Research Proposal:
Efficient and Effective Symbolic Model Checking

Subramanian Iyer

October 26, 2003

1 Introduction

In this proposal, we briefly outline some of the major challenges facing the adoption of Formal Verification techniques. We focus on the model checking approach, which is completely automated in principle and quite automated in practice. These challenges include handling the state explosion problem associated with large industrial designs, which manifests itself as large representation sizes and being able to reconcile verification with debugging.

Large State Spaces: As is well known, the main challenge in model checking for design verification is what is termed as the "state explosion problem" – given a design, the state space that it encompasses is often exponential in the size of the design description. Traditionally, the state explosion problem has been handled close to the design level, using for example abstraction, symmetry reduction, compositional reasoning, etc. These approaches have been shown to produce significant gains in many cases. Their main drawback is that the user needs to discover the applicability of these techniques on almost a case by case basis; hence, they cannot be easily automated.

It is therefore vital to handle large state spaces automatically in a manner that is transparent to the designer. The earliest approach to model checking [6] emphasized an enumerative implementation, while a symbolic technique using BDDs [3] was suggested by McMillan [15]. Symbolic model checking can exhaustively cover the state space when handling small designs but cannot handle industrial sized designs, which can often be orders of magnitude larger. The main issue is that current symbolic data structures quickly grow quite large, thereby often not fitting in main memory and being cumbersome to perform operations upon at such large sizes. One solution is functional partitioning as proposed by Jain, et.al.[13], and extended by Narayan, et.al.[17]. More recently satisfiability-based model checkinghas been proposed which can detect shallow bugs in large designs and sometimes quicker, often at the cost of selective coverage of the state space.

While symbolic model checking and bounded model checking have extended the domain of applicability of formal verification techniques to larger designs than was originally feasible, they fall considerably short of what would be considered as being "production sized" in the electronic design automation community.

Verification vs. Debugging: Another important issue is that of debugging, sometimes also referred to as falsification, as opposed to verification. While this may theoretically appear to be a trivial complementation issue, it is of paramount importance in practical verification. We consider the reasons.

Historically, designs have been checked using simulation and test techniques. These techniques can run for a unlimited number of instances, as each simulation can be considered to be independent of others. On the other hand, formal methods like model checking proceed one step at a time. These quickly produce rather large data structures, and are unable to progress any further. Consequently, on large designs, simulation based techniques can often be run for much longer.

With the increasing complexity and size of designs, the verification problem of certifying the correctness of the entire design only gets harder. Simultaneously, the cost of an undetected error can escalate. Simulation-based techniques are inherently incomplete, in the sense of not being exhaustive. Sophisticated statistical analysis techniques are used in test/simulation but design errors still slip by.

In short, simulation-based techniques scale to large designs but are not exhaustive in their coverage of the state space whereas formal methods can be exhaustive but can only handle smaller designs. In practice, scalability is perceived to be of a greater importance than exhaustive coverage. This is especially so because "time to market" considerations often dictate the need to locate and fix bugs as soon as possible. A certain threshold of error is considered acceptable, even unavoidable.

Verification techniques are increasingly being integrated into the design flow to complement simulation based tests. This creates an even greater need for formal verification techniques to be focused toward rapid falsification in order to actively and meaningfully interact with the modified design process – design, verify, find bug, fix bug via redesign, repeat. This has led to a much greater focus in practice on finding and fixing errors in designs rather than proving their correctness.

Accordingly, the focus of this work is on the use of formal verification, specifically symbolic model checking, to detect errors quickly, especially on large practical designs while retaining the ability to verify design correctness, as the situation demands.

2 Preliminaries

2.1 Temporal Logics

Temporal logic provides the simple but basic temporal operators Xp (next p), Fp (sometime p), Gp (always p), pUq (p until q) that can be easily combined in order to specify many interesting temporal properties. The restriction to formulae along single paths generates what is known as the *Propositional Linear Temporal Logic*. More generally, use of the existential and universal path quantifiers, E and A resp., generates the branching time temporal logic *CTL**. The *Computation Tree Logic*, CTL, is a special subset that pairs uniquely each temporal operator with exactly one quantifier. The propositional μ-calculus subsumes all the above mentioned temporal logics, and can be thought of as a unifying framework.

It should be noted that it is possible to express any CTL formula in terms of the Boolean connectives of propositional logic and the existential temporal operators EX, EU and EG. Such a representation is called the *existential normal form*.

We omit further details of the syntax and semantics of temporal logics as they are widely known and readily available in the literature. The interested reader is referred to [8].

2.2 Symbolic Reachability and Model Checking

We assume a Kripke structure $M = (S, T, L)$, where S is the set of states, the relation $T \subseteq S \times S$, and the labeling function L defined as $L(s) = s, \forall s \in S$. When working with BDDs, we further have $S = \mathbb{B}^n$ where $\mathbb{B} = \{0, 1\}$. So each state $s \in S$ is a bit string $\overrightarrow{b} = (b_1, b_2, \ldots b_n)$. A set of states P can be associate with formula p, described over boolean variables $\overrightarrow{s} = (s_1, s_2, \ldots s_n)$, such that $s \in P$ if and only if $\overrightarrow{b} \models p(\overrightarrow{s})$. State variables $\overrightarrow{s}$ are said to describe the *current* state. Analogously we can define *next* state variables $(s'_1, s'_2, \ldots s'_n)$ corresponding to state s', represented by the bit-string $b' = (b'_1, b'_2, \ldots b'_n)$. Then we can say that a transition $s \rightarrow s' \in T$ if and only if $(b_1, b_2, \ldots b_n, b'_1, b'_2, \ldots b'_n) \models T(s_1, s_2, \ldots s_n, s'_1, s'_2, \ldots s'_n)$. For the sake of brevity, we shall just write this as $(\overrightarrow{b}, \overrightarrow{b'}) \models T(\overrightarrow{s}, \overrightarrow{s'})$. Where clear from context, we use the symbol for a set of states to also stand for the propositional formula representing it. Similarly, the state variables will also represent the variables of the formula.

The standard reachability algorithm is based on a fixpoint computation which performs a breadth-first traversal of finite-state structures [7, 15, 18]. The algorithm takes as inputs the set of initial states, $I(\overrightarrow{s})$, expressed in terms of the present state variables, $\overrightarrow{s}$, and the transition relation, $T(\overrightarrow{s}, \overrightarrow{s'})$. For sequential designs, T is obtained as the conjunction of the transition relations, $s'_k = f_k(\overrightarrow{s}, i)$, of the individual state elements, i.e., $T(\overrightarrow{s}, \overrightarrow{s'}) = \prod_k (s'_k \equiv f_k(\overrightarrow{s}))$. Given a set of states, $R(\overrightarrow{s})$, that the system can reach, the set of next states is defined as $N(\overrightarrow{s'}) = \exists \overrightarrow{s} [T(\overrightarrow{s}, \overrightarrow{s'}) \wedge R(\overrightarrow{s})]$. This calculation of N is also known as *image computation*. The set of reached states is computed by adding $N(\overrightarrow{s})$, obtained by substituting variables $\overrightarrow{s}$ for $\overrightarrow{s'}$, to $R(\overrightarrow{s})$ and iteratively performing this image computation step until a fixpoint is reached.

In practice, Model Checking is usually performed in two stages: In the first stage, the finite state machine that represents the transition relation is reduced with respect to the formula being model checked and the reachable states are computed. The second stage involves computing the set of states falsifying the given formula. In this step, the reachable states computed earlier are used as a substitute for the entire state space. Thus the model checking step has to deal with only the reachable states, which is a smaller – often substantially smaller – fraction of the state space. It should be noted that model checking is performed usually in the backward direction, involving the computation of pre-images.

Since there exist computational procedures for efficiently performing Boolean operations on symbolic BDD data structures, including POBDDs, model checking of CTL formulas primarily is concerned with the symbolic application of the temporal operators. EXq is a backward image and uses the same machinery as image computation during reachability, with the adjustment for the direction. $E(pUq)$ (resp. EGp) has been traditionally represented as the least (resp. greatest) fixpoint of the operator $\tau(Z) = q \vee (p \wedge EXZ)$ (resp. $\tau(Z) = p \wedge EXZ$) and can therefore be computed as a fixpoint.

2.3 Partitioned-ROBDDs

The idea of partitioning was used to discuss a function representation scheme called partitioned-ROBDDs in [13, 12] which was extensively developed in [17].

Definition. [**17**] Given a Boolean function $f : B^n \rightarrow B$, defined over n inputs $X_n = \{x_1, \ldots, x_n\}$, the partitioned-ROBDD (henceforth, POBDD) representation χ_f of f is a set of k function pairs, $\chi_f = \{(w_1, f_1), \ldots, (w_k, f_k)\}$ where, $w_i : B^n \rightarrow B$ and $f_i : B^n \rightarrow B$, are also defined over X_n and satisfy the following conditions:
1. w_i and f_i are ROBDDs respecting the variable ordering π_i, for $1 \leq i \leq k$.
2. $w_1 \vee w_2 \vee \ldots \vee w_k = 1$
3. $w_i \wedge w_j = 0$, for $i \neq j$
4. $f_i = w_i \wedge f$, for $1 \leq i \leq k$ The set $\{w_1, \ldots, w_k\}$ is denoted by W. Each w_i is called a *window function* and represents a *partition* of the Boolean space over which f is defined. Each partition is represented separately as an ROBDDs and can have a different variable order. Most ROBDD based algorithms can be adapted easily for POBDDs.

Partitioned-ROBDDs are canonical and various Boolean operations can be efficiently performed on them just like ROBDDs. In addition, they can be exponentially more compact than ROBDDs for certain classes of functions. The practical utility of this representation is also demonstrated by constructing ROBDDs for the outputs of combinational circuits [17]. An excellent comparison of the computational power of various BDD based representations and POBDDs may be found in [2].

2.3.1 Creating Windows for Partitions

An approach involving partitioning clearly depends heavily on the criteria used for creating the partitions, i.e., for selecting the "windows".

A static algorithm is presented in [16] to obtain window functions when the number of partitions has been determined a priori. These window functions, $w(s)$'s, are cubes on the present state variables. The algorithm assigns a cost to each variable and selects the best $log_2 k$ variables (for k partitions) for partitioning. From these $log_2 k$ variables k partitions are created which correspond to all the binary assignments of these variables. The goal is to create small and balanced partitions. The cost of partitioning a transition relation $T(s, s', i)$ on variable s as $cost_s(T) = \alpha[p_s(T)] + \beta[r_s(T)]$ where $p_s(T)$ represents the partitioning factor and is given by, $p_s(T) = max(|T_s|, |T_{\overline{s}}|)/|T|$ and $r_s(T)$ represents the redundancy factor and is given by, $r_s(T) = (|T_s| + |T_{\overline{s}}|)/|T|$ Here, T_s and $T_{\overline{s}}$ represent the positive and the negative cofactors of T with respect to s respectively. Notice that a lower partitioning factor is good as it implies that the worst of the two partitions is small and similarly a lower redundancy factor is good since it implies that the total work involved in creating the two partitions is less.

It must be ensured that the functions being partitioned do indeed depend upon the variables selected, in the sense that the valuation of the function changes depending upon the valuation of the variable. The above approach performs such an analysis heuristically by requiring a low redundancy factor. Notice that if T does not depend on s, then T_s and $T_{\overline{s}}$ would be identical, and therefore the redundancy factor would equal 1, and therefore such a variable s would not be selected for partitioning.

2.3.2 Reachability using POBDDs

Since the state space S is disjunctively partitioned into n disjoint subspaces $S_1 \dots S_n$, any subset $s \subseteq S$ can be considered an implicit disjunction of corresponding subsets of the subspaces, $s_1 \subseteq S_1, \dots s_n \subseteq S_n$. This induces a partition of the transition relation $T \subseteq S \times S$ into n^2 pieces, denoted T_{jl}, where any T_{jl} may be considered as transiting from a state in partition j to a state in partition l. One can derive the transition relation T_{jl} by conjoining T with the respective window functions expressed appropriately in terms of present and next state variables, as $T_{jl}(s, s') = w_j(s)w_l(s')T(s, s')$.

The POBDD based traversal algorithm uses the ROBDD based algorithm in its inner loop to perform fixed point on individual partitions. Let us assume that we are given a partitioned representation of the set of reachable states $\chi_R = \{(w_j(s), R_j) | 1 \leq j \leq k\}$. If we take the image of the projection in the jth partition R_j under the projection of the transition relation T_{jj} from partition j into itself, we obtain the set of next states $N_j(s') = \exists s[w_j(s)w_j(s')T(s, s')R_j(s)]$ represented in terms of the next state variables. Since $w_j(s')$ is independent of the variables that are to be quantified, it can be taken out of existential quantification, giving us $N_j(s') = w_j(s')[\ \exists_s[w_j(s)T(s, s')R_j(s)]\ ]$

The image of R_j under T_{jj} lies completely within partition j. Similarly, the image, N_l of R_j under T_{jl} will lie completely within partition l. This observation motivates us to define the image computation in terms of the image computed within the same partition and the image *communicated* to another partition. The former will be called *ImgPart* and the latter will be called as *ImgComm*.

3 Proposed Research

We begin with an examination of the issues that confront the adoption of model checking for the verification of large designs. The primary issue is that the individual BDDs that represent sets of states often grow so large that boolean operations and image computation can no longer be performed efficiently on them. To counter this explosion of sizes, we propose a decompositional approach as opposed to the classical "monolithic" model checking. We present an algorithm for model checking CTL formulae that is designed to break up the model checking problem into partitions that are handled independently of each other.

Notice that decompositional model checking performed with a fixed number of such partitions, determined a priori, essentially encounters the same problems as classical symbolic model checking, namely of the BDDs growing too large *in each partition*. Therefore, we propose a new data structure, namely "dynamically partitioned BDD". This improves the statically partitioned OBDD data structure by introducing a technique to vary the number of partitions in the representation of state sets. This change is dynamic in nature as the need arises depending on the blowup of BDD sizes during the computation. Further more, this exploits the result of Wegener et.al. [2] that partitioned OBDDs with $k + 1$ partitions can be exponentially more succinct than ones with k partitions.

The key issue in any decompositional approach is the determination of how the partitions are created. Notice that partitions created as a set of minterm cubes can be assembled together as a Free BDD, the succinctness of which is the same as classical ROBDDs. Therefore, there is a need for effective partitioning in order to leverage the advantage in succinctness offered by the limited non-determinism available in POBDDs. We propose the study of such partitioning techniques.

The advantages proffered by such an approach, while considerable, can be further enhanced by parallelization and we could explore this in greater detail. From a practical standpoint, it is found that bugs in designs are often found only after the circuit is unrolled a number of times, which roughly corresponds to a notion of *depth* of the state space. It is an interesting question as to whether this intuitive notion of deep state space exploration can be put to practical use.

Some of these ideas are now described in greater detail. First we describe the decompositional model checking approach and the dynamically partitioned BDD data structure, both of which are in an advanced stage of completion. Then we move on to the issue of time scalability in verification, and to achieving more compact function representations by exploiting the non-determinism in partitioned BDDs using generalized window functions. Finally, we consider the problem of debugging "deeply" located errors in industrial designs.

3.1 Decompositional CTL Model Checking

In this section, we propose a technique to perform model checking in a decompositional fashion and break the computation into partitions that can be handled largely independently of each other.

Formulae of Computation Tree Logic can be expressed in terms of the Boolean connectives of propositional logic and the existential temporal operators EX, EU and EG. The operation EXp computes the pre-image of the set p. Classically, this pre-image is treated as the atomic operation in model checking and the other operations are defined in terms of EX. In the decompositional approach, this operator can be split into two operations - one of which is local to each partition and is therefore relatively inexpensive whereas the other operation transcends the boundaries of partitions and is consequently more expensive. This is analyzed in the next section. Then, we present the refined model checking algorithms.

3.1.1 Computing the pre-image EXp

We define the backward image, i.e. pre-image, as comprised of the computations *preImgPart* which computes the pre-image restricted to a specific partition and *preImgComm* which computes the pre-image across partitions. This is illustrated in the pseudo-code of Fig 1. The pre-image, i.e., *computeEX*, is then obtained by their union, as

$$\mathit{computeEX}(p) := \bigvee_i \mathit{preImgPart}(p_i, i) \vee \mathit{preImgComm}(p) \qquad (1)$$

Notice that two approaches are possible for the computation of the communicated image: In the first, an image is computed from partition j into each partition $k \neq j$ separately, using the transition relation T_{jk}. Alternately, one can compute the image from partition j into the boolean space that is the complement of partition j, denoted by $\overline{j}$. The former has the advantage that the BDD representations of the transition relations T_{jk} are much smaller, but in return it has to perform $O(n^2)$ image computations. We use the second method in defining *imgComm*. This method requires only $O(n)$ image computations, but each of these is followed by $O(n)$ restrict operations. Thus this method is seen to enjoy a linear gain over the naive approach in terms of number of image computation operations performed.

Further, we claim that this procedure for image computation (and pre-image computation) has comparable linear gains over the naive approach. Let us adopt the following model of computation. Each partition is handled by one process. The BDDs belonging to a partition are owned by that

```
preImgPart($Bdd$, j) {
    return $preImage(Bdd, T_{jj})$
}

preImgComm(S){
    $result := \emptyset$
    foreach (partition j)
        $temp := preImage(S_j, T_{j\bar{j}})$
        foreach (partition $k \neq j$)
            $temp_k := temp$ restricted to $w_k$
            reorder BDD $temp_k$ from partition order $j$ to order $k$
            $result_k := result_k \vee temp_k$
        end for
    end for
    return $result$
}

computeEX($p$) {
    $R := p$
    forall (partitions $j$)
        $S_j := preImgPart(R_j, j)$
    end for
    $S := S \vee preImgComm(R)$
    output $S$
}
```

Figure 1: pre-image Computation Algorithm

process. The transition relation from partition i to partition j is kept with process i, consequently image computations from partition i to partition j need to be performed by process i and the state set has to be transmitted to the process j.

Notice that multiple processes can be assigned to a single processor. Indeed our current implementation uses only one processor for all processes. In fact, a naive implementation is to serialize using a First Come First Served scheduling order.

If an extra process is dedicated to be "a communication cache", the image computation can be done using O(n) messages transmitted between processes, rather then the $O(n^2)$ messages required for the naive approach at the expense of performing some boolean operations when storing or retrieving data from the cache.

Communication is expensive

It is important to notice that there are fundamental differences between the two image operations - *preImgPart* and *preImgComm*. Observe that preImgPart(R_j) is in the same partition j as the original BDD R_j and therefore only one partition needs to be in memory for its computation. On the other hand, preImgComm(R_j) computes an image into $\overline{j}$, i.e., *every partition other than* j, therefore it needs to finally access and modify every partition. This gives rise to two important issues with respect to communication.

Firstly, the reached state set of every partition needs to be accessed. In the case of large designs, where the BDDs of even a single partition can run into millions of nodes, this usually means accessing stored partitions from secondary memory.

Secondly, the BDD variable order of the computed image set must be changed from the order of the j^{th} partition to that of each of its target partitions, before the new states can be added to the reached set in the target. Again, for large designs, reordering a large BDD can be an extremely expensive operation.

In this context, pre-image computation within a partition, *preImgPart*, is a relatively inexpensive operation as compared to communication between partitions, *preImgComm*. Therefore, in the interest of minimizing transfer of BDDs from one partition to another, we need to decrease the number of invocations of preImgComm when possible.

An associated advantage of performing pre-image computation repeatedly within a partition before communicating, is that it allows some errors to be caught much earlier. When a formula fails in any partition, it becomes unnecessary to explore the other partitions any further. In this manner, it may be possible to locate the error by exploring a smaller fraction of the state space than otherwise necessary.

Algorithms were proposed by Narayan, et. al.[16] to address this issue in the context of forward reachability. In the rest of this section, we will present, in the context of partitioning, the improved model checking algorithm designed to reduce inter-partition communication.

3.1.2 Evaluating the Least Fixpoint $E(pUq)$

The classical algorithm for the least fixpoint operator is presented in Figure 2 in a partitioned form.

Notice that in the computation of $E(pUq)$, the pre-image computation forms the bulk of the work performed by the algorithm. As noted before,

```
computeEU(p, q) {
    $S := q$ and $S.old := \phi$
    repeat
        $S.old := S$
        $S := q \vee (p \wedge computeEX(S))$
    until($S = S.old$)
    output $S$
}
```

Figure 2: Classical Algorithm for $E(pUq)$

the cost of performing communication during every pre-image is quite large in terms of the resources required to transfer BDDs between partitions, to reorder the BDDs before such transfer can occur and to fetch the partitions from storage in order that the new states can be conjuncted with p and disjuncted with q. Therefore, it is important to *postpone the call to preImg-Comm*, i.e., to perform as many image computations as possible locally within each partition before communication is performed.

A New Algorithm for $E(pUq)$

We now describe a new algorithm for model checking least fixpoint CTL formulas and sketch a proof of its correctness. Figure 3 for computing the

```
computeEU(p, q) {
    $S := q$ and $S.old := \phi$
    repeat
        $S.old := S$
        forall (partitions $j$)
            repeat
                $S_j.old := S_j$
                $S_j := S_j \vee (p_j \wedge preImgPart(S_j, j))$
            until($S_j = S_j.old$)
        end for
        $S := S \vee (p \wedge preImgComm(S))$
    until($S = S.old$)
    output $S$
}
```

Figure 3: New Algorithm for $E(pUq)$

set of states satisfying $E(pUq)$ is designed to take advantage of the partitioned nature of the data structure. Notice that we explore each partition independently of the others until they reach a fixpoint individually. Then, we perform the communication across partitions. This allows us to keep just one partition in memory at any given time. It also greatly reduces the number of communication induced BDD transfers, disk accesses and variable reordering calls.

Before proving the correctness of the new algorithm, we define some notation. Let the set of states S at the end of the k^{th} iteration of the outermost repeat-until loop in algorithm 3 be represented by S^k.

For every state $s \models E(pUq)$, either $s \models q$ or there exists a sequence of states $s_0, s_1, \ldots, s_k$ that has the smallest length $k \neq 0$ such that $s_0 = s$, $s_k \models q$, $\forall i < k : s_i \models p$ and $\forall i < k : s_i \in preImage(s_{i+1})$. Such a sequence of states is called a *witness* for the inclusion of s in $E(pUq)$, and k is its *length*. For the sake of convenience, we will use the symbol for a formula to also mean the set of states it represents. It should be clear from context as to which is meant.

We first show that algorithm 3 terminates.

Lemma 3.1 *(Termination) For any integer i, $S^{i+1} \supseteq S^i$. The inequality is strict unless a fixpoint is reached.*

The proof is evident from the construction of sets S^k. Since any step of the procedure must add at least one new state to the set S, we have termination at the end of at most as many iterations as there states in the space under consideration.

Theorem 3.2 *The procedure computeEU of algorithm 3, given the set of states corresponding to formulas p and q as inputs, terminates with the output S being precisely the set of states that model the formula $E(pUq)$.*

Proof: Soundness: We prove by induction on the sets S^k that the procedure is sound, i.e., at all times $S \models E(pUq)$. This clearly holds for any state in the initial set $S^0 = q$, since any state satisfying q also satisfies $E(pUq)$.

Assume, it holds for S^i, i.e., that $S^i \models E(pUq)$. Consider a state $s \in S^{i+1} - S^i$. Then, by construction of S^{i+1} from S^i, we have $s \models p$. Either s is added during some step of the inner fixpoint loop or it is added in a step of communication, i.e., $s \in preImgComm(S^i)$.

Suppose s is added in the inner fixpoint loop of some partition j. Since S^i is a POBDD, let us call the projection of S^i in partition j as S^i_j. From (1), for all j, $preImgPart(S^i_j, j) \subseteq preImage(S^i)$. Also notice that the variable for the inner fixpoint is initialized to S^i_j. Therefore, every state added in the first step of the inner fixpoint models $p \wedge EX(E(pUq))$ and therefore models $E(pUq)$. Consequently, we can show by induction that any state added in the inner fixpoint loop for partition j must model $E(pUq)$.

In the second case, s was added in some step of the communication. Considering that $preImgComm(S^i) \subseteq preImage(S^i)$, any state added in the communication step models $p \wedge EX(E(pUq))$, and therefore $E(pUq)$. In particular, $s \models E(pUq)$.

Consequently, $S^{i+1} - S^i \models E(pUq)$ and the soundness of the procedure follows by induction.

Completeness: We next show the completeness, i.e., that every state of $E(pUq)$ is indeed in set S. Let T^k be the set of states whose inclusion in $E(pUq)$ is witnessed by a path of length at most k. We prove by induction on k that $T^k \subseteq S$. In the base case, this trivially holds because $T^0 = q = S^0 \subseteq S$.

Now, let us assume that $T^i \subseteq S$. For any state $s \in T^{i+1}$ consider the sequence of states $s_0 = s, s_1, \ldots, s_{i+1}$ that witnesses its inclusion in $E(pUq)$. We will show that $s \in S$.

Now, the sequence $s_1, \ldots, s_{i+1}$ is a witness for s_1, therefore $s_1 \in T^i \subseteq S$. In particular, there exists a smallest j so that $s_1 \in S^j$. We know that $s \models p$ and $s \in preImage(s_1) \subseteq preImage(S^j)$. From the definition of S^j and Algorithm 3, we have that

$$\begin{aligned} S^{j+1} &\supseteq S^j \vee (p \wedge preImgPart(S^j)) \vee (p \wedge preImgComm(S^j) \\ &= S^j \vee (p \wedge (preImgPart(S^j) \vee preImgComm(S^j))) \\ &= S^j \vee (p \wedge (preImage(S^j))). \end{aligned}$$

Therefore, $s \in S^{j+1} \subseteq S$, whereby $T^{i+1} \subseteq S$. By induction, this gives us $E(pUq) \subseteq S$.

Together with lemma 3.1, this proves that algorithm 3 terminates with the set $S = E(pUq)$.

□

This work on the least fixpoint operator has been completed and was presented at IWLS 2003[**?**]. We propose to extend this idea to handle all of CTL.

3.1.3 Evaluating the Greatest Fixpoint *EGp*

The model checking of *EGp* is done by computation of the greatest fixpoint of the operator $\tau(Z) = p \wedge EXZ$. As in the case of least fixpoint, one would like to postpone the communication until after each partition has reached its individual fixpoint independent of the other partitions.

We will provide a new algorithm for the same.

3.2 Improved state space traversal

Having described a technique to perform model checking in a piecewise "decompositional" manner, we will now describe a data structure to perform this effectively. Partitioned Ordered Binary Decision Diagrams (POBDDs) described before serve as the starting point for our approach. However, the partitioning scheme proposed and presented there uses a fixed number of state space partitions, which are determined by the user before the computations are begun entirely based on the initial size of the transition relation.

We posit that this is insufficient for such a partitioning scheme based on an a priori selection of th number of partitions faces the sames obstacles as an approach based on ROBDDs - the data structure sizes eventually get large enough as to become unwieldy.

Consequently, we propose a dynamic partitioning scheme where the number of partitions can be increased or decreased as the computation progresses. This can be shown to be exponentially more succinct than the use of a fixed constant number of partitions. Such "dynamically partitioned OBDDs" can serve as a good data structure for handling designs much larger than what can be handled using ROBDDs or the statically partitioned OBDDs of Narayan, et.al.

Dynamic Partitioning: Dynamic repartitioning of the state space is triggered whenever the size of any partition under observation crosses a certain threshold. The partitioning variables are selected using the history of previously computed windows. Repartitioning is performed by splitting the given partition by cofactoring the entire state space based on one or more splitting variables until the blowup has been ameliorated in each partition created so far. Initially, the partitioning is done using one splitting variable. The choice of this variable is as explained before. At this point, each new partition is checked to see whether the blowup has subsided. If not, repartitioning is called again on that partition until the blowup has subsided in each partition created.

Sometimes it is found that the blowup in the BDD-sizes during an intermediate step of image computation is a temporary phenomenon which eventually subsides by the time the image computation is completed. In such a case the invocation of dynamic global repartitioning of the state space could create a large number of partitions, whose BDD-sizes become eventually very small. These partitions create an unnecessary amount of computational overhead. Hence, it is advantageous to create these partitions *locally* only for that particular image computation and then recombine them before the end of the image computation. To create these local partitions, we can cofactor the state space using the ordered list of splitting variables that was generated earlier.

Our algorithm for checking invariants performs successive steps of image computation on each R_j under T_{jj}. Since these steps, *imgPart*, of image computation add states only within the same partition, and since different partitions are disjoint, we are guaranteed that the same state is not being visited multiple times within different partitions. Once a fixpoint is reached within a partition j, the procedure *imgComm* is used to communicate the new set of states to the partition l for for $1 \leq l \leq k$ and $l \neq j$. At any stage, where new states are added into the reached states set, we check for the violation of the invariant presented. If failure is detected, we stop and call the error trace mechanism to retrieve a path from the initial states to an error state. Otherwise, we proceed with traversing more states until the entire state space is exhausted, at which point, the formula has passed.

Tracing Erroneous Paths: The idea behind the storage and retrieval of computation paths from a state violating the property back to an initial state, also known as an *Error Trace*, is now described.

To obtain a path from an error state e back to an initial state i, the naive idea would be to compute successive pre-images beginning with e, until i is reached. After a few steps of computing backward images, one would be faced again with a rapidly increasing BDD size. In order to avoid this blowup in BDD-size, we need to be able to isolate a set of candidate predecessors for the current state so that the next pre-image computation does not have have to handle too large BDDs. In the case of ROBDDs, this is accomplished by keeping the so called "onion rings" or the frontier of states encountered during each image computation.

Novel data structure for tracing errors with POBDDs: In the partitioned setting, the set of possible predecessors may be spread across multiple partitions. Thus it is possible to store these frontier states in a partitioned manner. Therefore the backward image can be computed with respect to only a portion of the frontier states.

So, the image computations need to be recorded in a tree-like data structure in order to be able to find the correct subspace for the backward image. For each state s in the set of reachable states S, this tree contains the image computation when the state s was first added to the reachable set S. The structure stores the information required to trace a backward path as follows:

For each partition of the boolean space, its *frontier* is defined as the states added to this partition by the most recent invocation of imgComm and the subsequent imgPart operations. Each such frontier is actually a collection of sets, each represented as a BDD, whose set union represents the set of all states that have been reached in this partitions for the first time, but have not yet been used for communication to other partitions. Thus, the number of BDDs in this frontier can be, in the worst case $O(M + d_i)$ where M is the number of partitions, and d_i is the depth of the fix-point in partition i. For the entire graph this can, in the worst case be, $O(M * (M + d_{max}))$.

To retrieve a path from an initial state to a state s, we do the following:
1. Obtain the location in the computation tree that contains s.
2. Take the predecessor frontier of this location in the tree, and compute a backward image into this frontier to find one or more predecessor states.
3. Pick one such predecessor state.
4. Repeat steps 2 and 3 on successive states until an initial state is reached.

This gives us the path from state s with an error to an initial state.
Advantages of partitioned error trace: Notice that in the case of ROBDDs, the frontier states can get large in size. An effect of having these large sized representations is that image computations get more expensive. As noted before, ignoring the frontier states and performing a backward reachability is even more expensive, and in that case the backward path can be longer in length too.

Observe that partitions can often be asymmetric with respect to the space and time required for performing image computations on them. Therefore, in the presence of multiple paths from an error state to the initial states, it would be advantageous to compute the shortest path in terms of computational effort rather than the length of the path. In order to do this, we annotate the nodes of the tree with information about the amount of time the corresponding image computation required. These annotations can be used as an indicator of how much time the backward image would take, and thus, in step 3 above, they can assist in reducing the time spent in finding a more practical path back to the initial states.

Status: This work has already largely been completedand was presented at CHARME 2003. The key idea is to extend the partitioning model to allow for on-the-fly repartitioning. This makes practical use of the result of Bollig and Wegener[2], that a $(k + 1)$-POBDD can be exponentially more succinct that a k-POBDD.

3.3 Time Scalability in Verification

One of the major disadvantages of using extant OBDD-based formal verification methods – besides memory explosion – is their lack of time scalability, i.e. that at the end of an assigned time for computation one often achieves no result at all, regardless of whether or not the design is correct. This is because the data structures – BDDs – may grow so large that the tool either thrashes, or sometimes even crashes.

We present algorithms based on partitioning to achieve *time scalability* in formal verification, so that as the time allocated for the total computation increases, so does the fraction of the design state space explored.

These algorithms find errors in designs faster and traverse the state space more efficiently than OBDDs and other known Partitioned-OBDD approaches. They tackle the core problems in practical adoption of Partitioned-OBDDs, namely choice and scheduling of partitions.

What is missing in the classical approach? For performing operations on many functions, ROBDDs suffice. In such cases, esp on small sized representations, they are sometimes more efficient than the partitioning based approaches. If we accept the premise that the function representations typically analyzed are too large for efficient monolithic representation as a single graph, such as an ROBDD, they should benefit by partitioning. In this context, certain problems arise naturally and have not been addressed effectively in the literature. For example, it is natural to ask:

1. When should a function be broken into disjoint subspaces?

2. How many subspaces should be created? Which subspaces of an exponential number of possibilities should be generated?

3. Further, we perform operations on the representations that are created. What if the results of these operations are very simple? Should we then combine a subset of such simple representations into a single graph? How and when should this be performed?

4. Finally, since partitioning generates multiple independent representation and operations can be performed on these largely independently, what is a good order of required computations?

We posit that above questions are fundamental to creating any practical partition handling algorithm. We wish to address questions like the above and expect that an efficient solution to these problems would lead to vastly improved practical results in the ability to handle large designs.

3.4 Generalized Windows and Multiply rooted BDDs

It is to be noted that the partitioning schemes in the literature are all based upon the use of windows that are defined as minterm cubes on present state variables. Clearly a set of windows constructed in this fashion can be combined into a tree such that each leaf of this tree represents one partition, and each path along the tree represents a unique window. Under these circumstances, such a partitioned BDD can be treated as a special case of Free BDDs, where all subtrees rooted beyond a certain depth are disallowed from sharing variables. It is well-known that as the number of variables increases, the succinctness of Free BDDs approaches that of regular ROBDDs in the asymptotic case.

In other words, the non-deterministic succinctness afforded by the partitioned BDD data structure is effectively lost when one uses cube based partitioning. It is consequently plausible that more compact representations may be generated by the use of non-cube windows for partitioning.

We would like to analyze the problems associated with state space exploration and generate compact partitioned representations. Clearly, this requires the selection of good window functions.

The current partitioning approach is based on minterm cubes, i.e., each window can be thought of as the conjunction of literals. We propose a shift from this model to one based on generalized boolean functions. Prior work in the area of combinational verification [14] shows that substantial gains may be achieved. We study this in the context of sequential verification.

The complexity of the "compose" operator is cubic for ROBDDs and quadratic for Partitioned BDDs. Therefore, an exponential gain can be obtained on multiple sequential nested compositions by the use of partitioning. This was shown by Jain, et.al. [14]. The details are provided in an **Appendix** of this document.

The discussion above shows the advantage of partitioning in performing multiple composition operations. We investigate a related question – can partitioning be defined in terms of composition points? This is expected to be greatly beneficial because, in practice, the construction of the Transition Relation for any given design as well as the image computation performs a sequence of nested functional compositions.

Reachability analysis is typically done by constructing a set of transition relations, and in conjuncting every member of the set. Then, all primary inputs and present state variables of the circuits are existentially quantified out from this formula. It is observed that the graphs typically blowup in size during this computation, especially during conjunction. Due to this blowup, BDD based formal verification is still not practical for large scale industrial designs. Often these procedures can be applied only on circuits with hundreds of latches. But industrial designs have tens of thousand latches, often even more. POBDDs can be used to make BDD sizes much smaller. However, all the partitioning techniques rely on splitting the circuit using cubes - these are assignments for a set of literals, either input variables or state space variables. A cube based partition can be also interpreted as being a special case of representation as a free BDD. However free BDDs and therefore such cube based partitions are known to be less compact then partitioning schemes where each window is generated using arbitrary functions, hence one would like to use arbitrary functions as windows. BDDs thus generated are multiply rooted BDDs since each window can have a different root variable. This is expected to make image computations and therefore reachability analysis much faster and more space efficient.

The central idea is as follows. We find splitting variables that are based upon decomposition points. Since decomposition points represent general functions (instead of cube like assignments of primary variables) so any partial assignment on such functions will also be a general function. We will like to thus process functions where the initial computation and model can be expressed in terms of decomposition and composition. The proposed procedure to build BDDs of transition relations is as follows:

1. Decomposition points are used to build BDDs of transition relations. In a practical verification tool, for e.g. VIS, all BDDs are built using decomposition points.

2. We compose the decomposition point variables until some composition blows up.

3. We create two disjunctive partitions for the given composition using standard BDD techniques and if each disjunction is lower then a predefined threshold then we can this partition to be the root of our non-cube based partitioning tree (NCPT).

4. For each leaf of the above defined NCPT, we recursively carry out steps 2 and 3 until all decomposition points have been composed and all the partitions are created.

5. Using each partition of TR, we carry a reachability analysis till a fixed point is reached.

The same partitioning as described above can be introduced even when actual conjunction/quantification are being done during image computation.

This above analysis procedures is essential to make use of the most compact form of POBDDs which are non-cube based, overlapping, POBDDs. Since good results are obtained even when using cube-based POBDDs, we expect greater savings when we utilize this generalized form of the partitioned data structure.

We would like to examine which computations of the reachability analysis can be written in terms of a sequence of composition operators. In the above, we have outlined the use of composition based generation of partitioning windows during the construction of the transition relation. It would be interesting to see if image computation can be thus expressed, because that would readily suggest a technique for dynamic repartitioning during image computation.

For practical circuits, where BDDs can become very large, any amount of space reduction is very welcome. Such multiply rooted data structures can offer exponential reduction in size over cube-based POBDDs.

3.5 "Deep" state space exploration

Satisfiability based Bounded Model Checking is able to explore the state space of larger designs by bounding the depth of exploration and successively increasing this bound. However it is difficult to control the direction of the search. In contrast, restricting the analysis to a subset of partitions provides a controlled way to perform deep, albeit partial, coverage.

It has been empirically observed that partitioning provides the means to symbolically explore the state space *deeply.* In fact, if each subspace is thought of as a "direction", then reachability with partitioning is a Breadth First Search along each such direction - in other words one can perform a hybrid of breadth first and depth first searches. It would be interesting to study whether the notions of state space depth and coverage can be formalized for a precise, quantitative analysis.

Perhaps partitioning can be used in conjunction with bounded model checking, maybe with ATPG-based model checking, which may be more suited to sequential verification than SAT-based bounded model checking.[1]

4 Comparison with Related Work

The conjunctive partitioning approach has been suggested for representation of the transition relation. Additionally, a technique for distributed model checking has been recently proposed based on disjunctive slicing.

We now compare our method with the above approaches.

4.1 Partitioned Transition Relations

The use of *partitioned transition relations* [4] was proposed to control the size of symbolic representation of transition relations. In this method, instead of using a monolithic ROBDD representation of the transition relation, the transition relations of different latches are kept as separate ROBDDs (or clustered into small groups of latches [**?**]). Since ROBDDs representing the individual latch transition relations are typically much smaller than when they are combined, this method can result in substantial memory savings. In addition, it allows for early quantification of variables which are not present in the support of other transition relations [11, 18]. This technique can also result in substantial savings in memory during image computation. Notice though, that the notion of 'partitioning' here is restricted to the building of the transition relation.

Cabodi, et.al. discuss a technique [5] in which the set of reachable states is decomposed into two or more sets during the intermediate stages of computation and reachability is performed on these decompositions separately. However, after a few steps of reachability, results from these different sets are typically combined to obtain a monolithic ROBDD representation of the reachable state set.

On the other hand, our goal is to construct a partitioned representation of not just the transition relation, but also the entire reachable state space as well as any other boolean functions that are created at all times, and to perform all operations on such functions on the corresponding partitioned representation. Thus, in order to distinguish the sense in which partitioning is performed, it would be more appropriate to call the former approach as *conjunctively clustered*-transition relations.

Indeed, the approach is complementary to our disjunctive partitioning approach, and we use these "clustered"-transition relations in the construction and use of transition relations.

4.1.1 Distributed Model Checking

Recently, a method for distributed model checking was studied by [10, 9]. It parallelizes the classical symbolic model checking algorithm using the partitioning approach suggested in [16]. This approach uses "slicing"[1], which is similar to partitioning, with the objective of doing model checking in a distributed fashion. This approach does not address issues related to costs of communication and variable ordering in different partitions. In particular,

[1]It may be noted that this is unrelated to the notion of program slicing.

this approach partitions the computation into a fixed number of fragments equal to the number of processors available in the distributed environment. However as noted in the literature [2], a partitioning scheme with k partitions can be exponentially more succinct than one with just $k-1$ partitions. Thus, the a priori selection of the number of fragments greatly limits the efficiency of the partitioned data structure. Indeed the gain from such a "static" method would be obtained substantially from parallelization rather than from the inherent algorithmic advantages offered by the POBDD data structure.

In contrast, our algorithms effectively capitalize on the partitioned nature of the data structure. We require only one partition to be in memory for any image computation, and each partition can be independently ordered. Significantly, this approach incorporates a dynamic repartitioning scheme which allows for an unbounded number of partitions to be automatically created when necessary. At the same time, we show how to drastically cut down the number of instances of inter-partition communications as compared to the classical approach. This reduces the number of transfers and re-orderings of large BDDs between partitions and is found to be a significant gain in practice. We also address the issue of efficient determination of error trace in the presence of partitioning.

5 Conclusion

The main bottleneck in practical BDD-based symbolic model checking is that it is restricted by the ability to efficiently represent and perform operations on sets of states. Symbolic representations like BDDs grow very large quickly due to their necessity to cover the state space in a breadth first fashion. Satisfiability based techniques result in a proliferation of clauses, since they have to replicate the transition relation numerous times.

We propose techniques to increase the capacity of automatic sate-based verification as applied to sequential designs, i.e., symbolic model checking. Firstly, we propose a decompositional approach to model checking, which splits the problem into multiple partitions handled independently of each other. Secondly, we propose the use of dynamically partitioned BDDs as a capable data structure. This leads to vast improvements in state space traversal in general and error detection in buggy designs, in particular.

Further, we consider the key issue of selecting good window functions for the partitioning approach. We would like to take advantage of the non-determinism afforded by POBDDs, and accordingly propose a study of windows based on general boolean functions. This is likely to shows its full benefit of potentially exponential savings in the context of operations expressible as a sequence of compositions.

We would like to address the issue of time scalability in verification, whereby the availability of larger amounts of computation time enables greater exploration of the state space. Finally, from a practical standpoint, we observe that extant verification approaches are unable to proceed very deep into the state space. It is our conjecture that partitioning can help in this context and we would like to explore this issue further.

References

[1] J. A. Abraham. Bounded Model Checking without SAT, *Private Communication.* 2003.

[2] B. Bollig and I. Wegener. Partitioned bdds vs. other bdd models. In *Proc. of the Intl. Workshop on Logic Synthesis*, 1997.

[3] R. E. Bryant. Graph based algorithms for Boolean function representation. *IEEE Transactions on Computers*, C-35:677–690, August 1986.

[4] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic Model Checking with Partitioned Transition Relations. In *Proc. of the Design Automation Conf.*, pages 403–407, June 1991.

[5] G. Cabodi, P. Camurati, and Stefano Quer. Improved reachability analysis of large finite state machines. *ICCAD*, pages 354–360, 1996.

[6] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. IBM Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[7] O. Coudert, C. Berthet, and J. C. Madre. Verification of Sequential Machines Based on Symbolic Execution. In *Proc. of the Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, 1989.

[8] E. A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, pages 995–1072. Elsevier Science, 1990.

[9] Orna Grumberg, Tamir Heyman, and Assaf Schuster. Distributed symbolic model checking for μ-calculus. In *Computer Aided Verification*, pages 350–362, 2001.

[10] Tamir Heyman, Daniel Geist, Orna Grumberg, and Assaf Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In *Computer Aided Verification*, pages 20–35, 2000.

[11] R. Hojati, S.C. Krishnan, and R. K. Brayton. Heuristic Algorithms for Early Quantification and Partial Product Minimization. Technical Report UCB/ERL M93/58, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, July 1993.

[12] J. Jain. On analysis of boolean functions. *Ph.D Dissertation, Dept. of Electrical and Computer Engineering, The University of Texas at Austin*, 1993.

[13] J. Jain, J. Bitner, D. S. Fussell, and J. A. Abraham. Functional partitioning for verification and related problems. *Brown/MIT VLSI Conference*, March 1992.

[14] J. Jain, K. Mohanram, D. Moundanos, I. Wegener, and Y. Lu. Analysis of composition complexity and how to obtain smaller canonical graphs. In *Proceedings of the 37th conference on Design automation*, pages 681–686. ACM Press, 2000.

[15] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[16] A. Narayan, A. Isles, J. Jain, R. Brayton, and A. Sangiovanni-Vincentelli. Reachability Analysis Using Partitioned-ROBDDs. In *Proc. of the Intl. Conf. on Computer-Aided Design*, pages 388–393, 1997.

[17] A. Narayan, J. Jain, M. Fujita, and A. L. Sangiovanni-Vincentelli. Partitioned-ROBDDs - A Compact, Canonical and Efficiently Manipulable Representation for Boolean Functions. In *Proc. of the Intl. Conf. on Computer-Aided Design*, pages 547–554, 1996.

[18] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Implicit State Enumeration of Finite State Machines using BDD's. In *Proc. of the Intl. Conf. on Computer-Aided Design*, pages 130–133, November 1990.

Appendix: Composition Complexity for ROBDDs

Bryant presented two fundamental algorithms, *apply*, and *compose*, for working with ROBDDs in [3]. The algorithm for *apply* takes as input two ROBDDs G_f and G_g representing functions f and g respectively and a binary operator $\odot$ and produces OBDD G_h representing function $f \odot g$. Bryant has proved that the size $|G_h|$ of G_h is bounded by $|G_f| * |G_g|$, and has provided a function where (for a given variable order) the above quadratic blow-up can be observed. The algorithm for *compose* can be regarded as replacement by functions; for two functions f (given by ROBDD G_f) and g (given by ROBDD G_g) and a variable x_i, the function $f_{x_i=g}$ defined by $h = ite(g, f_{x_i=1}, f_{x_i=0})$ (if g then $f_{x_i=1}$ else $f_{x_i=0}$) has to be represented. If G_h is the ROBDD that represents h, Bryant showed that $|G_h|$ has an upper bound of $O(|G_f|^2 * |G_g|)$. However, Bryant could not find any worst-case example requiring the above cubic blow-up. Thus, he observed that "([3],pp.261) It is unclear whether the efficiency of this algorithm truly has a quadratic dependence on the size of its first argument, or whether this indicates a weakness in our performance analysis."

Jain et. al [14] answered Bryant's open problem and presented a worst-case example of *compose* which is also a worst-case example for the more general *ite* operation. In the following we excerpt the proof of Jain et al. to understand the nature of composition and also on why it truly has a cubic worst-case complexity.

Proof ([14]): Let $MUX(a, x)$ be defined on $n+k$ variables $a_o, ..., a_{k-1}$ and $x_o, ..., x_{n-1}$ where $n = 2^k$. The a-variables are control variables describing a number $|a| \in \{0, ..., n-1\}$; the x-variables are data variables addressed by $\vec{a}$. Hence, $MUX(a, x)$ is defined as $x_{|a|}$. For this multiplexer (or direct storage access function) the ROBDD variable ordering (a, x) is optimal. The a-variables may be in arbitrary order. The same holds for the x-variables. The ROBDD size is $2n+1$ for this ordering. The complete binary a-tree contains $n-1$ nodes with n outgoing edges. For the outgoing edge representing the address a the variable $x_{|a|}$ is tested. This leads to n x-nodes and 2 sinks.

We define f on $n+2k+1$ variables: $a_o, ..., a_{k-1}$; $b_o, ..., b_{k-1}$; $x_o, ..., x_{n-1}$, and s. Let $f(a, b, s, x) = s \wedge MUX(a_o, ..., a_{k-1}, x_o, ..., x_{n-1}) + \bar{s} \wedge MUX(b_o, ..., b_{k-1}, x_o, ..., x_{n-1})$. Moreover, we define g on $n+k$ variables $c_o, ..., c_{k-1}; x_o, ..., x_{n-1}$ by $g(c, x) = MUX(c_o, ..., c_{k-1}, x_o, ..., x_{n-1})$.

Both functions are considered as functions on all $n+3k+1$ variables in a, b, c, s and x and we investigate for both functions the optimal variable ordering: $s, a_o, ..., a_{k-1}; b_o, ..., b_{k-1}$; $c_o, ..., c_{k-1}; x_o, ..., x_{n-1}$.

The ROBDD size of f is $1+2(n-1)+n+2 = 3n+1$. We start with an s-

node and then realize $MUX(a,x)$ and $MUX(b,x)$. The ROBDDs for these functions may share the x-nodes and the sinks. The ROBDD size of g is $2n+1$. Let $h = f_{s=g} = MUX(c,x) \wedge MUX(a,x) + \overline{MUX}(c,x) \wedge MUX(b,x)$.

By the characterization of the ROBDD size by Sieling and Wegener [?] the ROBDD for h contains at least as many x-nodes as there are different cofactors which are obtained by assigning constants to the control variables a, b, and c. If $|a| = i$, $|b| = j$, and $|c| = k$, we obtain the cofactor $x_k x_i + \bar{x}_k x_j$. If $i \neq j$, the cofactors depend essentially on all their three variables and are all different. For $i = j$ we obtain the function x_i. Hence, we get $n^2(n-1) + n = n^3 - n^2 + n$ different cofactors which have to be represented by different x-nodes. These x-nodes are reached by edges from the upper part where the control variables are tested. Since the ROBDD has a single source, the ROBDD contains at least $n^3 - n^2 + n - 1$ nodes where control variables are tested and 2 sinks. This leads to the lower bound $2n^3 - 2n^2 + 2n + 1$ for the ROBDD size of h. Hence, $|G_f|^2 |G_g| = 18n^3 + 21n^2 + 8n + 1$ and $h = f_{|s=g} = ite(g, f_{s=1}, f_{s=o}) = ite(MUX(c,x), MUX(a,x), MUX(b,x))$ has an ROBDD size of at least $2n^3 - 2n^2 + 2n + 1$. Hence, $|G_h| = \Theta(|G_f|^2 |G_g|)$.

Cubic-complexity of Composition and Partitioned-OBDDs

Note that on the right hand side of compose operations, the disjuncts $\overline{\psi_{i_{bdd}}} \wedge f_{d_{\overline{\psi_i}}}$ and $\psi_{i_{bdd}} \wedge f_{d_{\psi_i}}$ are mutually orthogonal, and can thus be represented as different partitions in a POBDD. In our method we successively compose the $\psi_{i_{bdd}}$s in f_d. If the graph size increases drastically for some composition (say ψ_j) we can abort the compose operation and instead separately compute each of the two disjuncts in the right hand side of equation 1; each of these disjuncts constitutes a separate partition. In other words, we can create two orthogonal partitions instead of continuing the ROBDD composition to completion.

$$Partition1 : \overline{\psi_{i_{bdd}}} \wedge f_{d_{\overline{\psi_i}}}; \;\; Partition2 : \psi_{i_{bdd}} \wedge f_{d_{\psi_i}} \qquad (2)$$

Note that each partition can be constructed in quadratic time using the *apply* algorithm. That is, the complexity of creating such partitions, during the composition of ψ_i in f_d, is only $O(|f_d| \cdot |\psi_{i_{bdd}}|)$. We can now individually call the composition routine on each of the partitions. As remaining decomposition points are composed inside any partition, a partitioning is performed each time a blow-up of composition is observed. If in composing f_d by $\psi_1, \ldots, \psi_k$, a composition blow-up is observed a total of c times, then we will have produced $c+1$ ROBDDs, $P_1, \ldots, P_{c+1}$. Together, $P_1, \ldots, P_{c+1}$

represent the complete Boolean space of F in terms of primary input variables $x_1, \ldots, x_k$ and constitute the POBDD representation of F.

Contrasting the Complexity of Composition using ROBDDs with that using a Partitioned Representation

It is instructive to compare the size of the monolithic ROBDD with the corresponding POBDD that a sequence of k worst case compositions may lead to.

Complexity of ROBDDs: Due to the cubic complexity of ROBDD composition, composing k ROBDDs $g_1, \ldots, g_k$, each of size $|g|$, in an ROBDD G can require $O(|G|^{2^k} * |g|^{2^k - 1})$. The size of POBDDs is bounded by a much smaller polynomial as discussed below.

Complexity of POBDDs: Considering each of the disjuncts in Equation 2 as a separate partition, each partition can be constructed in $O(|G| * |g|)$. While composing the remaining $k - 1$ decomposition points, each of the original partitions can be further split into 2^{k-1} partitions. Thus, in the worst case, we are left with 2^k partitions. Note that the size of each partition is bounded by $O(|G| * |g|^k)$. Thus, the composition complexity of creating POBDDs by our method is bounded by only $O(2^k * |G| * |g|^k)$. (In the above analysis, for simplicity, each decomposition point is assumed to be a function of primary inputs only.)