Combining Symmetry Reduction and Under-Approximation for Symbolic Model Checking

Sharon Barner[1,2] and Orna Grumberg[1]

[1] Computer Science Department, Technion
Haifa 32000, Israel
{skeidar,orna}@cs.technion.ac.il
[2] IBM Haifa Research Lab
Haifa, Israel

Abstract. This work presents a collection of methods, integrating *symmetry reduction*, *under-approximation*, and *symbolic model checking* in order to reduce space and time for model checking. The main goal of this work is *falsification*. However, under certain conditions our methods provide *verification* as well.

We first present algorithms that perform on-the-fly model checking for temporal safety properties, using symmetry reduction. We then extend these algorithms for checking liveness properties as well.

Our methods are fully automatic. The user should supply some basic information about the symmetry in the verified system. However, the methods are *robust* and work correctly even if the information supplied by the user is incorrect. Moreover, the methods return correct results even in case the computation of the symmetry reduction has not been completed due to memory or time explosion.

We implemented our methods within IBM's model checker RuleBase, and compared the performance of our methods with that of RuleBase. In most cases, our algorithms outperformed RuleBase with respect to both time and space.

1 Introduction

This work presents a collection of methods, integrating *symmetry reduction*, *under-approximation*, and *symbolic model checking* in order to reduce space and time for model checking. The main goal of this work is *falsification*, that is, proving that a given system does not satisfy its specification. However, under certain conditions our methods provide also *verification*, i.e., they prove that the system satisfies its specification.

Our methods are fully automatic. The user should supply some basic information about the symmetry in the verified system. However, the methods are *robust* and work correctly even if the information supplied by the user is incorrect. Moreover, the methods return correct results even in case the computation of the symmetry reduction has not been completed due to memory or time explosion.

D. Brinksma and K. G. Larsen (Eds.): CAV 2002, LNCS 2404, pp. 93–106, 2002.

Temporal logic model checking [6] is a technique that accepts a finite state model of a system and a temporal logic specification and determines whether the system satisfies the specification. The main problem of model checking is its high memory requirements. *Symbolic model checking* [15], based on BDDs [4], can handle larger systems, but is still limited in its capacity. Thus, additional work is needed in order to make model checking feasible for larger systems.

This work exploits symmetry reduction in order to reduce memory and time used in symbolic model checking. Symmetry reduction is based on the observation that many systems consist of several similar components. Exchanging the role of such components in the system does not change the system's behavior. Thus, system states can be partitioned into equivalence classes called *orbits*, and the system can be verified by examining only representatives from each orbit.

Two main problems arise, however, when combining symbolic model checking with symmetry reduction. One is building the orbit relation and the other is choosing a representative for each orbit. [13] proves that the BDD for the orbit relation is exponential in the number of the BDD variables, and suggests choosing more than one representative for each orbit in order to obtain a smaller BDD for the orbit relation. Yet, this method does not solve the problem of choosing the representatives. The choice of representatives is significant since it strongly influences the size of the BDDs representing the symmetry-reduced model. [11] suggests to choose generic representatives. This approach involves compiling the symmetric program to a reduced model over the generic states. Such a compilation can only be applied to programs written with a special syntax in which symmetry is defined inside the program. [12] introduces an algorithm for explicit model checking which chooses as a representative for an orbit the first state from this orbit, discovered by the DFS. This method avoids choosing the representatives in advance. Unfortunately, it is not applicable to symbolic model checking since performing DFS is very inefficient with BDDs.

We suggest a new approach that avoids building the orbit relation and chooses representatives on-the-fly while computing the reachable states. Unlike [12] the choice of the representatives is guided by BDD criteria. Reachability is performed using an *under-approximation* that, at each step, explores only a subset of the reachable states. Some of the unexplored states are symmetric to the explored ones. By exploiting symmetry information, those states will never be explored. Thus, easier symbolic forward steps are obtained.

We first apply this approach for verifying properties of the form $AG(p)$[1], where p is a boolean formula. If we find a "bad" state that does not satisfy p we conclude that the checked system does not satisfy $AG(p)$. On the other hand, if no "bad" state is found we cannot conclude that the system satisfies $AG(p)$ since reachability with under-approximation does not necessarily explore every reachable state. We next present a special version of the previous algorithm in which the under-approximation is guided by *hints* [3]. Under certain conditions this algorithm can also verify the system.

[1] $AG(p)$ means that p holds along every path, in every state on the path.

The algorithms described above are based on reachability, and are often referred to as *on-the-fly* model checking. It is well known how to extend on-the-fly model checking for $AG(p)$ to verifying general *safety temporal properties*. This is done by building an automaton describing the property and running it together with the system. We specify conditions on the automaton that guarantee the correctness of the on-the-fly algorithm also when the automaton runs together with the symmetry-reduced model. The suggested conditions hold for the tableau construction used for symbolic LTL model checking [5], when restricted to LTL safety properties. They also hold for the satellite used in symbolic model checking of RCTL formulas [2]. By running the automaton together with the reduced model we save both space and time while verifying these types of formulas.

On-the-fly symbolic model checking cannot handle liveness properties. In order to handle such properties we developed two extensions combining symmetry reduction with classical (not on-the-fly) symbolic model checking. One is easy to perform and is mainly suitable for falsification. The other is more expensive but can handle verification as well.

Previous works expect the user to provide a symmetry group that is also an invariance group [13]. In many cases two formulas checked on the same model require different invariance groups since each formula breaks differently the symmetry of the model. Thus, the user needs to supply different invariance groups for different formulas. In other works [17,8] the program is written in a special syntax, which enables finding the invariance group according to this syntax. In these cases only formulas which do not break the symmetry of the model are allowed. In contrast, we build the invariance group automatically, once the symmetry group is given. Supplying the symmetry group usually requires only a high level understanding of the system and therefore is easier than supplying the invariance group.

We implemented our methods within the enhanced model checking tool RuleBase [1], developed by the IBM Haifa Research Laboratories, and compared the performance of our methods with that of RuleBase. Our experiments show that our methods performed significantly better, with respect to both time and space, in checking liveness properties. For temporal safety properties they achieved better time requirements. However, their space requirements were worse for small examples and identical for larger ones.

The rest of the paper is organized as follows. Section 2 gives some basic definitions. Section 3 shows how to build the invariance group. Section 4 presents an algorithm for on-the-fly symbolic model checking with symmetry reduction and then introduces hints into this algorithm. Section 5 and 6 handle temporal safety properties and liveness properties, respectively and Section 7 presents our experimental results.

2 Preliminaries

Let AP be a set of atomic propositions. We model a system by a Kripke structure M over AP, $M = (S, S_0, R, L)$ where S is a finite set of states, S_0 is a set of initial states, $R \subseteq S \times S$ is a total transition relation, and $L : S \rightarrow 2^{AP}$ is a labeling function which labels each state with the set of atomic propositions true in that state.

As the specification language we use the branching time temporal logic CTL, defined over AP. The semantics of CTL is defined with respect to a Kripke structure. We write $M \models \varphi$ to denote that the formula φ is true in M. For a formal definition of CTL and its semantics see [7]. ACTL is the sub-logic of CTL in negation normal form in which all formulas contain only universal path quantifiers.

The *bisimulation equivalence* and *simulation preorder* are relations over Kripke structures (see [7] for definitions) that have useful logical characterizations. We write $M \equiv_{bis} M'$ to denote that M and M' are bisimulation equivalent and $M \leq_{sim} M'$ to denote that M is smaller than M' by the simulation preorder. The following lemmas relate bisimulation and simulation with logics.

Lemma 1. *[7] For every two Kripke structures M, M' over AP,*

- *if $M \equiv_{bis} M'$ then $\forall \varphi \in CTL$ over AP, $M' \models \varphi \Leftrightarrow M \models \varphi$.*
- *if $M \leq_{sim} M'$ then $\forall \varphi \in ACTL$ over AP, $M' \models \varphi \Rightarrow M \models \varphi$.*

BDDs: A Binary Decision Diagram (BDD) [4] is a data structure for representing boolean functions. BDDs are defined over boolean variables, they are often (but not always) concise in their memory requirement, and most boolean operations can be performed efficiently on BDD representations. In [15] it has been shown that BDDs can be very useful for representing Kripke structures and performing model checking symbolically. One of the most useful operations in model checking, and in particular on-the-fly model checking, is the *image computation*. Given a set of states S and a binary relation T, represented by the BDDs $S(\bar{v})$ and $T(\bar{v}, \bar{v}')$ respectively, the image computation finds the set of all states related by T to some state in S. More precisely, $Im_T(S(\bar{v})) = \exists \bar{v}(S(\bar{v}) \wedge T(\bar{v}, \bar{v}'))$.

Partial Search: While symbolic model checking can be very efficient, it might still suffer from explosion in the BDD size. One of the solutions is to perform partial search of the reachable state space while avoiding large BDDs [16]. Other methods perform partial search which is guided by the user [3] or by the checked specification [18]. In all these methods the set of reachable states discovered in each step is an under approximation of the set of reachable states which would have been discovered by a BFS. This property enables combining partial search with on-the-fly model checking.

Symmetry: A permutation on a set A, $\sigma : A \rightarrow A$ is a one-to-one and onto function. For a set $A' \subseteq A$, $\sigma(A') = \{a | \exists a' \in A' \ \sigma(a') = a\}$. In this paper we use permutations over the set of states of a Kripke structure. Given a CTL formula β and a structure M, $\sigma(\beta)$ refers to applying σ to the set of states in M that satisfy β.

A *permutation group* G is a set of permutations together with the composition operation such that the identity permutation e is in G and G is closed under the inverse and the composition operations. If there exists $\sigma \in G$ such that $\sigma(s) = s'$ we say that the two states s,s' are symmetric.

Definition 1. *$\sigma_1, \sigma_2, \ldots, \sigma_k$ are* generators *of a permutation group G (denoted $G = \langle\sigma_1, \sigma_2, \ldots, \sigma_k\rangle$) if G is the closure of the set $\{\sigma_1, \sigma_2, \ldots, \sigma_k\}$ under composition operation.*

Definition 2. *A permutation group G is a* symmetry group *of a Kripke structure M if every permutation $\sigma \in G$ preserves the transition relation. That is, $\forall s, s' \in S\ [(s, s') \in R \Leftrightarrow (\sigma(s), \sigma(s')) \in R]$.*

Definition 3. *A symmetry group G of a Kripke structure M is an* invariance group *for formula φ if for every atomic proposition β of φ, every $\sigma \in G$ and $s \in S\ \ [M, s \models \beta \Leftrightarrow M, \sigma(s) \models \beta]$.*

Given an invariance group G and a Kripke structure M we can partition S into equivalence classes. The equivalence class of s is $[s] = \{s' | \exists\sigma \in G,\ \sigma(s) = s'\}$. Each [s] is called an *orbit* and the relation OR $= \{(s, s') | s, s' \in S \textit{ and } [s] = [s']\}$ is called the *orbit relation*

For a Kripke structure $M = (S, S_0, R, L)$ and an invariance group G for φ the *quotient structure* is $M_G = (S_G, S_G^0, R_G, L_G)$ where $S_G = \{\ [s] \mid s \in S\ \}$, $S_G^0 = \{\ [s] \mid s \in S_0\ \}$, $R_G = \{\ ([s], [s']) \mid (s, s') \in R\ \}$ and $L_G([s]) = L(s)$. In [10,13] it has been proved that $M_G \equiv_{bis} M$. By Lemma 1 we therefore have that for every CTL formula ψ over the same AP as φ, $M_G \models \psi \Leftrightarrow M \models \psi$.

In order to build the quotient structure a representative should be chosen from each orbit In many cases, however, it is easier to choose more than one representative for each orbit. We then define a *representative relation* $\xi \subseteq Rep \times S$ which satisfies $(s, s') \in \xi \Leftrightarrow s \in Rep \wedge [s] = [s']$. In this case we define the structure $M_m = (S_m, S_m^0, R_m, L_m)$ (m for multiple representatives) where $S_m = Rep$, $S_m^0 = \{\ s \mid \exists s' \in S_0\ (s, s') \in \xi\}$, $R_m = \xi^{-1} R \xi$ and $L_m = L$. [13] shows that $M_m \equiv_{bis} M_G \equiv_{bis} M$.

3 Building the Invariance Group

In this section we show how to automatically compute the generators of an invariance group given the generators of a symmetry group.

Our method works as follows. Given a set of generators for a symmetry group G, an invariance group G_{inv} is defined by restricting the generators of G to those σ_i that satisfy $\sigma_i(\beta) = \beta$ for every $\beta \in AP$. The following lemma states the correctness of our approach.

Lemma 2. *Let $\sigma_1, \sigma_2, \ldots, \sigma_k$ be generators of a symmetry group G of a Kripke structure M and let φ be a formula over AP. Then $IG = \{\sigma_i | \forall\beta \in AP,\ \sigma_i(\beta) = \beta\}$ generates an invariance group G_{inv} of M for φ.*

4 Symmetry with On-the-Fly Representatives

The symbolic algorithm **Symmetry_MC** presented in this section is aimed at avoiding the two main problems of symmetry reduction, namely building the orbit relation and choosing a representative for each orbit.

Let $M = (S, S_0, R, L)$ be a Kripke structure and $\sigma_1, \ldots, \sigma_k$ be a set of generators of a symmetry group G of M. Also let $\varphi = AG(p)$ where p is a boolean formula. The algorithm **Symmetry_MC**, presented in Figure 1, applies on-the-fly model checking for M and φ, using under-approximation and symmetry reduction.

The algorithm works in iterations. Starting with the set of initial states, at each iteration a subset `under` of the current set of states is chosen. The successors of `under` are computed. However, every state which is symmetric to (i.e., in the same orbit with) a previously reached state is removed. The states that are first found for each orbit are taken to be the orbit representatives. Note that an orbit may have more than one representative if several of its states are reached when the orbit is encountered for the first time. At any step, the set of representatives are checked to see if they include a state that violates p. If such a state is found (line 9) then the computation stops and a counterexample is produced. We then conclude that $M \not\models AG(p)$. A useful optimization can be performed by deleting from memory the BDD for the set `full_reach` immediately after it is used (after line 7). This may avoid memory explosion when computing forward steps.

The set of symmetric states that should be removed are computed using the procedure **σ_Step** (Figure 2) instead of using the orbit relation. For a set of states A and a set of generators $IG = \{\ \sigma_1, \ldots, \sigma_k\ \}$, **$\sigma$_Step** returns the set of all states belonging to the orbits of states in A according to $G = \langle IG \rangle$. By using **σ_Step** we exploit symmetry information without building the orbit

Symmetry_MC(M, φ, $\sigma_1, \ldots \sigma_k$)
1 Calculate the generators of the invariance group of M
IG = $\{\sigma_i |$ for each atomic sub-formula β of φ: $\sigma_i(\beta) = \beta\}$
2 `reach_rep` $= S_0$, i=0
3 while $S_i \neq \emptyset$
4 choose `under` $\subseteq S_i$ (`under` is an under-approximation of S_i)
5 $S_{i+1} = Im_R($`under`$)$
6 `full_reach` = **σ_Step**(`reach_rep`, $\sigma_1, \ldots \sigma_k$)
7 $S_{i+1} = S_{i+1}$ / `full_reach`
8 `reach_rep` = `reach_rep` $\cup S_{i+1}$
9 if $S_{i+1} \wedge \neg p \neq \emptyset$
10 generate a counter example and break.
11 i = i+1.

Fig. 1. The algorithm **Symmetry_MC** performs on-the-fly model checking of φ on M, using symmetry reduction

```
σ_Step(A, σ1, σ2, ..., σk)
    sym_states = A;
    old_sym_states = ∅
    while old_sym_states ≠ sym_states
        old_sym_states = sym_states
        for i = 1...k
            new_sym_states = Im_σi(sym_states)
            sym_states = sym_states ∪ new_sym_states
    return sym_states
```

Fig. 2. σ**_Step** calculates the states belonging to the orbits of states in A. In order to calculate Im_{σ_i}(sym_states), σ_i can be viewed as the binary relation $\bar{v} = \sigma(\bar{v}')$

relation. σ**_Step** is expected to be smaller since it represents a set of states and not a relation. Furthermore it is applied only to reachable states which are usually represented by smaller BDDs. Indeed, our experiments successfully applied σ**_Step** to designs for which building the orbit relation was infeasible.

Computationally, σ**_Step** is quite heavy. To avoid this problem, in most of our experiments we stopped the computation of σ**_Step** before it got to a fixed point. In general there is a tradeoff between the amount of computation in σ**_Step** and the symmetry reduction obtained by **Symmetry_MC**.

4.1 Robustness of Symmetry_MC

We now discuss the robustness of the algorithm **Symmetry_MC** for falsification in the presence of an incomplete σ**_Step** and an incorrect set of generators. Consider first the case in which the computation of procedure σ**_Step** is stopped before a fixed point is reached. σ**_Step** then returns only a subset of the states in the orbits of states in A. In this case, less states are removed from S_{i+1} and as a result `reach_rep` contains more states. Thus, we might have more representatives for each orbit.

Consider now the case in which the algorithm is given an incorrect set of generators. If a "bad" generator (a permutation which associates states that are not symmetric) is given, then σ**_Step** returns states which are not symmetric to any state in `reach_rep`. These states are removed from S_{i+1} and we might not add any representatives of their orbits to `reach_rep`. Thus, `reach_rep` represents an under-approximation of the reachable orbits. Consequently, if there is a state $s \in$ `reach_rep` which does not satisfy p, this state is reachable in the original model and the counterexample generated by **Symmetry_MC** actually exists in the original model. If a "good" generator is missing, then σ**_Step** returns less states and as a result there is more than one representative for each orbit. However, like in the previous case, `reach_rep` contains only reachable states and therefore **Symmetry_MC** generates only real counterexamples. To summarize,

Lemma 3. *Given any set of generators, the algorithm* **Symmetry_MC** *is sound for falsification.*

At termination of **Symmetry_MC**, if `reach_rep` contains at least one state from each reachable orbit then M_m, defined according to `reach_rep` ($S_m =$ `reach_rep`) is bisimilar to M (see Section 2). Thus, if $M_m \models AG(p)$ then $M \models AG(p)$ as well. Note that $M_m \models AG(p)$ can be checked on-the-fly by **Symmetry_MC**.

4.2 Symmetry Reduction Combined with Hints

In this section we present a special case of the algorithm **Symmetry_MC** in which the under-approximation is guided by a sequence of hints given by the user [3]. The algorithm **Hints_Sym**, presented in Figure 3, gets as parameters also a sequence $h_1, \ldots, h_l$ of hints such that $h_l =$ TRUE.

If $\sigma_1, \ldots, \sigma_k$ contain no "bad" generator[2] then our hints guarantee that when $S_i = \emptyset$, `reach_rep` contains at least one state from each reachable orbit. In this case, the algorithm **Hints_Sym** is suitable for verification as well as falsification.

Hints_Sym(M, φ, $\sigma_1, \ldots, \sigma_k$, $h_1, \ldots, h_l$)
  Calculate IG = $\{\sigma_i |$ for each atomic sub-formula β of φ: $\sigma_i(\beta) = \beta\}$
  `reach_rep` = S_0, i = 0, `hint` = h_1, j = 2
  while $S_i \neq \emptyset$
    `under` = $S_i \cap$ `hint`
    $S_{i+1} = Im_R($`under`$)$
    `full_reach` = σ**_Step**(`reach_rep`, $\sigma_1, \ldots \sigma_k$)
    $S_{i+1} = S_{i+1}/$`full_reach`
    `reach_rep` = `reach_rep` $\cup S_{i+1}$
    if $S_{i+1} \wedge \neg p \neq \emptyset$
      generate counter example and break
    if $S_{i+1} = \emptyset \wedge j \leq l$
      `hint` = $hint_j$
      j = j+1
      S_{i+1} = `reach_rep`
    i = i+1
  φ is TRUE

Fig. 3. The algorithm **Hints_Sym** applies on-the-fly model checking of φ on M, using hints and symmetry reduction

[2] In many cases, the nonexistence of bad generators can be easily determined by the program syntax [17,8]. In other cases it is expensive but possible to check whether all generators are good.

5 Extension for Temporal Safety Properties

There are several known algorithms which use a construction A_φ for the evaluated formula φ and the product model $M \times A_\varphi$ in order to apply model checking more efficiently. We now show that it is possible to combine symmetry reduction with these algorithms. We first specify the requirements on the construction A_φ so that it can be used with symmetry reduction.

Definition 4. *Given a logic $\mathcal{L}$ and a construction that associates with each $\varphi \in \mathcal{L}$ a structure A_φ, the construction A_φ is safe for symmetry reduction w.r.t. $\mathcal{L}$ if it satisfies the following conditions:*

1. $\exists\psi\forall\varphi \in \mathcal{L}\ (M \models \varphi \Leftrightarrow M \times A_\varphi \models \psi)$.
2. *For every invariance group G_{inv} of M for φ, every $\sigma \in G_{inv}$ and every $(s,t) \in S_{M\times A_\varphi}$, $\sigma((s,t)) = (\sigma(s), t)$* [3].
3. *For every atomic proposition β of ψ and every $(s,t),(s',t) \in S_{M\times A_\varphi},\ (s,t) \models \beta \Leftrightarrow (s',t) \models \beta$.*

The second condition requires that σ is defined only on s and leaves t unchanged. The third condition requires that the truth of all β in ψ depend only on t.

Lemma 4. *For every construction A_φ which is safe for symmetry reduction w.r.t. $\mathcal{L}$, if G is an invariance group of structure M for formula $\varphi \in \mathcal{L}$ then G is an invariance group of structure $M \times A_\varphi$ for formula ψ.*

Corollary 1. *For every construction A_φ which is safe for symmetry reduction w.r.t. $\mathcal{L}$ and for every $\varphi \in \mathcal{L}$ and $\psi \in CTL$, the quotient structure $(M \times A_\varphi)_G$, built for $M \times A_\varphi$ and an invariance group G of M, satisfies $(M \times A_\varphi)_G \models \psi \Leftrightarrow M \models \varphi$.*

Note that using safe construction enables us to find the generators of the invariance group of M according to φ and then to evaluate formula ψ on $M \times A_\varphi$ with symmetry reduction that use the same generators. There are several A_φ constructions which are safe for symmetry reduction w.r.t. logic $\mathcal{L}$. One example is the tableau construction in [5] when restricted to LTL safety properties. In this case the tableau includes no fairness constraints and it fulfills the requirements of Definition 4. Another safe construction is the satellite for RCTL formulas defined in [2]. By combining safe construction with symmetry reduction we make symmetry reduction applicable together with a new set of algorithms, like symbolic on-the fly model checking for RCTL and symbolic LTL model checking, for which it was not applicable until now. We implemented our algorithms using the construction introduced in [2], which enabled us to check RCTL formulas on-the-fly while using a symmetry reduction.

[3] since s and $\sigma(s)$ agree on AP, $[(s,t) \in S_{M\times A_\varphi} \Leftrightarrow (\sigma(s),t) \in S_{M\times A_\varphi}]$.

6 Extensions for Liveness Formulas

We now describe two possible extensions that combine classical (not on-the-fly) symbolic model checking with symmetry reduction. These extensions are useful for checking liveness properties, and other properties which cannot be checked on-the-fly.

6.1 Liveness Restricted to Representatives

The purpose of this extension is to falsify ACTL formulas with respect to a structure M, while avoiding the construction of its quotient model M_G. The idea is to get a set of representatives *Rep* and to construct the restricted model $M|_{Rep}$. The restricted model $M|_A = (S|_A, S_0|_A, R|_A, L|_A)$ is a Kripke structure where $S|_A$ = A, $S_0|_A = S_0 \cap A$, $\forall s, s' \in S|_A \; [(s, s') \in R|_A \Leftrightarrow (s, s') \in R]$ and $\forall s \in S|_A \; [L|_A(s) = L(s)]$. Since $M|_{Rep} \leq_{sim} M$, we have that for every ACTL formula φ, if $M|_{Rep} \not\models \varphi$ then $M \not\models \varphi$. Thus, φ can be checked on the smaller model $M|_{Rep}$.

Note that in principle this idea works correctly with any set of representatives, even such that does not include a representative for each orbit. There are however advantages to choosing as *Rep* the set `reach_rep` which results from the algorithm **Symmetry_MC**. First, `reach_rep` includes only reachable states. Second, by construction, the states in `reach_rep` are connected by transitions while an arbitrary set of representatives *Rep* might not be connected, thus, $M|_{reach\_rep}$ often includes more behaviors than $M|_{Rep}$. Third, the states in `reach_rep` represent many other states in the system, thus if the system includes a bad behavior, it is more likely that `reach_rep` will reflect it.

Following the discussion above we suggest the Algorithm **Live_Rep** that works as follows: it first runs **Symmetry_MC** to obtain `reach_rep` and then performs classical symbolic model checking on $M|_{reach\_rep}$.

6.2 Liveness with the Representative Relation

We now present another possibility for handling liveness properties. It is applicable only if no bad generators exist. This method is more expensive computationally, but is suitable for verification of liveness properties. Similarly to the previous section we first compute `reach_rep` using the algorithm **Symmetry_MC**. However, now we apply the procedure **Create_ξ**, presented below, in order to compute the representative relation $\xi \subseteq$ `reach_rep` $\times S$ (see definition in Section 2). Next we construct a new structure $M' = (S', S'_0, R', L')$ where S' = `reach_rep`, $S'_0 = \{ \; s \mid \exists s' \in S_0 \; (s, s') \in \xi$, $R' = \xi^{-1} R \xi$ and $L' = L$. Finally, we run classical symbolic model checking on φ and M'.

Lemma 5. *If S' contains at least one representative for each reachable orbit then $M \equiv_{bis} M'$. Otherwise, $M' \leq_{sim} M$.*

Create_ξ $(\sigma_1, \sigma_2 \ldots \sigma_k, \text{Rep})$

1 $\xi(\bar{v}, \bar{v}') = \text{Rep}(\bar{v}) \wedge (\bar{v} = \bar{v}')$
2 `old_`$\xi(\bar{v}, \bar{v}') = \phi$
3 while `old_`$\xi(\bar{v}, \bar{v}') \neq \xi(\bar{v}, \bar{v}')$
4 `old_`$\xi(\bar{v}, \bar{v}') = \xi(\bar{v}, \bar{v}')$
5 for i = 1 . . . k
6 `new`$(\bar{v}, \bar{v}'') = \exists \bar{v}'(\xi(\bar{v}, \bar{v}') \wedge \sigma_i(\bar{v}'', \bar{v}'))$
7 $\xi(\bar{v}, \bar{v}') = \xi(\bar{v}, \bar{v}') \cup$ `new`$(\bar{v}, \bar{v}')$
8 return $\xi(\bar{v}, \bar{v}')$

Fig. 4. The algorithm **Create_ξ** for computing $\xi \subseteq Rep \times S$. Line 6 is implemented with the operator `compose_odd` [14] which computes $\exists \bar{v}'(\xi(\bar{v}, \bar{v}') \wedge \sigma_i(\bar{v}'', \bar{v}'))$ using only two sets of BDD variables instead of three

If `reach_rep` is the result of the algorithm **Hints_Sym**, then `reach_rep` indeed contains at least one representative for each orbit, and M' is bisimilar to M. Thus, M' can be used for verifying full CTL.

Figure 4 presents the BDD-based procedure **Create_ξ** for building the representative relation ξ for a given set of representatives *Rep* and a set of generators $\sigma_1, \ldots, \sigma_k$ of an invariance group G of M for φ.

7 Experimental Results

We implemented the algorithms **Hints_Sym**, **Live_Rep**, and **Create_ξ** in the IBM's model checker RuleBase [1]. We ran it on a number of examples which contain symmetry. For each example we tuned our algorithms according to the evaluated formula, the difficulty level of computing the reachable states and the difficulty level of building the transition relation. In most cases, our algorithms outperformed RuleBase with respect to both time and space. In the tables below time is measured by seconds, memory (mem) in bytes, and the transition relation size (TR size) in number of BDD nodes.

The Futurebus Example: We ran the algorithm **Live_Rep** in order to check liveness properties on the Futurebus cache-coherence protocol with a single bus and a single cache line for each processor. The table in Figure 5 presents the results of evaluating the property for a different number of processors. For comparison we ran also the RuleBase classical symbolic model checking algorithm. Both algorithms applied dynamic BDD reordering. The BDD order is very important since the best BDD order for the classical algorithm is different from the best BDD order of our algorithm. In order to obtain a fair comparison between these algorithms we ran each algorithm twice. In the first run each algorithm reordered a BDD without time limit in order to find a BDD order which is good

# of processors	# vars	classic algorithm time	mem	TR size	Live_Rep time	mem	TR size
5	45	132	43M	144069	101	41M	122769
6	54	607	118M	260625	265	56M	219572
7	63	2852	277M	418701	704	76M	379428
8	72	8470	589M	839055	3313	101M	457781
9	81	81,171	709M	1935394	4571	106M	819871
10	90	-	> 1G	-	4909	120M	642083

Fig. 5. Live_Rep on Future bus example

for this algorithm. The initial order of the second run was the BDD order which was found by the first run.

The most difficult step in the Futurebus example is building the transition relation. By restricting the transition relation to the representatives which were chosen on-the-fly, the transition relation became smaller and as a result the evaluation became easier. Figure 5 shows that both time and space were reduced dramatically using **Live_Rep**. We can also observe that as the number of processes increases, the results improved. This is to be expected, as the increase in the number of the reachable representatives is smaller than the increase in the number of reachable states.

The Arbiter Example: We ran algorithm **Hints_Sym** on an arbiter example with n processes. We checked the arbiter w.r.t. RCTL formulas which were translated to safe A_φ and ψ. For comparison we ran RuleBase on-the-fly model checking and on-the-fly model checking with hints (without symmetry). All algorithms used dynamic BDD reordering and partitioned transition relation [9]. In this case we calculated σ**_Step** only when we changed hints and stopped σ**_Step** before the fixed point has been reached. The table in Figure 6 presents the results of the three algorithms on arbiter with 6,8 and 10 processes. For each case we checked one property that passed and one that failed. We notice that **Hints_Sym** reduced time but not necessarily space. This can be explained by the fact that σ**_Step** produced large intermediate BDDs but resulted in a significant reduction in S_i, thus reduced the computation time of the image steps.

# of processors	status	# vars	on-the-fly time	mem	on-the-fly + hints time	mem	Hints_Sym time	mem
6	passed	65	53	40M	39	40M	42	40M
6	failed	65	213	52M	64	41M	51	87M
8	passed	84	581	64M	255	49M	179	87M
8	failed	84	745	71M	524	71M	292	83M
10	passed	105	1470	94M	598	67M	358	92M
10	failed	105	1106	93M	740	73M	520	91M

Fig. 6. Hints_Sym compared to other on-the-fly algorithms

num of generators	num of vars	orbit_to_ξ		Create_ξ	
		time	mem	time	mem
3	16	0.26	26M	0.23	26M
4	20	30.4	33M	1.2	28M
5	24	1017	114M	18	42M
6	28	-	>1.5G	735	132M
6	32	-	>1.5G	29083	1.2G

Fig. 7. Create_ξ compared to **Orbit_To_**ξ

Comparing Create_ξ and Orbit_To_ξ: [13] presents an algorithm for computing ξ by building the orbit relation and then choosing the representatives. We refer to this algorithm by **Orbit_To_**ξ. We compare this algorithm with **Create_**ξ. Both algorithms find the representative relation $\xi \subseteq Rep \times S$ for the set of representatives Rep chosen according to the lexicographic order. The results in Figure 7 show that **Create_**ξ gave better results in both time and space, We believe that this is due to the fact that it saves less information while building ξ.

Acknowledgment

We thank Cindy Eisner for many helpful discussions. Somesh Jha is thanked for his help with the examples.

References

1. I. Beer, S. Ben-David, C. Eisner, and A. Landver. RuleBase: An industry-oriented formal verification tool. In *Design Automation Conference*, pages 655–660, June 1996. 95, 103
2. I. Beer, S. Ben-David, and A. Landver. On-the-fly model checking of RCTL formulas. In Alan J. Hu and Moshe Y. Vardi, editors, *Proceedings of the 10th International conference on Computer-Aided Verification*, volume 1427 of *LNCS*, pages 184–194. Springer-Verlag, June 1998. 95, 101
3. R. Bloem, K. Ravi, and F. Somenzi. Symbolic guided search for CTL model checking. In *Design Automation Conference*, pages 29–34, June 2000. 94, 96, 100
4. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE transactions on Computers*, C-35(8):677–691, 1986. 94, 96
5. E. Clarke, O. Grumberg, and H. Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design*, 10(1), 1997. 95, 101
6. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986. 94
7. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT press, December 1999. 96
8. C. N. Ip and D. L. Dill. Better verification through symmetry. In D. Agnew, L. Claesen, and R. Camposano, editors, *Computer Hardware Description Languages and their Applications*, pages 87–100, Ottawa, Canada, 1993. Elsevier Science Publishers B. V., Amsterdam, Netherland. 95, 100

9. D. Geist and I. Beer. Efficient model checking by automated ordering of transition relation. In David L. Dill, editor, *Proceedings of the sixth International Conference on Computer-Aided Verification CAV*, volume 818, pages 299–310. Springer-Verlag, June 1994. 104
10. E. A. Emerson and A. P. Sistla. Symmetry and model checking. In C. Courcoubetis, editor, *Proceedings of the 5th International conference on Computer-Aided Verification*, volume 697 of *LNCS*. Springer-Verlag, June 1993. 97
11. E. Allen Emerson and Richard 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. 94
12. V. Gyuris and A. P. Sistla. On-the-fly model checking under fairness that exploits symmetry. *Formal Methods in System Design: An International Journal*, 15(3):217–238, November 1999. 94
13. S. Jha. *Symmetry and Induction in Model Checking*. PhD thesis, CMU, 1996. 94, 95, 97, 105
14. S. Katz. Coverage of model checking. Master's thesis, Technion, haifa, Israel, 2001. 103
15. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993. 94, 96
16. K. Ravi and F. Somenzi. High-density reachability analysis. In *Proc. Intl. Conf. on Computer-Aided Design*, pages 154–158, November 1995. 96
17. A. P. Sistla, V. Gyuris, and E. A. Emerson. SMC: a symmetry-based model checker for verification of safety and liveness properties. *Software Engineering and Methodology*, 9(2):133–166, 2000. 95, 100
18. C. Han Yang and David L. Dill. Validation with guided search of the state space. In *Design Automation Conference*, pages 599–604, June 1998. 96