HEURISTIC ALGORITHMS FOR EARLY QUANTIFICATION AND PARTIAL PRODUCT MINIMIZATION

by

Ramin Hojati, Sriram Krishnan, and Robert K. Brayton

Memorandum No. UCB/ERL M94/11

9 March 1994

HEURISTIC ALGORITHMS FOR EARLY QUANTIFICATION AND PARTIAL PRODUCT MINIMIZATION

by

Ramin Hojati, Sriram Krishnan, and Robert K. Brayton

Memorandum No. UCB/ERL M94/11

9 March 1994

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

HEURISTIC ALGORITHMS FOR EARLY QUANTIFICATION AND PARTIAL PRODUCT MINIMIZATION

by

Ramin Hojati, Sriram Krishnan, and Robert K. Brayton

Memorandum No. UCB/ERL M94/11

9 March 1994

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Heuristic Algorithms for Early Quantification and Partial Product Minimization

Ramin Hojati, Sriram Krishnan,
Robert K. Brayton

Abstract[1]

A set of interacting finite state machines is often used as a model for formal verification. Most formal verification algorithms, based on Binary Decision Diagrams (BDD's), build the transition relation of the product machine, and then existentially quantify out the non-state variables. The early quantification problem is to compute a schedule for multiplying and existentially quantifying variables, such that the maximum size BDD encountered at any point is minimized. We give two algorithms for the early quantification problem, one which is rather fast (running in linear time on sparse structures), and produces excellent results and another which produces an optimal schedule for a given linear ordering of the terms.

Even with early quantification partial products can become very large. In this paper, we present techniques to find don't cares, with respect to which the partial products are minimized. Some of our techniques involve state minimization and can result in smaller BDD's for the reachable states set. Two notions for state minimization are new, and involve approximations to trace equivalence. All the algorithms have been implemented and integrated in our formal verification software for design verification. We present our experimental results.

1 Introduction

Design Verification is the process of answering the question "Is what I specified what I wanted?" This is accomplished by specifying the system at a suitable level of abstraction and then proving properties of the system. For example, in a large design one can abstract the design to functional blocks and verify that the communication between blocks is correct. An example of a property is that no two functional units write to the global bus at the same time, a so-called safety property.

The most widely used scheme for modeling a system is as a set of interacting non-deterministic finite state machines, where the composite behavior is represented by their product machine. Recently, Binary Decision Diagrams (BDD's) ([Bry86]) have been used for representing FSM's. Using this scheme, the transition relation of each machine is represented by a BDD. Let $T_j(x_j, i_j, y_j)$ be the transition relation of the j-th FSM, where x_j represents the present state variable of the machine, y_j the next state, and i_j the set of inputs and outputs of the machine. We generally assume that the system is closed, so all inputs to a machine are produced by some other machine. The product machine is then represented by $T(x, i, y) = \prod_{j=1}^{n} T_j(x_j, i_j, y_j)$, where n is the number of FSM's, x and y are the set of present and next state variables, and i is the rest of the variables (referred to as the ***i/o variables***).

Most verification algorithms (see [HTKB92] for example) work just on the underlying graph, where the i/o variables have been existentially quantified out. The early quantification problem is to efficiently compute $\exists (i_1, ..., i_m) \prod_{j=1}^{n} T_j(x_j, i_j, y_j)$. The efficiency of this computation is in intelligently interleaving quantification with conjunction. Assume a partial product involves a variable i_k, which is not used in any other term. This variable can be quantified out after this partial product has been formed. A good solution to this problem can make a big difference in the efficiency of the verification tool. This is even more so, if the variables are poorly ordered.

There have been two earlier attempts at solving this problem. [TSLBS90] suggested using a balanced binary tree, where a variable is quantified out as soon as possible, i.e. as soon as all the terms it is present in are part of the current partial product. [Bur91] suggested multiplying the terms in a left-to-right fashion, again quantifying out a variable as soon as possible. Neither one of these methods offer a solution to the problem of automatically ordering the terms, which greatly affects the computation time and the largest BDD observed.

In this paper, we present two algorithms for this problem. The first a local optimization algorithm (Loc-Opt), is very fast, running in expected linear time on sparse structures, and produces excellent experimental results. The second algorithm (Ex-Lin-Ord) returns an optimum solution given a linear ordering of the terms. However, it has a cubic dependency on the number of relations.

Applications of early quantification come up in two places. The first is the problem mentioned above, where the product machine of a set of FSM's is computed. The second place early quantification arises is the translation of high-level languages, such as Verilog, into intermediate languages (for example BLIF-MV [Blifmv93]). During this process, many intermediate relations may be created. Early quantification can then be used to form the transition relation of the FSM.

The high-level structure of the Loc-Opt algorithm is simple. At every point, a forest, i.e a set of trees, is given. Each tree represents a partial product and involves a set of variables. The best merge is chosen next (some hill-climbing moves are allowed), and the corresponding trees are combined. A merge

1. During this work, the first author was supported by SRC grant 94-DC-008.

corresponds to multiplying two partial products and existentially quantifying out a set of variables. Since actual cost, i.e. the number of resulting BDD nodes cannot be used, an approximation to the cost is needed. As an approximation, we use the number of variables in the support of the resulting BDD after the multiplication and quantification. Although, this cost function is crude, it seems to work well in practice, and it is easy to compute.

The Ex-Lin-Ord algorithm is based on dynamic programming. Given a (fixed) linear ordering of the terms it computes all ways of computing a product of n terms subject to the restriction of the linear ordering. We have tried out two different cost functions. The first one is similar to the above, and uses the sizes of support sets as abstraction of the size, and 2. The second one called the *early quantification* cost function uses the notion of the cost of a variables, which is the number of *ands* starting from the primitive (original) terms performed to form the partial product from which the variable is being smoothed out. Ex-Lin-Ord subsumes [TSLBS90] and [Bur91] since their schedules are amongst the various schedules evaluated.

Even with early quantification, the partial products may grow large. We present techniques which minimize the BDD's based on extracting don't care information, using various techniques. Three previous methods have relevance. The first, known as ***incremental minimization***, was introduced in [BFH92]. The idea is to minimize the state graph as it is being built. The second approach, that of [FKM93], presents a BDD-based algorithm for state minimization using bisimulation after the product machine is built. One may wonder why one wants to minimize a transition relation of a machine after it is built. Since most verification algorithms need to compute the set of reachable states, and it could be the case that the product machine can be built, but not the set of reachable states, this approach has some applicability. Since state minimization reduces the size of the set of reachable states, it may also reduce the size of the BDD's representing the set of reachable states. Finally, the paper [CSSB92] has relevance to our work, since, in a somewhat different context, they proposed BDD minimization using don't cares during incremental checking of CTL formulas.

We present techniques for minimizing partial products when BDD's of an early quantification schedule are computed. The idea is to perform the multiplication and quantification on the early quantification tree until some BDD becomes large. Algorithms, such as state minimization can then be applied to find one or more sets of don't care information. The BDD's can then be minimized with respect to this don't care set. We refer to this problem as ***partial product minimization***.

We present algorithms for minimization using the reachability set, minimization using simulation equivalence and bisimulation, and minimization using approximations to trace equivalence. Our algorithms for approximating trace equivalence are new, and to the best of our knowledge provide the only practical ways to find lower bounds on how well bisimulation techniques perform as approximations to trace equivalence. Note that minimization with respect to trace equivalence can obtain more minimization, since bisimulation and simulation equivalence are conservative approximations to trace equivalence. All these algorithms have been implemented and integrated in our formal verification tool.

Our initial experiments show that bisimulation based minimization is rather expensive (basically in the same BDD complexity class as transitive closure), so it has to be applied with care. However, there are examples where bisimulation works well [FKM93]. To get a robust implementation, care is needed. A scheme, where more expensive minimizations are applied at the beginning, and less expensive ones, such as reachability minimization, are applied at the later stages can be used. Automatic ways of deciding when to apply each minimization remains an open question.

The flow of the paper is as follows. Section 2 presents the algorithm for early quantification. Sections 3 presents the Ex-Lin-Ord algorithm. Section 4 presents experimental results for the early quantification. Section 5 presents techniques for partial product minimization. Section 6 presents experimental results for partial product minimization. Section 7 concludes the paper.

2 The Problem and the Loc-Opt Algorithm

In this section, we define the early quantification problem, and describe the Loc-Opt algorithm for early quantification. In what follows, assume a set of relations $T_1, ..., T_n$, and a set of variables $v_1, ..., v_m$, to be quantified, are given.

2.1 Problem Setup

Definition An ***(early) quantification tree*** is a binary tree, whose leaves are the relations. At each node of the tree, a subset of variables $v_{n_1}, ..., v_{n_l}$, called the ***quantified variables at*** n, are marked as being quantified. If a variable is quantified at node n, then that variable must occur only in those relations which are in the subtree of node n. All variables in $v_1, ..., v_m$ have to be quantified at some node of the tree.

Intuitively, a quantification tree is a schedule to multiply and existentially quantify the variables. Note that by definition a variable can be quantified only after all relations, in which it occurs, have been multiplied. We use multiply and *and* synonymously with conjunct.

Definition The ***product of an early quantification tree*** is the relation obtained by recursively taking the product of the children of a node, and then quantifying the variables of the node. The partial product at a node n is the product of the early quantification tree rooted at n.

Definition The ***support of a node*** n, denoted $support(n)$, is the union of the supports of all relations in the subtree rooted at n, minus the variables which have been quantified at n or below. In other words, these are the variables which occur in some relation in the subtree at n, but have not been marked as

quantified.

Definition The ***early quantification problem*** is to find an early quantification tree, given a set of relations to be multiplied, and a set of variables to be quantified, such that the maximum support over all nodes in minimized.

Note that we try to minimize the maximum support over all partial products. This is an approximation to minimizing the BDD with maximum nodes in the process. Although this cost function is crude, in practice it works reasonably well, and it is very easy to compute.

The exact complexity of the problem remains unknown, although it is easy to prove that the problem is in NP. Though we have not proved it to be NP-complete, we suspect it is.

2.2 Overview of the Loc-Opt Algorithm

Definition A ***valid subtree*** is a quantification tree where the leaves are a subset of relations in $T_1, ..., T_n$, the variables being quantified are a subset of $v_1, ..., v_m$, and the quantified variables appear only in the relations in the subtree. Intuitively, a valid subtree can be extended to a full quantification tree.

Definition Assume two disjoint valid subtrees are given, where no relation appears in both. By ***merging*** two subtrees, we mean a new subtree whose children are the two given subtrees, and all variables which appear only in the two subtrees are marked as quantified. Let n be the node obtained by merging subtrees l with subtree r. Then, the ***cost of the merge*** is $support(n) - (MAX(support(l), support(r)))$, which is intuitively the local increase in the supports after the merge. For example, if there are 5 variables in the left subtree's support, and 8 variables, in the right one, and the support after the merge is 10, then the cost of the merge is 2.

The algorithm is defined as follows:

1. *Start with the set of all relations, each in individual subtrees. Mark as quantified those variables which appear in only one relation. Set done to false.*
2. *Until done is true, do the following*
   If only one subtree, set done to true.
   else
   - *2.a Consider a subset of merges between subtrees.*
   - *2.b Choose the minimum cost merge whose support is least.*
   - *2.c If the cost of the merge is less than some user's maximum cost, do the merge. Otherwise, set done to true.*

Note that the above algorithm builds the quantification tree from already built subtrees. It is a local optimization algorithm, since it chooses the merge which (heuristically) minimizes the increase in the size of a product of two partial products. We allow some hill-climbing moves by using a one-step look-ahead.

Definition ***Three way merges*** merge three subtrees (by building a subtree on three leaves). The cost of a three way merge is the difference in the support of the new node minus the maximum of the supports of the three subtrees.

In choosing the best next merge, both two and three way merges can be considered in the above algorithm.

2.3 Data Structures

To efficiently implement the algorithm, we need some data structures.

Definition Given a forest of subtrees, ***active variables*** are those which must be quantified but have not been quantified yet. The ***connections of an active variable*** are all subtrees in which it appears. Note that, after step 1 of the above algorithm, each active variable has at least two connections.

Definition A ***variable connection table*** (defined for active variables) is a hash table which contains for each variable the set of all subtrees in which the variable appears. Note that insertions and deletions into a variable connection table can be done in constant time. With every variable, we keep the number of connections it has. Hence, checking the number of connections of a variable can be done in constant time. This table is always kept up to date, i.e. after every merge it is updated.

Lemma 1 The size of the variable connection table is bounded by $O(mn)$.

Proof For each variable, there can be at most n connections. The lemma follows (QED).

Definition A ***merge*** consists of a type (two way or three way), all the subtrees involved in the merge, the support after merge, and the cost of the merge. Each merge points to its next links in the bucket (to be defined below).

Remark Every subtree is assigned a unique identifier (starting from 0). In the following discussion, we sometimes refer to subtrees as ***nodes***.

Lemma 2 A merge can be built in $O(m)$ time, where m is the number of variables.

Proof It suffices to show that the support and quantified variables can be determined in $O(m)$ time. For every subtree, we keep the list of variables in the subtree in sorted order. To find the new support after a merge, scan the set of supports in increasing order. For every variable, if the variable has two or three connections and all its connections are in the merge, quantify the variable. Note checking the number of connections of a variable can be done in constant time. Otherwise, leave it in the support. These operations can be done in $O(m)$ time (QED).

Definition A ***merge table*** consists of two ***node arrays***, corresponding to two way and three way merges respectively. Each subtree has a location in each node array (based on its id), which points to a hash table (called the ***node's merge list***), containing the set of merges the subtree is involved in.

Lemma 3 Insertion or deletion of a merge in the merge table can be done in constant time.

Proof Denote the merge in question by M. Based on the type of M, select the corresponding node array. For each node in M, add or delete the merge in the node's hash table (QED).

Definition A ***cost array*** is an array with a position for each cost. Each location C in the cost array points to a ***bucket item***, which has three pieces of information:

1. A ***support array***, which is an array of merges having a bin for each possible support size. All merges in a bin are linked

together. For example, all merges of cost C with support 5 are in the same bin.

2. A low and a high mark pointer, which say what the lowest and highest possible supports are in the current cost category C.

Definition A ***bucket*** consists of two (cost array, position) pairs, corresponding to two way and three way merges. Each position points to the merge with smallest cost in each cost array.

Lemma 4 Insertion of a merge in a bucket can be done in constant time.

Proof Denote the merge in question by M. Based on the type of M, get the corresponding cost array. Based on the cost of the merge, find the bucket item. Based on the support of M, find the corresponding bin. Insert the merge at the beginning of the bin. Adjust the low and high marks (QED).

In practice, each support array has a fixed size, let's say 200. Hence, all merges whose support is more than 200 will be in the same bin.

Lemma 5 Deletion of a merge from a bucket can be done in constant time.

Proof Since the items in a bin are maintained using a link list, where the pointers are maintained inside the merges, deletion from a bin can be done in constant time. We need to adjust the low and high mark pointers. Since the support arrays have constant size, this takes constant time (QED).

The high mark pointer is only used to signify empty bucket items. For example, assume there is only one merge of cost 5. Assume the support of the merge is 10. If this merge is deleted, its bin becomes empty. Since, the high mark pointer is set at 10, we do not need to march up the support arrays to find that it is empty.

2.4 Algorithm's Details

The detail of step 2 of algorithm of section 2.3 are as follows.

1. *Create small merge table.*
2. *Perform merges with some small maximum cost.*
3. *Create the full merge table.*
4. *Perform merges with large maximum cost.*

Each step of the algorithm is described below.

Creating small merge table. The purpose behind creating the small merge table is to do the easy merges on a reasonable subset of possible merges. For example, in some examples (in our formal verification tool), the algorithm is called with thousands of relations. In such cases, it is important that the algorithm works efficiently. In practice, this step prunes the number of subtrees substantially. The algorithm is as follows.

1. *For active variables with only two connections, create a merge between the two subtrees in which the variable appears.*
2. *For active variables with only three connections, create a three way merge between all the subtrees in which the variable appears.*
3. *For all other variables, do nothing.*

In practice, when the number of relations is large, there are many variables which are used in just two relations; since we have a subset of all possible merges, we will perform "safe" merges, i.e. those with low cost.

Lemma 6 The number of merges created by the small merge table algorithm is bounded by $O(m)$.

Proof Clear by definition of the algorithm.

Performing the merges. We assume some maximum cost is given. In our implementation, for the small merge table it is 0, whereas it is ∞ for the large merge table. The algorithm is as follows.

1. *Choose the best merge, i.e. one with least cost, and least support.*
2. *If the cost is not greater than maximum cost, perform the merge.*

To perform the merge, we proceed as follows.

1. *Create a new node. Delete the old nodes involved in the merge from the merge table.*
2. *Consider the merge lists of the nodes involved in the merge. Update all the merges i.e.*
   a. *Delete those merges which were among the nodes in the merge,*
   b. *update the merges between a node not in the merge and nodes in the merge, to involve the new node in the merge.*

Creating the full merging table. The following algorithm is used.

1. *For variables having connections less than some threshold value, create all pairwise merges among the relations which use the variable.*
2. *For variables having connections more than some threshold, arrange the relations in the connection list in a circular order. Create a link from every relation to its neighbor.*

The threshold value we use in our implementation is 50.

Lemma 7 The number of merges created by the full merge table algorithm is bounded by $O(m)$.

Proof Clear by description of the algorithm (QED).

2.5 Complexity of Algorithm.

We first analyze the worst case complexities, and then the complexity for sparse structures. We assume that the total number of variables in support of relations is $O(m)$.

Definition Let the ***connection table*** T be the table, which indicates the variables used in the support of each relation.

Lemma 8 The space consumption of the algorithm is bounded by $max(O(m^2), O(mn))$.

Proof This follows by lemmas 1, 6, 7 and by the fact that the space consumption of a merge is $O(m)$ (QED).

Lemma 9 Updating the variable connection table after a merge can be done in $O(m)$ time.

Proof A merge can involve variables. For every variable, the variable connection table has to be updated. Each update takes constant time. The results follows (QED).

Lemma 10 The running time of the algorithm is bounded by $max(O(n^2m), O(m^2))$.

Proof By lemmas 6 and 7, there are at most $O(m)$ merges created initially. By lemmas 1 and 9 creating the small and full merge tables takes at most $O(m^2)$ time. To construct the quantification tree, $O(n)$ merges are needed. After every merge, at

most $O(n)$ merges must be deleted and created, where each deletion or insertion by lemma 9 takes $O(m)$ time. Hence, the total time for merging and updating is bounded by $O(n^2m)$. The result follows (QED).

We believe the above lemma is very pessimistic, and the experimental data supports this. In practice, if the number of relations is large, then the connection table is very "sparse", i.e. every variable is used in a constant number of relations, and every relation has a constant number of variables in it. Note that for sparse connection tables $O(n) = O(m)$. A more meaningful theorem is the following.

Theorem 11 If the connection table is "sparse", then the running time (and hence space consumption) of the algorithm is $O(n)$.

Proof The difference with the above proof in the analysis is that updating after every merge can be done in constant time, since every subtree has a constant number of merges, and creating a merge takes constant time. Since, there are $O(n)$ merges to be done, the linear bound follows (QED).

Note that having efficient data structures is a must to get this bound.

2.6 Constant Propagation

In practice, one can take advantage of some structural information about the input to speed up the algorithm. We have noticed one technique known as constant propagation works well in practice. A ***constant BDD*** is one whose supports includes only one variable, and the number of minterms it represents is 1. An example is the BDD for relation $x = 5$. Our constant propagation algorithm is described below.

1. Put all constants in a constant list.
2. Until the constant list is not empty
2.a Pick a constant from the list.
2.b Multiply the constant everywhere it is used. Quantify the variable in the support of the constant BDD from the result.

3 Exact algorithm for linear ordering of terms

In this section we describe the Ex-Lin-Ord algorithm. We assume that we are given a linear ordering of the terms.

3.1 The algorithm

Definition Let $cost[i,j] = \text{cost}\left(\prod_{l=i}^{j} T_l\right)$. Intuitively $cost[i,j]$ is the cost of realizing the product of $T_i, \ldots, T_j$ quantifying out all variables which can be quantified. The algorithm is based on dynamic programming and goes as follows:

1. As a preprocessing step quantify out any variables present in only one relation.
2. Initialize the cost function of the leaves: initialize $cost[i,i]$.
3. For $l = 2, \ldots, n$
do for $i = 1$ *to* $n+l-1$
$cost[i,j] = \infty$
for $k = i$ *to* $j-1$
$q = compute - cost$
if $q < cost[i,j]$
then $cost[i,j] = q$
$seperator[i,j] = k$
return cost and separator

As a preprocessing step we quantify out any variables present in the support of only one function. The separator data structure keeps the index of the right most term in the left partial product for the optimum realization of the partial product $T_i, \ldots, T_j$.

We calculate the best way to realize partial products of increasing sizes. In step 2, we calculate the cost of the original terms. Then we calculate the best way to realize partial products of size 2, i.e. involving two neighboring terms of the linear order, and so on. The cost of realizing a larger partial product is calculated as the best way to compute it by using the optimum way of computing subproblems.

3.2 Complexity

By examining the loop structure of the algorithm we get the following lemma:

Lemma 12 The algorithm Ex-Lin-Ord runs in time $O(n^3m)$, where n is the number of relations and m is the number of variables to be quantified.

Proof The algorithm computes the value of $cost[i,j]$ for $i = 1, \ldots, n-1$ and $j \geq i$. There are $O(n^2)$ such calculations. To compute $cost[i,j]$ we have to compute the cost of merging partial products $i, \ldots, j$ and $j, \ldots, k$ for $i \leq k \leq j$. There are $O(n^2)$ such calculations. The cost of each merge can be computed in $O(m)$ time. The result follows (QED).

Remark The space consumption of the algorithm is $O(n^2m)$

3.3 Optimality

Theorem 13 Given a linear ordering, Ex-Lin-Ord determines the optimum schedule for the given cost function.

Proof The proof is by induction on the size of the partial products, i.e the number of primitive terms in a partial product. The base case holds since there is only one way of realizing a partial product. Assume the cost function is such that the cost of realizing the current partial product can be calculated from the cost of realizing smaller partial products optimally. Thus if we try all possible ways of realizing the current partial product we are guaranteed to find the optimum. The algorithm of section 3.1 tries all possible ways of realizing a partial product from sub-partial products subject to the linear order restriction (QED).

3.4 Cost Functions

We have experimented with two cost functions. The first one is that of the support set. The cost of a partial product p, realized as a product of l and r is $MAX(\text{support}(p), \text{support}(l), \text{support}(r))$. This cost function approximates our objective of minimizing the largest BDD seen through the computation.

The other cost function, the *early quantification* cost function minimizes the cost of quantifying the variables that have been quantified from the current partial product or its component par-

tial products. Define $|p|$ to be the number of *ands* needed to form partial product p starting from the original relations. Thus $cost(p) = \sum_{v} (|p| - 1) + cost(l) + cost(r)$, where p is realized as a product of partial products l and r. The summation runs over the variables that are spread across l and r, i.e those that can be quantified from p but not from either l and r.

Remark Both of these cost functions are such that the best way of realizing a partial product is by combining smaller partial products realized optimally.

3.5 Ordering the terms

Remark Corresponding to every quantification tree there exists a linear ordering of the terms such that the tree can be realized for that ordering by Ex-Lin-Ord.

Thus the optimum solution can be found by picking a suitable linear order and running Ex-Lin-Ord. This also serves as an easy way to see that the Early Quantification problem is in NP.

4 Experiments with Early Quantification

Since our package is used primarily in our verification tool, to test the package we have chosen a set of examples from verification. Some of these are from industry, while others are academic.

The input language to our system is Verilog, extended to handle non-determinism. Many high-level designs are described as a set of interacting non-deterministic FSM's. Early quantification comes up in two places:

1. As each FSM is translated into our intermediate format, BLIF-MV, many small non-deterministic relation are created (one can think of them as non-deterministic gates). To form the transition relation for the FSM, all the relations have to be multiplied, and intermediate variables quantified out. Since the relations are non-deterministic, i.e. they are not functions, the recursive techniques which work for combinational logic cannot be applied (these techniques amount to computing the function for each gate in terms of functions of its inputs). In this application, it is not uncommon to require early quantification with thousands of relations.

2. After the transition relation for each FSM is built, all these relation must be multiplied and non-state variables quantified out. In this application, the input usually consists of less than 100 relations.

The next two sections summarize our experience within each of these applications. For ease of programming, our initial implementation sometimes differs somewhat from the algorithms presented. So, the run times given may be a bit pessimistic.

4.1 Building Transition Relation of a Process

We quote results from our experiments with building the transition relations of FSM's written in the BLIF-MV format. While some of these examples have more than the few machines quoted here, we have the largest amongst them in the table. Gigamax is a multiprocessor cache coherency protocol distributed with the tool SMV[McM93]. Scheduler is from [Mil89]. 2mdlc is a message data link controller obtained from industry.

Column b gives the number of relations in the call to our package. The number after the / is the number of relations after constant propagation. Column (c) is likewise is the number of variables being quantified out. Column (d) gives the sum of the times (in seconds) to compute the schedule and perform the conjunction and existential quantification. In column (g), L stands for Loc-Opt, T for [TSLBS90], and B for [Bur91]. All our experiments were performed on a DECsystem 5900 with 440MB of physical memory.

Our experimental observations are as follows. The support set seems to track well with the size of the BDD's. In the vast majority of our experiments the algorithms of both [TSLBS90] and [Bur91] perform far worse in time and minimizing the largest BDD sizes, than Loc-Opt. In many examples, constant propagation (CP) significantly reduced the number of relations and variables to be quantified. The results quoted in the column (d) include the times for CP. For one FSM in 2mdlc (on the last call) we were unable to build the transition relation using the T and B methods but finished in about 12 seconds with the largest BDD observed having only 1644 nodes using Loc-Opt. In the other calls T was on average about 5 times slower with BDD's sometimes an order of magnitude larger. B was always slower and on average produced larger BDD's.

Table 1: Experimental data for transition reln. construction

Example (a)	No. of Rel/ after CP (b)	No. of vars/ after CP (c)	Time in secs. (d)	largest bdd seen (nodes) (e)	Max Support Seen (f)	Method (L,T,B) (g)
diners	122/80	128/86	0.1	40	12	L
			0.08	145	22	T
			0.12	205	26	B
gigamax	153/100	148/95	0.29	202	14	L
			0.89	3370	30	T
			3.63	3014	33	B
scheduler	106/69	104/67	0.25	253	12	L
			0.40	1074	23	T
			0.61	502	22	B
2mdlc	212/144	187/119	1.50	8982	37	L
			2.60	11261	42	T
			40.10	23731	49	B
2mdlc	820/556	809/545	3.02	274	24	L
			20.10	62468	46	T
			177.5	62468	46	B
2mdlc	2194/1388	2179/1373	11.58	1644	38	L
				Space	out	T
				Space	out	B

4.2 Building the Product Machine

The first step in the verification of a system of interacting

machines is the formation of the product machine. For the product machine construction we also tried out the Ex-Lin-Ord algorithm. Though the Ex-Lin-Ord algorithm has a cubic dependency on the number of relations, the schedule can be computed in less than a second for 1-30 relations. On larger instances on some examples the extra time in computing the schedule could some times be justified by savings in performing the operations.

ES stands for Ex-Lin-Ord with the support cost function and EE, for using the *early quantification* cost function of section 3.4. For ES we used the linear order corresponding to the quantification tree obtained with Loc-Opt, and for EE a greedy algorithm for a [Bur91] like scheme. Column (e) indicates the cost of the final solution. In the case of T and B methods it is the maximum support for their respective schedules.

The Ex-Lin-Ord algorithm always returns a better a priori cost than Loc-Opt. However, Scheduler is an example where the largest bdd observed was not smaller than with Loc-Opt, indicating that the support abstraction of size does not always work. However, it is our experience that it serves well in a vast majority of cases.

For the product machine application both [TSLBS90] and [Bur91] were slower and resulted in larger BDD's, sometimes several orders of magnitude. For example in Scheduler, both T and B spaced out. Using the Ex-Lin-Ord algorithm on the linear order returned by Loc-Opt produces some optimization in most cases.

We comment on other applications of Early Quantification and in particular Ex-Lin-Ord in section 7.

Table 2: Product machine construction

Example (a)	No. of machines (a)	No. of vars (c)	Max bdd seen (no. of nodes) (d)	Cost (e)	Time in secs. (f) schedule/ and-quantify	Method
Giga max	11	22	8507	31	2.36	L
			5693	21	0.04/1.48	ES
			4256	127	0.01/2.77	EE
			15042	31	6.42	T
			176163	36	6.87	B
2mdlc	12	39	49999	194	20.29	L
			49999	166	0.06/21.03	ES
			49999	143	0.01/27.47	EE
			102691	194	44.0	T
			102691	194	95.61	B
Scheduler	20	18	2393	40	0.32	L
			2739	40	0.12/4.22	ES
			2718	120	0. 02/2.86	EE
				Space	out	T
				Space	out	B

5 Partial Product Minimization

In this section, we assume we are building the product machine of a set of interacting FSM's. Even when early quantification is used for this purpose, the partial products can grow large. In the same spirit as [CSSB92], we pose the problem of minimizing the BDD's for partial products as finding don't cares. Our algorithm can be viewed as follows.

Given an early quantification tree, recursively multiply relations and quantify variables according to the schedule given by the tree. In current BDD packages, multiplication and quantification can be done in one step. We will refer to this procedure as ***and-quantify***.

Assume at some point, the children of a partial product grow too large. At that point, apply a minimizations technique to gather some (independent) don't care information for each child. The don't care information on the children can be used during the and-quantify at the current node to minimize the resulting BDD. Currently, no procedure is available for such minimization. So, as an approximation, the two children are minimized with respect to their individual don't cares, and the and-quantify is applied.

The techniques to find don't cares can be divided into three kinds: minimization with respect to reachable set, simulation equivalence and bisimulation state minimization, and trace equivalence state minimization. We describe our techniques in an environment based on language containment, with no fairness constraints. We will then comment on how our techniques can be extended to handle fairness constraints and CTL model checking. [HSBK93] describes the basics of our verification environment.

Using language containment for verification, the system is described as a set of automata. A set of fairness constraints are placed on the automata to remove extra behavior introduced by abstraction. The property is given as an automaton with fairness constraints. The property is complemented, and it is checked whether the intersection of the language of all automata (including the complement of the property automaton) is empty. To check for emptiness, the product machine is built, using early quantification. The techniques described here show how partial products of this computation can be minimized.

5.1 Reachability Based Minimization

Each partial product corresponds to the product of transition relations of a set of machines. The set of initial states corresponding to these automata can be computed. Assume the set of reachable states of this partial product is given. A set of don't cares can be obtained at this step, which are all the edges which originate from the unreachable set. The partial product can be minimized with respect to this set.

5.2 State Minimization Using Bisimulation and Simulation Equivalence

So far in the literature, bisimulation has been mostly used for verification purposes. To prove a system correct, a specification of the property is given in terms of an automaton. It is then

proved that the implementation is bisimilar to its specification. We feel that this form of verification is too restrictive (we share this view with [Mcm93]). Following [BFH92] & [FKM93], we propose to use bisimulation and simulation equivalence as state minimization techniques.[1] We will describe our algorithms below, which are simpler than those of [FKM93], since they keep track of partitions explicitly.

Definition Let a finite automaton A be given, with $L(A)$ denoting its language. Two states x and y are ***trace equivalent*** iff $L(B) = L(C)$, where B and C are exactly the same as A except the first has x as its initial state and the second y.

Bisimulation and simulation equivalence are stronger than trace equivalence, i.e. two states x and y may be trace equivalent but not bisimulation equivalent. We will define these notions below.

Definition Let an equivalence relation $E_{i-1}(x, y)$ be given. The ***one-step simulation relation*** of $E_{i-1}(x, y)$, denoted by $E_i(x, y)$, is defined as follows:

$E_i(x, y) = \forall\alpha\forall z(T(x, \alpha, z) \rightarrow \exists w T(y, \alpha, w) \wedge E_{i-1}(z, w))$,

which is read as y one step simulates x if whenever x can make a move to z on some action α, then there exists a state w such that y can make a transition to w on α, and $(z, w) \in E_{i-1}$.

Definition State a is said to ***simulate*** state b if $(a, b) \in E_S(x, y)$, where E_S is defined as the greatest fixpoint of the following computation:

1. *Start with all states related to all other states, i.e.* $E_0(x, y) = 1$.
2. *Apply one step simulations until convergence.*

Remark We usually define E_0 to only relate the reachable states to each other. This can be significant for convergence, since the unreachable states may have poor convergence with respect to simulation relation.

Definition An equivalence relation $E(x, y)$ is said to be ***symmetric*** if whenever $(x, y) \in E$, $(y, x) \in E$. By making $E(x, y)$, we mean the relation $E(x, y) \wedge E(y, x)$, where the pair $(x, y) \in E$ is removed if $(y, x) \notin E$.

Definition Two states x and y are ***simulation equivalent*** if x simulates y and y simulates x. To compute the set of simulation equivalent states, $E_E(x, y)$, use the following algorithm.

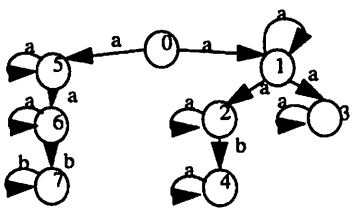
1. *Compute* $E_S(x, y)$ *as in the definition of simulation.*
2. *Make* $E_S(x, y)$ *symmetric.*

Definition State a is said to be ***bisimilar*** to b if $(a, b) \in E_B S(x, y)$, where E_B is defined as the greatest fixpoint of the following computation:

1. *Start with all states related to all other states, i.e.* $E_0(x, y) = 1$.
2. *Apply the following until convergence,*
   a. *Apply one step simulation to* $E_{i-1}(x, y)$ *in order to get* $E_i(x, y)$.
   b. *Make* $E_i(x, y)$ *symmetric.*

Note that $E_B(x, y)$ is symmetric. Note also that the only difference between algorithms for bisimulation and simulation equivalence is that for bisimulation the equivalence relation is made symmetric after each step, whereas for simulation equivalence it is made symmetric after the simulation relation has converged.

The difference between simulation equivalence and bisimulation can be subtle. The following example reveals this difference. Using bisimulation, we get 6 states, with states 1 and 7, and 2 and 6, being equivalent. Using simulation equivalence, we get 5 states, with state 1 being equivalent to 5, in addition to the other equivalences.



This example shows the difference between simulation equivalence and bisimulation.

[HTKB92] defined a complexity hierarchy for BDD computations. Basically, the complexity of a fixpoint computation is dependent on the arity of the relation which is being iterated on. For example, the arity of the iterated relation in reachable set computation is 1, whereas for transitive closure it is 2. In this sense, the above computation are equivalent to transitive closure, since we are iterating on $E(x, y)$, which is a binary relation.

5.3 Don't Cares Arising from State Minimization

The information about equivalent states can lead to two different sets of don't cares: the internal and external don't cares. Let $E(x, y)$ denote the set of equivalent states. The ***internal don't care set*** is obtained by adding transitions into equivalent states. The following formula computes this set: $\overline{T(x, \alpha, y)} \wedge \exists z \exists w(E(x, z) \wedge E(y, w) \wedge T(z, \alpha, w))$. Intuitively, if a transition $T(z, \alpha, w)$ exists in the original machine, the transition $T(x, \alpha, y)$ is a don't care if it is not a transition in the original machine, and x is equivalent to z, and y to w.

The ***compatible projection operator*** ([LN91]) is an operator which chooses a representative for each equivalence class. The ***external don't cares*** can be obtained as follows.

1. *Run the compatible projection operator.*
2. *Build the state minimized machine, called the quotient machine.*
3. *Minimize with respect to the unreachable set of the quotient machine.*

Note that the unreachable set of the quotient machine includes some reachable states of the original machine. Using the external don't cares has the advantage that the reachable state set changes. This can be crucial, if the reachable state set cannot be

1. Since we assume a synchronous concurrency model, weak bisimulation does not make sense.

built.

5.4 Approximations to Trace Equivalence

Using trace equivalence, one can possibly get more state minimization, compared to minimization using bisimulation or simulation equivalence. However, state minimization for non-deterministic automata is PSPACE-complete. Even if we start with deterministic automata, as we quantify out variables, the automata become non-deterministic. Here, we present two approximations to trace equivalence. These approximations are not conservative, i.e. two states may be equivalent according to these approximations, which are not actually trace equivalent. Such approximation have at least the following uses.

1. They can be used as abstractions. The minimization information using these approximations can be used to add behavior. One way to ensure this is by using internal don't care set. In a language containment based environment, if the check passes (i.e. the language of the product machine is empty), we are assured that the original check has passed. On the other hand, if it fails (in the same spirit as [BS93]), we can check the error trace. If the error trace is an error trace in the un-minimized system (for this check we don't need the product machine), then we have a counter-example. If not, we have to do some refinements based on some strategy.

2. These approximations can also be used as lower bounds on how well simulation equivalence and bisimulation perform as conservative approximations to language containment. We feel that this is an important contribution, since using current techniques (which check whether two states are equivalent by determinizing the machine) lower bounds on automata with only a few hundred states could be obtained. This is so, even when BDD's are used, since for determinization, usually one bit per state is created. Current BDD techniques can handle BDD's with only several hundred binary variables. Using our technique, we may be able to handle automata with several thousand or even more states.

Definition State x_0 is ***output simulated*** by y_0 if for all sequences of states $x_0, x_1, ..., x_n$, where $T(x_{n-1}, \alpha, x_n)$, there exists a sequence of states $y_0, y_1, ..., y_n$, such that $T(y_{n-1}, \alpha, y_n)$. Intuitively, if state x_0 can do action α in n steps, then y_0 can do action α in n steps.

Definition Two states are ***output simulation equivalent*** if each output simulates the other.

To compute output equivalent states, the following algorithm can be used.

1. Let $T_0(x, i, y) = T(x, i, y)$, $E_0(x, y) = 1$.

2. Repeat until convergence:

2.a $E_{i+1}(x, y) = E_i(x, y) \wedge \forall \alpha (T_i(x, \alpha) \leftrightarrow T_i(y, \alpha))$.

2.b $T_{i+1}(x, \alpha, y) = \exists z \exists \beta (T_i(x, \beta, z) \wedge T(z, \alpha, y))$.

$T(x, \alpha)$ represents a transition relation where the y variables have been quantified out. Note that (x, α, y) is an edge in $T_{i+1}(x, \alpha, y)$ if there is a path of length n between x and y whose last label is α. The BDD complexity of the above algorithm is the same as computing the transitive closure.

Lemma 14 If two states are not output simulation equivalent, then they are not trace equivalent.

Proof This basically follows from the definition: if two states x and y are not output simulation equivalent, then there is a path of length n, let's say from x, which is labeled with α on the n-th step. We cannot have any path from y which has the label α on its n-th edge. The result follows.

Definition Two states x and y are ***n-trace equivalent***, iff for all paths of length n from x with labels $\alpha_1, ..., \alpha_n$, there exists a path of length n from y with exactly the same labels, and vice versa.

Lemma If two states are not n-trace equivalent for some n, then they are not trace equivalent.

Proof Follows by definition.

The following algorithm can be used to find the set of states which are n-trace equivalent.

1. Let $E_1(x, y) = 1$, $T_1(x, i_1, y) = T(x, i, y)$.

2. For $j = 2, ..., n$, *do the following:*

2.a $T_j(x, i_1, ..., i_j, y) = \exists z T(x, i_1, ..., i_{j-1}, z) \wedge T(z, i_j, y)$.

2.b Let $I_j = (i_1, ..., i_j)$.

2.c Let $T_j(x, I_j, y) = T_j(x, i_1, ..., i_j, y)$.

2.d Perform one step simulation relation on $E_j(x, y)$ *and* $T_j(x, I_j, y)$ *with the selection variables being* I_j.

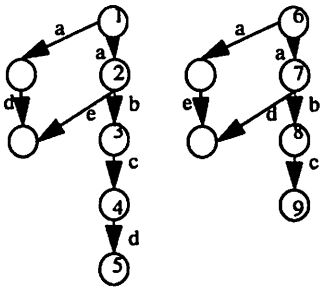
2e. Make $E_j(x, y)$ *symmetric.*

Intuitively, this algorithm builds relations $T_j(x, i_1, ..., i_j, y)$, where two states x and y are related iff there is a path of length n between them in $T(x, i, y)$, remembering all the labels on the path between x and y.

Lemma 15 The above algorithm computes the n-trace equivalent states.

Proof Assume two states x and y are n-trace equivalent. Let $j \leq n$. Then, if x has a transition to z by a path which assigns a set of values to I_j, then y has a path of length j to some state w with the same assignment to I_j. States x and y will remain in the same equivalence class for $j \leq n$. Now assume that x and y are not n-trace equivalent. Then, there exists a path of length j which distinguishes them. In the j-th step, x and y will be separated by the one step simulation relation (QED Lemma).

Unfortunately, in the above algorithm, if $E(x, y)$ converges, we cannot be sure that we have reached trace equivalence. The following is an example where in iteration 3 nothing changes, but we have not reached trace equivalence. The following theorem puts a bound on how many iterations are needed.



This example shows that the n-trace algorithm can converge, where there are still unequivalent states. After iteration 2, 1=6, 2 !=7, 3!=8, 4!=9. Iteration 3 does not change anything.

Theorem 16 If two states are n^2-trace equivalent, then they are trace equivalent.

Proof Consider two states x_0 and y_0. Consider the product machine of $T(x,y)$ by itself, where the initial state is (x_0, y_0). Assume the two states are not trace equivalent. Without loss of generality assume, there is a trace from x_0 which y_0 cannot produce. Then, there is a path in the product machine of length $k \leq n^2$ to some product state (x_k, y_k) such that x_k has a transition on some symbol α, but y_k does not. Moreover, for all reachable states $(\tilde{x}_k, \tilde{y}_k)$ in exactly k steps, with the same labels as the path between (x_0, y_0) and (x_k, y_k), $\tilde{y}_k$ does not have a transition on α. The transition relation $T_k(x, l_k, y)$ will expose this difference, and hence x_0 and y_0 will not be in the same equivalence class after the k-th step. By the algorithm, once two states get into two different equivalence classes, they can never get into the same class in the subsequent iterations. The theorem follows.

5.5 Minimization and CTL Model Checking

From among the minimization techniques that we presented, reachability minimization and bisimulation preserve all CTL properties ([BCG87]). Simulation equivalence perseveres ACTL formulas (GL91]), which are CTL formulas involving universal quantifiers only. Trace equivalence minimization cannot in general be used with CTL model checking. We will consider how bisimulation minimization can be done in the CTL model checking paradigm. The case of simulation equivalence is similar.

Assume a CTL formula is given. If the CTL formula involves only output variables, then the minimization can be done as before. The output variables denote a set of states of the original machine. This set has to be modified to reflect the state spaces of the quotient machine. If the formula involves state variables as well, then the initial equivalence relation has to be built so that only those states are equivalent whose state values for the state variables used in the CTL formula is the same.

5.6 Minimization and Fairness Constraints

We assume we are given a set of fairness constraints, which must all hold. One can then extend reachability minimization to minimize with respect to the set of ***fair reachable states***, which are those states that can reach some fair cycle, i.e. a cycle satisfying all fairness constraints. When language containment is used, if the set of fair reachable states of a partial product is empty, then the language of the product machine is empty, and the language containment check has passed.

Now, assume we are dealing with a state minimization technique. We account for fairness constraints in the initial step of our algorithms, when the initial equivalence relation is built. For simplicity, assume the fairness constraints are given to us in Buchi form. Then, the equivalence classes are initialized so that all final states are in the same class, and all non-final states are in the same class. Note that in general, a final state can be equivalent to a non-final state. Hence, our approximation is conservative.

Remark Using the language containment paradigm, safety properties can be specified with a Buchi automaton with all states final except for one, which acts as a non-accepting sink state. Usually, in proving safety properties, no fairness constraints on the system is needed. In this case, our algorithms give the maximum optimizations. On the other hand if the safety property was specified using a CTL formula, since there are no fairness constraints, maximum optimization is possible. We think of our algorithms as being most suitable in this context. Effective state minimization in the presence of fairness constraints remains a challenge with the work of [DHW91] being a starting point.

6 Initial Experiments with Partial Product Minimization

We have just begun experimenting with partial product minimization. In this section, we describe the results of our initial experience.

1. We have experimented with various minimization techniques on a few examples. In only one example, bisimulation and simulation equivalence gave different results. Simulation equivalence and n-trace equivalence, where the n-trace equivalence algorithm is stopped after convergence is achieved, always return the same answer. This is rather interesting since it implies that simulation equivalence may be a good approximation to trace equivalence. Output simulation equivalence in general returns a different answer than other methods. We believe output simulation equivalence can be used as an abstraction technique: it is not so naive so that all of the structure is removed, and it is not too restrictive so that some details is hidden.

2. Automation of when to apply what minimization technique remains a challenge. Simple strategies, such as apply some minimization technique after the number of BDD nodes grows beyond some threshold bound, do not yield good results. For example, carelessly applying state minimization can lead to long run-times. Currently, we have made a minimization environment which is interactive, where the user chooses what minimization method to apply next.

3. The notion of minimization after partial collapsing appears to be very powerful. For example, consider Milner's [Mil89] scheduler, which is ring of processes $P_1, ..., P_n$, and a set of processes $Q_1, ..., Q_n$, where Q_i communicates only with P_i. One can visualize this as a ring of processes with each process in the ring having another process hanging from it.

We built the product machine of the system and computed the reachable states. The number of reachable states in the rings without any minimization was 608655. The quantification tree we get back from early quantification first merges each Q_i into its corresponding P_i, and then multiplies these partial products by going around the ring. We did bisimulation minimization after collapsing of each Q_i into its corresponding P_i. The number of reachable states was reduced to 4288. The CPU time for the operation increased from 12 seconds to 22 seconds. This experiment suggests that state minimization can be used after local collapsing to reduce the number of reachable states with little cost. In cases, where the reachable set cannot be built, this technique may be vital.

7 Conclusion and Future Work

In this paper, we have presented two algorithms for the early quantification problem. Our first algorithm is fast and produces very good results. The other one can be used as an optimizer to improve the quality of the first one. We are examining the uses of our algorithms, especially Ex-Lin-Ord, in the case that we cannot form the product machine using the techniques presented here. We have presented experimental results for our algorithms. Even when early quantification is used, the sizes of the partial products can grow large. We have presented several techniques for reducing the size of partial products. We have done some initial experimentation with these techniques, which was presented here. An algorithm which controls when each minimization should be applied remains an open research question.

References

[BS93] F. Balarin and A. Sangiovanni-Vincentelli, "An iterative approach to Language Containment", Computer-Aided Verification, 1993.

[BFH92] A. Bouajjani, J. C. Fernandez, and N. Halbwachs, "*Minimal Model Generation*", Computer-Aided Verification, 1992.

[Bur91] J. R. Burch, E.C.Clarke, and D.E.Long, "Symbolic Model Checking with Partitioned Transition Relations", in Proceedings of the 28th ACM/IEEE DAC, pp. 403-407, June 1991.

[CSSB92] M. Chiodo, T. Shiple, A. Sangiovanni-Vincentelli, and R. Brayton, "*Automatic Reduction in CTL Compositional Model Checking*", Workshop on Computer-Aided Verification, 1992.

[DHW91] David L. Dill, Alan J. Hu, Howard Wong-Toi, "*Checking for language inclusion using simulation pre-orders*", Workshop on Computer-Aided Verification, 1991.

[FKM93] J. C. Fernandez, A. Kerbrat, L. Mounier, "*Symbolic Equivalence Checking*", Computer-Aided Verification, 1993.

[GL91] O. Grumberg, D. Long, "*Model Checking and Modular Verification*", CONCUR 91, Theories of Concurrency, Unification and Extension, LNCS 527.

[HTKB92] Ramin Hojati, Herve Touati, Robert P. Kurshan, Robert K. Brayton, "*Efficient ω-Regular Language Containment*", Computer-Aided Verification, 1992.

[HSBK93] R. Hojati, Thomas P. Shiple, R. K. Brayton, R. P. Kurshan, "A *Unified Approach to Language Containment and Fair CTL Model Checking*", Proceedings of Design Automation Conference, June 1993.

[LN91] B. Lin, A. R. Newton, "*Efficient symbolic manipulation of equivalence relations and classes*", IMEC-IFIP International Workshop on Formal Methods in VLSI Design, Miami, 1991.

[McM93] K.L.MacMillan, "Symbolic Model Checking", Kluwer Academic Publishers, 1993.

[Mil89] R. Milner, "Communication and Concurrency", Prentice Hall, New York 1989.

[TSLBS90] H.J.Touati, H.Savoj, B.Lin, R.Brayton, and A. Sangiovanni-Vincentelli, "Implicit State Enumeration of Finite State Machines using BDDs", in Proceedings of the ICCAD 1990.