Symmetry Detection for Model Checking

Shao Jie Zhang[2], Jun Sun[2], Chengnian Sun[1], Yang Liu[3], Junwei Ma[2], and Jin Song Dong[1]

[1] National University of Singapore
[2] Singapore University of Technology and Design
[3] School of Computer Engineering, Nanyang Technological University

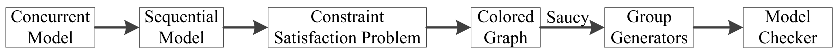
Abstract. We present an automatic approach to detecting symmetry relations for general concurrent models. Despite the success of symmetry reduction in mitigating state explosion problem, one essential step towards its soundness and effectiveness, *i.e.*, how to discover sufficient symmetries with least human effort, is often either overlooked or oversimplified. In this work, we show how a concurrent model can be viewed as a constraint satisfaction problem (CSP), and present an algorithm capable of detecting arbitrary symmetries arising from the CSP which induce automorphisms of the model. Unlike previous approaches, our method can automatically detect both various process and data symmetries as demonstrated via a number of systems. Further, we propose an inductive approach to inferring symmetries in a parameterized system from the symmetries detected over a small set of its instances.

1 Introduction

In practice, a certain (sometimes rich) degree of symmetries is ubiquitous in concurrent and distributed systems [29, 42]. The authors in [29] investigate the automorphism groups of a number of representative real-world complex networks, including a broad selection of biological, technological and social networks. All these systems have been found to have a nontrivial symmetric structure. In theory, given a model, a symmetry is an automorphism of its underlying state space (which can be viewed as a graph). A naive (and complete) symmetry detection method thus needs to explore the complete space. In general, if a symmetry detection method is performed on a state space, then the complete state space is required to be constructed in prior. It is not only computationally expensive or impossible, but also against the original goal of symmetry reduction to reduce the explored state space. A practical and popular approach is to use static analysis to derive symmetries at model level [24, 40].

Existing symmetry detection approaches have two main limitations. First, the soundness and efficiency highly depend on human effort. It is generally too difficult for machines to look through the behavior of concurrent models to pin down symmetries correctly. Most approaches require users to provide correct symmetries, which is tedious and error-prone. Some languages provide dedicated instructions for specifying symmetries [24, 36, 37]. For instance, Murφ provides a special data type with a list of syntactic restrictions. All values of a variable that belongs to this type are equivalent. Although there are automatic approaches which do not need expert insights, they are designed for specific languages [26, 25], or require models to be written in specific patterns [14, 15]. Thus they trade off generality for efficiency, and consequently a user has to transform his problem into a form amenable to the approach. Second, existing approaches can only handle a specific class of symmetries and largely ignore other classes of symmetries which could reduce state space significantly. As a result, symmetries in the underlying state space are only partially discovered.

In this work, we develop a novel approach for symmetry detection which addresses these two limitations. Not restricted to a particular modeling language, our approach works for general concurrent models (*i.e.*, concurrent composition of finite-state machines which could communicate through channels, synchronous events or shared memories) in a fully automatic way. Further, it is able to detect much more kinds of process symmetries and data symmetries together. The approach workflow is shown below.



First, a concurrent model is translated into an equivalent nondeterministic sequential model using existing approaches [3, 27]. The motivation behind is two-fold. First, it is nontrivial to analyze concurrent models whose behaviors are not obvious, such as subtle flexible communication patterns and numerous possible interleavings between processes. Second, we can take advantage of well-developed static analysis techniques for sequential models. Note that the idea of linking concurrent models to nondeterministic sequential models goes back to the work of Ashcroft and Manna [3] for proving the correctness of parallel programs. The translation has been also described in details in the book of Krzysztof and Olderog [27]. A sequential model can be built by simulating the behavior of the concurrent model and keeping track of local states of each process and global states all the time. The worst complexity of the translation is linear to the total number of atomic statements of all processes.

Second, we consider the problem of discovering symmetries from a new angle. Our key insight is recognizing the similarity between the role of symmetries in constraint programming and that in model checking. Our analysis transforms a sequential model into a constraint satisfaction problem, and extracts a graphical representation of the CSP called colored graph. Each automorphism of the colored graph is proved to correspond to one in the concurrent model, which is effectively discovered by applying Saucy [10]. The detected symmetries can be used later to speed up the performance of model checker.

The above steps can be performed fully automatically. The effectiveness and efficiency of our approach have been demonstrated via a variety of systems. Lastly, we extend the above approach to a parameterized system so that we can obtain symmetries for all its instances. For a parameterized system, we first use the above approach to detect symmetries over a sequence of its small instances, then generalize the detected symmetries to parameterized permutations, and validate whether each parameterized permutation is a real symmetry for all the instances.

2 Motivating Examples

In this section we introduce three example specifications. Each example highlights one particular scenario of symmetries, which is fairly hard or often impossible for current detection approaches to deal with. These examples also motivates us to develop an automatic symmetry detection approach, which handles arbitrary types of symmetries no matter where they come from, *i.e.*, homogenous processes, data values of the system, or together.

Example 1. 1. **Reader-Writer Problem** The first example focuses on partially symmetric systems. We borrow a variant of Reader-Writers problem from [39]. The problem consists of two reader processes (with ids 0 and 1) and one writer process with id 2. Each process may stay in one of three local states $\{N(\textit{the non-critical section}), T(\textit{the trying region}), C(\textit{the critical section})\}$ and its local transitions are the following. Assume s_i is the local state of process i,

- $s_i = N \rightarrow s_i = T$;
- $s_i = T \rightarrow s_i = C$, where either $s_0 \neq C \wedge s_1 \neq C \wedge s_2 \neq C$ or $i < 2 \wedge s_2 \neq C$;
- $s_i = C \rightarrow s_i = N$.

Every process may attempt to reach the critical section. If no process is currently in the critical section, any process can enter it. A reader process can also enter the critical section as long as the writer process is not in it. Thus these processes are quite similar but slightly different. The global behavior of the system is not totally but approximately symmetric. The automorphism group of this system arises from process symmetries of the two reader processes.

Existing approaches [24, 36, 37, 26, 25, 14, 15] are rather coarse grained and none of them supports statement-level detection. They either target at interfaces or communication structures [26, 25, 14, 15], or disallow such subtle statement-level difference that relies on the concrete process ids [24, 36, 37]. So they fail to detect the symmetries of this system.

Example 2. 2. **Message Routing in a Hypercube Network** The second example focuses on distributed systems with complicated communication patterns and arithmetic operations on process ids. We consider a system of message routing in a *hypercube* interconnection network in [13]. Hypercube is a popular implementation model of parallel computation applications. A d-dimensional hypercube is a special case of a d-dimensional $n_1 \times n_2 \times \cdots \times n_d$ array when $n_i = 2$ for $1 \leq i \leq d$. The hypercube has 2^d processors, each of which is directly connected to d other neighboring processors. The identifier of a processor node is a *binary* string $(x_1, x_2, \cdots, x_n)$. Two nodes are neighbors if and only if their identifiers differ in only one position. Figure 6 shows an example of a 3-dimensional hypercube network.

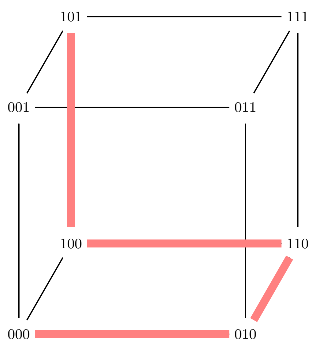


Fig. 1: A 3-dimensional hypercube

Algorithm 1 Static message routing algorithm

```
 1: For each node $u$:
 2: while $true$ do
 3:     receive a message whose destination node is $v$
 4:     if $u = v$ then
 5:         process message
 6:         choose a new destination node $t$
 7:         $v := t$
 8:     end if
 9:     send message to its neighbor $z$ whose id has one more bit in common with $v$
10: end while
```

This example models a parallel system where messages are routed through the hypercube architecture used in Algorithm 1. Upon receiving a message attached with the id x of the recipient, a node checks its own id with x. If they are the same, the node would process the message. Otherwise, the node would forward the message to its neighbor whose id differs from x only in one bit position. For example, a messages is sent from a node $(0, 0, 0)$ destined for a node $(1, 1, 1)$. The track of this message may be $(0, 0, 0) \to (0, 1, 0) \to (1, 1, 0) \to (1, 1, 1)$. The system is highly symmetric and has $2^d \times d!$ automorphisms.

Static channel diagram approach [13] is the only one which is able to automatically detect process symmetries from such a complex network topology. However, it has a substantial restriction dealing with the arithmetic and relational operations. It requires user to rewrite each arithmetic or relational operation by enumerating all the possible values of all variable involved and only using the logical disjunction of all satisfiable assignments. Take a 3-dimensional hypercube network as an example. The neighbor's id z at line 9 in Algorithm 1 is decided by the following expression where the current process id $u \in \{0, 1, \cdots, 7\}$

Algorithm 2 Nondeterministic 2-hop coloring with degree bound d

```
For each agent $u$:
State variables:
$color_u$ An integer recording the color of agent $u$, whose value is between 0 and $d \times (d-1)$.
$F_u$ A bit array whose size is $d \times (d-1) - 1$, indexed by colors.
The interaction between an initiator agent $u$ and a responder agent $v$:
1: if $F_u[color_v] \neq F_v[color_u]$ then
2:     assign an arbitrary color from $\{0..g-1\}$ to $color_u$
3:     $F_u[color_v] \leftarrow F_v[color_u]$
4: else
5:     $F_u[color_v] \leftarrow 1 - F_v[color_u]$
6:     $F_v[color_u] \leftarrow 1 - F_u[color_v]$
7: end if
```

, the destination process id $v \in \{0, 1, \cdots, 7\}$ and $u \neq v$ is satisfied.

$$\begin{array}{l} if(((u\ XOR\ v)\&(2^0)) = 2^0)\ \ \{z := (2^0) XOR\ u\} \\ elseif(((u\ XOR\ v)\&(2^1)) = 2^1)\ \ \{z := (2^1) XOR\ u\} \\ elseif(((u\ XOR\ v)\&(2^2)) = 2^2)\ \ \{z := (2^2) XOR\ u\} \end{array}$$

Then the expression must be rewritten to the following format:

$$\begin{array}{l} if(u = 0 \wedge v = 1)\{z := 1\} \\ else\ if(u = 0 \wedge v = 2)\{z := 2\} \\ else\ if(u = 0 \wedge v = 3)\{z := 1\} \\ else\ if(u = 0 \wedge v = 3)\{z := 2\} \\ \cdots \\ else\ if(u = 1 \wedge v = 0)\{z := 0\} \\ else\ if(u = 1 \wedge v = 2)\{z := 0\} \\ else\ if(u = 1 \wedge v = 2)\{z := 3\} \\ \cdots \\ else\ if(u = 7 \wedge v = 6)\{z := 6\} \end{array}$$

It will become cumbersome, impractical and more importantly, slow down the approach significantly when the sizes of variable domains are large.

Example 3. In the following, we use a non-deterministic 2-hop coloring protocol [2] as a running example. This protocol colors the agents deployed in a network of a degree-bounded graph such that no two agents adjacent to the same agent have the same color. Its goal is to enable each agent to distinguish between its neighbors. Each agent has one integer recording its color and one bit for each color. The transition rules applied during the interaction of two agents are described in Algorithm 2. Starting from an arbitrary configuration, the protocol guarantees to eventually reach a state in which any two neighbors of each agent have distinct colors after enough interactions.

The system exhibits both process and data symmetries. For ease of presentation, we only focus on undirected ring topology where $N \geq 3$. Three colors suffice for a ring of any size. Simple as the protocol is, it contains non-trivial symmetries: (a) process symmetries that rotate every process clockwise or counterclockwise; (b) data symmetries that swap any two colors; (c) another data symmetries that swap the bit values; (d) the combinations of process and data symmetries that permute processes, color values and bit values.

Existing data symmetry detection approaches [8, 24] rely on scalarset annotations. Although color values and bit values are fully symmetric respectively in this case, the arithmetic operations on the data values prohibit the use of scalarsets. Further, the protocol does not take asynchronous message-passing paradigm, so the approaches [14, 15, 26, 25] for detecting process symmetries are not applicable. Moreover, as far as we know, there is no approach that considers process and data symmetries which are not both full symmetries at the same time. In short, no existing approaches can find all symmetries in this example.

3 Preliminaries

This section is devoted to the background knowledge of symmetry reduction in model checking, the format of the sequential model into which we transform a concurrent model, and constraint satisfaction problems.

3.1 Model Checking with Symmetry Reduction

We present our work in the setting of Labeled Transition Systems (LTSs). An LTS is a tuple $\mathcal{L} = (S, init, \Sigma, \rightarrow)$ where S is a finite set of states, $init \in S$ is the initial state, Σ is a finite set of events and $\rightarrow$: $S \times \Sigma \times S$ is a labeled transition relation. A permutation σ is said to be an automorphism of an LTS $\mathcal{L}$ iff it preserves the transition relation and the initial state, *i.e.*, $(\forall s_1, s_2 \in S; e \in \Sigma.\ s_1 \xrightarrow{e} s_2 \Rightarrow \sigma(s_1) \xrightarrow{\sigma(e)} \sigma(s_2)) \wedge \sigma(init) = init$. A group G is an automorphism group of $\mathcal{L}$ iff every $\sigma \in G$ is an automorphism of $\mathcal{L}$. Given a set of propositions P and events E which constitute an SE-LTL formula ϕ, a permutation σ is said to be invariance iff the set of atomic state propositions and events is closed under the application of σ. Formally, σ satisfies the following condition:

$$(\forall s \in S; p \in P.\ s \vDash p \iff \sigma(s) \vDash p) \wedge (\forall e \in E.\ \sigma(e) = e)$$

where $s \vDash p$ denotes that p holds at state s. Intuitively, σ is invariance iff, after permutation, the truth of any proposition in P remains and event in E remains the same. A group T_ϕ is an automorphism group of an SE-LTL formula ϕ, $T = \{\pi : \pi(\phi) \equiv \phi\}$, where $\equiv$ denotes logical equivalence under all propositional interpretations [17]. G is an invariance group of $\mathcal{L}$ and ϕ iff every $\sigma \in G$ is an invariance of $\mathcal{L}$ and ϕ. Given a state $s \in S$, the orbit of s is the set $\theta(s) = \{t |\ \exists \sigma \in G.\ \sigma(s) = t\}$, *i.e.*, the equivalence group which contains s. From the orbit of state s, a unique representative state $rep(s)$ can be picked such that for all s and s' in the same orbit, $rep(s) = rep(s')$. Intuitively, if σ is an invariance of ϕ, states of the same orbit are behaviorally indistinguishable with respect to

ϕ. Based on this observation, an LTS $\mathcal{L}$ can be turned into a *quotient* LTS $\mathcal{L}_G$ where states in the same orbit are grouped together. Formally, a quotient LTS is defined as follows.

Definition 1. *Let $\mathcal{L} = (S, init, \Sigma, \rightarrow)$ be an LTS; G be an automorphism group of $\mathcal{L}$. Its quotient LTS $\mathcal{L}_G = (S_G, init_G, \Sigma, \rightarrow_G)$ is defined as follows:*

– *$S_G = \{rep(s) | s \in S\}$ is the set of representative states of orbits.*
– *$init_G = rep(init)$ is the initial representative state.*
– *$(r, e, r') \in \rightarrow_G$ iff there exists $r'' \in S$ such that $r \xrightarrow{e} r''$ and $rep(r'') = r'$.*

The following theorem [7] formalizes the idea that if G is an invariance group of $\mathcal{L}$ and ϕ, then $\mathcal{L}$ satisfies ϕ iff $\mathcal{L}_G$ satisfies ϕ [7]. It is proved by showing that the relation $(s, \theta(s))$ is a bi-simulation relation between $\mathcal{L}$ and $\mathcal{L}_G$.

Theorem 1. *Let $\mathcal{L} = (S, init, \Sigma, \rightarrow)$ be an LTS; ϕ be an LTL formula. If G be an invariance group of $\mathcal{L}$ and ϕ, then $\mathcal{L} \vDash \phi$ iff $\mathcal{L}_G \vDash \phi$.* $\square$

There are two common types of symmetries for improving the performance of model checking. A *process symmetry* is a permutation on identifiers of concurrent processes. A *data symmetry* is a permutation on data values. For example, suppose a state *st* is $(s_1, s_2, \cdots, s_n)$ where s_i is the local state valuation of process i. If σ is a process symmetry on the process ids $\{1, 2, \cdots, n\}$, then σ acts on *st* in the form $\sigma(st) = (s_{\sigma(1)}, s_{\sigma(2)}, \cdots, s_{\sigma(n)})$; if it is a data symmetry, then σ acts on *st* in the form $\sigma(st) = (\sigma(s_1), \sigma(s_2), \cdots, \sigma(s_n))$.

3.2 Linear Process Specification

In this work we transform a concurrent model into a sequential one in the form of linear process specification (LPS). An LPS [19] is a process algebra with data that describes a system as a set of guarded and nondeterministic transitions. It is composed of three parts: a series of type and function declarations, one *single sequential* process, so-called linear process equation (LPE) and its initial form. The detailed syntax of a linear process specification is summarized in the standard Backus-Naur form (BNF) as Figure 2 shows[4]. Let $\langle prod \rangle$ be a BNF production rule. We use the following shorthand notations to refer to occurrences of $\langle prod \rangle$ on the right hand side of other production rules.

– $\langle prod \rangle^?$ denotes an optional occurrence of $\langle prod \rangle$;
– $\langle prod \rangle^*$ denotes a sequence of zero or more occurrences of $\langle prod \rangle$;
– $\langle prod \rangle^+$ denotes a sequence of one or more occurrences of $\langle prod \rangle$;
– $\langle prod\text{-}list, '\diamond' \rangle$ denotes a $\diamond$-separated list of one or more occurrences of $\langle prod \rangle$.

[4] The syntax defined here is slightly different from the standard one [19]. We restrict the forms of type declarations and require that a function be defined in the imperative programming style instead of the functional programming style.

The language includes two elementary data types, *integer* and *boolean*, and allows users to declare a restricted subrange of integers as a new type, like *newtype* : 0..5. This kind of type is a subtype of integer. So integer constant enumeration can be defined via this type. For each type *elem* there is a corresponding array type *elem*[]. An array type can have an arbitrary number of dimensions. Note that in the $\langle expr \rangle$ production rule, $\circ$ denotes an arithmetic operator, such as $+, -, *, \div, \%$; in the $\langle guard \rangle$ production rule, $\bowtie$ denotes an operator taken from the set $\{=, \neq, <, \leq, >, \geq\}$.

Figure 3 shows a linear process specification of the 2-hop coloring protocol, where N denotes the number of nodes, and C denotes the number of colors. A linear process equation (LPE) is a parameterized recursive process definition. The left-hand side of an LPE is a process name with a vector of data parameters. Here we refer to these parameters as *global variables.* Addition operators in the right-hand side of the LPE 'sum' a list of nondeterministic transitions, to which we refer to as *summands.* A summand has a declaration of *local variables* followed by an *enabling* condition, an *action* function and a *next-state* function from left to right. Each local variable can be evaluated to any value of its type nondeterministically. It is *read-only* and cannot be of array type[5]. Executability of a summand is decided by its enabling condition that is a Boolean expression. The action of the summand is decided by the event name along with the data, which are determined by the action function. The effect of the summand is decided by its *next-state* function which updates the global variables. Each function is call-by-reference and can take in multiple parameters and return multiple values. Its definition is a declaration of *function variables* followed by a sequence of statements. A statement can be an assignment, conditional, or while-loop statement. For a function definition, we define the variables that will hold the values to be returned by the function (the second $\langle params \rangle$ in the production rule $\langle decl \rangle$). These variables must be evaluated before the end of the function body in order for the function to return values. Besides, there is an initial valuation of global variables denoted by *init*, which is the entry where the LPS starts to execute. The symbol $*$ denotes the nondeterministic choice of all possible evaluations of global variables.

An LPS is a symbolic representation of an LTS and has exactly one equivalent LTS [20]. each state is represented by the values of its global variables. At some state, if the enabling condition of a summand is true, then there exists a transition labeled with its event which is attached with parameters returned by its action function. This transition updates the global variables by invoking its next-state function at the same time. For a concurrent model, its corresponding LPS can be extracted in linear time as shown in Appendix A.

3.3 Constraint Satisfaction Problem

In the following, we introduce the terminology of constraint satisfaction problem used in the rest of this paper.

[5] If a local variable is an array, the language can be extended to support it easily.

$$
\begin{aligned}
\langle \mathit{program} \rangle &::= \langle \mathit{typespec} \rangle^* \langle \mathit{decl} \rangle^* \langle \mathit{lpe} \rangle \langle \mathit{init} \rangle \\
\langle \mathit{typespec} \rangle &::= \textbf{type } \langle \mathit{tid} \rangle : \langle \mathit{num} \rangle .. \langle \mathit{num} \rangle \\
\langle \mathit{decl} \rangle &::= \textbf{fun } \langle \mathit{fid} \rangle (\langle \mathit{params} \rangle) \langle \mathit{params} \rangle \ \{\langle \mathit{body} \rangle\} \\
\langle \mathit{lpe} \rangle &::= \textbf{proc } P(\langle \mathit{params} \rangle) = \langle \mathit{summand\text{-}list}, '+' \rangle \\
\langle \mathit{init} \rangle &::= \textbf{init } P((\langle \mathit{args} \rangle)) \\
\langle \mathit{summand} \rangle &::= \langle \mathit{params} \rangle^? . [\langle \mathit{guard} \rangle] \\
& \ \langle \mathit{eid} \rangle (\langle \mathit{fid} \rangle (\langle \mathit{args} \rangle) . P(\langle \mathit{fid} \rangle (\langle \mathit{args} \rangle))) \\
\langle \mathit{body} \rangle &::= \langle \mathit{vardecl} \rangle^* \langle \mathit{stmt} \rangle^* \\
\langle \mathit{vardecl} \rangle &::= \langle \mathit{param} \rangle = \langle \mathit{num} \rangle | \langle \mathit{bl} \rangle | \langle \mathit{vid} \rangle; \\
\langle \mathit{stmt} \rangle &::= \langle \mathit{varassgn} \rangle := \langle \mathit{expr} \rangle; \\
& \ |\textbf{if } (\langle \mathit{guard} \rangle)\{\langle \mathit{stmt} \rangle^*\} \textbf{ else } \{\langle \mathit{stmt} \rangle^*\} \\
& \ |\textbf{while } (\langle \mathit{guard} \rangle)\{\langle \mathit{stmt} \rangle^*\} \\
\langle \mathit{varassgn} \rangle &::= \langle \mathit{vid} \rangle \\
& \ |\langle \mathit{vid} \rangle [\mathit{expr}]^+ \\
\langle \mathit{expr} \rangle &::= \langle \mathit{num} \rangle \\
& \ |\langle \mathit{bl} \rangle \\
& \ |\langle \mathit{vid} \rangle \\
& \ |\langle \mathit{vid} \rangle [\mathit{expr}]^+ \\
& \ |\langle \mathit{guard} \rangle \\
& \ |\langle \mathit{expr} \rangle \circ \langle \mathit{expr} \rangle \\
\langle \mathit{guard} \rangle &::= !\langle \mathit{expr} \rangle \\
& \ |\langle \mathit{expr} \rangle \bowtie \langle \mathit{expr} \rangle \\
& \ |\langle \mathit{guard} \rangle \&\& \langle \mathit{guard} \rangle \\
& \ |\langle \mathit{guard} \rangle \ || \ \langle \mathit{guard} \rangle \\
& \ |\forall \langle \mathit{vid} \rangle : \langle \mathit{tid} \rangle . \langle \mathit{guard} \rangle \\
\langle \mathit{params} \rangle &::= \langle \mathit{param\text{-}list}, ',' \rangle \\
\langle \mathit{args} \rangle &::= \langle \mathit{arg\text{-}list}, ',' \rangle \\
\langle \mathit{param} \rangle &::= \langle \mathit{tid} \rangle [\langle \mathit{num} \rangle]^* \ \langle \mathit{vid} \rangle \\
& \ |\mathit{int}[\langle \mathit{num} \rangle]^* \ \langle \mathit{vid} \rangle \\
& \ |\mathit{bool}[\langle \mathit{num} \rangle]^* \ \langle \mathit{vid} \rangle \\
\langle \mathit{arg} \rangle &::= \langle \mathit{vid} \rangle | \langle \mathit{num} \rangle | \langle \mathit{bl} \rangle | \{\langle \mathit{num} \rangle \mathit{\text{-}list}, ',']\} | \{\langle \mathit{bl} \rangle \mathit{\text{-}list}, ',']\} \\
\langle \mathit{fid} \rangle, \langle \mathit{eid} \rangle, \langle \mathit{vid} \rangle, \langle \mathit{tid} \rangle &::= \mathit{string} \\
\langle \mathit{num} \rangle &::= \mathit{an\ integer} \\
\langle \mathit{bl} \rangle &::= \mathit{a\ bool}
\end{aligned}
$$

Fig. 2: Syntax of linear process specification

```
type NS : 0..N − 1
type CS : 0..2
type BITS : 0..1
fun upd₁(NS u, NS v, CS c, CS[N] color, BITS[N][3] F) CS[N] color′, BITS[N][3] F′
    {color[u] := c; F[u][color[v]] := F[v][color[u]]; color′ := color; F′ := F; }
fun upd₂(NS u, NS v, CS[N] color, BITS[N][3] F) CS[N] color′, BITS[N][3] F′
    {F[u][color[v]] := 1 − F[u][color[v]]; F[v][color[u]] := 1 − F[v][color[u]]; color′ := color; F′ := F; }
proc Interaction(CS[N] color, BITS[N][3] F) =
NS u₁.NS v₁.CS c.[(v₁ = (u₁ − 1)mod N ∨ v₁ = (u₁ + 1)mod N) ∧ F[u₁][color[v₁]] ≠ F[v₁][color[u₁]]]
                    .Interaction(upd₁(u₁, v₁, c, color, F)) +
NS u₂.NS v₂.[(v₂ = (u₂ − 1)mod N ∨ v₂ = (u₂ + 1)mod N) ∧ F[u₂][color[v₂]] = F[v₂][color[u₂]]]
                .Interaction(upd₂(u₂, v₂, c, F))
init Interaction(∗);
```

Fig. 3: Linear Process Specification of the 2-hop Coloring Protocol

Many CSPs naturally exhibit symmetries which induce a number of equivalent solutions. A number of approaches on detecting and breaking symmetries of solutions have been proposed. The last decade has witnessed a revolution of these approaches in speeding up the search for large practical CSPs involving tens of thousands of variables and constraints [30–32, 38, 28, 35]. Besides, constraint programming and model checking share much similarity as investigated in [11, 12]. This presents a real opportunity to leverage these advances for detecting symmetries in model checking.

A constraint satisfaction problem (CSP) is a triple (V, D, C) where V is a finite set of *variables*, D is a set of finite *domains* and C is a finite set of *constraints*. Each variable $v_i \in V$ has an associated domain $D_i \in D$ of possible values. A *literal* is a statement of the form $v_i = d$ where $v_i \in V$ and $d \in D_i$. For any literal l of the form $v_i = d$, we use $var(l)$ to denote its variable v_i. The set of all literals is denoted by χ. An *assignment* is a set of literals, each of which is a variable valuation of the CSP. A *solution* of a CSP is a complete assignment which satisfies each constraint in C. A constraint c is defined over a set of variables, and the set is denoted as $Var(c)$.

A *solution symmetry* is a permutation of literals that preserves the set of solutions. A *constraint symmetry* is a solution symmetry that preserves the constraints of the CSP. For a CSP $P = (V, D, C)$, a *variable symmetry* σ is a permutation on V such that for any constraint $c \in C$, $\{v_1{=}a_1, \cdots, v_n{=}a_n\}$ satisfies c iff $\{\sigma(v_1){=}a_1, \cdots, \sigma(v_n){=}a_n\}$ satisfies c; a *value symmetry* σ is a permutation on D such that for any constraint $c \in C$, $\{v_1{=}a_1, \cdots, v_n{=}a_n\}$ satisfies c iff $\{v_1{=}\sigma(a_1), \cdots, v_n{=}\sigma(a_n)\}$ satisfies c. A *variable-value* symmetry is a permutation of the literals (*i.e.*, $V \times D$) that is a constraint symmetry. Note that a variable-value symmetry is not necessarily a composition of a variable symmetry and a value symmetry.

Algorithm 3 Overview of our approach

```
$autos := \emptyset; VL := \emptyset; csps := \emptyset;$
identify the set of global variables $VG$
for each summand $sum$ in an $LPE$ do
    identify the set of local variables $locals$
    $VL := VL \cup locals;$
    for each function or enabling condition $f$ in $sum$ do
        $csps := csps \cup \{\texttt{Transform}(f)\};$
    end for
end for
$csps := csps \cup \{(\texttt{Transform}(init))\};$
$csp := Merge(csps);$
$autos :=$ DetectSymmetry $(csp, VG, VL);$
```

4 Automatic Symmetry Detection

In the section, we describe an automatic approach to detecting the symmetries of an LPS. It translates an LPS into a constraint system whose symmetries can be exploited using the state-of-the-art detection approaches for CSPs. There are two main steps. The first step, *conversion*, transforms each function in an LPE and its *init* statement to a semantics-equivalent CSP as shown by Procedure $Transform$. These CSPs are then merged into one single CSP. The second step, *detection*, detects variable and value symmetries in the merged CSP, as described in Procedure *DetectSymmetry*. Further, we prove that each detected symmetry is a real automorphism of the LTS of the original concurrent model. Lastly, we present two lightweight but effective optimization methods.

4.1 Step 1: Conversion

We describe how to convert a function or the *init* statement into the static single assignment form (SSA) below, from which an equivalent CSP is derived. SSA is a form of a semantics-preserving intermediate representation of a program, which requires that each variable be assigned exactly once [9]. SSA significantly simplifies and improves various compiler optimizations, *e.g.*, constant propagation, copy propagation, dead code elimination and register allocation. The key feature of SSA is that each variable with the same name always has the same value in everywhere in the program. The immutability of variables is the primary reason why we transform each function into a constraint system by the use of SSA.

Converting ordinary source code into SSA is relatively straightforward. In essence, it replaces the target variable of each assignment with a fresh name. Every usage of this variable in the succeeding statements is replaced with the new name, until a new assignment to the same variable occurs. We call the existing variables *original* variables, and other new variables *versioned* variables.

Further, SSA defines an artificial function ϕ to represent the choice between different branches of a conditional statement defined formally as follows. A new Boolean variable b, called *decision variable*, is introduced to store the value of the condition and the *if* and *else* branches are converted separately. For each variable x defined in the *if* or *else* branch, an additional assignment $x''' :=$

$\phi(x', x'', b)$ is inserted at the end of the block to achieve branch selection, where x' and x'' are the last definitions of x in the *if* and *else* branches respectively.

$$\phi(x', x'', b) = \text{if } b = true \text{ then } x' \text{ else } x''$$

Still, converting a program to SSA form becomes more complicated when *while*-loop statements are involved. A *while*-loop can be equivalently regarded as an infinite number of nested conditional statements. But it is impractical to transform it into such conditional statements. So the assumption here is that any loop can be finished in a finite number of iterations. In this way, we reduce the problem of converting a loop to converting a list of conditional statements.

Another challenge is handling array manipulation. The reason is that a new assignment statement of an array does not necessarily kill all the old values in the array. For instance, the meaning of the assignment $A[i] := A[i] + 5$ is two-fold. First, it increases the value of the i^{th} element in the array A by 5. Second, all the values of other elements are unchanged. We can not simply assign the left-hand side with a new name, which loses the second meaning. Thus we define a function φ as follows to handle array assignments. Suppose an array assignment is $array[index] := value$ and $array_0$ is the latest name of $array$ before the assignment in the SSA form. We replace the original assignment with $array_1 := \varphi(array_1, array_0, index, value)$ where $array_1$ is a fresh name. Note that φ can be a polymorphic function so as to handle multi-dimensional arrays.

$$array_1 := \varphi(array_1, array_0, index, value) = \begin{cases} array_1[index] = value \wedge \\ \forall j \neq index.\ array_1[j] = array_0[j] \end{cases}$$

The last challenge is handling function calls. Given a function F with formal parameters $x_1, x_2, \cdots, x_n$ and it is called with a list of arguments $a_1, a_2, \cdots, a_n$. Assignments $x_i := a_i$ are added before the function body to represent parameter assignments. Return values of a function are handled similarly. Classic SSAs do not handle shared variables. A shared variable may be used in multiple functions, so renaming potentially breaks the dependency among functions caused by it. For an LPS, local variables and global variables can be shared by multiple functions. Because local variables are read only, it is unnecessary to consider their side effects of function calls. For global variables, we first treat them like function variables and then separately take care of the data flow across function boundaries. Considering that the effect of other versioned variables is only in the scope of a function, we only consider the original and last versioned of a global variable as global variables, and other versioned variables as function variables.

The SSA form we obtain can be more succinct by applying copy propagation technique, commonly used in compiler optimization. It eliminates unnecessary temporary copies of a value generated by our transformation, and further facilitates our symmetry detection approach. An assignment is an *identity assignment* if it is in the form $x := y$ which assigns the value of y to x and y is either a variable or a constant. Copy propagation is the process of replacing the occurrences of targets of identity assignments with their values.

The SSA form of a program always has the same behavior as the original program [9]. After the conversion of a function to SSA, the next conversion

$$
\begin{aligned}
V &= \{u_1, v_1, u_2, v_2, c, color[N], color_1[N], F[N][3], F_1[N][3], F_2[N][3]\} \\
D &= \{\{0..N-1\}, \{0..N-1\}, \{0..N-1\}, \{0..N-1\}, \{0..2\}, \{0..2\}, \{0..2\}, \{0..1\}, \{0..1\}, \{0..1\}\} \\
C &= \begin{cases}
(v_1 = (u_1 - 1) mod\ N \vee v_1 = (u_1 + 1) mod\ N) \\
\wedge F[u_1][color[v_1]] \neq F[v_1][color[u_1]] \\
color_1[u_1] = c \wedge (\forall t \in NS.t \neq u_1 \rightarrow color_1[t] = color[t]) \\
F_2[u_1][color_1[v_1]] = F[v_1][color[u_1]] \\
\wedge(\forall t_1 \in NS.t_2 \in CS.t_1 \neq u_1 \wedge t_2 \neq color_1[v_1] \rightarrow F_2[t_1][t_2] = F[t_1][t_2]) \\
(v_2 \neq (u_2 - 1) mod\ N \wedge v_2 \neq (u_2 + 1) mod\ N) \\
\wedge F[u_2][color[v_2]] = F[v_2][color[u_2]] \\
F_1[u_2][color[v_2]] = c \\
\wedge(\forall t_1 \in NS.t_2 \in CS.t_1 \neq u_2 \wedge t_2 \neq color[v_2] \rightarrow F_1[t_1][t_2] = F[t_1][t_2]) \\
F_2[v_2][color[u_2]] = 1 - F_1[v_2][color[u_2]] \\
\wedge(\forall t_1 \in NS.t_2 \in CS.t_1 \neq v_2 \wedge t_2 \neq color[u_2] \rightarrow F_2[t_1][t_2] = F_1[t_1][t_2]) \\
\forall t \in NS.color_1[t] = color[t]
\end{cases}
\end{aligned}
$$

Fig. 4: Constraint satisfaction problem of the 2-hop coloring protocol

from SSA to a CSP is straightforward. Each assignment is directly mapped to a constraint by interpreting each assignment operator as an equivalence operator. Both representations are very similar. It is easy to know the SSA and its CSP representation have equivalent behaviors as the following proposition states.

Proposition 1. *Given a program P and its SSA representation $\mathcal{P}$, let $\mathcal{C}_{\mathcal{P}}$ be the CSP converted from $\mathcal{P}$. If for an input I the execution of $\mathcal{P}$ produces valuations V for all variables, then I and V is a solution of $\mathcal{C}_{\mathcal{P}}$ and vice versa.*

For the *init* statement, we convert it into a constraint in a very similar way. Suppose an LPE is $P(Dom_1\ v_1, \cdots, Dom_n\ v_n)$ and its *init* statement is $P(a_1, \cdots, a_n)$. It is converted to $v_1 = a_1 \wedge \cdots \wedge v_n = a_n$. Then we simply combine all the constraints derived from each function, enabling condition and the *init* statement to build one large CSP for this LPS.

For the running example, the conversion step builds the corresponding CSP for the LPS as shown in Figure 4. Since its *init* statement represents all possible evaluations of global variables, it has no effect on symmetry breaking in the CSP and thus is skipped for simplicity.

4.2 Step 2: Symmetry Detection

Next, we explain the procedure to discover constraint symmetries in the merged CSP which we denote as $\mathcal{C}_{\mathcal{F}}$ in the following. First, we present the state-of-the-art symmetry detection method for CSP, on which our detection approach is based. However, considering the role each constraint plays in the sequential model, this method is not completely suitable in terms of correctness and performance. To cope with this problem, we describe our alternations on this method.

Our approach is based on the automatic symmetry detection method for CSP proposed by Puget [35]. It allows us to detect variable symmetries, value symmetries and non-trivial ones involving both variables and values. For each constraint, the approach first calculates all the allowed assignments. Then the graph of this constraint c is constructed in the following way. A *variable* node is created for each variable in c. An array represents a collection of scalar variables.

So a distinct variable node is created for each element of the array. A *constraint* node is created for c. A *value* node is created for each value of each variable in c. An *assignment* node is created for each allowed assignment of c. Edges connect each value node to its variable node, each assignment node to the value node representing each variable-value literal occurring in the assignment, and each assignment node to the constraint node. So the number of nodes in the colored graph is the sum of the number of variables, literals, constraints and allowed assignments, and the number of edges is the sum of the number of literals, allowed assignments and the number of variables in allowed assignments.

The graphs for all constraints are combined into a single graph, called *colored graph*. The coloring scheme for this graph is described as follows.

- All constraint nodes representing the same kind of constraints have the same unique color;
- All variable nodes with the same domain have the same unique color;
- For a variable, all of its value nodes have the same unique color; if two variables have the same color, their value nodes have the same color.
- For a constraint, its assignment nodes all have the same unique color; if two constraints have the same color, their assignment nodes have the same color.

It addresses symmetries by computing the automorphisms of the colored graph. It has been proved that each automorphism of this graph corresponds to a constraint symmetry as restated in the following theorem.

Theorem 2. *Let $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ be a CSP. Its colored graph $\mathcal{G}$ is constructed as illustrated above. Suppose σ is an automorphism of $\mathcal{G}$ and s is an assignment of $\mathcal{P}$. For each constraint $c \in C$, s satisfies c iff $\sigma(s)$ satisfies c.* $\square$

Before applying this method to our problem, we have to address the concern raised by the differences of ordinary CSPs and the CSP we convert an LPS into. Some variables in an LPS are not used at the same time, local variables in different summands for example. So for its corresponding CSP, it is unreasonable to detect variable symmetries between those variables. Therefore, the original coloring strategy is refined such that variable nodes which have the same domain are of the same unique color iff

- each of them is a local variable of the same domain in the same summand,
- or each of them is an original global variable of the same domain,
- or each of them is the latest version of a global variable of the same domain.

It is not difficult to show that each automorphism found under the new coloring strategy is also an automorphism under the original coloring strategy. So Theorem 2 still holds. The soundness of our work is stated in the theorem below with the proof presented in Appendix B.

Theorem 3. *Let $\mathcal{L} = (S, init, \Sigma, \rightarrow)$ be its labeled transition system of an LPS $\mathcal{P}$. Each permutation σ we get in Algorithm 3 is an automorphism of $\mathcal{L}$.* $\square$

Figure 5 shows a part of the colored graph obtained from the CSP of the running example. Due to space restriction and graph complexity, we make the following alternations for simplicity in order to help users better understand its inherent symmetries while still preserving the essence of the graph. This graph fragment shown is built from the first constraint in the CSP, *i.e.*, $(v_1 = (u_1+1) mod\ N \vee v_1 = (u_1-1) mod\ N) \wedge F[C*u_1+color[v_1]] \neq F[C*v_1+color[u_1]]$. We skip the representation of

- all nodes generated from $(v_1 = (u_1+1) mod\ N \vee v_1 = (u_1-1) mod\ N)$
- the constraint node
- all nodes generated from an allowed assignment containing a literal $color[t] := 2$ for all $0 <= t < N$.

Each blue square node labeled i in the top part is the variable node representing $F[i]$; each purple triangle node labeled j in the bottom part is the variable node representing $color[j]$; each yellow node labeled val is a value node representing value val of variable $F[i]$; each green circle node labeled val' is a value node representing value val' of variable $color[i]$; each white pentagon node is an assignment node. The nodes in the dotted rectangle are the same nodes of variables $F[0], F[1]$ and $color[0]$ and their values, which are only for making the symmetries easy to discover. In this figure, the assignment nodes connected with dotted

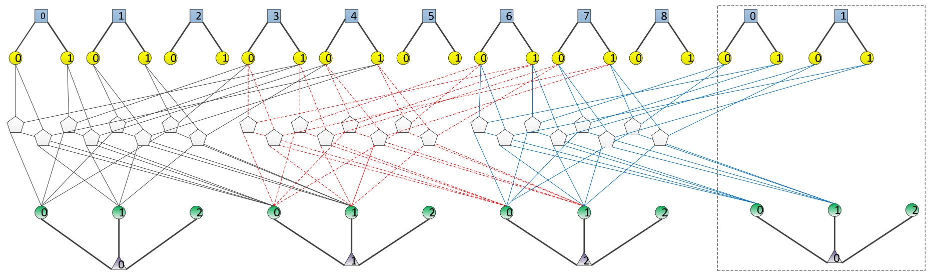


Fig. 5: Part of the colored graph of the running example's CSP

edges, the assignment nodes on their left and the assignment nodes on their left are isomorphic; swapping any literals of the form $F[i] := 0$ and $F[i] := 1$ for all $0 <= i < 3 \times N$ in all the assignments gets the same graph; swapping any literals of the form $c[i] := 0$ and $c[i] := 1$ for all $0 <= i < N$ in all the assignments gets the same graph.

Example 4. For the running example, assume there are three processes with ids 0, 1 and 2, it has 6 process symmetries from rotating the processes of an undirected ring, *i.e.*, $(0)(1)(2), (1,2), (1,0), (0,2), (0,1)(1,2), (0,2)(1,2)$[6]; it has 6

[6] Permutations are written in the cyclic notation. If $a_1, a_2, \cdots, a_n$ are distinct elements of Ω, then the cycle $(a_1, a_2, \cdots, a_n)$ denotes the permutation σ on Ω, *i.e.*, for $1 \leq i < n$, $\sigma(a_i) = a_{i+1}$, $\sigma(a_n) = a_1$ and for any $b \in \Omega \setminus \{a_1, a_2, \cdots, a_n\}$, $\sigma(b) = b$.

data symmetries from swapping all the possible colors, *i.e.*, $(0)(1)(2), (1,2), (1,0), (0,2)$, $(0,1)(1,2)$, $(0,2)(1,2)$; it has 2 data symmetries from swapping all the possible bit values, *i.e.*, $(0)(1), (0,1)$. Further, new symmetries are introduced by the product of these groups. Therefore, we discover 72 symmetries in total.

4.3 Optimization

In the step of symmetry detection, we perform two lightweight but effective optimization techniques, one to speed up the construction of the colored graph and the other to remove symmetries which are useless for model checking.

Breaking down array writing constraints For a constraint with n variables, it may have $O(m^n)$ possible assignments in the worst case, where m is the size of the largest domain. The time complexity of computing allowed assignments of one constraint is $O(m^n)$, and the time complexity of constructing the colored graph for a CSP accumulates to $t \times O(m^n)$ where t is in the number of constraints. Each array writing constraint is involved with at least all the variables of two arrays, which often becomes a performance bottleneck. In order to reduce the time consumption, one straightforward way is keeping n as small as possible. We transform it into $K+1$ simple constraints each involving much fewer variables in the following way[7] where K is the array size, and refine the coloring strategy such that elements of different arrays have different colors.

$$\begin{gathered} array_1[index] = value \wedge (\forall j \in \{0, \cdots, N-1\}. j \neq index \rightarrow array_1[j] = array_0[j]) \\ \Downarrow \\ array_1[index] = value \\ array_1[0] = array_0[0] \\ array_1[1] = array_0[1] \\ \cdots \\ array_1[N-1] = array_0[N-1] \end{gathered}$$

The soundness is established with a theorem, which is presented in Appendix C.

Removing redundant value symmetries The colored graph may contain some values of a variable which do not satisfy any constraint transformed from an enabling condition or the *init* statement. It means those values are impossible to appear at any time during the execution of the system. Take the CSP ($V = \{x, y\}, D = \{\{0,1,2\}, \{2,3,4\}\}, C = \{x > 1, y = x + 1\}$) as an example. A value symmetry $\sigma = (x := 0, x := 1)$ exists in the CSP. Suppose the constraint $x > 1$ is originally the enabling condition and $y = x + 1$ is the next-state function of the same summand in the LPS. So neither $x := 0$ nor $x := 1$ is valid in any state which makes σ useless for reducing the state space. Therefore, it is safe and appropriate to remove these values during the graph construction in order to avoid redundant symmetries later. For each variable's value, we record whether it appears in at least one allowed assignment of a constraint representing an enabling condition or the *init* statement. If not, it will be removed.

[7] For ease of presentation, we only show how to transform a writing constraint of a one-dimensional array. It can be easily extended to multi-dimensional arrays.

5 Case Studies

We have implemented the colored graph construction described in Section 4. The resulting graph is input to Saucy [10] which produces the generators of the automorphism group of a graph. Then the generators are input to GAP [21] which produces all the permutations in the group. The examples are briefly introduced in the following. Part of the experiment data is presented in Table 1. All relevant experiment information is available online [1].

1. Reader-Writer problem. [39].
2. Peterson's mutual exclusion protocol [34]. The N-process protocol manipulates shared arrays in such a way that it eliminates at least one process trying to access the critical section per round in a total of $N-1$ rounds until only one remains, so that it guarantees that no more than one processes are in the critical section at the same time.
3. A prioritized resource allocator [13]. The system consists of N client processes and one resource allocator process. It has a star topology and all the clients only communicate with the resource allocator. Each client has a priority level and may send requests for accessing the resource to the resource allocator. When the resource allocator receives multiple requests, it always grants access to the request from the client with the highest priority. If all the requesting clients have the same priority level, the allocator chooses one request in a nondeterministic way. A configuration is written in the form $a_0 - a_1 - \cdots - a_{k-1}$, where client processes $0, 1, \cdots, a_0$ have priority level 0, $a_0 + 1, a_0 + 2, \cdots, a_1$ have priority level 1, *etc.*
4. Message routing in a hypercube network [13]. A configuration is denoted by the number of dimensions of the hypercube. Note that the configuration d is composed of 2^d processes.
5. Server-client system in a three-tiered architecture [13]. As its name implies, the systems has three layers, one for client processes, one for server processes and one for the process representing the data storage system. Each process has two channels, one for receiving incoming requests and the other for sending queries to other processes in the neighboring layer. A client repeats the sequence of operations of sending a request message to the server it is connected to and waiting for receiving a response message from the server; a server repeats the sequence of operations of receiving a request from a client it is connected to, sending a query to the database, receiving data and sending a response message to the client; the database repeats the sequence of operations of receiving a query from a server and sending back data to the server. for each experiment of the system is written in the form $a_1 - a_2 - \cdots - a_k$, which denotes that the system consists of k server processes and a_i clients connected to server i.
6. Dining Philosopher Problem [23]. The N philosophers sit at a circular table with a large bowl of spaghetti in the center. A fork is placed in between each philosopher, and as such, each philosopher has one fork to his left and one fork to his right. As spaghetti is difficult to serve and eat with a single fork,

it is assumed that a philosopher must eat with two forks. The philosopher can only use the fork on his or her immediate left or right and he can only put down the forks after eating.

7. Miler's scheduler [33]. There are N processes, which are activated in a cyclic manner, *i.e.*, process i activates process $i + 1$ and after process n process 1 is activated again. Moreover, a process may never be re-activated before it has terminated. The scheduler is built from n cyclers who are positioned in a ring. The first cycler plays a special role as it starts up the system.
8. Non-deterministic two-hop coloring protocol in rings [2].
9. Self-stabilizing leader election protocol in complete graph [18]. Each agent has one bit of memory, denoting being the leader or not. There is a leader detector in the network to signal the presence of a leader and to broadcast a boolean value corresponding to the signal to each agent. The detector is not guaranteed to give correct answers all the time, but it will eventually give a correct answer permanently. This protocol guarantees that a unique leader will eventually be elected.
10. Self-stabilizing leader election protocol in directed rooted trees [6].
11. Self-stabilizing leader election protocol in ring [18].
12. Hanoi puzzle.
    The tower of hanoi is a classic mathematical puzzle. It consists of three pegs and a number of disks of different sizes can be put onto a peg. You may move the top disk from one peg to the top of another peg at a time. At no time can a larger disk be put on top of a smaller disk. Initially, all disks are stacked at one rod(called the initial rod) in order from the largest at the bottom to the smallest at the top. The other two rods are empty. The goal is to find the minimum number of moves to move all the disks to another rod.
13. Scheduling the social golfer problem [16].
    The social golfer problem was first posted on sci.op-research in May 1998. It is a famous combinational optimization problem. The task is to schedule $N = G \times P$ golfers into G groups of P players in W weeks, such that no two golfers play in the same group more than once. Here we consider a heuristic tabu-search scheduling algorithm for this problem proposed in [16]. It consider a special case of the problem where G is a prime, $S = G$ and $W = G + 1$. For week 0, the schedule is generated randomly such that each position has one distinct golfer. For any following week w, any group i, any position j, the algorithm chooses the golfer in week 0 whose position is $(i + (w - 1) \times j)$ *mod* S in group j. A configuration of the system is written in the form *G-S-W* where G is the number of groups, S is the number of golfers in one group and W is the number of weeks.

The experimental cases we choose here cover a variety of computing systems. From the perspective of execution patterns, they include concurrent systems and sequential systems. From the perspective of communication strategies, they include concurrent systems with synchronized communication using shared variables or shared actions, and distributed systems with asynchronous message passing mechanism. From the perspective of communication topologies, they include networks of layers, rings, trees, stars, complete graphs and hypercubes. From the perspective of symmetry types, there are systems with only process symmetries, with only data symmetries and with both of them.

In Table 1, *Construction Time* denotes the time taken to construct the colored graph associated with the corresponding configuration; $|generators|$ denotes the number of generators of the automorphism group G of the colored graph computed by Saucy; *Saucy Time* denotes the time (in seconds) taken by saucy to compute generators; $|Aut(G)|$ denotes the size of G computed by GAP provided the generators. As Table 1 shows, the overhead of our approach is quite low even for the systems with large automorphism groups. We study the same cases as the static channel diagram approach [14, 15] (*i.e.*, Peterson's protocol, resource allocator, three-tiered architecture and message passing in a hypercube network) and our performance is comparable to its. However, the effectiveness of our approach is not limited to message passing systems or process symmetries.

Performance Improvement The performance bottleneck of our approach lies in the size of the colored graph. First, allowed assignments for constraints often contribute the largest portion of the graph size. For a constraint with n variables, as discussed in Section 4.3, in order to reduce its time consumption, one straightforward way is keeping n as small as possible. So we break down a constraint into a set of sub-constraints and guarantee that the logical conjunction of sub-constraints is equivalent to the original constraint. This method has a side effect: it increases the number of constraints. Fortunately, this effect is negligible because the time consumption for computing allowed assignments is much more sensitive to the number of variables in a constraint than to the number of constraints, and the performance bottleneck is its time consumption instead of its memory. Second, we have observed that users may sometimes define larger variable domains than necessary. Our approach does not rely on the exact domain of variables, but can take advantage of it to construct a smaller colored graph.

6 Symmetry Detection for Parameterized Systems

Concurrent and distributed systems are often parameterized systems, *e.g.*, a network protocol is designed for a network containing n nodes where n is a parameter. To the best of our knowledge, all existing symmetry detection approaches only work on an instance of a parameterized system at a time. For parameterized systems, the ultimate goal is to provide a once-for-all solution of obtaining common symmetries for the entire class. This goal is perhaps feasible because the distinctive features of symmetries in a parameterized system are often determined by the essence of the system structure rather than valuations of the parameters. In the following, we propose an inductive symmetry detection approach for parameterized systems.

A parameterized system *model* represents a series of initializations of finite state systems and is defined in terms of a series of parameters. We write $\mathcal{M}(\mathcal{D})$

System	Construction Time	Saucy Time	|Generators|	|Aut(G)|
Reader-writer problem				
3	0.127	0.004	1	2
Peterson's mutual exclusion protocol				
3	0.183	0.041	2	6
6	0.512	0.011	5	720
9	0.695	0.018	8	362880
12	1.037	0.030	11	479001600
A prioritized resource allocator				
4-3	0.589	0.002	5	144
2-2-3	0.553	0.004	4	24
3-3-4	0.902	0.005	7	864
Three-tiered architecture				
3-3-2	0.480	0.005	5	144
3-3-3	0.515	0.006	6	1296
4-4-3	0.508	0.006	8	6912
Message passing in a hypercube network				
3	0.343	0.007	4	48
4	0.655	0.005	5	384
5	1.447	0.026	4	3840
6	3.317	0.066	5	46080
Dining philosophers				
5	0.316	0.004	1	5
10	0.492	0.005	1	10
20	1.033	0.007	1	20
Miler's scheduler				
10	2.665	0.001	0	0
Non-deterministic two-hop coloring protocol in undirected rings				
3	0.294	0.007	4	72
6	0.466	0.011	5	144
9	0.788	0.012	5	216
12	1.282	0.013	5	288
Self-stabilizing leader election protocol in complete graphs				
3	0.179	0.004	2	6
6	0.874	0.029	5	720
9	1.212	0.034	8	362880
12	2.684	0.394	11	479001600
15	15.783	8.326	14	1307674368000
Self-stabilizing leader election protocol in directed rooted trees				
3	0.216	0.004	2	2
7	0.322	0.007	2	8
11	0.959	0.011	4	16
15	3.954	0.275	4	16
19	7.404	0.005	6	128
Self-stabilizing leader election protocol in rings				
3	0.385	0.003	1	3
6	1.223	0.007	1	6
9	4.781	0.093	1	9
12	51.265	1.266	1	12
Hanoi puzzle				
3	0.523	0.003	1	2
6	1.636	0.023	1	2
Scheduling the social golfer problem				
3-3-4	1.374	0.009	9	725760

Table 1: Experiment results

to denote a parameterized system $\mathcal{M}$ with the parameters $\mathcal{D}$. Its *instance* is obtained by instantiating the system parameters. We write $\mathcal{M}(d)$ to denote a particular instance of M where $d \in D$. $\sigma[\mathcal{D}]$ is a *parameterized permutation* of $\mathcal{M}(\mathcal{D})$ iff $\forall d \in \mathcal{D}$, $\sigma[d]$ is a permutation of $\mathcal{M}(d)$. $\sigma[\mathcal{D}]$ is a *parameterized symmetry* of $\mathcal{M}(\mathcal{D})$ iff $\forall d \in \mathcal{D}$, $\sigma[d]$ is an automorphism of $\mathcal{M}(d)$. Our detection approach is described as performing the following three steps: (1) detect symmetries on a set of small instances of the parameterized system model; (2) generalize the detected symmetries to parameterized permutations; (3) check whether each parameterized permutation obtained in Step 2 is a parameterized symmetry.

6.1 Step One: Detecting Symmetries on a Set of Instances

We use the technique presented in Section 4 to detect instance symmetries. For a parameterized system model $\mathcal{M}(\mathcal{D})$, the parameters $\mathcal{D}$ are assumed to be a tuple of integers, $(v_1, v_2, \cdots, v_n)$. We generate the instances by increasing the value of a parameter at one time, starting from the smallest one. For the running example, it only has one parameter N to control the number of processes. We can start from 3 and then increment N to 4, 5 and 6 and get four instances $\mathcal{M}(3)$, $\mathcal{M}(4)$, $\mathcal{M}(5)$ and $\mathcal{M}(6)$.

Step Two: Generalizing Instance symmetries to Parameterized Permutations A system always has a group of permutations which generally contains a large number of elements, so calculating a parameterized permutation in isolation is not efficient. In the following we consider parameterized permutations in groups.

Each finite permutation group has a compositional nature. The Krull-Schmidt theorem states that a finite group satisfying certain conditions can be uniquely written as a direct product of a set of indecomposable subgroups. One of its variation states that It is a direct product decomposition of indecomposable disjoint subgroups and the decomposition is unique if its generating set satisfies certain conditions [29]. This decomposition is called *geometric decomposition*, formally defined as follows. Suppose two permutations σ and σ' act on a finite set S. They are called *disjoint* if and only if for all $s \in S$, $\sigma(s) \neq s$ implies that $\sigma'(s) = s$ and $\sigma'(s) \neq s$ implies that $\sigma(s) = s$. Likewise, two permutation groups G and G' acting on S are called *disjoint* iff every pair of permutations $\sigma \in G$ and $\sigma' \in G'$ are disjoint. Intuitively, G and G' act on disjoint parts of S. So disjoint permutations and disjoint permutation groups are commutative.

Let A be a generating set of an arbitrary permutation group G. Suppose A can be decomposed into disjoint subsets, $A = A_1 \cup A_2 \cup \cdots \cup A_n$, such that each subset A_i cannot be further decomposed into multiple disjoint subsets. G_i is the subgroup generated by A_i. Then $G = G_1 \times G_2 \times \cdots \times G_n$.

$$G = G_1 \times G_2 \times \cdots \times G_n. \tag{1}$$

Since the choice of generators affect the generating set which may induce different decompositions. To make sure the uniqueness of the decomposition , the generating set A has to satisfy two conditions: (i) A does not contain elements in the form $\sigma = gh$ such that g and h are not identity elements and they are not disjoint; (ii) if a subset $A_i \subseteq A$ generates the subgroup $G_i \leq G$ such that $G_i = H_1 \times H_2$ and H_1 and H_2 are disjoint groups, then there exists a partition $A_i = S_1 \times S_2$ such that S_i generates H_i. The geometric decomposition of a permutation group can be obtained using GAP [29]. Without loss of generality, we assume that any detected automorphism group is in its geometric decomposition form. For each disjoint subgroup, we investigate how to generalize it to a parameterized subgroup. The whole parameterized group will be the direct product of all parameterized subgroups. In this way, the generalization process of each subgroup is independent. If some subgroup fails to be generalized, the parameterized group will be the direct product of the successful ones. The automorphism group of $\mathcal{M}(3)$ is divided into three indecomposable disjoint subgroups as represented by their generating sets[8].

$$\begin{aligned}
G_1 =&\langle (u_1 := 0, u_1 := 1)(v_1 := 0, v_1 := 1)(u_2 := 0, u_2 := 1)(v_2 := 0, v_2 := 1)(V^0_{color} := 0, V^0_{color} := 1) \\
&\quad (V^0_{color_1} := 0, V^0_{color_1} := 1)(V^0_F := 0, V^0_F := 1)(V^0_{F_1} := 0, V^0_{F_1} := 1)(V^0_{F_2} := 0, V^0_{F_2} := 1), \\
&\quad (u_1 := 0, u_1 := 2)(v_1 := 0, v_1 := 2)(u_2 := 0, u_2 := 2)(v_2 := 0, v_2 := 2)(V^0_{color} := 0, V^0_{color} := 2) \\
&\quad (V^0_{color_1} := 0, V^0_{color_1} := 2)(V^0_F := 0, V^0_F := 2)(V^0_{F_1} := 0, V^0_{F_1} := 2)(V^0_{F_2} := 0, V^0_{F_2} := 2)\rangle \\
G_2 =&\langle (c := 0, c := 1)(color := 0, color := 1)(V^1_F := 0, V^1_F := 1)(V^1_{F_1} := 0, V^1_{F_1} := 1)(V^1_{F_2} := 0, V^1_{F_2} := 1), \\
&\quad (c := 0, c := 2)(color := 0, color := 2)(V^1_F := 0, V^1_F := 2)(V^1_{F_1} := 0, V^1_{F_1} := 2)(V^1_{F_2} := 0, V^1_{F_2} := 2)\rangle \\
G_3 =&\langle (F := 0, F := 1)(F_1 := 0, F_1 := 1)({F_2}^1 := 0, {F_2}^1 := 1)\rangle
\end{aligned}$$

The generalization problem is nontrivial as a generator is often rather too full of explicit permutation detail of a number of variables and values, which makes it hard to discover the regular patterns of subgroups. In order to simplify the generalization problem, we reduce each generator of multiple variables to an equivalent generator of a single variable. The basis of this idea is our common observation in practice that for most generators, all the variables involved in one generator have the same value symmetries, *i.e.*, for any variables x and y in one generator σ, $\sigma(x := a) = (x := b)$ implies $\sigma(y := a) = (y := b)$. There are two possible reasons behind this observation: the value of a variable reference is often used to assign another variable; a variable in a concurrent model may become multiple local variables after the transformation to its sequential counterpart. Thus based on this observation, our generalization requires that

[8] A permutation can be defined over variables and values. A permutation of variables can be represented by permutations on their values. (For example, a permutation σ swaps two variables x and y where the domain of x and y is *Dom*. σ can be rewritten to $(x = a, y = b), (x = b, y = a)$ for any pair of values $(a, b) \in Dom \times Dom$.) We normalize the form of a permutation via defined over only values, *i.e.*, $\sigma(v_1 := a_1, \cdots, v_n := a_n) = (v_1 := \sigma(a_1), \cdots, v_n := \sigma(a_n))$ for ease of presentation, where v_i can be a local, function, global variable, or represent a dimension of an array variable. V^i_A denotes dimension i of array A and is used for permutations between array elements only different in dimension i. If v_i is an array variable, σ acts on the values of all its elements.

all the variables of a generator always have the same value symmetries. If this condition is not satisfied, the subgroup is ruled out from generalization. After the simplification, for $\mathcal{M}(3)$ we get $G_1 = \langle (u_1 := 0, u_1 := 1),\ (u_1 := 0, u_1 := 2)\rangle$, $G_2 = \langle (c := 0, c := 1), (c := 0, c := 2)\rangle$ and $G_3 = \langle (F := 0, F := 1)\rangle$; similarly, in $\mathcal{M}(4)$ $G_1 = \langle (u_1 := 0, u_1 := 1, u_1 := 2),\ (u_1 := 0, u_1 := 2, u_1 := 1)\rangle$, $G_2 = \langle (c := 0, c := 1), (c := 0, c := 2)\rangle$ and $G_3 = \langle (F := 0, F := 1)\rangle$. After the simplification, the generating set of G_1 has a single generator $(0, 1)$, the generating set of G_2 includes two generators $(0, 1)$ and $(0, 2)$ The quality of the representation of a parameterized generator is vital to the effectiveness of our approach. A good representation should satisfy the following conditions in order to make parameterized permutations easy to prove and apply:

- It should be *universal*. It should be decided by the information in $\mathcal{M}(\mathcal{D})$ rather than the information only available in a particular instance $\mathcal{M}(d)$.
- It should be *compact*. An explicit representation of all permutations may be used. But such a representation may not be feasible all the time, since the number of permutations tends to increase exponentially with the values of parameters, fully symmetric systems for example.
- It should be *general*. It should be able to identify a large (or even complete) list of permutations.
- It should be *efficiently computable*. The construction of permutations from this representation should have low overhead in order to be practical.

A permutation group of a single variable *var* is generalized in two cases. First, if a subgroup is a fully symmetric group of $\mathcal{M}(d)$ and the domain of *var* is consecutive integers $[x, y]$, then the corresponding parameterized subgroup is also a fully symmetric group of $[gen(x), gen(y)]$ where

$$gen(v) = \begin{cases} exp(p_1, \cdots, p_n) & \text{if } v \text{ is equal to the value of an arithmetic expression} \\ & exp(p_1, \cdots, p_n) \text{ in CSP with parameters } p_1, \cdots, p_n in\ \mathcal{D} \\ v & \text{otherwise} \end{cases}$$

Since G_2 in both $\mathcal{M}(3)$ and in $\mathcal{M}(4)$ is fully symmetric subgroups of $[0, 2]$, we get two candidate parameterized subgroups $[0, N\text{-}1]$ and $[0, 2]$ respectively. Second, if a subgroup is not a fully symmetric group of $\mathcal{M}(d)$ (like G_1 in $\mathcal{M}(4)$). Each parameterized generator σ is represented as a single function, *i.e.*, $\sigma(var) = f(var, V)$ where V is a tuple of parameters from $\mathcal{D}$. $f(var, V)$ is discovered from the CSP: for any pair of variables (v, v') in the subgroup (before simplification), if v represents a dimension i of an array A, the expression $f(v', val(V))$ is used to assign the dimension i of A in the CSP; or there exists an atomic constraint[9] in the form $v = f(v', val(V))$. The generating set of the parameterized subgroup is composed of parameterized generators. Therefore, for G_1 in $\mathcal{M}(4)$, we found two parameterized generators for it, $\sigma(u_1) = (u_1 - 1) mod\ N$ and $\sigma(u_1) = (u_1 + 1) mod\ N$.

[9] An *atomic* constraint is a constraint of the form $x \bowtie y$ in which there exists no relational operators from $\bowtie$ in x or y, where $\bowtie$ denotes an arithmetic operator.

Step Three: Checking the Validity of a Parameterized Permutation It is possible that a parameterized generator is only a symmetry for a restricted set of instances. Thus it is necessary to check the validity of each parameterized generator. For each subgroup, each parameterized generator σ is applied to the corresponding CSP of the model $\mathcal{M}(\mathcal{D})$. For a fully symmetric subgroup, σ denotes an arbitrary permutation in it. If c and $\sigma(c)$ are semantically equivalent for each constraint c, then σ is a parameterized symmetry. The equivalence checking can be implemented with the help of a theorem prover.

The parameterized automorphism groups of the running example are direct product of three subgroups: the generating set of G_1 is $\langle(\sigma(x) = (x-1) mod\ N$, $\sigma(x) = (x+1) mod\ N\rangle$ where x is the value of variables $u_1, v_1, u_2, v_2, V^0_{color}, V^0_{color_1}$, $V^0_F, V^0_{F_1}$ and $V^0_{F_2}$; G_2 is a fully symmetric group of domain $[0, 2]$ of variables $c, color, V^1_F, V^1_{F_1}$ and $V^1_{F_2}$; G_3 is also a fully symmetric group of domain $[0, 1]$ of variables F, F_1 and F_2.

7 Related Work and Conclusion

The importance of detecting symmetries in model checking area has garnered much interest in recent years and several methods have emerged.

Scalarset Method One of the oldest and most widespread symmetry detection approaches is using *scalarset*. It is first introduced by Ip and Dill in the explicit model checker Murφ [24]. Scalarset is a data type which determines an unordered finite set of consecutive integer values. It is a fully symmetric type, *i.e.*, permuting any values of a scalarset type throughout the state space must result in an automorphism. So this method is only capable of handling fully symmetric components. For usage, a user may define a new scalarset type for a class of fully symmetric components and assign each component's identifier to a unique value of this type. Then the verifier automatically extracts the automorphisms from scalarset types. In this way, scalarsets provide a convenient and efficient way for users to define symmetries, considering the number of automorphisms generated by a scalarset is the factorial of its size. This method is applied to several other model checkers like Spin [4, 5], Uppaal [22].

However, it has two disadvantages that impose a heightened burden on designers. First, the applicability of this method relies on designers to have expert insights to precisely identify identical components in a system. Second, in order to make sure the symmetry extraction method is sound, a much rigorous syntactic requirement is placed on operations of scalarsets to rule out all possible symmetry breaking constructs. Loosely speaking, a variable of some scalarset type can only be referred to as the index of an array with the same scalarset index type and assigned to, compared for (in)equality with another variable of exactly the same type. Last but not least, it is applicable only for fully symmetric systems.

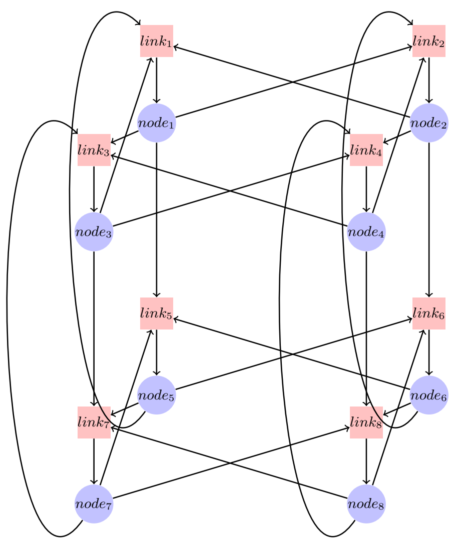


Fig. 6: A 3-dimensional hypercube

Static Channel Diagrams Donaldson and Miler design a fully automatic approach to detecting process symmetries for channel-based communication systems [14, 15]. Their approach also involves constructing a colored graph called *static channel diagram* from a Promela model, whose automorphisms possibly correspond to the automorphism of the Kripke structure along with the model. Each node is created for each process or channel. If a process possibly sends a message to a channel, then a directional edge is created from the process node to the channel node. Similarly, if a process possibly receives a message from a channel, then a directional edge is created from the channel node to the process node. All process (*resp.* channel) nodes representing the same type of processes (*resp.* channel) have the same unique color. Figure 6 shows the static channel diagram of the message routing algorithm in a three dimensional hypercube network [13]. The generators for the automorphism group in the static channel diagram are computed using a graph automorphism algorithm. But a computed generator may not be a real automorphism in the state space. In order to preserve the soundness of the detection approach, each generator obtained from the diagram has to be validated that it transforms the original program $\mathcal{P}$ into an equivalent program with the complexity $O(|\mathcal{P}| \log |\mathcal{P}|)$.

Similar to scalarset approaches, there is a series of limitations on input Promela programs to rule out symmetry breaking constructs. One of them is disallowing the use of process identifiers in relational and arithmetic operations, which is commonly thought to be the source of breaking symmetries. However, it is not necessary the case in many systems such as the motivating example. They propose a straightforward strategy to relax this restriction, *i.e.*, rewriting a relational or arithmetic operation into a disjunction of all possible combinations of variable valuations. But the validity checking for each generator would suffer a significant loss in performance because the size of the program becomes at most $O(n^k)$ of the original one, where n is the largest size of domains of variables representing process identifiers and k is the highest arity of any relational or arithmetic operations involving these variables.

Lastly, our method is remotely related to an on-the-fly symmetry detection and reduction approach proposed by Wahl and D'Silva [41]. It starts a reachability checking with the assumption that all processes are fully symmetric. As each transition is analyzed, the asymmetries it induces are used to partition the processes. Our approach can deduce how an arbitrary transition breaks symmetries not limited to process symmetries prior to model checking. So combining two approaches can potentially improve the performance of symmetry reduction.

The main contribution of our work is a new automatic symmetry detection approach. To the best of our knowledge, our study is the first work to relax all the syntactic restrictions on the model form, and also the first work to consider various process symmetries, data symmetries and their combinations. A variety of case studies showed that the overhead of symmetry detection is negligible.

References

1. `http://www.comp.nus.edu.sg/~pat/detection/`.
2. D. Angluin, J. Aspnes, M. J. Fischer, and H. Jiang. Self-stabilizing Population Protocols. *ACM Transactions on Autonomous and Adaptive Systems*, 3(4):643–644, 2008.
3. E. Ashcroft and Z. Manna. Formalization of Properties of Parallel Programs. In *Machine Intelligence*, volume 6. Edinburgh University Press, 1970.
4. D. Bosnacki, D. Dams, and L. Holenderski. Symmetric Spin. In *SPIN*, volume 1885, pages 1–19. Springer, 2000.
5. D. Bosnacki, L. Holenderski, and D. Dams. A Heuristic for Symmetry Reductions with Scalarsets. In *FME 2001: Formal Methods for Increasing Software Productivity*, volume 2021, pages 518–533. Springer Berlin / Heidelberg, 2001.
6. Canepa, Davide and Potop-Butucaru, Maria Gradinariu. Stabilizing token schemes for population protocols. *CoRR*, 2008.
7. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, 2000.
8. M. Cohen, M. Dam, A. Lomuscio, and H. Qu. A Data Symmetry Reduction Technique for Temporal-epistemic Logic. In *The 7th International Symposium on Automated Technology for Verification and Analysis*, ATVA '09, pages 69–83, Berlin, Heidelberg, 2009. Springer-Verlag.

9. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, Oct. 1991.
10. P. T. Darga, K. A. Sakallah, and I. L. Markov. Faster Symmetry Discovery using Sparsity of Symmetries. In *The 45th annual Design Automation Conference*, DAC '08, pages 149–154, New York, NY, USA, 2008.
11. G. Delzanno and A. Podelski. Model Checking in CLP. In *The 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 223–239, London, UK, UK, 1999. Springer-Verlag.
12. A. Donaldson, A. Miller, and M. Calder. Comparing the Use of Symmetry in Constraint Processing and Model Checking. In *The 4th international workshop on symmetry and constraint satisfaction problems*, SymCon'04, pages 18–25, Toronto, 2004.
13. A. F. Donaldson. *Automatic Techniques for Detecting and Exploiting Symmetry in Model Checking.* Dissertation, University of Glasgow, 2007.
14. A. F. Donaldson and A. Miller. Automatic Symmetry Detection for Model Checking Using Computational Group Theory. In J. Fitzgerald, I. Hayes, and A. Tarlecki, editors, *FM'05*, volume 3582 of *Lecture Notes in Computer Science*, pages 631–631. Springer Berlin / Heidelberg, 2005.
15. A. F. Donaldson and A. Miller. Automatic Symmetry Detection for Promela. *Journal of Automated Reasoning*, 41:251–293, November 2008.
16. I. Dotú and P. Van Hentenryck. Scheduling Social Golfers Locally. In *The 2nd international conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, CPAIOR'05, pages 155–167, Berlin, Heidelberg, 2005. Springer-Verlag.
17. E. A. Emerson and A. P. Sistla. Symmetry and Model Checking. *Formal Methods in System Design*, pages 105–131, 1996.
18. M. J. Fischer and H. Jiang. Self-stabilizing Leader Election in Networks of Finite-state Anonymous Agents. In *The 10th International Conference on Principles of Distributed Systems*, volume 4305 of *OPODIS'06*, pages 395–409, 2006.
19. W. Fokkink. Linear Process Equations. In *Modelling Distributed Systems*, Texts in Theoretical Computer Science. An EATCS Series, pages 69–79. Springer Berlin Heidelberg, 2007.
20. W. Fokkink and J. Pang. Cones and Foci for Protocol Verification Revisited. In *The 6th International conference on Foundations of Software Science and Computation Structures and joint European conference on Theory and practice of software*, FOSSACS'03/ETAPS'03, pages 267–281, Berlin, Heidelberg, 2003. Springer-Verlag.
21. The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.5.6*, 2012.
22. M. Hendriks, G. Behrmann, K. Larsen, P. Niebert, and F. Vaandrager. Adding Symmetry Reduction to UPPAAL. In *Formal Modeling and Analysis of Timed Systems*, volume 2791, pages 46–59. Springer Berlin / Heidelberg, 2004.
23. C. A. R. Hoare. *Communicating Sequential Processes.* International Series in Computer Science. Prentice-Hall, 1985. New version at www.usingcsp.com/cspbook.pdf.
24. C. N. Ip and D. L. Dill. Better Verification through Symmetry. *Formal Methods System Design*, 9(1-2):41–75, 1996.
25. M. M. Jaghoori, M. Sirjani, M. R. Mousavi, E. Khamespanah, and A. Movaghar. Symmetry and Partial Order Reduction Techniques in Model Checking Rebeca. *Acta Inf.*, 47(1):33–66, Jan. 2010.

26. M. M. Jaghoori, M. Sirjani, M. R. Mousavi, and A. Movaghar. Efficient Symmetry Reduction for an Actor-based model. In *Proceedings of the Second international conference on Distributed Computing and Internet Technology*, ICDCIT'05, pages 494–507, Berlin, Heidelberg, 2005. Springer-Verlag.
27. R. A. Krzysztof and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer-Verlag New York, Inc., New York, NY, USA, 1991.
28. C. Lecoutre and S. Tabary. Lightweight Detection of Variable Symmetries for Constraint Satisfaction. In *The 21st IEEE International Conference on Tools with Artificial Intelligence*, ICTAI '09, pages 193–197, Washington, DC, USA, 2009. IEEE Computer Society.
29. B. D. MacArthur, R. J. Sánchez-García, and J. W. Anderson. Symmetry in Complex Networks. *Discrete Applied Mathematics*, 156(18):3525 – 3531, 2008.
30. C. Mears, M. G. De La Banda, M. Wallace, and B. Demoen. A Novel Approach for Detecting Symmetries in CSP Models. In *The 5th international conference on Integration of AI and OR techniques in constraint programming for combinatorial optimization problems*, CPAIOR'08, pages 158–172, Berlin, Heidelberg, 2008. Springer-Verlag.
31. C. Mears, M. Garcia De La Banda, and M. Wallace. On Implementing Symmetry Detection. *Constraints*, 14(4):443–477, 2009.
32. C. Mears, T. Niven, M. Jackson, and M. Wallace. Proving Symmetries by Model Transformation. In *The 17th international conference on Principles and practice of constraint programming*, CP'11, pages 591–605, Berlin, Heidelberg, 2011. Springer-Verlag.
33. R. Milner. *Communication and Concurrency.* Prentice-Hall, Inc., 1989.
34. G. L. Peterson. Myths About the Mutual Exclusion Problem. *Information Processing Letters*, 12(3):115–116, 1981.
35. J.-F. Puget. Automatic Detection of Variable and Value Symmetries. In P. van Beek, editor, *Principles and Practice of Constraint Programming*, volume 3709 of *CP'05*, pages 475–489. Springer Berlin / Heidelberg, 2005.
36. A. P. Sistla, V. Gyuris, and E. A. Emerson. SMC: a Symmetry-Based Model Checker for Verification of Safety and Liveness Properties. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(2):133–166, Apr. 2000.
37. C. Spermann and M. Leuschel. ProB Gets Nauty: Effective Symmetry Reduction for B and Z Models. In *The 2008 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering*, TASE '08, Washington, DC, USA, 2008. IEEE Computer Society.
38. P. Van Hentenryck, P. Flener, J. Pearson, and M. Agren. Compositional Derivation of Symmetries for Constraint Satisfaction. In *The 6th international conference on Abstraction, Reformulation and Approximation*, SARA'05, pages 234–247, Berlin, Heidelberg, 2005. Springer-Verlag.
39. T. Wahl. Adaptive Symmetry Reduction. In *The 19th International Conference on Computer Aided Verification*, CAV'07, pages 393–405, Berlin, Heidelberg, 2007. Springer-Verlag.
40. T. Wahl and A. Donaldson. Replication and Abstraction: Symmetry in Automated Formal Verification. *Symmetry*, 2(2):799–847, 2010.
41. T. Wahl and V. D'Silva. A Lazy Approach to Symmetry Reduction. *Form. Asp. Comput.*, pages 713–733, 2010.
42. Y. Xiao, M. Xiong, W. Wang, and H. Wang. Emergence of Symmetry in Complex Networks. *Phys. Rev. E*, 77:066108, Jun 2008.

APPENDIX

A Transformation of Concurrent Programs

In this subsection, we show a straightforward principle of modeling parallel programs by means of nondeterministic sequential programs. Here we consider a simple concurrent specification model, which is however general enough to three different types of systems with respect to execution patterns, *i.e.*, sequential, parallel and distributed systems. Sequential systems execute one action at a time, possibly nondeterministically; parallel systems may execute multiple actions in parallel and achieve communication between different processes by shared variables; distributed systems also may execute actions in parallel but employ a handshaking mechanisms (like shared actions) for interprocess communication.

We consider a simple but general concurrent specification model in the style of event-driven processes. Figure 7 lists the syntax of our language. Omitted rules are identical to those in Figure 2. The language includes two familiar elementary data types, integer and boolean.[10]A system description consists of a set of global variable declarations, a set of process definitions and one initialization rule. Component processes of a parallel system are composed by $|||$ symbol which denotes that processes run concurrently *without* barrier synchronization; those of a distributed system are composed by $\|$ symbol which denotes that processes run concurrently *with* synchronization on common events. A component process is defined to be a sequential program with an option list of parameters. It is composed of a sequence of statements. Each statement may be an event-labeled statement composed of sub-statements which are atomically executed, an signal of process termination, if-else conditional choice, while-loop, or nondeterministically executed statements separated by $\Box$ symbol.

The translation function $\mathcal{T}$ is defined for translating each statement into one or more sequential programs recursively, separated by $\Box$ symbol. The preparatory step of the transformation is to introduce a new integer variable *state* for each component process to model its control points. Each atomic statement is labeled with a distinguished value of *state* of the form "k :". For a list of statements, say $\boldsymbol{S}$, let $first(\boldsymbol{S})$ be the value of *state* of its first statement in $\boldsymbol{S}$ and $last(\boldsymbol{S})$ be the value of its last statement, and $\mathcal{T}(\boldsymbol{S})(c)$ denote the transformation of $\boldsymbol{S}$ and c is the value of *state* of the successor statement of the last statement in $\boldsymbol{S}$. A component process can then be transformed by induction as follows:

- $\mathcal{T}(k : \mathit{Skip})(c) := [\mathit{state} = k] \to \{\mathit{state} := c\}$
- $\mathcal{T}(k : \langle \mathit{eid} \rangle \{\boldsymbol{S}\})(c) := [\mathit{state} = k]\langle \mathit{eid} \rangle \to \{\boldsymbol{S}; \mathit{state} := c; \}$
- $\mathcal{T}(\langle \mathit{estmt}_1, \mathit{estmt}_2, \cdots, \mathit{estmt}_n \rangle)(c) := \mathcal{T}(\langle \mathit{estmt}_1 \rangle)(first(\langle \mathit{estmt}_2, \cdots, \mathit{estmt}_n \rangle))$
$$\begin{aligned} &\Box \mathcal{T}(\langle \mathit{estmt}_2 \rangle)(first(\langle \mathit{estmt}_3, \cdots, \mathit{estmt}_n \rangle)) \\ &\Box \cdots \\ &\Box \mathcal{T}(\langle \mathit{estmt}_n \rangle)(c) \end{aligned}$$

[10] Composite data types, like arrays are excluded for ease of presentation.

$$\begin{aligned}
\langle \mathit{program} \rangle &::= \langle \mathit{vardecl} \rangle^* \langle \mathit{prodecl} \rangle^* \langle \mathit{init} \rangle \\
\langle \mathit{vardecl} \rangle &::= \textbf{var}\ g \\
\langle \mathit{prodecl} \rangle &::= \textbf{proc} \langle \mathit{pid} \rangle (\langle \textit{param-list}, ',' \rangle) \{ \langle \mathit{vardecl} \rangle^* \langle \mathit{estmts} \rangle \} \\
\langle \mathit{estmts} \rangle &::= \{ \langle \textit{estmt-list}, ';' \rangle \} \\
\langle \mathit{estmt} \rangle &::= \langle \mathit{eid} \rangle \{ \langle \textit{stmt-list}, ';' \rangle \} \\
&\quad | \mathit{Skip} \\
&\quad | \textbf{if}(\langle \mathit{guard} \rangle) \{ \langle \mathit{estmts} \rangle \}\ \textbf{else} \{ \langle \mathit{estmts} \rangle \} \\
&\quad | \textbf{while}(\langle \mathit{guard} \rangle) \{ \langle \mathit{estmts} \rangle \} \\
&\quad | \langle \mathit{estmt} \rangle \Box \langle \mathit{estmt} \rangle \\
\langle \mathit{proref} \rangle &::= \langle \mathit{pid} \rangle ((\langle \mathit{args} \rangle)) \\
&\quad | \langle \mathit{proref} \rangle \| \langle \mathit{proref} \rangle \\
&\quad | \langle \mathit{proref} \rangle ||| \langle \mathit{proref} \rangle \\
\langle \mathit{init} \rangle &::= \textbf{init}\ \langle \mathit{proref} \rangle
\end{aligned}$$

Fig. 7: Syntax of Concurrent Language

- $\mathcal{T}(k : \textbf{if}(\langle \mathit{guard} \rangle)\{\langle \mathit{estmts}_1 \rangle\}\ \textbf{else}\{\langle \mathit{estmts}_2 \rangle\})(c) :=$
  $[\mathit{state} = k \wedge \mathit{guard} = T] \rightarrow \{\mathit{state} := \mathit{first}(\langle \mathit{estmts}_1 \rangle)\}$
  $\Box[\mathit{state} = k \wedge \mathit{guard} = F] \rightarrow \{\mathit{state} := \mathit{first}(\langle \mathit{estmts}_2 \rangle)\}$
  $\Box\mathcal{T}(\langle \mathit{estmts}_1 \rangle)(c)$
  $\Box\mathcal{T}(\langle \mathit{estmts}_2 \rangle)(c)$
- $\mathcal{T}(k : \textbf{while}(\langle \mathit{guard} \rangle)\{\langle \mathit{estmts} \rangle\})(c) :=$
  $[\mathit{state} = k \wedge \mathit{guard} = T] \rightarrow \{\mathit{state} := \mathit{first}(\langle \mathit{estmts} \rangle)\}$
  $\Box[\mathit{state} = k \wedge \mathit{guard} = F] \rightarrow \{\mathit{state} := c\}$
  $\Box\mathcal{T}(\langle \mathit{estmts} \rangle)(k)$

Now we can transform a component process P that contains a sequence of statements $\boldsymbol{S_1}, \boldsymbol{S_2}, \cdots, \boldsymbol{s_n}$ as follows:

$$\mathcal{T}(P) := \mathcal{T}(\boldsymbol{S_1})(c_1)\ \Box\ \mathcal{T}(\boldsymbol{S_2})(c_2)\ \Box \cdots \Box\ \mathcal{T}(\boldsymbol{S_n})(c_n)$$

Let us now consider the transformation of concurrent composition of two processes $P_1|||P_2$.

$$\mathcal{T}(P_1|||P_2) := \mathcal{T}(P_1)\ \Box\ \mathcal{T}(P_2)$$

The transforation of the other concurrent composition of processes, $P_1\|P_2$, is slightly complicated, because P_1 and P_2 perform lock-step synchronization on common events. Then for any pair of common-event-labeled statements in P_1 and P_2, written in $(k_1 : e\{\boldsymbol{S_1}\})(c_1)$ and $(k_2 : e\{\boldsymbol{S_2}\})(c_2)$, their transformation will result in one statement, *i.e.*, $[\mathit{state}_1 = k_1 \wedge \mathit{state}_2 = k_2]e \rightarrow \{\boldsymbol{S_1}; \boldsymbol{S_2}; \mathit{state}_1 := c_1; \mathit{state}_2 := c_2\}$. Other statements are transformed in the same way as $P_1|||P_2$.

It is straightforward to write the sequential program in the format of linear process specification.

Complexity Analysis For a component process, the transformation takes one atomic statement at a time and translates it to one statement in the sequential program. For the interleving/parallel composition of processes, the resulting program at most has the total number of atomic statements of all processes in the worst case, the number of parameters are linear to the number of processes. Thus, the size of the analysis is linear to the number of processes instead of as exponentially large as the size of generating the state space.

B Proof of Theorem 3

Proof By definition, we must show that (i) if $s_1 \overset{e}{\rightarrow} s_2$, then $\sigma(s_1) \overset{\sigma(e)}{\rightarrow} \sigma(s_2)$, and (ii) $\sigma(init) = init$.

Suppose $s_1 \overset{e}{\rightarrow} s_2$ corresponds to the execution of the summand *sum* of $\mathcal{P}$. Without loss of generality, we assume there is only one global variable v_g in $\mathcal{P}$ and one local variable v_l in *sum*. $s_1 \overset{e}{\rightarrow} s_2$ is assumed to denote executing *sum* when $v_g := value_1$ and $v_l := value_2$. That is, when $v_g := value_1$ and $v_l := value_2$, its enabling condition f_e is true, event e is executed parameterized with the return value by its action function f_a, and global variables are updated in its *next-state* function f_n which leads to state s_2.

Suppose $\mathcal{C}$ is the CSP converted from $\mathcal{P}$ in Algorithm 3. By Proposition 1, all the constraints converted from f_e, f_a and f_n are satisfied when $v_g = value_1$ and $v_l = value_2$. By Theorem 3, σ is a constraint symmetry of $\mathcal{C}$. So all of the constraints from f_e, f_a and f_n are also satisfied when when $\sigma(v_g = value_1)$ and $\sigma(v_l = value_2)$. Again by Proposition 1, we get $\sigma(s_1) \overset{\sigma(e)}{\rightarrow} \sigma(s_2)$. Similarly, we can prove $\sigma(init) = init$. □

C Proof of Theorem 4

Theorem 4. *Let $\mathcal{C}$ be a CSP. and $\mathcal{C}'$ its corresponding CSP of $\mathcal{C}$ after transforming all array writing constraints. Then any constraint symmetry of $\mathcal{C}'$ is also a constraint symmetry of $\mathcal{C}$.* □

Proof Assume σ is a constraint symmetry of $\mathcal{C}'$. The constraints in $\mathcal{C}$ are separated into two sets: one containing all the array writing constraints S_1 and the other containing all the rest constraints S_2; similarly, the constraints in $\mathcal{C}'$ are separated into two sets: one containing all the constraints transformed from an array writing constraints S_1' and the other containing all the rest constraints S_2'. Since S_2 and S_2' are identical, σ is also a constraint symmetry for S_2.

We define a function $eval_s$ which takes an assignment s and a constraint c, and returns the satisfaction of c when evaluated as s. Without loss of generality, we assume there are no multi-dimensional arrays in $\mathcal{C}$. Suppose an array writing constraint c in S_1 is $array_1[index] = value \wedge (\forall j \in \{0, \cdots, N-1\}.j \neq index \rightarrow$

$array_1[j] = array_0[j])$. It is transformed into the list L containing $N + 1$ constraints $\{array_1[index] = value, array_1[0] = array_0[0], \cdots, array_1[N-1] = array_0[N-1]\}$ in S_1'. Let s be an assignment of $\mathcal{C}$. Because all elements of an array have the same color which is different from that of any other variable. For any element $array_0[k]$ where $k \in \{0, \cdots, N-1\}$, $\sigma(array_0[k]) = array_0[k']$ where $k' \in \{0, \cdots, N-1\}$. This also applies to elements of $array_1$. There are three conditions to be considered:

- If the first constraint in L is evaluated to false at s, i.e. $eval_s(array_1[index] = value) = false$, then $eval_s(c) = false$. Because σ is a constraint symmetry, $eval_\sigma(s)(\sigma(array_1[index] = value)) = eval_{\sigma(s)}(array_1[\sigma(index)] = value) = false$. So $eval_{\sigma(s)}(\sigma(c)) = false$.
- Otherwise if there exists $i \in \{0, \cdots, N-1\}$ such that $eval_s(array_1[i] = array_0[i]) = false$ where $i \neq eval_s(index)$, then $eval_s(c) = false$. Since $eval_s(array_1[i] = array_0[i]) = false$, $eval_s(c) = false$ and $eval_{\sigma(s)}(\sigma(array_1[i] = array_0[i])) = eval_{\sigma(s)}(array_1[\sigma(i)] = array_0[\sigma(i)]) = false$. Because $i \neq eval_s(index)$, $eval_{\sigma(s)}(\sigma(i)) \neq eval_{\sigma(s)}(\sigma(index))$. Therefore, $eval_{\sigma(s)}(\sigma(c)) = false$.
- Otherwise, $eval_s(c) = true$. That is, $eval_s(array_1[index] = value) = true$ and $\forall j \in \{0, \cdots, N-1\}$ and $j \neq eval_s(index)$ such that $eval_s(array_1[j] = array_0[j]) = true$. Considering σ is a constraint symmetry, $eval_\sigma(s)(\sigma(array_1[index] = value)) = eval_\sigma(s)(array_1[\sigma(index)] = \sigma(value)) = true$ and $\forall j \in \{0, \cdots, N-1\}$ and $j \neq eval_s(index)$ such that $eval_{\sigma(s)}(\sigma(array_1[j] = array_0[j])) = eval_{\sigma(s)}(array_1[\sigma(j)] = array_0[\sigma(j)]) = true$. Because $j \neq eval_s(index)$, $eval_{\sigma(s)}(\sigma(j)) \neq eval_{\sigma(s)}(\sigma(index))$. So $eval_{\sigma(s)}(\sigma(c)) = true$.

Therefore, σ is also a constraint symmetry of $\mathcal{C}$. $\square$