Constraint-Based Automatic Symmetry Detection

Shao Jie Zhang*, Jun Sun*, Chengnian Sun†, Yang Liu‡, Junwei Ma*, and Jin Song Dong†
*Singapore University of Technology and Design, Singapore
†National University of Singapore, Singapore
‡Nanyang Technological University, Singapore
{shaojie_zhang, sunjun, junwei_ma}@sutd.edu.sg, {suncn, dongjs}@comp.nus.edu.sg, yangliu@ntu.edu.sg

***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 efforts, 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 symmetries arising from the CSP which induce automorphisms of the model. To the best of our knowledge, our method is the first approach that can automatically detect both process and data symmetries as demonstrated via a number of systems.**

I. Introduction

In practice, a certain (sometimes rich) degree of symmetries is ubiquitous in concurrent and distributed systems [26], [36]. A number of representative real-world complex networks, including a broad selection of biological, technological and social networks, are found to have a nontrivial symmetric structure [26]. 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 prior to the exploration. 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 [22], [34].

Existing symmetry reduction approaches have two main limitations in the identification of symmetries in a model. First, the soundness and efficiency highly depend on human efforts. 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 [22], [30], [31]. For instance, Murφ provides a special data type with a list of syntactic restrictions. All values that belongs to this type are equivalent. Although there are automatic approaches which do not need expert insights, they are designed for specific languages [24], [23], or require models to be written in specific patterns [13], [14]. 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 many kinds of process symmetries and data symmetries together. The workflow of our approach is shown in Figure 1.

First, a concurrent model is translated into a semantics-equivalent nondeterministic sequential model using existing approaches [3], [25]. 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. The worst case complexity of the translation is linear in 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 the 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 a graph automorphism generator named Saucy [12]. The detected symmetries can be used later to speed up the performance of a state space exploration tool, *e.g.*, a model checker or a simulator.

The above steps can be performed fully automatically. The effectiveness and efficiency of our approach have been demonstrated via a variety of systems.

The rest of this paper is organized as follows. Section II presents a simple motivating example. Section III introduces relevant background information and terminology used throughout this paper. Section IV describes our automatic symmetry detection approach in details and proves the soundness of our approach. Section V presents the results of our case studies. Section VI surveys related work. Section VII concludes the paper and discusses possible future work.

978-1-4799-0215-6/13/$31.00 © 2013 IEEE

ASE 2013, Palo Alto, USA

Fig. 1: Automatic symmetry detection workflow

II. Motivating Example

In the following, we use a token circulation protocol [2] as a running example. All the agents, or nodes, are deployed in a directed ring. The protocol requires the existence of a leader. Each agent has two single-bit variables recording its token and label and one Boolean constant indicating whether it is a leader. Only agents that are adjacent can interact (the source node is the initiator and the target is the responder). During an interaction, two agents update both of their states according to two predefined transition rules . If two agents have the same label, the responder is a leader and the initiator is not, the responder sets its label to the complement of the initiator's label; otherwise the responder copies the label from the initiator. If an interaction triggers a label change, a token is passed from the initiator to the responder. Starting from an arbitrary configuration, the protocol guarantees that eventually there is always one and only one agent holding a token.

The concurrent model of this protocol with N agents is described in Figure 2 using the syntax of Communicating Sequential Programs [32]. Process $Rule_1$ (or $Rule_2$) defines how an initiator u interacts with a responder v. Every time there is an interaction in the network, the initiator and responder must update themselves according to the two transition rules. A rule is applicable only if the guard condition (*e.g.*, $!leader[v] \wedge label[u]! = label[v]$) is satisfied. An event (*e.g.*, $rule_2$) may be attached with variables updating (*e.g.*, $token[u] := 0; token[v] := 1; label[v] =: label[u]$). The whole token circulation protocol is described as process $TokenCirculation$, which is the interleaving (modeled by the operator $|||$) of all possible interactions in the network. Initially, the system can be in any possible configuration and the initial variable valuation is omitted here for simplicity.

$$
\begin{aligned}
&Rule_1(u,v) = [!leader[u] \wedge leader[v] \wedge label[u] = label[v]]\ rule_1\\
&\qquad \{token[u] := 0; token[v] := 1; label[v] := 1 - label[u];\}\\
&\qquad \rightarrow Rule_1(u,v);\\
&Rule_2(u,v) = [!leader[v] \wedge label[u]! = label[v]]\ rule_2\\
&\qquad \{token[u] := 0; token[v] := 1; label[v] := label[u];\}\\
&\qquad \rightarrow Rule_2(u,v\};\\
&TokenCirculation() = (|||x : 0..N-1@\\
&(Rule_1(x,(x+1)\ mod\ N)|||(Rule_2(x,(x+1)\ mod\ N));
\end{aligned}
$$

Fig. 2: Concurrent model of token circulation protocol

Simple as the protocol is, the protocol exhibits non-trivial symmetries: (a) process symmetries that rotate every process following the network direction; (b) data symmetries that swap the label values; (c) the combinations of process and data symmetries that permute processes and label values together.

Existing data symmetry detection approaches [10], [22] rely on scalarset annotations to discover fully symmetric components (*i.e.*, components which are identical up to rearranging their identifiers). Although values of all *label* variables are fully symmetric in this case, that is, permuting the values 1 to 0 and 0 to 1 for all *label* variables together over all the states and transitions of the state space results in the same state space, the arithmetic operations on the variables prohibit the use of scalarsets. Further, the protocol does not take message-passing paradigm, so the approaches [13], [14], [24], [23] 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, *i.e.*, no existing approaches can find all symmetries in this example.

III. Preliminaries

This section is devoted to the background knowledge of symmetry reduction in one application area of state space exploration, *i.e.*, model checking, and the relevant concepts of constraint satisfaction problems.

A. 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.*, $(\forall s_1, s_2 \in S; e \in \Sigma.\ s_1 \xrightarrow{e} s_2 \Rightarrow \sigma(s_1) \xrightarrow{e} \sigma(s_2)) \wedge \sigma(init) = init$. A group G is an automorphism group of $\mathcal{L}$ iff every $\sigma \in G$ is an automorphism of $\mathcal{L}$. A permutation σ is said to be an invariance of $\mathcal{L}$ and property ϕ iff it is an automorphism of $\mathcal{L}$ and $\sigma(\phi) \equiv \phi$ 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. If G is an invariance group of $\mathcal{L}$ and ϕ, then $\mathcal{L}$ satisfies ϕ iff $\mathcal{L}_G$ satisfies ϕ [8].

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 $\sigma(st) = (\sigma(s_1), \sigma(s_2), \cdots, \sigma(s_n))$.

B. Constraint Satisfaction Problem

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 [9]. A *constraint symmetry* is a solution symmetry that preserves the constraints of the CSP [9]. But a solution symmetry may not be a constraint symmetry. For example, a CSP is $(V = \{x, y, z\}, D = \{1, 2, 3\}, C = \{x < y, y < z\})$. It only has one solution $\{x = 1, y = 2, z = 3\}$. One of its solutions symmetries is $(x = 2, x = 3)$. But it is not a constraint symmetry, because it maps the literals $\{x = 2, y = 3\}$ which satisfies $x < y$ to $\{x = 3, y = 3\}$ which does NOT satisfy it. For a CSP (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.

IV. Automatic Symmetry Detection

In the section, we describe an automatic approach to detecting the symmetries of a concurrent model. It translates a concurrent model into a CSP whose symmetries can be exploited using the state-of-the-art detection approaches for CSPs.

Algorithm 1 gives an overview of the overall approach. There are three main steps. The first step, as described in procedure *Concurrent2Sequential*, converts a concurrent model *CurModel* into its semantics-equivalent nondeterministic sequential model *SeqModel*. The second step, as described in lines 3-9, separately transforms each enabling condition and each next-state program in *SeqModel*, and its *init* statement to a semantics-equivalent CSP as shown by Procedure *Transform*. These CSPs are then merged into one single CSP. The third step, detects variable, value and variable-value symmetries in the merged CSP, as described in Procedure *DetectSymmetries*. Further, we prove that each detected symmetry is a real automorphism of the

Algorithm 1: Overview of our approach

```
1  autos := ∅; VL := ∅; csps := ∅;
2  SeqModel := Concurrent2Sequential(CurModel)
3  identify the set of global variables VG in SeqModel;
4  foreach summand sum in SeqModel do
5      identify the set of local variables locals of sum;
6      VL := VL ∪ locals;
7      foreach function or enabling condition f in sum do
8          csps := csps ∪ {Transform(f)};
9  csps := csps ∪ {Transform(init)};
10 autos := DetectSymmetries (Merge(csps), VG, VL);
```

LTS of the original concurrent model. Lastly, we present two lightweight but effective optimization methods.

A. Step 1: Conversion from Concurrent to Sequential

We briefly introduce the principle of modeling concurrent models by means of nondeterministic sequential models. The corresponding 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. Basically, the preparatory step of the transformation is to introduce a new integer variable state for each process in the model to represent its control points, and then a syntactic transformation is performed to translate each statement into one or more sequential statements recursively. Then a concurrent model is reduced into a sequential one which captures all its behaviors. Note that the idea of linking concurrent models to nondeterministic sequential models goes back to the work of Ashcroft and Manna [3], [20] for proving the correctness of concurrent programs. The detailed transformation process is also explained in [37]. The transformation is general enough to handle three different types of systems with respect to execution patterns, *i.e.*, sequential and parallel systems with synchronous and asynchronous communication. Therefore our approach is not specific to one particular specification language. Moveover, for a concurrent model, its corresponding sequential model can be extracted in linear time [37]. The resulting model has the total number of atomic statements of all processes in the worst case.

Figure 3 shows the sequential model for the token circulation protocol. The nondeterministic sequential model is written in a single process with data variables that describes a system as a set of guarded and nondeterministic transitions. It contains a single parameterized recursive process definition and the initial parameter valuations of this process. The left-hand side of the process definition is a process name with a vector of data parameters. Here we refer to these parameters as *global variables*. An addition operator in the right-hand side 'sums' 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 event (if any) and a next-state program from left to right. Each local variable can be evaluated to any value of its type nondeterministically. It is *read-*

type *AG* : $0..N-1$
type *BIT* : $0..1$
proc *TokenCirculation*(*BIT*$[N]$ *token*, *BIT*$[N]$ *label*, *BOOL*$[N]$ *leader*) $=$
AG u_1.*AG* v_1.$[v_1 = (u_1 + 1)$ *mod* $N \wedge !leader[u_1] \wedge leader[v_1] \wedge label[u_1] = label[v_1]]$
$rule_1\{token[u_1] := 0; token[v_1] := 1; label[v_1] := 1 - label[u_1];\} \rightarrow$ *TokenCirculation*(*token*, *label*, *leader*)
$+$
AG u_2.*AG* v_2.$[v_2 = (u_2 + 1)$ *mod* $N \wedge !leader[v_2] \wedge label[u_2]! = label[v_2]]$
$rule_2\{token[u_2] := 0; token[v_2] := 1; label[v_2] := label[u_2];\} \rightarrow$ *TokenCirculation*(*token*, *label*, *leader*);
init *TokenCirculation*$(*)$;

Fig. 3: Sequential model of the token circulation protocol

only and cannot be of array type[1]. 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; the effect of the summand is decided by its next-state program which updates the global variables. A next-state program is composed of a sequence of statements. A statement can be an assignment, conditional, or while-loop statement. Besides, there is an initial valuation of global variables denoted by $init$, which is the entry where the process starts to execute. The symbol $*$ denotes the nondeterministic choice of all possible evaluations of global variables.

For the running example, the transition in Process $Rule_1$ (*resp.* $Rule_2$) is transformed into the first (*resp.* second) summand in the sequential model. There are two process identifiers used in each transition from the domain $\{0 \cdots N-1\}$. The initiator and responder ids u and v are transformed into u_1 and v_1 (*resp.* u_2 and v_2) in the first (*resp.* second) summand.

B. Step 2: Transformation from the Sequential Model to a CSP

We describe how to convert a function or the $init$ statement into the static single assignment form (SSA) [11] 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. 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.

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

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. Note that this assumption puts little limitation on our approach. Because the loop considered here is the loop included in one next-state program that is atomically executed. It is rare for a practical system to put the whole loop in one atomic step.

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 cannot 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.

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

Take the next-state program of the first summand in Figure 3 (*i.e.*, $token[u_1] := 0; token[v_1] := 1; label[v_1] :=$

[1]If a local variable is an array, the language can be extended to support it easily, as we have done in our tool.

$1 - label[u_1]$;) as an example. Its SSA form is

$$\begin{aligned} token_1 &:= \varphi(token_1, token, u_1, 0) \\ token_2 &:= \varphi(token_2, token_1, v_1, 1) \\ label_1 &:= \varphi(label_1, label, v_1, 1 - label[u_1]) \end{aligned}$$

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 behaviors as the original program [11]. After the conversion of a function to SSA, the next conversion 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 an 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 an enabling condition, since it is already a constraint, it does not need any transformation. For the $init$ statement, we convert it into a constraint in a very similar way. Suppose the process in the sequential model 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 next-state program, enabling condition and the $init$ statement to build one large CSP for this whole sequential model.

For the running example, the conversion step builds the corresponding CSP for its sequential model 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.

C. Step 3: Symmetry Detection on CSP

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 as follows.

Our approach is based on the automatic symmetry detection method for CSP proposed by Puget [29]. 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 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 in three rules:

- 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. *[29] Let $\mathcal{C} = (V, D, C)$ 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 $\mathcal{C}$. For each constraint $c \in C$, s satisfies c iff $\sigma(s)$ satisfies c.* □

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 the sequential model into. Some variables in a sequential model cannot be 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 scheme 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 as follows.

Theorem 3. *Let $\mathcal{L} = (S, init, \Sigma, \rightarrow)$ be its labeled transition system of a concurrent model $\mathcal{M}$. Each automorphism σ we get in Algorithm 1 is an automorphism of $\mathcal{L}$.*

$$
\begin{aligned}
V &= \{u_1, v_1, u_2, v_2, leader[N], token[N], label[N], token_1[N], token_2[N], token_3[N], label_1[N]\} \\
D &= \{AG, AG, AG, AG, BOOL, BIT, BIT, BIT, BIT, BIT, BIT\} \\
C &= \begin{cases}
v_1 = (u_1 + 1)\ mod\ N \wedge !leader[u_1] \wedge leader[v_1] \wedge label[u_1] = label[v_1] \\
token_1[u_1] = 0 \wedge (\forall t \in AG.t \neq u_1 \rightarrow token_1[t] = token[t]) \\
token_2[v_1] = 1 \wedge (\forall t \in AG.t \neq v_1 \rightarrow token_2[t] = token_1[t]) \\
label_1[v_1] = 1 - label[u_1] \wedge (\forall t \in AG.t \neq v_1 \rightarrow label_1[t] = label[t]) \\
v_2 = (u_2 + 1)\ mod\ N \wedge !leader[v_2] \wedge label[u_2]! = label[v_2] \\
token_3[u_2] = 0 \wedge (\forall t \in AG.t \neq u_2 \rightarrow token_3[t] = token[t]) \\
token_2[v_2] = 1 \wedge (\forall t \in AG.t \neq v_1 \rightarrow token_2[t] = token_3[t]) \\
label_1[v_2] = label[u_2] \wedge (\forall t \in AG.t \neq v_2 \rightarrow label_1[t] = label[t])
\end{cases}
\end{aligned}
$$

Fig. 4: Constraint satisfaction problem of the token circulation protocol

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

Suppose $\mathcal{P}$ is an equivalent sequential model of $\mathcal{M}$, and $s_1 \xrightarrow{e} s_2$ 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. $s_1 \xrightarrow{e} s_2$ 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 and global variables are updated in its next-state function f_n which leads to state s_2.

Suppose $\mathcal{C}$ is the constraint satisfaction problem converted from $\mathcal{P}$ in Algorithm 1. By Theorem 1, all the constraints converted from f_e 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 and f_n are also satisfied when $\sigma(v_g = value_1)$ and $\sigma(v_l = value_2)$. Again by Theorem 1, we get $\sigma(s_1) \xrightarrow{e} \sigma(s_2)$. Similarly, we can prove $\sigma(init) = init$. □

Note that the inverse of the theorem may not hold. For example, if two processes of the same type identical up to swapping their process identifiers are intentionally modeled as processes of two different types, this process symmetry is not reflected in its corresponding colored graph.

The number of nodes in the colored graph of a CSP is the sum of the number of literals, which is the product of the variable domain sizes, and the number of allowed assignments for 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 and space complexity of constructing the colored graph for a CSP accumulate to $t \times O(m^n)$ where t is in the number of constraints.

Figure 5 shows a part of the colored graph obtained from the CSP of the running example with $N = 3$. 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 part of the first constraint in the CSP, *i.e.*, $v_1 = (u_1 + 1)\ mod\ N \wedge label[u_1] = label[v_1]$. We skip the representation of all nodes generated from $v_1 = (u_1+1)\ mod\ N$ and variable and value nodes for v_1 and u_1. Note that rotating three $label$ variables clockwise still yields the same graph; swapping any literals of the form $label[i] := 0$ and $label[i] := 1$ for all $0 <= i < 3$ in all the assignments yields the same graph.

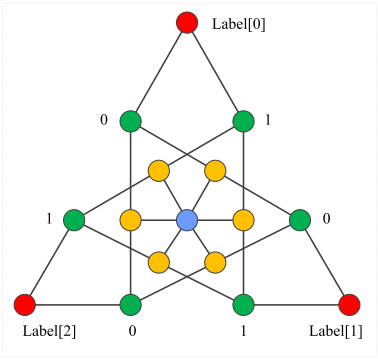

Fig. 5: Part of the colored graph of the running example's CSP

Example For the running example, assume there are three processes with ids 0, 1 and 2, it has 3 process symmetries from rotating the processes following the direction of the network, *i.e.*, $(0)(1)(2), (0,1)(1,2), (0,2)(1,2)$[2]; it has 2 data symmetries from swapping all the possible values of all $label$ variables, *i.e.*, $(0)(1), (0,1)$. Further, new symmetries are introduced by the product of these automorphisms. Therefore, we discover 6 symmetries in total.

D. Optimization

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

1) Breaking Down Array Writing 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 the number of variables as small as possible. We transform it into $K + 1$ simple constraints each involving

[2]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 $b \in \Omega \setminus \{a_1, a_2, \cdots, a_n\}$, $\sigma(b) = b$.

much fewer variables in the following way[3] 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 of the transformation is stated by the following theorem.

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: (1) 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 $eval_{\sigma(s)}(\sigma(c)) = false$; (2) 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, $eval_{\sigma(s)}(\sigma(c)) = false$; (3) 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 $\mathcal{C}$. □

2) 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 that 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 derived from the enabling condition and $y = x + 1$ is the *next-state* program of the same summand in the sequential model. 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.

V. Case Studies

We have implemented the colored graph construction described in Section IV. The resulting graph is input to Saucy [12] which produces the generating set of the automorphism group of a graph. For a group, its generating set is a subset whose elements are denoted by generators such that each element of the group can be obtained by the combination of generators of this subset. A generating set is often used as a compact representation of a group. Then the generating set is input to GAP [19] system which produces all the elements in the group. All experiment data is online [1], part of which is summarized in Table I.

The experimental cases cover a variety of computing systems. From the perspective of execution patterns, they include sequential systems, concurrent systems with synchronous 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 I, |*Colored Graph*| denotes the size of the colored graph generated for each configuration, *Construction* denotes the time (in seconds) taken to construct the colored graph; |*Generators*| denotes the size of the generating set of the

[3]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.

TABLE I: Symmetry detection results on a Linux laptop with Intel 2.8GHz and 3.8 GB memory

System	| Colored Graph |	Construction(s)	Saucy(s)	|Generators|	|Aut(G)|	Scalar	SCD
Reader-writer problem [33]							
3	120	0.127	0.004	1	2	N	N
Peterson's mutual exclusion protocol [28]							
9	2311	0.695	0.018	8	362880	N	Y
12	4207	1.037	0.030	11	479001600		
A prioritized resource allocator[1][14]							
2-2-3	393	0.553	0.004	4	24	N	Y
3-3-4	534	0.902	0.005	7	864		
Three-tiered architecture[2][14]							
3-3-2	419	0.480	0.005	5	144	N	Y
3-3-3	452	0.515	0.006	6	1296		
4-4-3	518	0.508	0.006	8	6912		
Message passing in a hypercube network[3][14]							
5	3586	1.447	0.026	4	3840	N	N
6	11555	3.317	0.066	5	46080		
Dining philosophers							
10	556	0.492	0.005	1	10	N	N
20	1086	1.033	0.007	1	20		
Miler's scheduler [27]							
10	487	2.665	0.001	0	0	N	N
Non-deterministic two-hop coloring protocol in undirected rings [2]							
9	2013	0.788	0.012	5	216	N	N
12	3105	1.282	0.013	5	288		
Self-stabilizing leader election protocol in complete graphs [18]							
12	21155	2.684	0.394	11	479001600	N	N
15	164809	15.783	8.326	14	1307674368000		
Self-stabilizing leader election protocol in directed rooted trees [6]							
15	466	3.954	0.275	4	16	N	N
19	580	7.404	0.005	6	128		
Self-stabilizing leader election protocol in rings [18]							
9	21378	4.781	0.093	1	9	N	N
12	214169	51.265	1.266	1	12		
Hanoi puzzle							
3	891	0.523	0.003	1	2	N	N
6	6520	1.636	0.023	1	2		
Scheduling the social golfer problem[4][16]							
3-3-4	1542	1.374	0.009	9	725760	N	N

[1] 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.*

[2] A configuration 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.

[3] A configuration is denoted by the number of dimensions of the hypercube. Note that the configuration d is composed of 2^d processes.

[4] A configuration 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.

automorphism group G of the colored graph computed by Saucy; *Saucy* denotes the time taken by Saucy to compute generators; $|Aut(G)|$ denotes the size of G computed by GAP. For systems whose configurations are not explained here, a configuration of each one is identified by the number of processes/components. The last two columns denote whether these symmetries can also be detectable by two popular existing approaches scalarset (*Scalar*) and static channel diagrams (*SCD*) (which are introduced in Section VI) without major changes on the original model, *e.g.*, rewriting each arithmetic or relational operation on variables related to process identifiers into the logical disjunction of all explicit variable values allowed by this operation, or remodeling the process communication mechanism into channels only. For a system with data symmetries, such as two-hop coloring protocol, existing approaches are still unable to discover them even if the system is changed into the form the approaches require.

As Table I 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 [13], [14] (*i.e.*, Peterson's protocol, resource allocator, three-tiered architecture and message passing in a hypercube network) and our approach is able to find all symmetries reported in their work efficiently. However, the effectiveness of our approach is not limited to message passing systems or process symmetries.

A. 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 IV-D1, 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

TABLE II: Symmetry reduction results I on a Windows laptop with Intel 3.4GHz and 8 GB memory with PAT 3.5 [32]

Model	States (Without Reduction)	States With Reduction	Gain
Dining philosophers			
10	154450	15489	90.0%
12	1684801	140536	91.7%
14	OM	1313052	-
Three-tiered architecture			
3-3-2	7840	462	94.1%
3-3-3	21952	286	98.7%
4-4-3	188272	OT	-
Non-deterministic two-hop coloring protocol in undirected rings			
3	13824	442	96.8%
4	331776	8058	97.6%
5	OM	OT	-

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.

B. Symmetry Reduction

We apply detected symmetries to the depth-first exploration of the whole state spaces of system configurations. A classic canonicalization function [22] is used to calculate a unique representative for each equivalence class of states, *i.e.*, applying all the automorphisms to a visited state to find the lexicographically smallest image. Table II contains the experimental results before and after symmetry reduction for part of systems configurations in Table I. In the table, *States* means the number of states stored, *OM* means exploring the configuration ran out of memory, *OT* means more than 2 hours, and *Gain* means the relative improvement on stored states brought by symmetry reduction. For the conducted experiments, the saving in terms of memory is 95.9% in average.

The computational overhead of symmetry reduction stems from checking whether the unique representative state of a visited state has been explored. Thus calculating representative states would be costly in time if there are a large number of automorphisms. It is known as constructive orbit problem (COP), which is *NP*-hard in general [7]. In practice, only systems with full symmetries are supported by existing symmetry reduction approaches, because representatives can be efficiently calculated in polynomial time.

One way of relaxing the prohibitive time requirement of COP is to allow multiple representatives for each equivalence class of states. Table III contains the experimental results for state space exploration without symmetry reduction, with symmetry reduction using unique representative, and with symmetry reduction using multiple representatives. From the table, it is shown that multi-representatives symmetry reduction stores more states than single-representative as expected. Here we consider the algorithm of calculating multiple representatives called local search in [15], which is only dependent on the generators of an automorphism group. A group with a large number of elements has a much smaller number of generators. So the multi-representatives approach is much faster than the single-representative one in most cases. It remains our future work to solve the COP problem efficiently for certain classes of automorphism groups in practice.

VI. Related Work

The importance of detecting symmetries for state space exploration has garnered much interest in recent years and several methods have emerged. The discussion on each method will largely be focused on the answers to two questions: (1) How much effort is required from model designers? (2) How many kinds of symmetries can be detected?

A. 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 [22]. 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 [21].

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. Last but not least, it is applicable only for fully symmetric systems.

It is worth to mention that the local variables in our work act as a much more generalized version of the popular scalarset. They both represent a subrange of values. A local variable may be the source of symmetries in a model, whereas a scalarset variable $must$ be the source of symmetries in a model. Since scalarset variables have to be specified by designers, the lack of a computer-assisted approach results in correctly expressing symmetries as wholly the designers' responsibility. But our approach automatically identifies which local variables are real symmetry makers and which operations are symmetry breaking constructs so as to remove all the burden from designers.

TABLE III: Symmetry reduction results II on a Windows laptop with Intel 3.4GHz and 8 GB memory with PAT 3.5 [32]

Model	Without Reduction		With Reduction (Unique)		With Reduction (Multi)	
	States	Time (Sec)	States	Time (Sec)	States	Time (Sec)
Dining philosophers						
10	154450	15.3	15489	14.2	106819	23.2
12	1684801	212	140536	242	1149178	341
14	OM	-	1313052	3563	OM	-
Three-tiered architecture						
3-3-2	7840	1.1	462	8.4	966	1.0
3-3-3	21952	3.6	286	60.4	2290	5.1
4-4-3	188272	42.3	-	OT	35524	103
Non-deterministic two-hop coloring protocol in undirected rings						
3	13824	10.3	442	15.4	1567	4.6
4	331776	511	8058	668	33415	160
5	OM	-	-	OT	661454	5718

B. Static Channel Diagrams

Donaldson and Miler design a fully automatic approach to detecting process symmetries for channel-based communication systems [13], [14]. Their approach also involves constructing a 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. 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 [35]. 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.

VII. Conclusion and Future Work

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 and detected symmetries save the majority of a state space to be explored.

A line of our future work is to design efficient algorithms for calculating representative states for automorphism groups that satisfy certain structural properties and are often used in practice. All existing symmetry detection approaches only work on one instance of a parameterized system at a time. We observe that, for a parameterized system, the distinctive features of symmetries are often determined by the essence of the system structure rather than concrete valuations of the parameters. So the other interesting line of future work is to provide a once-for-all solution of obtaining universal symmetries for the entire instances in a parameterized system.

Acknowledgements

We thank the anonymous reviewers for their invaluable comments. This work is supported by project "IDD11100102A/IDG31100105A" from Singapore University of Technology and Design and in part by NTU-NAP project: "Formal Verification on Cloud" from Nanyang Technological University.

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*, pages 643–644, 2008.

[3] E. Ashcroft and Z. Manna. Formalization of Properties of Parallel Programs. In *Machine Intelligence*, 1970.

[4] D. Bosnacki, D. Dams, and L. Holenderski. Symmetric Spin. In *SPIN*, pages 1–19, 2000.

[5] D. Bosnacki, L. Holenderski, and D. Dams. A Heuristic for Symmetry Reductions with Scalarsets. In *FME*, pages 518–533. 2001.

[6] Canepa, Davide and Potop-Butucaru, Maria Gradinariu. Stabilizing token schemes for population protocols. *CoRR*, 2008.

[7] E. Clarke, E. Emerson, S. Jha, and A. Sistla. Symmetry Reductions in Model Checking. In *CAV*, pages 147–158, 1998.

[8] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000.

[9] D. Cohen, P. Jeavons, C. Jefferson, K. E. Petrie, and B. M. Smith. Symmetry definitions for constraint satisfaction problems. *Constraints*, 11(2-3):115–137, 2006.

[10] M. Cohen, M. Dam, A. Lomuscio, and H. Qu. A Data Symmetry Reduction Technique for Temporal-epistemic Logic. In *ATVA*, pages 69–83, 2009.

[11] 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*, pages 451–490, 1991.

[12] P. T. Darga, K. A. Sakallah, and I. L. Markov. Faster Symmetry Discovery using Sparsity of Symmetries. In *DAC*, pages 149–154, 2008.

[13] A. F. Donaldson and A. Miller. Automatic Symmetry Detection for Model Checking Using Computational Group Theory. In *FM*, pages 631–631, 2005.

[14] A. F. Donaldson and A. Miller. Automatic Symmetry Detection for Promela. *Journal of Automated Reasoning*, pages 251–293, 2008.

[15] A. F. Donaldson and A. Miller. On the Constructive Orbit Problem. *Annals of Mathematics and Artificial Intelligence*, 57(1):1–35, 2009.

[16] I. Dotú and P. Van Hentenryck. Scheduling Social Golfers Locally. In *CPAIOR'05*, 2005.

[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 *OPODIS'06*, 2006.

[19] The GAP Group. *GAP – Groups, Algorithms, and Programming*, 2012.

[20] J. F. Groote, A. Ponse, and Y. S. Usenko. Linearization in Parallel pCRL. *Journal of Logic and Algebraic Programming*, 2001.

[21] M. Hendriks, G. Behrmann, K. Larsen, P. Niebert, and F. Vaandrager. Adding Symmetry Reduction to UPPAAL. In *FORMATS*, pages 46–59. 2004.

[22] C. N. Ip and D. L. Dill. Better Verification through Symmetry. *Formal Methods in System Design*, pages 41–75, 1996.

[23] 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.*, pages 33–66, 2010.

[24] M. M. Jaghoori, M. Sirjani, M. R. Mousavi, and A. Movaghar. Efficient Symmetry Reduction for an Actor-based model. In *ICDCIT'05*, pages 494–507, 2005.

[25] R. A. Krzysztof and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. 1991.

[26] B. D. MacArthur, R. J. Sánchez-García, and J. W. Anderson. Symmetry in Complex Networks. *Discrete Applied Mathematics*, pages 3525 – 3531, 2008.

[27] R. Milner. *Communication and Concurrency*. 1989.

[28] G. L. Peterson. Myths About the Mutual Exclusion Problem. *Information Processing Letters*, 1981.

[29] J.-F. Puget. Automatic Detection of Variable and Value Symmetries. In *CP*, pages 475–489, 2005.

[30] 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*, pages 133–166, 2000.

[31] C. Spermann and M. Leuschel. ProB Gets Nauty: Effective Symmetry Reduction for B and Z Models. In *TASE*, pages 15–22, 2008.

[32] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. In *CAV*, pages 709–714, 2009.

[33] T. Wahl. Adaptive Symmetry Reduction. In *CAV*, pages 393–405. Springer-Verlag, 2007.

[34] T. Wahl and A. Donaldson. Replication and Abstraction: Symmetry in Automated Formal Verification. *Symmetry*, pages 799–847, 2010.

[35] T. Wahl and V. D'Silva. A Lazy Approach to Symmetry Reduction. *Form. Asp. Comput.*, pages 713–733, 2010.

[36] Y. Xiao, M. Xiong, W. Wang, and H. Wang. Emergence of Symmetry in Complex Networks. *Phys. Rev. E*, page 066108, 2008.

[37] S. J. Zhang, J. Sun, C. Sun, Y. Liu, J. Ma, and J. S. Dong. Symmetry Detection for Model Checking. `http://www.comp.nus.edu.sg/~pat/detection/report`, 2013.