A Complete Method for Symmetry Reduction in Safety Verification

Duc-Hiep Chu* and Joxan Jaffar

National University of Singapore
hiepcd,joxan@comp.nus.edu.sg

Abstract. Symmetry reduction is a well-investigated technique to counter the state space explosion problem for reasoning about a concurrent system of similar processes. Here we present a general method for its application, restricted to verification of safety properties, but *without* any prior knowledge about global symmetry. We start by using a notion of *weak symmetry* which allows for more reduction than in previous notions of symmetry. This notion is relative to the target safety property. The key idea is to perform symmetric transformations on *state interpolation*, a concept which has been used widely for pruning in SMTand CEGAR. Our method naturally favors "quite symmetric" systems: more similarity among the processes leads to greater pruning of the tree. The main result is that the method is *complete* wrt. weak symmetry: it only considers states which are not weakly symmetric to an already encountered state.

1 Introduction

Symmetry reduction is a well-investigated technique to counter the state space explosion problem when dealing with concurrent systems whose processes are similar. In fact, traditional symmetry reduction techniques rely on an idealistic assumption that processes are *indistinguishable*. Because this assumption excludes many realistic systems, there is a recent trend [7, 4, 12, 14, 15] to consider systems of non-identical processes, where the processes are *sufficiently similar* that the original gains of symmetry reduction can still be accomplished. However, this necessitates an intricate step of detecting symmetry in the state exploration.

We start by considering an intuitive notion of symmetry, which is based on a standard adaptation of the notion of bisimilarity. We say two states s_1 and s_2 are symmetric if there is a "permutation" π such that $s_2 = \pi(s_1)$, and if each successor state of s_1 can be matched (via π) with a unique successor state of s_2 while at the same time each successor state of s_2 can be matched (via π^{-1}) with a unique successor state of s_1. In safety verification, we further require that s_1 is safe iff s_2 is safe.

We refer to this notion as *strong symmetry*. We mention that all recent works which deal with heterogeneous systems (processes are not necessarily identical)

* This author is supported by NUS Graduate School for Integrative Sciences and Engineering.

have the desire to capture this type of symmetry in the sense that they attempt, though not quite successfully, to consider only states which are *not* strongly symmetric to any already encountered state.

In this paper, we present a general approach to symmetry reduction for safety verification of a finite multi-process system, defined parametrically , without any prior knowledge about its global symmetry. In particular, we explicitly explore all possible interleavings of the concurrent transitions, while applying pruning on "symmetric" subtrees. We now introduce a new notion of symmetry: *weak symmetry*. Informally, this notion weakens the notion of permutation between states so that *the program counter* is the paramount factor in consideration of symmetry. In contrast, values of program variables are used in consideration of strong symmetry. The main result is that our approach is *complete* wrt. weak symmetry: it only considers states which are not weakly symmetric to an already encountered state.

More specifically, we address the state explosion problem by employing *symbolic learning* on the search tree of all possible interleavings. Specifically, our work is based on the concept of interpolation. Here, interpolation is essentially a form of *backward learning* where a completed search of a *safe* subtree is then formulated as a recipe for pruning (every state/node is a root associated to some subtree). There are two key ideas regarding our learning technique: First, each learned recipe for a node not only can be used to prune other nodes having the same future (same program point), but also can be *transfered* to prune nodes that having *symmetric* futures (symmetric program points). Second, each recipe discovered by a node will be conveyed back to its ancestors, which gives rise to pruning of *larger* subtree. Another important distinction is that our method learns *symbolically* with respect to the safety property and the interleavings. In Section 5, we will confirm the effectiveness of our method experimentally on some classic benchmarks.

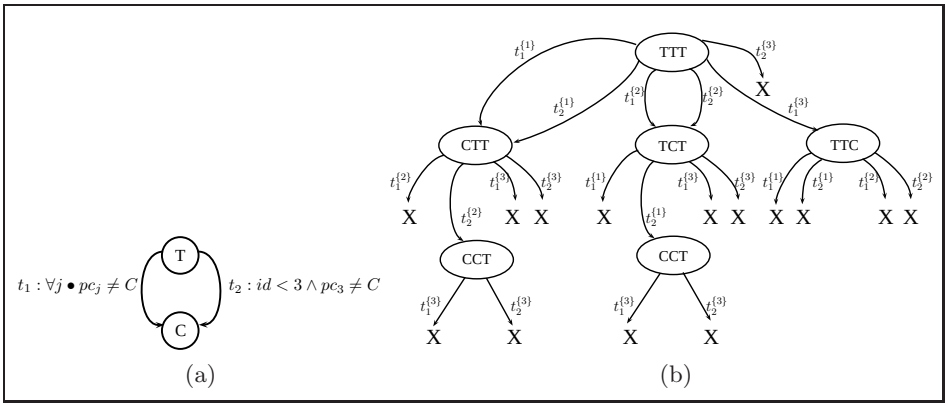

Fig. 1: (a) Modified 3-process reader-writer (b) Full interleaving tree

We conclude this subsection with two examples in order to demonstrate strong and weak symmetry. First we borrow with modification from [14, 15] wherein are two "reader" processes (indices 1, 2) and one "writer" process (in-

dex 3). We denote by C and T the local process states which indicate entering the critical section and in a "trying" state, respectively. See Figure 1(a). Note that pc_j is the local control location of process j and for each process, id is its *process identifier*. These concepts will be defined more formally in Section 2.

For each process, there are two transitions from T to C. The first, t_1, is executable by any process provided that no process is currently in its critical section ($\forall j \bullet pc_j \neq \mathsf{C}$). The second, t_2, is however available to only readers ($id < 3$), and the writer must be in a non-critical local state $pc_3 \neq \mathsf{C}$. This example shows symmetry between the reader processes, but because of their priority over the writer, we do not have "full" symmetry [14].

Figure 1(b) shows the full interleaving tree. Transitions are labelled with superscripts to indicate the process to which that transition is associated. *Infeasible* transitions are arrows ending with crosses. Note that nodes CTT and TCT are strongly symmetric, but neither is strongly symmetric with TTC.

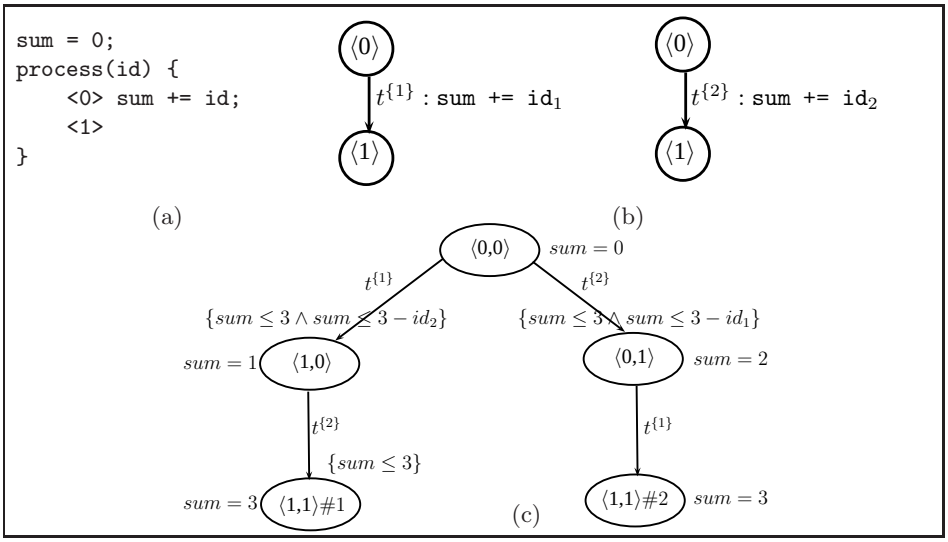

Fig. 2: (a) Sum-of-ids system (b) Its 2-process concretization (c) Full interleaving tree

Our second example is the system in Figure 2(a). Initially, the shared variable sum is set to 0. Each process increments sum by the amount of its process identifier, namely id. The local transition systems for process 1 and process 2 are shown in Figure 2(b). The full interleaving tree is shown in Figure 2(c).

Let π be the function swapping the indices of the two processes. We can see that the subtrees rooted at states $\langle\langle 1, 0\rangle; sum = 1\rangle$ and $\langle\langle 0, 1\rangle; sum = 2\rangle$ share the same shape. However, due to the difference in the value of shared variable sum, strong symmetry does not apply (in fact, any top-down technique, such as [14, 15, 12], cannot avoid exploring the subtree rooted at $\langle\langle 0, 1\rangle; sum = 2\rangle$, even if the subtree rooted at $\langle\langle 1, 0\rangle; sum = 1\rangle$ has been traversed and proved to be safe).

There is however a *weaker* notion of symmetry that does apply. We explain this by outlining our own approach, whose key feature is the computation of an

interpolant [10] for a node, by a process of backward learning. Informally, this interpolant represents a *generalization* of the values of the variables such that the traversed tree has a similar transition structure, and also remains safe. In the example, we require the safety property $\psi \equiv sum \leq 3$ at every state, and interpolants are shown as formulas inside curly brackets.

Using precondition propagation, the interpolant for state $\langle\langle 1,1\rangle; sum = 3\rangle$ is computed as $sum \leq 3$, and the interpolant for state $\langle\langle 1,0\rangle; sum = 1\rangle$ is computed as $\phi_{\langle 1,0\rangle} \equiv sum \leq 3 \wedge sum \leq 3 - id_2$. Using this, we can infer that $\phi_{\langle 0,1\rangle} \equiv sum \leq 3 \wedge sum \leq 3 - id_1$ (obtained by applying π on $\phi_{\langle 1,0\rangle}$) is a sound interpolant for program point $\langle 0,1\rangle$. As $\langle\langle 0,1\rangle; sum = 2\rangle \models \phi_{\langle 0,1\rangle}$, the subtree rooted at $\langle\langle 0,1\rangle; sum = 2\rangle$ can be pruned.

1.1 Related Work

Symmetry reduction has been extensively studied, e.g. [5, 2, 8, 6]. Traditionally, symmetry is defined as a transition-preserving equivalence, where an automorphism π, other than being a bijection on the reachable states, also satisfies that (s, s') is a transition iff $(\pi(s), \pi(s'))$ is. There, this type of symmetry reduction is enforced by *unrealistic* assumptions about indistinguishable processes. As a result, it does not apply to many systems in practice.

One of the first to apply symmetry reduction strategies to "approximately symmetric" systems is [7], defining notions of *near* and *rough* symmetry. Near and rough symmetry is then generalized in [4] to *virtual symmetry*, which still makes use of the concept of bisimilarity for symmetry reduction. Though bisimilarity enables full μ-calculus model checking, the main limitation of these approaches is that they exclude many systems, where bisimilarity to the quotient is simply not attainable. Also, these approaches work only for the verification of *fully symmetric properties*. No implementation is provided.

The work [12] allows arbitrary divergence from symmetry, and accounts for this divergence initially by conservative optimism, namely in the form of symmetric "super-structure". Specifically, transitions are added to the structure to achieve symmetry. A *guarded annotated quotient* (GAQ) is then obtained from the super-structure, where added transitions are marked. This approach works well for programs with syntacticly specified static transition priority. However, in general, the GAQ needs to be *unwound* frequently to compensate for the loss in precision (false positive due to added transitions). This might affect the running time significantly as this method might need to consider many combinations of transitions which do not belong to the original structure.

In comparison with our technique, this method has a clear advantage that it can handle arbitrary CTL* property. Nevertheless, our technique is more efficient both in space and time. Our technique is required to store an interpolant for each non-subsumed state, whereas in [12], a quotient edge might require multiple annotations. Furthermore, ours does not require a costly preprocessing of the program text to come up with a symmetric super-structure. Also, extending [12] to symbolic model checking does not seem possible.

The most *recent* state-of-the-art regarding symmetry reduction, and also closest to our spirit, is the *lazy approach* proposed by [14, 15]. Here only safety verification is considered. This approach does not assume any prior knowledge about (global) symmetry. Indeed, they initially and lazily ignore the potential lack of symmetry. During the exploration, each encountered state is annotated with information about how symmetry is violated along the path leading to it. The idea is that more similarity between component processes entails more compression is achieved.

In summary, the two main related works which are not restricted *a priori* on global symmetry are [12] and [14]. That is, these works allow the system to use process identifiers and therefore do not restrict the behaviors of individual processes. This is not the case with the previously mentioned works.

These works, [12] and [14], can be categorized as *top-down* techniques. Fundamentally, they look at the syntactic similarities between processes, and then come up with a reduced structure where symmetric states/nodes are merged into one abstract node. When model checking is performed, an abstract node might be concretized into a number of concrete nodes and each is checked one by one ([12] handles that by unwinding). For them, two symmetric parental nodes are not guaranteed to have correspondingly symmetric children. For us, by backward learning, we *ensure* that is the case. Consequently, and most importantly, they do not exponentially improve the runtime, only compress the state space.

Consider again the first example above (Figure 1). A top-down approach will consider TTC as a "potentially" symmetric state of CTT, and all three states CTT, TCT, and TTC are merged into one abstract state. While having compaction, it is not the case that the search space traversed is of this compact size. As a non-symmetric state (TTC) is merged with other mutually symmetric states (CTT and TCT), in generating the successor abstract state, the parent abstract state is required to be concretized and both transitions $t_2^{\{2\}}$ (emanating from CTT) and transition $t_2^{\{1\}}$ (emanating from TCT) are considered (in fact, infeasible transition $t_2^{\{3\}}$ is also considered). In summary, compaction may not lead to any reduction in the search space.

We finally mention that we consider only safety properties because we wish to employ abstraction in the search process. And it is precisely a judicious use of abstraction that enables us to obtain more pruning in comparison with prior techniques. We prove this in principle by showing that we are *complete* wrt. weak symmetry, and we demonstrate this experimentally on some classic benchmarks.

2 Preliminaries

We consider a parametrically defined n-process system, where n is fixed. In accordance with standard practice in works on symmetry, we assume that the domain of discourse of the program variables is *finite* so as to guarantee termination of the search process of the underlying transition system. Infinite domains may be accomodated by some use of abstraction, as we show in one benchmark example below.

We employ the usual syntax of a deterministic imperative language, and communication occurs via shared variables. Each process has a unique and predetermined *process identifier*, and this is denoted parametrically in the system by the special variable `id`. For presentation purpose, the concrete value of `id` for each individual process ranges from 1 to n. We note that the variable `id` cannot be changed. Even though the processes are defined by one parameterized system, their dynamic behaviors can be arbitrarily different. This would depend on how `id` is expressed in the parameterized system. Finally, we also allow a blocking primitive `await(b) s;` where `b` is a boolean expression and `s` is an *optional* program statement.

Consider the 2-process parameterized system in Figure 3(a). Note the (local) program points in angle brackets. Figure 3(b) "concretizes" the processes explicitly. Note the use in the first process of the variable id_1 which is not writable in the process, and whose value is 1. Similarly for id_2 in the other process.

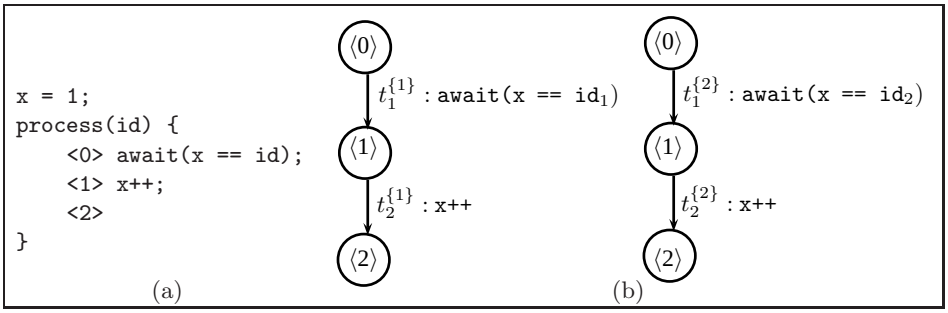

Fig. 3: (a) A parameterized system (b) Its 2-process concretization

In general, where P_i $(1 \leq i \leq n)$ is a process, let V_i be its local variables and V_{shared} be the shared variables of entire system. We note here that V_i does not include the special local variables which represent the process identifiers. Let $pc_i \in V_i$ be a special variable represent the local program counter, and the tuple $\langle pc_1, pc_2 \cdots, pc_n \rangle$ represent the global program point. Let $State$ be the set of all global states of the given program where $s_0 \in State$ is the initial state. A state $s \in State$ comprises of three parts: its *program point* $\texttt{pc}(s)$, which is a tuple of local program counters, its *valuation* over the program variables $\texttt{val}(s)$, and its valuation over the process identifiers `pids`. In other words, we denote a state s by $\langle \texttt{pc}(s); \texttt{val}(s); \texttt{pids} \rangle$. Note that all states from the same parameterized system share the same valuation of the individual process identifiers. Therefore, when the context is clear, we omit the valuation `pids` of a state.

We consider the *transitions* of states induced by the program. A transition $t^{\{i\}}$ pertains to some process P_i. It transfers process P_i from control location $\texttt{l}_1$ to $\texttt{l}_2$. In general, the application of $t^{\{i\}}$ is guarded by some condition `cond` (`cond` might be just `true`). At some state $s \in State$, when the i^{th} component of $\texttt{pc}(s)$, namely $\texttt{pc}(s)[i]$, equals $\texttt{l}_1$, we say that $t^{\{i\}}$ can be *scheduled* at s. And when the valuation $\texttt{val}(s); \texttt{pids}$ satisifies the guard `cond`, denoted by $\texttt{val}(s); \texttt{pids} \models \texttt{cond}$, we say that $t^{\{i\}}$ is *enabled* at s. Furthermore, we call the enabling condition of $t^{\{i\}}$ the formula: $(\texttt{pc}(s)[i] == \texttt{l}_1) \wedge \texttt{cond}$. For each state s, let $Scheduled(s)$ and

Enabled(s) denote the set of transitions which respectively can be scheduled at s and are enabled at s. Without further ado, we assume that the effect of applying an enabled transition $t^{\{i\}}$ on a state s to arrive at state s' is well-understood. This is denoted as $s \xrightarrow{t^{\{i\}}} s'$.

Again consider in Figure 3 with two processes P_1 and P_2 with variables $\texttt{id}_1 = 1$ and $\texttt{id}_2 = 2$ respectively. In the system, it is specified parametrically that each process awaits for `x == id`. In P_1, this is interpreted as $\texttt{await(x == id}_1\texttt{)}$ while P_2, this is interpreted as $\texttt{await(x == id}_2\texttt{)}$. Each process has 2 transitions: the first transfers it from control location $\langle 0 \rangle$ to $\langle 1 \rangle$, whereas the second transfers it from control location $\langle 1 \rangle$ to $\langle 2 \rangle$. Initially we have $x = 1$, i.e. the initial state s_0 is $\langle \langle 0, 0 \rangle; x = 1; id_1 = 1, id_2 = 2 \rangle$. We note that at s_0, both $t_1^{\{1\}}$ and $t_1^{\{2\}}$ can be scheduled. However, among them, only $t_1^{\{1\}}$ is enabled. By taking transition $t_1^{\{1\}}$, P_1 moves from control location $\langle 0 \rangle$ to $\langle 1 \rangle$, and the whole system moves from state $\langle \langle 0, 0 \rangle; x = 1; id_1 = 1, id_2 = 2 \rangle$ to state $\langle \langle 1, 0 \rangle; x = 1; id_1 = 1, id_2 = 2 \rangle$. We note that here the transition $t_1^{\{2\}}$ is still disabled. From now on, let us omit the valuation of process identifiers. The whole system then takes the transition $t_2^{\{1\}}$ and moves from state $\langle \langle 1, 0 \rangle; x = 1 \rangle$ to state $\langle \langle 2, 0 \rangle; x = 2 \rangle$. Now, $t_1^{\{2\}}$ becomes enabled. Subsequently, the system takes $t_1^{\{2\}}$ and $t_2^{\{2\}}$ to move to state $\langle \langle 2, 1 \rangle; x = 1 \rangle$ and finally to state $\langle \langle 2, 2 \rangle; x = 3 \rangle$.

Definition 1 (Safety). *We say a given concurrent system is* safe *wrt. a safety property ψ if $\forall s \in State \bullet s$ is reachable from s_0 implies $s \models \psi$.*

2.1 Symmetry

Given an n-process system, let $\mathcal{I} = [1 \cdots n]$ denote its *indices*, to be thought of as process identifiers. We write $Sym\ \mathcal{I}$ to denote the set of all permutations π on index set $\mathcal{I}$. Let Id be the identity permutation and π^{-1} the inverse of π.

For an indexed object b, such as a program point, a variable, a transition, valuation of program variables, or a formula, whose definition depends on $\mathcal{I}$, we can define the notion of permutation π acting on b, by simultaneously replacing each occurrence of index $i \in \mathcal{I}$ by $\pi(i)$ in b to get the result of $\pi(b)$.

Example 1. Consider the system in Figure 3(b). Let the permutation π swap the two indices $(1 \mapsto 2, 2 \mapsto 1)$. Applying π to the valuation $x = 1$ gives us $\pi(x = 1) \equiv x = 1$, as x is a shared variable. Applying π to the formula $x = id_1 \wedge id_1 = 1$ gives us $\pi(x = id_1 \wedge id_1 = 1) \equiv (x = id_2 \wedge id_2 = 1)$. On the other hand, applying π to the transition $t_1^{\{1\}} \equiv \texttt{await(x = id}_1\texttt{)}$ will result in $\pi(t_1^{\{1\}}) \equiv t_1^{\{2\}} \equiv \texttt{await(x = id}_2\texttt{)}$.

Definition 2. *For $\pi \in Sym\ \mathcal{I}$ and state $s \in State$, $s \equiv \langle \texttt{pc}(s); \texttt{val}(s); \texttt{pids} \rangle$, the application of π on s is defined as $\langle \pi(\texttt{pc}(s)); \pi(\texttt{val}(s)); \texttt{pids} \rangle$,*

In other words, permutations do not affect the valuation of process identifiers.

Example 2. Consider again the system in Figure 3(b). Assume the π is the permutation swapping the 2 indices $(1 \mapsto 2, 2 \mapsto 1)$. We then can have $\pi(\langle\langle 1, 0\rangle; x = 1; id_1 = 1, id_2 = 2\rangle) \equiv \langle\langle 0, 1\rangle; x = 1; id_1 = 1, id_2 = 2\rangle$. Please note that while π has no effect on shared variable x and valuation of process identifiers id_1, id_2, it does permute the local program points.

Definition 3. *For $\pi \in Sym\ \mathcal{I}$, a safety property ψ is said to be* symmetric *wrt. π if $\psi \equiv \pi(\psi)$.*

We next present a traditional notion of symmetry.

Definition 4 (Strong Symmetry). *For $\pi \in Sym\ \mathcal{I}$, and a safety property ψ, for $s, s' \in State$, we say that s is strongly π-similar to s' wrt. ψ, denoted by $s \overset{\pi,\psi}{\approx} s'$ if ψ is symmetric wrt. π and the following conditions hold:*
- $\pi(s) = s'$
- *for each transition t such that $s \xrightarrow{t} d$ we have $s' \xrightarrow{\pi(t)} d'$ and $d \overset{\pi,\psi}{\approx} d'$*
- *for each transition t' such that $s' \xrightarrow{t'} d'$ we have $s \xrightarrow{\pi^{-1}(t')} d$ and $d \overset{\pi,\psi}{\approx} d'$.*

One of the strengths of this paper is to allow symmetry by *disregarding* the values of the program variables.

Definition 5 (Weak Symmetry). *For $\pi \in Sym\ \mathcal{I}$, and a safety property ψ, for $s, s' \in State$, we say that s is weakly π-similar to s' wrt. ψ, denoted by $s \overset{\pi,\psi}{\sim} s'$ if ψ is symmetric wrt. π and the following conditions hold:*
- $\pi(\mathit{pc}(s)) = \mathit{pc}(s')$
- *$s \models \psi$ iff $s' \models \pi(\psi)$*
- *for each transition t such that $s \xrightarrow{t} d$ we have $s' \xrightarrow{\pi(t)} d'$ and $d \overset{\pi,\psi}{\sim} d'$*
- *for each transition t' such that $s' \xrightarrow{t'} d'$ we have $s \xrightarrow{\pi^{-1}(t')} d$ and $d \overset{\pi,\psi}{\sim} d'$.*

We note here that, from now on, unless otherwise mentioned, symmetry means *weak* symmetry while π-similar means *weakly* π-similar. Also, it trivially follows that if s is π-similar to s' then s' is π^{-1}-similar to s. Consequently, if s is symmetric with s', then s' is symmetric with s too.

2.2 State Interpolation

State-based interpolation was first described in [10] for finite transition systems. The essential idea was to prune the search space of symbolic execution, informally described as follows. Symbolic execution is usually depicted as a tree rooted at the initial state s_0 and for each state s_i therein, the descendants are just the states obtainable by extending s_i with an enabled transition. Consider one particular feasible path represented in the tree: $s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2 \cdots \xrightarrow{t_m} s_m$. We adapt the usual notion of *program point*, characterizing a point in the reachability tree in terms of all the remaining possible transitions. Now, this particular path is *safe* wrt. to safety property ψ if for all i, $0 \leq i \leq m$, we have $s_i \models \psi$. A

(state) interpolant at program point j, $0 \leq j \leq m$ is simply a set of states S_j containing s_j such that for any state $s'_j \in S_j$, $s'_j \xrightarrow{t_{j+1}} s'_{j+1} \xrightarrow{t_{j+2}} s'_{j+2} \cdots \xrightarrow{t_m} s'_m$, it is also the case that for all i, $j \leq i \leq m$, we have $s'_i \models \psi$. This interpolant was constructed at point j due to the one path. Consider now all paths from s_0 and with prefix $t_1, \cdots, t_{j-1}$. Compute each of their interpolants. Finally, we say that the interpolant for the subtree of paths just considered is simply the intersection of all the individual interpolants. This notion of interpolant for a subtree provides a notion of *subsumption* because we can now prune a subtree in case the root of this subtree are within the interpolant computed for some previously encountered subtree of the same program point.

Definition 6 (Safe Root). *Let s_i be a state which is reachable from the initial state s_0, we say that s_i is a* safe root*, denoted by $\triangle(s_i)$, if for all state s'_i reachable from s_i, s_i is safe.*

Definition 7 (State Coverage). *Let s_i and s_j be two states which are reachable from the initial state s_0 such that $\mathtt{pc}(s_i) \equiv \mathtt{pc}(s_j)$. We say that s_i covers s_j, denoted by $s_i \succeq s_j$, if $\triangle(s_i)$ implies $\triangle(s_j)$.*

During the traversal of the reachability tree, if we detect that $s_i \succeq s_j$ while s_i has been proved to be a safe root, the traversal of the subtree rooted at s_j can be avoided. We thus reduce the search space.

Definition 8 (Sound Interpolant). *Let $\mathtt{pp}$ be a global program point. We say a formula ϕ is a* sound interpolant *for $\mathtt{pp}$ if for all state s reachable from the initial state s_0, $\mathtt{pc}(s) \equiv \mathtt{pp} \wedge s \models \phi$ implies that s is a safe root.*

In practice, in order to determine state coverage, during the exploration of subtree rooted at s_i we compute an interpolant of $\mathtt{pc}(s_i)$, denoted as $\mathtt{SI}(\mathtt{pc}(s_i), \psi)$, where ψ is the target safety property. Note that trivially, we should have $s_i \models \mathtt{SI}(\mathtt{pc}(s_i), \psi)$. We assume that this condition is always ensured by any implementation of our state-based interpolation. Furthermore, $\mathtt{SI}(\mathtt{pc}(s_i), \psi)$ ensures that $\forall s_j \in \mathit{State} \bullet \mathtt{pc}(s_j) \equiv \mathtt{pc}(s_i) \wedge s_j \models \mathtt{SI}(s_i, \psi)$, then for all $t \in \mathit{Scheduled}(s_i)$ [1], the two following conditions must be satisfied:

- if t was disabled at s_i, it also must be disabled at s_j
- if t was enabled at s_j (by the above condition, it must be enabled at s_i) and $s_j \xrightarrow{t} s'_j$ and $s_i \xrightarrow{t} s'_i$, then s'_i must cover s'_j.

This observation enables us to determine the coverage relation as the form of backward learning in a recursive manner. Our symmetry reduction algorithm presented in Section 4 will implement this idea of state interpolation.

3 Motivating Examples

Figure 4 shows a parameterized system and its 2-process concretization. The shared array `x` contains 2 elements, initially 0. For convenience, we assume that

[1] Since $\mathtt{pc}(s_j) \equiv \mathtt{pc}(s_i)$, we have $\mathit{Scheduled}(s_j) \equiv \mathit{Scheduled}(s_i)$.

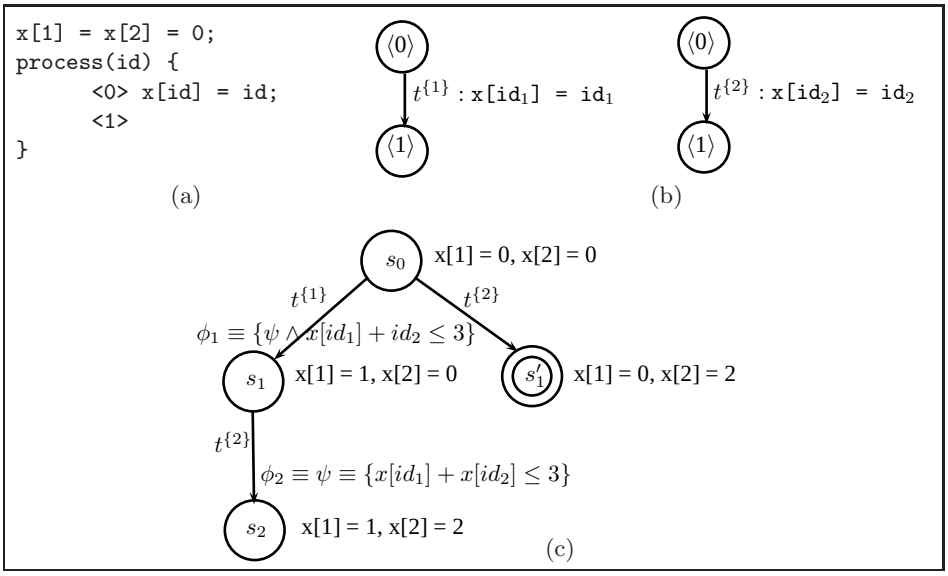

Fig. 4: (a) An example (b) Its 2-process concretization (c) Traversed tree

array index starts from 1. Process 1 assigns $\texttt{id}_1$ (whose value is 1) to $\texttt{x}[1]$ while process 2 assigns $\texttt{id}_2$ (2) to $\texttt{x}[2]$.

Consider the safety property $\psi \equiv x[1] + x[2] \leq 3$, interpreted as $\psi \equiv x[id_1] + x[id_2] \leq 3$. The reachability tree explored is in Figure 4(c). Circles are used to denote states, while double-boundary circles denote subsumed/pruned states.

From the initial state $s_0 \equiv \langle\langle 0, 0\rangle; x[1] = 0, x[2] = 0; id_1 = 1, id_2 = 2\rangle$ process 1 progresses first and moves the system to the state $s_1 \equiv \langle\langle 1, 0\rangle; x[1] = 1, x[2] = 0; id_1 = 1, id_2 = 2\rangle$. From s_1, process 2 now progresses and moves the system to the state $s_2 \equiv \langle\langle 1, 1\rangle; x[1] = 1, x[2] = 2; id_1 = 1, id_2 = 2\rangle$. Note that s_0, s_1, and s_2 are all safe wrt. ψ. As there is no transition emanating from s_2, the interpolant for s_2 is computed as $\phi_2 \equiv \psi \equiv x[id_1] + x[id_2] \leq 3$. The pair $\langle\langle 1, 1\rangle; \phi_2\rangle$ is memoized. The interpolant for s_1 can be computed as a conjunction of two formulas. One concerns the safety of s_1 itself, and the other concerns the safety of the successor state from $t^{\{2\}}$. In other words, we can have $\phi_1 \equiv \psi \wedge \mathsf{pre}(x[id_2] = id_2; \psi)$, where $\mathsf{pre}(t; \phi)$ denotes a precondition wrt. to the program transition t and the postcondition ϕ. Consequently, we can have $\phi_1 \equiv \psi \wedge x[id_1] + id_2 \leq 3$. The pair $\langle\langle 1, 0\rangle; \phi_1\rangle$ is memoized.

Now we arrive at state $s'_1 \equiv \langle\langle 0, 1\rangle; x[1] = 0, x[2] = 2; id_1 = 1, id_2 = 2\rangle$. This is indeed a symmetric image of state s_1 which we have explored and proved to be safe before. Here, we discover the permutation π to transform the program point $\langle 1, 0\rangle$ to program point $\langle 1, 0\rangle$. Clearly π simply swaps the two indices. We also observe that the safety property ψ is symmetric wrt. this π, i.e. $\pi(\psi) \equiv \psi$ (ψ is *invariant* wrt. π). In the next step, we check whether $\texttt{val}(s'_1)$ conjoined with $\texttt{pids}$ implies the *transformed interpolant* $\pi(\phi_1)$. We have $\pi(\phi_1) \equiv \pi(x[id_1] + x[id_2] \leq 3 \wedge x[id_1] + id_2 \leq 3 \equiv x[id_2] + x[id_1] \leq 3 \wedge x[id_2] + id_1 \leq 3$. As

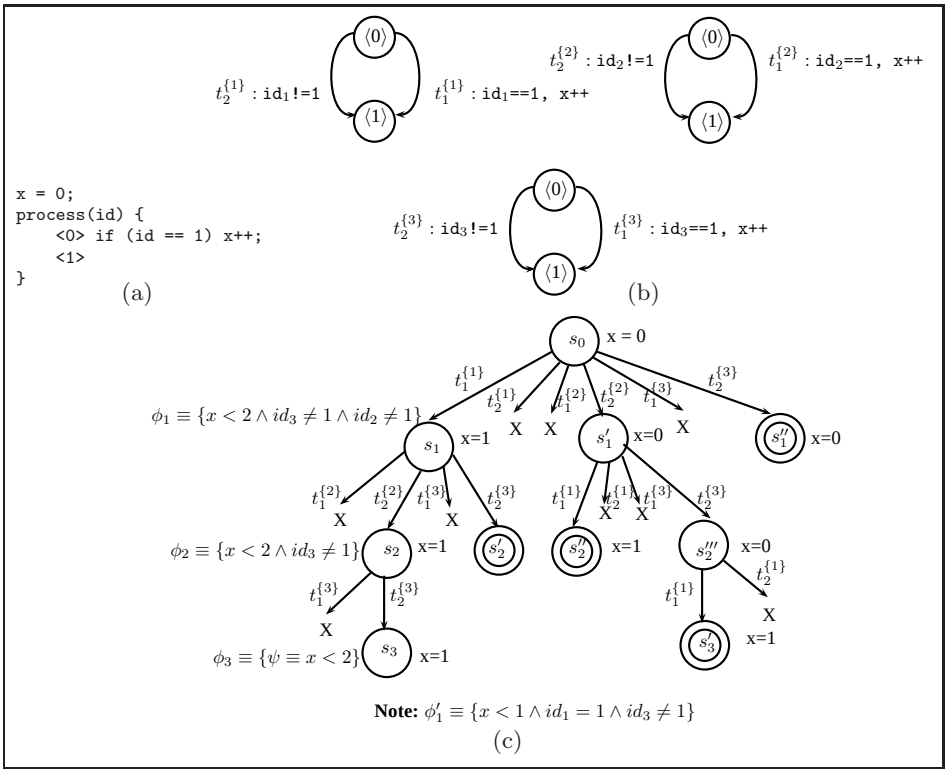

Fig. 5: (a) An example (b) Its 3-process concretization (c) Traversed tree

$\texttt{val}(s_1'); \texttt{pids} \models x[id_2] + x[id_1] \leq 3 \wedge x[id_2] + id_1 \leq 3$, we do not need to explore s_1' any further. In other words, the subtree rooted at s_1' is pruned.

Another example is Figure 5. We are interested in safety property $\psi \equiv x < 2$. As x is a shared variable, ψ is symmetric wrt. all possible permutations.

The reachability tree is depicted in Figure 5(c). From the initials state s_0 we arrive at states s_1, s_2, and s_3, where:

$s_0 \equiv \langle\langle 0,0,0\rangle; x = 0; id_1 = 1, id_2 = 2, id_3 = 3\rangle$
$s_1 \equiv \langle\langle 1,0,0\rangle; x = 1; id_1 = 1, id_2 = 2, id_3 = 3\rangle$
$s_2 \equiv \langle\langle 1,1,0\rangle; x = 1; id_1 = 1, id_2 = 2, id_3 = 3\rangle$
$s_3 \equiv \langle\langle 1,1,1\rangle; x = 1; id_1 = 1, id_2 = 2, id_3 = 3\rangle$.

At s_3 we compute its interpolant $\phi_3 \equiv \psi \equiv x < 2$. In a similar manner as before, we compute the interpolant for s_2, which is $\phi_2 \equiv x < 2 \wedge id_3 \neq 1$. When we are at state $s_2' \equiv \langle\langle 1,0,1\rangle; x = 1; id_1 = 1, id_2 = 2, id_3 = 3\rangle$, we look for a permutation π_1 such that $\pi_1(\langle 1,1,0\rangle) = \langle 1,0,1\rangle$. Clearly we can have π_1 as the permutation which fixes the first index and swaps the last 2 indices. Moreover, $\texttt{val}(s_2'); \texttt{pids} \equiv x = 1; id_1 = 1, id_2 = 2, id_3 = 3 \models \pi_1(\phi_2) \equiv x < 2 \wedge id_2 \neq 1$. Therefore, s_2' is pruned.

Similarly, the interpolant for s_1 is computed as $\phi_1 \equiv x < 2 \wedge id_2 \neq 1 \wedge id_3 \neq 1$. When at state $s_1' \equiv \langle\langle 0,1,0\rangle; x = 0; id_1 = 1, id_2 = 2, id_3 = 3\rangle$, we look for a permutation π_2 such that $\pi_2(\langle 1,0,0\rangle) = \langle 0,1,0\rangle$. Clearly we can have π_2 as

the permutation which fixes the third index and swaps the first two indices. However, $\texttt{val}(s'_1);\texttt{pids} \equiv x = 0; id_1 = 1, id_2 = 2, id_3 = 3 \not\models \pi_2(\phi_1) \equiv x < 2 \wedge id_1 \neq 1 \wedge id_3 \neq 1$. Thus the subtree rooted at s'_1 cannot be pruned and it requires further exploration. After s'_1 has been traversed, the intepolant for s'_1 is computed as $\phi'_1 \equiv x < 1 \wedge id_1 = 1 \wedge id_3 \neq 1$. Next we arrive at $s''_1 \equiv \langle\langle 0, 0, 1\rangle; x = 0; id_1 = 1, id_2 = 2, id_3 = 3\rangle$. We can find a permutation π_3 which fixes the first index and swaps the last 2 indices ($\pi_3 \equiv \pi_1$). We have $\pi_3(\langle 0, 1, 0\rangle) = \langle 0, 0, 1\rangle$. Also $\texttt{val}(s''_1);\texttt{pids} \equiv x = 0; id_1 = 1, id_2 = 2, id_3 = 3 \models \pi_3(\phi'_1) \equiv x < 1 \wedge id_1 = 1 \wedge id_2 \neq 1$. As a result, we can avoid considering the subtree rooted at s''_1.

In the two examples above, we have shown how the concept of interpolation can help capture the shape of a subtree. More importantly, computed interpolants can be transformed in order to detect the symmetry as well as the non-symmetry (mainly due to the use of `id`) between candidate subtrees.

4 Symmetry Reduction Algorithm

```
⟨1⟩  Intially : Explore(s_0, ∅)
function Explore(s, h)
⟨2⟩  if s ⊭ ψ then Report Error and TERMINATE
⟨3⟩  if ∃π • π(ψ) ≡ ψ ∧ ∃pp • pc(s) ≡ π(pp) ∧ ∃φ • memoed(pp, φ) ∧ s ⊨ π(φ) then
         return π(φ)
⟨4⟩  if s ∈ h                                              /* We hit a cycle */
⟨5⟩      let θ be the cyclic path
⟨6⟩      Assert(Cyclic(s, θ))
⟨7⟩      return true                        /* Initial value for fix-point computation */
     else
⟨8⟩      h ← h ∪ {s}
     endif
⟨9⟩  φ ← ψ
⟨10⟩ foreach t ∈ Scheduled(s) do
⟨11⟩     if t ∈ Enabled(s)
⟨12⟩         s' ← successor of s after t                          /* Execute t */
⟨13⟩         φ' ← Explore(s', h)
⟨14⟩         φ ← φ ∧ pre(t; φ')
         else
⟨15⟩         φ ← φ ∧ pre(t; false)
         endif
⟨16⟩ endfor
⟨17⟩ let Θ be {θ | Cyclic(s, θ)}
⟨18⟩ if Θ ≠ ∅ then φ ← FIX-POINT(s, Θ, φ)
         /* s is a looping point, so we ensure φ is an invariant along the paths Θ */
⟨19⟩ Retractall(Cyclic(s, θ))
⟨20⟩ h ← h \ {s}
⟨21⟩ memo(pc(s), φ) and return φ
end function
```

Fig. 6: Symmetry Reduction Algorithm (DFS)

Our algorithm, presented in Figure 6, naturally performs a depth first search of the interleaving tree. It assumes the safety property to be known as ψ. Initially, we explore the initial state s_0 with an empty *history*. During the search process, the function Explore will be recursively called. Note that termination is ensured due to the finite domain of discourse.

Base Cases: The first base case is when the current state does not conform to the safety property ψ (line 2). We then immediately report an error and terminate. The second base case applies when the current state (subtree) has a symmetric image (subtree) which has already been traversed and proved to be safe before (line 3). We have well exemplified such scenarios in previous sections.

The third base case requires some elaboration. Using the history h, we detect a cycle (line 4). Specifically, there is a cyclic path θ from s back to s. We note this down and return true. Later, after the descendants of s have been traversed, we require a fix-point computation of the interpolant for s, as shown in line 17-18. The function FIX-POINT computes an invariant interpolant for s, wrt. the initial value ϕ and the set of cyclic paths Θ. Essentially, this function involves computing, for each cyclic path, a *path invariant*. Such a computation is performed backwards, using a previously computed invariant at the bottom of the cyclic path, and then extracting a new invariant for s. Then each computed path invariant is fed into other paths in order to compute a new invariant. The process terminates at a fix-point. Termination is guaranteed because of monotonicity of the path invariant computation and the fact that there are only finitely many possible invariants (the state s itself is an invariant). Finally, the interpolant for each state appearing in these cyclic paths are now updated appropriately. This is in light of now having an invariant for all of them simultaneously.

We remark here that this fix-point task, though seemingly complicated, is in fact routine. We refer interested readers to [9] for more details regarding this matter. We also remark that for many concurrent protocols, where involved operations are mainly "set" and "re-set" operations, a fix-point is achieved just after one iteration.

Recursive Traversal and Computing the Interpolants: Our algorithm recursively explores the successors of the current state by the recursive call in line 13. The interpolant ϕ for the current state is computed as from line 9-18. As mentioned above, cyclic paths are handled in line 17-18. The operation $\mathsf{pre}(t; \overline{\phi})$ denotes the precondition computation wrt. the program transition t and the postcondition $\overline{\phi}$. In practice, we implement this as an approximation of the weakest precondition computation [3].

Theorem 1 (Soundness). *Our symmetry reduction algorithm is sound.*

Here, by soundness, we mean that all pruning performed in line 3 will not affect the verification result.

*Proof (**Outline**).* Let the triple $\{\phi\}\ \langle\langle pc_1, pc_2, \cdots, pc_n\rangle; P_1||P_2||\cdots||P_n\rangle\ \{\psi\}$ denote the fact that ϕ is a *sound* interpolant for program point $\langle pc_1, pc_2, \cdots, pc_n\rangle$ wrt. the safety property ψ and the concurrent system $P_1||P_2||\cdots||P_n$. Due to

space limit, we will not prove that our interpolant computation (line 9-18) is a sound computation. Instead, we refer interested readers to [10, 9]. Let us assume that the soundness of that triple is witnessed by a proof $\mathcal{P}$. By consistently renaming $\mathcal{P}$ with a renaming function $\pi \in Sym\ \mathcal{I}$, we can derive a new *sound* fact (i.e. a proof), which is:

$\{\pi(\phi)\}\ \pi(\langle\langle pc_1, pc_2, \cdots, pc_n\rangle; P_1||P_2||\cdots||P_n\rangle)\ \{\pi(\psi)\} \equiv$
$\{\pi(\phi)\}\ \langle\langle pc_{\pi(1)}, pc_{\pi(2)}, \cdots, pc_{\pi(n)}\rangle; P_{\pi(1)}||P_{\pi(2)}||\cdots||P_{\pi(n)}\rangle\ \{\pi(\psi)\}$

Since $P_1, P_2, \cdots P_n$ come from the same parameterized system and π is a bijection on $\mathcal{I}$, we have:

$P_{\pi(1)}||P_{\pi(2)}||\cdots||P_{\pi(n)} \equiv P_1||P_2||\cdots||P_n$

Therefore, $\{\pi(\phi)\}\ \langle\langle pc_{\pi(1)}, pc_{\pi(2)}, \cdots, pc_{\pi(n)}\rangle; P_1||P_2||\cdots||P_n\rangle\ \{\pi(\psi)\}$ must hold too. In the case that ψ is symmetric wrt. π, we have $\pi(\psi) \equiv \psi$. Thus $\pi(\phi)$ is a *sound* interpolant for program point $\langle pc_{\pi(1)}, pc_{\pi(2)}, \cdots, pc_{\pi(n)}\rangle$ wrt. the same safety property ψ and the same concurrent system $P_1||P_2||\cdots||P_n$. As a result, the use of interpolant $\pi(\phi)$ at line 3 in our algorithm is *sound*. □

Definition 9 (Symmetry Preserving Precondition Computation). *Given a parametrically defined n-process system and a safety property ψ, the precondition computation* pre *used in our algorithm is said to be* symmetry preserving *if for all $\pi \in Sym\ \mathcal{I}$, for all transition t and all postcondition ϕ • $\pi(\mathit{pre}(t;\phi)) \equiv \mathit{pre}(\pi(t);\pi(\phi))$.*

This property means that our precondition computation is consistent wrt. to renaming operation. In other words, the implementation of pre is *independent* of the naming of variables containted in its inputs. A reasonable implementation of pre can easily ensure this.

Definition 10 (Monotonic Precondition Computation). *Given a parametrically defined n-process system and a safety property ψ, the precondition computation* pre *used in our algorithm is said to be* monotonic *if for all transition t and all postconditions ϕ_1, ϕ_2 • $\phi_1 \rightarrow \phi_2$ implies $\mathit{pre}(t;\phi_1) \rightarrow \mathit{pre}(t;\phi_2)$.*

We emphasize here that the weakest precondition computation [3] does possess the monotonicity property. As is well-known, computing the weakest precondition in all the cases is very expensive. However, in practice (and in particular in the experiments we have performed), we often observe this monotonicity property with the implementation of our precondition computation. Incidentally, some possible implementations for this operation are discussed in [11, 10, 9, 1].

Definition 11 (Completeness). *In proving a parametrically defined n-process system with a global state space State and a safety property ψ, an algorithm which traverses the reachability tree is said to be* complete *wrt. a symmetry relation $\mathcal{R}$ iff for all $s, s' \in State$, $s\ \mathcal{R}\ s'$ implies that the algorithm will avoid traversing either the subtree rooted at s or the subtree rooted at s'.*

We remark here that our definition of completeness does not concern with the power of an algorithm in giving the answer to a safety verification question. This definition of completeness, however, is about the power of an algorithm in exploiting symmetry for search space reduction.

Theorem 2 (Completeness). *Our symmetry reduction algorithm is complete wrt. the weak symmetry relation if our operation* pre *is both monotonic and symmetry preserving.*

*Proof (**Outline**).* Assume that $s, s' \in State$ and s is weakly π-similar to s'. W.l.o.g. assume we encounter s first. If the subtree rooted at s is pruned (due to subsumption), the theorem trivially holds. The theorem also trivially holds if s is *not* a safe root. Now we consider that the subtree rooted as s is proved to be safe and the returned interpolant is ϕ. We will prove by structural induction on this interpolated subtree that s' will indeed be pruned, i.e. $s' \models \pi(\phi)$.

For simplicity of the proof, we will prove for loop-free programs only. In other words, we ignore our loop handling mechanism (line 4-7,17-18). Note that our theorem still holds for the general case. However, to prove this, we will require another induction on our fix-point computation in line 18.

For the base case that ϕ is ψ (when there is no schedulable transition from s) due to the definition of weak symmetry relation, there is no schedulable transition from s' and $s' \models \pi(\psi)$. Therefore, traversing the subtree rooted at s' is avoided.

As the induction hypothesis, assume now that the theorem holds for all the descendants of state s. Let assume that $\phi \equiv \psi \wedge \phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_k \wedge \phi_{k+1} \wedge \cdots \wedge \phi_m$, where $\phi_1 \cdots \phi_k$ are the interpolants contributed by enabled transitions in s and $\phi_{k+1} \cdots \phi_m$ are the interpolants contributed by schedulable but disabled transitions in s (line 14 and 15). Now assume the contrary that $s' \not\models \pi(\phi)$. We will show that this would lead to a contradiction. Using the first condition of weak symmetry relation, obviously $s' \models \pi(\psi)$. As such, there must exist some $1 \leq j \leq m$ such that $s' \not\models \pi(\phi_j)$. There are two possible cases: (1) ϕ_j is contributed by an enabled transition; (2) ϕ_j contributed by a disabled, but can be scheduled, transition.

Let us consider case (1) first. Assume ϕ_j corresponds to transition $t \in Enabled(s)$ and $s \xrightarrow{t} d$. By definition we have $s' \xrightarrow{\pi(t)} d'$ and d is weakly π-similar to d'. Let ϕ_d be interpolant for the subtree rooted at d. By induction hypothesis, we have $d' \models \pi(\phi_d)$. Obviously, we have $s' \models \mathsf{pre}(\pi(t); d')$, by monotonicity of pre, we deduce $s' \models \mathsf{pre}(\pi(t); \pi(\phi_d))$. As pre is symmetry preserving, $s' \models \mathsf{pre}(\pi(t); \pi(\phi_d)) \equiv \pi(\mathsf{pre}(t; \phi_d)) \equiv \pi(\phi_j)$. Consequently we arrive at the fact that $s' \models \pi(\phi_j)$ which is a contradiction.

For case (2), by using the symmetry preserving property of pre and the fact that $\pi(false) \equiv false$, we also derive a contradiction. □

5 Experimental Evaluation

We used a 3.2 GHz Intel processor and 2GB memory running Linux. Unless otherwise mentioned, timeout is set at 5 minutes, and '-' indicates timeout. In this section, we benchmark our proposed approach, namely Complete Symmetry Reduction (CSR), against current state-of-the-arts.

Our first example is the classic *dining philosophers* problem. As commonly known, it exhibits *rotational* symmetry. However, and more importantly, we ex-

Table 1. Experiments on Dining Philosophers

	CSR			RSR			NSR		
# Phil	Visited	Subsumed	T(s)	Visited	Subsumed	T(s)	Visited	Subsumed	T(s)
3	68	29	0.02	67	27	0.02	191	79	0.06
4	230	134	0.09	328	184	0.13	1246	702	0.81
5	662	446	0.28	1509	981	0.71	7517	4893	4.93
6	1778	1304	0.85	7356	5216	4.18	43580	30908	34.53
7	4584	3552	2.55	35079	26335	28.83	–	–	–
8	11526	9281	7.54	–	–	–	–	–	–
9	28287	23432	22.6	–	–	–	–	–	–
10	67920	57504	58.07	–	–	–	–	–	–
11	159738	137609	226.86	–	–	–	–	–	–

ploit far more symmetry than that. More specifically, at *any* program point, rotational symmetry is applicable. Nevertheless, for certain program points, when some transitions have been taken, the system exhibits more symmetry than just rotational symmetry. With this benchmark, we demonstrate the power of our complete symmetry reduction (CSR) algorithm. Here, we verify a *tight* safety property that 'no more than *half* the philosophers can eat simultaneously'.

Table 1 presents three variants: Complete Symmetry Reduction (CSR), Rotational Symmetry Reduction (RSR), and No Symmetry Reduction (NSR). The number of *stored states* is the difference between the number of visited states (Visisted column) and subsumed states (Subsumed column). Note that although RSR achieves linear reduction compared to NSR, it does not scale well. CSR significantly outperforms RSR and NSR in all the instances.

Table 2. Experiments on Reader-Writer Protocol

		Complete Symmetry Reduction			Lazy Symmetry Reduction	
# Readers	# Writers	Visisted	Subsumed	T(s)	Abstract States	T(s)
2	1	35	20	0.01	9	0.01
4	2	226	175	0.19	41	0.10
6	3	779	658	0.93	79	67.80
8	4	1987	1750	3.23	165	81969.00
10	5	4231	3820	9.21	–	–

Next consider the *Reader-Writer Protocol* from [14, 15]. Here we highlight the aspect of *search space size* as compared to top-down techniques, of which the most recent implementation of Lazy Symmetry Reduction [15] is chosen as a representative [2]. Table 2 shows that although lazy symmetry reduction has aggressively compressed the state space (which now grows roughly in linear complexity), the running time is still *exponential*. In other words, the number of abstract states is not representative of the search space. In contrast, the running time of our method is significantly better. In the instance of 8 readers and 4 writers, we extended the timeout for [15] to finish; and it takes almost 1 day.

[2] We receive this implementation from the authors of [15].

Table 3. Experiments on sum-of-ids Example

	Complete Symmetry Reduction			SPIN-NSR		
# Processes	Visited	Subsumed	T(s)	Visited	Subsumed	T(s)
10	57	45	0.02	6146	4097	0.03
20	212	190	0.04	11534338	9437185	69.70
40	822	780	0.37	–	–	–
60	1832	1770	1.91	–	–	–
80	3242	3160	7.62	–	–	–
100	5052	4950	22.09	–	–	–

Next we experiment with the 'sum-of-ids' example mentioned earlier. To the best of our knowledge, there is no symmetry reduction algorithm which can detect and exploit symmetry here. Table 3 shows we have significant symmetry reduction. In term of memory (stored states), we enjoy linear complexity. For reference, we also report the running time of this example, without symmetry reduction, using SPIN 5.1.4 [13].

Table 4. Experiments on Bakery Algorithm

	Complete Symmetry Reduction			SI		
# Processes	Visited	Subsumed	T(s)	Visited	Subsumed	T(s)
3	65	31	0.10	265	125	0.43
4	182	105	0.46	1925	1089	5.89
5	505	325	2.26	14236	9067	74.92
6	1423	983	11.10	–	–	–

In the fourth and last example, we depart slightly from our finite domain to allow infinite domain variables and loops. We choose the well-known Bakery Algorithm to perform the experiments, and we use the well-known abstraction of using an inequality to describe each pair of counters to close the loops. Again, as far as we are aware of, there has been no symmetry reduction algorithm which can detect and exploit symmetry for this example. Table 4 shows the significant improvements due to our symmetry reduction, compared to just symbolic execution with interpolation, denoted as SI.

6 Conclusion

We presented a method of symmetry reduction for searching the interleaving space of a concurrent system of transitions in pursuit of a safety property. The class of systems considered, by virtue of being defined parametrically, is completely general; the individual processes may be at any level of similarity to each other. We then enhanced a general method of symbolic execution with interpolation for traditional safety verification of transition systems, in order to deal with symmetric states. We then defined a notion of weak symmetry, one that allows for more symmetry than the stronger notion that is used in the literature. Finally, we showed that our method, when employed with an interpolation algorithm which is monotonic, can exploit weak symmetry completely.

References

1. D. H. Chu and J. Jaffar. Symbolic simulation on complicated loops for WCET path analysis. In *EMSOFT*, pages 319–328, 2011.
2. E. M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In *CAV*, pages 450–462, 1993.
3. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18:453–457, August 1975.
4. E. A. Emerson, J. W. Havlicek, and R. J. Trefler. Virtual symmetry reduction. In *Logic in Computer Science*, pages 121–131, 2000.
5. E. A. Emerson and A. P. Sistla. Model checking and symmetry. In *CAV*, pages 463–478, 1993.
6. E. A. Emerson and A. P. Sistla. Utilizing symmetry when model-checking under fairness assumptions. *ACM Trans. Program. Lang. Syst.*, 19(4):617–638, July 1997.
7. E. A. Emerson and R. J. Trefler. From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. In *Conference on Correct Hardware Design and Verification Methods*, pages 142–156, 1999.
8. C. N. Ip and D. L. Dill. Better verification through symmetry. *Form. Methods Syst. Des.* 9(1/2):41–75, 1996.
9. J. Jaffar, J. A. Navas, and A. E. Santosa. Unbounded symbolic execution for program verification. In *RV*, 2011.
10. J. Jaffar, A. E. Santosa, and R. Voicu. An interpolation method for CLP traversal. In *CP*, pages 454–469, 2009.
11. A. Rybalchenko and V. Sofronie-Stokkermans. Constraint solving for interpolation. In *VMCAI*, pages 346–362, 2007.
12. A. P. Sistla and P. Godefroid. Symmetry and reduced symmetry in model checking. *ACM Trans. Program. Lang. Syst.*, 26(4):702–734, July 2004.
13. SPIN model checker, `http://spinroot.com`
14. T. Wahl. Adaptive symmetry reduction. In *CAV*, pages 393–405, 2007.
15. T. Wahl and V. D'Silva. A lazy approach to symmetry reduction. *Form. Asp. Comput.*, 22:713–733, November 2010.