J Autom Reasoning (2008) 41:251–293
DOI 10.1007/s10817-008-9107-4

Automatic Symmetry Detection for Promela

Alastair F. Donaldson · Alice Miller

Received: 14 November 2008 / Accepted: 14 November 2008 / Published online: 3 December 2008

Abstract We introduce a specification language, Promela-Lite, which captures the essential features of Promela but which, unlike Promela, has a formally defined semantics. We show how we can detect symmetry in specifications defined in Promela-lite by constructing a directed, coloured bipartite digraph called a *static channel diagram*, and applying computational group theoretic techniques. We extend our approach to Promela and introduce a tool, SymmExtractor, for automatically detecting symmetries of Promela specifications. We demonstrate the effectiveness of our approach via experimental results for a suite of Promela specifications. Unlike previous approaches our technique is fully automatic, and not restricted to fully symmetric systems.

Keywords Automatic verification · Promela · Model checking · Symmetry reduction

1 Introduction

Errors in hardware and software development often appear at the design stage, yet are not detected until the final testing stage. The later errors are found, the more expensive they are to correct. Model checking [5, 7, 10, 36] is a popular automatic method that helps to find errors quickly by building small logical models of a system which can be automatically checked. Model checking is most commonly used to verify finite state concurrent systems, like those associated with intricate communications protocols and sequential circuits.

A. F. Donaldson
Codeplay Software Ltd., 45 York Place, Edinburgh, Scotland
e-mail: ally@codeplay.com

A. Miller (✉)
Department of Computing Science, University of Glasgow, Glasgow, Scotland
e-mail: alice@dcs.gla.ac.uk

Model checking involves constructing an underlying model, usually a *Kripke structure*, from a description of a system expressed using a specification language like Promela (for SPIN [28]), the Reactive Modules Language (for SMV [33]) or extended timed automata (for UPPAAL [31]). As the number of components in a system increases, the size of the underlying Kripke structure can grow exponentially. This phenomenon is known as *state space explosion* and can prohibit verification of complex systems. Many systems, however, consist of clusters of sets of identical (up to process id) components, resulting in symmetry of the underlying state space. If this symmetry can be detected it can often be exploited to reduce the cost of model checking sufficiently to allow for full verification.

Symmetry reduced model checking [8, 24, 30] involves checking a smaller *quotient* structure rather than the entire Kripke structure associated with a specification. The quotient structure is constructed during search using knowledge of the symmetry present in the original system. Provided a property to be checked is preserved by the symmetry used, searching the quotient structure is sufficient for verifying the truth or otherwise of the property for the system.

Most approaches to symmetry reduction in model checking require the user to manually specify the symmetry to be exploited, either directly [6, 27, 35] or by annotating the specification using additional keywords [2, 30]. This approach is error-prone and requires the user to have an in-depth knowledge of symmetry reduction theory. In many cases systems are assumed to be *fully symmetric*: all components are identical up to permutation of identifiers. However, this situation arises in practice only for very simple systems.

In order for symmetry reduced model checking to be a viable technique for verification of large, hitherto intractable models, it is vital to provide automatic means to extract symmetry information for *any* system (whether fully symmetric or not). Clearly, since construction of the entire state-space is (assumed to be) infeasible, any such technique must be *static* and rely only on the system specification.

In this paper we concentrate on the problem of automatic symmetry *detection*. We define a specification language Promela-Lite which captures the essential features of Promela but which, unlike Promela, has a formally defined semantics. We describe an automatic technique for detecting symmetry statically from Promela-Lite specifications. Given a Promela-Lite specification $\mathcal{P}$ representing a concurrent system, our approach involves extracting a graphical representation of the communications structure of the system, called a *static channel diagram*, $\mathcal{SCD}$ from $\mathcal{P}$. Having obtained a group of symmetries from $\mathcal{SCD}(\mathcal{P})$ we use computational group theory techniques to infer symmetries of the model associated with $\mathcal{P}$. Although our results apply specifically to Promela-Lite specifications, they extend naturally to specifications presented in a restricted version of Promela. We introduce a tool, SymmExtractor, for the automatic detection of symmetry in Promela specifications, and demonstrate its effectiveness via a series of experiments for a variety of families of Promela specifications.

In Section 2 we discuss related work and in Sections 3 and 4 we provide the necessary background material on model checking, Promela and SPIN, and group theory. We describe the process of symmetry reduction for model checking using quotient structures in Section 5. In Section 6 we introduce our specification language Promela-Lite and in Section 7 we formally define a static channel diagram and show

how it is used to detect symmetry in Promela-Lite specifications. We describe our SymmExtractor tool in Section 8 and provide experimental results and analysis. Our conclusions are presented in Section 9.

2 Related Work

Most symmetry reduction implementations involve replacing sets of states with equivalence class representatives [8, 22, 24]. Implementations include symmetry reduction for the Murϕ verification system [30] and a purpose-built model checker (SMC – Symmetry Model Checker), designed for the verification of highly symmetric systems [40].

A symmetry reduction package for SPIN, SymmSpin [2], exists but, like the Murϕ implementation, relies on user-defined symmetry implemented using a special datatype, a *scalarset* [30]. In a similar technique symmetry is again identified within Promela programs via the use of keywords [13] in a less restricted model of computation.

Neither of these approaches to symmetry reduction for SPIN involve automatic symmetry detection. The user has to specify symmetry via annotation of the specification to indicate the presence of symmetry.

Our approach to symmetry detection is fully automatic. The use of static channel diagrams for symmetry detection for Promela has been previously described in [16] (previous version appeared in proceedings of AVoCS 2004) and [19]. However, no proofs of theoretical results were provided. In this paper we introduce Promela-Lite for the first time and use it to provide a rigorous justification of our previous theoretical results in the context of Promela-Lite. We also include extensive experimental results and analysis, for a range of Promela specifications.

3 Model Checking, Promela and Spin

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

A software tool called a *model checker* then exhaustively searches the finite state model $\mathcal{M}(\mathcal{P})$, checking whether ϕ is true for the model. In *Linear Time Temporal Logic* (*LTL*) model checking this involves checking that ϕ holds for all paths of the model. If ϕ does not hold for some path, an error trace or *counter-example* is reported. Manual examination of this counter-example by the system designer can reveal that ϕ does not adequately specify the behaviour of the system, that ϕ does not accurately describe the given requirement, or that there is an error (bug) in the design. In this case, either $\mathcal{P}$, ϕ, or the system design (and thus also $\mathcal{P}$ and possibly ϕ) must be modified, and re-checked. This process is repeated until the model checker reports that ϕ holds in every initial state of $\mathcal{M}(\mathcal{P})$, in which case we say $\mathcal{M}(\mathcal{P})$ satisfies ϕ, written $\mathcal{M}(\mathcal{P}) \models \phi$. The model checking process is illustrated by Fig. 1.

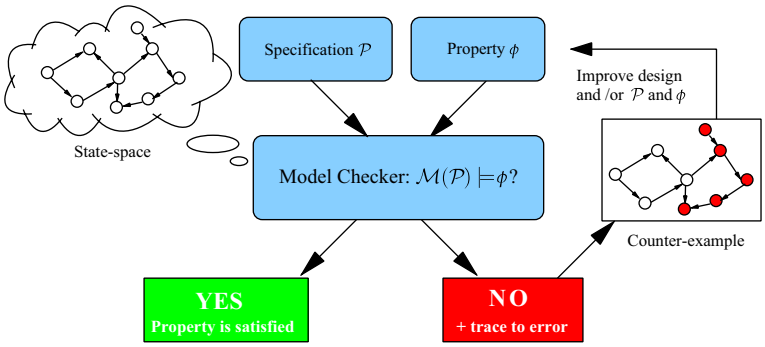

Fig. 1 The model checking process

Assuming that the specification and temporal properties have been constructed with care, successful verification by model checking increases confidence in the system design, which can then be refined towards an implementation.

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

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

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

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

We usually consider Kripke structures which have a single initial state $s_0 \in S$, in which case we write $\mathcal{M} = (S, s_0, R)$. Note that, following the convention of e.g. [9, 24], Definition 1 does not include a labelling function. Such a structure is sometimes referred to simply as a *transition system* [9]. We could equivalently define states as being labelled with atomic propositions of the form $(v_i = d_i)$ (where $d_i \in D_i$) [10].

Figure 2 shows the reachable part of a Kripke structure for a model of two process mutual exclusion. The model consists of two processes, each with three local states N, T and C. Each process has a single state variable, st_i say ($i \in \{1, 2\}$). Here $V = \{st_1, st_2\}$ and $D_1 = D_2 = \{N, T, C\}$. The values N, T and C denote that a process is in the *neutral*, *trying* or *critical* state respectively. For $A \in \{N, T, C\}$ we abbreviate the proposition $st_i = A$ by A_i. Only if process i is in the trying state (i.e. T_i holds) and process $j \neq i$ is *not* in the critical state (i.e. $\neg C_j$ holds) can process i can move into the critical state. Thus in the model it is not possible for both processes to be in the critical state. That is, the mutual exclusion property holds. Note that there is a single initial state in which both processes are in the neutral state.

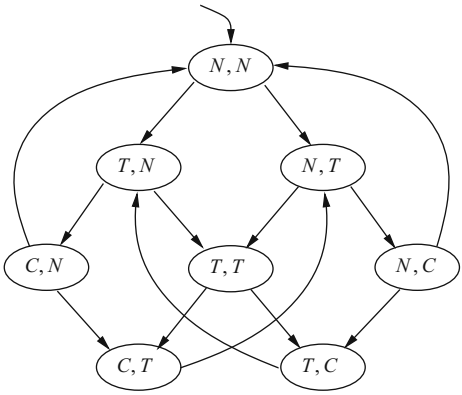

Fig. 2 Kripke structure for two-process mutual exclusion

The model checker SPIN (simple Promela interpreter) [28] allows one to reason about specifications written in the model specification language Promela (process meta language).

Promela is an imperative style specification language designed for the description of network protocols. In general, a Promela specification consists of a series of global variables, channel declarations and process type (proctype) declarations, together with an initialisation process. Logical properties are either specified using assertions embedded in the body of a proctype, or via *LTL* properties. Each proctype in a Promela specification can be viewed as a finite automaton, and the model associated with this specification is the asynchronous product of the automata for all proctype instantiations. This global automaton can be viewed as a Kripke structure, so we refer to the Kripke structure, rather than the automaton, associated with a Promela specification.

We do not give full details of Promela, but refer the reader to [28] and provide an illustrative example in Fig. 3, which shows a specification for a five-process version of the mutual exclusion protocol described in Section 3. In the example we specify that certain sequences of statements should be executed as a single update, using an `atomic` block. In addition we use a repetitive choice statement of the form `do` ⟨*options*⟩ `od`, where ⟨*options*⟩ is a list of Promela fragments, separated by the `::` token. A process executes a `do..od` statement by repeatedly executing one of the options, if any are executable. A `break` or `goto` statement may be used to exit a `do..od` loop. Non-repetitive choice can be specified similarly using an `if..fi` construct.

The specification of Fig. 3 consists of: an enumerated type definition for the symbolic constants `N`, `T` and `C`; a global array `st` which is used to hold the state of each process; a *user* proctype, and an `init` process which instantiates a number of *user* processes. The body of the *user* proctype consists of a single `do..od` statement. Each option in this loop is a `atomic` block consisting of a guard (e.g. `st[_pid]==N`) followed by an update (e.g. `st[_pid]=T`). A *user* process proceeds by repeatedly executing one of the `atomic` blocks, if any are executable. The `init` process consists of a sequence of `run` statements, each instantiating a *user* process. The `run` statements are contained within an `atomic` block, so that they are executed as an indivisible sequence.

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

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

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

Fig. 3 Promela specification of mutual exclusion with 5 processes

The declaration and use of channels is illustrated in Fig. 4. There is a single channel of length 1. The `agent` that reads a message first from the channel declares itself the winner. Note that the model has a deadlock, because the test for channel status and the send/receive is not executed atomically. Both processes can enter the first branch of the `if` statement, but only one of them will be able to send and declare itself the winner. The other process will be blocked. Note that `printf` statements are used to display information during simulation, but have no effect on verification.

4 Group Theory

Symmetries of a Kripke structure form a *group*, thus we require some definitions and results from group theory. For more details, see e.g. [3, 37].

Fig. 4 Promela specification with a buffered channel

```
chan chan1 = [1] of {bit};

proctype agent(){
  bit x=1;
  if
    :: empty(chan1) -> chan1!x; goto winner
    :: full(chan1) -> chan1?x; goto loser
  fi;
  winner: printf("I am the winner");
  goto stop;
  loser: printf("I am the loser");
  stop: skip
}

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

4.1 Groups, Graphs and Automorphisms

Definition 2 A *group* is a non-empty set G together with a binary operation $\circ : G \times G \to G$ which satisfies:

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

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

A *directed graph* (referred to as a *digraph*) is a pair (V, E) where V is a set of vertices and E a set of edges – ordered pairs of vertices written as a pair (u, v) $(u, v \in V)$. A digraph (V, E) is *bipartite* if $V = V_1 \cup V_2$, where $V_1 \subset V$ and $V_2 \subset V$ are disjoint non-empty sets and, for $(u, v) \in E$, $u \in V_1$ and $v \in V_2$, or $u \in V_2$ and $v \in V_1$. A *colouring* of digraph (V, E) is a mapping $\Upsilon : V \to K$, where K is a finite set of *colours*. A coloured digraph is a triple (V, E, Υ) such that (V, E) is a digraph and Υ a colouring of (V, E).

Definition 3 Let $\Gamma = (V, E, \Upsilon)$ be a coloured digraph and α a permutation of V. Then α is an *automorphism* of Γ if the following conditions are satisfied:

- For all $(u, v) \in E, (\alpha(u), \alpha(v)) \in E$
- For all $v \in V, \Upsilon(v) = \Upsilon(\alpha(v))$.

The set of all automorphisms of a coloured digraph Γ forms a group under composition of mappings, denoted $Aut(\Gamma)$.

Let G be a group and let $H \subseteq G$. If H is non-empty and is itself a group under the binary operation then H is a *subgroup* of G, denoted $H \leq G$. If $H \subset G$ then H is a *proper* subgroup of G, denoted $H < G$.

Definition 4 Let $X \subseteq G$. Then $\langle X \rangle$ denotes the smallest subgroup of G which contains X, and is called the subgroup *generated by* X. If $\alpha_1, \alpha_2, \ldots, \alpha_k \in G$ then we use the shorthand $\langle \alpha_1, \alpha_2, \ldots, \alpha_k \rangle$ to denote $\langle \{\alpha_1, \alpha_2, \ldots, \alpha_k\} \rangle$.

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

Definition 5 Let H be a subgroup of G and $\alpha \in G$. Then the set $H\alpha = \{\beta\alpha : \beta \in H\}$ is a (right) *coset* of H in G.

A similar definition can be given for a *left* coset of H in G. We will use *coset* to mean *right coset*. A set of *coset representatives* for H in G is a subset of G containing exactly one element from each coset of H in G.

A mapping between two groups which preserves products of elements is called a *homomorphism*. If a homomorphism θ from G_1 to G_2 is bijective, then θ is an *isomorphism* from G_1 to G_2, and G_1 and G_2 are said to be *isomorphic*, denoted $G_1 \cong G_2$. Isomorphic groups differ only in that their elements may be labelled differently.

4.2 Permutation Groups and Group Actions

Let X be a non-empty set. A *permutation* of X is a bijection $\alpha : X \to X$. The set of all permutations of X forms a group under composition of mappings, denoted $Sym(X)$.

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

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

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

Definition 6 Let $G \leq Sym(X)$ where X is a non-empty set. The group G induces an equivalence relation $\equiv_G$ on X thus: $x \equiv_G y \Leftrightarrow x = \alpha(y)$ for some $\alpha \in G$. The equivalence class under $\equiv_G$ of an element $x \in X$, denoted $[x]_G$, is called the *orbit* of x under G.

An important class of permutation group is the symmetric groups:

Definition 7 For $n > 0$, the group $Sym(\{1, 2, \ldots, n\})$ is called the *symmetric group of degree n*, denoted S_n. From the above, we have $|S_n| = n!$. S_n is often referred to as the *full* symmetry group.

Fundamental to most applications of symmetry reduction in model checking is the idea that a group of permutations of a given set *induces* a group of permutations on another (usually larger) set. For example, a group of process identifier permutations naturally induces a group of permutations of the set of states associated with a specification. We describe this idea formally using *group actions*. The following definition and theorem are adapted from [11] and [37].

Definition 8 Let X be a non-empty set and G a group. A group action of G on X is a mapping $\mu : X \times G \to X$ such that for every $x \in X$ and $\alpha, \beta \in G$,

- $\mu(x, \alpha\beta) = \mu(\mu(x, \alpha), \beta)$
- $\mu(x, id) = x$

If there is a group action of G on X we say that G acts on X. From now on we write $\alpha(x)$ for $\mu(x, \alpha)$, since the action μ is always clear from the context.

Theorem 1 *Let G act on X. Then to each $\alpha \in G$ there corresponds an element $\rho_\alpha \in Sym(X)$ defined by $\rho_\alpha : x \mapsto \alpha(x)$, and the map $\rho : G \to Sym(X)$ defined by $\rho : \alpha \mapsto \rho_\alpha$ is a homomorphism.*

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

Many of the groups that arise naturally from specifications of systems consisting of clusters of identical processes consist of products of smaller groups. Specific types of group product include the *(internal) direct product*, denoted $G = H_1 \times H_2 \times \cdots \times H_k$ for subgroups $H_1, H_2, \ldots, H_k$, and the *wreath product*, a product of two specific groups H and K, denoted $H \wr K$. We do not give details of these products here, but instead refer the reader to [3, 37].

5 Symmetry Reduction Using Quotient Structures

Although this paper is not concerned with symmetry reduction, but with symmetry detection, we include here the basic theory behind the technique for symmetry reduction using quotient structures. A full survey of symmetry reduction methods for temporal logic model checking is presented in [34].

Definition 9 Let $\mathcal{M} = (S, S_0, R)$ be a Kripke structure over D. An *automorphism* of $\mathcal{M}$ is a permutation $\alpha : S \to S$ which preserves the transition relation and set of initial states. That is α satisfies:

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

It can be shown that the automorphisms of a Kripke structure $\mathcal{M}$ form a group under composition of mappings, denoted $Aut(\mathcal{M})$. Note that in condition 1 of Definition 9 we only require implication in one direction ($\Rightarrow$). Implication in the other direction ($\Leftarrow$) follows from the fact that $Aut(\mathcal{M})$ is a group: if $(\alpha(s), \alpha(t)) \in R$ then, since $\alpha^{-1} \in Aut(\mathcal{M})$, we have $(\alpha^{-1}(\alpha(s)), \alpha^{-1}(\alpha(t))) \in R$, i.e. $(s, t) \in R$.

In a model of a concurrent system with many replicated processes, Kripke structure automorphisms typically involve the permutation of process identifiers throughout all states of the model. There is a group G which permutes the set of process identifiers, and an action of G on S (see Definition 8). Let ρ be the corresponding permutation representation (see Theorem 1). The group of automorphisms of $\mathcal{M}$ *induced* by G is $\rho(G)$, the image of G under the permutation representation. Given $\alpha \in G$, rather than referring to the automorphism ρ_α of $\mathcal{M}$ we say simply that α is an automorphism of $\mathcal{M}$.

Given a subgroup G of $Aut(\mathcal{M})$, the orbits of S under G (see Definition 6) can be used to construct a *quotient* Kripke structure $\mathcal{M}_G$ as follows:

Definition 10 The quotient Kripke structure $\mathcal{M}_G$ of $\mathcal{M}$ with respect to G is a tuple $\mathcal{M}_G = (S_G, S^0_G, R_G)$ where:

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

Efficiently restricting state-space search to a unique representative from each orbit is a challenging problem that has been considered in several symmetry reduction implementations [2, 6, 20, 22, 40].

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

Consider the model $\mathcal{M}$ for the mutual exclusion example shown in Fig. 2. Swapping process indices 1 and 2 throughout all states gives an automorphism, α say, of $\mathcal{M}$, and $Aut(\mathcal{M}) = \{\alpha, id\}$, where id is the identity mapping. Choosing a unique representative from each orbit we obtain the quotient Kripke structure $\mathcal{M}_{Aut(\mathcal{M})}$ illustrated by Fig. 5.

The following result [8, 24] states that a model and its quotient model satisfy the same *symmetric CTL** formulas (see e.g. [10] for details of the temporal logic *CTL**). A *CTL** formula ϕ is symmetric, or invariant, with respect to G if for every maximal propositional sub-formula f appearing in ϕ, and for every $\alpha \in G$, $\mathcal{M}, s \models f \Leftrightarrow \mathcal{M}, \alpha(s) \models f$. In this case, G is an *invariance group* for ϕ.

Theorem 2 *If $\mathcal{M}$ and $\mathcal{M}_G$ denote a model and its quotient model with respect to a group G respectively, $\mathcal{M} \models \phi \Leftrightarrow \mathcal{M}_G \models \phi$ for all symmetric CTL* formulas ϕ.*

Consider the two-process mutual exclusion property $\phi_1 = \mathbf{AG}(\neg(C_1 \wedge C_2))$. Clearly ϕ_1 is symmetric with respect to the automorphism group $G = \{\alpha, id\}$, where α is defined as above. Thus the Kripke structure $\mathcal{M}$ (represented by Fig. 2) satisfies ϕ_1 if and only if the quotient structure $\mathcal{M}_G$ (represented by Fig. 5) does. Therefore, to check the mutual exclusion property, it is sufficient to check the quotient model only.

The first step which must be accomplished by any reduction method which exploits symmetry is the identification of a suitable symmetry group G. Clearly we can not use $\mathcal{M}$ itself to find G as we only use symmetry reduction methods in situations where it is not possible to construct $\mathcal{M}$.

In most cases where symmetry reduction is used in model checking, the user is required to manually specify the symmetry present in a model [6, 35]. However this

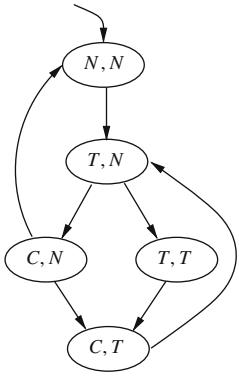

Fig. 5 Quotient Kripke structure for two-process mutual exclusion

is error-prone and is only realistic in cases where there is *full* symmetry, i.e. the symmetry group contains all permutations of component ids.

A popular approach to symmetry detection involves annotation of the system description via a purpose-built data type [30]. The data type is called a *scalarset*, and acts as documentation that certain symmetries are present in a specification expressed in the Murϕ description language [14]. A scalarset is an integer sub-range with restricted operations. These restrictions are sufficient to ensure that consistent permutation of scalarset variables in all states corresponds to an automorphism of the state-space. Scalarsets have been used to implement symmetry reduction techniques for the SPIN model checker via the SymmSpin tool [2].

Unlike SymmSpin, our approach to symmetry detection using static channel diagrams (see Section 7) is fully automatic, and not restricted to full symmetry.

6 Promela-Lite

In order to support our techniques with a formal proof, we present *Promela-Lite*, a specification language which captures the essential features of Promela. The Promela language includes a large set of keywords and language features to specify complex communication protocols. Consequently, proving properties about Promela specifications is laborious, requiring many case-by-case arguments. Rigorous proofs are also hindered by the lack of a formal definition of the semantics of Promela as implemented by SPIN.

Promela-Lite is a smaller specification language that includes core Promela features such as parameterised processes, first-class channels and global variables, but omits many language features such as enumerated types, record types, arrays and rendez-vous channels. We present a full grammar and type system for this smaller language, as well as a precise Kripke structure semantics for Promela-Lite specifications. In Section 7 we prove the correctness of our symmetry detection techniques for Promela-Lite. As Promela-Lite and Promela are so closely related, our results lend considerable weight to the proposition that our results can be applied to (a restricted form of) Promela. Omitting the more ornate features of Promela from Promela-Lite makes our proof easier to transfer to other specification formalisms.

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

The name *Promela-Lite* was inspired by *Featherweight Java*, a calculus which captures the core object oriented features of Java (classes, methods and inheritance), but omits features of the full language [29].

To illustrate features of Promela-Lite we discuss an example specification of a message passing system, given in Fig. 6. The specification consists of three *server* processes, six *client* processes and three *load-balancer* processes. A particular *client* has been blocked by the system, indicated by the global *pid* variable *blocked_client*. A *load-balancer* process continuously receives requests sent by *client* processes. A

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

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

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

pid blocked_client = 9;

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

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

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

init {
  atomic {

    run server(se1,null,1); run server(se2,null,2); run server(se3,null,3);

    run loadbalancer(lb1,null,0,1); run loadbalancer(lb2,null,0,1);
    run loadbalancer(lb3,null,0,1);

    run client(cl1,lb1,-1,1); run client(cl2,lb1,-1,1); run client(cl3,lb2,-1,1);
    run client(cl4,lb2,-1,1); run client(cl5,lb3,-1,1); run client(cl6,lb3,-1,1);

  }
}
```

Fig. 6 Promela-Lite specification of a load-balancing system

request consists of two parts: the identity of a *client* (derived from its `_pid` variable), and the input channel of the *client*. If the message originates from the blocked *client* then the *load-balancer* returns the value 0, indicating that the request has been

denied. Otherwise the *load-balancer* forwards the name of the input channel of the given *client* to the *server* with the shortest queue of incoming messages (choosing non-deterministically between *server*s which share the shortest queue length). On receiving a *client* channel name, a *server* uses it to send the value 1 to the *client*.

6.1 Syntax

We use the standard Backus-Naur form (BNF, see e.g. [1]) to specify the syntax of Promela-Lite.

Let ⟨*prod*⟩ be a BNF production rule. We use the following shorthand notation to refer to occurrences of ⟨*prod*⟩ on the right hand side of other production rules:

- ⟨*prod*⟩$^?$ denotes an optional occurrence of ⟨*prod*⟩
- ⟨*prod*⟩* denotes a sequence of zero or more occurrences of ⟨*prod*⟩
- ⟨*prod*⟩$^+$ denotes a sequence of one or more occurrences of ⟨*prod*⟩
- ⟨*prod-list,* '∘'⟩ denotes a ∘-separated list of one or more occurrences of ⟨*prod*⟩.

6.1.1 Syntax of Types

The syntax of Promela-Lite data types is summarised in Fig. 7 (see Fig. 8 for details of the ⟨*name*⟩ production rule).

The language includes two primitive data types, *int* and *pid*, representing integer values and process id values respectively. Basic channel types have the form $chan\{\overline{T}\}$, where $\overline{T}$ denotes a comma-separated list of types (which could include channel types).

It can be useful for a channel of type T to accept a channel of type T as one of its arguments. In this case, T is a *recursive type*. Accordingly, Promela-Lite includes syntax for *recursive* channel types (the ⟨*recursive*⟩ rule of Fig. 7). For example, consider a type T of the form $rec\ X\ .\ chan\{X, int\}$. Then T denotes a channel which accepts messages consisting of two fields: a channel of type T, and an integer. This recursive type can be *unfolded* by removing the initial '$rec\ X\ .$' and substituting 'X' for the original expression, resulting in the type expression $chan\{rec\ X\ .\ chan\{X, int\}, int\}$. Note that although Promela does not allow explicit declaration of recursive types, recursive channel types can be introduced implicitly via channel usage [18]. We use $chan\{\overline{T}\}$ to refer to an arbitrary channel type, since a channel type of the form $rec\ X\ .\ chan\{\ldots\}$ can always be unfolded into this form.

Fig. 7 Promela-Lite type syntax

⟨*type*⟩ ::= `int`
  | `pid`
  | ⟨*chantype*⟩
  | ⟨*typevar*⟩

⟨*chantype*⟩ ::= ⟨*recursive*⟩$^?$ `chan` { ⟨*type-list,* '`,`'⟩ }

⟨*recursive*⟩ ::= `rec` ⟨*typevar*⟩ `.`

⟨*typevar*⟩ ::= ⟨*name*⟩

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

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

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

⟨*proctype*⟩ ::= ⟨*name*⟩ `(` ⟨*param-list,* `;`⟩? `)` { `do` ⟨*statement-list,* `::`⟩ `od` }

⟨*param*⟩ ::= ⟨*type*⟩ ⟨*name*⟩

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

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

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

⟨*init*⟩ ::= `init` { `atomic` ⟨*run-list,* `;`⟩ }

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

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

⟨*expr*⟩ ::= ⟨*name*⟩
| ⟨*number*⟩
| `_pid`
| `null`
| `len` `(` ⟨*name*⟩ `)`
| `(` ⟨*expr*⟩ `)`
| ⟨*expr*⟩ ∘ ⟨*expr*⟩ (where ∘ ∈ {+, −, *})

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

⟨*number*⟩ ::= a positive integer

Fig. 8 Syntax of Promela-Lite

6.1.2 Syntax of the Language

A Promela-Lite specification consists of a series of channel and global variable declarations, one or more proctypes, and an `init` process. In the example specification of Fig. 6 there is a sequence of channel declarations, a global variable declaration (`blocked_client`), proctypes `loadbalancer`, `server` and `client` and an `init` process.

The syntax of Promela-Lite is given in Fig. 8, in which we have simplified the presentation of the rules ⟨*name*⟩ and ⟨*number*⟩. In the ⟨*guard*⟩ production rule, ⋈

denotes an operator taken from the set {==, !=, <, <=, >, >=}. For simplicity, to avoid the need for detailed semantics for division-by-zero errors, we have not included the division operator in Promela-Lite.

A channel declaration `chan` c `= [`a`] of` $\{\overline{T}\}$ defines a buffered channel c with type $chan\{\overline{T}\}$ and length $a > 0$. This is similar to a *globally instantiated* channel in Promela. We define the signature of c by $signature(c) = (a, \{\overline{T}\})$ and call channels declared in this way *static* channels. In Fig. 6 channel `se1` has type $chan\{chan\{int\}\}$ and length 3, and $signature(\texttt{se1}) = (3, \{chan\{int\}\})$

A static channel name cannot be re-assigned (via assignment or a channel receive operation). A global variable declaration $T\ x = a$ associates a name x with a type $T \in \{int, pid\}$ and an initial value a. Note that this restriction on T is imposed by the type system, not the grammar. Our channel requirements are formally specified in Appendix A via typing rules.

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

Each statement has the form `atomic { ⟨`*guard*`⟩ -> ⟨`*update-list,* `';'⟩ }` where ⟨*guard*⟩ is a boolean expression over variables and ⟨*update-list,* `';'`⟩ a sequence of updates to variables and channels. The `atomic` block surrounding the guard and updates ensures that executing the statement results in a single transition of the system. In Fig. 6 every guard of the `client` proctype contains a proposition relating to the program counter *pc* of the associated process, and in some cases also a check on the status of a channel. Promela keywords `len`, `nfull` and `nempty` are retained in Promela-Lite.

The `init` process consists of a set of `run` statements contained within an atomic block. If process i is an instantiation of proctype p, we write $proctype(i) = p$.

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

6.2 Type System and Kripke Structure Semantics

We say that a Promela-Lite specification $\mathcal{P}$ is well-typed if the statements and declarations in $\mathcal{P}$ are well-formed according to the typing rules of Appendix A. We note some important properties of well-typed specifications that are used in our proofs in Section 7.3.

1. The fullness/emptiness of a channel is checked before it is used for communication.
2. A Promela-Lite statement involves at most one send or receive update (at the beginning of the sequence of updates for the sequence).
3. If a guard has the form $e_1 == e_2$ or $e_1 != e_2$, where e_1 and e_2 are expressions, then e_1 and e_2 have the same type.

4. Similarly if a guard has the form $e_1 \bowtie e_2$ where $\bowtie \in \{<, <=, >, >=\}$ then e_1 and e_2 have type *int*.

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

Unlike Promela, there is a fully documented Kripke structure semantics for Promela-Lite (which we present in full in Appendix B). For each update u described by the $\langle update \rangle$ rule in Fig. 8, the effect of u on a state s can be defined via a further set of rules, presented in Fig. 1. Note that a state $s \in S$ can be expressed as an ordered tuple consisting of a value for each variable and channel (see Definition 1). However, it is more convenient here to reason about s as a set of propositions, one for each variable and channel of $\mathcal{P}$. Note also that for a proctype p, variable name x and process identifier i with $proctype(i) = p$, we define $var(x)$ to be x if x is a global variable, and $p[i].x$ otherwise. In addition, when representing the contents of a channel, $\vec{a}$ denotes a tuple of values (one per message field). In each case we define the update u, the conditions under which u applies, and the result of applying u to s (denoted $exec_{p,i}(s, u)$). For an expression e appearing in proctype p, $eval_{p,i}(s, e)$ denotes the (boolean or integer) result of evaluating e for process i at state s, where $proctype(i) = p$.

Given a sequence of updates $u_1, u_2, \ldots, u_k$ and a state s the rules of Fig. 1 can be applied repeatedly to define the state reached by executing the u_i in sequence, starting in state s (assuming that the conditions of the updates are satisfied in the relevant intermediate states - otherwise the Kripke structure is deadlocked). The resulting state is denoted $exec_{p,i}(s, u_1; u_2; \ldots; u_k)$, where

$$exec_{p,i}(s, u_1; u_2; \ldots; u_k) = exec_{p,i}(\ldots exec_{p,i}(exec_{p,i}(s, u_1), u_2) \ldots, u_k).$$

A statement is said to be well-defined if the application conditions of the statement are sufficient to ensure that the sequence of updates result in a well-defined state. A transition between well defined states is said to be a well-defined transition. A Kripke structure $\mathcal{M}$ associated with a Promela-Lite specification $\mathcal{P}$ is said to be well-defined if all of its states and transitions are well-defined. Note that $\mathcal{M}$ has a single initial state s_0.

In Appendix B we prove the following:

Theorem 3 *If $\mathcal{P}$ is a well-typed Promela-Lite specification then its associated model $\mathcal{M}$ is well-defined.*

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

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

The Promela-Lite semantics are based on the informal semantics for Promela described in [28] and the SPIN source code. They have been designed so that if

$\mathcal{P}$ is a well-typed Promela-Lite specification and $\mathcal{P}'$ the corresponding Promela specification, then $\mathcal{P}$ and $\mathcal{P}'$ have the same associated model.

7 Finding Symmetry by Static Channel Diagram Analysis

In this section we introduce the *static channel diagram* of a Promela-Lite specification. This is a graphical structure that can be extracted by syntactic inspection of a specification. We formally establish a correspondence between automorphisms of the static channel diagram and automorphisms of the Kripke structure associated with a Promela-Lite specification.

We present a symmetry detection technique based on this correspondence, which can be summarised as follows: generators for a group of *candidate* symmetries for a Promela-Lite specification are found by analysing the static channel diagram of the specification. These generators are checked individually against the specification to see if they induce valid automorphisms of the associated model. Starting with the set of candidate generators which are valid, the largest possible subgroup of candidate symmetries which are all valid is computed. These symmetries can then be used for reduced model checking.

Our approach can detect *arbitrary* component symmetries arising from the communication structure of a specification, and can be fully automated requiring no additional information from the user. The only requirement is that the specification satisfies the formally defined restrictions imposed by the Promela-Lite type system, which can be automatically checked. These restrictions are less strict than those imposed by, for example, the scalarset data type [30].

7.1 Static Channel Diagrams

Let $\mathcal{P}$ be a Promela-Lite specification with n processes. Let $V_P = \{1, 2, \ldots, n\}$ be the set of process identifiers, and V_C the set of static channel names for $\mathcal{P}$. Recall that a Promela-Lite statement involves at most one send or receive update, which must appear at the beginning of the sequence of updates for the statement.

Definition 11 The *static channel diagram* associated with $\mathcal{P}$ is a coloured, bipartite digraph $\mathcal{SCD}(\mathcal{P}) = (V, E, \Upsilon)$ where:

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

Note that a static channel diagram is similar to a *channel diagram* [38]. The difference is that the channel diagram records all *possible* channel-based communication, whereas the static channel diagram records *potential* communication on *certain channels*. The static channel diagram of a specification can be seen as a static approximation of the communication structure for the specification. It does not capture communication arising from dynamic passing of channel references, and edges of the diagram may result from send/receive updates which in practice cannot be executed in any reachable state of $\mathcal{M}$. Nevertheless, our symmetry detection techniques can be applied successfully to specifications which use dynamic channel passing. Note that, unlike channel diagrams, a static channel diagram $\mathcal{SCD}(\mathcal{P})$ can be computed directly from $\mathcal{P}$ without constructing $\mathcal{M}$.

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

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

Figure 9 shows the static channel diagram for the Promela-Lite specification of the load-balancer system, given in Fig. 6. Ovals and rectangles are used to represent processes and channels respectively. The type of a process is its proctype name, and channel signatures are indicated using the key in the figure. Note that there are no outgoing edges from the *server* processes to the *client* input channels as communication from a *server* process to a *client* channel is achieved *dynamically*.

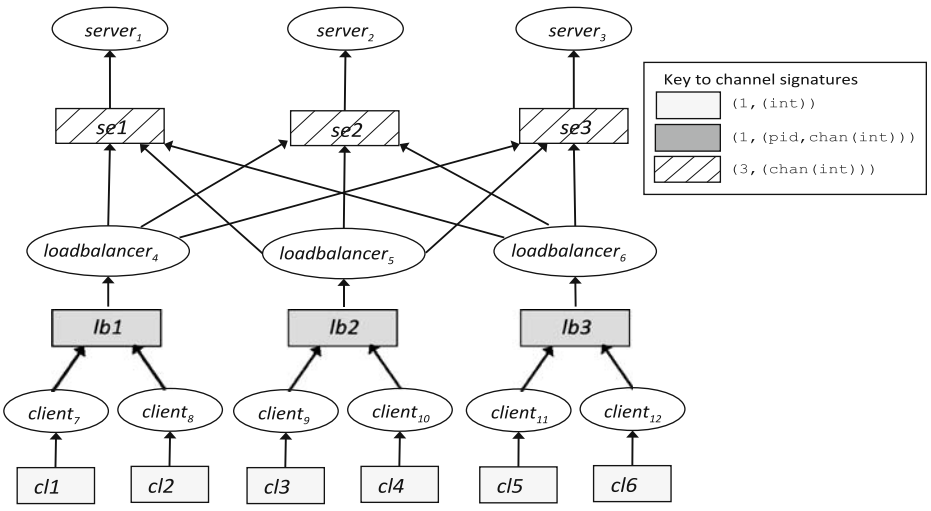

Fig. 9 Static channel diagram associated with the load-balancer specification (Fig. 6)

7.2 Static Channel Diagram Automorphisms

An automorphism of the static channel diagram $\mathcal{SCD}(\mathcal{P}) = (V, E, \Upsilon)$ is an automorphism of the directed, coloured graph (V, E, Υ) (see Definition 3). The group of all automorphisms of $\mathcal{SCD}(\mathcal{P})$ is denoted $Aut(\mathcal{SCD}(\mathcal{P}))$. For $\alpha \in Aut(\mathcal{SCD}(\mathcal{P}))$ we define $\alpha(0) = 0$ and $\alpha(\texttt{null}) = \texttt{null}$.

Since $\mathcal{SCD}(\mathcal{P})$ is relatively small, $Aut(\mathcal{SCD}(\mathcal{P}))$ can be efficiently computed directly using a standard algorithm such as *nauty* [32]. Let $\mathcal{P}$ denote the load-balancer specification of Fig. 6 with static channel diagram $\mathcal{SCD}(\mathcal{P})$ shown in Fig. 9. Recalling that i is the pid of the ith proctype instantiated in the `init` process, we find:

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

It is straightforward to check that each generator of this group is indeed an automorphism of $\mathcal{SCD}(\mathcal{P})$. We have used the computational group theory package GAP (groups, algorithms and programming) [26] to show that $Aut(\mathcal{SCD}(\mathcal{P})) \cong S_3 \times (S_2 \wr S_3)$ (see Section 4.2). Intuitively, the wreath product $S_2 \wr S_3$ arises due to symmetry within each of the three blocks of *clients* (the group S_2), combined with symmetry between the three blocks (the group S_3). The group S_3 on the left hand side of the direct product corresponds to permutation of the *server* processes (and their associated channels).

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

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

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

We consider the action of $Aut(\mathcal{SCD}(\mathcal{P}))$ on the states of $\mathcal{M}$. Let $\alpha \in Aut(\mathcal{SCD}(\mathcal{P}))$. We first define the effect of α on propositions which refer to variables and static channels of $\mathcal{P}$. Note that for a well-formed type T, $lit(T)$ denotes the set of all possible literal values which can have type T in the specification $\mathcal{P}$.

- Let $(x = a)$ be a proposition referring to a global variable x with $x : T$ and $a \in lit(T)$, where $T = pid$ or $T = int$. If $T = pid$ then $\alpha((x = a)) = (x = \alpha(a))$, otherwise $\alpha((x = a)) = (x = a)$.
- For some proctype p and process i such that $proctype(i) = p$, let $(p[i].x = a)$ be a proposition referring to a local variable x of process i, with $x : T$ and $a \in lit(T)$. If $T = pid$ or $T = chan\{\overline{T}\}$ then we have $\alpha((p[i].x = a)) = (p[\alpha(i)].x = \alpha(a))$. Otherwise $\alpha((p[i].x = a)) = (p[\alpha(i)].x = a)$. Since α preserves the colouring of processes according to their proctype, process $\alpha(i)$ is also an instantiation of

proctype p and therefore the local variable $p[\alpha(i)].x$ exists. Thus the action of α is well-defined.

– Let $(c = [\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_m])$ be a proposition referring to a static channel c with signature $(l, \{\overline{T}\})$ where $0 \leq m \leq l$. Then

$$\alpha((c = [\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_m])) = (\alpha(c) = [\vec{a}_1^{\,\alpha}, \vec{a}_2^{\,\alpha}, \ldots, \vec{a}_m^{\,\alpha}])$$

If $\vec{a}_i = (a_1, a_2, \ldots, a_k)$ then $\vec{a}_i^{\,\alpha} = (b_1, b_2, \ldots, b_k)$ where $b_i = \alpha(a_i)$ if $T_i = pid$ or $T_i = chan\{\overline{U}\}$ and $b_i = a_i$ otherwise. The action of α is well-defined as α preserves the signature of static channels.

Let $\mathcal{M} = (S, s_0, R)$ be the model associated with $\mathcal{P}$. Treating a state s as set of propositions as in Section 6.2, the state $\alpha(s)$ is defined as follows: $\alpha(s) = \{\alpha(z) : z \in s\}$.

For all $s \in S$ and $\alpha, \beta \in Aut(\mathcal{SCD}(\mathcal{P}))$, it is clear that $(\alpha\beta)(s) = \alpha(\beta(s))$ and $id(s) = s$, therefore the definition of $\alpha(s)$ is an *action* of $Aut(\mathcal{SCD}(\mathcal{P}))$ on S (See Definition 8).

7.3 Correspondence

Let ρ be the permutation representation of $Aut(\mathcal{SCD}(\mathcal{P}))$ corresponding to its action on S. By Theorem 1, $\rho(Aut(\mathcal{SCD}(\mathcal{P}))) \leq Sym(S)$. Now $Aut(\mathcal{M}) \leq Sym(S)$, but we cannot, in general, say anything about the relationship between $\rho(Aut(\mathcal{SCD}(\mathcal{P})))$ and $Aut(\mathcal{M})$.

In this section we define what it means for an element of $Aut(\mathcal{SCD}(\mathcal{P}))$ to be *valid* for $\mathcal{P}$, and show that the set of all valid elements of $Aut(\mathcal{SCD}(\mathcal{P}))$ form a group $G \leq Aut(\mathcal{SCD}(\mathcal{P}))$. We prove that if $\alpha \in Aut(\mathcal{SCD}(\mathcal{P}))$ is valid for $\mathcal{P}$ then $\rho(\alpha) \in Aut(\mathcal{M})$. Thus $\rho(G) \leq Aut(\mathcal{M})$. The relationship between the various groups is illustrated in Fig. 10.

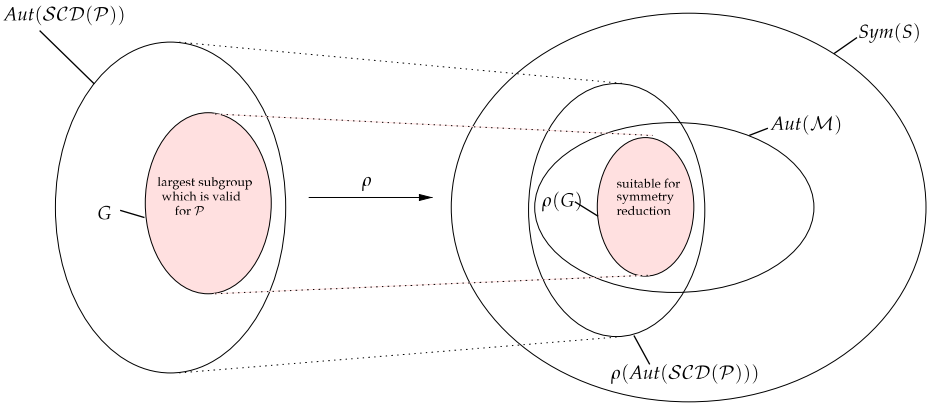

Fig. 10 Relationship between valid automorphisms of $\mathcal{SCD}(\mathcal{P})$ and automorphisms of $\mathcal{M}$

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

We say that two Promela-Lite specifications $\mathcal{P}_1$ and $\mathcal{P}_2$ are equivalent, and write $\mathcal{P}_1 \equiv \mathcal{P}_2$, if they are identical up to re-arrangement of statements in the `do...od` construct, and operands to the commutative operators `+`, `*`, `&&` and `||`. An element $\alpha \in Aut(\mathcal{SCD}(\mathcal{P}))$ is *valid* for $\mathcal{P}$ if $\alpha(\mathcal{P}) \equiv \mathcal{P}$.

Theorem 4 *Let* $G = \{\alpha \in Aut(\mathcal{SCD}(\mathcal{P})) : \alpha \text{ is valid for } \mathcal{P}\}$. *Then* $G \leq Aut(\mathcal{SCD}(\mathcal{P}))$.

Proof Since $id(\mathcal{P}) = \mathcal{P}$, clearly $id(\mathcal{P}) \equiv \mathcal{P}$, so $id \in G$. Associativity is inherited from $Aut(\mathcal{SCD}(\mathcal{P}))$. Let $\alpha, \beta \in G$. Then $\alpha\beta(\mathcal{P}) \equiv \mathcal{P}$ (by applying the re-arrangements of α to those of β), i.e. $\alpha\beta \in G$. As $Aut(\mathcal{SCD}(\mathcal{P}))$ is finite, $\alpha^{-1} = \alpha^k$ for some $k > 0$, thus $\alpha^{-1} \in G$ by the above argument. □

If H is a subgroup of $Aut(\mathcal{SCD}(\mathcal{P}))$ such that every element of H is valid for $\mathcal{P}$ we say that H is valid for $\mathcal{P}$. The group G of Theorem 4 is the largest valid subgroup of $Aut(\mathcal{SCD}(\mathcal{P}))$.

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

7.3.2 Correspondence Theorem

In this section we prove the following theorem:

Theorem 5 *Let* $\mathcal{P}$ *be a Promela-Lite specification, and* $\alpha \in Aut(\mathcal{SCD}(\mathcal{P}))$. *If* α *is valid for* $\mathcal{P}$ *then* $\rho(\alpha) \in Aut(\mathcal{M})$.

The proof of Theorem 5 uses four technical lemmas, which we prove below. First we clarify some notation.

As in Section 6.2, if p is a proctype and i an identifier for which $proctype(i) = p$, then: for an expression e appearing in proctype p, $eval_{p,i}(s, e)$ denotes the (boolean or integer) result of evaluating e for process i at state s; for a list of updates $u_1, u_2, \ldots, u_k$, $exec_{p,i}(s, u_1; u_2; \ldots; u_k)$ denotes the state resulting from the ordered execution of updates $u_1, u_2, \ldots, u_k$ by process i in state s. Similarly, for guard g appearing in proctype p, we use $s \models_{p,i} g$ to assert that g holds for process i at state s.

Treating a state s as set of propositions as in Section 6.2, if ϕ is a proposition, and s a state, we say $\phi \in s$ to denote that ϕ holds at s.

Lemma 1 *Let $\alpha \in Aut(\mathcal{SCD}(\mathcal{P}))$, and let e be an expression in $\mathcal{P}$. If $e : int$ then*

$$eval_{p,i}(s, e) = eval_{p,\alpha(i)}(\alpha(s), \alpha(e))$$

and if $e : pid$ or $e : chan\{\overline{T}\}$ then

$$eval_{p,\alpha(i)}(\alpha(s), \alpha(e)) = \alpha(eval_{p,i}(s, e)).$$

Proof If e has type *int* then the Promela-Lite syntax (Fig. 8) restricts e to a simple expression of the form:

1. a, where $a \in \mathbb{Z}$
2. x, where x is a local or global variable of type *int*
3. `len(null)`
4. `len(`c`)`, where c is a static channel name
5. `len(`x`)`, where x is a local variable of type *chan*

or an arithmetic combination of the above. Since α only acts on static channel names and values of type *pid*, if e is a simple expression of one of the first three forms above, clearly $\alpha(e) = e$ and it follows that $eval_{p,i}(s, e) = eval_{p,\alpha(i)}(\alpha(s), \alpha(e)) = eval_{p,\alpha(i)}(\alpha(s), e)$.

If e has the form `len(`c`)` where c is a static channel, and $(c = [\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_m]) \in s$ then $\alpha(e)$ has the form `len(`$\alpha(c)$`)`, $(\alpha(c) = [\vec{a}_1^{\,\alpha}, \vec{a}_2^{\,\alpha}, \ldots, \vec{a}_m^{\,\alpha}]) \in \alpha(s)$, and $eval_{p,i}(s, e) = eval_{p,\alpha(i)}(\alpha(s), \alpha(e)) = m$. If e has the form `len(`x`)` where x is a local variable of $\mathcal{P}$ and $(x = c) \in s$, with c a static channel name or `null`, then $eval_{p,i}(s, e) = eval_{p,i}(s,$ `len(c)`$)$. By the above argument, $eval_{p,i}(s,$ `len(`c`)`$)$ and $eval_{p,\alpha(i)}(s,$ `len(`$\alpha(c)$`)`$)$ are both equal to $eval_{p,\alpha(i)}(\alpha(s), \alpha(e))$.

If e is an arithmetic combination of simple expressions of type *int*, the result holds by induction.

If e has type *pid* then e has one of the forms:

1. a where $a \in lit(pid)$ and a occurs in a *pid* context,
2. `_pid`
3. x where x is a global/local variable with type *pid*.

Suppose e has the form a where $a \in lit(pid)$ and a occurs in a *pid* context. Then $\alpha(e) = \alpha(a)$. We have

$$\begin{aligned} eval_{p,\alpha(i)}(\alpha(s), \alpha(e)) &= eval_{p,\alpha(i)}(\alpha(s), \alpha(a)) \\ &= \alpha(a) = \alpha(eval_{p,i}(s, a)) = \alpha(eval_{p,i}(s, e)). \end{aligned}$$

If e has the form `_pid` then $eval_{p,\alpha(i)}(\alpha(s), \alpha(e)) = eval_{p,\alpha(i)}(\alpha(s),$ `_pid`$) = \alpha(i)$. Since it is also the case that $\alpha(i) = \alpha(eval_{p,i}(s,$ `_pid`$)) = \alpha(eval_{p,i}(s, e))$, the result follows.

Now suppose e has the form x where x is a global variable with $x : pid$. If $(x = a) \in s$, so that $(x = \alpha(s)) \in \alpha(s)$. Then $eval_{p,\alpha(i)}(\alpha(s), \alpha(e)) = eval_{p,\alpha(i)}(\alpha(s), x) = \alpha(a)$. Since $\alpha(a) = \alpha(eval_{p,i}(s, x))$ which is equal to $\alpha(eval_{p,i}(s, e))$, the result follows. The case where x is a local variable with $x : pid$ is similar.

If e has type $chan\{\overline{T}\}$ then it has one of the following forms:

1. `null`
2. c where c is a static channel name
3. x where x is a local variable with type $chan\{\overline{T}\}$.

If e has the form `null` then

$$\begin{aligned} eval_{p,\alpha(i)}(\alpha(s), \alpha(e)) &= eval_{p,\alpha(i)}(\alpha(s), \texttt{null}) \\ &= \texttt{null} = \alpha(\texttt{null}) = \alpha(eval_{p,i}(s, e)). \end{aligned}$$

The arguments for the cases where e is a static channel name, or e is a local variable with type $chan\{\overline{T}\}$, are analogous to those where e is a *pid* literal, or e is a local/global variable with type *pid*. □

Lemma 2 *If $\alpha \in Aut(\mathcal{SCD}(\mathcal{P}))$ and g is a guard in $\mathcal{P}$ then $s \models_{p,i} g \Leftrightarrow \alpha(s) \models_{p,\alpha(i)} \alpha(g)$.*

Proof A guard consists of a boolean combination of propositional formulas. Define the length of a guard to be one plus the number of (propositional) occurrences of `!`, `&&` and `||` or of `()` appearing in g.

If g has length 1 then g has one of the following forms:

1. $e_1 \bowtie e_2$ for expressions e_1 and e_2, where $\bowtie \in \{\texttt{==}, \texttt{!=}, \texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}$
2. `nfull`(c) or `nempty`(c) where c is a global channel name
3. `nfull`(x) or `nempty`(x) where x is a local variable of p with $x : chan\{\overline{T}\}$.

If g has the form e_1`==`e_2 then, by the typing rules we must have $e_1 : T$ and $e_2 : T$ for some type T. If $T = int$ then by Lemma 1 $eval_{p,i}(s, e_j) = eval_{p,\alpha(i)}(\alpha(s), \alpha(e_j))$ for $j \in \{1, 2\}$. We have

$$\begin{aligned} s \models_{p,i} e_1\texttt{==}e_2 &\Leftrightarrow eval_{p,i}(s, e_1) = eval_{p,i}(s, e_2) \\ &\Leftrightarrow eval_{p,\alpha(i)}(\alpha(s), \alpha(e_1)) = eval_{p,\alpha(i)}(\alpha(s), \alpha(e_2)) \\ &\Leftrightarrow \alpha(s) \models_{p,\alpha(i)} \alpha(e_1)\texttt{==}\alpha(e_2). \end{aligned}$$

If $T = pid$ then by Lemma 1 $eval_{p,\alpha(i)}(\alpha(s), \alpha(e_j)) = \alpha(eval_{p,i}(s, e_j))$ for $j \in \{1, 2\}$. We have

$$\begin{aligned} s \models_{p,i} e_1\texttt{==}e_2 &\Leftrightarrow eval_{p,i}(s, e_1) = eval_{p,i}(s, e_2) \\ &\Leftrightarrow \alpha(eval_{p,i}(s, e_1)) = \alpha(eval_{p,i}(s, e_2)) \\ &\Leftrightarrow eval_{p,\alpha(i)}(\alpha(s), \alpha(e_1)) = eval_{p,\alpha(i)}(\alpha(s), \alpha(e_2)) \\ &\Leftrightarrow \alpha(s) \models_{s,\alpha(i)} \alpha(e_1)\texttt{==}\alpha(e_2). \end{aligned}$$

If $T = chan\{\overline{T}\}$ the result follows similarly using Lemma 1. The case where g has the form e_1 `!=` e_2 is similar.

If g has the form $e_1 < e_2$ then $e_1 : int$ and $e_2 : int$. We have

$$s \models_{p,i} e_1 < e_2 \Leftrightarrow eval_{p,i}(s, e_1) < eval_{p,i}(s, e_1)$$

and

$$\alpha(s) \models_{p,\alpha(i)} \alpha(e_1) < \alpha(e_2) \Leftrightarrow eval_{p,\alpha(i)}(\alpha(s), \alpha(e_1)) < eval_{p,\alpha(i)}(\alpha(s), \alpha(e_2)).$$

By Lemma 1, $eval_{p,i}(s, e_1) = eval_{p,\alpha(i)}(\alpha(s), \alpha(e_1))$ and $eval_{p,i}(s, e_2) = eval_{p,\alpha(i)}(\alpha(s), \alpha(e_2))$. Therefore

$$eval_{p,i}(s, e_1) < eval_{p,i}(s, e_1) \Leftrightarrow eval_{p,\alpha(i)}(\alpha(s), \alpha(e_1)) < eval_{p,\alpha(i)}(\alpha(s), \alpha(e_2))$$

i.e.

$$s \models_{p,i} e_1 \texttt{ < } e_2 \Leftrightarrow \alpha(s) \models_{p,\alpha(i)} \alpha(e_1) \texttt{ < } \alpha(e_2).$$

The cases e_1 `<=` e_2, e_1 `>` e_2 and e_1 `>=` e_2 are similar.

Suppose g has the form $\texttt{nfull}(c)$ where c is a static channel name, and that $(c = [\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_k]) \in s$ for some $0 \leq k \leq cap(c)$ (where $cap(c)$ denotes the capacity of channel c). Then $(\alpha(c) = [\vec{a}_1{}^{\alpha}, \vec{a}_2{}^{\alpha}, \ldots, \vec{a}_k{}^{\alpha}]) \in \alpha(s)$ and

$$\begin{aligned} s \models_{p,i} \texttt{nfull}(c) &\Leftrightarrow cap(c) > k \\ &\Leftrightarrow \alpha(s) \models_{p,\alpha(i)} \texttt{nfull}(\alpha(c)). \end{aligned}$$

The case g has the form $\texttt{nempty}(c)$ is similar.

Let g be $\texttt{nfull}(x)$ where x is a local variable of p and $x : chan\{\overline{T}\}$. If $(p[i].x = \texttt{null}) \in s$, then it follows that $(p[\alpha(i)].x = \texttt{null}) \in \alpha(s)$, and we have $s \not\models_{p,i} \texttt{nfull}(x)$ and $\alpha(s) \not\models_{p,\alpha(i)} \texttt{nfull}(x)$. Suppose instead that $(p[i].x = c) \in s$ where c is a static channel name. Then $(p[\alpha(i)].x = \alpha(c)) \in \alpha(s)$. We have

$$\begin{aligned} & s \models_{p,i} \texttt{nfull}(x) \\ & \quad \Leftrightarrow \; s \models_{p,i} \texttt{nfull}(c) \\ & \quad \Leftrightarrow \; \alpha(s) \models_{p,\alpha(i)} \texttt{nfull}(\alpha(c)) \text{ (by the above argument for static channels)} \\ & \quad \Leftrightarrow \; \alpha(s) \models_{p,\alpha(i)} \texttt{nfull}(x). \end{aligned}$$

The case where g is $\texttt{nempty}(x)$ where x is a local variable and $x : chan\{\overline{T}\}$ is similar.

Now suppose that g has length $m > 1$ and that the result holds for all guards of length less than m. Let g_1, g_2 be guards with length less than m.

If g has the form $\texttt{!}\,g_1$ then

$$\begin{aligned} s \models_{p,i} g &\Leftrightarrow s \not\models_{p,i} g_1 \\ &\Leftrightarrow \alpha(s) \not\models_{p,\alpha(i)} \alpha(g_1) \text{ (by inductive hypothesis)} \\ &\Leftrightarrow \alpha(s) \models_{p,\alpha(i)} \texttt{!}\alpha(g_1) \\ &\Leftrightarrow \alpha(s) \models_{p,\alpha(i)} \alpha(g). \end{aligned}$$

If g has the form (g_1) the result follows similarly.

If g has the form g_1 `&&` g_2 then $s \models_{p,i} g \Leftrightarrow s \models_{p,i} g_1$ and $s \models_{p,i} g_2$. Now, by the induction hypothesis, $s \models_{p,i} g_1 \Leftrightarrow \alpha(s) \models_{p,\alpha(i)} \alpha(g_1)$ and $s \models_{p,i} g_2 \Leftrightarrow \alpha(s) \models_{p,\alpha(i)} \alpha(g_2)$. So

$$\begin{aligned} s \models_{p,i} g &\Leftrightarrow \alpha(s) \models_{p,\alpha(i)} \alpha(g_1) \text{ and } \alpha(s) \models_{p,\alpha(i)} \alpha(g_2) \\ &\Leftrightarrow \alpha(s) \models_{p,\alpha(i)} \alpha(g_1) \texttt{ \&\&} \alpha(g_2) \\ &\Leftrightarrow \alpha(s) \models_{p,\alpha(i)} \alpha(g). \end{aligned}$$

If g has the form g_1 `||` g_2 the result follows similarly. □

Lemma 3 *Let u be an update of $\mathcal{P}$, $\alpha \in Aut(\mathcal{SCD}(\mathcal{P}))$ and s a state such that $exec_{p,i}(s,u)$ is well-defined. Then $exec_{p,\alpha(i)}(\alpha(s),\alpha(u)) = \alpha(exec_{p,i}(s,u))$.*

Proof If u is `skip` the result is immediate.

Suppose u has the form $x = e$. As for Section 6.2, for a proctype p, variable name x and process identifier i with $proctype(i) = p$, we define $var(x)$ to be x if x is a global variable, and $p[i].x$ otherwise. Define $\alpha(var(x)) = x$ if $var(x) = x$, and $\alpha(var(x)) = p[\alpha(i)].x$ if $var(x) = p[i].x$.

If $x : int$ then suppose $(var(x) = a) \in s$. Then $\alpha((var(x) = a)) = (var(x) = a) \in \alpha(s)$ also. Suppose $eval_{p,i}(s,e) = b$. Then $eval_{p,\alpha(i)}(\alpha(s),\alpha(e)) = b$ by Lemma 1. Now we have $\alpha((var(x) = b)) = (var(x) = b)$, and so, from Table 1:

$$\begin{aligned} exec_{p,\alpha(i)}(\alpha(s), \text{`}x = \alpha(e)\text{'}) &= \big(\alpha(s) \setminus \{(var(x) = a)\}\big) \cup \{(var(x) = b)\} \\ &= \alpha\big(\big(s \setminus \{(var(x) = a)\}\big) \cup \{(var(x) = b)\}\big) \\ &= \alpha(exec_{p,i}(s, \text{`}x = e\text{'})). \end{aligned}$$

If $x : pid$, then suppose $(var(x) = a) \in s$. Then $\alpha((x = a)) = (\alpha(var(x)) = \alpha(a)) \in \alpha(s)$ also. Suppose $eval_{p,i}(s,e) = b$. Then $eval_{p,\alpha(i)}(\alpha(s),\alpha(e)) = \alpha(b)$ by Lemma 1. We have

$$\begin{aligned} \alpha((var(x) = b)) &= (\alpha(var(x)) \\ &= \alpha(b)) \end{aligned}$$

Table 1 Update execution rules

u	Conditions on s	Resulting state $exec_{p,i}(s,u)$
'skip'	None	s
'$x = e$'	$(var(x) = a) \in s$	$\big(s \setminus \{(var(x) = a)\}\big) \cup$ $\{(var(x) = eval_{p,i}(s,e))\}$
'$c!e_1, e_2, \ldots, e_k$'	$(c = [\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_m]) \in s$ $s \models_{p,i} \texttt{nfull}(c)$	$\big(s \setminus \{(c = [\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_m])\}\big) \cup$ $\{(c = [\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_m, (eval_{p,i}(s,e_1),$ $eval_{p,i}(s,e_2), \ldots, eval_{p,i}(s,e_k))])\}$
'$c?x_1, x_2, \ldots, x_k$'	$(c = [(a_{1,1}, a_{1,2}, \ldots, a_{1,k}),$ $\vec{a}_2, \ldots, \vec{a}_m]) \in s$ $s \models_{p,i} \texttt{nempty}(c)$ $(var(x_j) = b_j) \in s$ $(1 \le j \le k)$	$\big(s \setminus \{(c = [(a_{1,1}, a_{1,2}, \ldots, a_{1,k}), \vec{a}_2, \ldots, \vec{a}_m]),$ $(var(x_1) = b_1), (var(x_2) = b_2), \ldots,$ $(var(x_k) = b_k)\}\big)$ $\cup\{(c = [\vec{a}_2, \ldots, \vec{a}_m]), (var(x_1) = a_{1,1}),$ $(var(x_2) = a_{1,2}), \ldots, (var(x_k) = a_{1,k})\}$
'$x!e_1, e_2, \ldots, e_k$'	$(p[i].x = c) \in s$	$exec_{p,i}(s, \text{`}c!e_1, e_2, \ldots, e_k\text{'})$ (if well-defined)
'$x?x_1, x_2, \ldots, x_k$'	$(p[i].x = c) \in s$	$exec_{p,i}(s, \text{`}c?x_1, x_2, \ldots, x_k\text{'})$ (if well-defined)

Rules interpreted in the context of process i, an instantiation of proctype p

and so

$$\begin{aligned} exec_{p,\alpha(i)}(\alpha(s), \text{`}x = \alpha(e)\text{'}) &= \big(\alpha(s) \setminus \{(\alpha(var(x)) = \alpha(a))\}\big) \cup \{(\alpha(var(x)) = \alpha(b))\} \\ &= \alpha\big(\big(s \setminus \{(var(x) = a)\}\big) \cup \{(var(x) = b)\}\big) \\ &= \alpha(exec_{p,i}(s, \text{`}x = e\text{'}). \end{aligned}$$

The argument is similar if $x : chan\{\overline{T}\}$.

Suppose u has the form $x\,!\,e_1, e_2, \ldots, e_k$, where x is a static channel name, and that $x : chan\{T_1, T_2, \ldots, T_k\}$ so that $e_j : T_j$ $(1 \leq j \leq k)$. If $(x = [\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_m]) \in s$ for some $m < cap(x)$ it follows that

$$\alpha((x = [\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_m])) = (\alpha(x) = [\vec{a}_1{}^{\alpha}, \vec{a}_2{}^{\alpha}, \ldots, \vec{a}_m{}^{\alpha}]) \in \alpha(s).$$

For $1 \leq j \leq k$, let a_j denote $eval_{p,i}(s, e_j)$, and let $b_j = a_j$ if $T_j = int$, and $b_j = \alpha(a_j)$ otherwise, then by Lemma 1 we have $b_j = eval_{p,\alpha(i)}(\alpha(s), \alpha(e_j))$. Thus $(b_1, b_2, \ldots, b_k) = (a_1, a_2, \ldots, a_k)^{\alpha}$ (using the notation of Section 7.2). Then

$$\begin{aligned} &exec_{p,\alpha(i)}(\alpha(s), \text{`}\alpha(x)\,!\,\alpha(e_1), \alpha(e_2), \ldots, \alpha(e_k)\text{'}) \\ &\quad = \big(\alpha(s) \setminus \{(\alpha(x) = [\vec{a}_1{}^{\alpha}, \vec{a}_2{}^{\alpha}, \ldots, \vec{a}_m{}^{\alpha}])\}\big) \\ &\quad \cup \{(\alpha(x) = [\vec{a}_1{}^{\alpha}, \vec{a}_2{}^{\alpha}, \ldots, \vec{a}_m{}^{\alpha}, (b_1, b_2, \ldots, b_k)])\}. \end{aligned}$$

Now since

$$\begin{aligned} &\big(\alpha(s) \setminus \{(\alpha(x) = [\vec{a}_1{}^{\alpha}, \vec{a}_2{}^{\alpha}, \ldots, \vec{a}_m{}^{\alpha}])\}\big) \\ &\qquad \cup \{(\alpha(x) = [\vec{a}_1{}^{\alpha}, \vec{a}_2{}^{\alpha}, \ldots, \vec{a}_m{}^{\alpha}, (b_1, b_2, \ldots, b_k)])\} \\ &\qquad = \big(\alpha(s) \setminus \{(\alpha(x) = [\vec{a}_1{}^{\alpha}, \vec{a}_2{}^{\alpha}, \ldots, \vec{a}_m{}^{\alpha}])\}\big) \\ &\qquad \cup \{(\alpha(x) = [\vec{a}_1{}^{\alpha}, \vec{a}_2{}^{\alpha}, \ldots, \vec{a}_m{}^{\alpha}, (a_1, a_2, \ldots, a_k)^{\alpha}])\} \\ &\qquad = \quad \alpha\big(\big(s \setminus \{(x = [\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_m])\}\big) \\ &\qquad \cup \{(x = [\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_m, (a_1, a_2, \ldots, a_k)])\}\big) \\ &\qquad = \alpha(exec_{p,i}(s, \text{`}x\,!\,e_1, e_2, \ldots, e_k\text{'})) \end{aligned}$$

the result follows.

If x is a local variable of p and $(x = c) \in s$, where c is a static channel name, then $(x = \alpha(c)) \in \alpha(s)$, and

$$\begin{aligned} &exec_{p,\alpha(i)}(\alpha(s), \text{`}x\,!\,\alpha(e_1), \alpha(e_2), \ldots, \alpha(e_k)\text{'}) \\ &\quad = exec_{p,\alpha(i)}(\alpha(s), \text{`}\alpha(c)\,!\,\alpha(e_1), \alpha(e_2), \ldots, \alpha(e_k)\text{'}) \\ &\quad = \alpha(exec_{p,i}(s, \text{`}c\,!\,e_1, e_2, \ldots, e_k\text{'})) \text{ (by the above argument)} \\ &\quad = \alpha(exec_{p,i}(s, \text{`}x\,!\,e_1, e_2, \ldots, e_k\text{'})). \end{aligned}$$

Suppose that u has the form $x?x_1, x_2, \ldots, x_k$ where x is a static channel name with type $chan\{T_1, T_2, \ldots, T_k\}$ so that x_j has type T_j $(1 \leq j \leq k)$. Suppose that, for some $m < cap(x)$ and $(var(x_j) = b_j) \in s$ $(1 \leq j \leq k)$, $(x = [(a_{1,1}, a_{1,2}, \ldots, a_{1,k}), \vec{a}_2, \ldots, \vec{a}_m]) \in s$. Define $d_{1,j} = a_{1,j}$ if $T_j = int$, and $d_{1,j} = \alpha(a_{1,j})$ otherwise $(1 \leq j \leq k)$. Then, using the notation of Section 7.2, $(d_{1,1}, d_{1,2}, \ldots, d_{1,k}) =$

$(a_{1,1}, a_{1,2}, \ldots, a_{1,k})^\alpha$ and $\alpha(var(x_j) = a_{1,j}) = \alpha(var(x_j) = d_{1,j})$ $(1 \le j \le k)$. Similarly, define $d_j = b_j$ if $x_j : int$, and $d_j = \alpha(b_j)$ otherwise. Then $\alpha((var(x_j) = b_j)) = (\alpha(var(x_j)) = d_j)$. Since we have $\alpha((x = [(a_{1,1}, a_{1,2}, \ldots, a_{1,k}), \vec{a}_2, \ldots, \vec{a}_m])) = (\alpha(x) = [(d_{1,1}, d_{1,2}, \ldots, d_{1,k}), \vec{a}_2^\alpha, \ldots, \vec{a}_m^\alpha])$ which is in $\alpha(s)$ and $\alpha((var(x_j) = b_j)) = (\alpha(var(x_j)) = d_j) \in \alpha(s)$ $(1 \le j \le k)$, by Table 1 we have:

$$
\begin{aligned}
&exec_{p,\alpha(i)}(\alpha(s), \text{`}\alpha(x)?x_1, x_2, \ldots, x_k\text{'}) \\
&\quad = \big(\alpha(s) \setminus \{(\alpha(x) = [(d_{1,1}, d_{1,2}, \ldots, d_{1,k}), \vec{a}_2^\alpha, \ldots, \vec{a}_m^\alpha]), \\
&\qquad (\alpha(var(x_1)) = d_1), (\alpha(var(x_2)) = d_2), \ldots, (\alpha(var(x_k)) = d_k)\}\big) \cup \\
&\qquad \{(\alpha(x) = [\vec{a}_2^\alpha, \ldots, \vec{a}_m^\alpha]), (\alpha(var(x_1)) = d_{1,1}), (\alpha(var(x_2)) = d_{1,2}), \\
&\qquad \ldots, (\alpha(var(x_k)) = d_{1,k})\} \\
&\quad = \big(\alpha(s) \setminus \{(\alpha(x) = [(a_{1,1}, a_{1,2}, \ldots, a_{1,k})^\alpha, \vec{a}_2^\alpha, \ldots, \vec{a}_m^\alpha]), \\
&\qquad \alpha((var(x_1) = b_1)), \alpha((var(x_2) = b_2)), \ldots, \alpha((var(x_k) = b_k))\}\big) \cup \\
&\qquad \{(\alpha(x) = [\vec{a}_2^\alpha, \ldots, \vec{a}_m^\alpha]), \alpha((var(x_1) = a_{1,1})), \alpha((var(x_2) = a_{1,2})), \\
&\qquad\qquad \ldots, \alpha((var(x_k) = a_{1,k}))\} \\
&\quad = \alpha\big(\big(s \setminus \{(x = [(a_{1,1}, a_{1,2}, \ldots, a_{1,k}), \vec{a}_2, \ldots, \vec{a}_m]), \\
&\qquad (var(x_1) = b_1), (var(x_2) = b_2), \ldots, (var(x_k) = b_k)\}\big) \cup \\
&\qquad \{(x = [\vec{a}_2, \ldots, \vec{a}_m]), (var(x_1) = a_{1,1}), (var(x_2) = a_{1,2}), \ldots, (var(x_k) = a_{1,k})\}\big) \\
&\quad = \alpha(exec_{p,i}(s, \text{`}x?x_1, x_2, \ldots, x_k\text{'})).
\end{aligned}
$$

If x is a local variable of p and $(x = c) \in s$, where c is a static channel name. Then $(x = \alpha(c)) \in \alpha(s)$, and

$$
\begin{aligned}
&exec_{p,\alpha(i)}(\alpha(s), \text{`}x?x_1, x_2, \ldots, x_k\text{'}) = exec_{p,\alpha(i)}(\alpha(s), \text{`}\alpha(c)?x_1, x_2, \ldots, x_k\text{'}) \\
&\quad = \alpha(exec_{p,i}(s, \text{`}c?x_1, x_2, \ldots, x_k\text{'})) \text{ (by the above argument)} \\
&\quad = \alpha(exec_{p,i}(s, \text{`}x?x_1, x_2, \ldots, x_k\text{'})).
\end{aligned}
$$

□

The following lemma follows from the repeated application of Lemma 3:

Lemma 4 *Let $u_1, u_2, \ldots, u_k$ be updates of $\mathcal{P}$, $\alpha \in Aut(\mathcal{SCD}(\mathcal{P}))$ and s a state such that $exec_{p,i}(s, u_1; u_2; \ldots; u_k)$ is well-defined. Then*

$$exec_{p,\alpha(i)}(\alpha(s), \alpha(u_1); \alpha(u_2); \ldots; \alpha(u_k)) = \alpha(exec_{p,i}(s, u_1; u_2; \ldots; u_k)).$$

We can now prove the major result of this paper, Theorem 5:

Proof (Theorem 5) By Definition 9, we must show that (i) if $(s, t) \in R$ then $(\alpha(s), \alpha(t)) \in R$, and (ii) $\alpha(s_0) = s_0$.

If $(s, t) \in R$ then there is a process with pid i such that $proctype(i) = p$ (for some proctype p), and a statement z in p such that the guard of z holds for process i at s, and execution of the updates of z by process i at s leads to state t. Since $\alpha(\mathcal{P}) \equiv \mathcal{P}$

the statement $\alpha(z)$ (possibly re-arranged) also appears in proctype p. By Lemma 2, the guard of $\alpha(z)$ holds for process $\alpha(i)$ at $\alpha(s)$, and by Lemma 4, execution of the updates of $\alpha(z)$ by process $\alpha(i)$ at $\alpha(s)$ leads to state $\alpha(t)$. Therefore $(\alpha(s), \alpha(t)) \in R$.

We must show that for any proposition $(v = d)$ in s_0, $\alpha((v = d)) \in s_0$ also. In s_0, all static channels are empty, so for any static channel c, the propositions $(c = [\,])$ and $\alpha((c = [\,])) = (\alpha(c) = [\,])$ both belong to s_0. For each global variable x, $(x = x_0) \in s_0$, where x_0 is the initial value for x (specified at declaration). If $x : int$ then $\alpha((x = x_0)) = (x = x_0) \in s_0$. If $x : pid$ then we must have $\alpha(x_0) = x_0$ (since $\alpha(\mathcal{P}) \equiv \mathcal{P}$), so $\alpha((x = x_0)) = (x = \alpha(x_0)) = (x = x_0) \in s_0$.

For any local variable x, suppose x_0 is the initial value given for x in `run` statement i. Then $(p[i].x = x_0) \in s_0$. Let y_0 be the initial value given for x in `run` statement $\alpha(i)$, so that $(p[\alpha(i)].x = y_0) \in s_0$. If $x : int$ then, since $\mathcal{P} \equiv \alpha(\mathcal{P})$, the value for x in `run` statements i and $\alpha(i)$ must be the same, i.e. $x_0 = y_0$. So we have $\alpha((p[i].x = x_0)) = (p[\alpha(i)].x = x_0) = (p[\alpha(i)].x = y_0) \in s_0$. Suppose that $x : pid$ or $x : chan\overline{T}$. Then, since $\mathcal{P} \equiv \alpha(\mathcal{P})$, the value for x in `run` statement $\alpha(i)$ is the image under α of the value for x in `run` statement i, i.e. $y_0 = \alpha(x_0)$. We have $\alpha((p[i].x = x_0)) = (p[\alpha(i)].x = \alpha(x_0)) = (p[\alpha(i)].x = y_0) \in s_0$. □

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

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

Our algorithm for finding G (Algorithm 1) starts with a known valid subgroup H of $Aut(\mathcal{SCD}(\mathcal{P}))$, and adds valid coset representatives (see Definition 5) to the generators of H to obtain successively larger valid subgroups.

It can be shown [19] that

Theorem 6 *Algorithm* 1 *computes the largest valid subgroup of* $Aut(\mathcal{SCD}(\mathcal{P}))$.

We illustrate Algorithm 1 using the load-balancer example. Let $\mathcal{P}$ be the specification of Fig. 6. Generators for $Aut(\mathcal{SCD}(\mathcal{P}))$ computed by *nauty* are given in Section 7.2. The generators which do not fix the process identifier 9

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

```
X := generators of Aut(SCD(P))
H := ⟨{α ∈ X : α(P) ≡ P}⟩
U := representatives of right cosets of H in Aut(SCD(P)) except H
while U ≠ ∅ do
  U := U \ {α}
  if α(P) ≡ P then
    H := ⟨H ∪ {α}⟩
    if |Aut(SCD(P))|/|H| < |U| then
      U := representatives of right cosets of H in Aut(SCD(P)) except H
    end if
  end if
end while
```

are *not* valid for $\mathcal{P}$ since, if α is one of these generators, the declaration `pid blocked_client = 9` in $\mathcal{P}$ is replaced with `pid blocked_client =` α`(9)`, and $\alpha(9) \neq 9$, thus $\alpha(\mathcal{P}) \not\equiv \mathcal{P}$. The other generators are valid for $\mathcal{P}$, so $H = \langle (7\ 8)(cl1\ cl2), (11\ 12)(cl5\ cl6), (1\ 2)(se1\ se2), (2\ 3)(se2\ se3) \rangle$ is valid for $\mathcal{P}$. GAP tells us that $|Aut(\mathcal{SCD}(\mathcal{P}))| = 288$ and $|H| = 24$, so there are $|Aut(\mathcal{SCD}(\mathcal{P}))|/|H| = 12$ cosets of H in $Aut(\mathcal{SCD}(\mathcal{P}))$. We can use GAP to compute representatives $\alpha_1, \alpha_2, \ldots, \alpha_{11}$ for the 11 cosets of H in $Aut(\mathcal{SCD}(\mathcal{P}))$ which are distinct from H. The first nine of these are not valid for $\mathcal{P}$, but

$$\alpha_{10} = (4\ 6)(lb1\ lb3)(7\ 11)(cl1\ cl5)(8\ 12)(cl2\ cl6)$$

is valid for $\mathcal{P}$. When α_{10} is added to the generators of H, $|H| = 48$, so there are now $|Aut(\mathcal{SCD}(\mathcal{P}))|/|H| = 6$ cosets of H in $Aut(\mathcal{SCD}(\mathcal{P}))$. It is more efficient to check the final original coset representative α_{11} than to compute and check a new set of coset representatives. This is the purpose of the innermost conditional statement in Algorithm 1. As α_{11} is not valid for $\mathcal{P}$,

$$H = \langle (7\ 8)(cl1\ cl2), (11\ 12)(cl5\ cl6), (1\ 2)(se1\ se2), (2\ 3)(se1\ se2), \\ (4\ 6)(lb1\ lb3)(7\ 11)(cl1\ cl5)(8\ 12)(cl2\ cl6) \rangle$$

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

Algorithm 1 performs badly if the initial group H is small, and $Aut(\mathcal{SCD}(\mathcal{P}))$ very large. If H is the largest valid subgroup then $(|Aut(\mathcal{SCD}(\mathcal{P}))|/|H|) - 1$ coset representatives must be checked. For many examples, the initial size of H can be increased using an optimisation based on random conjugates [17, 19]. However, in the worst case, the largest valid subgroup may turn out to be trivial, thus the worst case complexity of Algorithm 1 is $O(|Aut(\mathcal{SCD}(\mathcal{P}))|)$.

It is possible to construct a Promela-Lite specification $\mathcal{P}$ such that an element $\alpha \in Aut(\mathcal{SCD}(\mathcal{P}))$ is invalid, but induces a genuine automorphism of the model $\mathcal{M}$ associated with $\mathcal{P}$. Thus, while Algorithm 1 is complete in that it computes the largest valid subgroup of $Aut(\mathcal{SCD}(\mathcal{P}))$, our notion of validity could be extended to determine more accurately whether an element of $Aut(\mathcal{SCD}(\mathcal{P}))$ induces a Kripke structure automorphism. As long as the set of valid elements of $Aut(\mathcal{SCD}(\mathcal{P}))$ still forms a group and validity can be checked automatically, Algorithm 1 can be used to compute the largest valid subgroup as usual.

8 SymmExtractor

In this Section we describe SymmExtractor, an automated symmetry detection tool for Promela based on the static channel diagram analysis techniques of Section 7. After providing an overview of the tool, we discuss the restrictions on the form of a Promela specification which must be satisfied before SymmExtractor can be applied. We then discuss two problems related to typechecking which SymmExtractor solves: how to deduce the type of an incompletely specified channel, and how to compare channel types.

We show how the GAP and *saucy* tools are used to compute the largest valid subgroup of $Aut(\mathcal{SCD}(\mathcal{P}))$, and give experimental results showing how SymmExtractor performs on a variety of example specifications.

8.1 An Overview of SymmExtractor

SymmExtractor is a Java program based on a Promela parser generated using the SableCC compiler generation framework [25]. The Promela grammar is adapted from a BNF grammar presented in [28], with the SPIN source code used to resolve ambiguity in the grammar specification.

The abstract syntax tree representation of the input specification is typechecked. Unlike for Promela-Lite, Promela specifications do not always contain the complete channel type information required by our approach, so we employ a type reconstruction algorithm from [18] to recover this information. Reconstructed channel types which are *recursive* are then converted to a minimised canonical form. The typed abstract syntax tree is checked to see whether it satisfies certain restrictions imposed by the theory of Section 7. If these restrictions are satisfied then the static channel diagram $\mathcal{SCD}(\mathcal{P})$ for the specification $\mathcal{P}$ is derived, and its automorphisms computed using the *saucy* program [12]. Algorithm 1 of Section 7.4 is then used to compute the largest subgroup of $Aut(\mathcal{SCD}(\mathcal{P}))$ which is valid for $\mathcal{P}$. Checking validity depends on the reconstructed type information obtained by SymmExtractor. SymmExtractor uses a GAP implementation of Algorithm 1 together with an optimisation based on random conjugates (see Section 7.4) to compute the largest valid subgroup of $Aut(\mathcal{SCD}(\mathcal{P}))$. The automatic symmetry detection process is summarised in Fig. 11.

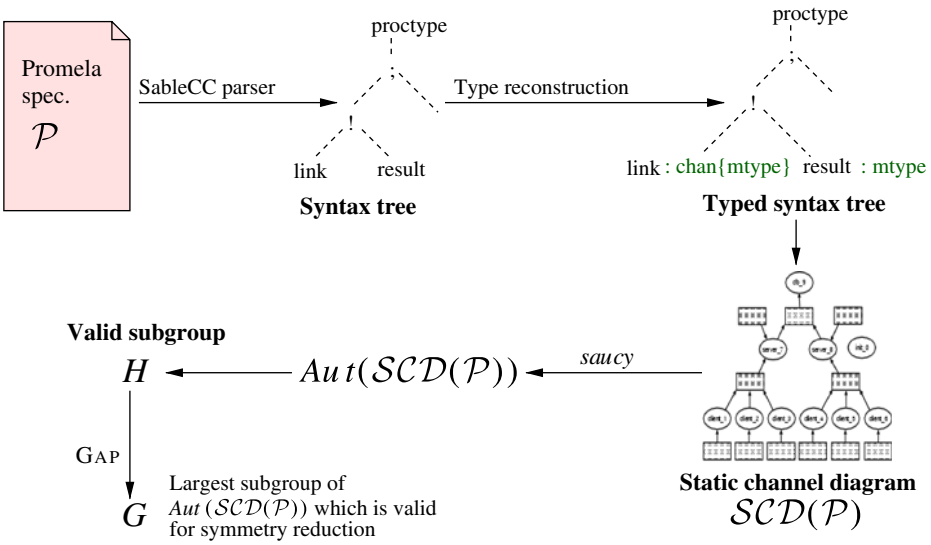

Fig. 11 The automatic symmetry detection processes used by SymmExtractor

SymmExtractor imposes two important restrictions on the form of a Promela specification. First, the `init` process must have the form:

```
init {
  atomic {
    run ⟨name⟩1(...);
    run ⟨name⟩2(...);
        ⋮
    run ⟨name⟩n(...);
    ⟨statement-list, ';'⟩
  };
  ⟨statement-list, ';'⟩
}
```

where the list ⟨*statement-list,* '`;`'⟩ of statements within the `atomic` block must be assignments of literal values to distinct variables. All `run` statements must appear within the atomic block of the `init` process. This ensures that the specification consists of a fixed number of running processes.

Second, the static channel diagram approach requires that all global channel declarations include a channel initialiser, and names of global channels are not reassigned.

The tool imposes some further minor restrictions which are documented in [17]. SymmExtractor also supports Promela features such as synchronous channels which, for ease of presentation, are not part of Promela-Lite. In particular, SymmExtractor allows a specification to include a *never claim*, an additional process used for *LTL* model checking [28]. A never claim can be viewed as a proctype with a single running instance, thus provision for never claims is handled by the theory of Section 7. Furthermore, since a valid static channel diagram automorphism must respect the structure of a never claim, a group of valid static channel diagram automorphisms is, by default, an invariance group for the property represented by a never claim.

8.2 Experimental Results

We now present experimental results running SymmExtractor on a variety of configurations of six families of Promela specifications.

8.2.1 Specification Families and Configurations

The families of specifications we consider are: *simple mutex*, *Peterson*, *Peterson without atomicity*, *resource allocator*, *three-tiered architecture* and *hypercube*.

The *simple mutex*, *Peterson* and *Peterson without atomicity* families are versions of mutual exclusion protocols. The first of these is illustrated in Fig. 3 for 5 processes and the others are based on variations of a specification presented in [2]. Configurations are identified via the number n of processes.

The *resource allocator* families consist of a set of prioritised *client* processes, each of which requires access to a resource, and a *resource allocator* process which takes requests from the clients wishing to use the resource, granting access to one client at a time, according to priority. The model is similar to an example used for symmetry reduction in partially symmetric systems [39]. We consider two kinds of

configuration. A configuration has signature a_0-a_1-...-a_{k-1}, where $a_i > 0$ ($0 \leq i < k$) if there are $k > 1$ distinct priority levels and *client* processes $1, 2, \ldots, a_0$ have priority level 0, $a_0 + 1, a_0 + 2, \ldots, a_1$ have priority level 1, etc. An alternating configuration, referred to as *alternating* x, where $x > 0$ is even, has two priority levels, x *client* processes, and the priority levels alternate between 0 and 1 every three *client* processes. For example, *alternating 10* denotes a 10-*client* configuration where *client* processes 1, 2, 5, 6, 9 and 10 have priority level 0 and *client* processes 3, 4, 7 and 8 have priority level 1.

A configuration in the *three-tiered architecture* consists of three process types: *client*, *server* and *database*. Each *client* process sends *request* messages to the *database* process, via their neighbouring *server* process, and *data* is returned to the *client* process (again, via the *server*). All channels are synchronous.

A configuration with $k > 0$ *server* processes and $a_i > 0$ *client* processes connected to server i ($1 \leq i \leq k$) has signature a_1-a_2-...-a_k. For example, 5-5-5-5 denotes a configuration with 4 *servers* and 20 *clients*, 5 connected to each server.

A member of the *hypercube* family consists of a set of *Node* processes sending messages through an m-dimensional hypercube network using a simple routing algorithm [41]. An m-dimensional hypercube is denoted *md*.

8.2.2 Results and Discussion

In Table 2 we give results for various configurations of the families described above. All of the Promela specifications used for these experiments are available online [15].

In each case we evaluate the following:

- $|Aut(\mathcal{SCD}(\mathcal{P}))|$ – size of the automorphism group of the static channel diagram associated with configuration $\mathcal{P}$
- $|H|$ – size of the initial subgroup generated by the valid generators of $Aut(\mathcal{SCD}(\mathcal{P}))$
- $|G|$ – size of the largest valid subgroup of $Aut(\mathcal{SCD}(\mathcal{P}))$, computed using Algorithm 1 (Section 7.4)
- *saucy* time – time (in seconds) for *saucy* to compute generators for $Aut(\mathcal{SCD}(\mathcal{P}))$
- *find largest* time– time (in seconds) to compute G given generators for $Aut(\mathcal{SCD}(\mathcal{P}))$
- *optimised find largest* time – time (in seconds) to compute G given generators for $Aut(\mathcal{SCD}(\mathcal{P}))$, using the random conjugates optimisation.

When all generators of $Aut(\mathcal{SCD}(\mathcal{P}))$ are valid, $Aut(\mathcal{SCD}(\mathcal{P}))$, H and G are equal, so there is no need to use Algorithm 1 (denoted by '='). In this case the random conjugates optimisation is irrelevant, indicated by the entry '–'. The entry '>12 h' indicates that $|G|$ could not be computed within 12 h. All experiments were performed on a PC with a 2.4 Intel Xeon processor and 3 Gb of main memory. The '*saucy* time' column illustrates an overhead associated with using *saucy* to compute $Aut(\mathcal{SCD}(\mathcal{P}))$, whatever its size.

Results for the three mutual exclusion families, as well as *three-tiered architecture* configurations, show that SymmExtractor is very efficient when all of the generators of $Aut(\mathcal{SCD}(\mathcal{P}))$ are valid. Here the value in the '*find largest* time' column is the time taken to check validity of the generators against $\mathcal{P}$. The results for *simple mutex*

Table 2 Experimental results for automatic symmetry detection

Configuration $\mathcal{P}$	$\lvert Aut(\mathcal{SCD}(\mathcal{P}))\rvert$ $\lvert Aut(\mathcal{SCD}(\mathcal{P}))\rvert$	$\lvert H\rvert$ $\lvert H\rvert$	$\lvert G\rvert$ $\lvert G\rvert$	Saucy time	Find largest time	Optimised find largest time
Simple mutex						
5	120	=	=	0.03	0.07	–
10	3.6×10^6	=	=	0.03	0.20	–
20	2.4×10^{18}	=	=	0.03	0.59	–
40	8.1×10^{37}	=	=	0.03	1.64	–
Peterson						
3	6	=	=	0.03	0.06	–
6	720	=	=	0.02	0.16	–
9	362880	=	=	0.04	0.30	–
12	4.8×10^8	=	=	0.03	0.56	–
Peterson without atomicity						
3	6	=	=	0.03	0.08	–
6	720	=	=	0.08	0.25	–
9	362880	=	=	0.03	0.52	–
12	4.8×10^8	=	=	0.03	0.89	–
Resource allocator						
3-4	5040	144	144	0.03	1.52	=
2-2-3	5040	24	24	0.03	5.24	=
5-5	3.6×10^6	14400	14400	0.05	8.34	=
3-3-4	3.6×10^6	864	864	0.03	114.49	=
Alternating 10	3.6×10^6	32	17280	0.03	18.12	8.5
Alternating 12	4.7×10^8	64	518400	0.04	314.87	33.59
Alternating 14	8.7×10^{10}	128	2.9×10^6	0.06	>12 h	116.39
Alternating 16	2.1×10^{13}	256	1.6×10^9	0.05	>12 h	543.87
Three-tiered architecture						
3-3-2	144	=	=	0.04	0.09	–
3-3-3	1296	=	=	0.05	0.14	–
4-4-3	6912	=	=	0.05	0.17	–
4-4-4	82944	=	=	0.05	0.26	–
5-5-5-5	5.0×10^9	=	=	0.07	0.55	–
Hypercube						
2d	8	2	4	0.04	0.59	0.84
3d	48	2	8	0.03	2.50	1.98
4d	384	2	16	0.04	99.11	26.73
5d	3840	2	32	0.08	7171.57	957.79

40 and 5-5-5-5 in the *three-tiered architecture* family show that SymmExtractor is sufficiently robust to handle large Promela specifications.

For the first four *resource allocator* configurations the initial valid subgroup H is the largest valid subgroup of $Aut(\mathcal{SCD}(\mathcal{P}))$. Since $H \neq Aut(\mathcal{SCD}(\mathcal{P}))$, it is necessary to run Algorithm 1 to confirm this. This is time-consuming for the 3-3-4 configuration. In these cases the random conjugates optimisation offers no benefit, indicated by the entry '='.

For the *alternating resource allocator* and *hypercube* specifications $\{id\} \subset H \subset G \subset Aut(\mathcal{SCD}(\mathcal{P}))$. Even when the number of cosets of H in $Aut(\mathcal{SCD}(\mathcal{P}))$ is large, if G is much larger than H the number of coset representatives which need to be checked for validity diminishes rapidly as valid representatives are found. However,

for the *alternating 14* and *alternating 16* configurations the number of cosets of H in $Aut(\mathcal{SCD}(\mathcal{P}))$ is so large that G could not be computed within 12 h without the random conjugates optimisation.

When $\mathcal{P}$ is the 3-dimensional *hypercube* specification, $Aut(\mathcal{SCD}(\mathcal{P}))$ has order 48. However G, the largest valid subgroup of $Aut(\mathcal{SCD}(\mathcal{P}))$, has order only 8. By examining the output from SymmExtractor it is clear that 2 of the 3 generators of $Aut(\mathcal{SCD}(\mathcal{P}))$ are discarded in this case simply because they do not strictly preserve `run` statements, so contravening the definition of *validity* given in Section 7.3.1. Since the effect on the `run` statements is superficial in this case, a slight relaxation of the notion of validity would allow us to automatically detect a much larger symmetry group. We observe a similar situation for the higher dimension hypercube specifications.

8.3 Integration with Spin

The SymmExtractor implementation forms part of TopSpin, our symmetry reduction package for the Spin model checker [20].

Given a Promela specification, the Spin tool outputs a C program, `pan.c`, which is then compiled and executed to produce a verification result. TopSpin uses the symmetry information provided by SymmExtractor to add symmetry reduction algorithms to `pan.c`. For details of these algorithms, together with experimental results showing the savings achieved by reduced model checking, see [17, 21]. Symmetry reduction using the groups computed by SymmExtractor makes verification of many of the specifications discussed in Section 8.2.2 tractable. However, verification of the larger, more complex specifications (e.g. *Peterson* 12, *Peterson without atomicity* 6+, and *three-tiered architecture* 5-5-5-5) remains intractable.

SymmExtractor also runs as a stand-alone tool which can be used as a front-end to other symmetry reduction packages for Promela specifications.

As discussed in Section 2, other implementations of symmetry reduction for Spin allow the user to specify the presence of symmetry using scalarsets. Our automatic approach has the advantage of requiring no user input and handling more general types of symmetry. However, the scalarset approach is very efficient since the scalarset conditions can be statically checked and require no computational group theoretic calculations. For a large, evolving specification with several families of processes, which is to be regularly modified and checked, we envisage a process where symmetry is initially detected using our automatic techniques. The modeller may then recognise that symmetries reported by our tool between a given set of components are amenable to specification via scalarsets. The modeller could then speed up symmetry detection by adding scalarset annotations to the parts of the specification modelling these components. This process would require a combination of our techniques with existing methods involving scalarsets.

9 Conclusions

In order to develop automatic symmetry detection techniques for Promela, we have presented the syntax and type system for Promela-Lite, a specification language which captures the essential features of Promela, but has a full Kripke structure

semantics and is sufficiently self-contained to allow for rigorous proof of theoretical results.

We have defined the *static channel diagram* $\mathcal{SCD}(\mathcal{P})$ associated with a Promela-Lite specification $\mathcal{P}$ and shown that it can be efficiently computed via a single pass of $\mathcal{P}$. After defining a group action of the automorphism group $Aut(\mathcal{SCD}(\mathcal{P}))$ on the states S of $\mathcal{M}$, the model associated with $\mathcal{P}$, we have proved that there is a largest *valid* subgroup $G \leq Aut(\mathcal{SCD}(\mathcal{P}))$ for which $\rho(G) \leq Aut(\mathcal{M})$, where ρ is the permutation representation of the group action. Furthermore, we have presented a computational group theoretic algorithm for computing the group G. This technique allows a subgroup of $Aut(\mathcal{M})$ to be efficiently derived from the specification $\mathcal{P}$ which can be exploited for symmetry reduced model checking.

We have described our tool SymmExtractor for the automatic detection of symmetry in Promela specifications. By imposing a small set of restrictions to the format of specifications accepted by SymmExtractor we can be confident, thanks to our thorough theoretical analysis of our techniques for Promela-Lite, that our approach is sound.

We have provided concrete evidence, via detailed experiments for a range of Promela specifications, of the usefulness of our tool.

Future work includes relaxing the restrictions on Promela-Lite to allow for the use of process identifiers in arithmetic operations, the automatic detection of other types of symmetry (arising from permutations of global variables, for example), the automatic detection of partial symmetry [23], and the combination of our automatic techniques with symmetry specification methods involving scalarsets.

Acknowledgements We would like to thank Simon Gay for his advice concerning the type theoretic aspects of this work, and the anonymous referees for their valuable suggestions towards improving this paper.

Appendix A Type System for Promela-Lite

An overview of the type system of Promela-Lite is given in Figs. 13 and 14. The notation used for typing rules is taken from [4]. The forms of type judgement that we use, together with the general form of a typing rule are summarised in Fig. 12.

For presentation of the type system we introduce a *tuple* type, to represent the form of arguments for a proctype. A proctype which accepts an ordered list of arguments of types $T_1, T_2, \ldots, T_k$ has type $(T_1, T_2, \ldots, T_k)$ $(k > 0)$. For brevity, we use $\overline{T}$ to refer to a list of types $T_1, T_2, \ldots, T_k$ $(k > 0)$, and similar notation for lists of expressions, literals, etc.

In the T-Pid-Literal typing rule, n denotes the number of process in the Promela-Lite specification. This information is necessary for type-checking since a *pid* variable must only be assigned to a literal value in the set $\{0, 1, \ldots, n\}$ (where 0 is used as a default value).

A recursive channel type can always be equivalently expressed in the form $chan\{\overline{T}\}$ by a one-step unfolding. For ease of presentation, the typing rules assume that all recursive types have been unfolded in this way. A type variable 'X' is said to be *bound* if it is introduced by the prefix '*rec* X .' and then occurs within the scope of this prefix. A type variable which is not bound is said to be *free*. A *well-formed*

Judgements

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

General form of a type rule

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

Fig. 12 Notation for type rules

type is one for which there are no free type variables. The types *int*, *chan*{*int*} and *rec X* . *chan*{*X*, *int*} are all well-formed; the type *chan*{*X*} is not (Figs. 13 and 14).

Appendix B Kripke Structure Semantics for Promela-Lite

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

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

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

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

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

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

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

Environment

$$\frac{}{\emptyset \vdash \diamond}\ (\text{T-ENV-}\emptyset) \qquad \frac{\Gamma \vdash \diamond \qquad \Gamma \vdash T \qquad x \notin dom(\Gamma)}{\Gamma, x : T \vdash \diamond}\ (\text{T-ENV-}x)$$

Expressions

$$\frac{\Gamma \vdash \diamond \qquad a \in \mathbb{Z}}{\Gamma \vdash a : int}\ (\text{T-INT-LITERAL}) \qquad \frac{\Gamma \vdash \diamond \qquad a \in \{0, 1, \ldots, n\}}{\Gamma \vdash a : pid}\ (\text{T-PID-LITERAL})$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{\_pid} : pid}\ (\text{T-\_PID}) \qquad \frac{\Gamma \vdash e : T}{\Gamma \vdash \texttt{(}e\texttt{)} : T}\ (\text{T-PARENTHESIS-}e)$$

$$\frac{\Gamma \vdash \diamond \qquad \Gamma \vdash \overline{T}}{\Gamma \vdash \texttt{null} : chan\{\overline{T}\}}\ (\text{T-NULL}) \qquad \frac{\Gamma, x : T, \Gamma' \vdash \diamond}{\Gamma, x : T, \Gamma' \vdash x : T}\ (\text{T-VAR})$$

$$\frac{\Gamma \vdash e_1 : int \qquad \Gamma \vdash e_2 : int \qquad \circ \in \{+, -, *\}}{\Gamma \vdash e_1 \circ e_2 : int}\ (\text{T-ARITH}) \qquad \frac{\Gamma \vdash c : chan\{\overline{T}\}}{\Gamma \vdash \texttt{len(}c\texttt{)} : int}\ (\text{T-LEN})$$

Guards

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

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

$$\frac{\Gamma \vdash g\ \texttt{OK}}{\Gamma \vdash \texttt{(}g\texttt{)}\ \texttt{OK}}\ (\text{T-PARENTHESIS-}g) \qquad \frac{\Gamma \vdash g\ \texttt{OK}}{\Gamma \vdash \texttt{!}g\ \texttt{OK}}\ (\text{T-NOT})$$

$$\frac{\Gamma \vdash g_1\ \texttt{OK} \qquad \Gamma \vdash g_2\ \texttt{OK}}{\Gamma \vdash g_1\ \texttt{\&\&}\ g_2\ \texttt{OK}}\ (\text{T-AND}) \qquad \frac{\Gamma \vdash g_1\ \texttt{OK} \qquad \Gamma \vdash g_2\ \texttt{OK}}{\Gamma \vdash g_1\ \texttt{||}\ g_2\ \texttt{OK}}\ (\text{T-OR})$$

Basic updates

$$\frac{\Gamma \vdash x : T \qquad \Gamma \vdash e : T \qquad x \text{ is not a static channel} \qquad T \neq (T_1, T_2, \ldots, T_k)}{\Gamma \vdash x = e\ \texttt{OK}}\ (\text{T-ASSIGN})$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{skip}\ \texttt{OK}}\ (\text{T-SKIP})$$

Fig. 13 Type system for Promela-Lite

States of a Specification

A state of a Promela-Lite specification $\mathcal{P}$ can be expressed as an ordered tuple consisting of a value for each variable in the specification. However, as in Section 7.3, it is more convenient here to reason about a state as a set of propositions.

The set S of (potential) states of $\mathcal{M}$ consists of every possible assignment to variables and channels of $\mathcal{P}$. Since the range of allowed integer values is finite, S is a finite set.

Statements

$$\frac{\Gamma \vdash g \texttt{ OK} \qquad \Gamma \vdash u_i \texttt{ OK } (1 \leq i \leq l)}{\Gamma \vdash \texttt{atomic \{ } g \texttt{ -> } u_1\texttt{; } u_2\texttt{; ...; } u_l \texttt{ \} OK}} \text{ (T-UPDATE)}$$

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

$$\frac{\begin{array}{c}\Gamma \vdash g \texttt{ OK} \qquad \Gamma \vdash c : \mathit{chan}\{\overline{T}\} \qquad \Gamma \vdash \overline{x} : \overline{T} \\ \Gamma \vdash u_i \texttt{ OK } (2 \leq i \leq l) \qquad \overline{x} \text{ are all different} \qquad \text{none of the } \overline{x} \text{ are static channels}\end{array}}{\Gamma \vdash \texttt{atomic \{ (}g\texttt{) \&\& nempty(}x\texttt{) -> }c\texttt{?}\overline{x}\texttt{; } u_2\texttt{; ...; } u_l \texttt{ \} OK}} \text{ (T-RECV)}$$

Declarations

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

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

Processes

$$\frac{\begin{array}{c}\Gamma \vdash \overline{T} \qquad \Gamma, \overline{x} : \overline{T} \vdash s_i \texttt{ OK } (1 \leq i \leq l) \\ \Gamma, p : (\overline{T}) \vdash \langle \mathit{proctype} \rangle^* \ \langle \mathit{init} \rangle \texttt{ OK} \\ \{p, \overline{x}\} \cap \mathit{dom}(\Gamma) = \emptyset \qquad p \text{ and the } \overline{x} \text{ are all different}\end{array}}{\begin{array}{c}\Gamma \vdash \texttt{proctype } p\texttt{(}\overline{T}\ \overline{x}\texttt{) \{ do :: } s_1\texttt{ :: } s_2\texttt{ :: ... :: } s_l\texttt{ od \}} \\ \langle \mathit{proctype} \rangle^* \ \langle \mathit{init} \rangle \texttt{ OK}\end{array}} \text{ (T-PROCTYPE)}$$

$$\frac{\Gamma \vdash p : (\overline{T}) \qquad \Gamma \vdash \overline{a} : \overline{T} \qquad \overline{a} \subseteq \mathbb{Z} \cup \{\texttt{null}\} \cup \{\text{static channels}\}}{\Gamma \vdash \texttt{run } p\texttt{(}\overline{a}\texttt{) OK}} \text{ (T-RUN)}$$

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

Fig. 14 Type system for Promela-Lite contd

Initial State

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

For a global variable x with $x : T$, let $\mathit{init}(x)$ denote the value in $\mathit{lit}(T)$ to which x is assigned at its declaration. For a local variable $p[i].x$ with $p[i].x : T$, let $\mathit{init}(p[i].x)$

denote the initial value in $lit(T)$ to which x is assigned in the ith `run` statement. $\mathcal{M}$ has a single initial state s_0, defined thus:

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

Expression Evaluation

Function $eval_{p,i}$ is defined in Section 6.2. Let $s \in S$ be a state of $\mathcal{M}$. Then:

- $eval_{p,i}(s, x) = a$ if $(x = a) \in s$ (i.e. x is a global variable)
- $eval_{p,i}(s, x) = a$ if $(p[i].x = a) \in s$ (i.e. x is a local variable of p)
- $eval_{p,i}(s, c) = c$ if c is a static channel name or `null`
- $eval_{p,i}(s, a) = a$ if $a \in \mathbb{Z}$
- $eval_{p,i}(s, \texttt{\_pid}) = i$
- $eval_{p,i}(s, \texttt{len(}c\texttt{)}) = k$ if c is a static channel and $(c = [\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_m]) \in s$ $(0 \leq k \leq cap(c))$
- $eval_{p,i}(s, \texttt{len(null)}) = 0$
- $eval_{p,i}(s, \texttt{len(}x\texttt{)}) = eval_{p,i}(s, len(c))$ if $(p[i].x = c) \in s$
- $eval_{p,i}(s, \texttt{(}e\texttt{)}) = eval_{p,i}(s, e)$
- $eval_{p,i}(s, e_1 \circ e_2) = eval_{p,i}(s, e_1) \circ eval_{p,i}(s, e_2)$ (where $\circ \in \{+, -, *\}$).

In practice int is a finite range of integers. Let $min(int)$ and $max(int)$ denote the minimum and maximum values in this range, and assume $min(int) < 0$. If the result $eval_{p,i}(s, e_1) \circ eval_{p,i}(s, e_2)$ falls outside of the allowed range, we define $eval_{p,i}(s, e_1 \circ e_2)$ to be

$$((eval_{p,i}(s, e_1) \circ eval_{p,i}(s, e_2) + |min|) \bmod (max - min)) - |min|.$$

This definition means that the result of such a calculation is *truncated* so that e.g. $max(int) + 1 = min(int)$. This follows the approach used by SPIN to deal with out-of-range operations in Promela specifications [28].

Satisfaction of Guards

We use the $eval_{p,i}$ function to define a relation $\models_{p,i}$ between states and guards which determines whether a guard holds at a given state. For a guard g of the form $\langle guard \rangle$ (see Fig. 8) and a state $s \in S$, with p and i as above, $s \models_{p,i} g$ means that the state s satisfies the guard g in the context of p and i. The relation $\models_{p,i}$ is defined as follows:

- $s \models_{p,i} e_1 \bowtie e_2 \Leftrightarrow eval_{p,i}(s, e_1) \bowtie eval_{p,i}(s, e_2)$ (where $\bowtie \in \{\texttt{==}, \texttt{!=}, \texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}\}$)[2]

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

- $s \models_{p,i} \texttt{nfull}(c) \Leftrightarrow (c = [\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_m]) \in s$ and $cap(c) > m$, where c is a static channel
- $s \models_{p,i} \texttt{nempty}(c) \Leftrightarrow (c = [\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_m]) \in s$ and $m > 0$, where c is a static channel
- $s \models_{p,i} \texttt{nfull}(x)/\texttt{nempty}(x) \Leftrightarrow (p[i].x = c) \in s$ and $s \models_{p,i} \texttt{nfull}(c)/\texttt{nempty}(c)$, where x is a local declared channel of p
- $s \models_{p,i} \texttt{!}\, g$ iff $s \not\models_{p,i} g$
- $s \models_{p,i} g_1 \texttt{\&\&}\, g_2$ iff $s \models_{p,i} g_1$ and $s \models_{p,i} g_2$
- $s \models_{p,i} g_1 \texttt{||}\, g_2$ iff $s \models_{p,i} g_1$ or $s \models_{p,i} g_2$
- $s \models_{p,i} \texttt{(}g\texttt{)}$ iff $s \models_{p,i} g$.

Effect of Updates

For a variable x, define $var(x)$ as for Section 6.2. For each update u described by the $\langle update \rangle$ rule in Fig. 8, the effect of u on a state s (in the context of a process i with proctype p) is given in Table 1.

Note that for certain updates it may be the case that *none* of the rules of Table 1 are applicable. For example, suppose $(c = [a_1, a_2, \ldots, a_m]) \in s$, where $m = cap(c)$, i.e. the static channel c is full in state s. In this case there is no rule which defines the effect of executing '$c\,!\,e_1, e_2, \ldots, e_k$', since a condition of the rule for sending on static channels is that the channel must not be full. We say that $exec_{p,i}(s, u)$ is undefined if no rule of Table 1 is applicable.

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

Theorem 3 (Section 6.2) holds if, for a well-typed Promela-Lite specification $\mathcal{P}$, if the guard associated with a statement of $\mathcal{P}$ is satisfied at state $s \in \mathcal{M}$, then the rules of Table 1 lead to a well-defined next-state t. In other words, the theorem holds if execution of a well-typed specification at a given state can always progress when some process has a guard which is true at the state.

The proof of Theorem 3 relies on the following lemma:

Lemma 1 *Let $\mathcal{P}$, $\mathcal{M}$ and s be as above. Let u be an update appearing in a statement of proctype p, and suppose $proctype(i) = p$. If u is* '`skip`' *or* 'x `=` e' *then $exec_{p,i}(s, u)$ is well-defined.*

Proof If u is '`skip`' then the definition of $exec_{p,i}(s, u)$ places no conditions on s, and it follows that $exec_{p,i}(s, u) = s$.

Let Γ be the typing environment comprised of entries for the global variables and static channels of $\mathcal{P}$, proctypes appearing before p in $\mathcal{P}$, and the local variables of p. If u has the form 'x `=` e', where x is an identifier and e an expression then, since

$\Gamma \vdash u$ OK, x is not a static channel name, and both x and e have type T where T is a well-formed type which is not the type of a proctype (rule T-Assign). Thus x is the name of a global variable or a local variable of p.

If x is the name of a global variable then we must have $(x = a) \in s$ for some $a \in T$. Therefore, according to Table 1, $exec_{p,i}(s, u) = \big(s \setminus \{(x = a)\}\big) \cup \{(x = eval_{p,i}(e))\}$, which is clearly well-defined.

On the other hand if x is the name of a local variable then $(p[i].x = a) \in s$ for some $a \in T$, and we have $exec_{p,i}(s, u) = \big(s \setminus \{(p[i].x = a)\}\big) \cup \{(p[i].x = eval_{p,i}(e)\}$. Again, this is a well-defined state. The result follows. □

Proof of Theorem 3 Let Γ be the typing environment as defined in the proof of Lemma 1, and let $\langle stmnt \rangle$ denote the Promela-Lite statement `atomic { g -> u1; u2; ... ul }`. We must show that if $s \models_{p,i} g$ then $exec_{p,i}(s, u_1, u_2) \ldots, u_l)$ is well-defined.

Suppose u_1 has the form `skip` or x `=` e. Then by Lemma 1, $exec_{p,i}(u_1)$ is well-defined.

Suppose u_1 has the form x`!`$\overline{e}$. Then x has type $chan\{\overline{T}\}$ in Γ, so x is either a local variable of p, or a static channel name. There is no typing rule from which $\Gamma \vdash u_1$ OK can be inferred, thus rule T-Update cannot be used to infer that $\Gamma \vdash \langle stmnt \rangle$ OK. Thus $\Gamma \vdash \langle stmnt \rangle$ OK must follow from rule T-Send. Therefore the guard g must have the form $\langle h \rangle$ `&&nfull(`x`)`, or just `nfull(`x`)` (see Section 6.2). Since, by hypothesis, $s \models_{p,i} g$, we must have $s \models_{p,i}$ `nfull(`x`)`. Suppose x is a static channel name, so that $(x = [\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_m]) \in s$, where $0 \leq m < cap(x)$. The conditions on s required by the rule for $exec_{p,i}(s, u_1)$ are satisfied. It is easy to see that the resulting state is well-formed. If x is a local variable of p then $(p[i].x = c) \in s$, where c is a static channel name or `null`. However, $s \models_{p,i}$ `nfull(`x`)` $\Leftrightarrow s \models_{p,i}$ `nfull(null)`, and we cannot have $s \models_{p,i}$ `nfull(null)`. Thus c is a static channel name, and $exec_{p,i}(s, u_1) = exec_{p,i}(s, c\,!\,\overline{e})$, which is well-defined by the above argument.

Suppose u_1 has the form x`?`$\overline{x}$. Then by a similar argument (using the fact that the x_i must be distinct, and that no x_i is a static channel name), $exec_{p,i}(s, u_1)$ is well-defined.

We have shown that $exec_{p,i}(s, u_1)$ is well-defined. Suppose that $exec_{p,i}(s, u_j)$ is well-defined for some $1 \leq j < l$. The type rules for statements (T-Update, T-Send and T-Recv) ensure that u_{j+1} has the form `skip` or x `=` a. By Lemma 1 it follows that $exec_{p,i}(exec_{p,i}(s, u_j), u_{j+1})$ is well-defined.

Since $exec_{p,i}(s, u_1)$ is well-defined,

$$exec_{p,i}(\ldots exec_{p,i}(exec_{p,i}(s, u_1), u_2) \ldots, u_l) = exec_{p,i}(s, u_1; u_2; \ldots; u_l)$$

is well-defined. □

Deriving a Kripke Structure

Let $\mathcal{P}$ be a Promela-Lite specification. The states S and initial state s_0 of $\mathcal{M}$ are as defined above. The transition relation R is defined as follows. Let $s \in S$ and let `atomic { g -> u1; u2; ...; uk }` be a statement of proctype p in $\mathcal{P}$. Suppose process i is an instantiation of p. If $s \models_{p,i} g$ then $(s, exec_{p,i}(s, u_1; u_2; \ldots; u_k)) \in R$. By the above theorem, $exec_{p,i}(s, u_1; u_2; \ldots; u_k)$ is well-defined.

References

1. Aho, A., Sethi, R., Ullman, J.: Compilers—Principles, Techniques and Tools. Addison-Wesley, Reading (1986)
2. Bosnacki, D., Dams, D., Holenderski, L.: Symmetric spin. Int. J. Softw. Tools Technol. Transf. **4**(1), 65–80 (2002)
3. Cameron, P.: Permutation groups. London Mathematical Society Student Texts, vol. 45. Cambridge University Press, Cambridge (1999)
4. Cardelli, L.: Type systems. In: Tucker, A. (ed.) The Computer Science and Engineering Handbook, New York, USA, pp. 2208–2236. CRC, Boca Raton (1997)
5. Clarke, E., Emerson, E.: Synthesis of synchronization skeletons for branching time temporal logic. In: Proceedings of the Workshop in Logic of Programs. Lecture Notes in Computer Science, Yorktown Heights, N.Y., vol. 131. Springer, New York (1981)
6. Clarke, E., Emerson, E., Jha, S., Sistla, A.: Symmetry reductions in model-checking. In: Hu, A., Vardi, M. (eds.) Proceedings of the 10th International Conference on Computer-aided Verification (CAV '98). Lecture Notes in Computer Science, Vancouver, British Columbia, Canada, vol. 1427, pp. 147–158. Springer, New York (1998)
7. Clarke, E., Emerson, E., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. **8**(2), 244–263 (1986)
8. Clarke, E., Enders, R., Filkorn, T., Jha, S.: Exploiting symmetry in temporal logic model checking. Form. Methods Syst. Des. **9**(1–2), 77–104 (1996)
9. Clarke, E., Grumberg, O., Long, D.: Model checking and abstraction. ACM Trans. Program. Lang. Syst. **16**(5), 1512–1542 (1994)
10. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT, Cambridge (1999)
11. Cohn, P.: Algebra. Wiley, New York (1982)
12. Darga, P., Liffiton, M., Sakallah, K., Markov, I.: Exploiting structure in symmetry detection for CNF. In: Proceedings of the 41st Annual Conference on Design Automation, San Diego, CA, USA, pp. 530–534. ACM, New York (2004)
13. Derepas, F., Gastin, P.: Model checking systems of replicated processes with Spin. In: Dwyer, M.B. (ed.) Proceedings of the 8th International SPIN Workshop (SPIN 2001). Lecture Notes in Computer Science, Toronto, Canada, vol. 2057, pp. 235–251. Springer, New York (2001)
14. Dill, D., Drexler, A., Hu, A., Yang, C.H.: Protocol verification as a hardware design aid. In: Proceedings 1992 IEEE International Conference on Computer Design: VLSI in Computer & Processors (ICCD'92), Cambridge, MA, USA, pp. 522–525. IEEE Computer Society, Los Alamitos (1992)
15. Donaldson, A.: Thesis website. http://www.dcs.gla.ac.uk/people/personal/ally/thesis/
16. Donaldson, A., Miller, A., Calder, M.: Finding symmetry in models of concurrent systems by static channel diagram analysis. Electron. Notes Theor. Comp. Sci. **128**(6), 161–177 (2005)
17. Donaldson, A.F.: Automatic Techniques for Detecting and Exploiting Symmetry in Model Checking. PhD Thesis. Department of Computing Science, University of Glasgow, UK (2007)
18. Donaldson, A.F., Gay, S.: ETCH: an enhanced type checking tool for Promela. In: Godefroid, P. (ed.) Proceedings of the 12th International SPIN Workshop (SPIN 2005). Lecture Notes in Computer Science, Barcelona, Spain, vol. 3639, pp. 237–242. Springer, New York (2005)
19. Donaldson, A.F., Miller, A.: Automatic symmetry detection for model checking using computational group theory. In: Fitzgerald, J., Hayes, I., Tarlecki, A. (eds.) Proceedings of the 13th International Symposium on Formal Methods (FM 2005). Lecture Notes in Computer Science, Newcastle, UK, vol. 3582, pp. 481–496. Springer, New York (2005)
20. Donaldson, A.F., Miller, A.: A computational group theoretic symmetry reduction package for the SPIN model checker. In: Proceedings of the 11th International Conference on Algebraic Methodology and Software Technology (AMAST'06). Lecture Notes in Computer Science, Kuressaare, Estonia, vol. 4019, pp. 374–380. Springer, New York (2006)
21. Donaldson, A.F., Miller, A.: Exact and approximate strategies for symmetry reduction in model checking. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) Proceedings of the 14th International Symposium on Formal Methods (FM 2006). Lecture Notes in Computer Science, Hamilton, Ontario, Canada, vol. 4085, pp. 541–556. Springer, New York (2006)
22. Donaldson, A.F., Miller, A.: Extending symmetry reduction techniques to a realistic model of computation. Electron. Notes Theor. Comput. Sci. **185**, 63–76 (2007)

23. Emerson, E., Havlicek, J., Trefler, R.: Virtual symmetry reduction. In: Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, CA, USA, pp. 121–131. IEEE Computer Society Press, Silver Spring (2000)
24. Emerson, E., Sistla, A.: Symmetry and model checking. Form. Methods Syst. Des. **9**(1–2), 105–131 (1996)
25. Gaggnon, E., Handren, L.: SableCC, an object-oriented compiler framework. In: Proceedings of TOOLS USA, pp. 140–154. IEEE Computer Society, Los Alamitos (1998)
26. Gap Group: GAP– Groups Algorithms and Programming, Version 4.4. Aachen, St. Andrews. http://www.gap-system.org/ (2006)
27. Godefroid, P.: Exploiting symmetry when model-checking software (extended abstract). In: Wu, J., Chanson, S., Gao, Q. (eds.) Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols and Protocol Specification, Testing and Verification (FORTE/PSTV '99). International Federation For Information Processing, Beijing, China, vol. 156, pp. 257–275. Kluwer, Deventer (1999)
28. Holzmann, G.: The SPIN Model Checker: Primer and Reference Manual. Addison Wesley, Boston (2003)
29. Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. ACM Trans. Prog. Lang. Syst. **23**(3), 396–450 (2001)
30. Ip, C., Dill, D.: Better verification through symmetry. Form. Methods Syst. Des. **9**, 41–75 (1996)
31. Larson, K., Pettersson, P., Yi, W.: UPPAAL in a nutshell. Softw. Tools Technol. Transf. **1**(1/2), 134–152 (1997)
32. McKay, B.: *Nauty* user's guide (version 1.5). Technical Report TR-CS-90-02. Australian National University, Computer Science Department (1990)
33. McMillan, K.L.: Symbolic Model Checking. Kluwer, Boston (1993)
34. Miller, A., Donaldson, A., Calder, M.: Symmetry in temporal logic model checking. Comput. Surv. **36**(3), September (2006)
35. Nalumasu, R., Gopalakrishnan, G.: Explicit-enumeration based verification made memory-efficient. In: Johnston, S. (ed.) Proceedings of the 12th IFIP International Conference on Computer Hardware Description Languages and their Applications (CHDL'95). IFIP, Chiba, Japan, pp. 617–622. Elsevier, Amsterdam (1995)
36. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in CÆSAR. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) Proceedings of the 5th International Symposium on Programming. Lecture Notes in Computer Science, Torino, Italy, vol. 137, pp. 195–220. Springer, New York (1982)
37. Rose, J.: A Course in Group Theory. Dover, New York (1994)
38. Saffrey, P.: Optimising Communication Structure for Model Checking. PhD Thesis. Department of Computing Science, University of Glasgow, July (2003)
39. Sistla, A., Godefroid, P.: Symmetry and reduced symmetry in model checking. In: Berry, G., Comon, H., Finkel, A. (eds.) Proceedings of the 13th International Conference on Computer-aided Verification (CAV 2001). Lecture Notes in Computer Science, Paris, France, vol. 2102, pp. 91–103. Springer, New York (2001)
40. Sistla, A., Gyuris, V., Emerson, E.: SMC: a symmetry-based model checker for verification of safety and liveness properties. ACM Trans. Softw. Eng. Methodol. **9**, 133–166 (2000)
41. Tanenbaum, A., van Steen, M.: Distributed Systems Principles and Paradigms. Prentice Hall, Englewood Cliffs (2002)