argument above for static channel write updates, that $t' = \alpha^*(t)$. If u_i is a dynamic channel read update then a similar argument applies.

Suppose u_i has the form *skip*. Then $u_{\alpha(i)}$ also has the form *skip*. In this case, $s = t$ and $t' = \alpha^*(s)$, therefore $t' = \alpha^*(t)$. Finally, if u_i is a sequence of updates executed simultaneously, then clearly $t' = \alpha^*(t)$.

Proof of Theorem 5.1. Let $\alpha^* \in G$ for some $\alpha \in Aut(\mathcal{C}(\mathcal{P}))$. Let $p = (x_i = d) \in AP$ for some $x_i \in X_i$ $(1 \leq i \leq m)$ and $d \in D$. Then

$$\begin{aligned} p \in L(s_0) &\Leftrightarrow (x_i = init(x_i)) \in L(s_0) \\ &\Leftrightarrow (x_{\alpha(i)} = init(x_i)^\alpha) \in L(s_0) \text{ (by Restriction 2)} \\ &\Leftrightarrow \alpha(x_i = init(x_i)) \in L(s_0) \\ &\Leftrightarrow \alpha(p) \in L(s_0). \end{aligned}$$

Let $p \in AP$ refer to channel c_j for some $m < j \leq m+n$. If $p \in L(s_0)$, then $p = (c_j = [\,])$, $p = (len(c_j) = 0)$ or $p = (next(c_j) = \perp)$. So $\alpha(p) = (c_{\alpha(j)} = [\,])$, $\alpha(p) = (len(c_{\alpha(j)}) = 0)$ or $\alpha(p) = (next(c_{\alpha(j)}) = \perp)$ respectively. In all cases, $p \in L(s_0) \Leftrightarrow \alpha(p) \in L(\alpha^*(s_0))$. It follows that $s_0 = \alpha^*(s_0)$.

Let $(s, t) \in R$. The transition (s, t) is made by a statement $g_i \rightarrow u_i$ of the program $\mathcal{P}$, executed by some process p_i. By Restriction 1, the statement $g_{\alpha(i)} \rightarrow u_{\alpha(i)}$ is also a statement of process $p_{\alpha(i)}$ in the program $\mathcal{P}$. Since

g_i holds in s, it follows by Lemma 5.2 that $g_{\alpha(i)}$ holds in $\alpha^*(s)$. Therefore the statement $g_{\alpha(i)} \to u_{\alpha(i)}$, executed in state $\alpha^*(s)$, results in a transition $(\alpha^*(s), t') \in R$ for some $t' \in S$. By Lemma 5.3, $t' = \alpha^*(t)$, so $(\alpha^*(s), \alpha^*(t)) \in R$.

We have shown that α^* is an automorphism of $\mathcal{M}$, i.e. $\alpha^* \in Aut(\mathcal{M})$. It follows that $G \subseteq Aut(\mathcal{M})$, and since G is a group, $G \leq Aut(\mathcal{M})$ as required.