On Combining Symmetry Reduction and Symbolic Representation for Efficient Model Checking

E. Allen Emerson and Thomas Wahl

Department of Computer Sciences and Computer Engineering Research Center
The University of Texas, Austin/TX 78712, USA

Abstract. BDDs allow succinct symbolic representation of digital circuits. Symmetry reduction factors out redundancy inherent in the regular organization of many systems. Both are successful techniques for combating state space explosion. It would be desirable to combine them into *symbolic symmetry reduction*. Unfortunately, the straight-forward approach to symmetry reduction requires the *orbit relation*, whose symbolic representation as a BDD is in general of exponential size. We investigate the use of *generic representatives* as a means of overcoming this problem for *fully symmetric* systems: instead of first representing the system as a BDD and then applying symmetry reduction, we translate the given program text into a symmetry-reduced version. The result can then be encoded using a BDD. We demonstrate that this method is superior not only to the traditional orbit-relation based symmetry reduction, but also to the approach using *multiple representatives*.

1 Introduction

Symbolic representation of systems, most notably in the form of binary decision diagrams (BDDs), is often more compact than explicit, enumerative representation. *Symmetry reduction* is a powerful technique to limit the state space explosion problem. In symmetric systems, two states are considered equivalent if they are identical up to certain permutations of the participating processes. This relation gives rise to equivalence classes of states, called *orbits*. The Kripke structure built over the orbits can be shown to be bisimulation-equivalent to the structure built over individual states.

The combination of symbolic representation with symmetry reduction was investigated in [CEFJ96]. The paper describes how the BDD for the *representative function* can be constructed, which maps a state to its unique orbit representative. Symbolic model checking in the presence of symmetry is then implemented by applying the representative function to the intermediate results during fixpoint evaluations.

Computing the representative function requires the *orbit relation*, which contains pairs of states that are permutations of each other. The orbit relation turned out to be the bottleneck of symbolic symmetry reduction, since its BDD is, for many underlying symmetry groups, of size exponential in the minimum of the number of components and

This work was supported in part by NSF grants CCR-009-8141 and CCR-020-5483, and SRC contract 2002-TJ-1026. Email: {emerson|wahl}@cs.utexas.edu

the number of states per component. A partial remedy is to permit *multiple representatives* per orbit, which might be computable without using the orbit relation. However, choosing too many representatives per orbit is contrary to the purpose of symmetry reduction.

To overcome the limitations of the above approach [CEFJ96] to combining symmetry reduction and symbolic representation using BDDs, we investigate, for *fully symmetric* systems, the use of *generic representatives* [ET99] as a means of avoiding the problems associated with picking representative states. Instead of first building a BDD for the system and then implementing symmetry reduction via the orbit relation, the symmetry is factored out at the source code level by compiling the original program into one that operates on counter variables. To keep track of equivalence classes of states, it is sufficient to store the *number* of processes in a given location, rather than their identities. For example, in a system with process locations N, T and C, the states (N, N, T, C), (N, C, T, N), and (T, N, N, C) are all symmetry-equivalent and can be represented generically as $(2N, 1T, 1C)$.

In this paper we show how this idea can be applied to practical systems, where processes communicate via shared variables. In many applications, a global variable is used to point to one distinguished process, like one that possesses a token, or one that is currently allowed to enter its critical section. Since generic representatives get rid of process identities, such a variable must be adapted to a generic program. We show how this can be done by replacing it with a new variable that keeps track of the location of the distinguished process, rather than its identity. This method presents a slight challenge, though: if the distinguished process executes a transition, then its identity remains the same, but its location changes. This change must be reflected in the new variable. The complexity of a transition might therefore grow when translated into its generic form, although only by a small constant amount.

We place suitable restrictions on the use of those global variables containing process indices in guards and actions in a program to ensure full symmetry. We also show the details of the program translation. We then define Kripke structures derived from the original and translated programs, respectively, and establish their bisimilarity. The generic method preserves all of the symmetry reduction, is applicable to a large class of fully symmetric systems and is efficient; in particular, it completely avoids the orbit relation. We demonstrate its usefulness in symbolic symmetry reduction by presenting experimental results for two systems with unique, multiple and generic representatives.

The remainder of this paper is organized as follows. In section 2, we review traditional symmetry reduction with BDDs. In section 3 we illustrate, by means of an example, the notion of generic representatives. Section 4 formally describes how to translate a program given as a synchronization skeleton into an “equivalent” generic program. The translation of this new program into BDDs is the topic of section 5. We compare our method against other symbolic symmetry reduction techniques experimentally in section 6. Related and future work are discussed in the concluding section 7.

2 Preliminaries

Notation. For $a, b \in \mathbb{N}$, we denote by $[a..b]$ the set $\{i \in \mathbb{N} : a \leq i \leq b\}$. For a permutation π, the symbol π^- stands for its inverse. In programs, we use an imperative language style syntax. Block structure is indicated by indentation (instead of `begin`/`end`); comments go from "//" to the end of the line.

2.1 Permutations Acting upon States

The ideas presented in this paper apply to *process symmetries*, which describe the phenomenon that in a system of replicated process components, processes can be rearranged in certain ways simultaneously in the source and target state of all transitions of the system, without changing the overall transition relation. This can be formalized as follows. The systems under consideration are similar to *shared variable programs* [ES96]. We assume there are n concurrently executing processes, following an interleaved model of computation, which share some global variables. The possible local states of a process are given by a set of process locations $\mathcal{L}$. A system state can therefore be written as $s = (v, l_1, \ldots, l_n)$, where v is a (possibly tuple-valued) global variable and $l_i \in \mathcal{L}$ is the location of process i. For $L \in \mathcal{L}$, we use L_i as a shorthand for the expression $l_i = L$. The rearrangement of processes in a state is formalized in terms of a permutation $\pi : [1..n] \to [1..n]$ acting upon process indices. The mapping π can be extended to act on system states by defining $\pi(s) = (v^\pi, l_{\pi(1)}, \ldots, l_{\pi(n)})$, where v^π describes the result of π acting on v. The definition of v^π depends on the character of v: some global variables, like a binary *semaphore*, are invariant under permutations, such that $v^\pi = v$. On the other hand, a global *token* variable pointing to (the id of) some process is directly affected by π, as we shall see in section 3. In this case, one defines $v^\pi = \pi(v)$.

2.2 Symmetry Reduction in Theory

Given a set of permutations G acting on $[1..n]$, a Kripke structure $M = (S, R, s_0)$, and a definition of $\pi(s)$ for $s \in S$, we say that M is *symmetric with respect to* G if, for all $\pi \in G$, $\pi(R) := \{(\pi(s), \pi(t)) : (s,t) \in R\}$ satisfies $\pi(R) \subset R$. In this case, it can be shown that in fact $\pi(R) = R$, and that G is a group with function composition as the group operation. If G contains *all* permutations over $[1..n]$, M is called *fully symmetric*. This paper focuses on fully symmetric systems. G, the *full symmetry group*, is therefore henceforth omitted.

The *orbit relation* $\theta(s,t) := \exists \pi : \pi(s) = t$ defines an equivalence between states; the equivalence classes it entails are called *orbits*. Instead of considering all states in S, it suffices now to choose a small set *Rep* of representatives. This choice is reflected by the *representative relation* $\xi \subset S \times Rep$, which assigns to every state in S elements of *Rep* such that:

soundness: for all $(s,r) \in \xi$, there exists π such that $\pi(s) = r$ (i.e. $\xi \subset \theta$), and
totality: for all $s \in S$, there exists $r \in Rep$ such that $(s,r) \in \xi$.

The symmetry-reduced transition relation is obtained by replacing source and target of edges in R by representatives:

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

The structure $\bar{M} := (\mathit{Rep}, \bar{R}, \bar{s}_0))$, for any $\bar{s}_0$ with $(s_0, \bar{s}_0) \in \xi$, is called the *quotient model* of M. For suitable choices of Rep and ξ it can be shown that $\bar{M}$ is bisimulation-equivalent to M, and therefore

$$M, s \models f \quad \Leftrightarrow \quad \bar{M}, \bar{s} \models f \tag{2}$$

for every $\bar{s}$ such that $(s, \bar{s}) \in \xi$ and every formula f such that for all π, every $u \in S$ and every maximal propositional subformula p appearing in f, $M, u \models p \Leftrightarrow M, \pi(u) \models p$.[1] The "suitable choices" for Rep and ξ turn out to be crucial for efficiency.

2.3 Unique Representatives

It seems natural to pick exactly one representative from each orbit, such that the relation ξ becomes a function. For instance, given a system state as an n-tuple over the process locations $\mathcal{L}$, ξ could return the tuple with the locations sorted according to some ordering within $\mathcal{L}$ [LN91]. This mapping is sound, since sorting amounts to applying a permutation. It is also total, since every system state can be sorted in this way. Finally, the structure $\bar{M}$ derived from this choice of Rep is indeed bisimulation-equivalent to M.

The only currently known way to construct a BDD for ξ with unique representatives is by first building the BDD for the orbit relation θ and then projecting the second component of θ onto Rep: $\xi = \{(s, r) \in \theta : r \in \mathit{Rep}\}$. Unfortunately, this approach is generally problematic [CEFJ96]: the BDD for the orbit relation is, for many common symmetry groups including the full symmetry group, of size at least $\min\{2^n, 2^{|\mathcal{L}|}\}$ and therefore requires both exponential space and time.

2.4 Multiple Representatives

A computationally less expensive choice of Rep and ξ is possible if the uniqueness requirement for the representatives is dropped. This approach imposes a few weaker constraints on Rep and ξ, which we sketch here briefly; for details see [CEFJ96].

Definition 3 ([CEFJ96]) *Let Rep be a set of representatives and ξ a sound and total representative relation. A set C of permutations is* complete *if:*

- *for all $(s, r) \in \xi$, there exists $\pi \in C$ such that $\pi(s) = r$, and*
- *for all $\pi \in C$ and $r \in \mathit{Rep}$, $(\pi(r), r) \in \xi$.*

Notice that if the representatives are unique as in 2.3, the full symmetry group G is a complete set. We hope, however, to find a small complete subset C. Intuitively, we can then restrict our attention to permutations from C in the search for representatives of a given state.

[1] The latter equivalence is guaranteed if $p \equiv \pi(p)$ is a tautology

Theorem 4 ([CEFJ96]) *Let Rep and ξ be as in definition 3. If there exists a complete set C, then $M, s \models f \Leftrightarrow \bar{M}, \bar{s} \models f$ with $\bar{M}$, $\bar{s}$ and f as in (2).*

In practice, it is the programmer's responsibility to first define a set *Rep* representable by a small BDD. In [CEFJ96], it is described how a suitable set C can be derived. By finally defining ξ as

$$(s, r) \in \xi \quad \text{iff} \quad r \in \mathit{Rep} \, \wedge \, \exists \pi \in C : \pi(s) = r, \tag{5}$$

C is a complete set for *Rep* and ξ. If the expression $\exists \pi \in C : \pi(s) = r$ and R can also be encoded succinctly, then the BDD for $\bar{R}$ as in (1) is small; the orbit relation is nowhere used. By theorem 4, we can now perform model checking on $\bar{M}$.

The symmetry reduction effect is negatively impacted by choosing several representatives per orbit. While this could still be advantageous when using BDDs, it is not clear that *Rep* can always be chosen to allow a small BDD for ξ and $\bar{R}$. In the remainder of this paper, we argue that in the case of full symmetry, a solution exists that avoids all these problems altogether.

3 Generic Representatives – a Case Study

Full symmetries occur frequently in practice, whenever a system is composed of unordered, pairwise interchangeable components. This is the case for clique networks of processes, but also for bus and star topologies, where components communicate via a centralized hub (such as in cache coherence protocols). In the latter cases, the bus or hub can be "factored out", while full symmetry reduction can be applied to the remaining processes.

A fully symmetric system is concisely specified by the number n of processes, possible global variables with initial values, and the common program executed by all processes. As an abstraction of this program, we assume, for the purpose of describing the formal translation into BDDs, the input model of *synchronization skeletons*. These skeletons are appropriate and powerful enough to describe most control-intensive synchronization problems over finite domains. Combinations of values for the local variables of a process are abstracted into a local state; assignments to those variables are represented as local state changes. Sequential code executed by a process in an atomic action is abstracted into a single transition.

As an example, consider a token-based solution to the n-process Mutual Exclusion problem with a global variable $\mathit{tok} \in [1..n]$, and the skeleton in figure 1. A skeleton's arcs can be labeled with *guards* (shown in the diagram above the arc) and actions (shown below it, executed after the transition). The skeleton in the figure allows a process to enter its critical section C if it currently possesses the token ($\mathit{tok} = \mathit{self}$). Upon leaving C, it sets tok to a nondeterministic value in $[1..n]$. The skeleton gives rise to a fully symmetric structure, as we will see in the next section for skeletons written in a specific input syntax.

We now want to construct a new program based on counters that yields a bisimulation-equivalent structure. Instead of a local state variable for each process, we somewhat conversely declare global counter variables for each local state, calling them n_N,

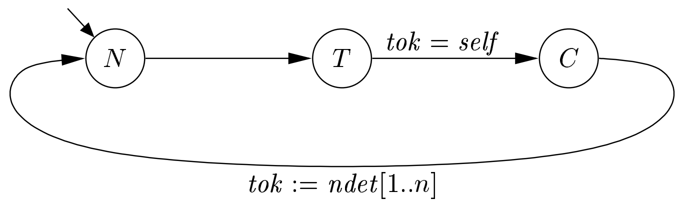

Fig. 1. The synchronization skeleton for a token solution to the Mutual Exclusion problem

n_T, n_C. A slight challenge is provided by the tok variable with range $[1..n]$. Since the counter variables deliberately ignore process identities, we cannot check a guard like $tok = self$ any more. However, notice that states with equal counter values are equivalent as long as the *locations* of the respective token processes are the same, such as in $(tok = 2, N, T, C)$ and $(tok = 3, N, C, T)$. It therefore suffices to remember, in a new variable TOK, the location of the token process. Thus, TOK ranges over $\{N, T, C\}$.

The translated program consists of the variables and statements shown in figure 2. The values of the counter variables range from 0 to the number of processes, n. The

Variables:		**Initial values:**	
n_N, n_T, n_C	$: [0..n]$	(n_N, n_T, n_C)	$:= (n, 0, 0)$
TOK	$: \{N, T, C\}$	TOK	$:= N$

// from transition
// $N \to T$:
if $n_N > 0$
  if $TOK = N$
    if $n_N = 1$
      $TOK := T$
    else
      $TOK := \{N, T\}$
  $n_N := n_N - 1$
  $n_T := n_T + 1$

// from transition
// $T \xrightarrow{tok=self} C$:
if $n_T > 0 \wedge TOK = T$
  $TOK := C$
  $n_T := n_T - 1$
  $n_C := n_C + 1$

// from transition
// $C \to N$, $tok := ndet[1..n]$:
if $n_C > 0$
  $n_C := n_C - 1$
  $n_N := n_N + 1$
  $TOK := ndet\{L : n_L > 0\}$

Fig. 2. Generic version of the token-based Mutual Exclusion solution

initial values of all four variables follow from the fact that all processes start out in location N. All transitions in the new program require that the counter of the source state is positive, since the transition can be taken only if there is a process in that state.

The first transition, $N \to T$, has apparently nothing to do with the token, since tok does not explicitly appear in it. However, the process executing it might be the one possessing the token, in which case the new variable TOK must be updated from N to T. If $TOK = N$ and $n_N = 1$, then the executing process has the token, and we set TOK to T. If $TOK = N$ and $n_N > 1$, then the process executing the transition may or may not be the one possessing the token, so we must set TOK to T, or TOK

must remain at N, respectively. Hence, the new program has two transitions in this case, which we abbreviate by a nondeterministic assignment $TOK := \{N, T\}$. Finally, the actual location change is reflected by decreasing n_N and increasing n_T. A similar, but simpler reasoning motivates the translation of the other two transitions; in particular, the condition $n_L > 0$ in the assignment to TOK in the last statement ensures that only locations in which there is at least one process are nondeterministically chosen.

The property to be verified also needs to be translated into counters. As an example, compare the mutual exclusion (safety) and communal progress (liveness) requirements in specific and generic notation:

	specific	generic
Safety:	$\mathsf{AG}\,\forall i,j : i \neq j : \neg(C_i \wedge C_j)$	$\mathsf{AG}(n_C < 2)$
Liveness:	$\mathsf{AG}(\exists i T_i \Rightarrow \mathsf{AF}\,\exists j C_j)$	$\mathsf{AG}(n_T > 0 \Rightarrow \mathsf{AF}\,n_C > 0)$.

The liveness property states that if there is *some* process in its trying region, then in any possible future, there should eventually be *some* process entering its critical section. This property is weaker than progress of an individual process, formally $\mathsf{AG}\,\forall i : (T_i \Rightarrow \mathsf{AF}\,C_i)$. The latter formula, however, is not symmetric, since the maximal propositional subformula T_i is not invariant under permutations. It can therefore not directly be verified over a symmetry reduced structure (whether specific or generic). One approach to overcoming this problem is to "factor out" one of the processes and treat its local variables as global. The progress property is formulated for this process, and symmetry reduction is applied to the remaining ones. This approach is described in more detail by Pnueli, Xu, and Zuck [PXZ02], incidentally for counter-abstracted programs.

To see that implementing the above translation is tantamount to performing symmetry reduction on the program text, notice that all states from one equivalence class of the original system are mapped by the translation to the same counter tuple (TOK, n_N, n_T, n_C). This tuple can therefore be viewed as an "unusual notation" for the representative of the orbit—we call it a *generic representative*. The new program can now be transformed into a Kripke structure, represented by BDDs, and model checked.

4 Translating Symmetric Programs into Generic Form

The global variable *tok* in the previous section contains a process index, which is lost after the introduction of counters. Such variables require special treatment during the translation process, since permutations act upon them "directly" by changing their index values. We call them *id-sensitive*. Global variables independent of process identities, for example a boolean *semaphore*, are, as we shall see, much simpler to handle. We refer to them as *id-independent* variables; permutations leave them invariant.

We assume a program $\mathbb{P}$ in the form of the following parameters: (1) the number n of processes, (2) any number of id-independent global variables, given as a single vector $\vec{v}$ with range V (cross product of individual ranges), along with initial value $\vec{x}_0$, (3) any number z of id-sensitive global variables, given as $\vec{d} = (d_1, \ldots, d_z)$ with range $[1..n]^z$, along with initial value $\vec{k}_0$, and (4) a synchronization skeleton. The latter is a finite directed graph, each node of which represents (and is identified with) a process location; call their number l. One of the nodes, I_0, is the distinguished initial location of every process. The edges may be labeled with a guard and an action (which default to $true$ and *no-op*, respectively).

Syntax of Guards. Guards are arbitrary propositional combinations of boolean-valued *basic guards*, the latter being conditions on process locations and expressions over global variables. In order to ensure full symmetry of the structure entailed by the program, basic guards must meet certain criteria.

Definition 6 *For a quantified boolean formula h over atoms of the form L_i, $i \in [1..n]$, and a permutation π on $[1..n]$, define $\pi(h)$ by π acting upon the indices. Formula h is* fully symmetric *if for every π, $h \Leftrightarrow \pi(h)$ is a tautology.*

Some basic guards satisfying this definition are listed in table 1. As an example, the

Table 1. Fully symmetric basic guards on process locations

no.	Basic Guard	Generic version	Meaning
0	$\forall i : \neg L_i$	$n_L = 0$	none
1	$\forall i : L_i$	$n_L = n$	all
2	$\exists i, j : i \neq j : L_i \wedge L_j$	$n_L \geq 2$	at least two

guard, *exactly one process is in location L*, formally $(\exists i : L_i) \wedge (\forall i, j : L_i \wedge L_j \Rightarrow i = j)$, is equivalent to $\neg 0 \wedge \neg 2$, where 0 and 2 are two basic guards from the table. It is more succinctly written as $n_L = 1$ in generic terms.

Any (syntactically valid) expression over id-independent global variables is "by nature" fully symmetric and thus a legal basic guard. As for an id-sensitive variable d, we allow the expressions $d = \mathit{self}$ and $d \neq \mathit{self}$ as basic guards.

Syntax of Actions. An action consists of at most one assignment to each of the global variables. The execution model for the assignments—e.g. parallel or sequential—is left to the implementation, since it is irrelevant for the translation of the source program into generic representatives.

As with guards, to ensure full symmetry the syntax of actions must be restricted. Any (syntactically valid) assignment to the id-independent variables is legal, since it does not affect the symmetry of the program. For an id-sensitive variable d we allow the following three types:

$$d := \mathit{self} \qquad d := \mathit{ndet}[1..n] \qquad d := \mathit{ndet}([1..n] \backslash \{\mathit{self}\}).$$

The last two actions intuitively assign a nondeterministic value in the given set to d. Their precise semantics is given by the derivation of a Kripke structure:

Definition 7 *A program specified in the above syntax defines a Kripke structure $M = (S, R, s_0)$ with $S = V \times [1..n]^z \times [1..l]^n$, $s_0 = (\vec{x}_0, \vec{k}_0, I_0, \ldots, I_0)$, and R containing all pairs (s, t) with*

$$s = (\vec{x}, \vec{k}, l_1, \ldots, l_{i-1}, A, l_{i+1}, \ldots, l_n), \quad t = (\vec{x}', \vec{k}', l_1, \ldots, l_{i-1}, B, l_{i+1}, \ldots, l_n)$$

such that there is an edge $e : A \to B$ *in the skeleton with a guard that evaluates to true for* $\vec{v} = \vec{x}$, $\vec{d} = \vec{k}$, $self = i$ *and process locations as in s, and e's action* $\mathcal{A}$ *satisfies the Hoare triple* $\langle \vec{v} = \vec{x} \rangle \mathcal{A} \langle \vec{v} = \vec{x}' \rangle$, *and for each id-sensitive variable d with values k and* k' *in s and t, resp.,* $\mathcal{A}$ *has an assignment* $d := self$ *and* $k' = i$, *or an assignment* $d := ndet Z$ *for some* Z *with* $k' \in Z$, *or* $\mathcal{A}$ *has no assignment to d and* $k' = k$.

The following theorem shows that symmetry reduction can be applied to M.

Theorem 8 *For* $s = (\vec{x}, k_1, \ldots, k_z, l_1, \ldots, l_n)$, *let* $\pi(s) = (\vec{x}, \pi^{-}(k_1), \ldots, \pi^{-}(k_z), l_{\pi(1)}, \ldots, l_{\pi(n)})$ *and* $\pi(R)$ *as in section 2.2.* M *is fully symmetric, that is,* $\pi(R) \subset R$ *for all permutations* π.

We are now ready to describe the translation of program $\mathbb{P}$ from its components (1) through (4) (beginning of this section): The new program $\hat{\mathbb{P}}$ consists of the same variable $\vec{v}$ with initial value $\vec{x}_0$, further variables $\hat{d}_j$, $j \in [1..z]$, with range $[1..l]$ and common initial value I_0, and variables $n_1, \ldots, n_l$ with range $[0..n]$ and initial values $n_{I_0} = n$, $n_L = 0$ for $L \neq I_0$. Every edge of the skeleton is translated into a statement as follows:

$$A \xrightarrow[\;action\;]{\;guard\;} B$$

if $n_A > 0 \;\wedge\; gen(guard)$
$\quad update1(guard)$
$\quad n_A := n_A - 1$
$\quad n_B := n_B + 1$
$\quad update2(action)$ (9)

The condition $n_A > 0$ ensures that there is a process in location A. The $guard$ is translated by a function gen as follows: each basic guard on process locations is replaced according to table 1. For an id-sensitive variable d_j, guard $d_j = self$ is replaced by $\hat{d}_j = A$, guard $d_j \neq self$ by $\hat{d}_j \neq A \vee n_A \geq 2$ (if $n_A \geq 2$, there is a process i in location A with $d_j \neq i$; hence $d_j \neq self$ is true for that process). Expressions over $\vec{v}$ are unchanged.

Function $update1$ performs updates of variable $\hat{d}_j$ that become necessary because of the location change. It is only required if $action$ does not assign to d_j (if it does, $\hat{d}_j$ is overwritten by $update2(action)$).

$guard$	$d_j = self$	$d_j \neq self$	otherwise (including $true$)

$update1(guard)$	$\hat{d}_j := B$	*no-op*	**if** $\hat{d}_j = A$
			  if $n_A = 1$
			    $\hat{d}_j := B$
			  else
			    $\hat{d}_j := ndet\{A, B\}$

Function $update2$ implements updates of $\vec{v}$ and $\hat{d}_j$ that are due to the $action$. It leaves *no-op* and assignments to $\vec{v}$ unchanged. For the assignments to d_j, we translate as follows:

$action$	$d_j := self$	$d_j := ndet([1..n] \setminus \{self\})$	$d_j := ndet[1..n]$
$update2$ $(action)$	$\hat{d}_j := B$	**if** $n_B = 1$ $\hat{d}_j := ndet(\{L : n_L > 0\} \setminus \{B\})$ **else** $\hat{d}_j := ndet\{L : n_L > 0\}$	$\hat{d}_j := ndet\{L : n_L > 0\}$

Definition 10 *Program $\hat{\mathbb{P}}$ defines a Kripke structure $\hat{M} = (\hat{S}, \hat{R}, \hat{s}_0)$ with $\hat{S} = V \times [1..l]^z \times [1..n]^l$, $\hat{s}_0 = (\vec{x}_0, I_0, \ldots, I_0,\ n_1, \ldots, n_l)$ such that $n_{I_0} = n$, $n_L = 0$ for all $L \neq I_0$, and $\hat{R}$ containing all pairs $(\hat{s}, \hat{t})$ such that there is a (nondeterministic) statement in $\hat{\mathbb{P}}$ whose top-level condition $n_A > 0 \wedge gen(guard)$ evaluates to true and that contains an execution that, applied to $\hat{s}$, results in $\hat{t}$.*

Theorem 11 *Structures M (definition 7) and $\hat{M}$ are bisimulation-equivalent via*

$$b : S \to \hat{S},\ b(\vec{x}, k_1, \ldots, k_z, l_1, \ldots, l_n) = (\vec{x}, l_{k_1}, \ldots, l_{k_z}, n_1, \ldots, n_l)$$

with $n_L := |\{j \in [1..n] : l_j = L\}|$. Function b maps every state to its unique generic representative. The following theorem shows that although generic representatives are not based on permutations, they define the same equivalence classes as the orbit relation:

Theorem 12 *For any $r, s \in S$, $b(r) = b(s)$ if and only if $\exists \pi : \pi(r) = s$.*

In order to model check over structure $\hat{M}$, the specification must be rewritten in generic notation. We assume it is a CTL formula whose atomic propositions are fully symmetric expressions on local state variables (translated like the examples in table 1) and expressions on the id-independent global variable (unchanged). Such a formula is symmetric in the sense defined right below (2).

Note that the translation of the program as well as of the formula can be done fully automatically, in time linear in the size of the program text.

5 Translating Generic Programs into BDDs

In this section, we show how the statements of the generic program, obtained in section 4, can be encoded in a BDD efficiently. We will also estimate the sizes of those BDDs, depending on n, l and the size of the input synchronization skeleton. In this section we ignore the existence of the id-independent variable $\vec{v}$: since expressions involving it are subject to no restrictions, BDD sizes cannot be estimated. However, those expressions are not altered during the translation; hence they do not contribute any change in BDD size.

The generic structure $\hat{M} = (\hat{S}, \hat{R}, \hat{s}_0)$ is the disjunction of statements of the form in (9) in section 4. BDDs implementing those statements can be obtained as follows:

$n_A > 0$ iff there is at least one *true* bit among the $\lceil \log(n+1) \rceil$ bits representing n_A. This can be implemented as a disjunction over all those bits. The resulting BDD size is linear in the number of participating bits: $\mathcal{O}(\log n)$.

$gen(guard)$ is a propositional combination of basic generic guards. Guards from table 1 can be realized as above with a BDD that compares the constant bit-wise against the counter variable; size $\mathcal{O}(\log n)$. Basic generic guards involving the id-sensitive variable have the form $\hat{d} = A$ or $\hat{d} \neq A \vee n_A \geq 2$, which can again be verified bit-wise; these BDDs thus have maximum size $\mathcal{O}(\log l \log n)$ ($\hat{d} \in [1..l]$, $n_A \in [0..n]$).

Let F denote the number of basic guards appearing in $guard$. The total BDD size for this part of a transition is then no more than $\mathcal{O}((\log l \log n)^F)$. Since F is typically a small constant, this bound is usually polynomial in practice.

$update1(guard)$**:** an **if-then-else** statement can be implemented using the common ITE operation for BDDs. Since the expressions contained inside the **if-then-else** are again comparisons against constants, the entire statement can be encoded in a BDD of size $\mathcal{O}(\log^\alpha l \cdot \log^\beta n)$, for small constants α and β.

$n_L := n_L \pm 1$**:** since the right-hand side is not a constant, a bit-wise comparison is not possible. The increment can be implemented by searching (using existential quantification) for a bit position i at which n_L is 0, n'_L (the next-state value) is 1, for all preceding bits n_L and n'_L are identical, and for all succeeding bits n_L is 1 and n'_L is 0. The worst-case BDD size over two variables of $\lceil \log(n+1) \rceil$ input bits is $2^{2\lceil \log(n+1) \rceil} = \mathcal{O}(n^2)$.

$update2(action)$**:** assignment $\hat{d} := ndet\{L : n_L > 0\}$ can be realized with a BDD for the expression $\bigvee_{L \in [1..l]} (n_L > 0 \wedge \hat{d}' = L)$ of size $\mathcal{O}((\log n \log l)^l)$. The BDD for the **if-then-else** statement then has size $\mathcal{O}(\log^2 n \cdot (\log n \log l)^{2l})$.

Assuming (very defendably) that F is a small constant, we can see that all parts of the translation of an edge can be expressed with a BDD that is low-degree polynomial in n, although, with respect to l, it can be of order $(\log l)^{2l}$ (caused by the $\hat{d} := ndet\{L : n_L > 0\}$ statement). The complexity of the overall transition relation depends on the way the individual statements are combined, but it is guaranteed to be polynomial in n as well.

It is interesting to investigate how the relative sizes of n and l influence the benefit of generic representatives. Because of the $n \log l$ input variables of the BDDs for the specific representatives algorithm (n variables of range $[1..l]$ for θ, ξ and $\bar{R}$), hence a maximum specific BDD size of roughly l^n, it can be assumed that the generic method is most useful if n is larger than l. Asymptotically, this is the case if l is a constant and n is considered variable. This situation occurs frequently in practice, since, for a given application, the number l of local states is often fixed. Our second experimental case, presented in the next section, is such an instance.

6 Experimental Results

We compare traditional to generic symmetry reduction using two examples:

The first is an artificial Mutual Exclusion scenario that allows us to show how the generic method scales for varying values for n and l. Each process can be in one of the local states $L^1, \ldots, L^l$, where L^{l-1} and L^l take the rôles of the trying region and critical section, respectively. The process must go through L^1 to L^{l-1} in this order before proceeding into L^l. In addition, the transition into L^l is protected by a binary semaphore, which is released again upon the process' return to L^1:

Transition		Guard	Action
$L^i \to L^{i+1}$	for $1 \leq i \leq l-2$	*true*	*no-op*
$L^{l-1} \to L^l$		$!sem$	$sem := 1$
$L^l \to L^1$		*true*	$sem := 0$

As a second example, we chose a variant of the MCS list-based queuing lock with atomic compare_and_swap instruction [MCS91, also used in ID96]. The algorithm consists of an *acquire* and a *release* operation for a lock with the property that a process waiting for the lock spins only on process-local variables, instead of spinning on a shared variable (like a semaphore). According to [MCS91], spins on shared variables can cause memory detention and severe system performance degradation.

For the second example, the input was not a synchronization skeleton, but the program text for the two operations. In order to perform counter abstraction on this symmetric system, the number of local states needs to be determined. The *acquire* operation forces processes to line up for the lock in a queue. Each process remembers its successor, which can be any of the $n-1$ other processes, such that the number of local states of a process is not constant. While forming a queue is a valuable property for enforcing a special type of liveness on the processes, it is less relevant for the verification of safety properties. We therefore generalized the system so as to allow any process that is "ready" to obtain the lock to do so. Since the safety property—no two processes can acquire the lock at the same time—turned out to be true for this conservative abstraction with a constant number of 28 local states, we conclude that it holds in a system that enforces FIFO order.

For both problems, we experimented with unique, multiple and generic representatives. For multiple ones, we chose the set Rep as follows:

$$r \in Rep \;\Leftrightarrow\; \exists i : 1 \leq i \leq l : \text{process 1 is in location } L^i \;\wedge \\ \text{locations } L^j \text{ with } j < i \text{ do not appear in } r.$$

For example, using $l = 3$, the states (L^1, L^2, L^1), (L^1, L^3, L^1), (L^1, L^3, L^2) and (L^2, L^3, L^2) belong to the set Rep, but are not unique representatives, in which the superscripts have to be in order. It turns out that the BDD for the representative relation ξ derived from Rep can be computed much more efficiently than that for the *function* ξ for unique representatives. Looking back at definition 3, the complete set C to be chosen contains the n permutations that swap index 1 with index i, for $1 \leq i \leq n$. It can be shown that C indeed satisfies the two properties required in definition 3. C is exponentially smaller than the full symmetry group.

For the first example, we verified the standard safety property: $\mathsf{AG}\,\forall i, j : i \neq j : \neg(L_i^l \wedge L_j^l)$ (generically: $\mathsf{AG}\, n_l < 2$). For the second example, we verified that no two processes can acquire the lock at the same time, and also that there is no deadlock in the system. The latter means that it is *never* the case that *all* processes are simultaneously spinning in one of the two busy-waits that are present in the operations. Such a situation would cause a deadlock since a process can not free itself from a busy-wait, but can only be unlocked by another processes.

These properties were verified using the CUDD BDD package [S01] for the standard symbolic fixpoint characterization of $\mathsf{EF}\, bad$. Table 2 shows how the space requirements and running times of the three methods of symmetry reduction compare.

Table 2. Space and run time comparisons (i686/1400 Mhz PC, 256MB memory)

	Choice of n, l		Unique Specific Representatives		Multiple Specific Representatives		(Unique) Generic Representatives	
	n	l	no. of live BDD nodes	time in sec. (% orbit rel.)	no. of live BDD nodes	time in sec.	no. of live BDD nodes	time in sec.
M	8	4	114,894	8.2 (97%)	2,211	0.0	703	0.0
U	6	5	2,152,710	137.3 (97%)	6,612	0.1	690	0.0
T	16	16	—	>15h (100%)	132,377	6.6	4,876	0.0
E	64	16	—	—	599,561	198.8	6,972	0.1
X	128	128	—	—	—	>15h	69,060	10.4
	256	128	—	—	—	—	78,060	12.6
M	3	28	113,188	2.4 (79%)	30,614	0.2	1,340	0.0
C	4	28	9,478,195	4386.7 (95%)	75,604	0.5	2,608	0.0
S	8	28	—	>15h (100%)	272,080	15.4	7,320	0.3
-	16	28	—	—	2,417,477	5055.3	24,094	2.7
L	20	28	—	—	—	>15h	34,170	5.0
K	60	28	—	—	—	—	293,981	266.8

Discussion. First, for multiple and generic representatives, it can be seen that there is still room to grow memory-wise, but not necessarily so for unique representatives. Indeed, the main motivation for research on alternatives to unique representatives was the impractical BDD size of the orbit relation.

Further, the unique representatives approach spends nearly all of its time on the orbit relation construction. The use of multiple representatives clearly reduces memory and time requirements. The generic representatives solution outperforms, by several orders of magnitude, the other two both in terms of memory and time, and hence in the size of problems it can handle. According to the table, although multiple representatives do remedy the major disadvantage of an orbit relation based solution somewhat, generic representatives have in turn an equally impressive benefit over multiple ones.

7 Conclusion

In this paper, we investigated the use of generic representatives in symbolic model checking of fully symmetric systems. Compared to unique representatives, with generic ones there is no need to construct the orbit relation. Compared to multiple representatives, the generic ones preserve full symmetry reduction. The BDD derived from the generic structure $\hat{M}$ turned out to be small for the examples we experimented with. For the class of programs presented here, the translation into generic representatives can be done automatically and in negligible time.

Generic representatives seem to prove useful outside the symbolic domain as well: we translated some of the fully symmetric example programs coming with the Murφ explicit state verifier [DDHY92] into generic representatives. For some examples, we obtained savings in terms of both time and space of several orders of magnitude over Murφ's symmetry reduction algorithms (using unique or multiple representatives).

Related and Future Work. Barner and Grumberg [BG02] considered combining symmetry and symbolic representation using BDDs mainly for falsification. They perform reachability analysis by discarding states symmetric to previously seen states. However, due to orbit complexity problems, the algorithm uses multiple representatives and therefore forgoes some of the symmetry reduction possible. Also, according to [BG02], computation costs often incur the use of under-approximations of the set of reached representatives, which renders the algorithm inexact.

Finite counters have been used previously to abstractly represent states of systems with many processes. Pnueli, Xu and Zuck [PXZ02] used truncated counters with values 0, 1, or 2 to approximate the number of processes in certain locations in reasoning about symmetric parameterized systems. Emerson and Trefler [ET99] used counters in the form of generic representatives in connection with fully symmetric programs. Other examples can be found in the work by Emerson and Srinivasan [ES90] on synthesis of parameterized programs and in the work by Pong and Dubois [PD95] on cache protocol verification.

Several years ago, Ip and Dill [ID96] introduced *scalar sets* in the description of the input program to enforce full symmetry. The Murφ verifier is an explicit-state implementation of this approach. Since Murφ was originally not designed to exclusively target symmetric systems, Murφ's input language is more general. In addition to non-symmetric programs, it allows one to write programs exhibiting symmetry other than process symmetry, which is discussed in this paper. To make our approach more readily applicable, we would like to allow a more convenient input language than synchronization skeletons, perhaps similar to that of Murφ.

The present formulation of generic representatives is directly only applicable to (the common case of) fully symmetric systems. We would like to do research on systems whose symmetry group is the product of full symmetry groups of subsystems [CEJS98, section 5.1], and systems that are almost, but not fully, symmetric [ET99]. The ultimate goal is to apply the generic method to some larger, perhaps industrial-size examples.

References

[B86] Randy E. Bryant: "Graph-based Algorithms for Boolean Function Manipulation." *IEEE Transactions on Computers*, 35(8), 1986.

[BG02] Sharon Barner, Orna Grumberg: "Combining Symmetry Reduction and Under-Approximation for Symbolic Model Checking." *CAV*, LNCS 2402, 2002.

[CE81] Edmund M. Clarke and E. Allen Emerson: "The Design and Synthesis of Synchronization Skeletons Using Temporal Logic." *Workshop on Logics of Programs*, LNCS 131, 1981.

[CEFJ96] Edmund M. Clarke, Reinhard Enders, Thomas Filkorn, Somesh Jha: "Exploiting Symmetry In Temporal Logic Model Checking." *Formal Methods in System Design*, volume 9, 1996.

[CEJS98] Edmund M. Clarke, E. Allen Emerson, Somesh Jha, A. Prasad Sistla: "Symmetry Reductions in Model Checking." *CAV*, LNCS 1427, 1998.

[DDHY92] David L. Dill, Andreas J. Drexler, Alan J. Hu, C. Han Yang: "Protocol Verification as a Hardware Design Aid." *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 1992.

[EJP97] E. Allen Emerson, Somesh Jha, Doron Peled: "Combining Partial Order and Symmetry Reductions." *TACAS*, 1997.

[ES90] E. Allen Emerson and Jai Srinivasan: *A Decidable Temporal Logic to Reason About Many Processes.* "Principles of Distributed Computing", 1990.

[ES96] E. Allen Emerson, A. Prasad Sistla: "Symmetry and Model Checking." *Formal Methods in System Design*, vol. 9, ISSN 0925-9856, 1996.

[ET99] E. Allen Emerson, Richard J. Trefler: "From Asymmetry to Full Symmetry: New Techniques for Symmetry Reduction in Model Checking." *CHARME*, LNCS 1703, 1999.

[ID96] C. Norris Ip, David L. Dill: "Better Verification Through Symmetry." *Formal Methods in System Design*, volume 9, nos. 1/2, 1996.

[LN91] Bill Lin, A. Richard Newton: "Efficient symbolic manipulation of equivalence relations and classes." *International Workshop on Formal Methods in VLSI Design*, 1991.

[MCS91] John M. Mellor-Crummey, Michael L. Scott: "Algorithms for Scalable Synchronization on Shared-memory Multiprocessors." *ACM Transactions on Computer Systems*, vol. 9, 1991.

[M93] Ken L. McMillan: "Symbolic Model Checking: An Approach to the State Explosion Problem." *Kluwer Academic*, 1993.

[P90] Carl Pixley: "Introduction to a Computational Theory and Implementation of Sequential Hardware Equivalence." *CAV*, 1990.

[PD95] Fong Pong, Michel Dubois: "A New Approach for the Verification of Cache Coherence Protocols." *IEEE Transactions on Parallel and Distributed Systems*, 6(8), 1995.

[PXZ02] Amir Pnueli, Jessie Xu, Leonore Zuck: "Liveness with $(0, 1, \infty)$-Counter Abstraction." *CAV*, LNCS 2404, 2002.

[S01] Fabio Somenzi: "The CU Decision Diagram Package", release 2.3.1. `http://vlsi.colorado.edu/~fabio/CUDD/`

[W86] Pierre Wolper: "Expressing Interesting Properties of Programs." *13th Annual ACM Symposium on Principles of Programming Languages* (POPL), 1986.