Variable Reuse for Efficient Image Computation

Zijiang Yang[1] and Rajeev Alur[2]

[1] Department of Computer Science
Western Michigan University, Kalamazoo, MI 49008
[2] Department of Computer and Information Science
University of Pennsylvania, Philadelphia, PA 19104

Abstract. *Image computation*, that is, computing the set of states reachable from a given set in one step, is a crucial component in typical tools for BDD-based symbolic reachability analysis. It has been shown that the size of the intermediate BDDs during image computation can be dramatically reduced via conjunctive partitioning of the transition relation and ordering the conjuncts for facilitating early quantification. In this paper, we propose to enhance the effectiveness of these techniques by reusing the quantified variables. Given an ordered set of conjuncts, if the last conjunct that uses a variable u appears before the first conjunct that uses another variable v, then v can be renamed to u, assuming u will be quantified immediately after its last use. In general, multiple variables can share the same identifier so the BDD nodes that are inactive but not garbage collected may be activated. We give a polynomial-time algorithm for generating the optimum number of variables that are required for image computation and show how to modify the image computation accounting for variable reuse. The savings for image computation are demonstrated on ISCAS'89 and Texas'97 benchmark models.

1 Introduction

In model checking, a finite-state model of a design is automatically verified with respect to temporal requirements to reveal inconsistencies [8, 10, 9]. The key step in a model checker is to compute the set of reachable states of a model. The reachable set is usually computed by iterative applications of image computation, where the image $Img(S)$ of a set S of states contains all states that can be reached from some state in S by executing one step of the model. In the past decade, a lot of research has centered around improving the image computation [4, 5, 11, 18, 17, 16, 6, 7]. In particular, for BDD-based symbolic model checkers, *conjunctive partitioning* and *early quantification* have significantly enhanced the applicability of model checkers. In this paper, we present a new technique, called *variable reuse*, that works synergetically with conjunctive partitioning and early quantification, leading to further savings in computational requirements of reachability analysis.

Consider a system M whose state can be described by n boolean variables $X = \{x_1, \ldots x_n\}$. A set S of states of M, then, can be viewed as a boolean

function of variables in X. If $X' = \{x'_1, \ldots x'_n\}$ denotes the set of *next-state* variables, then the dynamics of M is captured by the transition relation $T(X, Y, X')$, where $Y = \{y_1, \ldots y_l\}$ is the set of auxiliary variables such as the combinational variables and the primary inputs. The transition relation T is, thus, a boolean function over $m = 2n + l$ variables. The image computation can be defined as $Img(S) = [\exists X, Y.\, S(X) \wedge T(X, Y, X')][X' \rightarrow X]$, where $[X' \rightarrow X]$ denotes the renaming operation of replacing each next-state variable x'_i by the corresponding current-state variable x_i [4, 15]. A popular representation for the boolean functions for the purpose of symbolic reachability analysis is ordered binary decision diagrams [3], or its variants. For realistic designs, representing the transition relation T as a monolithic BDD is not possible. Consequently, symbolic analyzers such as COSPAN [14], SMV [15], and VIS [2], employ a partitioned representation of T as a set $\{C_1, \ldots C_k\}$ of transition relations such that T is the conjunction $C_1 \wedge \cdots \wedge C_k$. Instead of computing the conjunction T a priori, the conjunction $S \wedge C_1 \wedge \cdots \wedge C_k$, which constrains T by the current set S, is computed during each image computation [5, 15].

During image computation, all the variables in $X \cup Y$ are quantified, and since quantification normally leads to smaller BDDs, *early quantification* is employed, that is, variables are quantified as soon as logically possible. Let Q_i be the set variables u in $X \cup Y$ such that the conjunct C_i depends on u, but none of the subsequent conjuncts $C_{i+1}, \ldots C_k$ depend on u. Then, the image computation $Img(S)$ can be rewritten as $\exists Q_k \cdots (\exists Q_2(\exists Q_1.S \wedge C_1) \wedge C_2) \cdots][X' \rightarrow X]$. Thus, starting with S, at each step i, we take conjunction with the cluster C_i while quantifying the variables in Q_i. This scheme leads to significantly smaller intermediate BDDs. The effectiveness of this scheme depends on (1) *clustering*, that is, determining the clusters $\{C_1, \ldots C_k\}$ from the original description of the design, and (2) *ordering*, that is, sequencing the clusters $C_1, \ldots C_k$ so that as many variables get quantified out as early as possible. There has been a steady progress on good heuristics for clustering and ordering (see [11, 18, 17, 16, 6, 7] for sampling of this research).

Let us suppose that we have committed to a specific clustering as well as specific ordering of the clusters. Consider a variable u in Q_i, that is to be quantified at i-th step, and a variable v that does not appear in $S, C_1, \ldots C_i$. Then, the variable v can be renamed to u. In general, we partition the set $X \cup Y \cup X'$ of variables so that for every pair of variables within the same partition, the range of clusters that the two variables belong to are disjoint (we assume that all variables in X appear in the first cluster, and all variables in X' appear in the last cluster). Then, only a single variable identifier is needed per partition, and the image computation can proceed as before, but after renaming each variable to the unique identifier for its partition.

After formulating the problem of reducing the number of variables needed for image computation, we develop an algorithm for partitioning the variables into minimal number of sets. Our algorithm is quadratic in the number of variables. Our experiments with ISCAS'89 and Texas'97 benchmarks show a significant reduction in the number of variables by 40% to 68% . In fact, the number of

variables is much closer to the number n of variables in X than to the number m of variables in $X \cup Y \cup X'$. Since the reduced number of variables is the minimum necessary for image computation, we believe that the reduced number of variables is a better measure of complexity of the design compared to the number of state variables or the total number of variables.

After reducing the number of variables, image computation needs to be modified to account for renaming. A priori, it is difficult to estimate whether such a modification will improve or degrade the performance. On one hand there are clear advantages. Once a variable has been existentially quantified, the BDD nodes that are associated with the variable become inactive. However, the nodes are not garbage collected immediately. With variable reuse, the quantified variable will be reused in a different role. Instead of creating new BDD nodes and then increasing memory usage, the inactive nodes that are still in memory may be actived. Variable reuse will also benefit variable reordering algorithms because there are fewer BDD variables to be considered. Finally, we expect there is more sharing on BDD nodes because each node may take more than one roles. On the other hand,there is a potential disadvantage. Performance of BDD routines is extremely sensitive to the global ordering of variables, and reusing the same variable identifier in different roles can turn a good ordering into a bad one. Indeed, in presence of dynamic reordering, the computational requirements of image computation are very unpredictable.

We modified the image computation routine of VIS 1.4 model checker to verify the performance on ISCAS'89 and Texas'97 benchmarks. For the algorithm for generating the minimal number of variables, in many cases we obtained significant savings in memory and time, but in some cases, it performs worse than VIS. Consequently, we implemented a modified version of our strategy for partitioning variables which avoids pairing of a current-state variable x_i with a next-state variable x_j with $i \neq j$. The intuition for this lies in the fact that variable ordering and dynamic reordering schemes treat the variables x_i and x_i' as a pair, and renaming x_j' to x_i can be expensive. The modified partitioning requires more number of variables compared to the optimal one, but still significantly less than the number variables used by VIS. Another modification involves ordering of the clusters so as to reduce the number of variables required. In this greedy scheme, we pick the next cluster which will minimize the number of partitions of variables encountered so far, and when there are ties, we resort to the original VIS algorithm for ordering the clusters. With these modifications, we compared our reachability computation with time requirements for VIS 1.4. We get improvements in 29 out of 37 benchmarks, and in 6 cases, additional iterations of the image computation are feasible by our strategy (see Table 2 for details).

The remaining paper is organized as follows. Section 2 describes our strategy intuitively using an illustration. Section 3 formalizes the problem of reducing the number of variables, shows how to modify the image computation accounting for variable reuse, and gives algorithms for reducing the number of variables. Section 4 reports experimental results, and we conclude in Section 5 with directions for possible improvements.

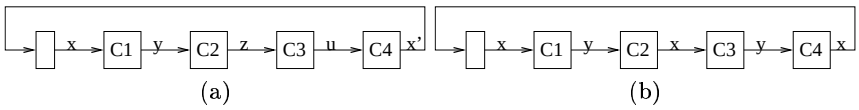

Fig. 1. Image computation: (a)without variable reuse, (b)with variable reuse

2 Variable Reuse Technique

Suppose we have fixed the ordering of clusters $C_1, ..., C_k$ and are committed to early quantification during image computation. The support set of the cluster C_i is the set of variables that appear in the BDD of C_i. The support set of S is considered to be X. Consider a variable $u \in X \cup Y$ such that u does not appear in the support sets of $C_{l+1}, \ldots C_k$, and thus, u can be quantified by step l. Consider a variable w that is not in the support sets of $S, C_1, ..., C_l$, then all occurrences of w in $C_{l+1}, ..., C_k$ can be replaced by u. Repeated applications of such variable replacement causes a variable appear repeatedly in different roles and get quantified repeatedly. As a result, fewer variables are involved in BDD computation. Since size of BDD in many cases is exponential in number of variables, variable reuse can reduce BDD size effectively by taking advantage of already constructed BDD nodes.

Figure 1 (a) shows a simple example for conventional image computation procedure. There are four clusters C_1, C_2, C_3, C_4 and five variables in which x is the current state variable, x' is the next state variable and y, z, u are internal variables. The following five steps are used to compute the set $Img(S(x))$: (1) $S_1(y) := \exists x.S(x) \wedge C_1(x, y)$, (2) $S_2(z) := \exists y.S_1(y) \wedge C_2(y, z)$, (3) $S_3(u) := \exists z.S_2(z) \wedge C_3(z, u)$, (4) $S_4(x') := \exists u.S_3(u) \wedge C_4(u, x')$, and (5) $Img(S(x)) := S_4(x')[x' \rightarrow x]$.

However, we observe that after computing $S_1(y)$, x is no longer in scope. Since the variable z has not been used until the computation of S_1 finishes, z can be renamed to x in the cluster C_2. By a similar reasoning, all of x, z and x' can share the same variable identifier, say x, and y and u can have the same name, say y. As shown in Figure 1 (b), only two variables are needed in the following image computation procedure: (1) $S_1(y) := \exists x.S(x) \wedge C_1(x, y)$, (2) $S_2(x) := \exists y.S_1(y) \wedge C_2(y, x)$, (3) $S_3(y) := \exists x.S_2(x) \wedge C_3(x, y)$, and (4) $Img(S(x)) := \exists y.S_3(y) \wedge C_4(y, x)$. Observe that, because we rename x' to x in cluster C_4, the last step of replacement of the next state variable by the current state variable is not needed. Next section discusses algorithms for partitioning the variable set. It turns out that variable partitioning is a low-cost procedure, and can easily be added to image computation.

By examining the modified image computation for our example, we can anticipate two potential benefits. First, the same variables, namely x and y, are involved in all four steps, and consequently, there can be more sharing, and reuse of existing BDD nodes during the computation. Second, with less variables involved, dynamic reordering, a key step in large examples, will take less

time. There is a potential drawback, however. A lot of research has been done to obtain a good initial variable order and dynamic reordering heuristics. With renaming, the initial order or dynamic reorder heuristics may be no longer effective. For example, if structure of C_1 requires x to be before y, and structure of C_2 requires y to be before z, then after renaming z to x, there is no good ordering.

3 Variable Reuse Algorithms

For a variable $u \in X \cup Y \cup X'$, let $M[u,i] = 1$ if variable u appears in the support set of cluster C_i. We consider $S(X)$ as the 0-th cluster as we wish to compute the conjunction $S \wedge C_1 \wedge \cdots \wedge C_k$. Since prior to the reachability computation, we don't know the exact set $S(X)$ at each image computation, we assume that all the current-state variables appear in $S(X)$. Therefore, $M[u,0] = 1$ for $u \in X$. We also let $M[u,k] = 1$ if $u \in X'$ because next-state variables cannot be quantified until the end of image computation. For each variable u, let $u.low = min\{i \mid M[u,i] = 1\}$, and $u.high = max\{i \mid M[u,i] = 1\}$. We declare the *range* of u to be the interval $Rng(u) = [u.low, u.high]$. Two variables u and v can be renamed to each other if $Rng(u)$ and $Rng(v)$ are disjoint.

Our goal is to partition the variables $X \cup Y \cup X'$ into disjoint sets $p_1, ..., p_r$ such that $Rng(u) \cap Rng(v) = \emptyset$ if two variables u and v belong to the same partition set p_i. Note that all the variables in a partition set p_i can have the same name, and the number of partitions is the number of variables needed for reachability computation. For a set p of variables, define $p.high = max\{u.high \mid u \in p\}$, $p.low = min\{u.low \mid u \in p\}$, and $Rng(p) = [p.low, p.high]$. The problem can be viewed as an application of constructing a maximal independent set in an interval graph. While maximal independent sets are hard to compute for general graphs, the problem is solvable in polynomial time for interval graphs [12].

3.1 The Minimal Gap Algorithm

Figure 2 shows an algorithm to partition the variables into disjoint sets. Let the A be the set of all the current state, next state and auxiliary variables. We first sort variables by their upper values. The variables with the same upper value are ordered by their lower values. That is, for all $1 \leq i < j \leq m$, either $(A[i].high < A[j].high)$ or $(A[i].high = A[j].high \wedge A[i].low \leq A[j].low)$ holds. The remaining code iterates over the sorted variables to find an existing partition or create a new partition. A variable u can be added to an existing partition p if $Rng(u) \cap Rng(p) = \emptyset$. There may exist several partitions that satisfies the condition. Our strategy is to insert u into a set p such that u and p have the minimal "gap". As implemented in the function `get_mingap_set`, u can be added to a set p if $u.low - p.high > 0$. This is because $p.high$ is the maximal upper value of a variable in p, therefore, $u.low - p.high > 0$ means the range of u is disjoint from the range of any variable in p. Although u can also be inserted into p if $p.low - u.high > 0$, this is not possible because all the variables in p appear

```
min_gap()
  A := X ∪ Y ∪ X';
  sort A by A[i].high;
  foreach variable u ∈ A
    p := get_mingap_set(P, u);
    if (p ≠ ∅)
      p := p ∪ {u}; p.high := u.high;
    else
      p' := {u}; p'.high = u.high;
      P = P ∪ {p'};
  end
end

set get_mingap_set(P, u)
  result := ∅;
  mingap := MAX;
  foreach set p ∈ P
    gap := u.low − p.high;
    if (gap>0 && gap<mingap)
      mingap := gap; result := p;
  end
  return result
end
```

Fig. 2. Minimal gap algorithm

before u in the array ordered by their high values. If no such p disjoint from u exists, `get_mingap_set` returns the default $\emptyset$, otherwise it returns p for which the value $u.low - p.high$ is the smallest.

Finally the partition P is modified in the function `min_gap`. If there exists an appropriate set p that u can be added to, u is inserted in p, and the value $p.high$ is updated with $u.high$ (note in this case $u.high$ is guaranteed to be greater than $p.high$). If u cannot be inserted in any existing set, a new set is created for u, and its *high* field is set to $u.high$ (note that the algorithm does not actually need to keep track of *low* fields for the partitions).

Theorem 1. *Given a set of variables u with specified ranges $[u.low, u.high]$, the algorithm* `min_gap` *shown in Figure 2 creates minimal number of sets such that variables in the same set have pair-wise disjoint ranges.*

Proof: For $i = 1 \ldots m$, let P_i be the (partial) partition created by our algorithm after processing the variables $A[1] \ldots A[i]$. Let us say that P_i is (partially) correct if P_i can be extended to obtain a partition with optimal number of sets. Let i be the smallest index such that P_{i+1} is not correct (if no such i exists, the final partition has optimum number of sets, and we are done). Let $u = A[i+1]$. Thus, $A[i]$ is extensible to an optimal partition, say P_f, but the algorithm makes a mistake while processing u.

First, suppose $Rng(p)$ overlaps $Rng(u)$ for every p in P_i. In this case, P_{i+1} is obtained by adding a new set $\{u\}$ to P_i. Since P_f extends P_i, it must extend P_{i+1} also, a contradiction.

Now suppose our algorithm decides to add u to an existing set p. Note that since the array is sorted, $u.high$ is the smallest among all unprocessed (i.e. not already in P_i) variables. Consequently, the left neighbor of u in the optimal partition P_f is a processed variable (the case that u has no left neighbor in P_f is similar). Suppose the left neighbor of u in P_f belongs to the set $p' \neq p$ in P_i. Both p and p' cannot overlap with u: $p.high > u.low$ and $p'.high > u.low$. The way the algorithm `get_mingap_set` chooses p, the gap between p and u is less than

(or equal to) the gap between p' and u: $p'.high \leq p.high$. Consider P'_{i+1} obtained by adding u to p' in P_i. The only relevant information about a partial partition, for possible ways of adding the unprocessed variables to it, is the sequence of high end-points of the sets in the partition. If we compare the high end-points of the sets in P_{i+1} and P'_{i+1}, they are $\{p'.high, u.high\} \cup \{p''.high \mid p'' \in P_i, p'' \neq p, p'' \neq p'\}$ for P_{i+1}, and $\{p.high, u.high\} \cup \{p''.high \mid p'' \in P_i, p'' \neq p, p'' \neq p'\}$ for P'_{i+1}. Thus, P_{i+1} extends less to the right. It is straightforward to show that P_{i+1} is also extensible, a contradiction. ■

As far as time complexity of the algorithm is concerned, observe that the number of partitions is at most m, where m is the total number of variables. Consequently, `get_mingap_set` is $O(m)$. As a result, the complexity of `min_gap` is $O(m^2)$.

3.2 Modifying the Image Computation

Let $P = \{p_1, ..., p_r\}$ be the partition sets created by our algorithm. Let l_i be the *leading variable* in p_i to which all the other variables appearing in the same set will be renamed. Let L be a mapping function such that $L(u) = l_i$ if $u \in p_i$. The mapping L naturally extends to sets of variables.

A common strategy for partitioning and ordering of the transition relations proceeds in two steps: obtain clusters of transition relations from the design description usually by combining the BDDs for fine-grain transition relations for the atomic blocks, and then order clusters with a quantification schedule to allow for maximum early quantification. Assume after the second step, we have a sequence of clusters $C_1, \ldots, C_k$ and a sequence of variable sets $Q_1, \ldots, Q_k$ for quantification. The standard image computation algorithm follows: $\exists Q_k \cdots \exists (Q_2 (\exists Q_1 . S \wedge C_1) \wedge C_2) \cdots][X' \rightarrow X]$. We need to introduce a renaming step before the image computation step starts.

Let T_i be the support set of C_i. We compute a BDD C'_i by substituting the variables appearing in C_i with $L(T_i)$, i.e, $C'_i = C_i[T_i \rightarrow L(T_i)]$. Similarly, for $Q_i \subseteq T_i$, we obtain a new set $Q'_i = L(Q_i) \subseteq L(T_i)$. The revised image computation is: $\exists Q'_k \cdots \exists (Q'_2 (\exists Q'_1 . S' \wedge C'_1) \wedge C'_2) \cdots][A \rightarrow B]$.
Note that the substitution at the end of image computation becomes $[A \rightarrow B]$ instead of $[X' \rightarrow X]$, where A and B are obtained as discussed below.

Although any variable in a partition set can be a lead variable, picking the right one leads to more efficient strategy. Following rules are followed in our implementation while choosing a lead variable.

1. *Choose current state variables first.*
   If there exists a current state variable $x_i \in X$ in a partition set, we always choose x_i as the lead variable. Note that two current state variables cannot be in the same partition because $x_i.low = x_j.low = 0$, and thus $Rng(x_i) \cap Rng(x_j) \neq \emptyset$, for $i \neq j$. Such renaming has the benefit that if the matching next state variable x'_i belongs to the same partition, the substitution $[x'_i \rightarrow x_i]$ at the end of image computation is not necessary.

2. *Choose next state variable if no current state variable exists.*
   A next state variable $x' \in X'$ should not be renamed to an internal variable $y \in Y$. This is because at the end of image computation we need to convert next state variables to current state variables by $[X' \rightarrow X]$. For this reason, we always rename all the internal variables to the next state variable x' in a partition p. Note a partition can have at most one next state variable because for any $x'_i, x'_j \in X'$, $x'_i.high = x'_j.high$.
3. *Choose any variable if no state variable exists.*
   If there are no current state or next state variables in a partition, any variable can be chosen as a lead variable.

The above rules specify how to choose the lead variables in each partition, and thus, how to fix the renaming map L. Finally, consider the substitution $[X' \rightarrow X]$ at the end of image computation. If x'_i is renamed to x_j by L, where $i \neq j$, we cannot use $[X' \rightarrow X]$ as it is. We first need to rename x_j to x_i, followed by the standard renaming $[X' \rightarrow X]$. A better method is to combine the two conversion steps into one step $[A \subseteq (X \cup X') \rightarrow B \subseteq X]$. The two arrays are defined as follows: for each state variable $x_i \in X$, if $L(x'_i) = x_j$ with $j \neq i$, then $A[i] = x_j$ and $B[i] = x_i$, else $A[i] = x'_i$ and $B[i] = x_i$. It should be noted that BDD packages perform such renaming in parallel.

3.3 The Least Effort Algorithm

Although Minimal Gap Algorithm creates minimal number of partitions, it requires extra effort to get the substitution arrays of A and B, and the substitution $[A \rightarrow B]$ can be expensive. This section presents a greedy algorithm `least_effort` that has additional constraints in renaming. In particular, it treats the next-state variables specially since variable ordering and dynamic reordering try to keep the variables x_i and x'_i together as a pair. The strategy `least_effort` uses more variables than `min_gap`, but no changes are needed for the substitution step $[X' \rightarrow X]$. Therefore, we only need to change variable names in clusters and quantification arrays before reachability computation starts and no modification is required in existing image computation code.

The algorithm `least_effort` is the same as `min_gap` except that the function call to `get_mingap_set` is replaced by `get_constrained_mingap_set` shown in Figure 3. Note that unlike `get_mingap_set`, `get_constrained_mingap_set` treats next state variables differently. For each $x' \in X'$, it first tries to obtain a set that satisfies two constraints: (1) it contains the matching current state variable x, and (2) $p.high < x'.low$. The first constraint gives priority to a set such that x' can be renamed to its matching current state variable. The second constraint ensures that x' can be added to the set.

If the second constraint cannot be satisfied (note that the first constraint can always be met), it tries to find any set p that $(p.high < x'.low)$ and $p \cap X = \emptyset$. The constraint of $p \cap X = \emptyset$ ensures that x' will not be in the same set with a non-matching current state variable. If such a set cannot be found, it returns empty set, and x' will be put in a new partition by itself.

```
set get_constrained_mingap_set(P, u)
  if u ∈ X'
    choose v such that v ∈ X and v matches u;
    choose p such that v ∈ p and p.high < u.low;
    if p = ∅
      choose p such that p.high < u.low and (p ∩ X = ∅);
  else
    p := get_mingap_set(P, u);
  return p
end
```

Fig. 3. Get disjoint partition with constraints

For any auxiliary and current state variables, it uses `get_mingap_set` shown in Figure 2 to get the appropriate set. Note that the algorithm prevents the case that a current state variable is added to a set containing a non-matching next state variable. This is because the variables are sorted by first their high end-points and then their low values. Since $x'.high = k$, $x'.low > 0$, $x.high \leq k$, and $x.low = 0$, we have all current-state variables appearing before all next-state variables in the sorted order. Therefore, all the current-state variables have been allocated to some partition sets before any of the next-state variables are handled.

With the constraint that a next state variable cannot be renamed to a non-matching current state variable, we don't need to adjust existing image computation algorithm. Image computation can use the formula $[\exists Q'_k \cdots \exists (Q'_2 (\exists Q'_1 . S \wedge C'_1) \wedge C'_2) \cdots][X' \rightarrow X]$.

3.4 The Greedy Algorithm for Ordering Clusters and Partitioning Variables

The previous algorithms partition variables assuming a given fixed ordering of clusters. The greedy algorithm presented in this section orders the clusters so as to favor minimal number of partitions at each step.

The function `greedy_partition` in Figure 4 shows the algorithm with the set of clusters C as the input. Initially the sorted cluster array S is empty. The first step is to initialize the range values for variables by function call `init_var_range`. As explained earlier, current-state variables have a common lower bound 0 and next-state variables have a common upper bound $|C|$. The lower and upper bounds of internal variables, as well as the upper bounds of current-state variables and lower bounds of next-state variables, are UNKNOWN. The initial partition set P consists of $|X|$ singleton sets each containing a distinct current-state variable. The iteration of `while` picks a cluster from C and adds it to the sorted cluster array S in each step. The partition P is then modified accordingly. The function `modify_partition` uses the same code as the lines 6 to 10 in `min_gap` in Figure 2. To decide which cluster to pick, the loop of `foreach` goes

```
greedy_partition(C)
  init_var_range();
  P = {{x}| x ∈ X };
  while C ≠ ∅
    foreach C_i ∈ C
      cost=compute_cost(S,C_i,C,P);
      if cost<MIN
        picked = C_i; MIN = cost;
    end
    S = S ∪ {picked}; C = C − {picked};
    modify_partitions(P, picked)
  end
end
```

```
int compute_cost(S,C_i,C,P)
  foreach v ∈ support_set(C_i)
    if v.low == UNKNOWN
      v.low = |S| + 1;
    if v ∉ support_set(∪_{j≠i} C_j
      v.high = |S| + 1;
    if ∃P_i|P_i.high < v.low
      P_i.high = v.high;
    else
      num_new_partitions ++;
  end
  return num_new_partitions;
end
```

Fig. 4. The greedy algorithm to order clusters and partition variables

through all the remaining clusters and calculate the effect if the variables from a cluster were added to the existing partitions. The one with the least cost is chosen. Finally the existing partitions are modified by incorporating the variables from newly selected clusters. Since the position of the selected cluster in sorted array is known, the ranges of partitions and variables can be updated.

The function `compute_cost` in Figure 4 shows how to calculate the number of partitions if variables from cluster C_i were added to partitions. There are four parameters: S, the sorted cluster array; C_i, the cluster under consideration; C, the clusters not sorted yet; P, the current variable partitions. Initially the number of new partitions is 0. If C_i is appended to the sorted cluster array, the index of C_i will be $|S| + 1$. We may set the ranges of the variables based on the location of C_i. The function `compute_cost` tries to set the lower and upper bounds for variables in the support set of C_i . The lower bound of the variable v is $|S|+1$ if $v.low$ is UNKNOWN. This is because C_i must be the first occurrence of v in the sorted cluster array. If v does not appear in the supports of clusters $C - C_i$, C_i would be the last occurrence of v in S. Therefore, the upper bound of v should be $|S| + 1$. After a variable has been assigned upper (possibly still UNKNOWN) and lower bound, the function looks for a partition P_i such that $P_i.high < v.low$. Note that partitions with open upper bound value UNKNOWN does not satisfy $P_i.high < v.low$. If such partition exists, the upper bound of P_i is increased to the upper bound of v (may be UNKNOWN) because we consider v is to be added to P_i; otherwise, a new partition has to be created. Therefore, number of partitions is increased by 1. However, we do not need to really allocate a new partition because we only need to know how many new partitions have to be created, and no two variables from C_i may reside in the same partition, which makes the knowledge of upper and lower bounds of new partitions unnecessary.

The function `compute_cost` considers the cost to be the number of new partitions created. In the implementation, we distinguish between *open partitions* and *close partitions*. The partitions with unknown upper bound are close partitions;

otherwise they are open partitions. Since open partitions do not accept any new variables, we prefer close partitions. In this case, the cost to add a new cluster c becomes $cost = num\_close\_partitions + weight * num\_close\_partitions$ where $weight > 1$. We also consider the number of variables V_{sorted} that have been assigned to a partition set so far. In such case, the cost becomes $num\_partitions/|V_{sorted}|$.

4 Experimental Results

4.1 Reduction in the Number of Variables

In order to evaluate the effectiveness of our variable renaming algorithms, we ran experiments on ISCAS'89 and Texas'97 benchmarks. All experiments were done on a 800MHz Pentium III processor machine running the Linux operating system with 512MB of main memory. In all the experiments, we used the default VIS options (partition threshold=5000, frontier method for building partition BDDs, and image cluster size=5000).

The performance metric we measured in this section are the number of variables required for reachability computation. Table 1 shows experimental results with the first column indicating circuit names. The second column shows the number of latches in the circuits. Since the current state variables have to be in distinct partitions, the number of latches is the lower bound on the number variables for any of the strategies. The third column indicates the number of clusters. Finally, the last four columns provide the number of variables required for different algorithms.

Column "VIS" lists the number of variables by using original VIS-1.4 code. Columns "MG", "LE" and "GD" report the number of variables by `min_gap`, `least_effort` and `greedy` algorithms, respectively. The last column shows the minimal number of variables for one of our renaming algorithms compared with VIS. We can see that only 40%-74% of VIS variables are actually required for reachability computation.

4.2 Savings during Reachability Computation

As discussed in the introduction, it is difficult to predict the impact of our strategy on memory and time requirements during image computation, and can be estimated only by experiments with benchmarks. We integrated our modifications within VIS. For these experiments, we invoke dynamic variable reordering in the CUDD package (see `http://vlsi.colorado.edu/~fabio/CUDD/` for information). A time limit of one hour was used for all experiments. The results are reported in Table 2. It compares VIS 1.4 with the best results obtained from either the modification MG with `min_gap`, the modification LE with `least_effort` or the modification GD with `greedy`. For each algorithm the table lists, the number of steps of image computation, the number of states (this is only to verify that actual reachable sets are identical), and time required.

Of the 37 benchmarks tested, variable renaming optimization performs better than VIS on 29 examples, while VIS does better on 8 examples. The last column indicates the comparison. If the value t is greater than 1, our modification achieves a speedup of $t\times$(e.g., on `three_processor_bin`, reachability computation after variable renaming speeds up by 3.61). If the value t is in the format of $+ts$, variable rename algorithm can do t steps further than VIS (e.g., on `s1423`, reachability computation after variable renaming can do one more step). On the other hand, If the value in the last column is less than 1 or in the format of $-ts$, VIS without variable rename is better. Given the non-robust nature of computational requirements of BDD packages, particularly due to dynamic reordering, we consider these results to be promising.

5 Conclusion

In this paper, we have proposed a technique for reducing the number of auxiliary variables required for image computation in symbolic reachability analysis. This idea complements the previous research on conjunctive partitioning and early quantification. Unlike typical optimization problems in design automation, reducing the number of variables optimally turns out to have a polynomial-time solution. Our experiments concerning reducing the number of variables indicate that the number of auxiliary variables necessary for image computation is quite small compared to the number of state variables. In terms of savings in time, our heuristic improves on the state-of-the-art model checker VIS 1.4 on many benchmarks, sometimes leading to a speed up of 7.61 , and sometimes allowing extra iterations of image computation. There is little effort needed to compute the renaming, and our strategy can be incorporated in other model checkers with minimal effort.

The benefits of the proposed heuristic can potentially be improved in many ways. First, the techniques for generating clusters and ordering clusters can be modified to account for variable renaming strategy. In fact, there seems to be no need to assign variable identifiers for all the variables right from the beginning, but to combine all of these preprocessing steps. Second, and possibly more importantly, the heuristics for choosing the initial order and dynamic reordering during image computation need to be examined carefully in light of our strategy. In particular, the fact that a variable appears in multiple roles can be taken into account while choosing the ordering, and the pairing of a current-state variable with next-state variable can be non-essential in our setting. Finally, similar ideas can be explored in conjunction with recent efforts on bounded model checking using SAT solvers [1, 13].

References

1. A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of the 36th ACM/IEEE Design Automation Conference*, pages 317–320, 1999.

2. R. Brayton, G. Hachtel, A. Sangiovanni-Vincentell, F. Somenzi, A. Aziz, S. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. Ranjan, S. Sarwary, T. Shiple, G. Swamy, and T. Villa. VIS: A system for verification and synthesis. In *Proceedings of the Eighth International Conference on Computer Aided Verification*, LNCS 1102, pages 428–432. Springer-Verlag, 1996.
3. R.E. Bryant. Graph-based algorithms for boolean-function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.
4. J.R. Burch, E.M. Clarke, D.L. Dill, L.J. Hwang, and K.L. McMillan. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.
5. J.R. Burch, E.M. Clarke, and D.E. Long. Symbolic model checking with partitioned transition relations. In *Proceedings of the IFIP International Conference on Very Large Scale Integration: VLSI'91*, pages 49–58, 1991.
6. P. Chauhan, E. Clarke, S. Jha, J. Kukula, H. Veith, and D. Wang. Using combinatorial optimization methods for quantifier scheduling. In *Proceedings of CHARME'01*, LNCS 2144, pages 293–309, 2001.
7. P. Chauhan, E.M. Clarke, S. Jha, J. Kukula, T. Shiple, H. Veith, and D. Wong. Non-linear quantification scheduling in image computation. In *Proceedings of the International Conference on Computer Aided Design: ICCAD'01*, 2001.
8. E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logic of Programs*, LNCS 131, pages 52–71. Springer-Verlag, 1981.
9. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model checking*. MIT Press, 2000.
10. E.M. Clarke and R.P. Kurshan. Computer-aided verification. *IEEE Spectrum*, 33(6):61–67, 1996.
11. D. Geist and I. Beer. Efficient model checking by automated ordering of transition relation partitions. In *Computer Aided Verification, Proc. 6th Int. Conference*, LNCS 818, pages 299–310. Springer-Verlag, 1994.
12. M. Golumbic. *Algorithmic Graph Theory and Perfect Graph*. Academic Press, 1980.
13. A. Gupta, Z. Yang, P. Ashar, and A. Gupta. SAT-based image computation with applications in reachability analysis. In *Proceedings of the Third International Workshop on Formal Methods in Computer-Aided Design*, LNCS 1954, pages 354–371. Springer, 2000.
14. R. Hardin, Z. Har'El, and R.P. Kurshan. COSPAN. In *Proceedings of the Eighth International Conference on Computer Aided Verification*, LNCS 1102, pages 423–427. Springer-Verlag, 1996.
15. K. McMillan. *Symbolic model checking: an approach to the state explosion problem*. Kluwer Academic Publishers, 1993.
16. I.-H. Moon, J.H. Kukula, K. Ravi, and F. Somenzi. To split or to conjoin: the question in image computation. In *Proceedings of the 37th Design Automation Conference*, pages 26–28, 2000.
17. I.-H. Moon and F. Somenzi. Border-block traingular form and conjunction schedule in image computation. In *Proceedings of the Third International Workshop on Formal Methods in Computer-Aided Design*, LNCS 1954, pages 73–90. Springer, 2000.
18. R. Ranjan, A. Aziz, B. Plessier, C. Pixley, and R. Brayton. Efficient BDD algorithms for FSM synthesis and verification. In *Proceedings of the IEEE/ACM International Workshop on Logic Synthesis*, 1995.

Circuit Name	Latches	Clusters	Number of variables				min
			VIS	MG	LE	GD	%
three_processor_bin	62	10	142	89	111	100	63%
three_processor	62	12	142	90	115	95	63%
IFetchControl1	59	5	146	90	113	99	62%
IFetchControl2	59	5	145	91	114	91	63%
TWO	64	9	204	93	122	99	46%
PCIabnorm	295	23	617	355	588	359	58%
PCInorm	295	31	612	355	588	342	56%
test4	153	16	315	189	314	213	60%
timestamp	79	7	168	114	168	128	68%
multi_master	382	24	781	479	764	441	63%
p62_L_L_V01	308	34	618	374	598	397	61%
p62_L_S_V02	308	34	618	369	595	396	60%
p62_ND_S_V02	308	35	622	362	597	405	58%
p62_L_L_V02	308	34	618	374	598	397	61%
p62_S_S_V01	308	34	618	374	598	397	61%
p62_LS_LS_V01	308	34	618	374	598	397	61%
p62_S_S_V02	308	34	618	374	598	397	61%
p62_LS_LS_V02	308	34	618	374	598	397	61%
p62_LS_L_V01	308	34	618	369	595	396	60%
p62_V_LS_V02	308	34	622	369	595	396	59%
p62_LS_L_V02	308	34	618	369	595	396	60%
p62_ND_ND_V01	308	35	622	395	599	405	64%
p62_V_S_V01	308	34	622	369	595	396	59%
p62_LS_S_V01	308	34	618	369	595	396	60%
p62_ND_ND_V02	308	35	622	395	599	405	64%
p62_V_S_V02	308	34	622	369	595	396	59%
p62_LS_S_V02	308	34	618	369	595	396	60%
p62_ND_ND_V	308	34	622	395	599	405	64%
p62_L_S_V01	308	34	618	369	595	396	60%
p62_ND_S_V01	308	35	622	375	597	405	60%
s1269	37	8	92	64	70	58	63%
s1512	57	4	143	95	114	95	66%
s1423	74	7	165	122	151	134	74%
s4863	104	33	263	112	132	105	40%
s9234	211	23	461	253	387	267	55%
s13207	638	42	1339	638	1001	710	48%
s15850	522	51	1147	665	1014	706	58%

Table 1. Number of variables

Circuit Name	VIS			RENAME			
	#step	#states	Time	#step	#states	Time	Speedup
three_processor_bin	32	3.6e+08	145.23	32	3.6e+08	40.25	3.61
three_processor	32	3.6e+08	98.02	32	3.6e+08	26.96	3.64
IFetchControl1	27	4.3e+08	4.69	27	4.3e+08	10.48	0.45
IFetchControl2	27	2.5e+08	2.25	27	2.5e+08	1.70	1.32
TWO	21	1.3e+14	42.76	21	1.3e+14	43.76	0.98
PCIabnorm	35	2.6e+06	23.09	35	2.6e+06	9.78	2.36
PCInorm	30	86528	1.15	30	86528	0.99	1.16
test4	17	4.0e+18	1677.67	18	7.8e+20	3121.66	+1s
timestamp	26	3.2e+22	21.75	26	3.2e+22	9.33	2.33
multi_master	41	1.1e+06	20.56	41	1.1e+06	3.27	6.29
p62_L_L_V01	48	2445	46.09	48	2445	15.11	3.05
p62_L_S_V02	73	1327	13.52	73	1327	15.09	0.90
p62_ND_S_V02	106	143788	316.05	106	143788	260.00	1.22
p62_L_L_V02	71	2398	54.30	71	2398	29.53	1.84
p62_S_S_V01	45	437	45.07	45	437	5.92	7.61
p62_LS_LS_V01	61	2823	123.81	61	2823	39.24	3.16
p62_S_S_V02	42	317	2.82	42	317	2.41	1.17
p62_LS_LS_V02	58	1045	41.57	58	1045	12.37	3.36
p62_LS_L_V01	61	2743	77.27	61	2743	38.57	2.00
p62_V_LS_V02	125	93185	385.94	125	93185	381.38	1.01
p62_LS_L_V02	66	1124	12.78	66	1124	11.31	1.13
p62_ND_ND_V01	25	310073	1178.86	32	1.3e+06	2209.86	+7s
p62_V_S_V01	121	59667	374.39	121	59667	386.23	0.97
p62_LS_S_V01	61	2743	77.51	61	2743	38.48	2.01
p62_ND_ND_V02	36	1.1e+06	1674.22	40	2.3e+06	2138.61	+4s
p62_V_S_V02	106	22309	142.28	106	22309	128.63	1.11
p62_LS_S_V02	66	1124	12.84	66	1124	11.34	1.10
p62_ND_ND_V	25	310073	1181.82	32	1.3e+06	2206.98	+7s
p62_L_S_V01	85	3637	79.04	85	3637	46.06	1.72
p62_ND_S_V01	55	378695	1263.28	68	1.6e+06	2742.20	+13s
s1269	9	1.1e+09	2460.04	9	1.1e+09	2668.87	0.92
s1512	1023	1.6e+12	1292.22	1023	1.6e+12	954.87	1.35
s1423	10	1.6e+09	1120.93	11	7.9e+09	2811.42	+1s
s4863	4	2.1e+19	675.58	4	2.1e+19	454.04	1.49
s9234	8	1.3e+14	1137.86	8	1.3e+14	1285.95	0.88
s13207	10	6.6e+26	2976.30	6	8.1e+22	1970.93	-4s
s15850	2	7.4e+12	595.92	2	7.4e+12	895.19	0.67

Table 2. Reachability Computation