Dynamic Symmetry Reduction*

E. Allen Emerson and Thomas Wahl

Department of Computer Sciences and Computer Engineering Research Center,
The University of Texas, Austin/TX 78712, USA
{emerson, wahl}@cs.utexas.edu

Abstract. *Symmetry reduction* is a technique to combat the state explosion problem in temporal logic model checking. Its use with symbolic representation has suffered from the prohibitively large BDD for the *orbit relation*. One suggested solution is to pre-compute a mapping from states to possibly *multiple representatives* of symmetry equivalence classes. In this paper, we propose a more efficient method that determines representatives dynamically during fixpoint iterations. Our scheme preserves the uniqueness of representatives. Another alternative to using the orbit relation is *counter abstraction*. It proved efficient for the special case of full symmetry, provided a conducive program structure. In contrast, our solution applies also to systems with less than full symmetry, and to systems where a translation into counters is not feasible. We support these claims with experimental results.

1 Introduction

Model checking [CE81, QS82] is a successful approach to formal verification of finite-state concurrent systems. Numerous attempts have been made to combat its main obstacle, the state space explosion problem. *Symmetry reduction* is a technique that exploits replication. The state space is reduced by considering global states equivalent that are identical up to applications of structure automorphisms, for example permutations that interchange the identities of participating components. This equivalence, the *orbit relation*, gives rise to a bisimilar quotient structure over the equivalence classes (orbits) [ES96, CEFJ96].

Symmetry reduction was first successfully incorporated into explicit-state verifiers, such as Murφ [ID96]. Disappointingly, it was discovered that *symbolic representation* using BDDs, by then becoming a standard in large-scale system verification, seemed not to combine favorably with symmetry reduction [CEFJ96]. The reason is that the BDD for the orbit relation is provably intractably large.

In this paper we present a strategy of bypassing the orbit relation. To perform symmetry reduction, it is not necessary to build a representation of the quotient structure. Instead, the reduction can be achieved by computing transition images

* This work was supported in part by NSF grants CCR-009-8141 and CCR-020-5483, and SRC contract 2002-TJ-1026.

N. Halbwachs and L. Zuck (Eds.): TACAS 2005, LNCS 3440, pp. 382–396, 2005.
© Springer-Verlag Berlin Heidelberg 2005

with respect to the unreduced structure, and immediately afterwards mapping the new states to their respective representatives. The main contribution of this paper is to provide an efficient *symbolic* algorithm for a function that takes a set of states and computes their representatives, given the underlying symmetry group and the permutation action. We first concentrate on full component symmetry, the most frequent and beneficial case in practice. We go on to show how to extend our algorithm to other symmetry groups and to *data symmetry* (see section 2).

Our solution compares with previous approaches as follows. Clarke et. al. [CEFJ96] proposed the admission of *multiple* orbit representatives. This affords the possibility to map a state to that representative of its orbit for which this mapping is most efficient. The relation ξ associating states with their representatives is pre-computed in a BDD. This method, albeit an improvement, was not effective enough for systems of interesting size. This is in part because the BDD for ξ is generally still huge, and in part due to the multiplicity of the representatives, such that symmetry is not exploited to the fullest extent. In comparison, our method *dynamically* computes representatives of states, i.e. embedded in the model checking process. In addition to preserving the uniqueness of orbit representatives, this has the important advantage that there is no need to compute, let alone store for the lifetime of the program, the representative mapping ξ. Further, for reachability analysis we only maintain representatives actually encountered during the computation, which might be few. In contrast, pre-computing representatives (irrespective of reachability) may consume a lot of resources, only to find afterwards that a state close to an initial state already has a bug.

Another technique, *generic representatives* [ET99], applies to the special case of fully symmetric systems. The idea is that two global states are symmetry-equivalent exactly if for every *local* state L, the number of components residing in L is the same in both global states. This approach requires a translation of the original program text into one that represents global states as vectors of counters, one counter for each local state. The Kripke structure derived from the new program can then be model-checked without further symmetry considerations. This method, more generally known as *counter abstraction* [PXZ02], proved to be very efficient—if applicable: it is limited to full component symmetry, its performance degrades if there are too many local states, and the translation to counters can be non-trivial [EW03]. Our new reduction algorithm is not based on counting processes in local states and thus does not suffer from any of these problems.

In summary, this paper presents an exact, yet flexible and efficient response to the orbit relation dilemma of symbolic symmetry reduction. It works with many common symmetry groups and even applies to data symmetry. It requires no expensive up-front computation of a representative mapping and no translation of the input program, nor does it depend unreasonably on the number of local states. Experimental results document the superiority of our approach to existing ones, often by orders of magnitude.

2 Background: Symmetry Reduction

Intuitively, the Kripke model $M = (S, R)$ of a system is symmetric if it is invariant under certain transformations of its state space S. In general, such a transformation is a bijection $\pi\colon S \to S$. The precise definition of π depends on the type of symmetry; common ones are discussed in the next paragraph. Given π, we derive a mapping at the transition relation level by defining $\pi(R) = \{(\pi(s), \pi(t)) : (s,t) \in R\}$. Structure M is said to be *symmetric* with respect to a set G of bijections if $\pi(R) = R$ for all $\pi \in G$. The bijections with this property are called *automorphisms* of M and form a group under function composition, M's *symmetry group*.

The most common type of symmetry is known as *component symmetry*. In this case, the automorphism π takes over the task of permuting the components. For example, if l_i denotes the local state of component $i \in [1..n]$, π is derived from a permutation on $[1..n]$ and acts on a state s as $\pi(s) = \pi(l_1, \ldots, l_n) = (l_{\pi(1)}, \ldots, l_{\pi(n)})$. Under *data symmetry* [ID96], an automorphism acts directly on the variable values, in the form $\pi(l_1, \ldots, l_n) = (\pi(l_1), \ldots, \pi(l_n))$. For example, the permutation π on $\{a, b\}$ that flips a and b acts on state (a, a) under component symmetry by exchanging *positions* 1 and 2 in the pair to yield the same state (a, a). Under data symmetry, π exchanges the *values* a and b to yield (b, b).

2.1 Exploiting Symmetry

Given a group G of automorphisms $\pi\colon S \to S$, the relation $\theta := \{(s,t) : \exists \pi \in G : \pi(s) = t\}$ on S defines an equivalence between states, known as *orbit relation*; the equivalence classes it entails are called *orbits* [CEFJ96]. Relation θ induces a quotient structure $\bar{M} = (\bar{S}, \bar{R})$, where $\bar{S}$ is a chosen set of representatives of the orbits, and $\bar{R}$ is defined as

$$\bar{R} = \{(\bar{s}, \bar{t}) \in \bar{S} \times \bar{S} :\ \exists s, t \in S : (s, \bar{s}) \in \theta\ \wedge\ (t, \bar{t}) \in \theta\ \wedge\ (s,t) \in R\}. \tag{1}$$

Depending on the size of G, $\bar{M}$ can be up to exponentially smaller than M. In case of symmetry, i.e. given $\pi(R) = R$ for all $\pi \in G$, $\bar{M}$ is bisimulation equivalent to M; the bisimulation relation is $\xi = (S \times \bar{S}) \cap \theta$. Relation ξ is actually a function and maps a state s to the unique representative $\bar{s}$ of its equivalence class under θ. Summarizing, for two states s, $\bar{s}$ with $(s, \bar{s}) \in \xi$ and any *symmetric* formula f, i.e. such that $p \Leftrightarrow \pi(p)$ is a tautology for every propositional subformula p of f and every $\pi \in G$,

$$M, s \models f \quad \text{iff} \quad \bar{M}, \bar{s} \models f. \tag{2}$$

2.2 Unique Versus Multiple Representatives

Equation 1 defining the quotient transition relation makes use of the orbit relation θ. Clarke et. al. [CEFJ96] show that computing this relation can be expensive in both time and space, especially in a symbolic context. There is currently no polynomial-time algorithm for deciding, for an arbitrary symmetry group G,

whether there is a permutation mapping a given state to another. In addition, a symbolic representation of the orbit relation using BDDs can be shown to require space exponential in the smaller of the number of components and the number of local states per component. This is even true for special symmetries, such as the important full symmetry group.

An alternative that avoids the orbit relation is provided by the same authors [CEFJ96]. In their approach, the quotient structure is allowed to have more than one representative per orbit. The programmer supplies a choice of representative states *Rep*. In [CEFJ96], precise conditions are given for the existence of a set $C \subset G$ of permutations such that

$$\xi := \{(s, r) : \ r \in Rep \wedge \exists \pi \in C : \pi(s) = r\}$$

is a suitable representative relation from which a bisimulation equivalent quotient structure can be derived. This quotient is the structure $\bar{M} = (Rep, \bar{R})$, where $\bar{R}$ is defined as before in equation 1, except that $\bar{S}$ is replaced by *Rep*, and θ by ξ. The intuition behind this is that for any state s, in order to find a representative r for it (i.e. $(s, r) \in \xi$), it suffices to try applying permutations from C to s, instead of all permutations from G. If C is exponentially smaller than G, this is a big win in the search for a representative of s. Indeed, as experiments have shown, avoiding the orbit relation this way certainly outweighs the cost of extra representative states.

2.3 Counter Abstraction of Fully Symmetric Systems

For the case of fully symmetric systems of concurrently executing components, one can make use of the following observation in order to represent orbits. Two global states, viewed as vectors of local state identifiers, are identical up to permutation exactly if for every local state L, the frequency of occurrence of L is the same in the two states—permutations only change the order of elements, not the elements themselves. An orbit can therefore be represented as a vector of counters, one for each local state, that records how many of the components are in the corresponding local state. For example, in a system with local states N, T and C, the states (N, N, T, C), (N, C, T, N), and (T, N, N, C) are all symmetry-equivalent; their orbit (which contains other states as well) can be represented compactly as $(2N, 1T, 1C)$ or just $(2, 1, 1)$.

In practice, it may be possible to rewrite the program describing a fully component-symmetric system such that its variables are local state counters in the first place (before building a Kripke structure). This procedure is known as counter abstraction [PXZ02]. The advantage of the counter notation is that the symmetry is implicit in the representation; the very act of rewriting the program from the *specific* notation of local state variables into the *generic* [ET99] notation of local state counters implements symmetry reduction. Subsequently, model checking can be applied to the structure derived from the counter-based program without further considerations of symmetry.

In addition to being applicable only to fully symmetric systems, counter abstraction requires that automorphisms act on states only by changing the order

of state components, not their values (as they do under data symmetry), since only then counters are insensitive to automorphism application. Further, since rewriting the program in terms of counters in fact anonymizes the components, variables containing component identifiers complicate matters, for example turn variables and pointers to other components [EW03].

3 Dynamically Computing Representatives

Symmetry reduction and model checking can be combined in two principally different ways. The straightforward method seems to be to build a representation of the quotient structure $\bar{M}$ and then model check it. Fig. 1 (a) shows the standard fixpoint routine to compute the representative states satisfying $\mathsf{EF}\,\mathit{bad}$, assuming we have a BDD representation of the quotient transition relation $\bar{R}$. We use $\overline{\mathit{bad}}$ to denote the representatives of *bad* states of M.

In practice, this algorithm is usually not the method of choice for symbolic model checking. The reason is that direct computation of the BDD for the quotient transition relation $\bar{R}$ is very expensive. Equation 1 involves the orbit relation directly and is thus intractable as an algorithm. In our experiments, we were only able to compute this BDD in a reasonable amount of time for very simple examples. An intuitive argument for the complexity is that the orbit relation, even if not used during the computation of $\bar{R}$, is essentially embedded in the BDD for $\bar{R}$.

An alternative is to modify the model checking algorithm. Consider the version in fig. 1 (b). It is identical to (a), except that it uses the operation $\alpha(\mathsf{EX}_R\, Z)$ in the computation of the next iterate: It first applies to Z the backward image operator with respect to R, rather than with respect to $\bar{R}$. It then employs some mechanism α that maps the results to representatives, formally defined as

$$\alpha(T) = \{\bar{t} \in \bar{S} : \exists t \in T : (t, \bar{t}) \in \theta\}. \tag{3}$$

Viewing the quotient $\bar{M}$ as an abstraction of the concrete system M, α precisely denotes the abstraction function, mapping concrete to abstract states. The algorithm in fig. 1 (b) is an instance of the *abstract interpretation* framework [CC77]. Generally, abstract images can be computed by mapping the given

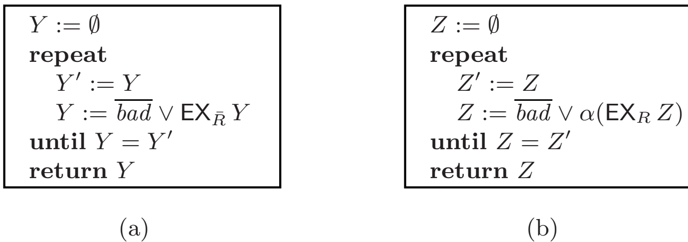

Fig. 1. Two ways to compute the representative states satisfying $\mathsf{EF}\,\mathit{bad}$

abstract state to the concrete domain using the concretization function γ, then applying the concrete image, and finally mapping the result back to the abstract domain using α. Symmetry affords the simplification that γ can be chosen to be the identity function, since abstract states (i.e. representatives) are embedded in the concrete state space. We can thus apply EX_R (the concrete backward image operator) directly to a set of abstract states (Z, in fig. 1 (b)), obtaining the set of concrete successor states. Applying α produces the final abstract backward image result.

Given different realizations of α, fig. 1 (b) actually represents a family of symmetry reduction algorithms. The definition of α (3) is based on the orbit relation and is therefore inappropriate as a recipe for an algorithm. Another way to compute α is as forward image under a precomputed representative relation $\xi \subset S \times \bar{S}$. This technique was used by Clarke et. al. [CEFJ96] in connection with multiple representatives; the authors describe ways to obtain such a relation without explicitly using the orbit relation θ. In contrast, we propose in this paper to compute the set of representatives of a set of states *dynamically* during the execution of symbolic fixpoint algorithms, instead of a priori *statically*. This has two advantages:

1. We avoid computing, and storing at all times, the table ξ associating states with representatives, which is expensive.
2. For reachability analysis, we do not need the complete set of representatives $\bar{S}$, which is required for the computation of ξ. Rather we only maintain representatives encountered during the computation.

The algorithm to compute α depends on the type and underlying group of symmetry. In the following section, we first describe in detail the algorithm for the most common and important case of full component symmetry. Later, in section 6, we present extensions to other symmetries and also generalize our algorithm to full CTL model checking.

4 Computation of α Under Full Component Symmetry

A scheme for defining representatives frequently used in the case of full component symmetry is the following. Recall that an orbit consists of all states that are identical up to permutations of components, which amounts to permutations of the local states of the components. Given some total ordering among the local states, there is a unique state in each orbit where the local states appear in ascending order. Thus, the unique representative of a state can be chosen to be the lexicographically least element of the state's orbit. This element can be computed by *sorting* the local state vector representing the given state.[1]

How can this be accomplished symbolically? Not every sorting algorithm lends itself to symbolic implementation. Compared to an explicit-state algorithm,

[1] We assume for now that there are no symmetry-relevant global variables; section 6 below generalizes.

instead of sorting one vector of local states, we want to sort an entire set of local state vectors in one fell swoop. One algorithm that allows this efficiently is *bubble sort*. It is a comparison-based sorting procedure that rearranges the input vector in-place by swapping out-of-order elements. To symbolically bubble-sort a set of vectors simultaneously, we remark: Instead of comparing two elements of *the* input vector, the algorithm forms a sub*set* of vectors for which the two elements in question are out of order. Instead of swapping one pair of out-of-order elements, we apply the swap operation to all vectors in the subset, in one step.

The operation of swapping two items turns out to be the factor dominating efficiency. Its complexity depends heavily on the distance, in the BDD variable ordering, of the bits involved in the swap. In order to keep this distance small, we exploit one key feature of bubble sort: it is optimal in the *locality* of swap operations—it swaps only adjacent elements.

The lexicographical order of global states is based on a total order $\leq$ on the local states of the components. For a fixed global state z, this order $\leq$ induces a total order $\leq_z$ on the components via

$$p \leq_z q \quad \text{iff} \quad l_p(z) \leq l_q(z),$$

where $l_i(z)$ is the local state of component i in global state z. Given $\leq_z$, and denoting by n the total number of components, the set of representative states (the lexicographically least members of some orbit) is defined as

$$\bar{S} = \{z \in S : \forall p < n : p \leq_z p+1\} = \bigcap_{p<n} \{z \in S : p \leq_z p+1\}. \tag{4}$$

For our algorithm, the exact definition of $\leq_z$ is irrelevant; we only need it to be a total order on the components. This flexibility turns out to be useful in situations where considering just the local states of components is not enough to characterize representative states; these situations are discussed in section 6. Our sorting algorithm looks for states z with components that are not in correct order with respect to $\leq_z$, and swaps them. This is repeated until a fixpoint is reached, cf. fig. 2.

For p ranging from 1 to $n-1$, the predicate transformer τ in (b) computes Z_{bad}, the set of states in Z in which components p and $p+1$ are not in the correct order (line 2). If Z_{bad} is non-empty, the algorithm first saves the set of states in Z in which p and $p+1$ are in correct order (line 4) and then swaps components p and $p+1$ in all states in Z_{bad} (line 5). The simultaneous swapping can be achieved by swapping the bits that store components p and $p+1$ in the BDD for Z_{bad}, which effects all states in Z_{bad}. This is the expensive step of the algorithm; it profits from the fact that these bits are nearby (see section 5). Finally, the untouched and the swapped states in Z are combined to give the new value for Z (line 6).

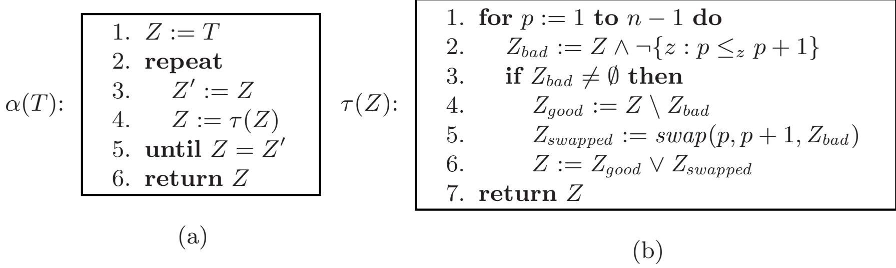

Fig. 2. Computing the representative mapping α using subroutine τ

5 Correctness and Efficiency of the Algorithm

Our algorithm is an instance of the template shown in fig. 1 (b). We first show more generally that the template computes the same result as the algorithm (a) in the same figure. We only assume that α maps the states of its argument set to representatives.

Lemma 1. Let α satisfy

$$\alpha(T) = \{\bar{t} \in \bar{S} : \exists t \in T : (t, \bar{t}) \in \theta\}. \tag{5}$$

Then, for an arbitrary set $P \subset \bar{S}$ of representatives, $\mathsf{EX}_{\bar{R}}\, P = \alpha(\mathsf{EX}_R\, P)$.

Proof: In the following, we also use the notation $\alpha(t)$ for the unique representative of a single state t, i.e. the unique element of $\alpha(\{t\})$.

$$\begin{array}{ll}
 & \bar{s} \in \alpha(\mathsf{EX}_R\, P) \\
\Leftrightarrow & \langle \text{def. of backward image and function application} \rangle \\
 & \exists s, \bar{t} : \bar{s} = \alpha(s) \wedge (s, \bar{t}) \in R \wedge \bar{t} \in P \\
\Leftrightarrow & \langle \text{“}\Rightarrow\text{”}: t := \bar{t} \text{ and note } \bar{t} \in P \subset \bar{S}, \text{ so } \bar{t} = \alpha(\bar{t}) = \alpha(t) \rangle \\
 & \langle \text{“}\Leftarrow\text{”}: s := \pi(s') \text{ for } \pi : \pi(t) = \bar{t}. \text{ Then } \alpha(s') = \alpha(s),\ \pi(s', t) = (s, \bar{t}) \in R \rangle \\
 & \exists s', t, \bar{t} : \bar{s} = \alpha(s') \wedge \bar{t} = \alpha(t) \wedge (s', t) \in R \wedge \bar{t} \in P \\
\Leftrightarrow & \langle \text{def. of } \bar{R} \rangle \\
 & \exists \bar{t} : (\bar{s}, \bar{t}) \in \bar{R} \wedge \bar{t} \in P \\
\Leftrightarrow & \langle \text{def. of backward image} \rangle \\
 & \bar{s} \in \mathsf{EX}_{\bar{R}}\, P.
\end{array}$$

□

Corollary 2. The two algorithms in fig. 1 return the same set (and they do so with the same number of iterations of the **repeat** loop).

Proof: Let Y_i and Z_i denote the ith iterates of the two algorithms. Then for all i, $Y_i \subset \bar{S}$, $Z_i \subset \bar{S}$ (by the definitions of $\overline{bad}$, backward image in $\bar{R}$ and α). Thus, utilizing lemma 1, for all i, $Y_i = Z_i$, from which the two claims follow. □

Lemma 3. The algorithm in fig. 2 computes α satisfying equation 5.

Proof: We will show termination and partial correctness.

Termination: The argument is essentially the same as for standard bubble sort. Every call to $swap(p, p+1, Z_{bad})$ brings the local state of at least one of the components p and $p+1$ closer to its correct position. Hence, after about n^2 swaps, there is no pair $(p, p+1)$ left with $\neg(p \leq_z p+1)$. Thus, Z_{bad} as computed in line 2 (fig. 2 (b)) is empty in every iteration of the **for** loop, Z remains unchanged, and the condition $Z = Z'$ in line 5 (a) is true.

Partial correctness: We use two observations.

(I) When the algorithm terminates, we know that for all values of p, Z_{bad} as computed in line 2 (b) is empty. Hence, $Z \subset \bigcap_{p<n}\{z : p \leq_z p+1\} = \bar{S}$ (equation 4), so $Z = \alpha(T) \subset \bar{S}$.

(II) Predicate transformer τ manipulates the set Z by applying transpositions (*swap*) to states in Z. Hence, at the end T and $\alpha(T)$ contain the same states up to permutations.[2]

These observations allow us to prove $\alpha(T) = \{\bar{t} \in \bar{S} : \exists t \in T : (t, \bar{t}) \in \theta\}$ as two inclusions:

$\subset$: Consider $\bar{t} \in \alpha(T)$. From (I) we know $\bar{t} \in \bar{S}$. From (II) we conclude that there exists t in T with $(t, \bar{t}) \in \theta$.

$\supset$: Consider $\bar{t} \in \bar{S}$, $t \in T$ such that $(t, \bar{t}) \in \theta$. From (II) we conclude that there exists π such that $\pi(t) \in \alpha(T)$. From (I) we conclude $\pi(t) \in \bar{S}$. Since there is exactly one representative of t in $\bar{S}$, we derive $\pi(t) = \bar{t}$, so $\bar{t} \in \alpha(T)$. $\square$

Corollary 4. The algorithm in fig. 1 (b), using the computation of α in fig. 2, correctly implements backward reachability analysis on the quotient structure.

Efficiency Considerations

The set $\{z : p \leq_z p+1\}$, which is by definition $\{z : l_p(z) \leq l_{p+1}(z)\}$, needs to be calculated only once for each p. The condition that the local state of component p is at most that of component $p+1$ can be expressed symbolically with a BDD of size $\mathcal{O}(l^2)$, for the number l of possible local states.

As indicated earlier, the *swap* operation in line 5, fig. 2 (b), is the bottleneck of the algorithm. In BDD terms, it corresponds to pairwise swapping of all bits that represent the two items to be swapped. The complexity of swapping two bits in all elements of a set T, i.e. computing

$$\{(\ldots x_j \ldots x_i \ldots) : (\ldots x_i \ldots x_j \ldots) \in T\},$$

depends exponentially on the distance d of x_i and x_j in the BDD variable ordering. To illustrate this claim, we observe that in the BDD for T, every subtree rooted at a node labeled x_i contains at most 2^d nodes labeled x_j. Each such node labeled x_j has an immediate subtree that corresponds to one of the cases

[2] There is, however, in general no single π such that $\alpha(T) = \pi(T)$, i.e. α by itself is not just a permutation.

affected by the swap, namely $(x_i, x_j) = (0, 1)$ and $(x_i, x_j) = (1, 0)$. These 2^d subtrees must be moved.

BDD variable orderings usually have the property that it is possible to index the components as $1, \ldots, n$ such that the distance between corresponding bits of components p and q is proportional to $|p - q|$. Consider, for example, the following frequently used orderings:

concatenated:	$b_{11} \ldots b_{1\log l}$	$b_{21} \ldots b_{2\log l}$	$\ldots$	$\ldots$ b_{n1}	$\ldots b_{n\log l}$
interlaced:	$b_{11} \ldots b_{n1}$	$b_{12} \ldots b_{n2}$	$\ldots$	$\ldots$ $b_{1\log l}$	$\ldots b_{n\log l}$

where b_{ij} denotes the jth bit of component i. For the concatenated ordering, the distance between the jth bit of component p and the jth bit of component q is $\log l \cdot |p - q|$; for the interlaced ordering, it is $|p - q|$.

Bubble sort, among the numerous sorting procedures, enjoys the unique feature of swapping only adjacent components. The distance $|p - q|$ is hence 1, for every swap operation, thus minimizing the complexity of swapping. This proves bubble sort optimal for our purpose of symbolic sorting.

6 Generalizations

6.1 Other Types of Symmetry

The idea of sorting to obtain unique orbit elements only works for the case of full component symmetry. Without proof, we give here the idea of how to compute α for other, less lucrative, but still somewhat common types of symmetry.

Consider first the case of component symmetries. Permutations act on states in the form $\pi(l_1, \ldots, l_n) = (l_{\pi(1)}, \ldots, l_{\pi(n)})$. Our solution for full symmetry generalizes as follows. Call a symmetry group G of permutations on $[1..n]$ *nice* if there exists a "small" subset F of G with the following property: *A state z is lexicographically least in its orbit exactly if there is no $\pi \in F$ with $\pi(z) <_{lex} z$.* Many common symmetry groups are nice. For full symmetry, F can be chosen as the set of $n - 1$ transpositions $(i \;\; i+1)$ $(1 \leq i < n)$. Set F also happens to be a generating set for the full symmetry group. If the group G itself is small, $F := G$ is a viable choice. This is, for example, the case for the n rotations generated by the left shift cycle $(1 \; 2 \; \ldots \; n)$. Note that in this case the generating set $\{(1 \; 2 \; \ldots \; n)\}$ is not a valid choice for F: The vector $z := (BCA)$ is not lexicographically least, yet applying the generating permutation does not make z smaller (applying it twice does).

Given a nice group G, consider the algorithm for α as before in fig. 2 (a), but with subroutine τ as shown in fig. 3. Again, Z_{bad} in line 2 selects the states z in Z that are not lexicographically least. By the niceness of G, this means that for some $\pi \in F$, $\pi(z) <_{\text{lex}} z$. Line 5 applies π element-wise to Z_{bad}. This algorithm terminates, since $<_{\text{lex}}$ is a well-order on the set of local state vectors. Hence, eventually there will be no π such that for some z, $\pi(z) <_{\text{lex}} z$. Partial correctness follows from an argument similar to that in lemma 3.

If G is nice, we expect to have a small set F of permutations that can be traversed in line 1. The direct application of π in line 5 may be expensive.

```
        1. for π in F do
        2.     Z_bad := Z ∧ {z : π(z) <_lex z}
        3.     if Z_bad ≠ ∅ then
τ(Z):   4.         Z_good := Z \ Z_bad
        5.         Z_swapped := {π(z) : z ∈ Z_bad}
        6.         Z := Z_good ∨ Z_swapped
        7. return Z
```

Fig. 3. Subroutine τ for "nice" symmetry groups

However, π can be expressed as a product of at most $1/2n(n-1)$ transpositions of *adjacent* elements. As argued in section 5, transpositions of neighbors are the least expensive permutations, as for implementation using BDDs. The important point is that the algorithm for τ in fig. 3 resembles bubble sort, in that it is in-place, and it only swaps neighboring processes, if the π's in F are rewritten as products of transpositions.

For data symmetry (see section 2), the idea of lexicographically least orbit elements no longer applies. A set of unique representatives can be defined as $\{(l_1, \ldots, l_n) : \forall i : l_i \leq i\}$. To compute the mapping α, the algorithm in fig. 2 can still be used, with only slight modifications. Set Z_{bad} (line 2) contains the states from Z that satisfy $l_p > p$. Line 5 swaps the values p and l_p in all states in Z_{bad}. Since l_p may vary from state to state in Z_{bad} (even for fixed p), a loop over the possible values of l_p is required.

6.2 Process Id Variables

Often, systems have *id-sensitive* global variables containing component ids, such as the identity of a process holding a token or a reference to a process having an exclusive copy of some cache data. In this case, the condition $\forall p : p < n : l_p(z) \leq l_{p+1}(z)$ is not enough to guarantee that z is a unique representative state. Consider, for instance, the two states $(A, A, B, 1)$ and $(A, A, B, 2)$ of a three-process system with one id-sensitive global variable (listed last). Since components 1 and 2 are both in local state A in both states, the permutation that flips 1 and 2 proves the states equivalent[3]. The local states appear in ascending order: AAB. Yet, the states differ, compromising uniqueness. The solution is to define the unique representative as the orbit element with ascending local states where the id-sensitive variables have minimal values (1, in the example above). In this case, $p \leq_z p+1$ means for state z and the local states of p and $p+1$ that either $l_p(z) < l_{p+1}(z)$, **or** $l_p(z) = l_{p+1}(z)$ and none of the id-sensitive variables has value $p+1$. This condition is violated for $z := (A, A, B, 2)$ and $p := 1$. Thus, the permutation $(1\ 2)$ will be applied to z, whereupon it turns into $(A, A, B, 1)$.

This solution can be extended to the more challenging case of id-sensitive *local* variables, the general treatment of which is beyond the scope of this paper.

[3] This permutation acts on (permutes) the id-sensitive variable; see [EW03] for details.

6.3 Full CTL Model Checking

Section 4 can be summarized as having presented an efficient algorithm for the computation of $\mathsf{EX}_{\bar{R}}\, Z$, used in backward reachability analysis. This algorithm generalizes to all CTL formulas as follows. Existential modalities (EG, EF, EU) have a fixpoint characterization based on existential backward images. For example, $\mathsf{EG}\, f$ can be calculated as the greatest fixpoint of the predicate transformer $\lambda(Z) = f \wedge \mathsf{EX}\, Z$. For the quotient structure, an algorithm similar to that in fig. 1 (b) can be used.

The universal backward image $\mathsf{AX}_{\bar{R}}\, Z$ cannot be replaced by an analogous construct involving α. Suppose we wish to compute the representative states satisfying $\mathsf{AG}\, \mathit{good}$ on the quotient structure. An algorithm similar to that in fig. 1 (a) exists, which computes the greatest fixpoint of $\lambda(Z) = \overline{\mathit{good}} \wedge \mathsf{AX}_{\bar{R}}\, Z$. However, in general $\alpha(\mathsf{AX}_{R}\, Z) \subsetneq \mathsf{AX}_{\bar{R}}\, Z$. The underlying problem is that the abstraction function α distributes over set union, but not intersection:

$$\begin{array}{ll} \alpha(P \cup Q) = \alpha(P) \cup \alpha(Q), & \text{but} \\ \alpha(P \cap Q) \subsetneq \alpha(P) \cap \alpha(Q) & \text{(in general).} \end{array}$$

The solution is to reduce universal to existential modalities. Care must be taken in that negation over the quotient structure is with respect to $\bar{S}$, the set of representatives. Thus, in a context where states are encoded as elements of S, we have to compute $\{\bar{s} \in \bar{S} : \bar{s} \notin Z\}$ as $\bar{S} \wedge \neg Z$. We obtain

$$\mathsf{AX}_{\bar{R}}\, Z = \bar{S} \wedge \neg(\mathsf{EX}_{\bar{R}}(\bar{S} \wedge \neg Z)) = \bar{S} \wedge \neg(\alpha(\mathsf{EX}_{R}(\bar{S} \wedge \neg Z))).$$

The above solutions for the EG modality—a *greatest* fixpoint—and the universal modalities require the set $\bar{S}$ of all representatives (unlike the case of EF [reachability] and EU, where only representatives encountered during the computation are stored). Depending on the application and the definition of representatives, the BDD for this set can be (but is not always) costly. It can be computed as $\alpha(\mathit{true})$, but the "direct" way based on the expression $\bigcap_{p<n} p \leq_z p+1$ is often more efficient. Other than $\bar{S}$, the above equations only involve boolean primitives, existential backward image with respect to R, and the abstraction function α. This makes our technique complete for CTL.

7 Experimental Results

We present results of verifying example systems using our technique, with respect to properties that came along with the system specification. Our tool uses the CUDD BDD package [Som01]. We ran the examples on an i686/1400 Mhz PC with 256MB main memory. In tables, the figure behind the name of an example indicates the number of components involved. "Number of BDD Nodes" refers to the peak number of BDD nodes allocated at any time during execution. It represents the memory bottleneck of the verification run. The abbreviations s, m, h, M stand for seconds, minutes, hours, and million, respectively.

Table 1. Comparison to Multiple Representatives and Counter Abstraction

	Multiple Representatives		Counter Abstraction		Dynamic Symmetry Reduction	
Problem	Number of BDD Nodes	Time	Number of BDD Nodes	Time	Number of BDD Nodes	Time
MsLock 10	369,239	1:15m	68,154	29s	24,092	15s
MsLock 20	4,407,127	4:05h	325,325	7:06m	139,990	9:35m
MsLock 30	(>13M)	(>28h)	725,672	24:26m	375,649	1:23h
CCP 03	16,522,710	13:12h	1,988,991	7:55m	14,088	1s
CCP 05	(>12M)	(>35h)	4,001,573	1:49h	74,754	14s
CCP 10	—	—	(>14M)	(>18h)	1,075,206	26:35m
CCP 15	—	—	—	—	4,947,726	6:17h

Table 2. Comparison to unreduced Model Checking and Multiple Representatives

	Without Symmetry Reduction		Multiple Representatives		Dynamic Symmetry Reduction	
Problem	Number of BDD Nodes	Time	Number of BDD Nodes	Time	Number of BDD Nodes	Time
Comp&Swap 40	376,681	1m	157,470	25s	48,433	10s
Comp&Swap 50	(>14M)	(>24h)	4,259,627	37:34m	419,529	4:03m
Comp&Swap 60	—	—	(>10M)	(>24h)	6,246,717	2:10h
Fetch&Store 40	1,083,830	4:12m	413,036	2:02m	160,628	40s
Fetch&Store 50	(>12M)	(>24h)	(>11M)	(>24h)	2,017,634	29:43m
Fetch&Store 60	—	—	—	—	(>12M)	(>24h)
Distrib. List 30	861,158	28s	708,339	20s	60,394	2s
Distrib. List 40	6,380,209	4:35m	2,963,024	2:37m	213,448	5s
Distrib. List 50	(>15M)	(>24h)	13,580,042	29:30m	271,366	11s

In Table 1, we compare our dynamic symmetry reduction technique to the aforementioned alternative methods, Multiple Representatives and Counter Abstraction. To ensure fair comparison, we set various BDD parameters individually for each technique such that it performed best. The MsLock example is a simplified model of a queuing lock algorithm [MS91]. The simplification was necessary to make the system amenable to counter abstraction. The example denoted CCP refers to a buggy version of a cache coherence protocol suggested by S. German, see for example [LS03]. Due to the presence of errors, parameterized model checking (for arbitrary n) initially fails on this protocol (an inductive invariant proving the safety property does not exist). Model checkers such as our tool can then be used to provide an error trace for fixed values of the size parameter. This example is characterized by a large number of local states, which is why counter abstraction performs much worse on it than our dynamic technique.

The Multiple Representatives approach suffers from the high cost of building the representative mapping ξ.

The second table presents examples to which counter abstraction can not be applied. The reason is that here permutations act upon states by not only changing the order of local state components, but also their values. "Comp&Swap" and "Fetch&Store" are two versions of the queuing lock [MS91], a simplification of which was used in the MsLock example above. The "Distrib. List" example is a distributed protocol for processes in a FIFO queue sending and receiving messages, acting as a relay if asked to do so [MD96]. Symmetry exists in both the processes and the messages. In this table we also show results of the verification run *without* symmetry reduction, where the intermediate BDDs become huge quickly. Our technique invariably outperforms the other two, for large problem instances by orders of magnitude.

8 Summary

In this paper, we have presented a dynamic symmetry reduction technique that surpasses, to the best of our knowledge, previously known techniques dramatically. Multiple Representatives suffer from symptoms similar to those of orbit relation-based approaches (although alleviated). Counter abstraction is often efficient if operative, but does not scale well for systems of components with a large local state space, requires full symmetry, and is only applicable to symmetries with "simple" permutation action. In contrast, our solution is not based on counters and thus more flexible, yet it does not suffer from the problems associated with storing pairs of states and their representatives.

Our method can generally be seen as a symbolic abstraction technique that avoids pre-computing the abstraction function, but rather offers an efficient symbolic algorithm to map concrete to abstract states *on the fly*. In connection with symmetry reduction, there was a need for such a technique, due to the ongoing difficulties with the orbit relation.

Bubble sort is traditionally regarded naive and not successful on large sorting problems. Our decision to use it in the representative mapping under full symmetry is an instance of a phenomenon often seen in parallel programming: The most clever and sophisticated sequential algorithms are not always the best in a new computational model. Instead, a simple-minded routine can prove very suitable. In our case, we believe that the *locality* of bubble sort, i.e. its affecting only nearby elements and being in-place, is paramount.

Related Work. In addition to the references mentioned in the introduction, the work closest to ours is the paper by Barner and Grumberg [BG02], who considered combining symmetry and symbolic representation using BDDs mainly for falsification. If too large, the set of reached representatives is under-approximated, which renders the algorithm inexact. Also, their work uses multiple representatives and therefore forgoes some of the symmetry reduction possible. Finally, there is a lot of other work on symmetry not directly related to symbolic representation [PB99, God99, SGE00, HBL+03–among many].

References

[BG02] Sharon Barner and Orna Grumberg. Combining symmetry reduction and under-approximation for symbolic model checking. *Computer-Aided Verification (CAV)*, 2002.

[Bry86] Randy E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 1986.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Principles of Programming Languages (POPL)*, 1977.

[CE81] Edmund M. Clarke and E. Allen Emerson. The design and synthesis of synchronization skeletons using temporal logic. *Logic of Programs (LOP)*, 1981.

[CEFJ96] Edmund M. Clarke, Reinhard Enders, Thomas Filkorn, and Somesh Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design (FMSD)*, 1996.

[ES96] E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. *Formal Methods in System Design (FMSD)*, 1996.

[ET99] E. Allen Emerson and Richard J. Trefler. From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. *Conference on Correct Hardware Design and Verification Methods (CHARME)*, 1999.

[EW03] E. Allen Emerson and Thomas Wahl. On combining symmetry reduction and symbolic representation for efficient model checking. *Conference on Correct Hardware Design and Verification Methods (CHARME)*, 2003.

[God99] Patrice Godefroid. Exploiting symmetry when model-checking software. *Formal Methods for Protocol Engineering and Distributed Systems (FORTE)*, 1999.

[HBL+03] Martijn Hendriks, Gerd Behrmann, Kim Guldstrand Larsen, Peter Niebert, and Frits W. Vaandrager. Adding symmetry reduction to uppaal. *Formal Modeling and Analysis of Timed Systems (FORMATS)*, 2003.

[ID96] C. Norris Ip and David L. Dill. Better verification through symmetry. *Formal Methods in System Design (FMSD)*, 1996.

[LS03] Shuvendu Lahiri and Sanjit Seshia. *UCLID: A Verification Tool for Infinite-State Systems*. `http://www-2.cs.cmu.edu/~uclid/`, 2003.

[MD96] Ralph Melton and David L. Dill. *Murφ Annotated Reference Manual, rel. 3.1*. `http://verify.stanford.edu/dill/murphi.html`, 1996.

[MS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 1991.

[PB99] Manish Pandey and Randal E. Bryant. Exploiting symmetry when verifying transistor-level circuits by symbolic trajectory evaluation. *IEEE Transactions on Computer-Aided Design*, 1999.

[PXZ02] Amir Pnueli, Jessie Xu, and Leonore Zuck. Liveness with $(0, 1, \infty)$-Counter abstraction. *Computer-Aided Verification (CAV)*, 2002.

[QS82] Jean-Pierre Quielle and Joseph Sifakis. Specification and verification of concurrent systems in cesar. *5th International Symposium on Programming*, 1982.

[SGE00] A. Prasad Sistla, Viktor Gyuris, and E. Allen Emerson. Smc: a symmetry-based model checker for verification of safety and liveness properties. *ACM Transactions on Software Engineering and Methodology*, 2000.

[Som01] Fabio Somenzi. *The CU Decision Diagram Package, release 2.3.1*. `http://vlsi.colorado.edu/~fabio/CUDD/`, 2001.