Copyright

by

Subramanian Krishnan Iyer

2006

The Dissertation Committee for Subramanian Krishnan Iyer
certifies that this is the approved version of the following dissertation:

Efficient and Effective Symbolic Model Checking

Committee:

E. Allen Emerson, Supervisor

Jacob A. Abraham

Aloysius K. Mok

Donald S. Fussell

Jawahar Jain

Efficient and Effective Symbolic Model Checking

by

Subramanian Krishnan Iyer, B.Tech.; M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2006

To Amma and Appa

Acknowledgments

Even though the title page bears my name alone, this dissertation would not have been possible without the contributions of many others.

The notions of model checking and state partitioning, which have been developed and refined over the years by Prof. E. Allen Emerson and Dr. Jawahar Jain, respectively, form the basis of this work.

I gratefully acknowledge the supervision of Prof. Emerson, for teaching me the theory behind this subject; for his vast knowledge and uncommon commonsense; for his ready accessibility; for his approach to, and integrity in, conducting superior scientific research; and for the stimulating discussions we have had on a plethora of topics – academic and otherwise. I truly cannot imagine having had a better dissertation supervisor.

I am grateful for the opportunity to collaborate closely with Dr. Jain, Research Fellow at Fujitsu Laboratories of America, not only for his role in shaping my research but also for being a friend, philosopher and guide.

My collaboration with Debashis Sahoo, graduate student at Stanford, was a pleasure. My interaction with Drs. Amit Narayan, Christian Stangier and Mukul Prasad was very valuable and I thank them for their help and guidance.

I am indebted to all my teachers, especially in Mathematics and Computer Science. In particular, Prof. S. Ramesh at the Indian Institute of Technology, Bombay, India and Prof. R. Ramanujam at the Institute of Mathematical Sciences,

Madras, India introduced me to formal methods, and without them I would not have traveled down this road.

I wish to thank Profs. Abraham, Fussell and Mok for serving on my committee, as well as other faculty members and student colleagues of the Computer Sciences Department for providing a stimulating and fun environment. My gratitude goes out to the people and staff at the University of Texas at Austin who have made this academic institution wonderful.

My stay in Austin was made especially pleasant by numerous friends, to whom I am grateful for being my surrogate family in our years spent here.

Last but not the least, I wish to thank my parents, to whom this thesis is dedicated, as well as friends and relatives, for their endless patience and encouragement when I most needed it.

SUBRAMANIAN KRISHNAN IYER

The University of Texas at Austin

December 2006

Efficient and Effective Symbolic Model Checking

Publication No. ____________________

Subramanian Krishnan Iyer, Ph.D.
The University of Texas at Austin, 2006

Supervisor: E. Allen Emerson

The main bottleneck in practical symbolic model checking is that it is restricted by the ability to efficiently represent and perform operations on sets of states. Symbolic representations, like Binary Decision Diagrams, 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, one reason being that they have to replicate the transition relation numerous times.

We propose techniques to increase the capacity of automatic state-based verification as applied to sequential designs, i.e., symbolic model checking. Firstly, we propose the use of dynamically partitioned ordered Binary Decision Diagrams as a capable data structure. This leads to vast improvements in state space traversal in general and error detection in buggy designs, in particular. Secondly, we propose

a partitioned approach to model checking, which splits the problem into multiple partitions handled independently of each other.

State space partitioning-based approaches have been proposed in the literature to address the *state explosion problem* in model checking. These approaches, whether sequential or distributed, perform a large amount of work in the form of inter-partition (*cross-over*) image computations, which can be expensive. We present a model checking algorithm that aggregates these expensive cross-over images by localizing computation to individual partitions. This algorithm is more suited to parallelization than existing model checking approaches. It reduces the number of cross-over images and drastically outperforms extant approaches in terms of *cross-over* image computation cost as well as total model checking time, often by two orders of magnitude.

We address the issue of time scalability in verification, whereby the availability of larger amounts of computation time enables greater exploration of the state space. 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 explore this issue further.

Finally, we study the combination of partitioned binary decision diagrams with bounded model checking for more scalable and efficient model checking. We give a technique to scale formal verification to a large grid of processors that demonstrates marked superiority over existing approaches.

The contributions of this dissertation are in improving the capacity of symbolic model checking approaches to formal verification, in terms of time and memory requirements, as well as in the development of techniques that are more readily amenable to parallel and distributed model checking.

Contents

Acknowledgments v

Abstract vii

List of Tables xiii

List of Figures xv

Chapter 1 Introduction 1

Chapter 2 Background and Related work 4

2.1 Temporal Logics 4

2.2 Reachability and Model Checking 5

2.3 State Partitioning 7

2.4 Related Work 8

2.4.1 Other Partitioning Related Approaches 8

2.4.2 Other Model Checking Approaches 9

Chapter 3 Partitioning for Symbolic Reachability 12

3.1 Improving state space traversal 13

3.2 Time Scalability in Verification 14

3.2.1 What is missing in the classical approach? 15
3.3 The Partitioning Methodology 16
3.3.1 Whether to partition: Initial Partitioning 17
3.3.2 How to Partition: Choice of partitioning variable 18
3.3.3 Are more partitions required: Global Dynamic Repartitioning 19
3.3.4 Partition only Image: Local Partitioning 20
3.3.5 How to order partitions: Scheduling 21
3.4 Tracing Erroneous Paths 24
3.5 Addressing Instability in BDD-based verification 26
3.5.1 Cause 1: Sensitivity to Parameters 27
3.5.2 Cause 2: Variations of state space traversal 27
3.5.3 Cause 3: Sensitivity to scheduling of partitions 28
3.5.4 Trace-centric approach to stability 28
3.5.5 Parallelized search using partitioning 29
3.6 The complete Reachability Algorithm 30
3.7 Analysis of Experimental Results 32
3.7.1 Benchmarks 32
3.7.2 Comparison with Non-partitioned Invariant Checking 32
3.7.3 Comparison with Non-partitioned Reachability Analysis . . . 34
3.7.4 Comparison with Static Partitioning 34
3.7.5 Comparison with and without traces 35
3.7.6 Multiple traversals smooth out instability 36
3.8 Conclusions 41

Chapter 4 On Partitioning and Model Checking 42

4.0.1 Contributions of this work 44

4.0.2 Organization of this chapter 45
4.1 Partitioned Model Checking: A simple algorithm 45
4.1.1 Partitioned Image Computation 46
4.1.2 Partitioned Computation of fix-points 47
4.1.3 Issues: Local and cross-over images 47
4.2 Improving Partitioned Model Checking 49
4.2.1 Image Computation 49
4.2.2 Fix-point computations 50
4.2.3 Correctness of least fix-point algorithm 51
4.2.4 Construction of the Greatest fix-point algorithm 53
4.2.5 Correctness of greatest fix-point algorithm 53
4.2.6 Analysis 55
4.3 Experimental Results 56
4.3.1 Experimental Setup and Benchmarks 56
4.3.2 Results and Analysis 58
4.4 Conclusions 59

Chapter 5 Partitioning for Bounded Model Checking 60

5.1 Introduction 60
5.1.1 Our Approach 62
5.1.2 Organization of this chapter 63
5.2 Related Work 64
5.2.1 Our approach in context of other hybrid approaches 65
5.2.2 Our approach in context of other parallel approaches 66
5.3 Preliminaries 67
5.3.1 SAT-based Bounded Model Checking 67

5.3.2 The Grid Framework 68

5.4 Algorithm: Under-approximation based Grid-BMC 69

5.4.1 Key Idea: Under-approximation for Grid-BMC 69

5.4.2 Approximation and Usage of Grid 72

5.4.3 Outline of Grid-BMC Algorithm 73

5.4.4 Comments on technique 74

5.5 Experimental Results 76

5.5.1 Experiments with Simulation alone 77

5.5.2 Experiments on Grid-BMC 77

5.5.3 Case Study 79

5.5.4 Analysis of the Results 80

5.5.5 Effect of under-approximation 81

5.6 Conclusions 83

Chapter 6 Future Work and Conclusions **85**

6.0.1 Generalized Windows and Multiply rooted BDDs 86

6.0.2 Simulation-guided circuit slicing for BMC 88

6.1 Conclusions 90

Bibliography **92**

Vita **101**

List of Tables

3.1 Comparison of BDD-based VIS with POBDD-VIS on VIS-benchmarks for all circuits where VIS requires more than 250K BDD nodes. The time includes CPU time for simultaneous check of all properties of a given circuit 33

3.2 Comparison of BDD-VIS with POBDD-VIS on ISCAS89 benchmark 34

3.3 Comparison of Proposed POBDD with trace and without trace on all designs where VIS runs out of time or memory. 35

3.4 BDD-based falsification is moderately sensitive to parameters, all runs are BFS. 36

3.5 POBDD-based falsification is much more sensitive to parameters, runs differ in state traversal. 37

3.6 Comparison of time for falsification. The columns titled sequential refer, resp., to the baseline runs of Fig 3.8 and 3.9. The columns titled parallel refer, resp., to performing five traversals in parallel using OBDDs and POBDDs. 38

3.7 The largest BDD created during parallel falsification 39

4.1 Comparison of existing and proposed algorithms for partitioned model checking CTL properties on circuits in the VIS Verilog benchmark suite. 57

5.1 Run-times for BMC seeded from simulation to various depths. 77
5.2 Comparison of the time taken in seconds by various approaches. . . 78
5.3 Details of performance data of proposed hybrid Grid-BMC approach. 79
5.4 Effect on Grid-BMC performance (time in seconds) of relaxing the severity of the approximation 82
5.5 Effect on Grid-BMC performance (time in seconds) of drastically relaxing the severity of the approximation 82

List of Figures

3.1 Initial Partitioning Algorithm 17

3.2 Selecting Partitioning Variable 18

3.3 Dynamic Partitioning 20

3.4 Computing Image with Local partitioning 21

3.5 Scheduling-based Reachability Algorithm 23

3.6 Trace-based Reachability Algorithm 31

3.7 Comparison of Dynamic trace-centric and Static Partitioning Approaches on all Large designs (time>1000s) of VIS-benchmarks. The actual runtime in seconds is shown at the top of each bar. 35

3.8 BDD-based falsification is moderately sensitive to parameters, all runs are BFS. 36

3.9 POBDD-based falsification is much more sensitive to parameters, runs differ in state traversal. 37

4.1 Backwards Image Computation with Partitioning 46

4.2 Classical Model Checking of Fix-points in presence of Partitioning . 47

4.3 Local and Cross-over Components of Image Computation with Partitioning 50

4.4 Fix-point Computations localized by postponing cross-over images . 51

5.1 ILA for SAT-based BMC 68

5.2 Seeding multiple SAT-BMC runs from POBDD reachability 71

Chapter 1

Introduction

In this dissertation, we address 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 [CE81] emphasized an enumerative implementation, while a symbolic technique using BDDs [Bry86] was suggested by McMillan [McM93]. 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.[JBFA92], and extended by Narayan, et.al.[NJFSV96]. More recently, satisfiability-based model checking [CBRZ01] has 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.

Chapter 2

Background and Related work

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.

Any formula in the Computation Tree Logic, CTL, can be written 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 details of the syntax and semantics of temporal logics like CTL as they are widely known and readily available in the literature, see for instance [Eme90].

2.2 Reachability and Model Checking

The earliest approach to model checking [CE81] emphasized an enumerative implementation, while a symbolic technique using Reduced Ordered Binary Decision Diagrams (ROBDDs, or just BDDs) [Bry86] was suggested by McMillan [McM93].

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 associated 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 $s \to s' \in T$ 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 fix-point computation which performs a breadth-first traversal of finite-state structures [CBM89, McM93, TSL$^+$90]. The algorithm takes as inputs the set of initial states, $I(s)$, expressed in terms of the present state variables, s, and the transition relation, $T(s, s')$, relating the set of next states, $N(s')$, that a system can reach from a state s on an input i. For brevity, we do not henceforth refer to the input i. For sequential designs, T is obtained as the conjunction of the transition relations, $s'_k = f_k(s, i)$, of the individual state elements, i.e., $T(s, s') = \prod_k (s'_k \equiv f_k(s))$. Given a set of states, $R(s)$, that the

system can reach, the set of next states is defined as $N(s') = \exists s[T(s,s') \wedge R(s)]$. This calculation of N is also known as *image computation*. Similarly, the *backward image computation*, which calculates the set of states $N(s)$ from which the system can reach given set of states $R(s')$, uses the equation $N(s) = \exists_{s',i'}[T(s,s',i) \wedge R(s')]$. The set of reachable states is computed by adding $N(s)$, obtained by substituting variables s for s', to $R(s)$ and iteratively performing this image computation step until a fix-point 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. In this document, the term model checking will refer to this traditional backward model checking, rather than the recently suggested [IN97] forward model checking procedure.

Since there exist computational procedures for efficiently performing Boolean operations on symbolic BDD data structures, 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) fix-point of the operator $\tau(Z) = q \vee (p \wedge EXZ)$ (resp. $\tau(Z) = p \wedge EXZ$) and can therefore be computed as

a fix-point.

2.3 State Partitioning

The idea of partitioning was used to discuss a function representation scheme called partitioned-ROBDDs in [JBFA92, Jai93] which was extensively further developed in [NJFSV96].

Definition 1 *[NJFSV96] Given a Boolean function* $f : \mathbb{B}^n \rightarrow \mathbb{B}$*, 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,* $w_i : \mathbb{B}^n \rightarrow \mathbb{B}$ *and* $f_i : \mathbb{B}^n \rightarrow \mathbb{B}$*, 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 of the function is represented separately as an ROBDD 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 [NJFSV96]. An excellent comparison of the computational power of various BDD based representations, including POBDDs, may be found in [BW97].

Although "partitioning" suggests disjoint windows, referred to as unique nondeterminism in complexity theory, it can be renounced for "overlapping" windows which yield a more powerful POBDD model [BW97]. This is beyond the scope of the current work, and we do not refer to it again until the chapter on future work. In the rest of this document, we only consider *disjoint window-based state partitioning*. The reason for this is that this representation is canonical, and allows negation to be performed locally in each partition. Other schemes for dividing the state sets, notably that of [GHS01], need to perform a global synchronization operation to perform negation.

2.4 Related Work

We now briefly survey related work in this area and compare it to our approach. We look at approaches that use partitioning, as well as recent advances in model checking.

2.4.1 Other Partitioning Related Approaches

The use of *partitioned transition relations* [BCMD92, JED91] was proposed to control the size of the symbolic representation of the transition relation. 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 that are then conjuncted during image computation [RAB$^+$95, MHS00]. 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 [HKB93, TSL+90]. 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 [CCQ96] 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. We 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, that approach is complementary to our disjunctive state partitioning approach, and we use these "clustered"-transition relations in the construction, representation and use of transition relations.

2.4.2 Other Model Checking Approaches

A *distributed model checking* algorithm was proposed in the literature [HGGS00a, GHS01]. This approach uses "slicing", which is similar to a prior partitioning approach [NIJ+97], with the objective of doing model checking in a distributed fashion.

This approach partitions the computation into a fixed number of fragments equal to the number of processors available in the distributed environment. It does not address issues related to costs of communication and variable ordering in different partitions. Further, negation is a global operation and requires synchronization between every pair of slices in this approach.

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. We show how to drastically cut down the number of instances of inter-partition communications. This reduces the number of transfers and re-orderings of large BDDs between partitions and is found to be a significant gain in practice. Our use of window-based partitioning allows for all boolean operations to be performed locally. We also give a way to compute the image using computations that are only linear rather than quadratically many, in the number of partitions.

A similar idea has been used by Lerda, et.al. [LST03] for cycle detection in the context of *software model checking*. Their work combines partial order reduction with symbolic model checking. The state space is a cross product of the states of all processes, and the transition relations are a result of structural partitioning of the state space. Taking advantage of the concurrent processes, they create transition relations for individual processes and do cycle detection in this context.

In contrast, we have one finite state system, as opposed to their scenario of multiple concurrent systems. Accordingly, we have to use one transition relation to represent the entire system. The partitioning in our work refers to functional partitioning. In our algorithm the “Border” is the candidate set for those states that are the result of transitions that cross over from one partition to another and

may have the property under consideration. The critical issue is that it is not possible to check whether these states satisfy the property without doing the cross-over images so they need to be carried along until such time as the next cross-over image is performed. Section 4.4 has the details.

Since the focus of this work is to improve symbolic BDD-based model checking in the presence of partitioning, a comparison with purely *SAT-based approaches* like Bounded Model Checking [CBRZ01] is not in the scope of the current work. However, we describe a hybrid method combining BDDs and SAT solvers in a subsequent chapter.

Chapter 3

Partitioning for Symbolic Reachability

Partitioned Ordered Binary Decision Diagrams (POBDDs) described before serve as the starting point for our approach. However, the partitioning scheme proposed and presented there [NJFSV96] 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. Such a partitioning scheme based on an a priori selection of the 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.

3.1 Improving state space traversal

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 co-factoring 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 invariant properties 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 fix-point 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.

3.2 Time Scalability in Verification

One of the major disadvantages of using extant BDD-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 thrashes.

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.

3.2.1 What is missing in the classical approach?

For performing operations on many functions, ROBDDs suffice. In such cases, especially 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 ROBDD, then such representations 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, operations are performed the representations that are created. What if the results of these operations are very simple? Should a subset of such simple representations be combined 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 scheduling order for the required computations?

We posit that the above questions are fundamental to creating a practical partition handling algorithm.

In this chapter, we wish to address questions like the above. An efficient solution to these problems leads to vastly improved practical results in the ability to handle large designs.

3.3 The Partitioning Methodology

The problem of reachability is about representation of sets of states and relations, as well as operations performed on them. The key operation is successive image computation on fragments of the state space until all reachable states have been explored. Thus, there is a need to develop an approach which can be efficient for both aspects – creating subspaces so as to represent functions succinctly as well as doing image computation.

In this context, the following questions naturally arise:

1. Is partitioning required at all?

2. If we must partition, what constitutes the "axis of partitioning"? In other words, along what lines should the partitioning be performed? Fro instance, what splitting variables should be used for creating windows?

3. As computation is performed, is the partitioning effective or is more partitioning required?

4. If the blowup is likely to be temporary (local), can the partitioning be likewise?

5. Once partitions are generated, in what order should they be processed?

These issues give more heuristic challenges on the POBDDs which can lead to a successful strategy in managing the behavior of BDDs in verification. Further due to the dynamic nature of partitioning, our approach can reduce the memory explosion in many circumstances. In contrast, the monolithic approach can exert no control on the program to prevent it from generating huge data structures that overflow memory.

We begin by discussing the algorithms for construction and utilization of partitioned representations, which address the questions raised above. Then, we detail the essential points of a trace-centric approach in the next section. This is used to impose some stability on the performance of the OBDDs with respect to the selection and setting of appropriate parameter values. At the end we give a complete reachability algorithm based on all the heuristics described in this chapter.

We now describe the mechanism for the construction and practical application of Partitioned BDDs.

3.3.1 Whether to partition: Initial Partitioning

Since reachability needs manipulation of image BDD using transition relation, if either of them shows signs of blowup then partitioning seems to be the prudent choice. Figure 3.1 shows how partitioning is invoked. If the transition relation is

```
InitialPartitioning(T, I) {
    If (T is large) {
        R := I
        Do Partitioning using T as basis.
    } else {
        R := Do Reachability from I using T until Blowup.
        Do Partitioning using R and T as basis.
    }
    return Partitioned R;
}
```

Figure 3.1: Initial Partitioning Algorithm

small, then many initial steps of reachability identical to the classical approach using a single BDD can be performed and partitioning can be delayed. Reachability is performed using BDDs until such time as a "blowup" in BDD size is detected. This may be measured either absolutely as a maximum size of the symbolic representation

of the image or in a relative way as the ratio of the representation of the reached states before and after any image computation. We adopt the latter approach with a threshold factor chosen *a priori*. However, if the transition relation cannot be easily constructed, then it is advantageous to partition quickly.

3.3.2 How to Partition: Choice of partitioning variable

After a "blowup" is detected, we select n splitting variables and the corresponding 2^n partitioning windows are created. The choice of the splitting variables is critical to the the effectiveness of the partitioning approach. The goal is to create small and relatively balanced partitions that represent *non-compatible* functions. A set of functions is said to be *non-compatible* if the totality of their individual representations using different orders is far more compact, than their combined representation as a whole. The splitting variable is selected by means of a cost function, for example, as described in [NIJ+97]. For each variable, the cost function takes into account the relative BDD sizes of the positive and negative co-factors with respect to the BDD size of the original graph.

```
SelectPartitioningVars(basis BDD F) {
    for (each method i := 1 to m) {
        get ordered splitting variable list using F.
        select top k variables.
        for (each subspace j := 1 to 2^k) {
            cost[i][j] := size of the cofactor F_j.
        }
        cost of method i := Σ_j cost[i][j]
    }
    select method with lowest cost.
    return corresponding vars.
}
```

Figure 3.2: Selecting Partitioning Variable

Even though the transition relation does not change, its BDD size can vary drastically due to dynamic variable reordering. The measurement of graph sizes for determining a blowup and for recognizing its subsidence can be done with respect to the BDD size of the transition relation or the image representation or both. We try to get separate splitting variable choices from each of these three methods. We select that choice which gives the smallest co-factor graphs after reordering as illustrated in the Figure 3.2. Intuitively, this selects a variable that creates two partitions that are as non-compatible as possible.

3.3.3 Are more partitions required: Global Dynamic Repartitioning

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[BW97], that a $(k+1)$-POBDD can be exponentially more succinct that a k-POBDD.

Whenever a BDD size blowup is detected during computation in a partition, dynamic repartitioning [ISS+03] is performed, as illustrated in Figure 3.3. Repartitioning is performed by splitting the given partition by co-factoring the entire state space based on one or more suitable, newly calculated, splitting choices until the blowup has been ameliorated. Initially, the partitioning is done using one splitting variable. To prevent excessive overhead in the new splitting variable selection, they are obtained by recalculating the cost of only the top few choices provided by the partitioning variable selection method discussed before. At this point, each new partition is checked to see whether the blowup has subsided. If not, repartitioning is recursively performed on that partition. A threshold on maximum number partitions is kept to prohibit the method to produce exponential number of partitions.

```
DynamicPartition(basis BDD F, partition i){
    v := SelectPartitioningVars(F)
    create partition i1 from i with v := 0
    if (blowup in i1)
        DynamicPartition(F_{v:=0}, i1)
    create partition i2 from i with v := 1
    if (blowup in i2)
        DynamicPartition(F_{v:=1}, i2)
}
```

Figure 3.3: Dynamic Partitioning

It must be noted that the variable selection algorithm ensures that superfluous partitions are not created and that the ones created are somewhat balanced. In practice this imposes a bound on how many partitions are actually created.

3.3.4 Partition only Image: Local Partitioning

During each step of image computation, many steps of alternating composition and conjunction are performed. Often it is found that the blowup in the BDD sizes during such a *micro-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 could create a large number of partitions, whose BDD sizes become eventually very small. 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. If local partitioning does not reduce the blowup, then dynamic global repartitioning can be done. To create the local partitions, we cofactor using the ordered list of splitting variables that was generated earlier. Figure 3.4 describes how this is done.

Next, we will describe how to effectively use the partitioned data structure

```
ComputeImage(TR, state set R, variable list L){
    do {
        one microstep of image
        if (blowup) {
            varList := top k vars from L
            create partitions using varList
            for (each new partition)
                recursively do all remaining micro steps
        }
    }while(microsteps remain)
}
```

Figure 3.4: Computing Image with Local partitioning

once it has been created.

3.3.5 How to order partitions: Scheduling

In this section we describe our technique for state space traversal which schedules partitions based on their difficulty of traversal. The goal of the scheduling is to discover error states as early as possible in the state space traversal. The expectation is that the probability of catching an error is higher as more of the state space is covered. Therefore, some errors can be caught earlier if more state space is covered sooner. We characterize partitions in terms of how quickly it has been possible to cover state space symbolically in that partition. This is measured in terms of a cost for processing the partitions. The details of how this cost is computed is described in the following. Once this characterization of the level of difficulty is available, we schedule the partitions for processing in ascending order of their costs. Thus, the state space can be explored in a way that speeds up the rate at which new states are discovered. We measure how difficult it is to represent and manipulate the state space within that partition by means of BDDs. Those partitions that are

less amenable to manipulation as BDDs will be termed “hard” partitions. All other partitions are called “easy”. The scheduler tries to schedule easy partitions ahead of hard partitions. The cost metric described above can only be used to compare two different partitions. The scheduler characterizes top half lowest cost partitions as easy partitions. This is a reasonable assumption to make because the costs are relative.

It must be noted that scheduling partitions in this fashion enables a certain amount of control in case the number of partitions gets too large - for the priority of empty or sparse partitions can be lowered, and thereby the overhead associated with traversing them repeatedly can be reduced. Notice that in the “worst” case, this processes all the partitions and thus traverses the entire state space if the design is correct. This may occur for example, in a where no error can be detected.

Scheduling Cost metrics

We will now describe two metrics that are used for assigning a scheduling cost for processing the partitions.

Density based scheduling

Similar to [RS95] we define the *density* of a partition as the ratio of the number of reachable states discovered in that partition to the size of the BDD representing the reachable states. It may be noted that large function representation sizes, i.e. BDD sizes, are the most important bottleneck in symbolic verification techniques. In the case of traditional methods, large BDD sizes prohibit any further exploration of the state space. In the partitioned approach, large BDD sizes trigger repartition, thus leading to an increase in the number of partitions to be explored, and in the worst case they can be a cause for abandoning the exploration of that partition. Thus, in

the interest of greater and faster state space coverage, it is advisable to first process partitions with a higher *density* of states. This ratio measures how efficiently the reachable states can be represented in terms of BDDs. If this ratio is normalized with respect to size, partitions with a greater density should be considered easy, and therefore assigned a lower cost of processing.

```
Reachability(T, I) {
    R := InitialPartitioning(T, I)
    Initialize Priority Queues in Scheduler S;
    do {
        Get LFPList from S.LFPQueue
        for each partition i in LFPList
            Calculate LeastFixedPoint in i and update S
        Get CommList from S.CommQueue
        for each partition i in CommList
            Communicate from i to all parts and update S
    }until (No new state is added to R);
}
```

Figure 3.5: Scheduling-based Reachability Algorithm

Time based scheduling

Note that each partition may require many fixed point computations. Hence, another useful metric takes into account the time required for the latest fixed point computation within each partition, but excluding the time spent in communicating either to or from this partition. Let us examine what it means to say that this time is small. In the corner case that this partition is almost fully explored, there will be very few image computations and therefore this time would be small. But in the general case, when there are a significant number of image computations, the total tme can be small only if each image requires very little time. The partition

with faster fixed point computation is intuitively more attractive as it may be more amenable to symbolic manipulation using BDDs. Therefore, it is advantageous to select partitions which have historically been known to take lesser time. In the above calculations the time spent in communicating either to or from any partition was excluded.

The cost for processing a partition is the ratio of the time taken for the most recent fixed point computation to the density of that partition. Intuitively, this prioritizes partitions that are more amenable to symbolic traversal. In our reachability algorithm (Figure 3.5), priority queues are used to schedule partitions in increasing order of their cost. Intuitively, this covers a large number of states quickly. In effect, we postpone the traversal of states that are more expensive to discover.

3.4 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.

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.

3.5 Addressing Instability in BDD-based verification

When BDDs are used to perform a Breadth First Search (henceforth, BFS) traversal of the state space, the performance is sensitive to parameters. Instability in BDD based verification refers to the sensitivity of BDD sizes and therefore, performance as well, to various parameters like the size of the clusters in the implicitly conjoined transition relation, the selection of variable reordering methods, etc. It is observed that a single choice seldom works uniformly for all cases and therefore, such parameters need to be tweaked manually. The performance of BDD based methods can vary widely and unexpectedly based on these settings. Note that in each case, during reachability, the same sets of states are computed.

We now consider in greater detail three main causes of instability in POBDD based approaches to formal verification.

3.5.1 Cause 1: Sensitivity to Parameters

In a partitioned scheme, there are an even greater number of specific choices available for state space traversal. In other words, the order in which the states are traversed depends on the various partitioning factors. The degrees of freedom include the number of partitions, when and whether to dynamically re-partition, how to schedule the image computations involving multiple partitions, the selection of the partitioning functions, the instance when partitioning is commenced, etc.

3.5.2 Cause 2: Variations of state space traversal

Notice that the POBDD-based reachability algorithm is not a strict BFS traversal. It performs a BFS which is local to individual partitions, and then synchronizes to add states that result from transitions crossing over from one partition to another. We may characterize this as a *region-based BFS*, where individual regions of the state space, *i.e,* the partitions, are traversed independently in a breadth first manner.

POBDD-based traversals which differ even slightly in their choice of partitioning factors creates dissimilar partitions of the state-space and can thus traverse the states in a substantially different order. Based upon the partitioning variables, the size of the graph can also greatly vary, especially since POBDDs have a great potential for compactness. This potential for greater compactness and the possibility of large variation in the path chosen for state space traversal exacerbates the sensitivity of POBDD-based approaches to their settings. This is consistent with our empirical experience, presented in Section 3.7.6, where we find that POBDD-

based approach to falsification has a higher sensitivity to the choice of parameters than OBDD-based methods.

3.5.3 Cause 3: Sensitivity to scheduling of partitions

Another issue with the use of POBDD-based reachability for falsification is the sequence in which partitions are explored relative to partitions that have errors. Exploring partitions with no error state ahead of those with error states can lead to a large delay in discovering the error. Further, it may be relatively easier to reach an error state in one partition than in another. Thus the discovery of an error state critically depends upon the scheduling of partitions for exploration. Notice that this problem is unique to falsification, where we expect to find an error, as opposed to verification, where we seek to explore the entire space and show the absence of an error.

In the rest of this section, we describe our approach to tackling these issues.

3.5.4 Trace-centric approach to stability

We propose a *trace-centric* approach to parameter selection which can dynamically fine-tune the partitioning choices and balance the various options available to a BDD based method. This help to efficiently decide the required configuration, and makes the method automatic and stable.

A small set of identical computations are separately executed, each using a different choice of the various options. Each of these is referred to as a *trace*. The length of a trace is how far it proceeded into the entire computation. A large number of traces may lead to a high cost in overhead. Therefore, we elect to look at just a few traces with orthogonal settings. These traces are only observed until the size of

the BDDs exceeds a pre-determined threshold.

The traces are compared with one another on various factors, for example, the blowup of BDDs when performing the image operation, the number of image operations completed, number of states traversed, etc. The configuration for the full run is adopted from that of the most efficient trace. Needless to say that if a trace completes the reachability in the allowed space and time, no further computation is required. We have found that even very simple dynamic examinations can be dramatically effective in stabilizing the performance of BDDs.

Also, we find that the overhead of generating multiple traces is minor when balanced against the savings. Even if graph size is reduced by only a small factor in the more efficient configuration, it proves to be significant, for the following reason. During the reachability computation, multiple re-orderings are triggered, and even saving one large reordering of a BDD with millions of nodes compensates for the calculation of multiple traces, especially since the traces have a much smaller maximum threshold for the BDD sizes.

3.5.5 Parallelized search using partitioning

In traversing the states using POBDDs, when partitions are explored sequentially, then the order in which they are handled is a critical factor in the overall effectiveness of the method. We enhance reachability with a simple scheduling algorithm as shown in Figure 3.5. Essentially, this algorithm maintains a queue of partitions to process. Depending upon the cost function for this queue, the order of processing the partitions varies. Discovering an optimal schedule for state space search is known to be a hard problem. Therefore any such schedule is only expected to perform well heuristically.

In contrast, we propose the handling of partitions in parallel. Each partition is allowed to execute independently of the others as far as it can. Occasionally, we synchronize the different partitions. We use shared memory for this approach to reduce the overhead involved in handling the memory among multiple processors. This distributes the work performed in reachability and side-steps the otherwise difficult and critical problem of finding a good schedule amongst partitions. This parallel search alleviates the dependence of reachability based falsification approaches on the partition scheduling order.

For falsification, once an error is found, we can stop immediatelyand the gain may be superlinear with respect to the number of parallel processors.

In comparison to the sequential execution, the parallel setup can provide a dramatic gain, suggesting that POBDD can be suitable for fast falsification. Also, in the sequential case partitions that have no error may be explored unnecessarily. Even when an error is detected, it may not be in the partition where it is easiest to reach. The parallel case has an obvious advantage that it needs no such ordering, and it gets to the error state quickest to reach. This expectation of dramatic improvement is borne out by the results on multi-threaded reachability [SJI+05], a detailed analysis of which is beyond the scope of this dissertation.

3.6 The complete Reachability Algorithm

A flow chart for the complete reachability algorithm is shown in Figure 3.6. The input to the reachability algorithm is a transition relation T and initial states I. The algorithm first picks a best parameter configuration by running a few short traces with orthogonal settings. Then it runs the $InitialPartitioning$ procedure described in section 3.3.1. After this the algorithm performs POBDD-based state

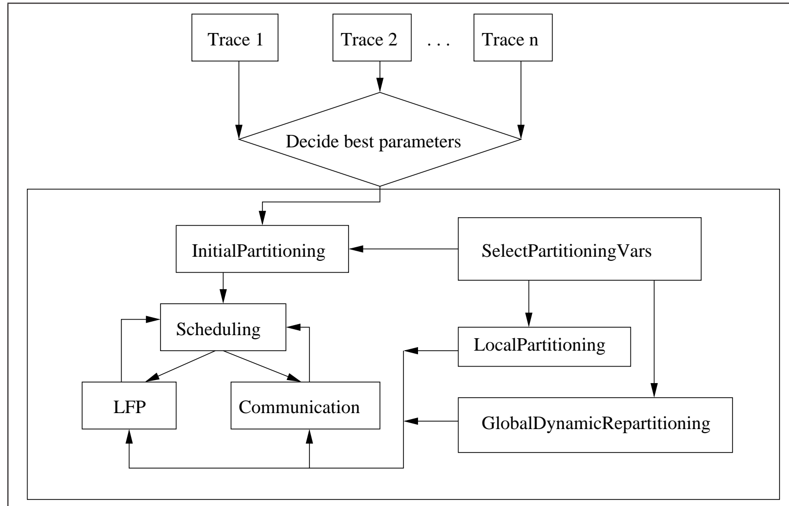

Figure 3.6: Trace-based Reachability Algorithm

traversal guided by the scheduling heuristics described in section 3.3.5. There are two important steps in the POBDD-based state traversal algorithm. The first one is computing a fixed point (called LFP) inside a partition and the other one is to compute image of a function in other partitions (called Communication from one partition to other partitions). The scheduler selects the partition to process next for the above operations. The scheduler implements two priority queues, one for each of the above operations described. Each partition is assigned a cost described in section 3.3.5. The local partitioning described in section 3.3.4 and the global dynamic repartitioning described in section 3.3.3 is enabled during the fixed point and communication. The $SelectPartitioningVars$ heuristics described in section 3.3.2 is called when partitioning is needed.

3.7 Analysis of Experimental Results

Our implementation of the POBDD-data structure and algorithms uses VIS-2.0, which is the state-of-the-art public domain BDD-based formal verification package. We have chosen VIS for its Verilog support and its powerful OBDD-package (i.e. CUDD [Som01]). As our techniques affect only the BDD-data structures and algorithms, they can – with moderate effort – be implemented in other packages as well. These techniques work with any method of image computation; for this implementation, both ROBDDs and POBDDs use the IWLS95 [RAB+95] method.

We found that lazy_sift reordering method works better for most cases. All the experiments use lazy_sift BDD reordering method.

3.7.1 Benchmarks

For experiments on reachability and invariant checking, we chose various public domain circuits: the VIS-Verilog [vis] benchmark suite and ISCAS89 benchmark suite. We choose only invariant checking properties in VIS-Verilog benchmark suite. For sake of brevity, results are omitted for the smaller examples and presented only on those circuits where VIS requires more than 250,000 BDD nodes.

We now compare the methodology of this chapter with three other approaches: the non-partitioned approach of VIS, the static partitioning approach and our own partitioning approach without computation traces. We find that the computation of small traces outperforms, sometimes significantly, all other approaches.

3.7.2 Comparison with Non-partitioned Invariant Checking

Table 3.1 compares the non-partitioned approach of VIS with the proposed method on the time and space needed to check invariant properties from the VIS-Verilog

Circuit	Space (BDD nodes)			Time (sec)		
	Vis	Proposed	Gain	Vis	Proposed	Gain
palu	371K	7K	53.0	186	5	37.2
s1269b	2.6M	38K	68.4	1189	27	44.0
sp_product	919K	70K	13.1	1299	440	3.0
FIFOs	975K	131K	7.4	1704	1521	1.1
vsaR	5.2M	1.3M	4.0	5281	2409	2.2
blackjack	3.2M	1.1M	2.9	16298	11739	1.4
ns3	4.7M	1.0M	4.7	18592	19093	1.0
am2910	11.7M	67K	>174	M/O	222	>392
ball	18.8M	17K	>1106	T/O	168	>518
spinner32	1.4M	248K	> 5.6	T/O	335	>260
rotate32	827K	240K	> 3.4	T/O	293	>297
vcrc32_8	20.5M	2.4M	> 8.5	T/O	3871	>23
am2901	20.8M	2.9M	> 7.2	T/O	20247	> 4.3
b12	4.2M	800K	> 5.3	T/O	T/O	–
vsa16a	11.1M	4.8M	> 2.3	T/O	T/O	–

"T/O" – timeout of 1 day, M/O – memory out of 512MB.

Table 3.1: Comparison of BDD-based VIS with POBDD-VIS on VIS-benchmarks for all circuits where VIS requires more than 250K BDD nodes. The time includes CPU time for simultaneous check of all properties of a given circuit

benchmark suite. The time includes cpu time for simultaneous check of all properties of a given circuit. The memory required is measured in terms of the cumulative peak live nodes for all BDDs that are maintained.

The first column shows the memory required when running VIS. The second column lists the memory for our trace-centric partitioning method, and the next column shows the corresponding space gain. In the runtime comparison, the first column shows time taken by VIS in seconds. The second column lists the effect of our improved partitioning method when combined with a trace-centric approach. The last column shows the time gain of the trace centric partitioning over VIS.

In both time as well as space, the trace-centric partitioning approach provides dramatic gains. Our approach completes all circuits except two in the VIS Verilog benchmark suite. Notably, it verified six circuits where the VIS failed to finish. For some circuits such as *palu, s1269b, am2910, ball*, verification was completed by the very first POBDD trace of 100k nodes. In most cases, there is an order of magnitude

or more improvement in both time as well as in space.

3.7.3 Comparison with Non-partitioned Reachability Analysis

In identical format, Table 3.2 compares our method with Vis-2.0 on formal reach-

Circuit	Space (BDD nodes)			Time (sec)		
	Vis	Proposed	Gain	Vis	Proposed	Gain
s1269	2.4M	31K	77	2305	28	82
s3330	1.3M	263K	4.9	**T/O**	948	>92
prolog	976K	138K	7.1	**T/O**	592	>147
s4863	438K	264K	1.7	1382	1717	0.8
s1423	3.3M	1.7M	1.9	**T/O**	**T/O**	–
	States covered			2e+10	1e+13	419
	Time for 2.3e+10 states			87000	633	137

Table 3.2: Comparison of BDD-VIS with POBDD-VIS on ISCAS89 benchmark

ability analysis for some ISCAS89 benchmark circuits. The proposed POBDD implementation works better in first three circuits. In s4863, the time required in computing traces made the method slightly slower than VIS.

We also found that in s1423 our method covers more states than VIS does in the same time. Also notice that the partitioned approach covers the same number of states as VIS in a small fraction of the time required.

3.7.4 Comparison with Static Partitioning

Figure 3.7 compares the run time of our trace-centric POBDD method to the static POBDD approach of [NIJ+97]. The initial number of partitions for both methods were kept identical to have a level playing field. The graph shows the normalized runtimes by size of the bar. The actual runtime in seconds is shown at the top of each bar. One can observe that the proposed method noticeably improves on the static partitioning scheme for most of the circuits, especially when the time taken is large. In once case that could not be completed by the static partitioning approach, the current method is able to complete reachability.

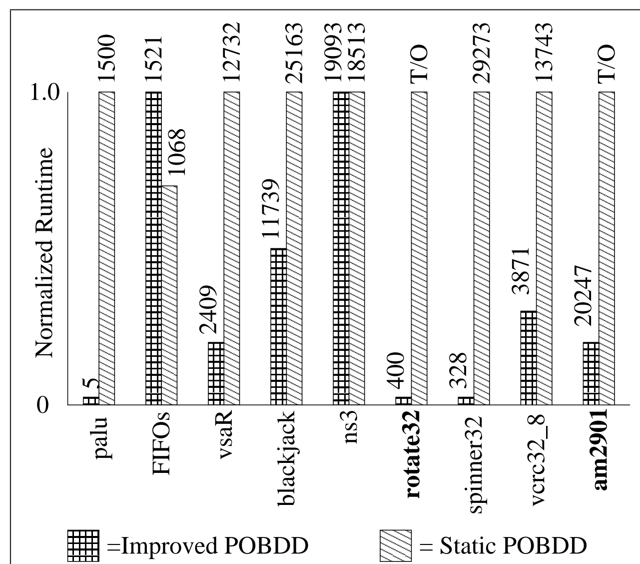

Figure 3.7: Comparison of Dynamic trace-centric and Static Partitioning Approaches on all Large designs (time>1000s) of VIS-benchmarks. The actual runtime in seconds is shown at the top of each bar.

3.7.5 Comparison with and without traces

Table 3.3 compares proposed POBDD approach with and without traces. The pro-

Circuit	Time (sec)		
	Vis	Proposed (without trace)	Proposed (with trace)
am2910	M/O	222	222
ball	T/O	168	168
spinner32	T/O	9305	335
rotate32	T/O	5537	293
vcrc32_8	T/O	51576	3871
am2901	T/O	T/O	20247
b12	T/O	T/O	T/O
vsa16a	T/O	T/O	T/O

Table 3.3: Comparison of Proposed POBDD with trace and without trace on all designs where VIS runs out of time or memory.

posed POBDD with trace finishes one more circuit that the method without trace. It has noticeable improvements on three other circuits, viz., *spinner32, rotate32, vcrc32_8* . Table 3.3 shows that the partitioning methodology is definitely improved

Circuits	Time in seconds Run 1	Run 2	Run 3	Run 4	Run 5
ball_7	3433.4	1808.6	2941.9	1692.4	2183.5
palu	222.8	1093.4	671.7	522.1	908.7
vsa16a_1	3006.3	4169.5	2757.8	2251.9	1505.6
vsa16a_2	1021.1	3997.9	6435.8	3496.7	3624.8
vsa16a_3	2832.5	3943.4	3196.8	2479.6	1766.0
vsaR	6297.9	8446.0	–	–	–
am2910	–	–	–	–	–
ball_6	–	–	–	–	–
blackjack	–	–	–	–	–
rotate32	–	–	–	–	–
spinner32	–	–	–	–	–

"–" indicates a timeout of 10,000s

Table 3.4: BDD-based falsification is moderately sensitive to parameters, all runs are BFS.

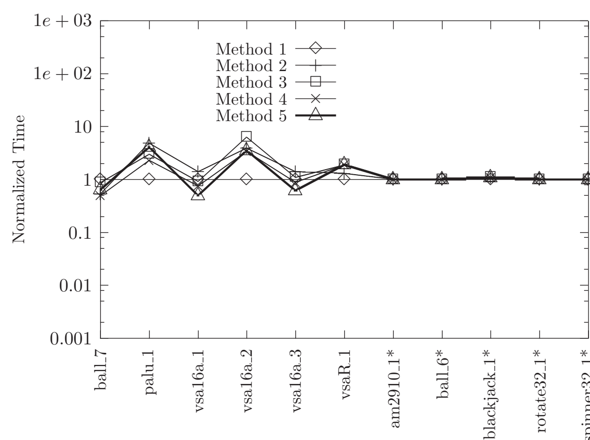

Figure 3.8: BDD-based falsification is moderately sensitive to parameters, all runs are BFS.

by using short traces.

3.7.6 Multiple traversals smooth out instability

We show the variations in time required for falsification using OBDDs and POBDDs when critical BDD parameters vary over a small number of runs. Note that a method using POBDDs is receptive to a larger set of parameters, many of which are not available to methods using OBDDs. Therefore, it is difficult to compare such

	Time in seconds				
Circuits	Run 1	Run 2	Run 3	Run 4	Run 5
ball_7	20.5	2244.9	1797.8	99.5	880.2
palu	461.0	1.2	1.0	16.7	100.0
vsa16a_1	–	–	7289.1	–	529.7
vsa16a_2	–	–	–	–	794.0
vsa16a_3	–	–	7577.9	–	531.6
vsaR	–	946.2	302.9	–	–
am2910	8165.9	–	–	156.6	699.8
ball_6	181.3	3580.6	–	5467.7	3542.6
blackjack	–	204.2	306.8	959.7	2259.8
rotate32	1588.9	–	–	569.2	–
spinner32	8831.6	–	–	1971.7	–

"–" indicates a timeout of 10,000s

Table 3.5: POBDD-based falsification is much more sensitive to parameters, runs differ in state traversal.

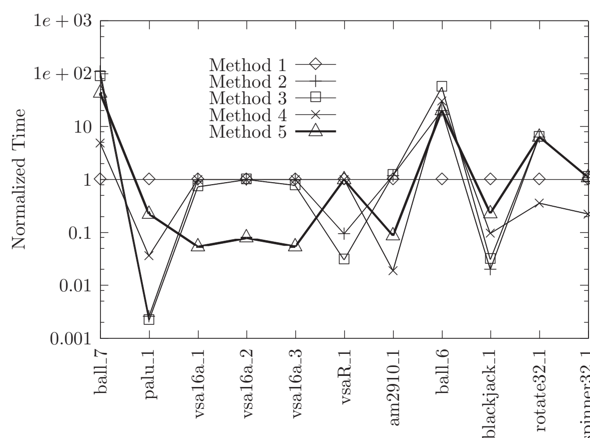

Figure 3.9: POBDD-based falsification is much more sensitive to parameters, runs differ in state traversal.

methods with respect to their parameter sensitivity. However, a larger number of runs would create a large amount of data, compromising the quality of presentation and making it harder to analyze. So we analyze an equal and small number of runs for both methods. We now describe in detail our experiments.

Circuits	Time in seconds				
	Sequential		Parallel		
	OBDD	POBDD	OBDD	POBDD	Gain
ball_7	3433.4	20.5	1692.4	20.5	82
palu	222.8	461.0	222.8	1.0	222
vsa16a_1	3006.3	–	1505.6	529.7	2.8
vsa16a_2	1021.1	–	1021.1	794.0	1.3
vsa16a_3	2832.5	–	1766.0	531.6	3.3
vsaR	6297.9	–	6297.9	302.9	20
am2910	–	8165.9	–	156.6	∞
ball_6	–	181.3	–	181.3	∞
blackjack	–	–	–	204.2	∞
rotate32	–	1588.9	–	569.2	∞
spinner32	–	8831.6	–	1971.7	∞

"–" indicates a timeout of 10,000s, ∞ is gain over timeout

Table 3.6: Comparison of time for falsification. The columns titled sequential refer, resp., to the baseline runs of Fig 3.8 and 3.9. The columns titled parallel refer, resp., to performing five traversals in parallel using OBDDs and POBDDs.

Experiments using OBDDs

A critical parameter that is easily changeable in OBDD-based reachability is the cluster size of the partitioned transition relation. We vary this from 100 to 5000, specifically, we use 100, 200, 500, 2500, 5000. The baseline setting, to which results of the remaining methods are compared for variation, has a cluster size of 2500 which is approximately the average of the extremes. Another parameter is the threshold for introduction of temporary variables but it was not varied in our runs. Recall that despite variation in parameters, each traversal using OBDDs remains strictly BFS.

Experiments using POBDDs

The order in which states are traversed using POBDDs critically varies with parameter choices. There are two different sets of parameters. The first set is parameters for data structure representation, namely the partitioning variables, number of partitions, etc. The parameters in the second set relate to reachability, *e.g.*, the cluster

Circuits	BDD Peak Node par-OBDD	par-POBDD	Gain
ball_7	828265	234508	3.5
palu	961826	313044	3.1
vsa16a_1	1128056	507706	2.2
vsa16a_2	1726586	478170	3.6
vsa16a_3	1128056	914098	1.2
vsaR	5687387	323640	17.6
am2910	>2817298	411963	> 6.8
ball_6	>2391459	234508	> 10.2
blackjack	>1764404	312803	> 5.6
rotate32	>3856029	325989	> 11.8
spinner32	>4698621	440549	> 10.7

">" refers to gain over timeout

Table 3.7: The largest BDD created during parallel falsification

size of the Transition Relation (TR), the schedule for processing the partitions, when to initiate partitioning, dynamic partitioning thresholds, etc. To limit POBDDs to the same number of runs, *i.e.* 5, as OBDDs, given such large number of choices, we severely limit the number of parameters varied.[1] We only vary the choice of partitioning variables and the cluster size of the transition relation. For each run, we use four partitioning variables to generate 16 initial partitions.

For the first three runs, we fix the reachability parameters, *i.e.*, the TR cluster size is set to the previous baseline value of 2500. We try the following choices of partitioning variables:

- The first few variables in BDD variable ordering list
- From the transition relation analysis as per [NIJ+97].
- From a similar analysis but using the state set graph.

We do not vary any other partitioning parameters. For the last two runs, we fix all partitioning parameters and change reachability parameters. We use the first

[1]The fluctuations in performance by varying all the parameters can be quite drastic. Our intention is not to dwell on the sensitivity of POBDDs. Rather it is to show that the state traversal order can be significantly varied from BFS, thus producing dramatic variation in performance.

partitioning choice above and the cluster sizes 200 and 500, skipping the extremes in cluster size.

Analysis of experiments

Table3.4 shows the sensitivity of time taken for reachability analyses to 5 settings of parameters using OBDDs. Figure 3.8 depicts this data graphically. Table 3.5 and Figure 3.9 show the corresponding parameter sensitivity of time taken by 5 different settings of partitioned OBDDs in tabular and graphical form, respectively.

We see that the OBDD results only moderately fluctuate with change in parameter values. In the case of POBDDs, the variations are often many orders of magnitude. Clearly partitioning is very sensitive to the choice of its settings.

More importantly, no single choice of parameter settings is able to complete all falsification properties in the VIS benchmark suite. Each setting is essential for one or more circuits. For some circuits, none of the OBDD-based runs finds the error. However, the POBDD-based runs, when considered together, are able to falsify all the properties.

POBDD runs also show much larger range in their behavior. Thus the need for an automatic method which tries various settings in parallel and rejects the inefficient cases is self-evident. We implement such an automatic method that makes multiple traversals in parallel and when any one finishes, terminates the rest. We apply this method to falsification based on OBDDs as well as POBDDs.

In Table 3.6, we show the time for the baseline method and for the parallelized method using each of OBDDs and POBDDs. Observe that only the method using POBDD-based multiple traversals is able to falsify all the properties.

In Table 3.7, we show that the largest BDD created during multiple parallel

traversals using POBDDs can be much smaller than when using OBDDs, especially on the circuits that the OBDD-based method does not complete.

3.8 Conclusions

We have discussed an efficient methodology for improving difficult instances of reachability based verification using the approach of state space partitioning. We have investigated relevant problems posed in creating a partitioned data structure during BDD-based verification, and provided efficient and practical algorithms for the same.

We have also addressed the issue of instability in BDD-based approaches where parameters are seldom found to work well uniformly. We developed a trace-centric approach to automatic selection of such parameters. The resulting method dramatically improves the space and run time, often from one to three orders of magnitude, on various public-domain benchmark circuits that are otherwise known to be difficult.

It is found that methods based on a monolithic representation of the state sets often encountered space explosion early on in the computation, after which they could not make much progress due to memory limitations. However, the trace-centric partitioning method scaled well, and could finish most circuits in the VIS Verilog benchmark suite. It was also able to reduce the extreme sensitivity of BDD-based verification.

Chapter 4

On Partitioning and Model Checking

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 [CE81] emphasized an enumerative implementation, while a symbolic technique using BDDs [Bry86] was suggested by McMillan [McM93]. Symbolic model checking can exhaustively cover the state space when handling small designs but can-

not handle industrial sized designs, which can often be orders of magnitude larger. The main issue is that 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.[JBFA92], and extended by Narayan, et.al.[NJFSV96].

A partitioned approach clearly has advantages over a non-partitioned approach in its ability to handle larger state sets. A big obstacle in the use of partitioned data structures to store state sets during reachability and model checking is the number and frequency of image computation operations that need to be performed across partition boundaries. Each partition can be thought of as being the *owner* of a set of states. Transitions from each partition naturally comprise of two components - ones that are wholly local to individual partitions, and ones that span multiple partitions. Correspondingly, the computed image comprises of a *local* component and a *cross-over* component. The states in the local component can be computed wholly within individual partition. On the other hand, the states in the cross-over component arise from state transitions that originate at a state in one partition and terminate at a state in another, thus, "crossing over" into the destination partition. Computing cross-over component of the image is often significantly more expensive than the local component for various reasons as will be explained later. A simple combination of partitioning with the classical model checking algorithm [CE81, McM93], for instance the distributed model checking algorithm of [GHS01], performs repeated exact images, thus incurring this cost of numerous cross-over computations and communications between partitions.

In this work, we propose an alternative *piece-wise* algorithm for model checking CTL formulae that makes effective use of state partitioning by postponing cross-

over image computations.

4.0.1 Contributions of this work

This is the first algorithm for full CTL model checking to explicitly recognize the distinction between the local and cross-over image computations and to effectively exploit their separability for image as well as fix-point computations. When the design is defective and is falsified, this algorithm discovers bugs faster, by virtue of computations being localized to individual partitions. Even when the design is correct and is verified, this algorithm converges after fewer cross-over image computations.

This method can be thought of as a form of "localized BFS" of the state space. The search is localized to individual partitions in stages and global synchronization is performed between stages. This technique is often able to discover states faster and in the case of debugging, manifests directly as a reduction in the error detection time.

Further, this approach reduces the number of cross-over image computations performed and provably has no more cross-over images than the standard BFS-based algorithm. This reduces the overhead involved in activities like communication between partitions, BDD variable re-ordering and rebuilding in different partitions, etc.

Additionally, this algorithm reduces the frequency of cross-over image computations as a fraction of the total number of images computed. This reduces the dependence between partitions and makes this approach very attractive for parallel or distributed model checking.

We find empirically that during state space traversal, if each partition re-

quires many steps of image computation to reach a local fix-point, then the proposed algorithm shows significant gain which is proportional to the depth of the fix-point.

4.0.2 Organization of this chapter

This chapter is organized as follows. In the next two sections, we give the partitioned model checking algorithm. We first show a naive partitioned model checking algorithm and then a modified algorithm designed to localize computation by postponing cross-over image computations. Section 4.4 has our experimental results documenting the increased efficiency of our technique. Finally, section 4.5 gives conclusions and directions for further research.

4.1 Partitioned Model Checking: A simple algorithm

We now examine the classical model checking algorithm modified for a partitioned representation of the state sets. This is a simple algorithm, along the lines of the distributed model checking algorithm of [GHS01].

First, a word on our terminology. Each partition *owns* states that are in its subspace, as defined by its window function. Conversely, such states *belong* to the partition. We say that a partition performs operations on sets that it owns. The result of such operations may lie in a different partition and may then need to be transferred. It is important to make this distinction between the partition where the operation is performed and the partition to whom the result finally belongs, because they may obey different variable orders. Further, in case of a parallel implementation, such partitions may be physically on different processors. For now, we ignore this detail.

It suffices to consider the Boolean connectives and the existential temporal

operators EX, EG and EU, as they form a basis set for CTL. Model checking of boolean connectives is well-known for the partitioned approach [NJFSV96]. It is noteworthy that all boolean operations – conjunction, disjunction as well as negation – are local to individual partitions and involve no interactions among them. This is an important consequence of window-based partitioning.

4.1.1 Partitioned Image Computation

The computation of EXp can be done using the backward image computation.

State space partitioning into n disjoint parts induces a partitioning of the transition relation T into n^2 parts T_{jk} consisting of transitions from a state in partition j to a state in partition k. We can derive T_{jk} by conjoining T with the respective window functions as $T_{jk}(s, s', i) = w_j(s)w_k(s')T(s, s', i)$. Thus we can express the transition relation $T(s, s', i) = \bigvee_j \bigvee_k T_{jk}(s, s', i)$.

Figure 1 shows how to calculate EXp separately on each partition. Here the

```
ComputeEX(Set R, Transition Relation T) {
  foreach (partition j)
    foreach (partition k)
      PreImg^{jk}(s) = ∃_{s',i}[T_{jk}(s, s', i) ∧ R_k(s')]
      reorder BDD PreImg^{jk}(s) from partition order k to order j
    end for
    N_j(s) = ⋁_k PreImg^{jk}(s)
  end for
  output N
}
```

Figure 4.1: Backwards Image Computation with Partitioning

set R_j is the set of states that represent p in partition j, and the set N_j represents EXp in partition j which are computed by application of the transition relation $T_{jk}(s, s', i)$.

4.1.2 Partitioned Computation of fix-points

$E(pUq)$ (resp. EGp) has been traditionally represented as the least (resp. greatest) fix-point of the operator $\tau(Z) = q \vee (p \wedge EXZ)$ (resp. $\tau(Z) = p \wedge EXZ$) and can therefore be computed as a fix-point. The classical fix-point algorithms for $E(pUq)$ and EGp as modified to use a partitioned data structure are illustrated in Fig. 2. Notice that these rely on performing exact image computation and therefore perform

```
computeEU(p, q) {
  S := q and S.old := φ
  repeat
    S.old := S
    temp := computeEX(S)
    forall (partitions j)
      S_j := q_j ∨ (p_j ∧ temp_j)
    end for
  until(S = S.old)
  output S
}
```

a) Least fix-point, $E(pUq)$

```
computeEG(p) {
  S := p
  repeat
    S.old := S
    temp := computeEX(S)
    forall (partitions j)
      S_j := p_j ∧ temp_j
    end for
  until(S = S.old)
  output S
}
```

b) Greatest fix-point, EGp

Figure 4.2: Classical Model Checking of Fix-points in presence of Partitioning

one set of cross-over images in each iteration.

We now look at the main cause of inefficiency in this algorithm.

4.1.3 Issues: Local and cross-over images

To compute the pre-image, n^2 computations using $T_{jk}(R_k)$ need to be performed, followed by n disjunctions as shown. It is natural that the pre-image of states in partition k under the transitions leading to each partition j, i.e. $T_{jk} \wedge (R_k)$, is performed in partition k. Each partition k computes states that potentially belong to every other partition. Subsequently the disjunction to obtain the pre-image lying with partition j, i.e. the computation of $\bigvee_k PreImg^{jk}$, is performed by partition j. As a consequence, the set $PreImg^{jk}$ needs to be transferred from partition k

to partition j, when j and k differ. Thus a single computation of EXp from p comprises of n^2 image computations and n^2 transfers between partitions.

Of these, the n computations where $j = k$ are completely local to individual partitions, namely to partition j. We call the union of the states resulting from these computations as EX_l, the *local component* of image.

We call the other $n^2 - n$ computations as *cross-over image* computations, and denote the union of the those states by EX_c.

Performing too many *Cross-over image computation is inefficient* for the following reasons. First, a quadratic number of image computations need to be performed as above and the BDDs need to be accessed from every partition. 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. Then, the BDD variable order of the computed image set must be changed from the order of the source partition to that of each of its target partitions, before the new states can be added to the reached set in the target. Reordering large BDDs can be very expensive. Finally, there may also be other overhead, for eg., in the case of a distributed implementation there is the overhead of physically transmitting a large number of such BDDs over the network.

Thus the execution time for cross-over image computation is significantly greater than that for performing local image computation.

We now present a model checking algorithm that localizes computation to individual partitions by postponing the cross-over image computations.

4.2 Improving Partitioned Model Checking

4.2.1 Image Computation

We find that performing the cross-over images one partition at a time is memory intensive and often the intermediate BDDs get very large for many examples. Therefore, we advocate performing these cross-over image computations from each partition into many partitions at a time. The partitioned approach easily extends to distributed and parallel computing environments and our improvements are expected to scale accordingly. However, for now, we only address the case of verification using uniprocessor systems.

In order to perform cross-over images efficiently, we maintain a *transfer manager* M. Given the set p, in order to compute EXp, each partition i computes the image $T_{ii}(p_i)$ which it keeps locally and the set of unowned states $U_i = T_{i\bar{i}}(p_i)$ which is communicated to the manager M. M uses the window functions w_j to calculate the sets $S_j = \bigvee_{i \neq j} U_i * w_j$ and then transmits the states S_j to partition j. Thus EXp is computed by doing $2n$ image computations and $2n$ transfers between partitions, although the number of re-orderings remains n^2.

It should be mentioned here that in a multiprocessor environment, such a manager can become a bottleneck, and should perhaps be dispensed with. But the point is that, in each partition, only a constant number of image computations be performed, rather than a number linear in the number of partitions. Thus the total number of image computations is linear rather than quadratic in the number of partitions.

We call the fraction $T_{ii}(p_i)$ that is computed locally using T_{ii} as the ith projection of the local image EX_l. The rest of the images comprise the cross-over image EX_c. The algorithms to compute EX_l and EX_c are shown in Figure 4.3.

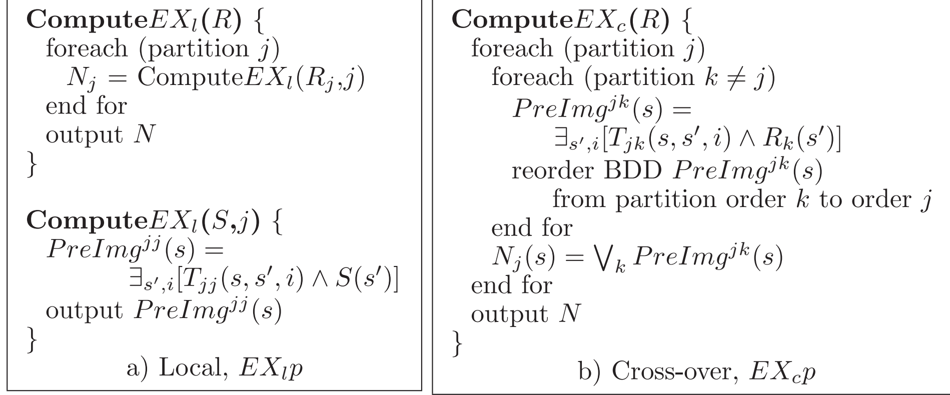

Figure 4.3: Local and Cross-over Components of Image Computation with Partitioning

4.2.2 Fix-point computations

The main idea for model checking fix-points is that the computations can be significantly localized to individual partitions by postponing the cross-over image computations EX_c, which are then aggregated and performed infrequently. Accordingly, we define the fix-point operators in terms of two operations – local image computations EX_l and cross-over image computations EX_c, rather than the classical definition in terms of just the image computation operation, EX.

The algorithm for computing $E(pUq)$ (resp. EGp) is shown in Figure 4.4. The key idea is to create an under-approximation (resp. over-approximation) to EXp, which can be wholly calculated locally within individual partitions, so that the least (resp. greatest) fix-point computation can be localized as much as possible.

Definition 2 *Each iteration of the outermost repeat-until loop in Algo.4.2 (shown in Fig. 4.2) and Algo.4.4 (resp. Fig. 4.4) is called a* phase *of the respective algorithm.*

From this definition, we note the following.

Lemma 4.2.1 *Every phase has one and only one cross-over image.*

computeEU(p, q) {	**computeEG**(p) {
$S := q$	$S := p$
$S.old := \phi$	$Border := p \wedge EX_c(S)$
repeat	repeat
$S.old := S$	$S.old := S$
forall (partitions j)	forall (partitions j)
repeat	repeat
$S_j.old := S_j$	$S_j.old := S_j$
$S_j := S_j \vee (p_j \wedge EX_l(S_j, j))$	$S_j := p_j \wedge (EX_l(S_j, j) \vee Border_j)$
until$(S_j = S_j.old)$	until$(S_j == S_j.old)$
end for	end for
$S := S \vee (p \wedge EX_c(S))$	$Border := p \wedge EX_c(S)$
until$(S = S.old)$	until$(S == S.old)$
output S	output S
}	}
a) Least fix-point, $E(pUq)$	b) Greatest fix-point, EGp

Figure 4.4: Fix-point Computations localized by postponing cross-over images

We will show that Algo.4.4 terminates with the correct result and that the number of its phases is at most the number of phases in Algo.4.2. Since each such phase has precisely one cross-over image computation, we have that the number of cross-over images computed by the new algorithm is, in the worst case, no more than that for the existing algorithm.

4.2.3 Correctness of least fix-point algorithm

Theorem 4.2.2 *a)[ISS$^+$03] The procedure computeEU of Fig 4.4a, 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 $E(pUq)$.*
b) The number of its phases does not exceed the number of phases for Algo.4.2a.

Proof: Let the set of states S at the end of the ith phase be called S^i. The termination is guaranteed because the sequence of sets S^i is strictly monotonic increasing. We first show the soundness of Algo.4.4a, i.e., at all times $S \models E(pUq)$. We show this by induction on the sets S^k. This clearly holds for any state in S^0,

since every state in S^0 satisfies q and therefore $E(pUq)$. Assume 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 in the local image computation EX_l for some partition j or in the cross-over image computations EX_c. In either case, $s \models p$. It remains to show that s is the predecessor of a state that models $E(pUq)$. In the first case, such a state is in the same partition as s and in the second case, such a state exists in partition k such that s was added in the cross-over image computation from k to j. Thus in either case, s models $EX(E(pUq))$. Consequently, Algo.3a is sound.

Next, we show completeness, i.e., that every state of $E(pUq)$ is indeed in set S. For every state $s \models E(pUq)$, there exists a sequence of states $s_0, s_1, \ldots, s_k$ that has the smallest length $k \geq 0$ such that $s_0 = s$, $s_k \models q$, $\forall i < k : s_i \models p$ and $\forall i < k : s_i \in EX(s_{i+1})$. This sequence of states is called a *witness* for the inclusion of s in $E(pUq)$, and k is its *length*. 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, 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 $E(pUq)$. 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 $s \in EX(s_1) \subseteq EX(S^j)$. From the definition of S^j and Algo.4.4a, we have that $s \in S^{j+1}$, whereby $T^{i+1} \subseteq S^{j+1} \subseteq S$. By induction, this gives us $E(pUq) \subseteq S$.

This proves that Algo.4.4a terminates with the set $S = E(pUq)$. Notice that the set of states at the end of the kth phase of Algo.4.2a is precisely T^i. As above, $\forall i, T^{i+1} \subseteq S^{j+1} \subseteq S^{i+1}$. Hence Algo.4.4a has at most as many phases as Algo.4.2a. ■

4.2.4 Construction of the Greatest fix-point algorithm

Before proving an analogous result for the greatest fix-point operator EGp, we briefly motivate its construction. As $EX_l p$ is a subset of EXp, the result of localizing the computation by performing repeated EX_l operations yields an under-approximation at every step. Since the greatest fix-point operator converges by a sequence of monotonically decreasing sets, under-approximation leads to some states being pruned too early and being lost for ever. States that may be incorrectly pruned early in the computation of EG comprises of states, each of which lies in a different partition from its predecessor, and can therefore be discovered only by performing the operation EX_c, which is the expensive component of image computation.

Algo.4.4b compensates for this by maintaining a set *Border*, which is the set of all states which have a successor in a different partition than themselves. This is, clearly an over-approximation to EX_c in each partition. This superset of EX_c is used to calculate a superset of EX at every image. This *Border* is updated only once in each phase, when each partition has reached a fix-point with respect to local images EX_l. These over-approximations are monotonically decreasing, and so the computed set eventually converges to the desired set EG.

We now prove the following theorem.

4.2.5 Correctness of greatest fix-point algorithm

Theorem 4.2.3 *a) The procedure computeEG of Fig 4.4b, given the set of states corresponding to formula p as input, terminates with the output S being precisely the set of states that model the formula EGp.*
b) The number of its phases does not exceed the number of phases for Algo.4.2b.

Proof: Again, let the set of states S at the end of the ith phase be called S^i.

The termination is guaranteed because the sequence of sets S^i is strictly monotonic decreasing.

We first show the soundness of Algo.4.4b, i.e., the algorithm only deletes states which do not satisfy EGp. Note that a state can be deleted only in the two circumstances. The first is if it does not satisfy p and is deleted in the very beginning. We can therefore assume that all states under consideration satisfy p. The second way a state may be deleted is during some phase, when it is not a predecessor to any state in its own partition, and it is not on the Border, i.e., it has been determined previously that this state is not a predecessor to any state in another partition. Thus all successors to such a state satisfy $\neg p$, and therefore any deleted state is not in EGp.

Next we show completeness, i.e., the algorithm deletes all states that do not satisfy EGp. Consider a state $s \not\models EGp$. Then there exists a sequence of states $s_0, s_1, \ldots, s_k$, which is cycle-free that has the greatest length $k \geq 0$ such that $s_0 = s$, $s_k \models \neg p$, $\forall i < k : s_i \models p$ and $\forall i < k : s_i \in EX(s_{i+1})$. This sequence of states is called a *witness* for the exclusion of s from EGp, and k is its *length*. Now, let T^k be the set of states whose exclusion from EGp is witnessed by a longest cycle-free path of length at most k. We prove by induction on k that $T^k \cap S^k = \phi$. In the base case, this trivially holds because $T^0 = \neg q$ and $S^0 = q$. Now, assume that $T^i \cap S^i = \phi$. 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 exclusion from EGp. The sequence $s_1, \ldots, s_{i+1}$ is a witness for s_1, therefore $s_1 \in T^i$, and therefore $s_1 \notin S^i$. In particular, there exists a smallest j so that s_1 was deleted in the jth stage of the algorithm. Two cases arise, either both s_0 and s_1 are in the same partition or they are in different partitions. If they are in the same partition, then s_0 is deleted in the jth stage also when a fix-point is computed

locally in that partition. If they are in different partitions, then s_0 is in the border set for its partition, and is deleted from this border set at the end of the jth stage because its last successor s_1 is deleted and no other successors can exist because this is the longest witness. Therefore s is deleted in the $j+1^{th}$ stage, as required to be proved.

This proves that Algo.4.4b terminates with the set EGp. Notice that the set of states T^i is precisely the set of states deleted in phase i of Algo.4.2b. As above, states in T_i have all been deleted by the end of i phases of the algorithm. Hence Algo.4.4b has at most as many phases as Algo.4.2b. ■

4.2.6 Analysis

In the worst case, Algo.4.4 requires at most as many phases as Algo.4.2. However, in practice, Algo.4.4 outperforms Algo.4.2, because when computing the least (resp greatest) fix-point by localizing computation to individual partitions, Algo.4.4 often discovers (resp. prunes) many more states than when performing just one image computation in each phase. Thus the postponement of cross-over images affords a significant benefit in overall faster convergence of the model checking algorithm, often reducing the number of phases.

We now analyze the benefit of reducing the number of cross-over images. Consider a simple model where the number of image computations performed is the same in each partition, say n. Further, assume the time for computing EX_l is L and that for EX_c is C.

Thus Algo.4.2 performs n phases, each with one computation of EX_l and EX_c, and incurs a total time $C_{old} = n * (L + C)$. Algo.4.4 needs potentially fewer phases, say $m \leq n$. Each such phase has one EX_c computation and a number of

EX_l computations, for concreteness say there are $k \geq 1$ of them. This gives a total time $C_{new} = m * (k * L + C)$.

Recall from Section 4.1.1 that $C >> L$. Thus for reasonable k, the reduction in the number of cross-over images is directly reflected in the reduction of the total model checking time. Further, in the best case scenario, when $m * k = n$, the reduction may be by as much as a factor of k.

In practice, a significant gain is observed, as borne out by the experimental results as described in the next section.

4.3 Experimental Results

4.3.1 Experimental Setup and Benchmarks

We implemented the algorithm of Fig. 3 on top of the CUDD package for BDDs using the VIS-2.0 verification environment [BHSV+96], which is a state-of-the-art public domain BDD-based formal verification package. We have chosen VIS for its Verilog support and its powerful BDD-package (i.e. CUDD [Som01]). As our techniques affect only the BDD-data structures and algorithms, they can – with moderate effort – be implemented in other packages as well. These techniques work with any method of image computation; all experiments here are conducted using the so-called IWLS95 [RAB+95] method in VIS.

We use the partitioning scheme detailed in [SIJ+04] for performing reachability analysis. Once the reachable states are computed, the model checking algorithms use the same partitions created during reachability analysis.

We chose all the public domain circuits and their model checking properties from the VIS-Verilog [vis] benchmark suite. For sake of brevity, results are omitted for some of the smaller examples.

Circuit_ Property	Cross-over images Number Old	New	Time (s) Old	New	Speedup	Model Checking time(sec) Old	New
bpbs	4	1	24	1	24	398	313
gcd_1	15	7	19.11	.7	27	68.97	108.07
gcd_2	15	7	18.27	.16	114	27.56	9.06
gcd_3	10	8	37.13	4.29	8.6	134.65	56.32
gcd_4	10	8	37.41	3.44	11	108.76	42.11
gcd_5	11	9	37.13	46.3	0.8	107.31	92.19
gcd_6	12	9	42.96	3.79	11	121.66	53.69
gcd_7	13	9	42.98	3.99	11	132.7	50.51
gcd_8	14	9	35.68	1.41	25	128.04	48.94
gcd_9	15	9	31.77	0.91	35	119.63	48.04
gcd_10	16	9	28.72	0.57	50	111.89	46.47
ghg	9367	6	166.12	0.15	1107	280.75	27.31
idu32_1	3	3	12.35	89.96	0.13	294.49	406.61
idu32_2	2	2	0.07	0.02	3.5	0.12	0.03
idu32_3	3	3	0.07	0.02	3.5	0.06	0.02
idu32_4	8	4	0.61	0.02	30	0.82	0.1
idu32_5	8	4	0.61	0.03	20	0.83	0.11
idu32_6	7	2	0.83	0.02	41	1.27	0.1
idu32_7	7	4	1.31	0.02	65	2.11	0.22
idu32_8	8	4	13.63	0.03	454	14.15	0.28
idu32_9	8	4	0.38	0.02	19	0.52	0.05
idu32_10	23	9	0.58	0.04	14	0.8	0.15
luckySeven	64	35	80.12	55.89	1.4	114.64	82.03
nosel	7	3	106.01	10.2	10	270.18	130.87
product	1	1	3	3	1	1798	418
s1269b_1	1	1	9.42	0.01	942	15	13
s1269b_2	8	8	67	1.01	67	93	1
soap_44	53	5	592.09	1.2	493	714.81	28.24
soap_45	80	8	106.76	1.86	57	224.19	104.11
soap_46	53	5	92.9	1.14	81	187.79	28.76
soap_47	52	5	41.87	1.11	37	94.89	31.83
soap_48	60	5	42.3	0.76	55	98.91	56.41
soap_49	79	9	94.68	1.61	58	207.18	73.78
soap_50	60	5	199.6	1.05	190	299.4	22.9
sppint2_1	5	4	86.26	45.06	1.9	100.39	58.26
sppint2_4	1	1	2.8	0.01	280	3.06	2.82
sppint2_5	7	3	4.4	0.01	440	4.94	0.75
sppint2_6	5	3	0.2	0.13	1.5	0.68	0.28
sppint2_7	16	6	4.23	0.7	6	24.66	2.27
sppint2_8	5	4	1.22	0.37	3.3	1.87	0.71
sppint2_9	14	6	1.29	0.74	1.7	5.01	1.81
sppint2_10	5	4	1.31	0.17	7.7	1.83	0.32
two	38	24	30.6	18.8	1.6	46	28
usb_phy_1	49	23	16	19	0.8	43	29
usb_phy_3	40	19	108.51	11.83	9	24.89	28.97
usb_phy_4	21	11	5.6	2.32	2.4	12.01	8.26
usb_phy_6	5	5	0.97	1.05	0.9	2	2.99
usb_phy_7	39	17	10.96	2.14	5	24.32	8.72

Table 4.1: Comparison of existing and proposed algorithms for partitioned model checking CTL properties on circuits in the VIS Verilog benchmark suite.
For each circuit and property, the first pair of columns shows the number of inter-partition cross-over images performed by the two methods, the second set shows the time required for these cross-over images, and the speedup achieved by the new method and the final set shows the total model checking time.

4.3.2 Results and Analysis

We notice that crossover image computation is indeed a bottleneck in verification. On a uniprocessor machine, in the VIS-Verilog benchmark suite, there are examples where the program runs out of memory while performing the crossover images. Thus a reduction in the number and frequency of such cross-over images is critical for the full utilization of computing resources in a multi-processor environment.

The run-times for our sequential implementation are shown in Table 1. The first column of Table 1 has the name of the circuit and the property being checked. This is followed by the data for cross-over image computation. Firstly, the number of cross-over images is shown for the Naive algorithm, labeled *Old* and for our proposed algorithm, labeled *New*. The next two columns show the respective time taken. This is followed by the speedup achieved by the proposed algorithm over the older one. The last two columns show the total time taken by the model checking, after reachability has finished.

Experimentally, the proposed algorithm converges faster, both in terms of total time, as well as in terms of number of expensive cross-over image computations that are performed. Further, the time taken by cross-over images as a percentage of total time is reduced. These are demonstrated in the table that follows. In numerous cases (e.g. s1269, soap, ghg, etc.), we find that total cross-over image time is reduced by two orders of magnitude or more.

Using the data in Table 1, we can compare the total time taken for model checking by the two methods. Notice that the proposed algorithm reduces, often dramatically, the number of cross-over images and the proportion of the total time that is spent in doing them. In almost all the examples, this leads to a direct improvement in the total time.

In the worst case, the proposed method would be identical to the naive one, with strict alternation between localized (EX_l) and cross-over (EX_c) image operations in every fix-point calculation. In practice, this is not very likely because it happens only in those cases where every "path" corresponding to a formula comprises of states *each* of which lies in a different partition from its predecessor. Notice that there are only a couple of examples in the entire benchmark set of Table 1 which exhibit such behavior.

4.4 Conclusions

We have presented a model checking algorithm in the presence of state space partitioning, that aggregates and postpones cross-over image computations, allowing for significant localization of image computations. This is also found in practice to reduce the number of cross-over images required for convergence in fix-point computations.

If during state space traversal, each partition requires many steps of image computation to reach a local fix-point, then the proposed algorithm shows significant gain, proportional to the depth of the fix-point.

We believe this algorithm can be generalized to more expressive logics like the full μ-calculus with a few modifications.

Our experiments have been conducted on uniprocessor machines, but this algorithm can be easily parallelized and we believe its benefits would scale to an implementation in a multi-processor environment. A parallel form of the proposed algorithm can also provide better resource usage than existing distributed model checking algorithms.

Chapter 5

Partitioning for Bounded Model Checking

In this chapter, we explore a parallelization of Bounded Model Checking (henceforth, BMC) based on state space partitioning. The parallelization is accomplished by executing multiple instances of BMC independently from different seed states. These seed states are *deep* states, selected from the reachable states in different partitions. In this scheme, all processors work independently of each other, thus it is suitable for scaling verification to a grid-like network. Our experimental results demonstrate improvement over existing approaches, and show that the method can scale to a large network.

5.1 Introduction

Satisfiability based Bounded Model Checking (SAT-BMC) is able to explore the state space of larger designs by bounding the depth of exploration and successively increasing this bound [CBRZ01].

Due to notable improvements in the art of satisfiability-testing, SAT-BMC is now routinely applied to detect errors during property verification for many industrial designs [CFF+01, BLM01, BCRZ99, AKMM03].

SAT based BMC approaches are the preferred method for detecting error states that are not very deep. However, these techniques can become quite expensive when many time-frames are required to be analyzed. BDD based approaches are better choices for those "deep cases" where the image BDDs remain moderately small as constructing large BDDs for many image steps can be very expensive. These observations have been validated in a recently reported industrial case study [AKMM03]. Thus the class of problems which may require many steps of image analysis to detect the error, *but* where BDD sizes grow large, remain an attractive research target.

Since such techniques, running on a single CPU have clear limitations due to the limited computing power of their execution environment, researchers have suggested distributed approach towards BDD based model checking as well as parallel SAT solvers [FDH04, GGYA03]. However these methods remain inadequate as they essentially analyze the state space from a breadth-first centric point of view. The BDD based approaches also require communication between different processors in form of large BDDs which precludes these methods from using large parallel computing environments.

In this chapter we try an intuitive and attractive approach to address the above identified class of problems. Our approach is to create a method that can find various candidate deep states which can be *seeds* from which SAT-BMC can be run in *parallel* to explore the adjacent state space. Starting from such potential deep seed states, multiple BMC runs may be able to reach further deep states, and

locate errors, which may be out of reach for existing methods.

5.1.1 Our Approach

Our approach is to use BDD based methods, while the graph sizes are still under control, to generate such seed states. Note, for a few initial steps of reachability, rapid progress can be made using BDDs. However, as the graphs get larger, the BDD-based state space analysis becomes very expensive. We find that this can severely limit the capability of BDD-based approaches to seed BMC runs. Another problem worth a solution in the above mentioned hybrid approach is that given many seeds it is not clear as to which ones are more important and should be prioritized.

A practical solution to the above problems may significantly augment the state of the art in detecting deep errors, and is the focus of this work.

Generating Seed States: To control the size of BDDs using state space analysis we will use state-space partitioning. It has been empirically observed that partitioning provides the means to symbolically explore the state space *deeply* [SIJ+04]. In fact, if each subspace is thought of as a "direction", then reachability with partitioning is a localized-Breadth First Search (L-BFS) along each such direction. Deep states provided from such local BFS traversals can be used to provide initial seed states to subsequent BMC runs to help it explore regions that could not be explored before using popular SAT-BMC approaches.

Our idea appears useful on many circuits. However, a detailed analysis of BDD behavior suggested that this approach is still impractical. For many circuits, the BDD based analysis can be very expensive even with partitioning. Thus the time to generate deep seed states can be prohibitively large. Since the BDD run-

time is directly proportional to the size of the graphs, we will like to further limit graph sizes of each partition using massive under-approximations. Thus we will rely on an under-approximation based method on top of partitioned BDDs.

Using Seed States: In order to keep the graph size under control during reachability computation, one must make many partitions. However in such cases it is very difficult to know *a priori* as to which partitions provide the best seed states for bounded-model checking. We augment our ideas of combining Partitioning and BMC by generating multiple instances of BMC and run each such case in parallel on a grid of computers. This idea looks even more attractive when we consider that large computing grids are slowly becoming available in many computing environments [AK04].

In this following, we will not only show that this can be an attractive approach to discover deep bugs but also our approach indicates a novel way to use a large computing grid for a verification solution. Also, it suggests a non-traditional way to parallelize BMC.

5.1.2 Organization of this chapter

Although such deep cases are not routinely available in various public benchmark cases such as VIS or Texas97 benchmark suites, we have often encountered them in an industrial context. We consider (Section 5.5) many such cases and show that they are indeed solved more efficiently using our approach than by just using OBDDs, POBDDs, or BMC. In Section 5.2 we survey related work and in Section 5.3 we present terminology relevant to discuss our algorithm, which is presented in Section 5.4. The experimental setting is explained along with our results in Section 5.5,

and in Section 5.6 we present our conclusions.

5.2 Related Work

This work lies at the cross-roads of two bodies of work, namely hybrid techniques for smart simulation or efficient bug-finding and recent early efforts for performing verification in a parallel framework.

Our research is motivated by the recent success of BMC methods where using SAT-based or ATPG-based approaches it has been possible to process circuits that were hitherto not possible to be analyzed using BDDs. However, the techniques discussed in this work have a nature of an hybrid approach using multiple engines. Thus, they can easily use improvements in the individual technologies such as SAT or ATPG engines [MMZ+01, IPC03], or BMC formulation [CBRZ01]. Thus, a detailed comparison with such techniques is orthogonal to the objective of our work and correspondingly will not be further detailed.

The last few years have seen the emergence of a class of hybrid methods which employ a combination of formal engines and simulation in order to achieve a good trade-off between the capacity and scalability of simulation methods and the validation coverage of formal methods. The work by Yang and Dill [YD98] performs a pre-image computation from the target states to provide an enlarged target for simulation. The approach of [KMB99] proposes a guided search algorithm whereby states are assigned a probability of reaching the target state through an approximate state space traversal and explored in accordance with this probability. The **SIVA** tool [GAK99] performs best first search in a simulation environment, augmented with BDDs and SAT, with the hamming distance between the current and target state as the guiding cost function. A subsequent enhancement [YSA00] adds

automatically generated intermediate goals or "light-houses" to guide the simulator towards a deep target. The approach of [HSH+00] uses interleaved runs of simulation, BDD-based symbolic simulation and ATPG-based BMC to maximize the state coverage over a set of interesting signals.

In contrast to the above "bug hunting" approaches another class of techniques employ a combination of formal engines mainly for the purpose of verifying properties (possibly bounded depth properties). For example, [CNQ03, GGW+03a] use BDD-based reachability analysis and [GGW+03b] compute CNF clauses through BDD functional analysis to prune the search space of SAT-based BMC, [KPKG02] structurally partition the property check into parts solved by SAT and BDDs and [HWKF02] employ a combination of symbolic trajectory evaluation and SAT/BDD based symbolic model checking. The aim of this work is somewhat orthogonal to the above approaches since our aim is primarily in efficient bug-hunting rather than in formally verifying properties. In our opinion, some of the above approaches may prove to be an overkill for this objective. However, specific ideas from the above can potentially be used to enhance the efficacy of the individual engines used in our approach.

5.2.1 Our approach in context of other hybrid approaches

Our work differs from the above in two key aspects. Firstly, simulation forms the search backbone of many of the above methods. This is not the case in our approach. In the case of hard, deep bugs, a simulation based approach may not be able to access interesting regions of the search space. Secondly, our hybrid approach is formulated in a novel way with the specific aim of being able to generate multiple sub-problems that cover orthogonal regions of the state space which are then explored in parallel.

Note that previous hybrid approaches which combined BDDs and SAT-BMC used OBDDs as the chosen data structure which typically lead to a breadth-first processing of the state space. However when using OBDDs, it is difficult to control the direction of the search as well as the size of graphs that are typically generated. This makes it often hard to efficiently reach any states (even if arbitrary or only a handful) which are deep in state space. 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* while still keeping the individual graphs within a partition under control.In fact, if each subspace is thought of as a "direction", then reachability with partitioning localizes the Breadth First Search along such directions. The size of graphs generated can be further reduced by using under-approximation albeit at the cost of loss of information. Deep states provided from such, multiple, local BFS traversals can be used to provide many different initial states to seed subsequent BMC runs. These may then explore regions that could not be explored using traditional SAT-BMC approaches. If any particular local BFS traversal leads it in a direction which corresponds to a bug in the design then a BMC started from corresponding seed can prove to be far more efficient then classical BMC started from the original initial state.

5.2.2 Our approach in context of other parallel approaches

Our discussion in the above has been made in the context of a single processor framework. Note, typical hybrid approaches do not naturally lend themselves to parallel exploration. Our approach results in a non-traditional parallel-BMC approach which allows for multiple-BMC runs to be fired in parallel to concurrently

explore different state space regions.

Several other methods have been proposed to do parallel verification. Stern and Dill [SD97] parallelize an explicit model checker. In [SB96], parallelized BDDs are used for reachability analysis. Verification using parallel reachability analysis has been studied in [GMS01, HGGS00b, YO97]. Our work is different from other distributed model checking approaches which are geared towards completeness rather than bug hunting. Most techniques such as by Grumberg et al. [HGGS00b] are to a large extent only parallelizing the breadth-first traversal. Thus their limitations to reach deep states remains under a severe handicap. Further, these techniques require message passing between different processors in form of large BDDs. That can severely limit how large a grid can be effectively employed. A method proposed in [GGYA03] distribute the SAT-based BMC over a network of heterogeneous workstations. Their algorithm performs distributed BCP to solve a large SAT problem. Similarly, in [FDH04] a parallel multi-threaded SAT solver is discussed.

5.3 Preliminaries

In this section, we briefly look at some background related to state space partitioning and image computation.

5.3.1 SAT-based Bounded Model Checking

This chapter is based upon the following SAT-BMC framework. We are given a temporal logic property p to be verified on a finite transition system $\mathcal{M}$. For simplicity of exposition let us assume that $\mathcal{M}$ has a single initial state I and that the property p is an *invariant* [1]. The BMC problem is posed on a k-timeframe unrolled *iterative*

[1]The discussion can be easily generalized to multiple initial states and arbitrary LTL properties [CBRZ01].

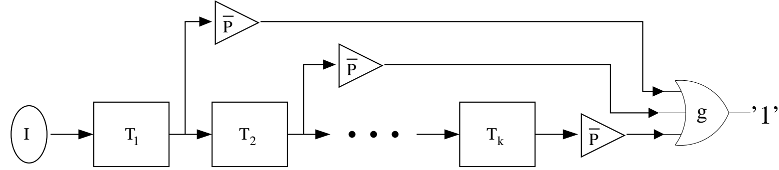

Figure 5.1: ILA for SAT-based BMC

logic array (ILA) of $\mathcal{M}$. as shown in Figure 5.1. The module $\overline{P}$ is a monitor that checks for the violation of property p in a particular time-frame. Thus, the circuit of Figure 5.1 encodes the condition that there exists a counter-example to p in the space of all executions of $\mathcal{M}$ of length at most k, starting with the initial state I. This circuit is typically translated into a *conjunctive normal form (CNF)* formula and decided by a conventional SAT solver such as the `zchaff` SAT solver [MMZ+01]. A satisfying assignment returned by the SAT solver translates to a counter-example to p while an *unsatisfiable* formula guarantees error-free behavior of $\mathcal{M}$ up to k depth of behavior. In the latter case, the process is usually iterated by increasing k by n time-frames (referred to as the *step size* of the BMC problem in the sequel) till a violation of property p is detected or a user specified bound on k or some resource constraints are exceeded.

5.3.2 The Grid Framework

We used agrid middle-ware, CyberGRIP [AK04], developed at Fujitsu Labs Limited, Japan, to manage the computing resources on grid. CyberGRIP operates on a central UNIX or Linux server and consists of three components: Organic Job Controller (OJC), Grid Resource Manager (GRM), and Site Resource Manager (SRM). CyberGRIP can realize an environment in which the user can submit jobs to virtualized computing resources consisting of not only Solaris and Linux machines but

also Windows machines for office use via the Web portal of a central server. The user can do this without being aware of the performance and other characteristics of the individual computers.

5.4 Algorithm: Under-approximation based Grid-BMC

As discussed earlier, our work specifically targets the problems where deep-states have to be explored and current BDDs and BMC methods may be inadequate. BMC based on SAT has a limitation on how deep a state it can explore as it is based on explicitly unrolling multiple time frames, equal in number to the length of the suspected error path. BDDs calculate successive image computations for reachability and can go deep, provided, the size of the transition relation is manageable and the successive images are small in size. Unfortunately, such images often get large, at a very small depth, and BDDs are unable to make further progress. Even if BDDs do not exhibit dramatic blowup in size, the computed image sets grow in size steadily, until they are so large that the calculations need impractically long time. Thus, for smaller depths SAT may be able to proceed further as it does not store sets of states, instead merely computes paths. We suggest a method for exploring deep states in the following.

5.4.1 Key Idea: Under-approximation for Grid-BMC

In this section, we describe the under-approximation heuristics that are used to perform deep state space traversal. Under-approximation is performed at two different places – firstly, during reachability analysis and secondly, while selecting initial states for SAT-based BMC.

Find Deep States: We perform a traversal of the state space by partitioning the transition relation, as well as the computed sets of states so that the BDDs and associated calculations remain tractable. When the BDD sizes are no longer manageable, we perform successive under-approximations. At each step of image computation, we use a subset of the actual set of states. Such massive under-approximation may result in successive traversal not always leading to a deeper state. The quality of this approximation can be further improved, especially with input from designer, for instance by using guided traversal [RS99, BRS00]. Under-approximation allows some control over the size of BDDs, which can otherwise exhibit dramatic blow-ups. We find that the above simple approach is quite effective and study it in detail in this paper.

Parallel Seed SAT: In order to determine the initial seed states for SAT, a large number of partitions are explored very rapidly with under-approximation, and the resulting deep states are written out at regular intervals, as CNF clauses. Currently, we create a new seed after a fixed number image computations, say, after every 5 images. In order to do this, a snapshot of the reachable states is taken and a subset of those states is used to seed the SAT solver, as depicted pictorially in Figure 5.2. It is critical to pick a small number of states and not all states, otherwise the SAT solver can choke as it gets a very large number of clauses. The SAT solver instance executes a BMC-like algorithm from the seed thus obtained. By making multiple BMC runs, starting from various points along the state traversal, we can ensure that at least a subset of the BMC executions start from a deep state. These may then explore regions that could not be explored using traditional SAT-BMC approaches. Since all BMC runs can be made in parallel so this leads to a non-traditional method of parallelizing BMC.

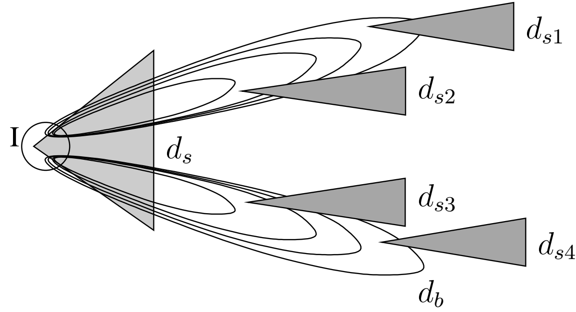

Figure 5.2: Seeding multiple SAT-BMC runs from POBDD reachability

Typically we execute SAT-BMC to a small depth that can be computed in a matter of minutes using zchaff SAT solver. Note that it is critical to run these SAT instances separately rather than run a single instance of SAT from their union. The reason is that seed states from different portions of the state space may be very dissimilar and if they are combined together in generating the clauses for SAT, the effectiveness of the SAT solver may reduce drastically.

Degree of Approximation: Each partition can be considered as an automatic selection of "direction" with respect to exploration of states. Reachability with partitioning localizes the Breadth First Search along such directions. The under-approximation in this exploration is performed simply by picking a few random states from the set of new states found during each image computation. The size of BDDs generated is reduced by this using under-approximation albeit at the cost of loss of information. Keeping BDD sizes small in this manner allows for deeper exploration in selected "directions" and can be used to provide many different initial states to seed subsequent BMC runs. If any particular local BFS traversal leads it in a direction which corresponds to a bug in the design then a BMC started from corresponding seed can prove to be far more efficient then classical BMC started from the original initial state. Notice that there is a trade-off between going deep

versus exploring all directions.

5.4.2 Approximation and Usage of Grid

There is a clear trade-off between the degree of approximation and number of CPUs required. If many states are selected as a seed state, then effectively it corresponds to simultaneously searching in many directions with BMC. However, the corresponding BMC run may become slower. To solve this dilemma we propose the following method. The grid-BMC starts by dividing the grid into multiple sub-grids. One sub-grid uses severe under approximation to go very deep. The others run POBDDs and BMC with varying degrees of approximation. Then we monitor for either of the following condition to arise: (a) Though a more complex seed is used, the BMC runtime is not adversely effected; (b) Though a less complex seed is used, the number of nodes in the grid are rapidly used up. In such cases, the algorithm automatically switches to using the larger number of min-terms as seed for BMC on the main sub-grid. Otherwise, we continue to use the less complex seeds with fewer min-terms. In order to switch between approximation levels, the runs with a higher approximation are canceled and the corresponding nodes on the grid are freed for the runs that use lower approximation (more number of min-terms used as seed for BMC as well as frontier for POBDD image calculation).

We divide the grid amongst BMC runs with different under-approximations. The question is how to dynamically decide which sub-grid is showing sub-optimal behavior so that its given run may be canceled and the corresponding CPUs freed up. The quality of a run can be indicated by the following three resources consumed. A disproportionately large value for either resource in any of the sub-grids should lead to the run being aborted, and the nodes in that sub-grid being freed up.

(a) POBDD time: If in one of the sub-grids the BDD size (resp. time) starts increasing significantly as compared to the BDD size (time) in other sub-grid, then it indicates a run that is being hampered by suboptimal variable order or quantification schedule and should be aborted. Hence the time for each step of image computation acts as a good filter for the sub-optimal runs.

(b) Ratio of BMC time/time-frame: At times it is seen that the BMC runs from some seeds start consuming a disproportionately large time and become impractically slow. This generally happens when the number of states used as the initial seed state become very large. A ratio of the average runtime/time-frame can be maintained, and when in one of the grid this number is significantly exceeded then the corresponding run on the sub-grid is aborted.

(c) Number of CPUs used - Under extreme conditions, this should be used as another measure for tagging the runs on a given sub-grid to be sub-optimal. For greater under-approximation (fewer states in seed), usage of a larger number of CPUs decreases its attractiveness. This is because when initial seed state set is smaller, each BMC run is starting from the end of traversal in a very narrow direction. Hence, the use of a large number of CPUs potentially indicates a large number of unsuccessful BMC runs, which in turn implies that the starting seed states are not a good choice and are perhaps not in the area corresponding to the error.

5.4.3 Outline of Grid-BMC Algorithm

Divide_Grid: Create multiple sub-grids to dynamically determine a good approximation threshold in order to detect whether an error exists. On each sub-grid, do the following:

(i) *Partition_reach*: Use state partitioning in reachability to get different and divergent paths exploring state space.

(ii) *Approx_Partition_reach*: Target deep space traversals – from each frontier, select an under-approximation of the newly reached states to do the next image computation.

(iii) *Generate_seed*: At regular intervals, whenever a threshold is crossed, store a *seed* – a few reachable states.

(iv) *Start_Seeded_SAT*: From each seed, pass it as the initial state to a new instance of the SAT solver.

(v) *Run_in_Parallel*: Run all instances of SAT-based BMC to a small depth, as nodes become available on the grid.

(vi) *Abort_Sub_Grid*: If the BDD image time and SAT-BMC time for each time-frame in this sub-grid are disproportionately larger than other sub-grids, then abort runs on this sub-grid.

Termination_condition: Allow BDD and SAT explorations to continue in parallel on all sub-grids until error is found or timeout is reached.

5.4.4 Comments on technique

Correctness: The *soundness* of the procedure is fairly clear by construction, i.e., if an error is detected, then it is indeed a real error in the design and a trace is generated from an initial state to the state with an error. *Bounded completeness*, i.e., completeness to certain depth, can be achieved by adding in the initial set of states to every seed set passed to BMC. Alternately, we can get the same result by running BMC separately from the initial states – then the design is guaranteed to be error-free until this depth.

Depth analysis: Let depth of the initial BDD stage be d_b and the depth of the SAT stage be d_s. Then the combination described is guaranteed at least one state to a depth of max(d_b,d_s), and it is guaranteed to be error-free to depth d_s. i.e., in terms of completeness this is no worse than BMC and at the same time, it also has some deeper seeds. Empirically, these deeper seeds are very effective in locating errors.

Figure 5.2 shows this pictorially using two partitions and four instances of SAT. The triangles represent a search using SAT, and the ellipses denote successive image computations using BDDs. Notice that from the initial states, BMC can only proceed to depth d_s effectively. Instead, in the proposed approach, BDDs go to a depth d_b, and many instances of SAT are seeded until then, which may be effective to differing depths $d_{s1}, d_{s2} \ldots d_{sn}$. Consequently, this can reach errors that are otherwise difficult to catch.

Notes about implementation

Each partition can be considered as an automatic selection of "direction" with respect to exploration of states. In our preliminary implementation the under-approximation is performed simply by picking a few random states from the set. The quality of this approximation can be further improved, esp with input from designer, for instance by using guided traversal [MKSS99]. The SAT solver instance executes a BMC-like algorithm from the seed thus obtained. Currently, our simple implementation creates a new seed after a fixed number image computations, say, after every 5 images. In order to do this, a snapshot of the reachable states is taken and a subset of those states is used to seed the Sat solver. It is critical to pick a small number of states and not all states, otherwise the SAT solver can choke as

it gets a very large number of clauses. Typically we run the SAT-BMC to a small depth of 20, as this can be computed in a matter of minutes using zchaff SAT solver. Note that it is critical to run these SAT instances separately rather than run a single instance of SAT from their union. The reason is that seed states from different portions of the state space may be very dissimilar and if they are combined together in generating the clauses for SAT, the effectiveness of the SAT solver may reduce drastically.

5.5 Experimental Results

In this section, we present experimental results that demonstrate the efficacy of our method. The experiments are run on a grid of computers that included up to 100 independent Xeon CPUs (ranging from 1.5 GHz to 2.3 GHz) running Linux. As explained earlier, we used an in-house grid middle-ware (CyberGrip) developed at Fujitsu Labs Limited, Japan, for submitting and controlling jobs executed on the grid. Our program is implemented on top of VIS-2.0 and used CUDD BDD package and zchaff as the SAT-solver. The POBDD algorithm is run on a single processor but the CNF files generated are transfered to different nodes on the grid where a BMC run is fired in parallel. Due to the lack of fine grained user control on the functionality of grid middle-ware we were unable to control the number of nodes used in each experiment. We were also unable to exactly measure the time taken in transferring the files but in our experience it is very small.

Benchmarks: We used 9 circuits and properties, $b1, \ldots b9$, that were obtained during verification of a variety of industrial circuits. Several of these properties are deep and pose some difficulty for SAT-BMC as well as for simulation based methods. Thus they form a good benchmark for judging the efficacy of our

			Runtimes (sec) for BMC seeded from simulation									
	Num.	Error	2k		4k		6k		8k		10k	
Ckt	latches	Depth	Sim	BMC	Sim	BMC	Sim	BMC	Sim	BMC	Sim	BMC
b1	125	59	22	215	25	207	48	151	56	66	63	196
b2	70	85	18	117	21	105	32	96	45	110	55	118
b3	66	23	21	333	25	195	38	232	50	258	63	258
b4	66	59	27	2211	26	2747	60	2442	56	2239	63	2060
b5	170	36	653	1047	923	2067	1605	1561	1822	1735	2333	2493
b6	201	29	629	487	921	509	1258	396	1756	346	2423	313
b7	123	60	892	T/O	1305	T/O	2951	T/O	2694	T/O	3515	T/O
b8	169	23	105	T/O	122	T/O	193	T/O	361	T/O	476	T/O
b9	148	27	106	T/O	130	T/O	295	T/O	256	T/O	462	T/O

"T/O" is a timeout of 2 hrs

Table 5.1: Run-times for BMC seeded from simulation to various depths.

approach.

5.5.1 Experiments with Simulation alone

First we ran experiments based on random simulation, using the VIS-2.0 package. Simulation was done twice, first to 5,000 and then to 100,000 steps, but it is unable to find a bug in any of the circuits in the benchmark. Then, we used simulation to find deep states and seed BMC from there. This is similar to the approach of [HSH$^+$00], except that we use a different random seed for each simulation depth. For each circuit, we run simulation, in steps of 1,000 from 2,000 to 10,000. When the depth is reached, we pick the state reached at the end of the simulation and seed SAT from there. To limit the amount of data, the results of this are shown in table 5.1 for depths 2k, 4k, 6k, 8k and 10k. We found that simulation, even when it seeds SAT at periodic depths of every 1000 steps is unable to find any bug in any of the circuits in the benchmark.

5.5.2 Experiments on Grid-BMC

Next, in table 5.2, we compare the following methods against each other:

Ckt	Num. latches	Error Depth	Total Time (sec)					
			BDD	POBDD	BMC	Sim	Sim+BMC	Proposed
s1423b	153	16	T/O	45	800	NB	T/O	**35**
b1	125	59	7	**3.2**	T/O	NB	167	**3.2**
b2	70	85	3.4	**2**	T/O	NB	115	**2**
b3	66	23	1.9	**1.3**	T/O	NB	268	**1.3**
b4	66	59	1.9	**1.3**	T/O	NB	3097	**1.3**
b5	170	36	T/O	T/O	T/O	NB	2758	**63**
b6	201	29	3148	2857	T/O	NB	1407	**176**
b7	123	60	**258**	976	T/O	NB	T/O	464
b8	169	23	T/O	T/O	T/O	NB	T/O	**253**
b9	148	27	T/O	T/O	T/O	NB	T/O	**1860**

"T/O" is a timeout of 2 hrs, "NB" means no bug found.

Table 5.2: Comparison of the time taken in seconds by various approaches.

1. BDD: Invariant checking using BDD package CUDD and the reachability algorithms of VIS-2.0 package.

2. POBDD: Invariant checking using state partitioning and POBDDs, on top of VIS-2.0 package.

3. BMC: Bounded Model Checking as implemented in VIS-2.0 using SAT-solver zchaff.

4. simulation: Random simulation to 5,000 steps.

5. Simul + BMC : An application of SAT solver after 5,000 random simulations.

6. Our proposed Grid-BMC method of this chapter, with a 10 minute initial phase for POBDD based seed generation, and 2 hours for SAT.

In the case of Grid BMC, there are many seed states, so the table shows how long it took for POBDD based reachability to discover the "best" seed state and time for the SAT-solver to find the bug from there. The final column shows how many CPUs of the grid were actually used. We allow each method to run until a time out of 2 hours. The results for all the methods are shown in table 5.2. Note that Grid-BMC is the only method that finds the error in benchmarks b8 and b9.

Ckt	Num. latches	Error Depth	Time (sec) Seed	BMC	Total	Num. CPU
s1423b	153	16	15	20	35	5
b1	125	59	3.2	N/A	3.2	1
b2	70	85	2	N/A	2	1
b3	66	23	1.3	N/A	1.3	1
b4	66	59	1.3	N/A	1.3	1
b5	170	36	27	36	63	9
b6	201	29	156	20	176	3
b7	123	60	35	429	464	14
b8	169	23	198	55	253	28
b9	148	27	280	1580	1860	70

"N/A" – BMC was not required.

Table 5.3: Details of performance data of proposed hybrid Grid-BMC approach.

Table 5.3 shows the details of time spent by the proposed method. The table shows how many CPUs were necessary in the grid, and how long it took for (a) POBDD based reachability to discover the seed state and (b) the SAT-solver to find the bug from there.

Before we analyze these results in detail, let us consider a case study.

5.5.3 Case Study

Consider the circuit s1423b shown in the tables. This is a modification of the circuit s1423 from the ISCAS89 benchmark suite. We took two copies of the circuit and attached them to a miter to check that they produce the same result. We then artificially planted a bug at depth 16, which is reached by a 4 bit counter. This bug is difficult to reach by using classical BDDs as the DD sizes get very large. Reachability analysis using partitioned BDDs, with 16 partitions, is able to find a path to the error state in 45 seconds. SAT based Bounded Model Checking takes about 800 seconds to find the error, even though the depth of 16 is considered a shallow depth. On analyzing the behavior of various methods, we find that the first 10 steps of reachability are very easy for BDDs and Partitioned BDDs, and

after that they get progressively more difficult. On the other hand, if SAT-BMC is started from the initial states, it spends a lot of time disproving the existence of an error at the shallower depths. Thus BDDs are ineffective due to large sizes, and BMC is sensitive to its start point. Notice that the hybrid method is the fastest of all for this circuit – it creates seeds at various depth and each launches an instance of SAT-BMC in parallel on a different processor. The SAT instance from the seeds at depth 10 is able to complete in about 20 more seconds. This is the time that it takes POBDD based reachability to reach depth 13, so the hybrid approach finishes before the pure BDD or pure BMC approaches.

5.5.4 Analysis of the Results

Our results show a strong evidence of a positive synergy between the two key ideas: (a) How to go deep; (b) How to process multiple seeds. Specifically, from the tables 5.2 and 5.3, we note the following:

- Grid-BMC can often find errors significantly faster than BDDs or POBDDs alone.
- Grid-BMC finds errors on circuits where BMC runs out of time.
- BDD based seeding of SAT solvers works well, and is often more effective than seeding using random simulation, which is widely accepted as one of the best strategies for industrial designs.
- On every example, Grid-BMC is superior to BMC, random simulation and a combination of the two; either in finding an error faster, or by finding an error that is not otherwise found.

- On examples that are BDD-friendly, Grid-BMC performs better than the other BMC techniques.

For some circuits such as b1, b2, b3 and b4 the error can be detected in the POBDD phase itself. At the time the error was detected, no BMC run that had been been fired had yet completed. Thus grid-BMC is not required for these entries, but we show it so we can analyze the effect of approximation in further tables.

Based upon our analysis of the experimental results, we believe that the proposed hybrid method has various benefits. It is computationally inexpensive in terms of overhead and an alternate way of parallelizing SAT-based BMC – each of many processors can execute a BMC from a different set of initial states. The only data that is passed over the network is at the very beginning, after that no synchronization is required, until termination. Such parallelization has no interdependence at all, and can therefore very effectively utilize a number of processors in a large grid, without creating communication overhead between the processors. This method also effectively exploits the advantage of symbolic BDD based search as well as SAT. If there are a large number of partitions or if certain partitions are difficult, performing cross-over images between them can be difficult, and this may be the bottleneck in getting to the error. This can be overcome by SAT based BMC, which is "locally complete" from its originating point and does not compute sets of states.

Although a very large grid was available, in typical experiments only a small number of CPUs were used. This suggests significant scope to improve the quality of results and possibility to tackle larger problems with further research.

	pickTwo				pickThree				pickFive			
Ckt	Seed	SAT	Total	#CPU	Seed	SAT	Total	#CPU	Seed	SAT	Total	#CPU
b1	18	395	413	16	185	551	736	4	47	302	349	14
b2	450	29	479	10	197	52	249	7	4	42	46	24
b3	1	3	4	2	1	5	6	3	7	2	9	8
b4	18	505	523	21	23	624	647	21	19	316	335	15
b5	252	83	335	9	241	78	319	5	26	80	106	24
b6	154	24	178	9	324	19	343	6	191	24	215	4
b7	333	623	956	23	43	489	532	3	95	716	811	28
b8	167	7	194	27	93	104	197	3	91	7	98	8
b9	88	745	833	21	342	234	576	87	270	218	488	89

Table 5.4: Effect on Grid-BMC performance (time in seconds) of relaxing the severity of the approximation

	pickTen				pickTwenty				pickFifty			
Ckt	Seed	SAT	Total	#CPU	Seed	SAT	Total	#CPU	Seed	SAT	Total	#CPU
b1	223	406	629	28	19	756	775	27	155	546	701	14
b2	408	72	480	44	84	103	187	20	65	945	1010	18
b3	1	3	4	2	1	4	5	3	11	4	15	3
b4	8	362	370	9	6	447	453	5	48	613	661	7
b5	25	69	94	6	91	91	182	14	127	183	310	20
b6	150	23	173	6	155	27	182	4	157	44	201	4
b7	43	711	754	8	296	798	1094	26	176	1222	1398	10
b8	82	84	166	16	114	38	152	5	124	30	154	4
b9	145	698	843	28	175	1213	1388	28	11	2977	2988	34

Table 5.5: Effect on Grid-BMC performance (time in seconds) of drastically relaxing the severity of the approximation

5.5.5 Effect of under-approximation

We now study the effect of varying the approximation involved in our Grid-BMC approach.

Table 5.4 shows the effect of relaxing the approximation, by picking more states at each step (call this m). In our experimental runs, we varied m from 1 to 5 for each circuit. Table 5.5 shows the effect of drastically relaxing the under-approximation, by successively using 10, 20 and then 50 states at each step.

The above results can be viewed in the context of running two or more configurations (conceptually each can be conceived as a sub-grid) in parallel (say, $m = 1$ and $m = 5$). We suggest an approach to automatically decide the better configuration. Our decision approach needs to monitor for each sub-grid, the corre-

sponding BDD image-time, and BMC-SAT time for each time-frame. If these run times become disproportionately and significantly larger then the given configuration is displaying sub-optimality. Note that as the under-approximation is relaxed, but the corresponding BDD image-time, and BMC-SAT time for each time-frame do not proportionately increase, then the more accurate approximation starts yielding faster error detection. This is logically expected since when we relax the severity of approximation, the resulting BMC can be deemed as searching simultaneously in multiple directions, and thus a larger state space. The utility of such an approach is confirmed for almost all cases by the experimental results presented here. For example, entry $b9$ requires 1860 seconds on the grid for $m = 1$. As m increases progressively from 1 to 5; the run-times proportionally decrease by a factor of 4. Identical observations hold for circuit $b8$ too.

5.6 Conclusions

In this chapter, we explore the problem of error detection for the cases where the error is very deep. These cases can be difficult for both BDDs as well as SAT-BMC methods. Our method is based upon two concepts: (a) Generating seed states using state partitioning and under-approximation. (b) A non-conventional Parallelization of BMC by executing multiple instances of SAT-based BMC independently and simultaneously on different nodes of a computing grid. Our work focuses only on the problems where deep-states have to be explored and current BMC methods may be inadequate even though the circuit size may not be large. Experimentally we find that both Partitioning and SAT-BMC approaches can be quite complementary for a number of such cases. For such problems we show that our approach can significantly speed-up Bounded Model Checking. The techniques discussed in this work are akin

to a hybrid approach using multiple engines. Thus, they can easily use improvements in the individual technologies such as BDDs, SAT or BMC formulation. Further, we show that BDD-based reachability is a viable way of finding seeds quickly for error detection using BMC. In our experience, on industrial circuits, this may be significantly better than the combination of random simulation and BMC.

Chapter 6

Future Work and Conclusions

Parallel and distributed algorithms and heuristics for reachability and model checking are increasingly becoming important with the advent of multi-core processors and wide-spread adoption of computation cluster systems. Partitioning is inherently suitable for adoption of such systems. A number of directions for future research are suggested by this work.

The problem of selecting good partitioning variables still remains open. We have empirically observed that co-factoring with respect to the function that corresponds to a control signal is an effective way of partitioning. This leads us to believe that window selection can be further generalized, and we outline an approach for this in the following.

Another significant bottleneck for increasing the capacity of BDD-based symbolic verification is the construction of the transition relation for very large circuits. We describe briefly an approach to this problem using simulation-guided circuit slicing in this chapter.

6.0.1 Generalized Windows and Multiply rooted BDDs

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, the partitioning schemes in the literature are all based upon the use of windows that are defined as min-term 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.

The current partitioning approach is based on min-term 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 [JMM+00] shows that substantial gains may be achieved.

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 as was shown by Jain, et.al. [JMM+00].

The discussion above shows the advantage of partitioning in performing multiple composition operations. We raise 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. 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. 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. Compose the decomposition point variables until some composition blows up.

3. 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 use this partition to be the root of our non-cube based partitioning tree (NCPT).

4. For each leaf of the above defined NCPT, 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 transition relation, perform a reachability analysis until a fixed point is reached.

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.

6.0.2 Simulation-guided circuit slicing for BMC

When the design is large the transition relation cannot be built, even in a n implicitly conjuncted form. Therefore, BDD and POBDD based reachability cannot be done. One approach to this problem, is to work with implicitly disjunctive transition relations to perform full model checking. Another idea is to use simulation-guided circuit slicing as a preprocessing step before applying the grid-BMC method of the previous chapter. In what follows, we examine the latter idea in some detail.

The grid-based model checking approach of the previous chapter starts out with POBDD based state exploration to find deep or intelligent states from which to seed BMC. This is feasible only when the transition relation can be built in order to initially perform reachability analysis. So this limits the sizes of designs that the method can handle as input.

To handle large industrial sized designs, we need to reduce the size of the transition relation very brutally, perhaps by doing severe under-approximations. We can try to do automatic circuit slicing and discard some of the circuit, thus allowing for a transition relation to be built only for the interesting portions of the circuit.

It is conceivable that the slicing may abstract away so much behavior and that such a "sliced TR" may have so little information left in it that it does not fire any transitions. So we need a guide for the slicing. The guide method should be capable of reading in the entire circuit, starting at an initial state and finding some other states. It needs to do this fast.

The only method that can handle thousands of flip flops and find states fast is simulation. Accordingly the methodology we suggest is as follows:

1. Use random simulation to find some reachable states quickly.
2. Use the above states to perform circuit slicing.
3. Use this reduced circuit to build a transition relation for image computations.
4. Now use the grid-BMC method of the previous chapter.

The question arises as to how does knowing some reachable states help in doing massive but not too massive slicing? The slicing technique is as follows: First, assign constant values to some subset of the input variables as well as the state variables. In the second step, propagate these deterministic constants in the

circuit to discover and remove redundant latches and signals. Repeat the two steps until circuit becomes small enough.

In assigning values to variables, two issues arise. Firstly, the values need to be consistent with each other. Secondly, the resulting reduced circuit needs to have reachable states, i.e., not all behavior should be sliced away. If the assignments are inspired by states known to be reachable, and we select values consistent with those states to do the slicing, there are some reachable states in the slice, namely those reachable states themselves.

We expect this approach to circuit slicing to be fully automated and to scale to large design sizes.

6.1 Conclusions

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.

Our techniques have been increase the capacity of automatic state-based verification as applied to sequential designs, i.e., symbolic model checking. Firstly, our approach to reachability and model checking splits the problem into multiple partitions handled independently of each other. Secondly, we have shown 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.

Parallelized search using partitioning in multi-threaded and multi-processor environments has shown show significant speedups over the sequential approaches. The trace-centric approach has enabled a significant improvement in the stability of BDD-based formal verification, leading to a fast and stable falsification methodology.

We have addressed the issue of time scalability in verification, whereby the availability of larger amounts of computation time enables greater exploration of the state space, by mean of a dynamic partitioning methodology. From a practical standpoint, we observe that extant verification approaches are unable to proceed very deep into the state space. Our work on a partitioning approach to Bounded Model Checking has been shown to combine the advantages of deep state space exploration with the speed and memory efficiency of bounded model checking.

We have identified various directions for future work. We wish to consider the key issue of selecting good window functions for the partitioning approach. This would take advantage of the non-determinism afforded by POBDDs, and is likely to shows its full benefit of potentially exponential savings in the context of operations expressible as a sequence of compositions. Another promising area of investigation is simulation-guided Bounded Model Checking.

Bibliography

[AK04] Akira Asato and Yoshimasa Kadooka. Grid Middleware for Effectively Utilizing Computing Resources: CyberGRIP. In *Fujitsu Scientific and Technical Journal*, volume 40, pages 261–268, 2004.

[AKMM03] Nina Amla, Robert Kurshan, Kenneth McMillan, and Ricardo Medel. Experimental Analysis of Different Techniques for Bounded Model Checking. In *TACAS*, volume 2619 of *LNCS*, pages 34–48. Springer, April 2003.

[BCMD92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, and David L. Dill. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, 1992.

[BCRZ99] Armin Biere, Edmund Clarke, Richard Raimi, and Yunshan Zhu. Verifying safety properties of a PowerPC microprocessor using symbolic model checking without BDDs. In *CAV*, volume 1633 of *LNCS*, pages 60–71. Springer, July 1999.

[BHSV$^+$96] R. K. Brayton, G. D. Hachtel, A. L. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. Cheng, S. A. Edwards, S. P. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple,

G. Swamy, and T. Villa. VIS: A System for Verification and Synthesis. In *Computer Aided Verification*, 1996.

[BLM01] Per Bjesse, Tim Leonard, and Abdel Mokkedem. Finding Bugs in an Alpha Microprocessor Using Satisfiability Solvers. In *CAV*, volume 2102 of *LNCS*, pages 454–464. Springer, July 2001.

[BRS00] Roderick Bloem, Kavita Ravi, and Fabio Somenzi. Symbolic guided search for CTL model checking. In *Proc. of the Design Automation Conf.*, pages 29–34, 2000.

[Bry86] Randall. E. Bryant. Graph based algorithms for Boolean function representation. *IEEE Transactions on Computers*, C-35:677–690, August 1986.

[BW97] Beate Bollig and Ingo Wegener. Partitioned BDDs vs. other BDD models. In *Proc. of the Intl. Workshop on Logic Synthesis*, 1997.

[CBM89] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of sequential machines based on symbolic execution. In *Proc. of the Workshop on Automatic Verification Methods for Finite State Systems*, pages 365–373, Grenoble, France, 1989.

[CBRZ01] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 19(1):7–34, July 2001. Kluwer Academic Publishers.

[CCQ96] G. Cabodi, P. Camurati, and Stefano Quer. Improved reachability analysis of large finite state machines. *ICCAD*, pages 354–360, 1996.

[CE81] Edmund M. Clarke and E. Allen 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.

[CFF+01] Fady Copti, Limor Fix, Ranan Fraer, Enrico Giunchiglia, Gila Kamhi, Armando Tacchella, and Moshe Y. Vardi. Benefits of Bounded Model Checking in an Industrial Setting. In *CAV*, pages 436–453, July 2001.

[CNQ03] Gianpiero Cabodi, Sergio Nocco, and Stefano Quer. Improving SAT-based Bounded Model Checking by Means of BDD-based Approximate Traversals. In *Proc. of the Design Automation and Test in Europe*, pages 898–903, March 2003.

[Eme90] E. Allen 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.

[FDH04] Yulik Feldman, Nachum Dershowitz, and Ziyad Hanna. Parallel multi-threaded satisfiability solver: Design and implementation. In *PDMC*, 2004.

[GAK99] Malay Ganai, Adnan Aziz, and Andreas Kuehlmann. Enhancing Simulation with BDDs and ATPG. In *Proc. of the* 36th *Design Automation Conference*, pages 385–390, June 1999.

[GGW+03a] Aarti Gupta, Malay Ganai, Chao Wang, Zijiang Yang, and Pranav Ashar. Abstraction and BDDs Complement SAT-based BMC in Di*Ver*. In Jr. Warren A. Hunt and Fabio Somenzi, editors, *Proc. of*

the 15th *Conf. on Computer-Aided Verification*, volume 2725 of *LNCS*, pages 206–209. Springer, July 2003.

[GGW+03b] Aarti Gupta, Malay Ganai, Chao Wang, Zijiang Yang, and Pranav Ashar. Learning from BDDs in SAT-based Bounded Model Checking. In *Proc. of the* 40th *Design Automation Conf.*, pages 824–829, June 2003.

[GGYA03] Malay K. Ganai, Aarti Gupta, Zijiang Yang, and Pranav Ashar. Efficient Distributed SAT and SAT-Based Distributed Bounded Model Checking. In *CHARME*, pages 334–347, 2003.

[GHS01] Orna Grumberg, Tamir Heyman, and Assaf Schuster. Distributed symbolic model checking for μ-calculus. In *Computer Aided Verification*, pages 350–362, 2001.

[GMS01] Hubert Garavel, Radu Mateescu, and Irina Smarandache. Parallel state space construction for model-checking. In *Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 217–234. Springer-Verlag New York, Inc., 2001.

[HGGS00a] 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.

[HGGS00b] Tamir Heyman, Danny Geist, Orna Grumberg, and Assaf Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In Orna Grumberg, editor, *CAV*, volume 1855, pages 20–35. Springer, 2000.

[HKB93] Ramin Hojati, Sriram C. Krishnan, and Robert 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.

[HSH+00] Pei-Hsin Ho, Thomas Shiple, Kevin Harer, James Kukula, Robert Damiano, Valeria Bertacco, Jerry Taylor, and Jiang Long. Smart Simulation Using Collaborative Formal and Simulation Engines. In *Proc. of the IEEE/ACM International Conference on Computer-Aided Design*, pages 120–126, November 2000.

[HWKF02] Scott Hazelhurst, Osnat Weissberg, Gila Kamhi, and Limor Fix. A Hybrid Verification Approach : Getting Deep into the Design. In *Proc. of the* 39th *Design Automation Conference*, pages 111–116, June 2002.

[IN97] Hiroaki Iwashita and Tsuneo Nakata. Forward model checking techniques oriented to buggy designs. In *ICCAD '97: Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*, pages 400–404, Washington, DC, USA, 1997. IEEE Computer Society.

[IPC03] Madhu K. Iyer, Ganapathy Parthasarathy, and Kwang-Ting Cheng. SATORI-A Fast Sequential SAT Engine for Circuits. In *Proc. of the IEEE/ACM International Conference on Computer-Aided Design*, pages 320–325, November 2003.

[ISS+03] Subramanian Iyer, Debashis Sahoo, Christian Stangier, Amit Narayan, and Jawahar Jain. Improved symbolic Verification Using Partitioning

Techniques. In *Proc. of CHARME 2003*, volume 2860 of *Lecture Notes in Computer Science*, 2003.

[Jai93] Jawahar Jain. On analysis of boolean functions. *Ph.D Dissertation, Dept. of Electrical and Computer Engineering, The University of Texas at Austin*, 1993.

[JBFA92] Jawahar Jain, James Bitner, Donald S. Fussell, and Jacob A. Abraham. Functional partitioning for verification and related problems. *Brown/MIT VLSI Conference*, March 1992.

[JED91] Jerry R. Burch, Edmund M. Clarke, and David E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P.B. Denyer, editors, *International Conference on Very Large Scale Integration*, pages 49–58, Edinburgh, Scotland, 1991. North-Holland.

[JMM+00] Jawahar Jain, Kartik Mohanram, Dinos Moundanos, Ingo Wegener, and Yuan 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.

[KMB99] Andreas Kuehlmann, Kenneth McMillan, and Robert Brayton. Probabilistic State Space Search. In *Proc. of the IEEE/ACM International Conference on Computer-Aided Design*, pages 574–580, November 1999.

[KPKG02] Andreas Kuehlmann, Viresh Paruthi, Florian Krohm, and Malay K. Ganai. Robust Boolean Reasoning for Equivalence Checking and Functional Property Verification. *IEEE Trans. on CAD*, 21(12):1377–1394, Dec. 2002.

[LST03] Flavio Lerda, Nishant Sinha, and Michael Theobald. Symbolic Model Checking of Software. In *Workshop on Software Model Checking*, 2003.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[MHS00] In-Ho Moon, Gary D. Hachtel, and Fabio Somenzi. Border-Block Triangular Form and Conjunction Schedule in Image Computation. In *Proc. of Formal Methods in CAD (FMCAD'00)*, volume 1954 of *Lecture Notes in Computer Science*, 2000.

[MKSS99] In-Ho Moon, James Kukula, Tom Shiple, and Fabio Somenzi. Least fixpoint approximations for reachability analysis. In *Proceedings of the International Conference on Computer-Aided Design*, pages 41–44, November 1999.

[MMZ+01] Matthew Moskewicz, Conor Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. of the* 38th *Design Automation Conf.*, pages 530–535, June 2001.

[NIJ+97] Amit Narayan, Adrian Isles, Jawahar Jain, Robert Brayton, and Alberto L. Sangiovanni-Vincentelli. Reachability Analysis Using Partitioned-ROBDDs. In *ICCAD*, pages 388–393, 1997.

[NJFSV96] Amit Narayan, Jawahar Jain, Masahiro Fujita, and Albertoi L. Sangiovanni-Vincentelli. Partitioned-ROBDDs - A Compact, Canonical and Efficiently Manipulable Representation for Boolean Functions. In *ICCAD*, pages 547–554, November 1996.

[RAB+95] Rajeev K. Ranjan, Adnan Aziz, Robert K. Brayton, Carl Pixley, and

Bernard Plessier. Efficient BDD Algorithms for Synthesizing and Verifying Finite State Machines. In *Proc. of the Intl. Workshop on Logic Synthesis*, 1995.

[RS95] Kavita Ravi and Fabio Somenzi. High-density reachability analysis. In *ICCAD*, pages 154–158, 1995.

[RS99] Kavita Ravi and Fabio Somenzi. Hints to accelerate symbolic traversal. In *CHARME*, pages 250–264, 1999.

[SB96] Tony Stornetta and Forrest Brewer. Implementation of an efficient parallel BDD package. In *Proceedings of the 33rd annual conference on Design automation*, pages 641–644. ACM Press, 1996.

[SD97] Ulrich Stern and David L. Dill. Parallelizing the murphy verifier. In *CAV*, volume 1254, pages 256–267. Springer-Verlag, June 1997.

[SIJ+04] Debashis Sahoo, Subramanian Iyer, Jawahar Jain, Christian Stangier, Amit Narayan, David L. Dill, and E. Allen Emerson. A Partitioning Methodology for BDD-based Verification. In *Formal Methods in Computer-Aided Design*, volume 3312 of *Lecture Notes in Computer Science*, pages 399–413. Springer-Verlag, January 2004.

[SJI+05] Debashis Sahoo, Jawahar Jain, Subramanian Iyer, David L. Dill, and E. Allen Emerson. Multi-threaded Reachability. In *Proceedings of the 42nd conference on Design automation*, pages 467–470, June 2005.

[Som01] Fabio Somenzi. CUDD: CU Decision Diagram Package ftp://vlsi.colorado.edu/pub, 2001.

[TSL+90] Hervé J. Touati, Hamid Savoj, Bill Lin, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Implicit State Enumeration of Finite State Machines using BDD's. In *ICCAD*, pages 130–133, November 1990.

[vis] VIS Verilog Benchmarks http://vlsi.colorado.edu/˜ vis/.

[YD98] C. Han Yang and David L. Dill. Validation with Guided Search of the State Space. In *Proc. of the* 35th *Design Automation Conference*, pages 599–604, June 1998.

[YO97] Bwolen Yang and David R. O'Hallaron. Parallel breadth-first bdd construction. In *Proceedings of the sixth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 145–156. ACM Press, 1997.

[YSA00] Praveen Yalagandula, Vigyan Singhal, and Adnan Aziz. Automatic Lighthouse Generation for Directed State Space Search. In *Proc. of Design Automation and Test in Europe (DATE)*, pages 237–242, March 2000.

Vita

Subramanian Iyer was born in Bombay, India, on November 2nd, 1977, the son of Papparimadham Sankaranarayanan Kaveri and Pallassana Subramanian Krishnan. After completing his secondary schooling, he spent the four years from 1995 to 1999 at the Indian Institute of Technology, Bombay and obtained a Bachelors in Technology in Computer Science and Engineering. He spent three summers at the Institute of Mathematical Sciences in Madras, India, studying Mathematics and Computer Science. Since Fall 1999, he has been in the Doctoral Program in Computer Sciences at the University of Texas at Austin, where he was awarded Masters in Science in May 2005 and Ph. D. in December 2006.

Permanent Address: D-78 Anjali, 90 Feet Road
Mulund East, Mumbai 400081
Maharahtra, India

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.