Efficient Distributed Bounded Property Checking

Dissertation
der Fakultät für Informations- und Kognitionswissenschaften
der Eberhard-Karls-Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Dipl.-Inform. Pradeep Kumar Nalla
aus Hyderabad, Indien

Tübingen
2008

Tag der mündlichen Qualifikation:	09.07.2008
Dekan:	Prof. Dr. Michael Diehl
1. Berichterstatter:	Prof. Dr. Thomas Kropf
2. Berichterstatter:	Prof. Dr. Wolfgang Rosenstiel

Dedicated to

Amma and Nanna

First your parents, they give you your life, but then they try to give you their life.

- Chuck Palahniuk

Acknowledgements

Coming together is a beginning. Keeping together is progress. Working together is success.
- Henry Ford

I would like to thank Prof. Wolfgang Rosenstiel and Prof. Thomas Kropf for giving me the opportunity to do research and pursue Ph.D. studies in Formal Methods Group. In particular, I acknowledge the assistance and logistical support provided by Prof. Rosenstiel. I strongly believe he is the personification of a perfect professor. Apart from technical issues, he also helped me in solving bureucratical issues like VISA extensions, etc. My special thanks to Thomas Kropf. He is impeccable and always talks to the point. Healthy, positive and constructive criticism he does during presentations makes us improving our confidence and lifts our morale. We persons in individual become more tough and do goal oriented research work. He also nurtured how to be a good research scientist. I am grateful to Dr. Jürgen Ruf for his constant encouragement. Ups and downs are all part and parcel of the research. During lows Jürgen was always there for me and supported me. Thanks to his excellent coding skills and the tools he developed. Most of the times, the code he implemented is self-explanatory. Thereby, I learned lot of inside depths of formal verification. My sincere thanks to Dr. Roland J. Weiss for some of his original ideas in my Ph.D. thesis. He motivated me to be more challenging and competitive at work. During his stay in Tübingen, he had inculcated me a sense of responsibility in writing worthful papers and good organizing skills. After all he is a nice human being and made my stay very special at Tübingen. I am particularly grateful to Dipl. Ing. Ingo Kühn. He has inspired and motivated me to do research while I was doing Masters thesis at AMD Saxony, Dresden. My special thanks to all the members of Formal Methods Group (FMG) for the care with which they reviewed the original manuscript of my thesis. In particular, I would like to thank Dr. Prakash Mohan Peranandam. He has assisted me in resolving initial problems of mine concerning research direction. In many ways I have used both his technical and personal experiences. These experiences made my life much easier in Tübingen. I express my deep gratitude also to M.Sc. Djones Lettnin and Dipl. Inf. Jörg Behrend. It was fun working with them. I must acknowledge my colleagues Dipl. Inf. Michael Bensch and Dipl. Inf. Dominik Brugger. Michael was characteristically generous in taking time to review many of my research papers and this thesis. Dominik is truly inspirational figure and he has provided encouragement and ideas in applying concepts from different realms into formal verification of software. I really appreciate the support provided by the folks at Technische Informatik. Their friendship and professional collaboration meant a great deal to me. I really enjoyed their company in ski-seminars, collective outings, afternoon lunch, etc. I must acknowledge my wife for patience and understanding she has shown during final

phase of my thesis writing. My special thanks to my sister and her family for supporting and taking care of my parents during my period of stay in Germany. I am thankful to German Research Council (DFG), German Federal Ministry of Education, Science, Research and Technology (BMBF) and Edacentrum for providing financial support to pursue my research studies in Germany. Last but not least, thanks to www.google.com(/de) and www.wikipedia.com(/de). They made our life much easier to access the enormous amount of information through world wide web.

Finally I would like to make a note: *It has been a very long journey I have taken, but a very fulfilling one.*

Abstract

Today, verification of industrial size designs like multi-million gate ASICs (Application Specific Integrated Circuit) and SoC (System-on-a-Chip) processors consumes up to 75% of the design effort. The trend to augment functional verification with formal verification tries to alleviate this problem. Efficient property checking algorithms based on binary decision diagrams (BDDs) and satisfiability (SAT) solvers allow the automatic verification of medium-sized designs. However, the steadily increasing design sizes still leave verification as the major bottleneck, because formal methodologies do not yet scale to very large designs.

To address these problems researchers came up with the idea of combining symbolic simulation and bounded model checking on-the-fly. The current tools pioneer in handling comparatively larger designs by partitioning the state set and they can be represented using partitioned ordered BDDs (POBDDs). These partitions will be explored in a divide-and-conquer manner. However, still they face memory exhaustion for very large models due to the BDD explosion problem. Even the SAT based bounded model checking (BMC) can search up to a maximum depth allowed by the physical memory on a single server. These observations motivated the parallelization of symbolic state space traversal algorithms. Distributed algorithms verify larger models and return results faster than sequential versions. Existing schemes for parallelizing BDD-based verification algorithms often suffer from state overlap or duplicate work, cross over states among partitions, inefficient work distribution, improper load balancing, synchronicity and communication overhead. The algorithms concentrate heavily either on reachability (validation) or falsification but not both together.

My main contributions include a *dynamic overlap reduction* and a *hybrid* distributed algorithms. The dynamic overlap reduction technique smoothens the state space traversal of each network node by removing the state overlap that it suffers from. The removal of overlap works in an asynchronous manner, i.e., with out waiting for other processors to complete their image computations. This method has the natural side effect of dynamic load balancing among network nodes, i.e., the nodes that deal with a large state space at one time point will be later assigned a small state space and vice versa. Since all the nodes perform asynchronous state space traversal on their whole state subsets, the method is best suitable for validation.

The hybrid method is an asynchronous distributed algorithm, suited for both fast error detection and complete validation. This approach combines well known *windowing* and *dynamic overlap reduction* techniques. The windowing technique has partitions that are identified by unique combination of variables. Each *window* restricts its state space at regular intervals to keep the reachable state space within it's *window* region. The real state space is discriminated by the window as owned and non-owned states. The nodes on the cluster machine are employed

with two different types of tasks. Some nodes, *windows* aim at faster falsification on the basis of the windowing technique. The other type of nodes called *helpers* are intending for validation on the basis of the dynamic overlap reduction. All the network nodes asynchronously traverse their local state spaces for both error states detection and reachability of a time bound. Thus, the *hybrid* algorithm efficiently combines both *windowing* and *dynamic overlap reduction* techniques to obtain more synergy and gains the advantages of using both the approaches. Further, the algorithm expedites the verification process by reassigning the work to idle nodes as quickly as possible. As a result it avoids the wasted computation power and makes the system work efficient.

The dynamic overlap reduction and hybrid algorithms are best suitable for homogeneous system configurations like a cluster. However, this thesis also presents a grid-based parallelization algorithm which is suitable for fast falsification of very large designs. In addition, all the parallel algorithms in this thesis partition the state space using the *Minimal overlap* algorithm which pioneers in statically minimizing the cross over states or state overlap among the partitions.

The parallel algorithms speedup the distributed verification and automatically checks the correctness of very large hardware designs. The distributed computation shows approximately linear speedups in execution time and enables faster verification of properties. As a supplement, this thesis also presents a novel distributed algorithm, which uses mixed forward and backward traversal mechanism for Black Box verification.

Zusammenfassung

Heutzutage benötigt die Verifikation von industriellen Designs wie z.B. ASICs (Application Specific Integrated Circuit) und SoC (System-on-a-Chip) bis zu 75 % der Entwicklungskosten. Der Trend funktionale Verifikation durch formale Verifikation zu erweitern versucht dieses Problem zu lösen. Effiziente Algorithmen basierend auf BDDs (Binary Decision Diagrams) und SAT (Satisfiability) erlauben die automatische Verifikation von mittelgroßen Designs. Durch die ständig wachsenden Designgrößen bleibt die Verifikation trotzdem der Flaschenhals, da formale Methoden noch nicht für sehr große Designs skalieren.

Um diese Probleme zu beheben wurde in der Forschung die Idee entwickelt *symbolic simulation* und *bounded model checking* direkt zu kombinieren. Aktuelle Tools sind in der Lage vergleichsweise große Designs durch Partitionierung des Zustandsraums zu bewältigen. Der Zustandsraum wird dabei auf partitionierte geordnete BDDs (POBDDs) abgebildet und mit einer Teile-und-Herrsche Strategie durchsucht. Trotzdem kämpfen diese Tools immer noch mit Speicherproblemen bei sehr großen Modellen bedingt durch den bei BDDs auftretenden Speicherüberlauf. Auch SAT-basierende *bounded model checking* Tools (BMC) sind in der Suchtiefe beschränkt, durch den physikalisch zur Verfügung stehenden Hauptspeicher eines einzelnen Rechenknotens. Diese Beobachtungen motivierten die Parallelisierung der Algorithmen zur Traversierung von symbolischen Zustandsräumen. Die verteilten Algorithmen können größere Modelle verifizieren und liefern die Ergebnisse schneller als die sequentiellen Versionen. Existierende Methoden zur Parallelisierung von BDD-basierenden Verifikationsalgorithmen kämpfen oft mit Zustandsraumüberlappungen, ineffizienter Verteilung schlechter Lastbalancierung, sowie Synchronisierungs- und Kommunikations-Overhead. Die Algorithmen konzentrieren sich entweder auf Erreichbarkeit (Validierung) oder auf das schnelle Finden von Fehlern aber nicht auf beides zusammen.

Meine Hauptarbeit beinhaltet eine dynamische Reduzierung der Überlappung (*dynamic overlap reduction*) und einen hybriden verteilten Algorithmus. Die Technik zur dynamischen Reduzierung von Überlappungen verbessert die Traversierung des Zustandsraumes von jedem Rechenknoten durch Reduzierung der Überlappungen. Die Überlappungs-Reduzierung arbeitet asynchron d.h. ohne zu warten bis andere Knoten ihre Image Berechnung beendet haben. Diese Methode hat den natürlichen Seiteneffekt der dynamischen Lastbalancierung zwischen den Netzwerkknoten, d.h. die Knoten, die zu Beginn einen großen Teil des Zustandsraums bekommen, werden im nächsten Durchlauf einen kleineren Teil bekommen und vice versa. Da alle Knoten eine asynchrone Exploration ihres eigenen Zustandsraums durchführen ist diese Methode sehr gut zur Validierung geeignet.

Bei der hybriden Methode handelt es sich um einen asynchronen verteilten Algorithmus, der sowohl geeignet ist zum schnellen Auffinden von Fehlern als auch zur vollständigen Verifikation. Der Ansatz kombiniert bereits bekannte *windowing* und *dynamic overlap reduction* Techniken. Der windowing Ansatz benutzt

Partitionen, die durch eindeutige Variablenkombinationen identifiziert werden. Jedes window beschränkt seinen Zustandsraum in unregelmäßigen Abständen, um den Erreichbaren Zustandsraum in den eigenen window Grenzen zu halten. Der gesamte Zustandsraum wird eingeteilt in benutzte und noch nicht benutzte Zustände. Den Knoten des Rechen-Clusters werden zwei unterschiedliche Aufgaben zugeteilt. Einige Knoten, windows, versuchen schnell Fehler zu finden auf Basis der windowing Technik. Der andere Typ von Knoten, helper, versucht zu validieren auf Basic der dynamischen Überlappungs-Reduzierung. Alle Netzwerkknoten traversieren asynchron ihren lokalen Zustandsraum mit zwei Zielen: auffinden von Fehlerzuständen und Erreichbarkeit einer Zeitschranke. Somit kombiniert der hybride Algorithmus auf effiziente Weise sowohl windowing als auch dynamische Überlappungs-Reduzierung, um dadurch mehr Synergien zu erzeugen und um die Vorteile beider Methoden zu nutzen. Der Algorithmus beschleunigt außerdem den Verifikationsprozess durch ein so schnell wie möglich erneute Verteilung der Arbeit auf im Leerlauf befindliche Knoten. Als Ergebnis wird dadurch die Verschwendung von Rechenleistung verhindert und die Effizienz des Systems gesteigert.

Beide Ansätze sind bestens geeignet für homogene System-Konfigurationen wie z.B. Cluster. Außerdem wird innerhalb der Dissertation auch ein Grid-basierender Algorithmus vorgestellt, welcher für die schnelle Fehlersuche in großen Entwürfen geeignet ist. Zusätzlich benutzen alle parallelen Algorithmen in dieser Dissertation den *Minimal overlap* Algorithmus welcher eine statische Minimierung der Kreuz-Überlappungs-Zustände bzw. der Überlappungszustände zwischen den Partitionen durchführt.

Die parallelen Algorithmen beschleunigen die verteilte Verifikation und überprüfen automatisch die Korrektheit von sehr großen Hardware-Designs. Die verteilte Berechnung zeigt annähernd linearen Speedup in der Ausführungszeit und ermöglicht die schnellere Verifikation von Eigenschaften. Als Ergänzung dieser Arbeit wird ein neuer verteilter Algorithmus präsentiert, welcher einen gemischten Vorwärts- und Rückwärts traversierungs-Mechanismus benutzt zur Black-Box-Verifikation.

Contents

1 Introduction **1**

1.1 What is Verification 1

1.2 Simulation 2

1.3 Formal Verification 3

1.3.1 Equivalence Checking 4

1.3.2 Theorem Proving 4

1.3.3 Model Checking 4

1.3.4 State Explosion Problem 7

1.3.5 Thesis Focus 8

1.3.6 Distributed Verification 8

1.4 Thesis Structure 10

2 Preliminaries **12**

2.1 Boolean Functions 12

2.1.1 Support Set 12

2.1.2 Minterms 13

2.1.3 Cofactor 13

2.1.4 Shannon Expansion 13

2.1.5 Generalized Cofactor 13

2.2 Finite State Machines 14

2.3 Binary Decision Diagrams (BDDs) 14

2.3.1 Notations 16

2.3.2 Partitioned-ROBDDs 16

2.4 Temporal Logics 18

2.4.1 CTL* 18

2.4.2 CTL 19

2.4.3 LTL . . . 20
2.4.4 Application of Temporal Logics to Formal Verification . . . 20
2.5 Symbolic State Machine Traversal . . . 21
2.5.1 Transition Relation . . . 21
2.5.2 Image and Pre-image Computations . . . 22
2.5.3 The Traversal Algorithm . . . 23
2.6 Formal Bounded Property Checking Tool - *SymC* . . . 24
2.6.1 FLTL and AR-automata . . . 24
2.6.2 AR-automata . . . 26
2.6.3 SymC Overview . . . 27
2.6.4 Optimizations . . . 28
2.7 Partitioned Based Traversal . . . 30
2.8 Black Box Verification . . . 30
2.8.1 Verification Goals . . . 31
2.8.2 Transition Types . . . 32

3 State-of-the-art **33**
3.1 Sequential Based Optimizations . . . 33
3.1.1 Symbolic Verification Using Partitioning Techniques . . . 33
3.1.2 Mixed Traversals . . . 35
3.1.3 Symmetry Reduction and Underapproximation . . . 36
3.1.4 State-of-the-art Black Box Verification . . . 37
3.1.5 Guided Search . . . 39
3.1.6 Multithreaded Reachability . . . 39
3.1.7 Partitioning in SymC . . . 40
3.2 Distributed Verification . . . 40
3.2.1 Distributed Explicit Model Checking . . . 40
3.2.2 Distributed SAT and SAT based BMC . . . 41
3.2.3 Distributed Symbolic Reachability . . . 41
3.2.4 Grid-based Bounded Model Checking . . . 43
3.3 Partitioning Heuristics . . . 43
3.3.1 Approximation and Decomposition . . . 44
3.3.2 Grumberg et al. Partitioning Heuristic . . . 44

3.3.3 Eager Decomposition 45
3.3.4 Minimal Overlap 45
3.3.5 Guiding Heuristics 46
3.4 Un-addressed Problems 47
3.4.1 State-of-the-art Sequential Problems 47
3.4.2 State-of-the-art Parallel Problems 49
3.5 Contributions 52

4 Parallelization **55**
4.1 Parallelization of a Symbolic Bounded Property Checking 55
4.1.1 State Set Decomposition in Parallel 56
4.1.2 Counterexample Computation 58
4.1.3 State Set Overlap 59
4.2 Conclusion 60

5 Treatments for State Set Overlap **62**
5.1 Overlap Reduction 62
5.1.1 Static Overlap Reduction 63
5.2 Distributed Checking Algorithm 65
5.2.1 Dynamic Overlap Reduction 66
5.2.2 Counterexample Computation 70
5.2.3 Limitations of Dynamic Overlap Reduction Method 72
5.3 Conclusion 72

6 Hybrid Distributed Approach **74**
6.1 Methodology 74
6.1.1 Window States and Cross over States 75
6.2 The Distributed Algorithm 77
6.2.1 Window Node Algorithm 78
6.2.2 Helper Node Algorithm 79
6.2.3 Coordinator Node Algorithm 79
6.3 Counterexample Computation 80
6.4 Window Variable Selection 81
6.4.1 Partitioning Algorithm 81

6.5 Conclusion 82

7 Grid-based Fast Falsification 90

7.1 Why Grid 90

7.1.1 Grid Computing Overview 90

7.1.2 Grid Application 91

7.2 Grid-based Distribution Algorithm 91

7.2.1 Underapproximation using Guiding Algorithms 94

7.3 Conclusion 95

8 Parallelization of Black Box Verification 96

8.1 Black Box Verification 96

8.1.1 Why Parallelization 97

8.2 Combination of Forward and Backward Traversal on a Distributed Platform 97

8.2.1 Backward Node Algorithm 99

8.2.2 Forward Node Algorithm 101

8.2.3 Coordinator Node Algorithm 103

8.3 Conclusion 105

9 Implementation 106

9.1 Ingredients Needed for the Parallelization 106

9.2 Data Transmission 107

9.2.1 Blocking Communication 107

9.2.2 User-defined BDD Data Type 109

9.3 BDD Transmission 111

9.3.1 BDDs to Sequence of Bytes Conversion 111

9.3.2 BDDs Construction From Sequence of Bytes 112

9.4 The Grid Framework 114

9.4.1 UNICORE 114

9.5 Conclusion 116

10 Experimental Results 117

10.1 Details of Experiments 117

10.1.1 Parallel Environment 117

10.1.2 Implementation Details 118
10.1.3 Checked Designs 118
10.1.4 Checked Properties 118
10.2 Verification Using a Single Processor 119
10.3 Distributed Verification 119
10.3.1 Basic Parallelization 119
10.3.2 Dynamic Overlap Reduction 122
10.3.3 Hybrid vs Dynamic Overlap Reduction 124
10.3.4 Variable Ordering Comparison 127
10.4 State-of-the-art Comparison 128
10.4.1 Falsification 129
10.4.2 Reachability 132
10.5 Grid-based Fast Falsification 133
10.5.1 Grid Configuration 133
10.5.2 Discussion 135

11 Conclusions and Future Work **137**
11.1 Conclusions 137
11.1.1 Benefits to the Industry 139
11.2 Future Work 139

Chapter 1

Introduction

In my experience, there is only one motivation, and that is desire. No reasons or principle contain it or stand against it.
- Jane Smiley

The increasing importance of automated formal verification in industry is driving a growing interest in those aspects that directly impact the applicability to real world problems. One of the main technical challenges lies in devising tools or methodologies that allow to handle large design state spaces. Over the last years various approaches have been developed. Recently, an increasing interest in parallel and distributed verification has been emerged. Parallel processing can help speedup the verification process by providing many CPUs that can work on the problem. The aim of this thesis is to enable efficient distributed verification by minimizing the problems that would come across during parallel verification.

This chapter first introduces the basic concepts of verification. Second, it explicates different forms of verification and their differences. Third, it details the prevalent problem faced by the formal verification. Next, it discusses the form of verification this thesis focuses on and details the thesis synopsis. Finally, it gives an exposition on distributed verification and factors that influence the effectiveness of verification in a distributed environment.

1.1 What is Verification

Today, hardware and software systems are widely used in applications where failure is unacceptable [1]. Recent examples of such notorious failures are available in [2, 3]. These failures can range from mild annoyance to major catastrophes and the cost - to individuals, to companies, to society as a whole - can likewise be immense [4]. These hazards result from the fact that the relentless increase in technology allows us to produce increasingly complex designs, which contain an increasing number of bugs. Therefore, the need for reliable hardware or software systems is critical. Verification is a process used to demonstrate the functional

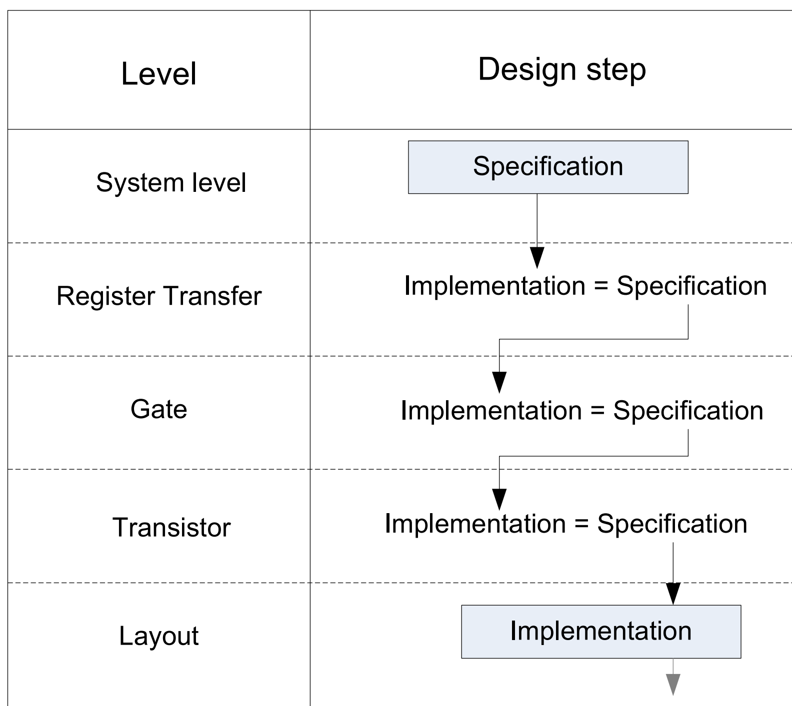

Figure 1.1: Design flow.

correctness of a design. However, today in the era of multi-million gate Application Specific Integrated Circuits (ASICs), reusable Intellectual Properties (IP), System-on-a-Chip (SoC) designs, verification consumes up to 70% of the design effort and the code that implements the verification process makes up to 80% of the total design project code volume [5]. The principal verification methods for complex systems are simulation and formal verification. Simulation checks whether the design exhibits the proper behavior as elicited by a series of functional tests, whereas formal verification uses formal proofs to demonstrate the validity of a design against formal specification. Fig. 1.1 demonstrates the typical refinement principle of circuit design. The design process is divided into several steps, where the implementation resulting on a certain abstraction level is used as the specification for the next lower one [6]. The verification task at each level is to check the implementation against a specification. The register transfer level (RTL) of the circuit is typically described using hardware description languages (HDLs) like Verilog and VHDL.

1.2 Simulation

The conventional verification method to discover design errors is *simulation*. The design is usually described using a hardware description language, which can be simulated. Input sequences are created which reflect the critical execution traces of the design to examine the design functionality. However, in order to get full confidence in the design we would have to perform a complete simulation which

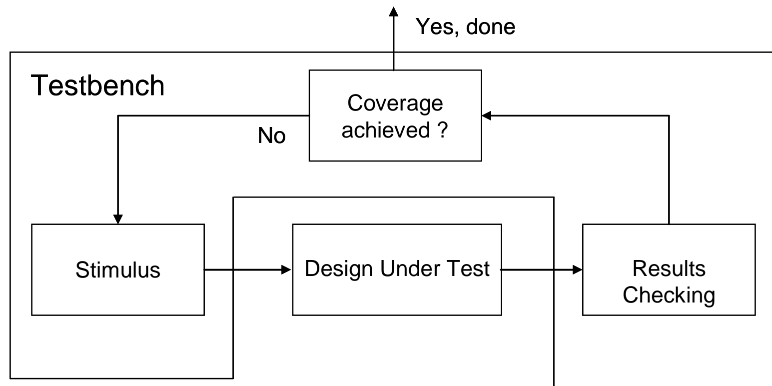

Figure 1.2: Overview of a testbench.

covers all possible input combinations. In case of sequential circuits, the number of input sequences increases exponentially with the number of inputs and the circuit states, it is a laborious task even for designs of moderate size to exercise all possible input sequences. This problem is typically referred as the incomplete *coverage* problem. Simulation consists of four major tasks: 1. Generating the functional tests; 2. Executing the test stimulus on the design; 3. Determining if the design behavior satisfies its specification during the execution; 4. Collecting coverage statistics. The second task, execution, is performed by a simulator. The other three tasks are performed by a testbench. The key ingredients of a testbench are shown in Fig. 1.2. Nowadays, people are combining temporal assertions with *simulation*, for example in [7, 8, 9, 10]. Temporal assertions are properties based on time that are evaluated within an execution engine, i.e., a simulator. Verification based on assertions is known as *assertion-based verification* methodology. It has the ability to monitor internal signals and catch violations locally. Furthermore, it improves observability and debug ability. In many ways it is different from a pure testbench based verification methodology, where output sequences are manually checked against the specification. It leverages the power of formal verification by adopting some internal formal semantics.

1.3 Formal Verification

Formal verification is the act of proving correctness of a system with respect to a formal specification or a *property* using formal methods. In contrast to traditional simulation, it covers exhaustively all the possible executions of a system. However, formal verification can be further divided into three broad categories: *equivalence checking*, *theorem proving*, and *model checking*. Because of the *coverage* problems in *simulation*, formal verification plays a complementary role as far as the whole verification process is concerned.

1.3.1 Equivalence Checking

Equivalence checking is a process of comparing two circuits supposed to have identical behavior in order to verify the correctness of logic optimization, register retiming, state re-encoding, etc. The problem of *equivalence checking* can be converted into the problem of reachable states of the product state machine. Given two state machines to compare, the equivalence checking process ties the input lines of both machines together, sends the outputs to a comparator and provides the clock. This combination is just another bigger state machine. The original two state machines have identical behavior if and only if the new state machine indicates the outputs are equal for all reachable states [1]. However, the complexity of *equivalence checking* continues to increase exponentially with the design size.

1.3.2 Theorem Proving

In *theorem proving*, the design and the properties are expressed as formulas by using mathematical logic. A property is proved if it can be derived from the design using a set of axioms and inference rules. Sometimes the proof consists of intermediate definitions and lemmas in addition to the axioms and rules. Although *theorem proving* has several applications like software verification [11, 12, 13], mathematics, and security/cryptographic protocols, it requires a great deal of skilled human guidance due to insufficient automation. For example, if theorem prover uses first order or higher order logic then proof problems are semidecidable or undecidable and human guiding is needed. In addition, it lacks support for digital design languages and is hard to compose systems and proofs.

1.3.3 Model Checking

Applying model checking [1, 14] to a design consists of two main tasks *modelling* and *specification*. The task *modelling* is to covert a system or design into a formalism, typically an FSM, accepted by a model checking tool. It require the use of abstraction techniques to eliminate unimportant details of the system. The *specification* is usually given in *temporal logic*. A *temporal logic* is a formal language to express properties changing over time. Most predominantly used temporal logics are Computation Tree Logic (CTL) and Linear Time Temporal Logic (LTL) [15]. Each logic has its own expressive capabilities. The detailed explanation on these logics is postponed to the next chapter. Model checking provides a means to check whether the model of the design satisfies a given specification. There are two main paradigms for model checking: explicit state model checking and symbolic model checking. Fig. 1.3 visualizes an overview of a model checking process.

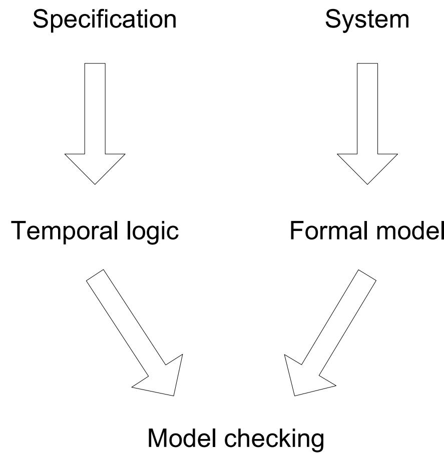

Figure 1.3: Overview of a model checking process.

1.3.3.1 Explicit State Model Checking/Enumerative Model Checking

Explicit state model checking uses an explicit representation of the system's global state graph, usually given by a state transition function. An explicit state model checker evaluates the validity of the temporal properties over the model by interpreting its global state transition graph. The property validation amounts to a partial or complete exploration of the state space. During the exploration the explicit model checker treats states one by one. Tools like SPIN [16] and PV [17] are well known explicit model checkers.

1.3.3.2 Symbolic Model Checking

Symbolic simulation [18] combines conventional simulation with symbolic methods. The advantage of conventional simulation is accuracy, but it needs one simulation vector at a time. In Fig. 1.4, we would require four simulation runs with inputs 00, 01, 10 and 11. Symbolic simulation adds two innovations to conventional logic simulations [19]. The first innovation of a third logic value X represents the *unknown* value. Setting an input to X gives the effect of simulating the circuit with both 0 and 1. As a result, one can reduce the number of simulation runs. However, the value X loses information. In Fig. 1.4 setting one or both inputs to X yields an X at the output, which is an futile result for verification. The simulation with values X, is called as X-simulation. The prominent second innovation is the symbolic simulation. In contrast to X-simulation, symbolic simulation assigns a symbolic value for each input in the design, that can be either 0 or 1, rather than to a constant 0,1 or X . The symbolic simulator computes symbolic expressions for each output in terms of input variables. In Fig. 1.4, assume we set input x to 1 and input y to the symbolic value z. The symbolic simulator would then calculate that the upper AND gate will set to 0 and lower AND gate will set to symbolic value z. Finally, the OR gate will settle to z. Thus, we have effectively run two simulation vectors (xy equal to 10 and 11) and computed the output as a function

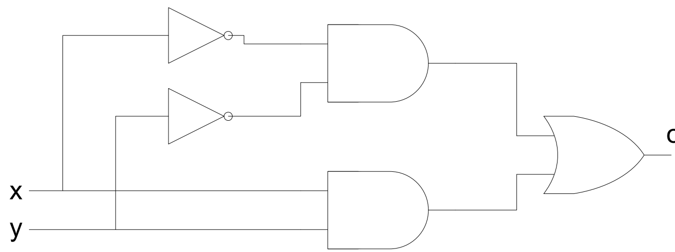

Figure 1.4: A simple XNOR circuit.

of symbolic values. Typically, a BDD (Binary Decision Diagram) [20] is used to represent the values on wires as functions of the symbolic values. In practice, it is the users choice to decide on conventional simulation, X-simulation or symbolic simulation. In conventional simulation we have to exercise test runs one by one, which is a time consuming process. Assigning an input to X in X-simulation reduces the number of simulation runs, but loses the information so that simulation results might not be useful. Symbolic simulation also reduces the number of simulation runs and does not lose information, but too many symbolic values will make the BDDs too large to build. Symbolic model checking means using BDDs in the model checking algorithm. Model checking lets us verify that a state machine or formal model obeys a property we specify using temporal logic. The key idea is to systematically explore the state space of a finite state machine in order to check that the given temporal logic formula holds of a machine. The detailed explanation on temporal logic is postponed to next chapter. In simple words *symbolic model checking* [21, 22, 23, 24] typically uses the symbolic simulation algorithm based on BDDs. Ken McMillan et al. were the first to create a publicly available symbolic model checker based on BDDs [25]. The detailed explanation on BDDs is deferred to the next chapter.

It is accepted that *symbolic model checking* is suited for verifying hardware systems and *explicit state model checking* has advantages of verifying concurrent or asynchronous software [26].

1.3.3.3 Bounded Model Checking based on SAT Solvers

In *bounded model checking* [27, 28, 29], a Boolean formula is satisfiable if and only if the underlying state transition system comprehends a finite sequence of state transitions that reaches certain states of interest. In simple words: we reduce the model checking problem to propositional satisfiability for a finite number of simulation cycles. The idea of BMC is to unroll the sequential circuit into k time-frames of a combinational circuit representing the state transition and output function, and counterexamples are searched in this unrolled system description. If no bug is found then one increases k until either a bug is found or some defined upper time bound is reached. For example, we have a transition system M, a temporal logic formula p and a time bound k. Then we construct a propositional formula, which will be satisfiable if and only if the formula f is valid along

some computation path of M with length k. A propositional formula F is formed as follows. For the state transition system M and time bound k, the unrolled transition system is

$$F = I(q_0) \wedge \bigwedge_{i=0}^{k-1} T(q_i, q_{i+1}) \wedge \bigvee_{i=0}^{k-1} p(q_i)$$

where $I(q_0)$ is the characteristic function of the set of initial states q_0, $T(q_i, q_{i+1})$ is the characteristic function of the transition relation and $p(s_i)$ is the characteristic function of the property that we wish to check. A characteristic function [6] indicates whether an element x of a set $\mathcal{A}$ is part of a subset $\mathcal{B}$ of $\mathcal{A}$. Characteristic functions are functional representations of a subset.

Definition 1 (Characteristic function) *Given two sets $\mathcal{A}$ and $\mathcal{B}$ with $\mathcal{B} \subseteq \mathcal{A}$. A characteristic function $X_{\mathcal{B}} : \mathcal{A} \rightarrow \mathcal{B}$ is defined as*

$$X_{\mathcal{B}}(x) = \begin{cases} \textit{True, if } x \in \mathcal{B} \\ \textit{False, if } x \notin \mathcal{B} \end{cases}$$

The robustness and the capacity increase of *bounded model checking* makes it a good choice for industrial use. This is due to the fact that satisfiability solvers seldom require exponential space. Current solvers use different heuristics in order to solve the general satisfiability problems for *bounded model checking*. All of them use the Davis Putnam algorithm [30] and variants of it. However, *bounded model checking* based on satisfiability solvers is time intensive and it can only search up to a maximum depth allowed by the physical memory on a single server. As the search bound k becomes larger, the memory requirement due to unrolling of the design also increases.

1.3.3.4 BDD vs SAT

Comparing BDD and SAT based *model checking* approaches, each technique has its own advantages and disadvantages. For some particular type of designs BDD based model checkers outperform SAT based model checkers. For others, it is vice-versa. It is greatly a debatable topic to compare both approaches. Therefore, it is the user's choice to select which mechanism to follow. In [31, 32], the authors made comparisons between different approaches and inferred that more synergy can be achieved by combining both techniques.

1.3.4 State Explosion Problem

Formal verification methods reach their limitations very fast. As aforementioned in section 1.2 the state space of the system grows exponentially with respect to the number of flip-flops present in the design. Even a small circuit with 4 registers, each 32 bit wide, exceeds the treatable amount of state space. This is know as

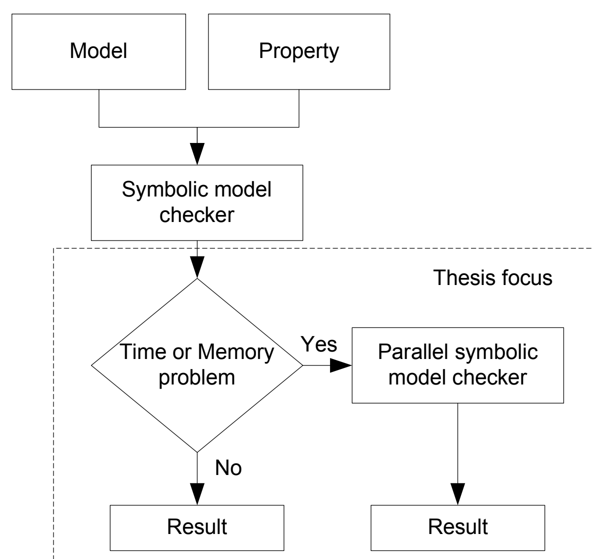

Figure 1.5: Overview of thesis focus

the *state explosion* problem. Therefore, it is impossible to explore the entire state space of the system with limited resources of time and memory. Over the years considerable effort has been made in dealing with this prevalent problem. Albeit researchers have made noticeable progress to deal this problem [33, 34, 35, 31], for larger designs the problem still persists.

1.3.5 Thesis Focus

This thesis will focus mainly on formal verification. To be more specific, symbolic model checking based on BDDs. Formal verification provides the exhaustive coverage and has the calibre to uncover subtle bugs in comparison with traditional simulation. However, for large industrial designs these techniques do not scale well due to the *state explosion* problem. One way to handle this problem is to use sequential based partitioned approaches. For large designs either they consume more time or postpone the *state explosion* problem. One feasible solution is to parallelize the BDD based symbolic model checking algorithm. Fig. 1.5 delineates the wider picture of the thesis focus. If the generic model checker has problems with time or memory then we can enable the parallel version of the symbolic model checker.

1.3.6 Distributed Verification

Two approaches can be considered in fighting the state explosion problems [36]: clever methods or brute-force methods.

The first category can use any of the following methods:

1. Design of better modeling languages (small languages, formal semantics, built-in abstractions, compositionality properties) [37, 38, 39]

2. Invention of better verification algorithms by operating on higher-level models (abstractions, data flow analysis, reductions) or by exploiting [40, 41] structural information or by avoiding redundancies (partial orders, symmetries) [42, 43, 44] , or by using locality (bounded-memory algorithms) [45].

The brute force methods lead to the use of a more powerful machines (increase memory and processing power to handle larger state spaces or use a supercomputer) or the use of several machines instead of one (combine the resources of several machines). In an ideal case, using n machines can solve the problem n times faster.

Often, people have access to large parallel computers, but cannot make full use of them because most model checkers are designed for single processor systems. The aim in exploiting a parallel or distributed environment for verification is to extend the applicability of verification algorithms to larger and more complex systems. A parallel super computer, grid or network of workstations can provide extra resources needed to tackle realistic verification problems.

Cluster computer: [46] A cluster computer is a type of parallel or distributed processing system, which consists of a collection of interconnected standalone computers together as a single integrated computing resource.

A computer node can be a single or multiprocessor system (PCs, workstations, Symmetric Multi Processors (SMPs)) with memory, I/O facilities, inter connection, and an operating system. A cluster refers to two or more computers (nodes) connected together. The nodes can exist in a single cabinet or be physically separated and connected via a LAN, etc. In general two types of computer cluster systems are widely used:

Homogeneous cluster: A homogeneous computer cluster is a group of coupled computers of equal configurations.

Heterogeneous cluster: A heterogeneous cluster is a collection of machines of varying architectures.

Distributed computation can help to speed up verification by providing many CPUs that can work on the problem and provide higher aggregated memory capacity.

The main idea of a distributed algorithm is to break the verification task into pieces and assign each piece among the participating workstations. The verification process on all nodes is terminated once at least one of the nodes detected a bug or error state. This way we can achieve a parallel, iterative, and interactive

verification process. In general, computation on a smaller subset requires less memory compared to the whole set. This method enables us to find errors that are far,transitions away from the initial states. Moreover, parallel computation takes less verification time compared to the sequential approach. However, there are several factors that significantly influence the overall effectiveness of verification in the distributive environment. These factors have to dealt with a nontrivial manner as far as parallel model checking is concerned. Some of them include:

Load balancing: This is a technique to spread work between many processes or other resources in order to get optimal resource utilization and to decrease computing time. With respect to model checking, each network node is assigned approximately the same number of states, thus achieving good speed-up.

Synchronization: Refers to the idea that multiple processes are to join up at a certain point to ensure correctness or to reach an agreement or to commit to a certain sequence of action.

Communication: Parallel processes typically need to exchange data in order to efficiently accomplish the tasks assigned to them. The lesser the communication involved, the better the parallelization we can achieve.

1.4 Thesis Structure

This thesis is structured as follows:

Chapter 2: Sheds light on basics that will be deployed to construct the methodologies described in the thesis.

Chapter 3: Deals with state-of-the-art sequential, parallel and partitioning heuristics. Next, it discusses a few of the open problems that need to be treated. Finally, it covers the thesis contributions.

Chapter 4: Describes the basic parallelization algorithm for bounded property checking. It covers the raw materials needed for the basic parallelization. It also introduces the prevalent problem called *state set overlap*.

Chapter 5: Presents some techniques for treating *state set overlap*. It explicates the application of static overlap reduction on a distributive environment. The overlap can be confined to only a certain number of simulation steps using static reduction technique. Therefore a better algorithm is needed to remove the overlap. It describes a novel on-the-fly asynchronous state space traversal algorithm using *dynamic overlap reduction* technique.

Chapter 6: Presents a hybrid distributed symbolic verification algorithm based on *windowing* and *dynamic overlap reduction* techniques, suited for full validation and fast falsification. The approach is mainly asynchronous.

Chapter 7: Expounds a new grid based distributed bounded symbolic verification approach based on effective combination of state-of-the-art intelligent partitioning algorithms that suits best for fast falsification.

Chapter 8: Elucidates the parallelization of a Black Box verification for incomplete designs. The work described in this chapter was done as a cooperative task between University of Tübingen and University of Freiburg under the research project FEST (Funktionale Verifikation von Systemen) [47] in the year 2007.

Chapter 9: Explicates all the implementation details. First it explains the ingredients needed for the parallelization. Second, it explains the methods that can transfer the data between network nodes. In addition, it describes an efficient way to communicate BDD functions among network nodes. Finally, it gives the components required for the grid framework.

Chapter 10: Gives experimental results. First it details the utilized parallel environment, designs and properties. Next, it delineates the advantage of parallel approach over sequential based verification. Then first it compares the dynamic overlap reduction and hybrid distributive approaches and second it compares the algorithms presented in thesis with state-of-the-art parallel algorithms. Finally, it presents the results using the grid based distributive approach.

Chapter 11: Briefly presents the concluding remarks for this thesis and provides directions for future work.

Chapter 2

Preliminaries

Success is neither magical nor mysterious. Success is the natural consequence of consistently applying the basic fundamentals.
- Jim Rohn

2.1 Boolean Functions

A Boolean function describes how to determine a Boolean value output based on a logical calculation from Boolean inputs. It plays a crucial role in hardware verification.

Definition 2 *A Boolean function with m inputs is a mapping $f : B^m \rightarrow B$, where B = [0,1] is a Boolean domain and m is a positive integer.*

Boolean functions are used to express formalisms in hardware verification. For example transition relations or output functions of finite state machines are Boolean functions [6]. Boolean functions can be represented in various ways, e.g. as function tables, formulas of propositional logic and graphs. A Boolean function is called *valid* if it results in the value *true* for all interpretations of Boolean variables. The function is called *satisfiable* if there exists at least one interpretation which results in the value $true$.

2.1.1 Support Set

Definition 3 *The set of variables which constitute a Boolean function f is called support of f.*

2.1.2 Minterms

Definition 4 *A Boolean function $f : B^n \rightarrow B$, a product term in which each of the n variables appears once (either complemented, or uncomplemented) is called a minterm.*

A product term is a conjunction of literals, where each literal is either a variable or its negation

2.1.3 Cofactor

Definition 5 *The cofactor of a Boolean formula f with respect to a variable v is the formula obtained by replacing every occurrence of of v in f by constant 1 and is denoted by $f\big|_v$ or f_v.*

f_v is called as *positive cofactor* and $f_{\bar{v}}$ is called as *negative cofactor*, where every occurrence of v in f is replaced by constant 0.

2.1.4 Shannon Expansion

Definition 6 *Given a Boolean function $f : B^n \rightarrow B$, f is partitioned into two functions f_1 and f_2 on a variable v.*

$$f = f_1 \vee f_2 \text{ where } f_1 = v \wedge f_v,\ f_2 = \bar{v} \wedge f_{\bar{v}}$$
$$f_v = f \wedge v$$
$$f_{\bar{v}} = f \wedge \bar{v}$$

.

2.1.5 Generalized Cofactor

Definition 7 *[6] Given two functions $f : B^k \rightarrow B$ and $c : B^k \rightarrow B$, where B = [0,1]. The function $co(f, c)$ or $f\big|_c$ is called generalized cofactor f with respect to c, if the following conditions holds.*

$$f\big|_c = \begin{cases} \text{False} & \text{if c = False} \\ f & \text{if c = True} \\ \text{True} & \text{if c = } f \\ \text{False} & \text{if c = } \neg f \end{cases}$$

This definition can be applied to reduce the sizes of BDDs in the context of state space traversal techniques. And it will be explained in detail in later chapters of this thesis.

2.2 Finite State Machines

Typically sequential circuits are modeled as Finite State Machines (FSMs).

Definition 8 *A FSM $\mathcal{M}$ is a 6-tuple $\mathcal{A} = (S, I, \delta, \lambda, S_0, O)$, where $S = \{s_1, \ldots, s_n\}$ is a finite set of states, I is a finite alphabet, $\delta : S \times I \rightarrow S$ is the next state function, $\lambda : S \times I \rightarrow O$ is the output function, $S_0 \subseteq S$ is the initial state, $O \subseteq S$ is a set of final states.*

The formal model deals with the Boolean domain. The operators $\wedge$, $\vee$, $\neg$, $\rightarrow$ and $\leftrightarrow$ used for Boolean conjunction, disjunction, negation and implication and equivalence respectively. Where $a \rightarrow b$ can also be written as $\overline{a} \vee b$ and $a \leftrightarrow b$ can be written as $(a \rightarrow b) \wedge (b \rightarrow a)$

Usually, FSMs are modelled using **Mealy** or **Moore** machines.

Moore Machine: A Moore machine is a state machine which uses only entry actions so that its output depends on the current state alone ($\lambda : S \rightarrow O$) [48]. An entry action is an action that is performed *when entering* the state. The state diagram for a Moore model will include an output signal for each state. The number of states in a Moore machine will be greater than or equal to the number of states in the corresponding Melay machine [49].

Mealy Machine: A Mealy machine is a state machine which uses only input actions, so that the output depends only on the current state and also on inputs ($\lambda : S \times I \rightarrow O$) [50]. An input action an action that is performed depending on the present state and input conditions. The use of a Mealy FSM leads often to a reduction of number of states [49].

For each Mealy machine there is an equivalent Moore machine whose states are the union of Mealy machine's states and the Cartesian product of the Mealy machine's states and the input alphabet. The choice of machine type depends on the application and personal preference of the designer.

2.3 Binary Decision Diagrams (BDDs)

A BDD is a data structure for representing a Boolean function. Bryant first [51] introduced BDD in its current form.

Definition 9 *A BDD is a rooted, directed acyclic graph. Conceptually, a BDD for a Boolean function can be built, obeying the following restrictions [52] :*

1. *One or two terminal nodes of out-degree zero labeled 0 or 1.*
2. *A set of variable nodes u of out-degree two. The two outgoing edges are given by two functions $if(u)$ and $else(u)$.*

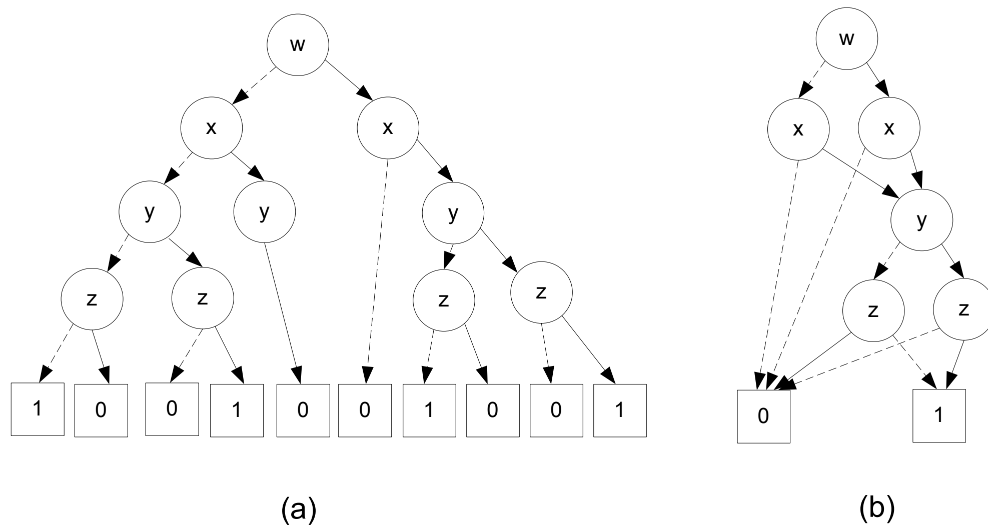

Figure 2.1: (a) An example decision diagram. (b) An ROBDD with variable ordering $w < x < y < z$. Dashed lines in both figures denote $else$ branches and solid lines represent if branches.

3. *Along any path from the root to a terminal node, no variable appears more than once.*

Definition 10 *A BDD is ordered (OBDD) if on all paths through the graph the variables respect a given linear order* $x_1 < x_2 < ... < x_n$

Fig. 2.1 (a) shows the Boolean expression $t = (w \leftrightarrow x) \wedge (y \leftrightarrow z)$ as a binary decision diagram.

Definition 11 *A (O)BDD is Reduced (R(O)BDD) if it conformed to the following rules:*

Uniqueness: *No distinct nodes u and v have the same variable name and if and $else$ successors are shown in Fig. 2.2 (a) , i.e.,*

$$\left.\begin{array}{c} var(u) = var(v) \\ if(u) = if(v) \\ else(u) = else(v) \end{array}\right\} \Rightarrow u = v$$

Non-redundant tests: *No variable node u has identical if and $else$ successor, i.e.,*

$$if(u) \neq else(u)$$

Fig. 2.2 (b) denotes this rule. ROBDDs provide compact representations of Boolean expressions. However, ROBDDs may grow exponentially with respect to the number of variables. Efficient algorithms exist for performing all kinds of operations (AND, OR, NOT, etc.) on ROBDDs.

Variable ordering: The size of an ROBDD for a Boolean function is largely influenced by the used variable ordering. However, finding an optimal ordering for a

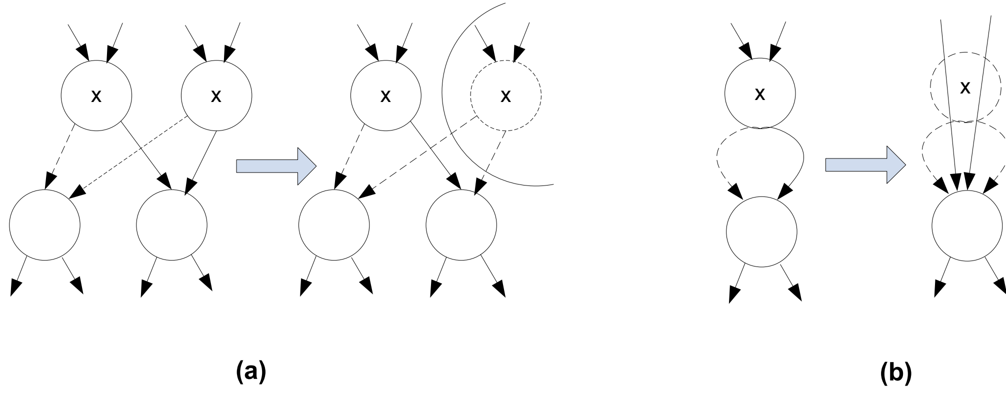

Figure 2.2: Reduction rules for constructing the ROBDD.

given Boolean function is an NP complete problem. Fig. 2.1 (b) shows a BDD for $(w \leftrightarrow x) \wedge (y \leftrightarrow z)$ with ordering $w < x < y < z$. Therefore, in practical applications heuristics are used. There are two kinds of heuristics available for finding variable orderings [6] : *static* and *dynamic approaches*. Static heuristics [53] derive the ordering from analyzing the structural information of the circuit before the ROBDD is constructed. Whereas, dynamic approaches [54, 55, 56, 57] make the graph more compact by changing the original variable ordering during or after the ROBDD construction. Once the variable order is fixed, a BDD is a canonical representation for a Boolean function. Every distinct Boolean function has a exactly one unique BDD representation. There exist several publicly available BDD packages [58, 59, 60]. Each package associated with functions that implement *dynamic variable reordering* , efficient vectorized BDD operations, automatic garbage collection, interfaces with different programming languages, etc.

2.3.1 Notations

Let S be a state set represented using BDD. Then $|S|$ denotes the number of BDD vertices or nodes and $\|S\|$ denotes the number of states in S, which is given by the number of maximal minterms of the BDD.

2.3.2 Partitioned-ROBDDs

The idea of *partitioning* was introduced and discussed extensively in [61, 62].

Definition 12 *[63] Given a Boolean function $f : B^n \rightarrow B$, defined over n input variables $X_n = \{x_1, ..., x_n\}$, the POBDD representation $\mathcal{X}_f$ of f is a set of k function pairs, $\mathcal{X}_f = \{(w_1, f_1), ..., (w_k, f_k)\}$ where, $w_i : B^n \rightarrow B$, $f_i : B^n \rightarrow B$ are also defined over X_n and satisfy the following conditions:*

1. w_i and f_i are boolean functions, for 1 ≤ i ≤ k.
2. $w_1 \vee w_2 \vee \ldots \vee w_k$ = 1.

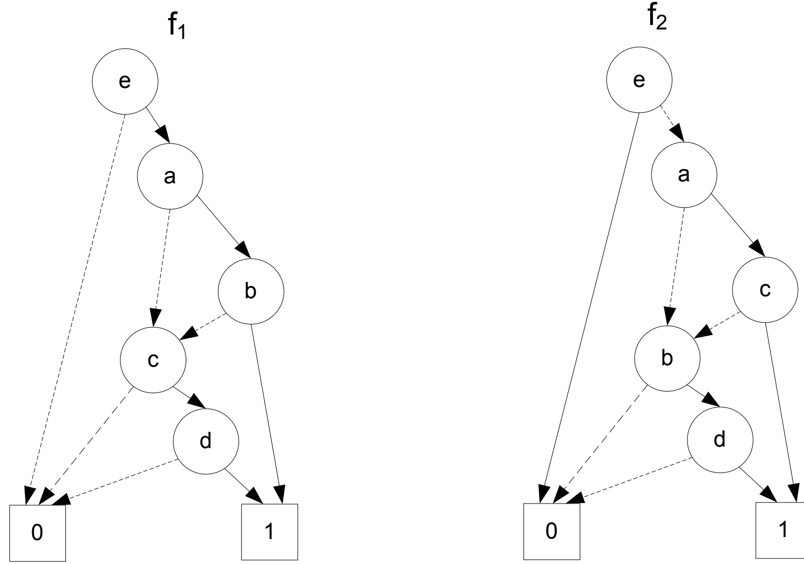

Figure 2.3: Partitioned BDDs f_1 and f_2

3. $w_i \wedge w_j$ = 0, for i $\neq$ j.

4. f_i = $w_i \wedge f$, for 1 $\leq$ i $\leq$ k.

Each w_i is called a *window* function that can be used to partition the Boolean space over which f is defined. In Partitioned-ROBDDs the Boolean space is divided into k partitions using window functions ($w_i \in W$). The functionality of f is represented over each partition as a separate ROBDD f_i. Boolean operations can be efficiently performed on them just like ROBDDs. They can be exponentially more compact than ROBDDs for certain classes of functions.

A simple example BDD f:

$$f = e \wedge \big((a \wedge b) \vee (c \wedge d)\big) \bigvee \overline{e}\big((a \wedge c) \vee (b \wedge d)\big)$$

with window functions e and $\overline{e}$ can be partitioned into two POBDDs f_1 and f_2:

$$f_1 = e \wedge \big((a \wedge b) \vee (c \wedge d)\big)$$
$$f_2 = \overline{e} \wedge \big((a \wedge c) \vee (b \wedge d)\big)$$

The Fig. 2.3 depicts the corresponding POBDDs with different variable reorderings.

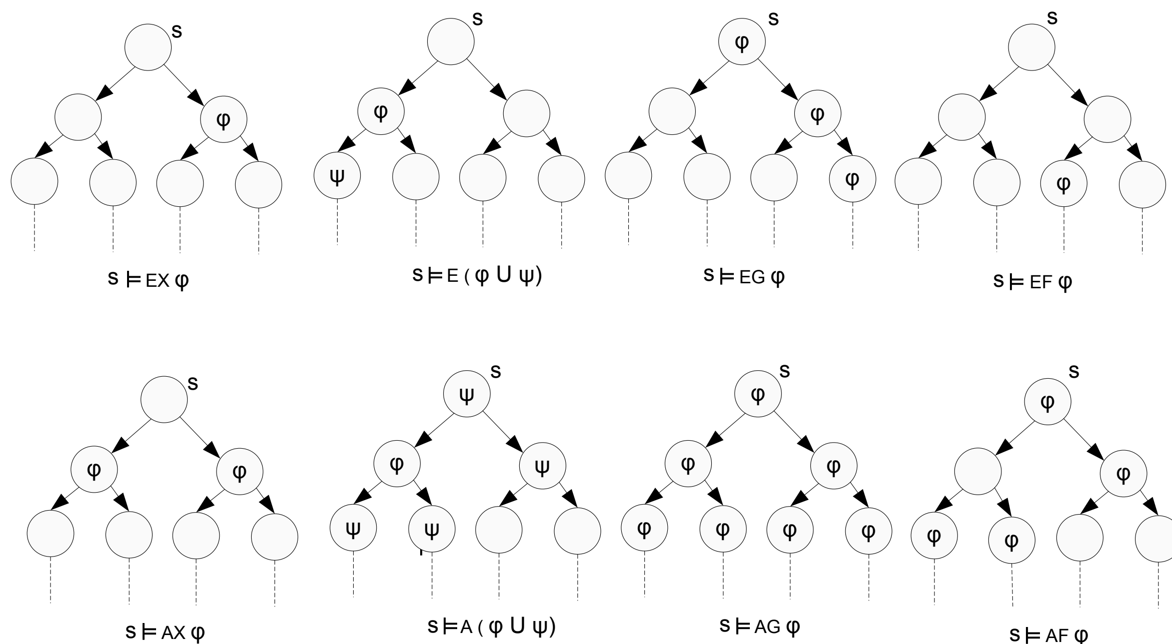

Figure 2.4: The base operators of CTL.

2.4 Temporal Logics

Temporal logic is a formal way of expressing properties that change over time [6]. It describes sequences of transitions between states in a reactive system [1]. More precisely the logic reason about variables with truth values which change over time. A reactive system is a system that changes its actions, outputs and conditions/status in response to stimuli from within or outside it. Temporal logics have the ability to reason about a time line. The most predominantly used temporal logic is CTL* (Computation Tree Logic*). It contains two sublogics CTL (Computation Tree Logic) and LTL (Linear Time Temporal Logic), which are used mostly in practice [15].

2.4.1 CTL*

CTL* formulas describe properties of *computation trees*. The tree is formed by unwinding the Kripke structure into an infinite tree with the initial state as root. Where a Kripke structure is a type of nondeterministic finite state machine used in model checking to represent the behavior of a system. It is basically a graph whose nodes represent the reachable states of the system and whose edges represent state transitions. Temporal logics are traditionally interpreted in terms of Kripke structures.

CTL* formulas are composed of *path quantifiers*:

- A (for all computation paths)

- E (for some computation path)

and *temporal operators*:

- X (next time) property holds in the next state of the path.
- F (eventually) property will hold at some state on the path.
- G (globally) property holds at every state on the path.
- U (until) combines two properties. It holds if there is a state on the path where the second property holds, and at every preceding step, the first property holds.

There are two types of formulas in CTL* : state formulas and path formulas. Let AP be the set of atomic proposition names. The syntax of state formulas is given by the following rules:

- If $p \in AP$, then p is a state formula.
- If f and g are state formulas, then $\neg f$, $f \vee g$ and $f \wedge g$ are state formulas.
- If f is a path formula, then $\mathbf{E}\ f$ and $\mathbf{A}\ f$ are state formulas.

The syntax of path formulas:

- If f is a state formula, then f is also a path formula.
- If f and g are path formulas, then $\neg f$, $f \vee g$ and $f \wedge g$, $\mathbf{X}\ f$, $\mathbf{F}\ f$, $\mathbf{G}\ f$ and f $\mathbf{U}\ g$ are path formulas.

The logics CTL and LTL can be derived by applying certain restrictions to CTL*.

2.4.2 CTL

CTL is a *branching-time* logic. In CTL the temporal operators (**X**,**F**,**G** and **U**) quantify over the paths that are possible from a given state. CTL is the subset of CTL* that is obtained by restricting the syntax of path formulas using the following rule:

- If f and g are state formulas, then $\neg f$, $f \vee g$ and $f \wedge g$, $\mathbf{X}\ f$, $\mathbf{F}\ f$, $\mathbf{G}\ f$ and f $\mathbf{U}\ g$ are path formulas.

The eight CTL base operators are depicted in Fig. 2.4.

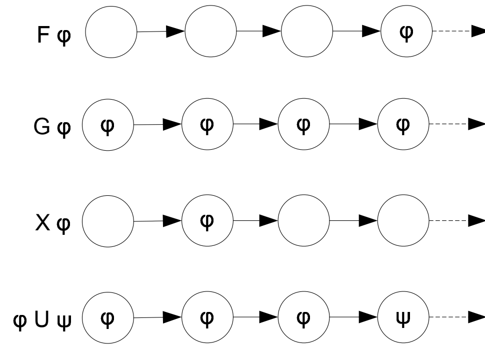

Figure 2.5: Semantics of the LTL operators.

2.4.3 LTL

LTL is a *linear-time* logic. LTL consist of formulas that have the form $\mathbf{A}\ f$ where f is a path formula in which the only state subformulas permitted are atomic propositions. More precisely, an LTL path formula is either

- If $p \in AP$ then p is a state formula
- If f and g are path formulas, then $\neg f$, $f \vee g$, $f \wedge g$, $\mathbf{X}\ f$, $\mathbf{F}\ f$, $\mathbf{G}\ f$ and $f\ \mathbf{U}\ g$ are path formulas.
- If f is a path formula, then $\mathbf{E}\ f$ and $\mathbf{A}\ f$ are state formulas.

The semantics of the LTL operators is visualized in Fig. 2.5.

Each logic is neither less expressive nor more expressive than the other. For example [1], the LTL formula $\mathbf{A}(\mathbf{FG}p)$, which says that along all paths there is some state from which p will hold forever, cannot be expressed using CTL. Similarly, the CTL formula $\mathbf{AG}(\mathbf{EF}p)$, cannot be expressed in LTL. The formula $\mathbf{A}(\mathbf{FG}p) \vee \mathbf{AG}(\mathbf{EF}p)$ can be expressed only using CTL* but neither CTL nor LTL.

Two kinds of properties are primary importance and have been exhaustively used as far as verification of hardware and software systems are concerned: *safety properties* and *liveness properties*. Safety properties assert that something bad never happens. Liveness property assert that something good will eventually happen.

2.4.4 Application of Temporal Logics to Formal Verification

Temporal logic has found an important application in formal verification, where it is used to state requirements of hardware or software systems In CTL model checking, based on a CTL formula we systematically explore the system state space that satisfies the formula. The exploration process continues until we find a fixed-point condition. The fixed-point is a condition we met where we have no more new states to explore. In LTL model checking, we express desired properties using LTL and check if the model satisfies this property. One technique

is to obtain an automaton that is equivalent to the property. Later the property automaton is combined with the automaton that is derived from the model. The state space traversal is performed on the product automaton in order to check the correctness of the property on-the-fly. Most widely used automatons to represent the LTL property are Büchi automata [64] and AR-automata [65].

2.5 Symbolic State Machine Traversal

Symbolic state machine traversal is the core computation in design verification. In symbolic verification, states are represented as sets. State transitions of the automaton are represented by a *transition relation*. Sets are represented by characteristic functions. Set operations are performed by Boolean operations on characteristic functions, representing the sets. The state machine traversal makes extensive use of *transition relation*, *image computation* and *pre-image computation*. In the following we will see the definitions of these three prominent elements of state space traversal.

2.5.1 Transition Relation

For a synchronous circuit with m state variables and k input variables the sets $E = \{q_1, ..., q_m\}$, $E' = \{q'_1, ..., q'_m\}$ and $I = \{x_1, ..., x_k\}$ consist of present state, next state and input variables, respectively. The partitioned transition relation [66] is constructed based on the piece of combinational logic that determines how a state variable q_i is updated. Let f_i be the next state function computed by this logic, then q_i's value in the next state is given by $q_i{}'$ = $f_i(E, I)$. These equations in turn define the whole *transition relation* T as

$$T(\overrightarrow{q}, \overrightarrow{x}, \overrightarrow{q}') = T_1(\overrightarrow{q}, \overrightarrow{x}, q'_1) \wedge \ldots \wedge T_m(\overrightarrow{q}, \overrightarrow{x}, q'_m)$$
$$\text{where } T_i(\overrightarrow{q}, \overrightarrow{x}, q'_i) = (q'_i \equiv f_i(\overrightarrow{q}, \overrightarrow{x}))$$

Let us consider a small example [67] to construct the transition relation T. The sequential circuit in Fig. 2.6 is a modulo 8 counter. Let $E = \{q_0, q_1, q_2\}$ be the set of state variables and $E' = \{q'_0, q'_1, q'_3\}$ be the set of next state variables. The transitions of the module 8 counter are given by:

$$T_0(E, q'_0) = (q'_0 \equiv \overline{q}_0)$$

$$T_1(E, q'_1) = (q'_1 \equiv (\overline{q}_0 \oplus q_1))$$

$$T_2(E, q'_2) = (q'_2 \equiv \overline{(}q_0 \wedge q_1) \oplus q_2)$$

The above equations describe the constraints each q'_i must satisfy in a legal transition. These constraints can be combined by taking their conjunction to form the transition relation.

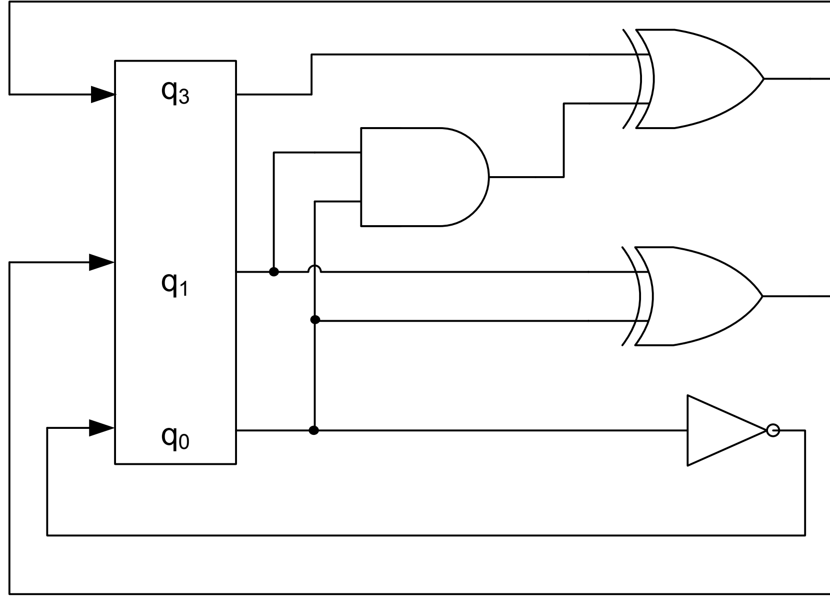

Figure 2.6: Synchronous modulo 8 counter

$$T(E, E') = T_0(E, q_0') \wedge T_1(E, q_1') \wedge T_2(E, q_2')$$

Given a BDD for each transition T_i, it is straightforward to compute the *monolithic* BDD that represents T. The transition relation is monolithic because it is represented by a single BDD.

2.5.2 Image and Pre-image Computations

Image computation computes $next$ states from a given set of states.

The image of a set of states is described by its characteristic function $S(\overrightarrow{q})$ according to the transition relation T is defined as:

$$image\big(S(\overrightarrow{q}), T\big) = \Big(\exists \overrightarrow{x} \Big(\exists \overrightarrow{q} \big(T(\overrightarrow{q}, \overrightarrow{x}, \overrightarrow{q}') \wedge S(\overrightarrow{q})\big)\Big)\Big)\Big|_{\overrightarrow{q}' \leftarrow \overrightarrow{q}}$$

The operation $[\overrightarrow{q}' \leftarrow \overrightarrow{q}]$ denotes the renaming operation of replacing each next variable q_i' with current state variable q_i.

Similarly, *pre-image computation* computes the set of $present$ states given a set of next states.

$$pre - image\big(S(\overrightarrow{q}'), T\big) = \Big(\exists \overrightarrow{x} \Big(\exists \overrightarrow{q}' \big(T(\overrightarrow{q}, \overrightarrow{x}, \overrightarrow{q}') \wedge S(\overrightarrow{q}')\big)\Big)\Big)\Big|_{\overrightarrow{q} \leftarrow \overrightarrow{q}'}$$

In *pre-image computation* the quantification is done with respect to the $next$ state variables. This operation is useful for *backward* traversal.

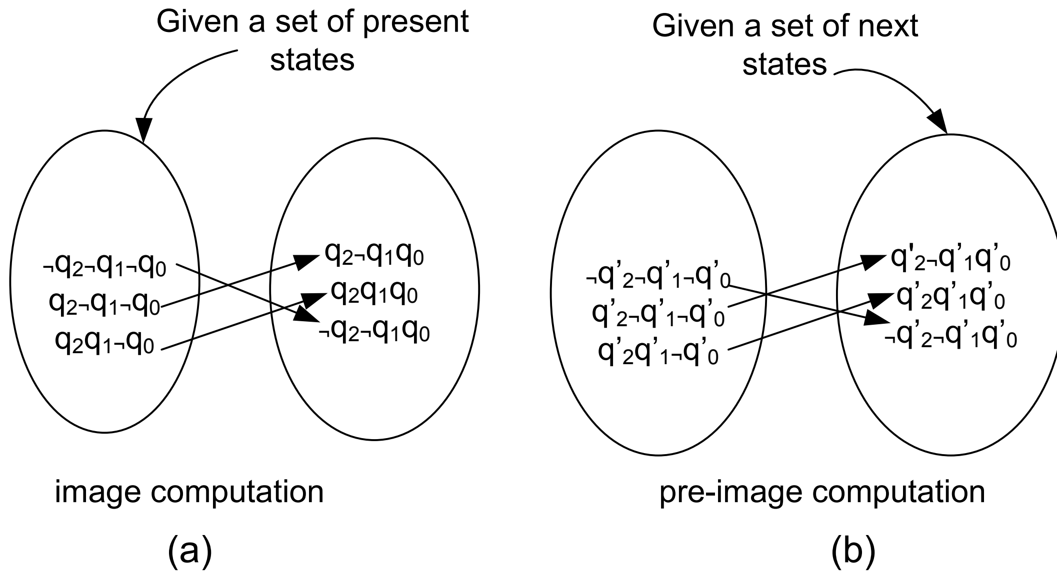

Figure 2.7: Synchronous module 8 counter

The operation $[\overrightarrow{q} \leftarrow \overrightarrow{q}']$ denotes the renaming operation of replacing each current state variable q_i with next state variable q_i'.

Fig. 2.7 visualizes an example *image* and *pre-image* computation steps for the state sets using transition relation T (modulo 8 counter circuit) described in section 2.5.1. Each state in Fig. 2.7 represent a numerical, for instance the state $\neg q_2 \neg q_1 \neg q_0$ represent the numerical 0, state $q_2 \neg q_1 \neg q_0$ represent the numerical 4, etc. So, image computation step increments the numerical specified in the present state set and *pre-image* computation step decrements the numerical specified in the next state set.

2.5.3 The Traversal Algorithm

A standard BDD-based forward symbolic traversal is a breadth-first search that returns at each iteration the set of reachable states from the current state set. In Fig. 2.8, initially S equals S_0. The state set S_n represents newly reached next states that have not been visited. During the image computation step the current state set will be conjoined with all the transition partitions according to the quantification schedule, i.e., an ordering of the conjunctions which minimizes the number of peak variables. The termination condition is to find a least fixed-point or to test for the emptiness of S_n at each step. The number of iterations of this algorithm gives the *sequential depth* of the state machine. Researchers started altering this algorithm slightly in order to handle state space traversal more efficiently.

```
Sr = Sn = S = S0;     // Initial state set      1
iteration = 0;                                   2
while Sn ≠ ∅                                     3
  Sn := image(T,S) - Sr                          4
  Sr := Sr ∨ Sn                                  5
  S := Sn                                        6
  iteration++;                                   7
```

Figure 2.8: Forward state space traversal algorithm.

2.6 Formal Bounded Property Checking Tool - *SymC*

In order to show the efficiency of the algorithms developed in this thesis the tool *SymC* [68, 69] is used. It was developed at the University of Tübingen. *SymC* is an efficient and robust symbolic bounded property checker based on BDDs. The formal verification tool combines bounded property checking and symbolic traversal. It takes a formal model and temporal expressions in PSL foundation language or FLTL (Finite Linear time Temporal Logic) [70]. The formal model, which is the input to *SymC* works only with Boolean variables and consists of five basic blocks. They are **declaration**, **init**, **define**, **trans** and **invariant** blocks. In the first block we can declare both state (latches) and input variables. The state variable list consists of flip-flops present in the design. The input variable list consists of external inputs. In the **init** block we can define initial states of our FSM. The **define** block is composed of the definition of state encodings and some predicate conditions. The variables defined in the **define** block can also be used as output variables. The **trans** block consists of transition relations of all state variables. The final block **invar** is composed of invariant conditions of the system. The description of the formal model is very similar to a Verilog gate list or as a simple SMV like [71] finite state description. The temporal logic formulas are converted to special finite state machines called AR-automata [65]. Fig. 2.10 shows the general operation of *SymC*. In the following subsections we see in detail the semantics of both FLTL and AR-automata.

2.6.1 FLTL and AR-automata

FLTL extends LTL with bounded temporal operators. The main difference however lies in the definition of the formal semantics. LTL is defined over infinite sequences, whereas FLTL is defined over finite sequences. The reason for defining FLTL over finite state sequences comes from its application in bounded property checking.

$$\phi := \ v \mid \neg\phi \mid \phi \wedge \phi \mid X_{[m]}\,\phi \mid F_{[m,n]}\,\phi \mid G_{[m,n]}\,\phi$$

with $v \in$ *Vars*, where *Vars* $= \{a, b, c, \ldots\}$ be a finite set of distinct symbols, called the *variable domain*, $m \in \mathbb{N}$ and $n \in \mathbb{N} \cup \{\infty\}$

Since we deal with a finite traces, the changes in variables can be represented by traces:

A trace $T[n..m]$ $(m \geq n)$ is a mapping $T : \{n, \ldots, m\} \rightarrow 2^{Vars}$. If n and m are clear from the context, we often simply write T instead of $T[n..m]$. The set of all traces is denoted by $\mathcal{T}$. The set of all traces $T[0, m]$ with $m = \infty$ is denoted by $\mathcal{T}^{\infty}$.

Finite traces may be extended. These extensions are used to formally define the semantics of FLTL over a three valued logic.

(Trace extension) Let $T[0, m], T'[0, n]$ be two traces with $n > m$. T' is called a *trace extension* of T if

$$\text{for all } j \text{ with } 0 \leq j \leq m : T(j) = T'(j) \tag{2.1}$$

FLTL formulas are interpreted over traces. First, the satisfiability relation over infinite traces can be defined as:

The satisfiability relation $\models_i \subset (\mathcal{T}^{\infty}, FLTL)$ is defined recursively over the structure of FLTL formulas:

$$\begin{array}{lcl}
T \models_i a & \Leftrightarrow & a \in T(i) \\
T \models_i \neg f & \Leftrightarrow & T \not\models_i f \\
T \models_i f \wedge g & \Leftrightarrow & T \models_i f \text{ and } T \models_i g \\
T \models_i \mathrm{X}_{[m]} f & \Leftrightarrow & T \models_{i+m} f \\
T \models_i \mathrm{G}_{[m,n]} f & \Leftrightarrow & \text{for all } j \text{ with } i + m \leq j \leq i + n \\
 & & \text{holds that } T \models_j f \\
T \models_i \mathrm{F}_{[m,n]} f & \Leftrightarrow & \text{ex. a } j \text{ with } i + m \leq j \leq i + n \\
 & & \text{such that } T \models_j f
\end{array}$$

Where a is a propositional variable, f is a FLTL formula, X, G, F are temporal operators and $m, i \in \mathbb{N}$ and $n \in \mathbb{N} \cup \{\infty\}$. The standard temporal operators (F,G) are special cases of the timed operators by instantiating m, n with 0 and ∞, respectively. The semantics of FLTL is given by the following definition. Let f be a FLTL formula and $T \in \mathcal{T}^{\infty}$ be a trace. T is called to *satisfy* f (i.e., $T \models f$) if $T \models_0 f$.

We now interpret FLTL formulas over finite traces. That is the reason why the logic is called as Finite Linear time Temporal logic. A formula has one of three states with respect to a given trace:

Let $T[0..n]$ be a trace and f be a FLTL formula. f is called **true** with respect to T (denoted by $T \models f$) if for all trace extension $T'[0..\infty]$ of T holds that $T' \models f$. f is called **false** with respect to T if there exists no trace extension $T'[0..\infty]$ of T such that $T' \models f$. Otherwise f is called **pending**.

If a trace ends up with pending, the corresponding formula is neither proven true nor proven false. It depends on the future of the finite trace whether the formula will be satisfied or not.

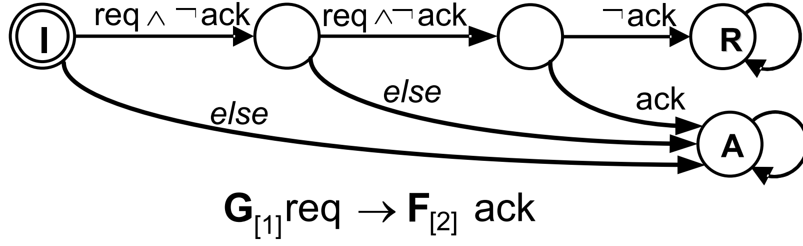

Figure 2.9: Example of an AR-automaton for a simple FLTL property. The states labeled with R, A and I represent rejecting, accepting and Initial states respectively.

2.6.2 AR-automata

The FLTL formulas are translated into AR-automata. The automaton checks the correctness or the violation of this formula against a symbolic simulation run.

Definition 13 *A deterministic is a 5-tuple* $\mathcal{A} = (S, \rightarrow, A, R, s_0)$*. where* $S = \{s_1, \dots, s_n\}$ *is a finite set of states,* $\rightarrow$ *is the deterministic transition relation,* $A \subset S$ *is the set of* accepting states*,* $R \subset S$ *is the set of* rejecting states*, and* $s_0 \in S$ *is the start state of* $\mathcal{A}$*. The input of* $\mathcal{A}$ *are all elements of* 2^{Vars}*.*

We write $s_i \xrightarrow{a} s_j$ *to express that there is a transition from* s_i *to* s_j *labeled with* a*.*

Let $\mathcal{A}$ be an and $T[0..m]$ be a trace. A *run* of T with respect to $\mathcal{A}$ is a sequence of states $s_0, s_1, \dots, s_n$ such that $s_i \xrightarrow{T(i)} s_{i+1}$ holds for $0 \leq i < m$.

Let $\mathcal{A} = (S, \rightarrow, A, R, s_0)$ be a deterministic and $T[0..m]$ be a trace.

- T is called an *accepted trace* if for the run $s_0, s_1, \dots, s_{m+1}$ induced by T, there is a j with $0 \leq j \leq m+1$ with $s_j \in A$ and for all $k < j$ holds $s_k \notin R$. Accordingly, this particular run is called an *accepted run*.

- T is called a *rejected trace* if for the run $s_0, s_1, \dots, s_{m+1}$ induced by T, there is a j with $0 \leq j \leq m+1$ with $s_j \in R$ and for all $k < j$ holds $s_k \notin A$. Accordingly, this particular run is called a *rejected run*.

The main translation of temporal logic formulas into AR-automata works bottom-up based the syntax graph of the formula. The algorithm starts in the leaves and constructs successively more and more complex AR-automata until it reaches the root of the graph. This means at each internal node the computation of a new AR-automaton out of one or two AR-automata (depending on the arity of the logic operator) is necessary. The AR-automata accepts/rejects a trace if a signal is true/false in the current simulation cycle. For a detailed discussion on translation of FLTL into AR-automata refer to [65, 69].

Fig. 2.9 shows the AR-automaton that corresponds to the FLTL property: $G_{[1]}req \rightarrow F_{[2]}\ ack$. This property checks globally that whenever the *req* signal

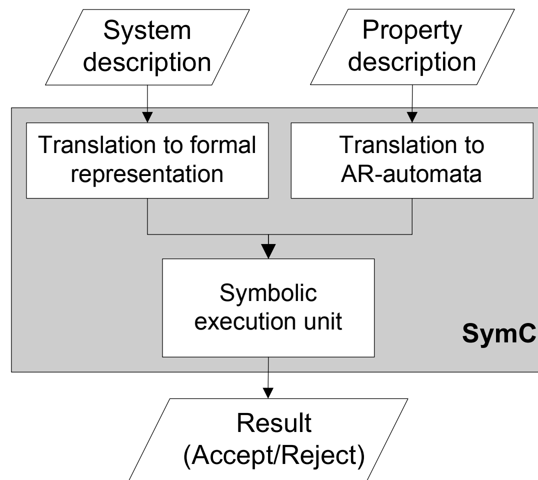

Figure 2.10: Overview of *SymC* operation.

becomes true, signal *ack* triggers within two steps. Notice that because of the globally operator, no accept state is part of the AR-automaton.

2.6.3 SymC Overview

Coming back to the *SymC* exposition, after having both the system description and the AR-automata *SymC* converts them into a BDD form. *SymC* uses a set of conjunctive partitioned transition relations as described in 2.5.1. The structure of the state machine looks like a Mealy state machine. *SymC* traverses the design and the properties simultaneously and observes the state of the properties and reports success or failure to the user.

The bounded property checking algorithm shown in Figure 2.11 works in two steps. In the first step the computation of the successor states of the AR-automata and checking condition of a formula whether it is accepted or rejected takes place. In the second step of each iteration one symbolic execution step on the system under inspection is performed. During image computation the tool builds the conjunction of all partitions on-the-fly in order to obtain the successor state set.

Similar to bounded model checking *SymC* do not traverse the state space exhaustively but from a given start set it examines all states reachable up to a given time bound, which is either given explicitly by the user or implicitly by the property. Traversal proceeds on the current subset until the time bound is reached or the termination condition is satisfied. Termination stops the verification with finding either a validation or a violation of the property. The termination condition differs if one checks the property on all paths, i.e., universal quantification, or on one path, i.e., existential quantification. Informally, the sequential termination condition is defined as follows:

Universal If one reject state is detected in the current state set, a violation of the

property is found. If all states in the current state set are accepting states, a validation of the property is found. Otherwise, the property is still pending.

Existential If one accept state is detected in the current state set, a validation of the property is found. If all states in the current state set are rejecting states, a violation of the property is found. Otherwise, the property is still pending.

The properties that are *universally rejected* and *existentially accepted* are categorized under the falsification scenario. The properties that are *universally accepted* and *existentially rejected* are classified under the full validation scenario. From now onwards *reject* states in case of a *universal* property and *accept* states in case of an *existential* property are indicated as target or error states.

```
S := Sys_start ∧ AR_start;                                   1
while iteration < time_bound                                 2
 S := image_AR(S) // Compute image of AR-automata.          3
 if (check universally)                                      4
    if(S ∧ AR_Reject ≠ ∅)  reportFailure();                  5
    if(S ∧ AR_Accept = S)  reportAcceptance();               6
 if (check existentially)                                    7
    if(S ∧ AR_Accept ≠ ∅)  reportAcceptance();               8
    if(S ∧ AR_Accept = S)  reportFailure();                  9
 S := image_T(S) // Compute image of the system.             10
 iteration++;                                                11
```

Figure 2.11: SymC main computation loop.

2.6.4 Optimizations

Burch et al. [66] have shown that the transition relation of a model can be partitioned into smaller partitions, each handling the next value of a single state variable. Operations with the partitioned transition relation can be iteratively executed, i.e., one partition at a time. Albeit the partitioning has been successfully used in the verification of models, the efficiency of this process strongly depends on the order in which the partitions are executed during the *image computation*. A good order can be derived by examining the model to be verified and its semantics. The ordering heuristics for the partitions were described in [72, 73]. The tool *SymC* applies the ordering of partitions by constructing an early quantification tree.

Definition 14 *[73] A quantification tree is a rooted binary tree, where the leaves are the partitioned transitions, i.e., T_i's. Each node t in the tree contains a set of variables $q(t) \subseteq E$ (where the set E represents the set of current state variables), which occur only in the transitions contained in the subtree rooted at t. The set $q(t)$ represents the set of variables which should be quantified after the left and right children have been multiplied.*

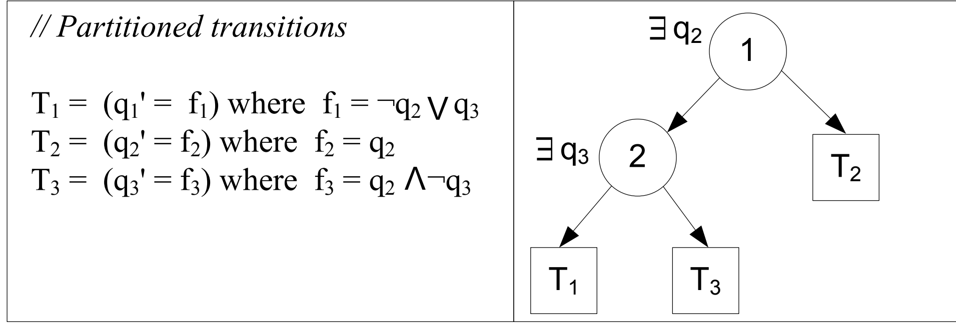

Figure 2.12: The left hand side shows an example partitioned transitions. The right hand side shows their respective early quantification tree.

Fig. 2.12 visualizes an example early quantification tree (right hand side) with respect to the partitioned transitions in the left hand side of the figure. The support set of both T_1 and T_3 consist the common variables q_2 and q_1. Therefore, they categorized under subtree with the root node 2. The variable q_3 present in support set of all three transitions consequently it should be quantified in the last and located in the root node 1 of the tree. Let $S(\overrightarrow{q})$ be a state set if we perform *image computation* using the partitioned transitions in Fig. 2.12 then quantification can be done using the following order:

$$\exists q_2 \Big(\big(\exists q_3 \, \big(S(\overrightarrow{q}) \wedge T_1 \wedge T_3\big) \wedge T_2 \Big)$$

The other optimization *SymC* performs is based on a *Cone Of Influence* (COI) reduction based on the property, i.e., only the logic necessary to define the property under check has to be considered. Usually, the property is influenced by only part of the design, while other parts are irrelevant to its truth or falsity. For example, if the property verifies a design output, only this output and its input cone of logic are necessary. *SymC* identifies unnecessary parts and removes them. Knowing the expected behavior of inputs and outputs allows *SymC* to reduce the design state space further.

The tool performs the state set *partitioning* optimization technique. Boolean functions represent all the state sets in symbolic traversal. In order to reduce the memory requirements one can partition a Boolean function to smaller parts, whose union is the whole set. The basic idea for the partitioning is the *Shannon expansion* defined in section 2.1.4. A given state set S is partitioned into two subsets S_1 and S_2 based on variable v from the support set of S. Fig. 2.13 visualizes the state set S is partitioned into two subsets using the variable q_2. Since state sets are represented and manipulated using BDDs, the partitioning can be implemented easily with BDD operations. Partitioning a function f into two functions f_1 and f_2 with a poor choice of v may not necessarily reduce the memory requirements of the split functions and can result in $|f_1| \approx |f_2| \approx |f|$. Therefore, the variable that we are going to select is always plays a crucial role.

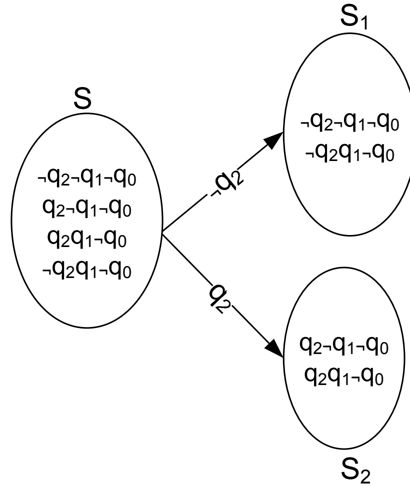

Figure 2.13: State set partitioning based on a variable.

Whenever, the size of the current state set reaches a threshold limit, the tool *SymC* partition the state set into two disjoint parts and continues working on these subsets in a divide-and-conquer manner. One of the partitions is considered for immediate traversal, while the other split is stacked for future exploration. The terminating condition of *SymC* also has to be adopted for this partitioning. The following section explains the partitioned based traversal.

2.7 Partitioned Based Traversal

The partitioned symbolic traversal was first proposed in [34]. Whenever the complexity exceeds a threshold limit, the problem is partitioned into sub-problems whose complexities are smaller. Each sub-problem will be exploited separately and during the exploration if the tool finds a target state then we can skip the traversal on the other partition's. Thus, we can save time and memory. Fig. 2.14 shows the partitioned symbolic traversal. However, if our design is error free or the target state can only be reached using the final partition then we have to explore all the partitions sequentially, which is a painstaking task. The work described in my thesis is similar to the partitioned approach but all the partitions are explored simultaneously in a parallel framework.

2.8 Black Box Verification

Model checking of *incomplete* designs, i.e., designs which contain unknown parts and they can be combined into so called Black Boxes is known as *Black Box* verification [74, 75, 76]. It has several advantages: 1. Instead of forcing the verification runs to the end of the design process where the design is completed, it rather allows model checking in early stages of the design, where parts may not be fin-

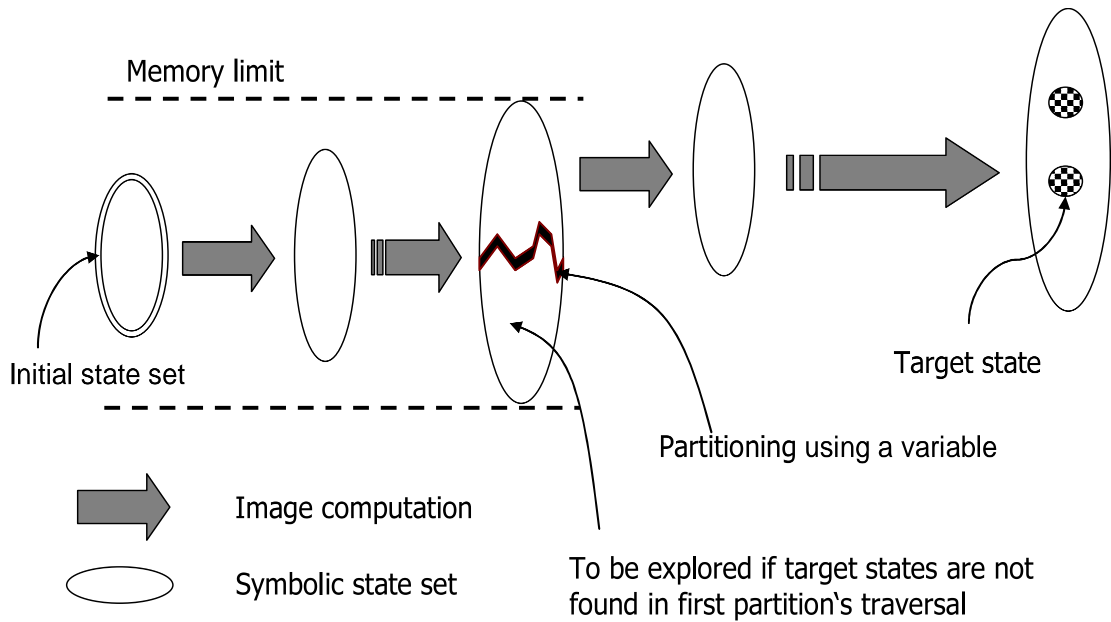

Figure 2.14: Partitioning based symbolic traversal.

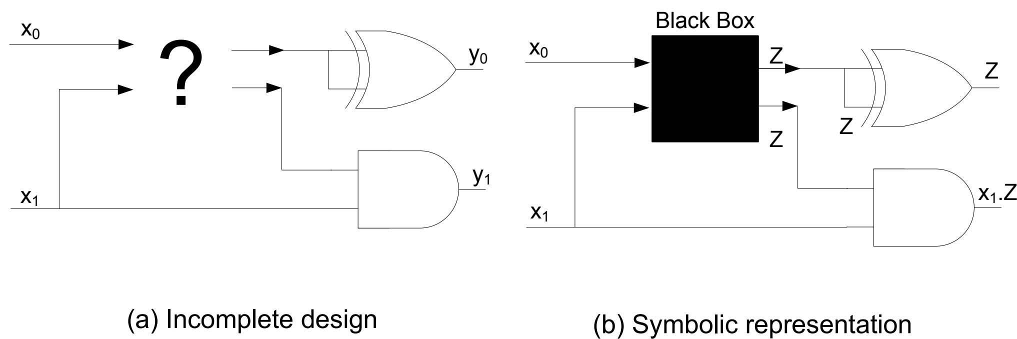

Figure 2.15: An incomplete design.

ished. So, the errors can be detected earlier; 2. Complex parts of a design can be replaced by Black Boxes, simplifying the design, while many properties of the design still can be proven, yet in shorter time; 3. The location of design errors in circuits not satisfying a model checking property can be narrowed down by iteratively masking erroneous parts of the design. Fig. 2.15 (a) shows an example incomplete design with one Black Box. Fig. 2.15 (b) shows the symbolic representation of the incomplete design. Each output of the Black Box is modelled using a symbolic variable Z.

2.8.1 Verification Goals

Given an incomplete design with Black Boxes and a formula, the questions that must be considered are *realizability* and *validity*.

Realizability: Is there a replacement of the Black Boxes in the incomplete design,

so that the resulting circuit satisfies a given formula φ. If this is true, then the property φ is called realizable for the incomplete design. The corresponding decision problem is called the realizability problem.

Validity: Is there a formula φ satisfied for all possible replacements of the Black Boxes. If this is the case, then φ is valid for the incomplete design; the corresponding decision problem is denoted as the validity problem.

For a given incomplete design and a formula φ, approximate symbolic model checking for an incomplete design considers two sets of states: $Sat_A(\varphi)$, an underapproximation of the set of states in which φ is satisfied for all Black Box substitutions and $Sat_E(\varphi)$, an overapproximation of the set of states in which φ is satisfied for at least one Black Box substitution.

Given these two sets, it is potentially possible to prove validity and to disprove realizability. If all initial states lie within $Sat_A(\varphi)$, then for all initial states and all Black Box substitutions, φ is satisfied and thus φ is valid. If there is one initial state lying outside $Sat_E(\varphi)$, then there is a initial state that does not satisfy φ for any Black Box substitution and thus φ is not realizable.

2.8.2 Transition Types

The approximations of $Sat_A(\varphi)$ and $Sat_E(\varphi)$ can be computed based on an approximate transition relation. In incomplete designs we have Black Boxes in the functional block defining the transition function δ and the output function λ. For this reason there are transitions which exist independently from the replacement of the Black Boxes, i.e., for all possible replacements of the Black Boxes (they can be called *fixed transitions*) and transitions which may or may not exist in a complete version of the design - depending on the implementation for the Black Boxes (they can be called *possible transitions*).

The transition relation is classified under two types of approximations. An *underapproximation* T_A containing the fixed transitions and an *overapproximation* T_E containing at least all possible transitions, this includes all fixed transitions. States that resulted through fixed (possible) transitions are called *fixed* (possible) states.

$$T_A(\overrightarrow{q}, \overrightarrow{x}, \overrightarrow{q}') = \forall Z \Big(T_1(\overrightarrow{q}, \overrightarrow{x}, Z, q_1') \wedge \ldots \wedge T_m(\overrightarrow{q}, \overrightarrow{x}, Z, q_m') \Big)$$
$$T_E(\overrightarrow{q}, \overrightarrow{x}, \overrightarrow{q}') = \exists Z \Big(T_1(\overrightarrow{q}, \overrightarrow{x}, Z, q_1') \wedge \ldots \wedge T_m(\overrightarrow{q}, \overrightarrow{x}, Z, q_m') \Big)$$
$$\text{where } T_i(\overrightarrow{q}, \overrightarrow{x}, Z, q_i') = (q_i' \equiv f_i(\overrightarrow{q}, \overrightarrow{x}, Z))$$

Chapter 3

State-of-the-art

Go around asking a lot of damn fool questions and taking chances. Only through curiosity can we discover opportunities, and only by gambling can we take advantage of them.
- Clarence Birdseye

This chapter deals with the state-of-the-art technologies in three different areas: sequential optimizations, distributed verification and partitioning heuristics. First, it details all the sequential based optimizations and also provides some hints on more synergy that can be achieved by parallelizing the conventional verification process. Second, it delineates the existing distributed approaches. Since state partitioning plays a prominent role in distributed verification, the thesis also discusses the contemporary partitioning heuristics. Next, it explicates unaddressed problems in state-of-the-art technologies. Finally, it explains clearly the thesis contributions.

3.1 Sequential Based Optimizations

3.1.1 Symbolic Verification Using Partitioning Techniques

In order to deal with the explosive memory requirements a partitioning approach has been suggested in [61]. By partitioning the system's state space into disjoint subspaces and representing as well as processing all functions in each subspace independently of other subspaces, reduction in time and space can be achieved. The partitioned reachability techniques were proposed in [35]. However, they do not scale well due to the fact that many of the practical issues involved with partitioning were not addressed. Iyer et al. in [77] have shown the usefulness of partitioning technology for reachability and model checking on difficult designs. They dynamically repartition the state space and exploit the partitioned nature of the data structure. In [78] the authors proposed a partitioning methodology that addressed the key questions of how to perform partitioning and provided the suitable algorithms. They also addressed the problem of instability of BDD-based verification by automatically picking the best configuration.

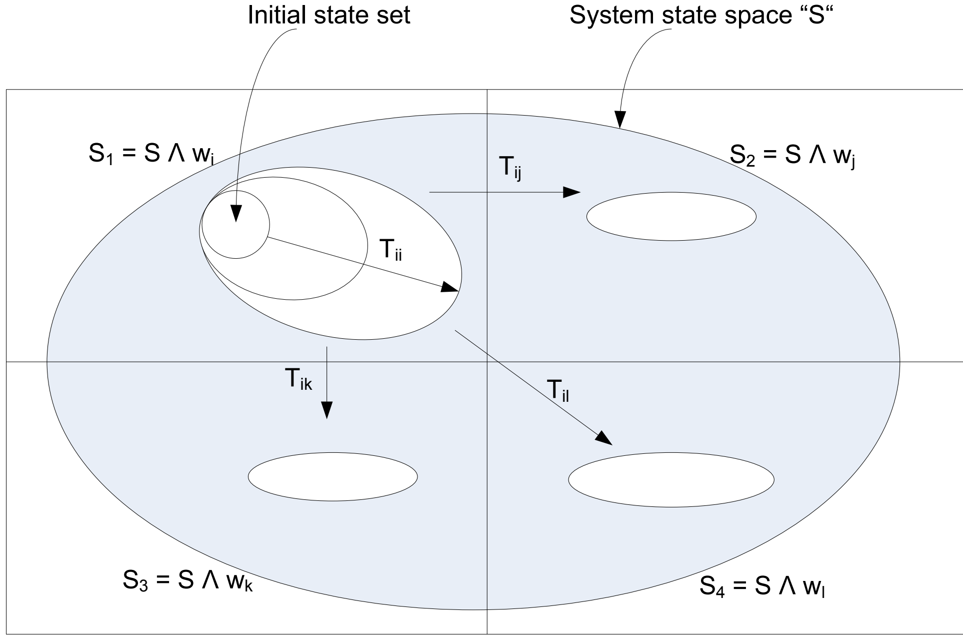

Figure 3.1: The reachability using partitioned approach.

The core idea of the partitioned based verification is to split the system state space and that eventually inducing a partitioning on the transition relation. The transition relation, T_{ij} comprised of transitions from states in partition i to partition j, can be derived by conjoining T with the respective $window$ functions expressed appropriately in terms of present and next state variables as follows:

$$T_{ij}(\overrightarrow{q}, \overrightarrow{x}, \overrightarrow{q}') = w_i(\overrightarrow{q}) \wedge w_j(\overrightarrow{q}) \wedge T(\overrightarrow{q}, \overrightarrow{x}, \overrightarrow{q}')$$

The main algorithm performs image computations within each partition i using T_{ii} until the *least fixed point* condition is reached. When no more images can be computed, it synchronizes between other partitions i.e., for each partition i it computes the states that communicate from partition i to other partitions using TR_{ij}. Fig. 3.1 visualizes an example partitioned based reachability approach. The system state space S is divided among 4 partitions using window functions w_i, w_j, w_k and w_l. The authors in [78] selected window functions based on the goal to create small and 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 they selected by means of a cost function from [35].

3.1.1.1 Partitioned Based Reachability

The POBDD based reachability algorithm is shown in Fig. 3.2. The algorithm performs as many steps as possible of *image computation* within each partition i

using T_{ii}. They named it as a step of *least fixed point* within the partition. When the *least fixed point* conditions of all the partitions are reached, each partition i computes the cross over states with respect to other partitions. This step is named as $communication$, and is performed from partition i to each partition j using the transition T_{ij}.

```
S_r = S_n = S_0;     // Initial state set                          1
iteration = 0; S_co := φ                                           2
while S_n ≠ ∅                                                      3
   for each partition i                                            4
     S_ri = computeLeastFixedPoint(T_ii, S_r);                     5
     S_r := S_r + S_ri;                                            6
   for each partition i                                            7
      for each partition j                                         8
        if i ≠ j                                                   9
          S_co := S_co + computeCrossOverStates(T_ij, S_ri);       10
   S_r := S_r + S_co                                               11
   S_n := S_r - S_n                                                12
  iteration++;                                                     13
```

Figure 3.2: Partitioned based reachability algorithm.

3.1.2 Mixed Traversals

The idea of mixed traversals was introduced by Govindarajulu et al. in [79]. They extended the idea of approximations using overlapping projections [80] to symbolic backward and forward reachability. The main algorithm alternates forward and backward passes; Each pass uses the approximations computed by the other pass. A forward pass finds a subset of the current state set which appears to be reachable from the initial state, while a backward pass finds a subset of the current state which seems to consists of predecessors of the bad states. The process is repeated until a fixed point is reached. If the fixed point is reached, no target states are reachable from the initial state, so the property has been validated. Otherwise, there is a chance in having a path from the initial state to target state. Later a simple heuristic is used to compute a subset of the reachable states from the initial state that is likely to contain a bad state. If it succeeds, the method finds a genuine error and a counterexample path can be computed.

A similar approach was used by Cabodi et al. in [81]. In brief they performed a backward verification procedure based on prioritized traversal and named the method *inbound-path-search*. The algorithm first performs an approximate forward traversal and produces overapproximate *onion-ring* frontier sets. Next, these rings are used as distance estimators and guides to partition state sets in terms of the estimated distance from the target set of states. Finally, while the subsequent search is performed, the higher priority is given to the subset with

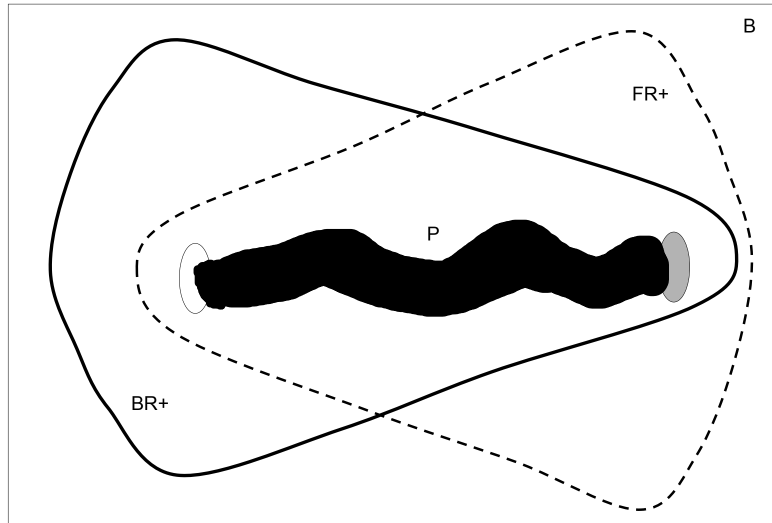

Figure 3.3: Overview of mixed traversals.

the smallest estimated distance.

In [82] the authors combined BDD and SAT-based BMC methods in order to increase the efficiency of the verification process. The basic idea is to help the SAT solver with information collected from a BDD-based reachability analysis. The approach first performs a BDD-based forward breadth-first traversal or a backward breadth-first traversal. A tighter overapproximation is obtained with a forward-backward strategy, where forward estimates of the reachable state set constrain backward traversals and vice-versa. The result of this preliminary phase is an overestimation of the paths leading from the initial state set to the target state set. Then, the collected overestimated reachable state space can be used to restrict the search space of the SAT-based BMC. This is possible by feeding the SAT-solver with a description that is the combination of the original BMC problem with the extra information coming from the BDD-based symbolic analysis. Fig. 3.3 shows the combined (BDD and SAT) approach using mixed traversals. FR+(dashed cone) and BR+(solid cone) represent the overapproximate forward and backward traversals. The state set marked by white colour represents the initial state set and the one in gray represents the target state set. P represents the set of paths from the initial state set to the target state set.

3.1.3 Symmetry Reduction and Underapproximation

In order to reduce space and time [83] presents a collection of methods *symmetry reduction*, *underapproximation* and *symbolic model checking*. However, the work is best suitable for *falsification*. Since many systems consist of several similar components, the authors partitioned the system state space into equivalence classes and named them as $orbits$. The reachability algorithm performs *underapproximation* at each step and explores only a subset of the reachable state. Some of the unexplored states are symmetric to the explored ones and those states will never

be explored. Thus, both memory and time can be saved by running a property automaton together with the symmetry reduced model. For checking the safety properties they used *on-the-fly* model checking. Whereas, for liveness properties they developed two extensions by combining symmetry reduction with classical (not on-the-fly) symbolic model checking. One is for *falsification* and the other is more expensive but for *validation*.

3.1.4 State-of-the-art Black Box Verification

Efficient approximate symbolic model checking algorithms for incomplete designs presented in [74, 75, 76]. For a given incomplete design and a formula φ: $Sat_A(\varphi)$, an underapproximation of the set of states in which φ is satisfied for all Black Box substitutions and $Sat_E(\varphi)$, an overapproximation of the set of states in which φ is satisfied for at least one Black Box substitution. As described in section 2.8, given these two sets, it is potentially possible to prove validity and to disprove realizability. The authors in [75, 76] mentioned that it is not required to perform two separate model checking runs to compute both $Sat_E(\varphi)$ and $SatA(\varphi)$. By using an additional encoding variable e and defining:

$$T(\overrightarrow{q}, \overrightarrow{x}, \overrightarrow{q}', e) = T_A(\overrightarrow{q}, \overrightarrow{x}, \overrightarrow{q}') \bigvee e \wedge T_E(\overrightarrow{q}, \overrightarrow{x}, \overrightarrow{q}')$$

both $Sat_A(\varphi)$ and $Sat_E(\varphi)$ can be computed in parallel. Therefore, it is possible to combine two computations into one and can derive the following formula.

$$Sat(\varphi) = (\overline{e} \wedge Sat_A(\varphi)) \bigvee (e \wedge Sat_E(\varphi)) (= Sat_A(\varphi) \bigvee (e \wedge Sat_E(\varphi)) \text{ due to } Sat_A(\varphi) \subseteq Sat_E(\varphi))$$

3.1.4.1 Computation of $Sat_A(\neg\psi)$ and $Sat_E(\neg\psi)$

$Sat_E(\varphi)$ is an overapproximation of all states in which φ may be satisfied for some Black Box replacement. Thus, for an arbitrary state in the state set S with $S = B^{|\overrightarrow{q}|} \times B^{|\overrightarrow{x}|} \backslash Sat_E(\varphi)$ there is no Black Box satisfied in this state or, equivalently $\neg\varphi$ is satisfied for all Black Box replacements. Therefore, S can be used as an underapproximation, i.e., $Sat_A(\neg\varphi)$. Similarly, $Sat_A(\varphi)$ holds for $Sat_E(\neg\varphi)$. Consequently, we can define:

$$Sat_A(\neg\varphi) = \overline{Sat_E(\varphi)} \text{ and } Sat_E(\neg\psi) = \overline{Sat_A(\psi)}$$

Since both $Sat_E(\varphi)$ and $Sat_A(\varphi)$ can be computed in parallel, it is possible to prove both validity and falsify realizability.

Fig. 3.4 illustrates the backward traversal approach for the Black Box verification. It is noticeable in the figure that each frontier state set consists of both *fixed* and *possible* states.

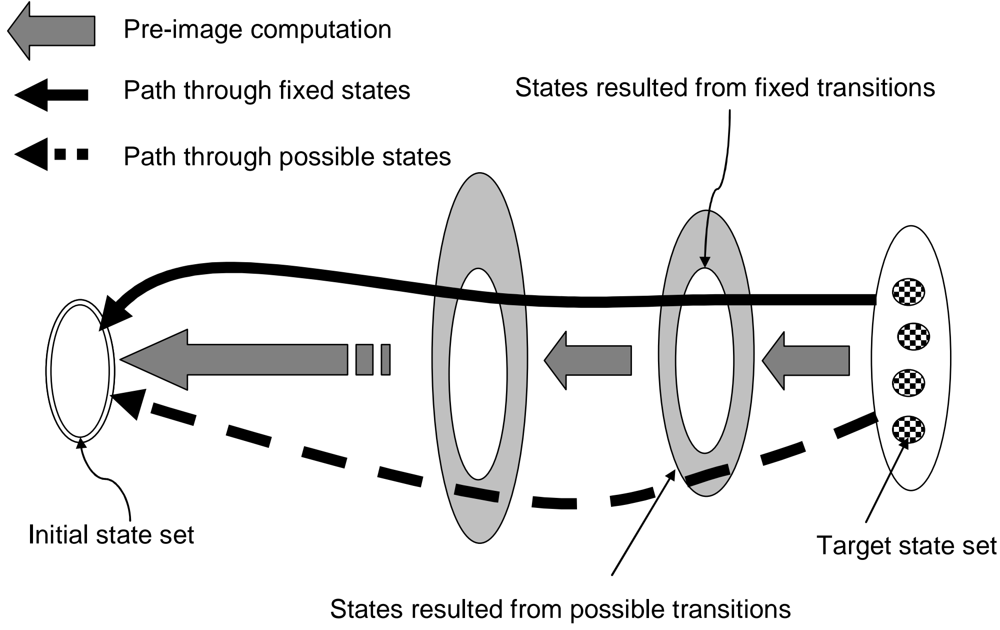

Figure 3.4: Backward symbolic simulation for Black Box verification.

3.1.4.2 An Improved Construction of $Sat_A(\varphi)$

The computation of $Sat_A(\varphi)$ can be improved, rendering the fixed transitions unnecessary. For example, the formula $Sat_A(EX\varphi)$ essentially says: *A state surely satisfies $EX\varphi$, "if there is an input value for this state and a fixed transition marked with this input value to a successor state that surely satisfies φ"*. On the other hand one can also see that *a state surely satisfies $EX\varphi$, "if there is an input value for the current state so that for all possible transitions marked with this input value, the successor state surely satisfies φ"*. This leads to the following improved formula for $Sat_A(EX\varphi)$. It was introduced by Nopper et al. in [84].

$$Sat_A(EX\varphi) = \exists\vec{x}\forall\vec{q}'\Big(T_E(\vec{q},\vec{x},\vec{q}') \rightarrow (Sat_A(\varphi)|_{\vec{q}\leftarrow\vec{q}'})(\vec{q}')\Big)$$

which can be written as:

$$Sat_A(EX\varphi) = \exists\vec{x}\neg\bigg(\exists\vec{q}'\Big(T_E(\vec{q},\vec{x},\vec{q}') \wedge \neg((Sat_A(\varphi)|_{\vec{q}\leftarrow\vec{q}'})(\vec{q}'))\Big)\bigg)$$

3.1.4.3 Termination Condition

Fig. 3.5 shows the possible termination conditions for the Black Box symbolic verification. The first column in this table shows the possibility of having a path from the initial state set to the target state set through either *fixed* or *possible* transitions. If there exists no such path then the main computation loop of the algorithm is executed until a fixed-point condition is reached.

Init state → Bad state (through transition)	Exist a path	Result
Fixed	Yes	Unrealizable
Fixed	No	Not proven
Possible	Yes	Not proven
Possible	No	Valid

Figure 3.5: Possible termination conditions.

3.1.5 Guided Search

Symbolic guided search has been proposed as a way to reduce the time and memory requirements of BDD-based invariant checking in [85, 86] and LTL model checking [87]. The authors used hints, which are assertions on the primary inputs and state variables of the model, to guide the exploration of the state space. The good hints can often be found with the help of simple heuristics by the designer. The extension of symbolic guided search to CTL model checking was introduced by Bloem et al. in [88].

The work in [89] presents guided heuristics for finding error states. One among them is *target enlargement*, where the error states are enlarged so they can be found with less searching. The second technique is to use *hamming distance* as the search metric i.e., those states that have the lowest hamming distance to the largest target enlargement are explored first. The third technique, called *tracks* uses *approximate* preimages that are based on a subset of the state variables to find the violations to assertions. The last technique uses explicit hints, called *Guideposts*, a series of conditions given by the designer to help direct the search.

3.1.6 Multithreaded Reachability

In [90] Sahoo et al. presented a multithreaded reachability algorithm. In standard POBDD-based reachability analysis, the complexity of BDD-based image computation can vary significantly between different partitions. In order to find an optimal schedule, the relative order in which the partitions are analyzed, the authors presented a solution based on a *multi-threaded* reachability approach. In simple words the method adds a parallel flavor to POBDD based reachability algorithm using multiple threads on a single machine. They introduced two techniques *early communication* and *partial communication*. In case of an *early communication*, after a partition finishes its local LFP (Least Fixed Point) then it will immediately communicate its states to other partitions. Whereas in *partial communication*, hard partition communicates using a small subset of its state space to the ideal node. This introduces new states in the easy partitions.

3.1.7 Partitioning in SymC

Sequential *SymC* [68] partitions the frontier state set into smaller subsets when the BDD reaches a certain threshold size. Then, it explores these subsets sequentially. Once it reaches a target state, *SymC* can save time and space by skipping exploration of the rest of the partitions. *SymC* uses many of the state-of-the-art partitioning technologies and in addition it has included some intelligent heuristics that aim at *fast falsification* and smooth traversal.

3.2 Distributed Verification

While sequential based model checking reduce the state space to be simultaneously treated by orders of magnitude, typical verification tasks still take modern sequential computers to their memory limits. One direction to enhance the applicability of today's model checkers is to use the accumulated memory and computation power of parallel computers. The use of distributed computing to increase the speedup and capacity has recently begun to generate interest [91, 92, 93, 94, 95, 96, 97, 98]. The algorithms are classified into several categories. Explicit state representation based [91, 92, 93, 94, 96], symbolic state representation based (BDD-based) [97, 98, 99], and SAT based [100, 101, 102].

3.2.1 Distributed Explicit Model Checking

Stern et al. [91] presented a parallel version of the explicit model checker Murφ. The state table that stores all reachable states, is partitioned very evenly over the network nodes. Each node maintains a work queue of unexplored states. If one of the nodes generates a new state, the $owner$ node for this state is calculated with a hash function and the state is sent to this node. If the node receives a state then it first checks whether it has been already visited. If the state is new, it is inserted in the state table and the local work queue. This process is repeated until a termination condition is detected. [93] reports a distributed version of SPIN for checking safety properties. The approach is very similar to [91] but the authors used different ways to partition the state space. The partition function tries to minimize the cross-transitions (transitions between states belonging to different processes) and eventually the communication by using the structure of the global system states. In [103] the author presented techniques to perform parallel state space construction. The author considered two types of processors nodes: $generators$ which compute the transition function, and $tabulators$ for state storage and search. The state set is partitioned between the tabulators using a hash function. In [104] Palmer et al. have shown a technique to combine partial order reduction and parallel distributed model-checking by picking best partial order reduction algorithm. In [92] Braberman et al. presented a distributed timed model checker tool ZEUS. The process of automatically verifying properties over real-time systems is known as *timed model checking*. ZEUS has been extracted from

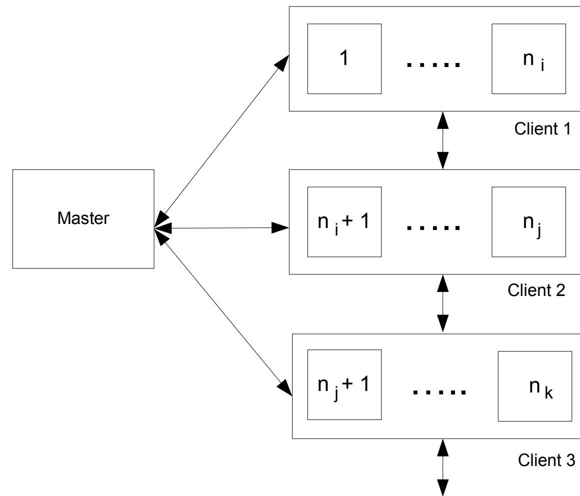

Figure 3.6: Unrolled circuit partitions.

the real-time model checker KRONOS [105]. It handles backward computation of TCTL(Timed CTL)-reachability properties over timed-automata.

3.2.2 Distributed SAT and SAT based BMC

Parallelization of SAT solvers is not new and it has been proposed by many researchers [101, 102, 106]. The key idea is that the search space is partitioned among different processors using partial assignments on the variables. Each processor communicates with other processors only after it is done searching its portion of the disjoint clauses.

In [107] the authors evenly distribute the partitioned disjoint clauses on processors. However, the variables appear in the clauses are not disjoint. Therefore, whenever a client finishes BCP (Boolean Constraint Propagation) on its set of clauses, it must broadcast the newly implied variables to all other processors.

The authors in [100] presented a method for distributed-SAT over a network of workstations using a Master/Client mechanism and a method for distributing the SAT based BMC using the distributed SAT. The key idea is unrolling the circuit in different time frames to provide a disjoint partitioning of the problem. Fig. 3.6 from [100] shows the partitioning of unrolled circuit. The topology has one master and several clients. Each client C_i hosts a part of the unrolled circuit and is responsible for performing BCP on its partitioned clauses.

3.2.3 Distributed Symbolic Reachability

The paper [97] presented a parallel algorithm for reachability analysis on a network of workstations (NOW) with disjoint memory that communicate via message passing. At the core they partition the state space on which the reachability

is performed in to k slices, where each slice is *owned* by one cluster machine. Each machine performs a Breadth First Search (BFS) algorithm on its owned slices. However, the BFS algorithm used by a machine can discover states that do not belong to the slice that it owns (called *non-owned* states) they are sent to the machines that owns them. In simple words, the *non-owned* states that do not belongs to slice/partition are called *cross over states*. As a result a process only requires memory for storing the reachable states it owns and computing the set of immediate successors for them. The load balancing is achieved by adjusting the *slices* if the initial balance is lost. Slicing is often inefficient because it partitions a relatively small BDD into many small slices. The more processes in the system, the less efficient the slicing is.

In [98] the authors presented an algorithm that tries to keep only as many network nodes busy as necessary by splitting and joining BDDs on demand. The algorithm works iteratively. It is initialized with one active node that runs a symbolic reachability algorithm, starting from the set of initial states. During its run, extra nodes are allocated and freed, as needed. At any iteration after an image computation the node sends those states it does not own to their owners and waits to receive the states that it owns from other network nodes. As a result, this algorithm works in synchronized iterations and it will result in unnecessary and sometimes lengthy idle time for *fast* processes. A synchronization phase is time-consuming when the number of processors is high. As a result processes underutilize the given computation power.

An effort based on a message passing infrastructure is undertaken in [99] to achieve asynchronous exchange of non-owned states. In the asynchronous approach, when a process completes an iteration it can continue in the image computation on the newly discovered states and receive owned states discovered by other processes at a later time. In order to achieve this the authors used a *distributed forwarding mechanism* that avoids synchronization and assures that states will eventually reach their owners.

The work [108] proposes a hybrid algorithm for slicing the state space and dynamically distribute the work among the worker processes on top of a large-scale distributed environment. The work also proposes a checkpoint/restart as part of the distributed reachability computation. The checkpoint/restart mechanism is needed in order to recover from a single computer failure and in order to better utilize clusters of computers when memory requirements vary significantly during computation. The checkpoint/restart mechanism is better suitable for non-dedicated networks.

[109] proposes a distributed symbolic algorithm for model checking of propositional μ calculus formulas. The algorithm distributively assesses subformulas. It results in sets of states which are evenly distributed among the processes. The algorithm uses a memory balancing procedure that ensures each set is partitioned evenly among processes. In the distributed phase, each process owns one part of the state space for every set of states associated with a certain subformula. When a computation of a subformula produces states owned by other processes, these

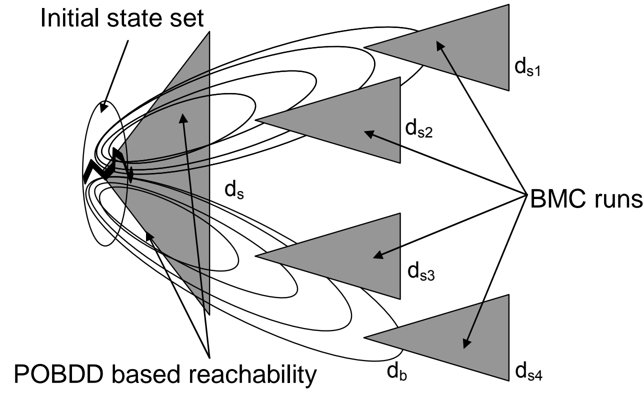

Figure 3.7: Combination of POBDD based reachability and SAT-BMC.

states are sent to the respective processes.

3.2.4 Grid-based Bounded Model Checking

Parallel bug hunting efforts were taken in [110] by applying the BDD-based massive underapproximation on a hybrid approach using both BDDs and SAT-BMC for error detection on computer grids. The method first finds deep reachable states, i.e., those states that can be reached using a more number of simulation cycles. Later these states are used as *seeds* for running SAT-BMC in *parallel* to explore the state space adjacent to such seeds. Fig. 3.7 [110] depicts the grid based BMC using two partitions and four instances of SAT. The triangles represent a search using SAT, and the ellipses denote successive image computations using BDDs. In their proposed approach, BDDs can go to a depth d_b, and many instances of SAT are seeded until then, which may be effective to differing depths $d_{s1}, d_{s1}, .., d_{sn}$. Consequently, this can reach deep error states.

3.3 Partitioning Heuristics

Many of the symbolic algorithms perform state space traversal until a blow up in BDD size is detected. Blow up can be detected by measuring the size of the symbolic representation of the frontier state set. After a blow up is detected, state set splitting based on the BDD variables is performed. The goal of partitioning is to create small and relatively balanced partitions. They should be disjoint in order to avoid the duplication of work. Since state sets are represented as Boolean functions our partitioning is based on Boolean function slicing as aforementioned in section 2.1.4, i.e., splitting a Boolean function represented as BDD into two parts depending on a variable. The variable is chosen to balance the sizes of the two resulting functions and to keep them small. The following discourse will explain the state-of-the-art heuristics for selecting a variable.

3.3.1 Approximation and Decomposition

The underapproximation algorithms based on a BDD data structure are presented in [111]. They are *Heavy branch subsetting* and *short path subsetting*. An underapproximation based on a BDD is the process of deriving a smaller BDD (in terms of its size) from a given BDD. One way to measure the rate of approximations is by their density. The density factor can be computed through following equation:

$$density = \frac{|f|}{\|f\|}$$

High density corresponds to a concise representation, i.e., a relatively smaller BDD can represent more number of states. This high density of interest for the symbolic verification as smaller the BDD size faster the verification process. Moreover, the high density BDD is not only small but represent majority of the state space.

Heavy branch subsetting: Determines how many states are in the function rooted at each internal node and builds a subset by throwing away one of the children of each node, starting from the root, until the result reaches a given threshold. The child that is eliminated from the result is the one that contributes fewer states.

Short path subsetting: It is based on idea that short paths, a path from root node to the terminal node, in a BDD give many states and contribute few nodes. The algorithm computes the short paths through each node and extracts the dense subset by removing the nodes with no short paths through them.

These two algorithms work better for sequential verification, i.e., whenever the size of the frontier state set reaches the threshold limit then one of these algorithms can be applied to extract the dense state subset. The traversal is performed on this dense subset by keeping the remaining state set on a stack. These algorithms are of minor interest in a distributed approach as they result in unbalanced cluster nodes in terms of memory and computation power consumption.

The other algorithms are *variable disjunctive decomposition* and *generative disjunctive decomposition* from [58]. These algorithms consume more time for decomposition. They try to estimate all the variables' positive and negative co-factors and take the best one, the variable that will balance the sizes of the two disjuncts and keep them small.

3.3.2 Grumberg et al. Partitioning Heuristic

The algorithm [97] takes reduction and redundancy factors into account for giving a better decomposition of a BDD. The cost function for partitioning is defined as:

$$Cost(f, x_i, \alpha) = \underbrace{\alpha * \frac{MAX(|f_{x_i}|, |f_{\bar{x}_i}|)}{|f|}}_{Reductionfactor} + \underbrace{(1-\alpha) * \frac{|f_{x_i}| + |f_{\bar{x}_i}|}{|f|}}_{Redundancyfactor}$$

The *reduction* factor gives an approximate measure to the reduction achieved by the partition. The *redundancy* factor gives an approximate amount of sharing BDD nodes between f_{x_i} and $f_{\bar{x}_i}$.

The cost function rely on the selection of the value of α: $0 \leq \alpha \leq 1$. An $\alpha = 0$ ignores the *reduction* factor, while $\alpha = 1$ ignores the *redundancy* factor. Based on this cost function the algorithm select a good splitting variable and partition the state into subsets using *Shannon expansion* principle. Due to the extra computations of *redundancy* and *redundant* factors the algorithm consumes a notable amount of time.

3.3.3 Eager Decomposition

Fig. 3.8 delineates the pseudo code for the *eager decomposition* [68] algorithm. Compared to the above algorithms, the splitting algorithm tries to select a variable for which its positive and negative co-factors are well balanced according to a balancing condition (line 8) . If it cannot find such a variable, it picks the variable resulting in the least difference in the sizes of its positive and negative co-factors (lines 11-15). Since the algorithm eagerly checks for the variable that satisfies the balancing condition, the algorithm is known as *eager decomposition*. That is whenever we find an appropriate variable, the algorithm skips the exploration of the remaining variables.

3.3.4 Minimal Overlap

The partitioning heuristic *minimal overlap* [112] aims at minimizing the state overlap between the partition traversals. After a *blowup* is detected during the state space traversal the state set is partitioned to subsets and these subsets will be traversed in a divide and conquer manner. The algorithm may reduce the effort spent on the network nodes, as redundant computations are avoided.

The algorithm relies on the fact that the transition relation is conjunctively partitioned. All these partitions are statically analyzed to find dependencies between the state variables. First, the algorithm determines the number of present state variables that influence the truth value of a next state variable. Next, it orders the state variables according to this dependency count. The variables with the highest dependency count are selected for splitting. The crucial point behind the *minimal overlap* heuristic is that if splitting is done on one of the selected variables, then image computations in these partitions are less likely to produce the

```
                                                                        1
// S is the state set that need to decomposed based on a               2
variable
selectVariable(in: S; out: bestVar)                                     3
  bestDiff = |S|; // Initialize with number of nodes of S              4
  for all x_i  // x_i ∈ S.support()                                    5
    balance = |S_{x_i}| / |S_{x̄_i}|;                                    6
    // balancing condition                                              7
    if ( 0.75 ≤ balance ≤ 1.25 )                                        8
      bestVar = x_i;                                                    9
      break;                                                            10
    // check if variable is better than current one                     11
    diff = abs(|S_{x_i}| − |S_{x̄_i}|);                                  12
    if ( diff < bestDiff )                                              13
      bestDiff = diff;                                                  14
      bestVar = x_i;                                                    15
```

Figure 3.8: Variable selection for *eager decomposition*.

same truth values in the dependent next state variables.

Often, the selected variable cannot partition the state space into balanced subsets. In order to overcome this problem, the algorithm uses a cost function from [97] with the set of selected variables. It returns one variable for achieving a reasonably balanced partitioning.

Of course, in the worst case the minimal overlap condition holds only for one or few traversal steps, as many other conditions can change the values of state variables. For example, the variable with the maximal dependency count can depend on an input variable disjunctively. The selected set of variables is further scrutinized to avoid such trivial situations. Since this algorithm is used by the thesis with minor adjustments for parallel environments, it will be explained in detail in chapter 5.

Similar efforts are undertaken for model checkers with an explicit state graph representation [113]. They apply graph algorithms that heuristically try to find partitions with few crossover transitions in order to reduce the communication effort between processes.

3.3.5 Guiding Heuristics

Some guiding heuristics that aim at *fast falsification* were presented in [114]. Guiding is based on the property that is supposed to be verified. There are two different ways of guiding:

- State variable guiding : The actual state set is partitioned into two parts using one of the influencing state variables.

- Input variable guiding : The actual state set is restricted with the set of input variables (hints) such that the transitions satisfying those set of variables are allowed and the others are discarded.

The algorithm treats the properties of the kind "if A then C", i.e., $A \rightarrow C$, where A and C are LTL/FLTL expressions. For guiding, only the C part of the property is of interest. Because the tool *SymC* is highly optimized for the traversal, the traces that are not satisfying the A part of the property will be removed from the traversal and in some cases the assumption part can also be empty, i.e., *True*. Hence, the guiding algorithm collects all the signals of the C part.

The algorithm takes the property and identifies the set of output signals and other state variables that are involved in the property, which is the set of interesting signals. All the interesting output signals that are collected are then reduced to sets of state variables. This interesting set of variables are the ones that jointly define the property to be valid or invalid. Hence, the algorithm decomposes the validation of one property to the validation of a set of variables. This set of variables are the target variables that have to be restricted in order to guide the traversal. This set of new influencing variables can be either the state or input variables.

3.4 Un-addressed Problems

3.4.1 State-of-the-art Sequential Problems

As aforementioned in section 2.7, the partition approach is good once any of the partitions finds an error state and it is therefore not necessary to traverse all other partitions. Thus, time and memory can be saved. However, if our design is error free or the target state can only be reached using the final partition then we have to explore all partitions sequentially. Fig. 3.9 delineates the partitioned based sequential verification. The state space is partitioned into four disjoint parts once the threshold for the size of the BDD representing the frontier state set is reached. The sequential approach performs the traversal on each partition i (i = 1,2,3,4) in a divide-and-conquer manner. The figure shows that the target states can only be reached using the final partition.

Let n be the number of partitions, t_i the time it takes for partition i to be fully traversed, and $max(t_i)$ the maximum time taken by a partition for exploration of its state space. Then $\sum_{i=1}^{n} t_i$ is the time taken for full exploration of the system's state space.

Let b_j be the time it takes to reach a target state in partition j. Suppose the partitions are scheduled such that we are reaching the target state in partition j, then $b_j + \sum_{i=1}^{j-1} t_i$ is the time taken by the sequential algorithm to reach the target state.

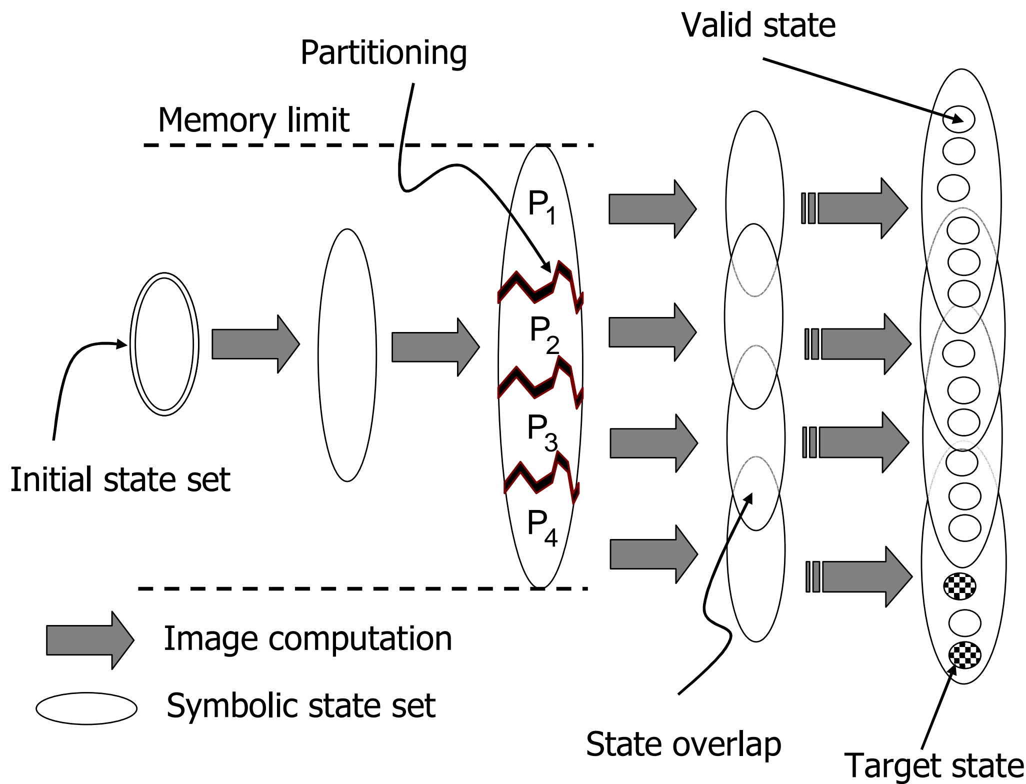

Figure 3.9: Partitioning problems.

Let $R = \{P_1, \ldots, P_k\}$ be the set of partitions from which one can reach the target state and *min*(P_i) the minimum time taken by a partition to reach the target state. When all partitions are executed in parallel, the time taken for exploration of the whole state space is *max*(t_i) and the fastest reachability of the target state is achieved in *min*(P_i). In the sequential approach it may happen that partitions $P_i \notin R$ are explored unnecessarily as they cannot reach the target state. Even if we reach the target state, it may not be the partition which takes the least time to reach the target state that is explored first.

In addition, the partitioned sequential approach suffers with the state overlap that is shown in Fig. 3.9. Albeit the algorithm [112] tries to reduce the overlap up to certain number of traversal steps, for some designs the overlap is so high that it can exacerbate the advantage of using the partitioning methodology. Furthermore, in order to remove overlap using the *dynamic overlap reduction* method, the whole visited state space should be maintained, which again has an adverse effect on the bounded property checking approach used by the tool *SymC*. In contrast to classical model checkers, *SymC* does not maintain any visited state space history but examines all reachable state space up to a given time bound.

In comparison to all the sequential approaches [115, 116, 112, 114], the parallel approach has several benefits. The idea of the distributive approach is to partition the state space upon reaching a threshold limit and assign the traversal of the subsets to network nodes. The approach enables the verification of larger models

than those capable with the regular nonparallel version. The sequential version fails for these designs because it often encounters state space explosion early on in the computation, after which it could not make much progress due to memory limitations. However, the reduced memory requirements for the cluster nodes in the distributed version still allow progress in the traversal process. Therefore, it is able to finish large circuits. The parallel approach can exploit any network size and its utilization of network resources make it suitable for solving very large verification problems. Also, because the distributed approach is not sensitive to the traversal scheduling order of partitions, the termination condition can be found as quickly as possible.

The sequential methodologies in [81, 82] can be superseded by parallelizing their core algorithms. The authors in [90] themselves motivated towards parallelization.

3.4.2 State-of-the-art Parallel Problems

In principle, two types of parallel techniques are widely used: 1. *windowing* based; 2. *partitioning* based. Below are the enumerated problems encountered by both the state-of-the-art algorithms:

3.4.2.1 Windowing based Distributed Approaches Problems

Many of the previous windowing based distributed schemes are synchronous [91, 97, 98, 117]. They consist of interleaved rounds of computation and communication. Conceptually, the system state space on which the reachability is performed is divided into a pre-defined number of subsets. Each of these subsets is assigned to a network node (called as *window*) and each window has a restriction that has to be obeyed in all future time steps, i.e., the state space is restricted at every time step. After obtaining the state subset, each node applies image computation algorithm on its window state sets. However, after image computation the node can discover states that do not belong to the slice that it owns (*non-owned* states or *cross over states*). In such cases, the node should wait for others to complete their current iterations.

Fig. 3.10 depicts the conceptual method used by the state-of-the-art distributed approaches. After each iteration, the lightly loaded machine must wait for the slowest one at the end of each computation step. This will create unnecessary and lengthy idle times for fast processes. Next, synchronization phase is time-consuming, especially when the number of processes is high. In addition, the communication overhead can grow when more number of processes involved in the parallel environment. Further, a synchronous load balancing is performed at the end of each iteration, i.e., all the network nodes need to be stopped in achieving load balancing. This leads to the underutilization of computation power, since available free processes are not used until there is not other choice but to join once the slow processes finish their computations for achieving load balanc-

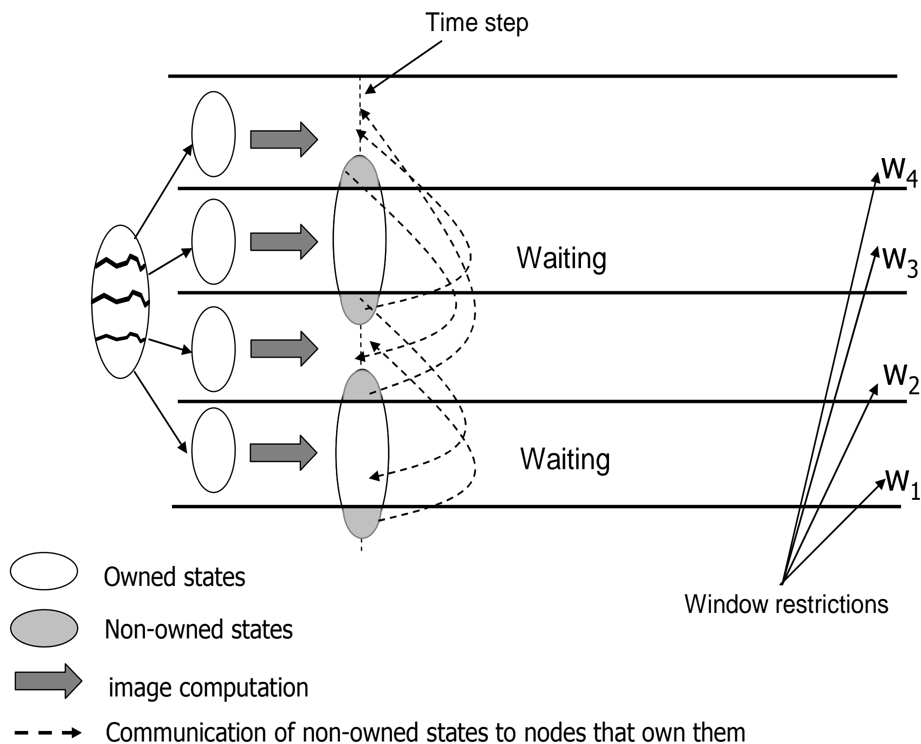

Figure 3.10: The windowing based distributed method used by the state-of-the-art approaches.

ing. These drawbacks limit the scalability of the parallel algorithms and make them slow down substantially. To be specific, the windowing based distributed approaches frequently face the following problems:

1. The communication of cross over states or non-owned states at every step to other *windows* is an expensive operation and requires the other nodes to wait.

2. The *windows* could get often empty images i.e., all the successor states of that particular *windows* are cross over states. This eventually leads to improper load balancing among network nodes (*windows*). In such cases, the *windows* should be redefined. The redefinition of *windows* cause the whole system should be synchronized.

Hence, these 1 and 2 factors make the *windowing* technique synchronous and reduces the potential speedup because processors are kept waiting for others to complete. However, asynchronous efforts were taken in [99] but it is not natural the authors used message passing infrastructure for asynchronous exchange of non-owned states.

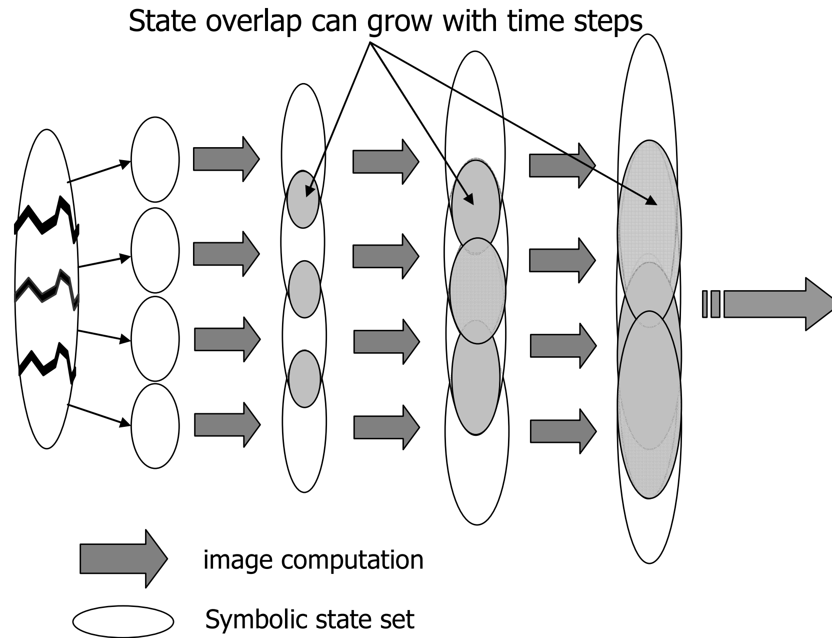

Figure 3.11: Partitioning based distributed approach.

3.4.2.2 Partitioning based Distributed Approach Problems

This type of distributive approach [118] partitions the whole set of states into several subsets. Partitioning is based on the variable restriction as per the *Shannon expansion* explained in Chapter 2. In contrast to *windowing* approach the state set is restriction is performed only at the time of partitioning (which avoids later the synchronization problem). Fig. 3.11 visualizes the partitioning based distributed approach. Each node independently performs BFS on its state subset. However, after a few steps of traversal among partitions a state overlap may emerge among network nodes. The state-of-the-art partitioning heuristic [44] reduces the overlap only up to certain time steps by static means. However, in general the overlap may still grow after a few steps of traversal and that need to be removed. Otherwise it would worsen the advantage of using the parallel approach.

3.4.2.3 Other Common Problems

The parallel algorithms engage only one node to perform state space partitioning, which is an expensive operation as far as time is concerned. The partitioning heuristics used by the state-of-the-art parallel algorithms take only balancing conditions and the sizes of the resulting partitions [97, 98, 117, 99, 110] into account but not subsequent state overlap or cross over transitions that would appear during traversals on each partition. The algorithms concentrate heavily either on reachability (validation) [97, 98, 117, 99] or falsification but not both together. The most of the algorithms are suitable for homogeneous system configurations where it is often the cases that only a limited number of resources available. The

authors in [110] proposed a combined (POBDD reachability and SAT based BMC) underapproximation approach for fast falsification on a grid-framework. However, the approach provides no intelligence in selecting the seed states for BMC.

In general, existing schemes for parallelizing BDD-based verification algorithms often suffer from state overlap or duplicate work, cross over states among partitions, inefficient work distribution, improper load balancing, synchronicity and communication overhead. These state-of-the-art problems have to be solved in order to achieve speedups in a parallel environment.

3.5 Contributions

My main contributions in thesis addresses the above listed problems by new optimized algorithms and therefore outperform the state-of-the-art algorithms. The algorithms were developed in an incremental manner aiming at efficient asynchronocity to gain speedups. First the efforts were started with the basic parallelization. Though the algorithm is fairly simple to construct but it has problem with the state overlap. However, the state set decomposition in this algorithm can be done in parallel.

After having seen the state overlap problem, the *dynamic overlap reduction* technique is developed. The algorithm smoothens the state space traversal of each partition by removing the overlap that it suffers from. Albeit, the algorithm supersedes the basic algorithm it suits best only for validation. Since, fast falsification is of high interest to the industry, the hybrid methodology is developed, which suits best for both fast falsification and validation. The *hybrid* algorithm efficiently combines both *windowing* and *dynamic overlap reduction* techniques to obtain more synergy and gains the advantages of using both the approaches.

Both *dynamic overlap reduction* and *hybrid* methods use the high communication mechanism hence these algorithms are best suitable for homogeneous cluster environments where high-speed inter connections are possible. This thesis also presents a novel asynchronous grid based distributed algorithm based on a effective combination of state-of-the-art intelligent underapproximation heuristics that suits best for fast falsification. The main asynchronous parallel verification algorithms that are contributed by this thesis are listed below:

Parallel partitioning: The algorithm partitions the state set in parallel by distributing the partitioning effort on all nodes equally. The nodes on the cluster environment are restricted to n nodes, where $n = 2^k$ and $k \in \{1, 2, 3, \dots\}$. Depending on the identity, each node partitions the state set and obtains its disjoint subset. Thus, the algorithm reduces both partitioning time and effort. The state set partitioning using only one node consumes notable amount of time, i.e., for n partitions we need *n-1* splits but whereas if we distribute the partitioning effort on all nodes equally then we require log_2^n splits. This algorithm alleviate the state space partitioning problem (in terms

of effort and time) described in section 3.4.2.3. The parallel partitioning algorithm is detailed in Chapter 4.

Dynamic overlap reduction: Conceptually, after the state set distribution, the method uses some extra resources like a *Coordinator* to periodically remove the overlap in an asynchronous manner, i.e., with out waiting for other processors to complete their image computations. In addition, this method has the natural side effect of dynamic load balancing among network nodes. The nodes that are heavily loaded with state sets at one time point will later be alleviated by assigning few number of states. Since all the nodes perform asynchronous state space traversal on their whole state subsets, the method is best suitable for validation. In comparison with state-of-the-art parallel schemes (described in section 3.4.2.1) the *dynamic overlap reduction* algorithm is an asynchronous algorithm and it has the natural side effect of load balancing. Therefore, it alleviates the 1 and 2 state-of-the-art problems described in 3.4.2.1. In comparison with state-of-the-art static overlap reduction (described in section 3.4.2.2), which can reduce overlap only up to a certain time steps, the *dynamic overlap reduction* obliterates the state overlap. The distributed algorithm based on dynamic overlap reduction is expounded in Chapter 5.

Hybrid method: Conceptually, the nodes of the distributed environment are categorized into *windows* and *helpers*. Windows aim at *fast falsification* by restricting their state spaces at regular intervals, whereas *helpers* are responsible for cross over state space traversals, i.e., the state space left over by the *windows*. Thus, in turn leads to *fast validation*. The exchange of cross over states to helper nodes is done in an asynchronous manner. If any of the *windows* get empty images during traversal then it dynamically becomes *helper* and shares work with other helpers. Since the approach also uses *dynamic overlap reduction* the overlap among *helper* nodes can be removed without waiting for other nodes to finish their computation steps. Further, the algorithm expedites the verification process by reassign the work to idle nodes as quickly as possible (without disturbing the other node's computations), thereby it avoids the wasted computation power and makes the system work efficient. In comparison with state-of-the-art parallel strategies (described in sections 3.4.2.1 and 3.4.2.2), the *hybrid* algorithm is an asynchronous distributed algorithm suited for both fast falsification and full validation. The algorithm asynchronously distributes (without stopping other node's computations) the cross over states to helper nodes. Further, the algorithm gains the advantages of using both *windowing* and *dynamic overlap reduction* techniques. This algorithm is detailed in Chapter 6.

Grid-based algorithm: The approach effectively combines the underlying verification algorithm, bounded property checking with the state-of-the-art intelligent guiding heuristics based on property and adapted for grid environment to handle the very large hardware designs. The approach is best suits for fast falsification. In comparison with state-of-the-art grid based verifi-

cation scheme (described in section 3.4.2.3), the grid algorithm described in this thesis effectively combines the intelligent underapproximation, ingenious guiding and static overlap reduction algorithms. Further, the algorithm uses very little communication, i.e., only at the time of global violation. This algorithm is detailed in Chapter 7.

These algorithms are evaluated on publicly available benchmarks from the ISCAS89 suite, IBM benchmark suite and our in-house Holonic model. The algorithms show approximately linear speedups in execution time and enables faster verification of properties for very large hardware designs. A significant improvement in results were obtained when compared with state-of-the-art parallel verification schemes.

As an application, this thesis also presents a novel distributed algorithm for Black Box verification, which uses mixed forward and backward traversal mechanism (detailed in Chapter 8).

Chapter 4

Parallelization

The first step towards getting somewhere is to decide that you are not going to stay where you are.
- John Pierpont Morgan

This chapter discusses the basic parallelization algorithm. The basic algorithm is fairly simple to construct, but it has a side effect on the *state set overlap*. This chapter delineates each individual network node's algorithm. It covers how to compute counterexamples in a parallel environment if any of the nodes uncover the target state set. It also gives a formal definition of the notorious *state set overlap*. The prominent optimizations to treat the *state set overlap* are described in later chapters.

4.1 Parallelization of a Symbolic Bounded Property Checking

The basic parallelization method parallelizes the symbolic state space traversal on a network of processors that communicate via the message passing paradigm. First, the method starts traversal on set of initial states and iteratively computes the frontier set until its size reaches a given threshold limit. Then the frontier state set is partitioned into subsets, where each subset is assigned to one node of the cluster computer. As soon as a node has its state subset, it proceeds to compute forward state space traversal in iterative BFS (Breadth First Search) steps. In general, computation on a smaller subset requires less memory compared to the whole set. This method enables us to find errors that are far from the initial states.

The distributed approach consists of the initial sequential stage and a subsequent parallel stage. Fig. 4.1 illustrates this process. The nodes of the cluster computer are categorized into one master and the remaining slave nodes. For better memory balancing the basic method restricts to use n nodes, where $n = 2^k$ and $k \in \{1, 2, 3, \dots\}$. Each node in the parallel environment is identified by a unique rank ($rank \in \{0, \dots, 2^k - 1\}$). Typically, the rank system is taken care of

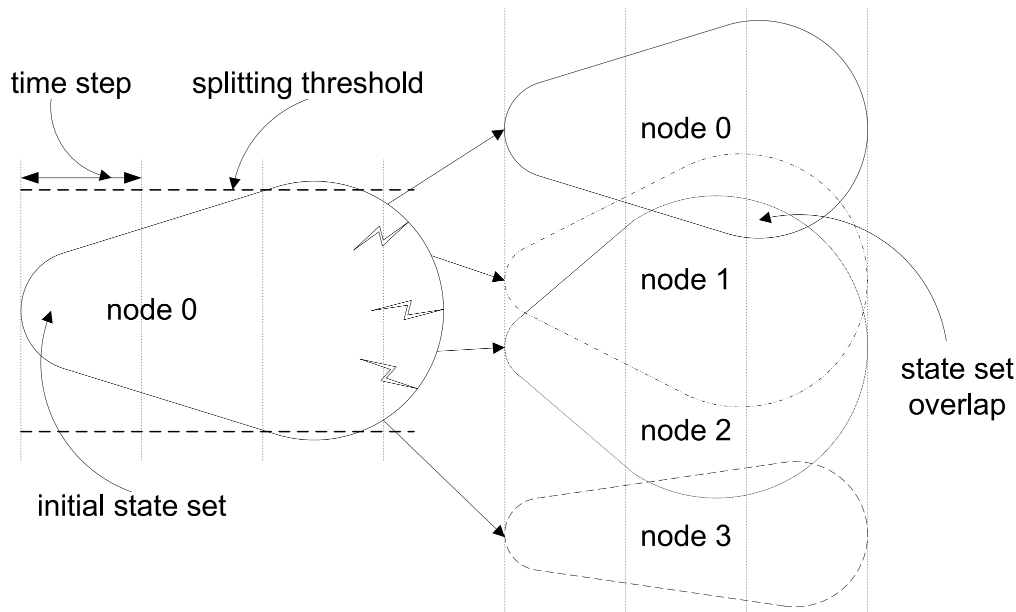

Figure 4.1: Outline of the basic distributed algorithm. After reaching the splitting threshold, the state set is partitioned into subsets and these are distributed on computation nodes for independent traversal.

by the message passing mechanism.

First, all the nodes, i.e., both master and slave nodes, parse the system description and the property specification and translate them into BDD form. These are basically the transition relation of the system and the properties' AR-automata. Parsing on every machine makes the method best suitable for homogeneous clusters. In case of heterogeneous parallel environment, a powerful master node alone can be assigned with the parsing and distribution tasks.

After successful parsing on each node, the master starts its symbolic state traversal algorithm, whereas the slaves will remain in waiting state after receiving the BDDs. The property checking algorithm for the master continues until the size of the frontier state set reaches the initial threshold limit. At this point, it broadcasts the frontier state set and indicates the nodes to split it. The main reason for doing this is to reduce the splitting time and distribute the splitting effort on all nodes in the network. In detail, the master node can split the frontier state set into n subsets. However, this process consumes a notable amount of time.

4.1.1 State Set Decomposition in Parallel

Splitting the state set in parallel yields better results. Fig. 4.2 illustrates this process. Depending on the rank each node splits the frontier state set and obtains its subset. Fig. 4.2 illustrates the initial state set distribution algorithm. It iteratively splits a state set S into two parts and drops one of the resulting sets. In the end it keeps the subset that belongs to the node identified by its rank.

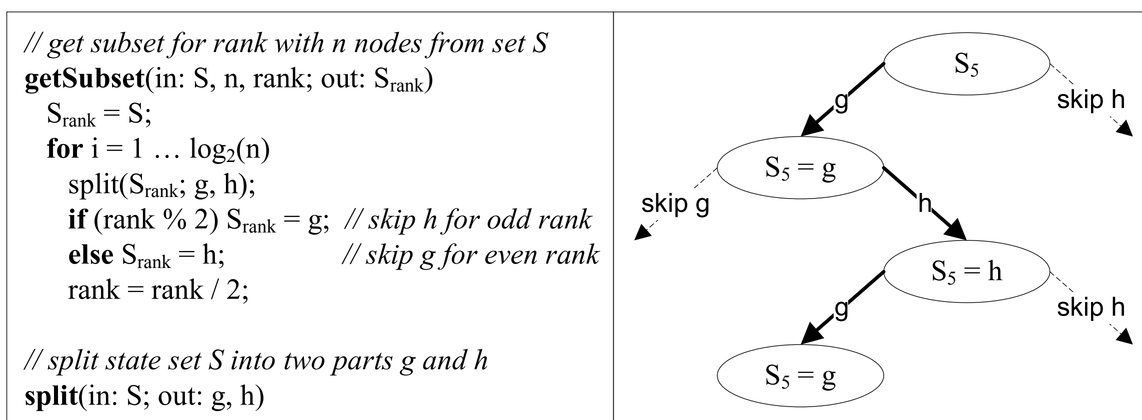

Figure 4.2: Algorithm for state set distribution. The left hand side contains the distribution algorithms. An example application is shown on the right hand side.

The left hand side of Fig. 4.2 shows the distribution algorithm *getSubset*. It returns the subset S_{rank} for node *rank* from set S for n possible nodes. It calls the algorithm *split* that partitions a set S into two disjunct subsets g and h. An example application of *getSubset*$(S, 8, 5, S_5)$ is shown on the right side of Fig. 4.2. The algorithm iterates $log_2(8) = 3$ times. First, it splits S into g and h and updates S_{rank} with g and skips h. Second, the algorithm splits S_{rank} and updates S_{rank} with h and skips g. Third, it splits S_{rank} and sets S_{rank} to g and skips h. The control flow of this example is indicated by the bold arrows in the figure.

The algorithm *getSubset* uses the procedure *split*, which partitions the given state set S into two subsets based on a splitting variable. The resulting subsets are relatively small, well balanced and disjoint. In order to meet these conditions, the split procedure takes both a balancing factor and a cost function into account. The detailed explanation of the split function is postponed to next chapter.

After the assignment of the state subsets to each node, all nodes will proceed with forward state space traversal. Whenever one of the nodes detects the violation condition, it initiates abortion of the other nodes. Fig. 4.3 and Fig. 4.4 delineate the master and slave node computation loops, respectively. The first three operations (lines 2-4) in both algorithms perform preprocessing and generate the transition relation of the system and the property. Since this thesis uses the *SymC* core verification algorithm as described in section 2.6, the property's transition relation is represented as an AR-automaton. The operation *nodesInCommWorld* (line 6) in both algorithms computes the number of nodes in a parallel environment using message passing interface. The pseudo code in lines 11-14 in Fig. 4.3 controls the sequential phase of the master node algorithm. The operation *getSubset* (line 13 in Fig. 4.3 and line 8 in Fig. 4.4) represents the same algorithm as depicted on the left hand side of Fig. 4.2. After the state set partitioning, all nodes first compute the image of the AR-automata and check for global violation and local termination conditions of the property (line 16-20 in Fig. 4.3 and lines 13-17 in Fig. 4.4). If any of these conditions do not meet, both algorithms compute the image of the system. The operation *checkAbortCondition* in both algorithms checks

if any of the other nodes detected a target or error state. The corresponding node contacts the master node to terminate the simulation. Optionally the master node computes the counterexample. The following discourse will expound the counterexample generation.

```
// Preprocessing
parseModel();
generateSystemTrans();
generateARTrans();
symbolicSimulate()
   N = nodesInCommWorld();
   S = Sys.start ∧ AR-aut.start;
   splitFlag = true;
   while i ≤ k  // k is the time bound
     checkAbortCondition();
     if ( (|S| ≥ threshold) ∧ (splitFlag) )
       distributeFrontierStateSet();
       S = getSubset(S, N, rank);
       splitFlag = false;
     // Compute image of AR-Automata
      S = image_AR(S);
      if( propertyViolation() == true )
        reportFailure();
        abortSlaveNodes();
      checkLocalTerminationCondition();
      //Compute image of the system
      S = image_T(S);
      i++;
```

Figure 4.3: Master node main computation loop.

4.1.2 Counterexample Computation

If any of the slave nodes uncovers one of the target states during the exploration then it immediately broadcasts an asynchronous message to the master node. Depending on the option to generate a counterexample, the node that detected the target state will send all its frontier state space, i.e., the covered state space during the parallel phase of its computation to the master node. In general the node that detected the target state set can compute the counterexample. But sending this information to master node avoids the multiple computations of counterexamples if more than one node detect the target state set relatively at the same simulation cycle. In other words, the master node has better control over the system and also has the sequential state space. Therefore, it is alone compute the counterexample. Upon receiving this message, the master node first terminates all other nodes computations by broadcasting an asynchronous message. Second,

```
// Preprocessing                                                    1
parseModel();                                                       2
generateSystemTrans();                                              3
generateARTrans();                                                  4
symbolicSimulate()                                                  5
   N = nodesCommWorld();                                            6
   waitForCurrentStateSet();                                        7
   S = getSubset(S, N, rank);                                       8
   // n is the time point of the initial partitioning               9
   while n ≤ k  // k is the time bound                              10
     checkAbortCondition();                                         11
     // Compute image of AR-Automata                                12
     S = image_AR(S);                                               13
     if( propertyViolation() == true )                              14
       reportFailure();                                             15
       abortOtherNodes();                                           16
     checkTerminationConditionLocally();                            17
     //Compute image of the system                                  18
     S = image_T(S);                                                19
     n++;                                                           20
```

Figure 4.4: Slave node main computation loop.

it will add the frontier state space covered by the node that detected the target state to its sequential frontier state space. Finally, the master node executes the pre-image computations on property's and system's transition relations in order to compute the counterexample. Fig. 4.5 details the master node's counterexample generation algorithm. The size of S_{seq} (line 7) represent the length of the counterexample. The operation *store* (lines 9 and 16) constructs the path from the target state to initial state by executing the standard counterexample generation algorithm [6].

4.1.3 State Set Overlap

This simple basic parallelization scheme fails to provide significant speedups on many models because of crossover transitions. These transitions start in a state of the current subset but lead to a state that is already present in one of the other state subsets. This phenomenon is called state set overlap, or just overlap. Of course, image computation for overlapping states is performed redundantly. As image computation is one of the key components of any formal verification tool, redundancy of such a component badly affects the time and memory requirements of the whole verification process. State overlap between network nodes is salient in Fig. 4.1.

Definition 15 *Let S be a nonempty set and $S_1, \ldots, S_k \subseteq S$ with $k \geq 2$. Then we define*

```
// St_seq is the vector of sequential frontier state sets                1
// T is the target state                                                  2
generateCounterExample(in: St_seq, T)                                     3
  St_par = receiveTheParallelFrontierStateSets();                         4
  terminateOtherNodes();                                                  5
  appendParallelFrontierStatesIntoSequential();                           6
  i = St_seq.size() - 1;                                                  7
  S = T ∧ St_seq[i];                                                      8
  S = selectRandomState(S); store(S);                                     9
  while i ≥ 0                                                             10
    i--;                                                                  11
    // Compute pre-image of AR-Automata                                   12
    S = pre − image_AR(S);                                                13
    // Compute pre-image of the system                                    14
    S = pre − image_T(S) ∧ St_seq[i] ;                                    15
    S = selectRandomState(S); store(S);                                   16
```

Figure 4.5: Master node counterexample computation algorithm.

the state overlap $o_k \in [0,1]$ *of these partitions as:*

$$o_k = \frac{\sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \|S_i \cap S_j\|}{\|S\| \sum_{i=1}^{k-1} i}. \tag{4.1}$$

The overlap is thus the normalized average of states in the pairwise intersection of subset permutations. The sum in the denominator ranges from 1 to $k-1$ because this yields the number of pairs S_i, S_j with $i < j$. An overlap of $o_k = 0$ corresponds to disjoint partitions and an overlap of $o_k = 1$ corresponds to partitions containing the same states.

4.2 Conclusion

This chapter presented a parallel version of the bounded property checking tool. The idea of the approach is to partition the state space upon reaching a threshold limit and assign traversal of the subsets to network nodes. The parallel algorithm has several advantages. It enables the verification of larger models than those with the regular nonparallel version. Sequential version fails for these designs because it often encounters state space explosion early on in the computation, after which it could not make much progress due to memory limitations. However, the reduced memory requirements for the cluster nodes in parallel version still allow progress in the traversal process. Therefore, it is able to finish verification for large circuits. The parallel approach can exploit any network size and its utilization of network resources make it suitable for solving very large verification problems. Further, this chapter detailed a novel state decomposition algorithm [118], which partitions the state set in parallel. Thereby, the algorithm reduces

both partitioning time and effort. The parallel algorithms described in further chapters of this thesis utilizes the parallel state set decomposition algorithm.

Chapter 5

Treatments for State Set Overlap

Don't find fault, find a remedy.
- Henry Ford

This chapter presents several solutions to reduce the *state set overlap* during distributed symbolic verification. First, it details the static overlap reduction algorithm and points out that the algorithm can only reduce overlap up to few number of simulation cycles. Second, it explicates the new distributed bounded property checking algorithm based on the *dynamic overlap reduction* technique, which uses extra network resources. The algorithm becomes more feasible only when the shared states due to crossover transitions are reduced to avoid duplicate work. Third, it explains in detail the counterexample computation algorithm that is relevant to the new distributing approach. Fourth, it gives information on the impacts of overlap removal on resulting BDDs and also explains some of the state-of-the-art BDD minimization algorithms and their application to distributed checking algorithm. Finally, it explains the only snag (best suited for full validation but not for fast falsification) that could result by using the new distributed algorithm based on *dynamic overlap reduction* technique.

5.1 Overlap Reduction

Chapter 4 indicated that the *state set overlap* slows down the pace of symbolic traversal on every node. In the worst case, all nodes perform redundant computations, i.e., there is no difference between sequential and parallel verifications. Therefore, *state set overlap* must be treated in order to speedup the parallel verification and to achieve end results faster. Two approaches were developed to remove or reduce the overlap: *static overlap reduction* and *dynamic overlap reduction*. The former uses a partitioning algorithm that aims at minimizing the state overlap based on structural information of the design. Whereas the latter uses extra resources in order to remove the overlap. These methods will be explained in detail in the following subsections.

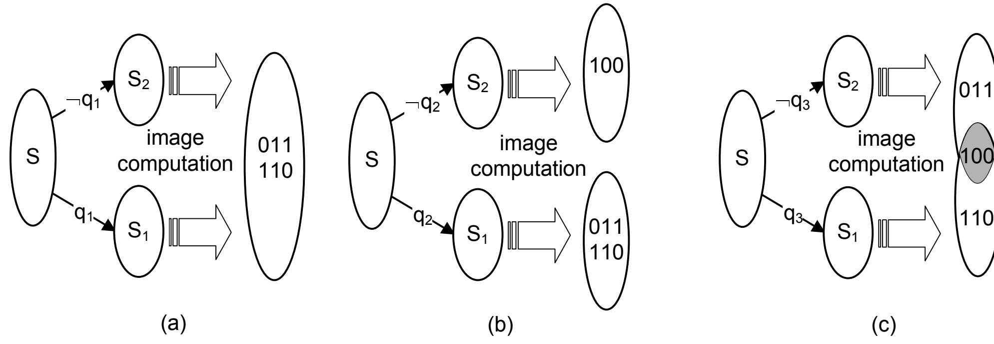

Figure 5.1: Possible overlap of subsets after an image computation with dependencies on the splitting variable.

5.1.1 Static Overlap Reduction

Overlap originates from states in different sets having transitions to the same next states. In order to minimize the overlap of splits, the selected splitting variable v should not allow states that have common next states to be in different splits. In other words, v should partition the states such that they have no common next states. However, in reality such a partitioning is not possible, but one can put some effort in selecting the splitting variable v to minimize overlap. For finding a good splitting variable the *Minimal overlap* algorithm [112] is used. The algorithm statically analyzes the design which is represented as a finite state machine (FSM).

The idea of selecting a good splitting variable v relies on the conjunctively partitioned transition relation T, as aforementioned in section 2.5.1. For every $i \in 1, \ldots, m$ a partition T_i of the transition relation corresponds to the truth value of next state variable q'_i such that: $T = T_i \wedge \ldots \wedge T_m$. The algorithm picks variable v from the set of state variables $E = \{q_1, \ldots, q_m\}$ based on the observation that only a small number variables in E is forming the next state function f_i, formally $support(f_i) \subset E$.

Let us refer to Fig. 5.1 in order to see how static information affects the overlap of sets of states. For example [112], assume that the set of states $S = \{001, 010, 011, 100, 110, 111\}$ that are encoded by the state variables $E = \{q_1, q_2, q_3\}$. The partitioned transition relation is given as,

$$\begin{aligned} T_1 &= (q'_1 \equiv f_1) where f_1 = \neg q_2 \vee q_3 \\ T_2 &= (q'_2 \equiv f_2) where f_2 = q_2 \\ T_3 &= (q'_3 \equiv f_3) where f_3 = q_2 \wedge \neg q_3 \end{aligned} \tag{5.1}$$

Let us see with this small example how the splitting variable could affect the overlap of the two sets. Assume the set S is splitted into two sets S_1 and S_2 by the variable q_1 as shown in Fig. 5.1 (a). The splitted set of states will be $S_1 = \{100, 110, 111\}$ and $S_2 = \{001, 010, 011\}$. Then the image of both the sets S_1 and S_2 by applying the partitioned transition relation (see equation (5.1)) will

be $\{011, 100, 110\}$, where there is 100% overlap. If the splitting variable is q_2 as shown in Fig. 5.1 (b). Then the splitted sets will be $S_1 = \{010, 011, 110, 111\}$ and $S_2 = \{001, 100\}$. Then the image of the set S_1 will be $\{011, 110\}$ where as the image of set S_2 will be $\{100\}$, therefore there is 0% overlap. Finally, if the splitting variable is q_3 as shown in Fig. 5.1 (c). Then the splitted sets will be $S_1 = \{001, 011, 111\}$ and $S_2 = \{010, 100, 110\}$. Then the image of the set S_1 will be $\{100, 110\}$ and the image of set S_2 will be $\{011, 100\}$, where there is a partial overlap. The interesting point is, given a set of states and the partitioned transition relation, the overlap of the image of the splitted sets can vary depending on the splitting variable.

The *Minimal overlap* algorithm heuristically finds a good splitting variable by analyzing the static information of the design that is implicitly given by the partitioned transition relation. To understand how the static information could help finding the best variable, assume the same example in Fig. 5.1 (a), where q_1 is the splitting variable. The static information shows that none of the next state functions f_i depend on that variable q_1, i.e., $q_1 \notin support(f_i)$. In simple words, the variable q_1 does not influence any of the next state function (next state variables). Obviously, splitting with q_1 does not restrict any of the next state variable q_i' values. Hence this *increases the probability of having same values in both the split, eventually leading to the state overlap*. In the other case shown in Fig. 5.1 (b), e_2 is the splitting variable. The static information result shows that the variable (q_2) influences all the three f_i's. Trivially, the splitting with q_2 adds restrictions like positive cofactor split would restrict the variable q_2 to 1 in all the functions and vice versa. Hence this restriction constrains the possibilities of the next state variables, and tries to pull the image state space of the splits wide apart resulting in minimal overlap.

The *Minimal overlap* algorithm collects static information *influence* in the preprocessing step and represented as tables mapping each variable to its value in descending order. The value of a variable is the number of partition transition's (next state variables) it is involved in. This *influence* factors can be calculated over steps by repeated influence collection. In the end we will have two different factors, first, *lookaheads* denoted by $\mathcal{D}^{\uparrow}(q, l)$, is a set containing all the next state variables that are influenced by the variable q over l steps. Second, *lookback* denoted as $\mathcal{D}^{\downarrow}(q', l)$, is a set containing all the state variables that influence the next state variable q' in l steps back. Formal definition of *influence* is given below,

Definition 16 *Let $l_1, l_2 \in \mathbb{N}$ be influence lookaheads. For a given FSM $\mathcal{A}$, the influence $\Phi_{l_1,l_2}(q) \in [-1, 1]$ of a state variable $q \in E$ and where E is a set of state variables, with $|E| = m$, is defined as*

$$\Phi_{l_1,l_2}(q) = \frac{|\mathcal{D}^{\uparrow}(q, l_1)| - |\mathcal{D}^{\downarrow}(q, l_2)|}{m}. \tag{5.2}$$

- Set $\mathcal{D}^{\uparrow}(q, l_1)$ contains all state variables that get influenced by e in l_1 steps
- Set $\mathcal{D}^{\downarrow}(q, l_2)$ contains all state variables that influence e in l_2 steps

These sets are determined iteratively starting with $l_1 = 1$ and $l_2 = 1$. Each T_i directly corresponds to the truth value of the next state variable q'_i, so these sets will be computed by analyzing all T_i and q_i. For $\mathcal{D}^{\uparrow}(q, 1)$, the algorithm counts the partitions T_i that contain q, whereas for $\mathcal{D}^{\downarrow}(q, 1)$ the algorithm counts the state variables in the support set of T_i.

As indicated above, splitting on a variable v with high influence will lead to fewer cross transitions between the resulting partitions. The algorithm performs better if the splitting variable is conjunctively connected and degrades if disjunctively connected in the partitioned transition relations T_i. However, it is computationally expensive to analyze all Boolean connectives of the clauses of every T_i. The actual *Minimal overlap* algorithm (refer Fig. 5.2 for the pseudo code) picks a viable state variable for splitting. The state variables are categorized based on their influence and put into different sets (see line 9). The algorithm starts with the set containing variables with a high influence and check them against a balancing condition (see line 11). It is observed in many of the practical designs that the tight balancing condition, i.e., $\delta = \frac{1}{2}$, would not lead to satisfy many of the highly influenced variables. Therefore, this thesis relaxed the condition., i.e., $\delta = \frac{2}{3}$, to increases the chance for the algorithm to select the highly influenced variables that satisfy the balancing condition. Alongside, the algorithm computes the cost of these variables (lines 8 and 14) with the cost function from [97] that consists of a redundancy and a reduction factor defined in section 3.3.2.

$$\text{cost}(S, v, \alpha) = \alpha * \frac{MAX(|S_v|, |S_{\bar{v}}|)}{|S|} + (1 - \alpha) * \frac{|S_v| + |S_{\bar{v}}|}{|S|};$$

If none of the examined variables satisfied the balancing condition, the variable with minimal cost is selected (see line 15).

5.2 Distributed Checking Algorithm

Although the *Minimal overlap* algorithm is developed for sequential approaches, it can be applied to parallel algorithms very conveniently in order to obtain more synergy. The newly proposed distributed property checking algorithm utilizes the *Minimal overlap* algorithm. Similar to the basic parallelization, the algorithm is composed of an initial sequential stage and a subsequent parallel stage. In the sequential stage as a pre-processing all the nodes first parse the model and construct the transition relation of the design and the property. Second, the nodes create the state variable's influence table (L_Φ in Fig. 5.2) by analyzing all the partitioned transitions, which is later used by the *Minimal overlap* algorithm. Once the nodes are ready with the pre-processing phase, state space traversal proceeds sequentially on one node until a threshold limit on the BDD size triggers the state set distribution. The splitting into k subsets is performed in parallel and every node is responsible for getting its own disjoint part of the whole state set. All

```
// S is the current state set
// S1 and S2 are the resulting partitions
// LΦ is the ordered list of variables according to their influence
// δ is the memory balance factor
// α is the weight for the cost function
split(in: S, LΦ, δ, α; out: S1, S2)
  bestCost := LΦ.top()
  minCost := cost(S, bestVar, α)
  while (C = getCandidateSet(Φ) ) ∧  C ≠ ∅
    for all w ∈ C
      if max(|Sv|,|Sv'|) ≤ δ|S| then
        v := w; goto do_split
      else
        thisCost := cost(S, w, α)
        if thisCost < minCost then
          minCost := thisCost; bestCost := w
  v := bestCost
  do_split: S1 := Sv; S2 := Sv
```

Figure 5.2: State set splitting with the *Minimal overlap* algorithm.

the nodes use the *Minimal overlap* algorithm for state set splitting. In the parallel phase, the nodes start state space traversal independently on these disjoint subsets.

5.2.1 Dynamic Overlap Reduction

Initially, the overlap between state sets of network nodes is reduced using the *Minimal overlap* algorithm. However, in general the overlap may still pursue after a few steps of state space traversal. In order to further confine the overlap we perform *dynamic overlap reduction*. This is a methodology where we allow overlap to some extent and heuristically select a time frame to remove it periodically. The overlap removal is performed after the state set distribution. This method is iteratively executed either throughout the verification process or up to n times. An extra node called *coordinator* organizes the communication between the nodes and performs dynamic removal of state overlap. Fig. 5.3 illustrates the parallel approach using *dynamic overlap reduction*. The example parallel environment consists of 4 working nodes and 1 coordinator. The figure delineates only the parallel phase of the verification algorithm. The overlap removal algorithm for each node works in three steps:

1. Upon reaching a reduction time point, the node sends its frontier state set to the coordinator. The state set distribution time point, i.e., the *iteration* in Fig. 5.3 corresponds to the beginning of parallel phase and can be incremented thereafter. The reduction period usually given by the user it corresponds to the period of steps at which overlap removal is performed. The

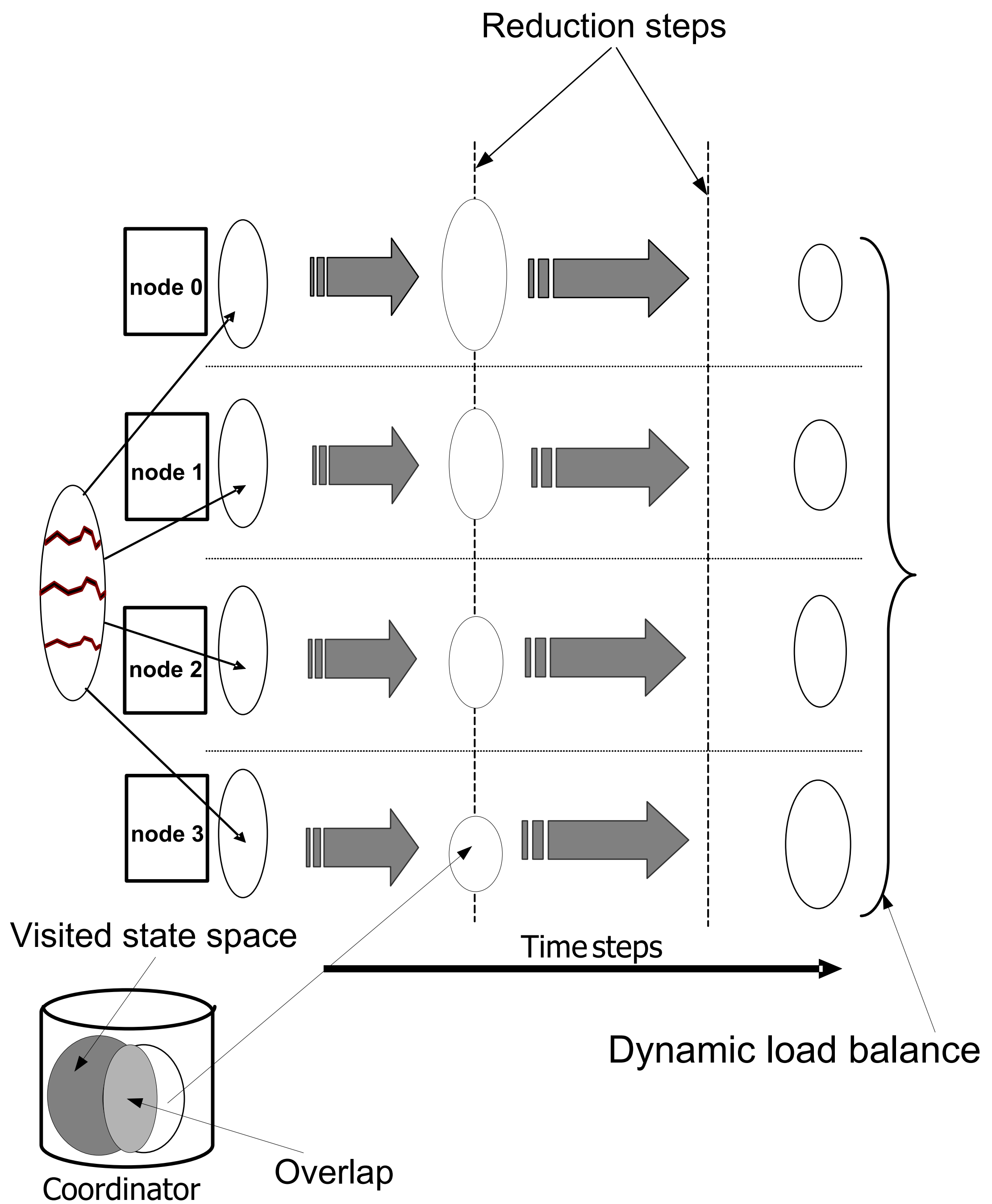

Figure 5.3: Dynamic overlap reduction based distributive approach.

iteration and reduction period determine the reduction time point. Which is described in line 5 of Fig. 5.5. Fig. 5.5 depicts the pseudo code for *removeOverlap* operation, which is described in line 19 of Fig. 5.3.

2. The coordinator removes the overlap of the node with respect to the already visited state space by other nodes at this time point (lines 12-13 show this operation in Fig. 5.5), and updates the history of the visited state space (line 20 shows this operation in Fig. 5.5). Then it informs the corresponding node to proceed with the reduced state space by sending the trimmed state set.

3. Finally, if all nodes passed a reduction time point, the coordinator removes the state space history of that time point (line 18 shows this particular operation in Fig. 5.5).

Fig. 5.4 delineates the usage of overlap reduction in the main computation loop of the symbolic simulation algorithm in the parallel stage. We have to check the termination condition locally, i.e., only in the frontier subset (line 14), and globally, which requires communication with the other nodes (line 9). For example, in order to show an universal validation, all nodes have to finish in accept states locally, which can only be checked globally.

```
// S is the set of initial states                                        1
// t is the checking time bound                                          2
// p is the period of steps at which overlap removal is                  3
performed
// n is the overlap removal limit, 0 indicates continuous                4
reduction
simulate(in: S, t, p, rank, n)                                           5
  reduction_limit := 0; reduction_step := 0                              6
  if n > 0 then tillEnd := false else tillEnd := true                    7
  while iteration < t                                                    8
    checkTerminationConditionGlobally();                                 9
    S := image_AR(S) // Compute image of AR-automata.                    10
    if( propertyViolation() == true )                                    11
       reportFailure();                                                  12
       contactCoordinator();                                             13
    checkTerminationConditionLocally();                                  14
    S := image_T(S) // Compute image of the system.                      15
    if (reduction_limit < n) ∨ tillEnd then                              16
      reduction_step++                                                   17
      if reduction_step = p then                                         18
        S := removeOverlap(S, iteration, p, rank)                        19
        reduction_step := 0; reduction_limit++                           20
    iteration++;                                                         21
```

Figure 5.4: Main computation loop for state overlap removal.

The main advantage of the dynamic reduction method is that nodes do not have to wait for slow nodes. After sending their current state set, faster nodes can

continue to traverse the product automaton. Therefore, we achieve asynchronous overlap removal between network nodes. Although nodes have to wait for the coordinator to update their state set, this time is not significant compared to the time spent on image computation.

The asynchronous methodology has the side effect of natural load balancing among network nodes. The very last node that reaches a reduction time point gets its overlap removed with respect to all other nodes. So this last node has no states in common with the other nodes at this reduction step. Usually the last node that reaches the reduction time point after overlap removal has the smallest subset. This in turn means faster image computation, enabling this node to reach the forthcoming reduction time point faster. Hence, at that reduction time point this particular node will arrive earlier than other nodes, and therefore continues with a larger state set. This process alternates among the nodes accordingly depending on the weight of image computation, resulting in natural load balancing between the network nodes. And it is tangible in Fig. 5.3.

```
// S is the set on which the overlap removal operation is          1
performed
// t is the checking time bound                                    2
// p is the period of steps at which overlap removal is            3
performed
removeOverlap(in: S, iteration, p, rank; out: S_new)               4
  r_p := iteration/p // reduction time point                       5
  // If this is the first node to reach this reduction time        6
point
  if(new(r_p))                                                     7
    addStateSpaceAtNewRedTimePoint(r_p, S);                        8
    S_new := S                                                     9
    return S_new                                                   10
  else                                                             11
    overlap := visitedStateSpace(r_p) ∧ S;                         12
    S := S - overlap                                               13
    if S == φ                                                      14
      waitUntilNextImmediateNode();                                15
    else                                                           16
      if (finalNodeAtThisRedTimePoint(r_p, rank))                  17
        removeVisitedStateSpace(r_p);                              18
      else                                                         19
        updateTheVisitedStateSpace(r_p);                           20
  S_new := S                                                       21
  return S_new                                                     22
```

Figure 5.5: Overlap removal operation.

For some models, the overlap is so high that the late nodes become empty after overlap removal (line 14 in Fig. 5.5). This special situation is handled by state set sharing by partitioning with the following node that reaches any reduction time point.

If any of the nodes detects the target state set during the state space traversal, it contacts the coordinator and asks to abort other nodes. The coordinator optionally computes the counterexample. Otherwise, the main computation loop of the algorithm is executed until the time bound specified in the property or an external time bound is reached. The following subsection explains the counterexample generation algorithm.

5.2.2 Counterexample Computation

Fig. 5.6 depicts the coordinator node's counterexample generation algorithm. The node that discovered the target state set first contacts the coordinator with its frontier state set, which contains the target states. Later it sends the sequential frontier state sets to the coordinator. After receiving these state sets (lines 4-5), the coordinator tracks down the reduction time step information of the node that detected the target state set. That is the information on each reduction time step the node shared its state space with any of other nodes. Sometimes, the nodes get empty state set result after overlap removal. In such cases it has to share state space with next immediate that reaches any reduction time point. For example, the node with rank 5 detected the target state set after reaching the reduction time step 8, but it has shared its state space at reduction time step 6 with node rank 4 due to an empty state set result after overlap removal. Then the state subsets of node 4 for all previous reduction steps have to be considered, i.e., from reduction time steps 1 till 5 from node 4 and 6 till 8 from node 5. The operation *trackTheRpDatabase* in the algorithm computes such information. Depending on the information, the coordinator receives the frontier state subsets of the nodes that are involved in the track record for their respective reduction time periods. This operation takes place in line 7 of the algorithm. After receiving the parallel frontier state subsets, the coordinator terminates the state space traversals of all other nodes (line 8). Depending on the track record, the operation *orderFrontierStateSubsets* creates the vector of parallel frontier state subsets and sorts them in chronological order, i.e., with respect to the reduction time steps. The lines 10-20 represent the standard counterexample computation code similar to the code specified in algorithm (in section 4.1.2).

5.2.2.1 BDD Minimization

The operation *overlap removal* is more effective if the size of the resulting pruned state subset is relatively small compared to the original state set. Otherwise, it could further aggravate the symbolic state traversal. The pace of symbolic traversal is sensitive to the sizes of BDDs that represent intermediate state sets during traversal. So, the sizes of BDDs determine the run-time efficiency of the methodology.

Possible cases for resulting state subsets after overlap removal are shown in

```
// T is the target state                                                    1
// rank is the rank of the node that detected the target                    2
state set
generateCounterExample(in: T, rank)                                         3
  receiveTheTargetStateSet();                                               4
  S_seq = receiveTheSequentialFrontierStateSets();                          5
  tRecord = trackTheRpDatabase(rank);                                       6
  receiveParallelFrontierStateSetsOfNodes(tRecord);                         7
  terminateOtherNodes();                                                    8
  St_par = orderFrontierStateSubsets();                                     9
  appnedParallelFrontierStatesIntoSequential();                             10
  i = St_seq.size() - 1;                                                    11
  S = T ∧ St_seq[i];                                                        12
  S = selectRandomState(S); store(S);                                       13
  while i ≥ 0                                                               14
    i--;                                                                    15
    // Compute pre-image of AR-Automata                                     16
    S = pre − image_AR(S);                                                  17
    // Compute pre-image of the system                                      18
    S = pre − image_T(S) ∧ St_seq[i];                                       19
    S = selectRandomState(S); store(S);                                     20
```

Figure 5.6: Coordinator node counterexample computation algorithm

Fig. 5.7. If we remove overlap (O) from the state set (S) it could result either in S_1 or S_2, i.e., the removal operation can either increase or decrease the resulting BDD representation of the state set. Obviously, S_1 is the optimal case. Therefore, special care should be taken in obtaining optimal sized BDDs during traversal. This new goal eventually leads to the BDD minimization problem.

The BDD minimization problem is a prevalent problem. There exist several heuristic algorithms [119, 120] and some of them include *constrain* and *restrict*. The *constrain* and *restrict* are also known as *generalized co-factors*. However, Hong et al. introduced the *Basic compaction* and the *Leaf-identifying compaction* algorithms in [121]. The underlying key idea is that for incompletely specified functions many BDDs can be used to represent the function and each associated with a different assignment of don't cares to binary values. The traditional algorithm *restrict* is often effective in BDD minimization but sometimes it can increase the BDD size. However, the user can adapt the algorithm such that if the application of *restrict* results in a BDD larger than the input BDD then the input BDD can be returned otherwise the BDD resulted from *restrict* operator can be returned. The following if statement depicts the usage of *restrict* operator. Where f is the BDD to minimize, @ represents the *restrict* operator and c represents the care set.

$$(|(f@c)| > |f|)?f : (f@c)$$

The Hong et al. algorithms perform safe BDD minimization, i.e., they guaranteed never to increase the size of the BDD.

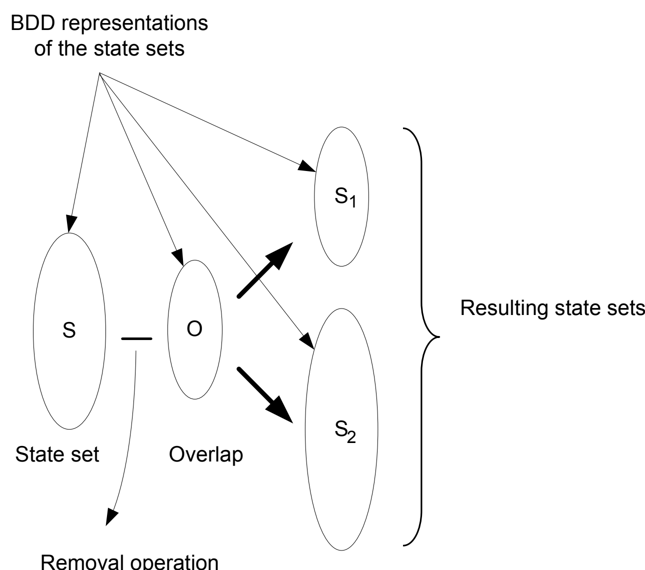

Figure 5.7: Possible cases for resulting state subset after overlap removal.

The distributed checking algorithm optionally uses (with user choice) *Leaf-identifying compaction*, *constrain* and *restrict* algorithms and performs BDD minimization, thereby reducing the peak memory requirements of intermediate BDDs.

5.2.3 Limitations of Dynamic Overlap Reduction Method

The *dynamic overlap reduction* method is best suitable for validation of designs. Since all the nodes perform asynchronous state space traversal on their whole state subsets, it is an onerous task for them to find deep errors, i.e., they need more traversal steps to find error states. This makes the method suitable for validation of a design with respect to the property. It is always a preferable choice to use a tool or method that supports both fast falsification and full validation together.

5.3 Conclusion

This chapter described an asynchronous algorithm for distributed bounded property checking. The main algorithm deploys a state-of-the-art *Minimal overlap* splitting heuristic which takes overlap reduction into account. Further, it proposed a novel *dynamic overlap reduction* on-the-fly algorithm for asynchronous state space traversal, which has the consequence of natural load balancing. *Dynamic overlap reduction* is an important technique in enabling verification of larger designs and significantly improves the applicability of the distributed algorithm. Reassigning idle nodes avoids wasted computation power. The state-of-the-art heuristic specified in section 3.3.4 reduces the overlap only up to a certain time steps (which in general can grow with time steps) by static means, whereas the *dynamic overlap*

reduction technique [122, 112] completely removes the state overlap in an asynchronous manner. Thereby, it smoothens the state space traversal of all the nodes.

Chapter 6

Hybrid Distributed Approach

But how shall I get ideas? Keep your wits open! Observe! Observe! Study! Study! But above all, Think! Think! And when a noble image is indelibly impressed upon the mind –Act!
- Orison Swett Marden

This chapter first gives an overview on the new hybrid distributed approach for fast property verification. Then it details the core algorithms, i.e., the types of nodes that are involved in this hybrid method and their responsibilities are introduced. Later, it explains each type of node in the parallel environment in detail. Then it expounds a counterexample computation algorithm in pertinent to the hybrid method. Finally, it discusses how window variables are generated and the partitioning algorithm is used for their generation.

6.1 Methodology

The parallel approach described in the previous chapter efficiently performs asynchronous state space traversals using *dynamic overlap reduction*. It heavily concentrates on reachability analysis within the time limit specified in the property. Since all the nodes work with their entire state spaces, the method requires more time to uncover error states that have longer counterexample lengths. This problem actuated the thesis to focus in developing a methodology to buttress both fast falsification and validation together in order to scale distributed verification and check the correctness of very large hardware designs.

The proposed new distributed algorithm is based on the well known technique called *windowing* [63, 35]. The windowing technique has partitions that are identified by unique combination of variables. Each *window* restricts its state space conforming to window variables. One aspect that significantly affects the overall effectiveness of windowing is *locality*. Locality means that most of the state's successors are assigned to the same computational node as the parent state, i.e., to the same window. Thus, this results in reducing the communication and

cooperation overhead. The states that belong to other windows are known as *cross over* or *border* or *non-owned* states.

6.1.1 Window States and Cross over States

Let S be the system state space and w_i and w_j are the window functions. Then the respective window state space can be computed using following equations:

$$S_{w_i} = S \wedge \mathrm{w}_i$$
$$S_{w_j} = S \wedge \mathrm{w}_j$$

Definition 17 *Let $T(\overrightarrow{q}, \overrightarrow{x}, \overrightarrow{q}')$ be the transition relation, where the sets $E = \{q_1, ..., q_m\}$ and $E' = \{q'_1, ..., q'_m\}$ represent the present and next state variables and $I = \{x_1, ..., x_k\}$ is a finite input alphabet. The S_i ($S_i \subset S_{w_i}$) represent the window state set. Then the successor window states from S_i can be computed using the window function w_i as follows*

$$S_{ii}(\overrightarrow{q}') = w_i(\overrightarrow{q}') \wedge \exists \overrightarrow{x}, \overrightarrow{q} \big(S_i(\overrightarrow{q}) \wedge T(\overrightarrow{q}, \overrightarrow{x}, \overrightarrow{q}')\big)$$

Cross over states are the states that transit from states in window i to the states in window j. They can be derived by conjoining the transition relation T with the respective window function.

Definition 18 *Let T, E, E' and I represent the transition relation, present state variables, next state variables and finite input alphabet, respectively (as described in above definition). The set S_i ($S_i \subset S_{w_i}$) represent the window state set. Then the cross over states from window i to window j, S_{ij} can be computed using the window function (defined in section 2.3.2) w_j as follows.*

$$S_{ij}(\overrightarrow{q}') = w_j(\overrightarrow{q}') \wedge \exists \overrightarrow{x}, \overrightarrow{q} \big(S_i(\overrightarrow{q}) \wedge T(\overrightarrow{q}, \overrightarrow{x}, \overrightarrow{q}')\big), \text{for } i \neq j.$$

The efficiency of the proposed algorithm mainly depends on the selected window functions. Therefore, this thesis also proposes a method to find the window variables on-the-fly, aiming at balanced partitions and reduction in cross over states. The explanation of window variable selection is postponed to section 6.4. Fig. 6.1 shows an example of a system and its windows. Cross over states are marked with dotted circles. Further, the approach also utilizes the *dynamic overlap reduction* technique in order to confine the overlap that could occur once a node switches its traversing mechanism from windowing to normal. The hybrid approach efficiently combines both windowing and *dynamic overlap reduction* techniques. The following algorithm explains the methodology in detail.

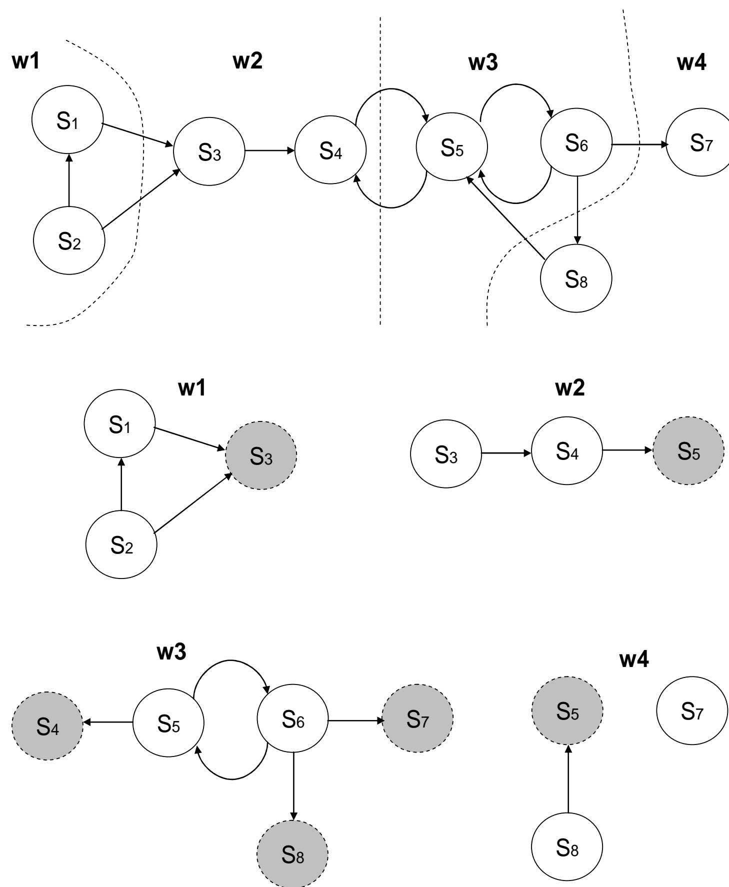

Figure 6.1: An example state graph and it's windows. The dotted circles represent cross over states of respective window.

6.2 The Distributed Algorithm

The nodes in the distributed environment are categorized into *windows*, *helpers* and one *coordinator*. Each window node restricts its state space to hold its uniqueness during the traversal. The helper nodes are responsible for cross over state traversal. The *coordinator* organizes the communication between *windows* and *helpers* and performs the removal of state overlap.

The core algorithm is composed of an initial sequential stage and a subsequent parallel stage. First, state space traversal sequentially proceeds on one node until a threshold limit on the BDD size triggers the state set distribution. Then the parallel stage starts with $k + 2$ nodes including one coordinator, one helper - is called *static helper* and k windows. The partitioning into k subsets is already performed in parallel (as described in section 4.1.1) and each window is responsible for getting its disjoint part of the whole state set and the window variables. The helper will wait for cross over states. Fig. 6.2, Fig. 6.3, Fig. 6.4 and Fig. 6.5 visualize the prominent cases involved in parallel phase of the algorithm. Each figure consists of two scenarios explaining both precondition (top) and postcondition (below) of that particular case. These cases will be explained in detail in the following paragraph.

Every window runs asynchronously on its restricted state space by sending the non-owned states to the coordinator. If error states occur in the window states, they are detected quickly. The coordinator assigns the traversal of cross over states that are not computed by windows to the helper (see Fig. 6.2). For some designs the windows could get empty images during the traversal steps. This emptiness of windows means that all the successor states of that particular windows are cross over states. From now onwards these windows are called *empty windows*. In such cases, the nodes of emptied windows become helpers. The newly converted helpers share the work with old helpers (see Fig. 6.3). The helpers run asynchronously with their assigned cross over states. The normal expected behavior is that the node containing window states, i.e., windows should run faster than the helpers. If any one of the window nodes is slower than one of the helpers then the respective window becomes the helper by sending its state space to the coordinator and eventually gets some work from the coordinator (see Fig. 6.4). The window that reaches the time bound also gets the helper status and receives work from the coordinator (see Fig. 6.5). The windows that dynamically becoming helpers are called as *dynamic helpers*. For the validation of a design, all the network nodes except the coordinator are terminated with the helper status. The state overlap between helper nodes is reduced by the coordinator using the *dynamic overlap reduction* principle. Since BDDs are used for state space traversal, sometimes the increase in state set representation due to overlap removal can be curtailed by using *BDD minimization* principles that have been presented in the previous chapter. In addition, natural load balance among helper nodes is achieved due to dynamic reduction of state overlap. The helper nodes that deal with large state spaces at one time point will be later assigned a small state space, and vice versa in an asynchronous way.

Thus all the window nodes try to find an error state quickly by restricting their state sets. However, the helpers complete the verification process left over by other network nodes by considering the entire cross over states of the windows.

The methodology can be varied to restrict the window nodes at a defined number of steps, rather than at every step. Consequently, windows consider the whole state space (window state space and cross over state space) in between each reduction step. This trivially reduces the helper's work and the communication overhead. The proposed symbolic verification algorithm is well suited for faster falsification and full validation. The approach is fully asynchronous and dynamic. Experiments (detailed in Chapter 10) show significantly faster verification on standard benchmarks. The following subsections cover each type of network node algorithm in detail.

6.2.1 Window Node Algorithm

Once the sequential phase is finished the pre-defined number of windows are enabled by providing the state space. For n windows we require $log_2(n)$ variables. The window variables are selected with the cost function explained in section 6.4. Every window loops the process mentioned in algorithm shown in Fig. 6.6 until the time bound is reached. This window restriction is performed either throughout the verification process or up to n times.

The function *checkTerminationConditionGlobally* (line 10) checks whether a property has been already proved by any of the other nodes. In case of a proved property, the whole process will be terminated. This information will be passed on by the network communication. If the property is not globally proved yet then the image of the AR-automata will be computed and checked for the correctness of the property in its state subset. In case it has detected a global violation condition then it asks the coordinator (line 14) to abort other nodes computations. This function *checkTerminationConditionLocally* (line 15) checks the local validation condition of a property. If it find the local validation then it contacts the coordinator to change the status to helper and to get some additional work. This condition is checked using the statement *checkToChangeTheStatus*. If no violation or validation condition is detected, then the image of the system will be computed.

Upon reaching a restriction time point, each window node sends its window state set S_i (line 23) and cross over state set CO_i (line 24) to the coordinator. While taking the disjoint part of the state subset in the beginning of parallel phase each window computes its window function on-the-fly. The detailed explanation on window variable calculation is postponed to section 6.4. After sending the state subsets the window asks the the coordinator to change its status to helper, in case its state set becomes empty after restriction, or if it is slower than the helpers computation phase. This work will be done by using the statement *sendStateSets* (line 25). Every window node iterates this whole loop till the time bound is reached.

6.2.2 Helper Node Algorithm

As mentioned earlier in this section the parallel stage starts with one *static helper* and *k windows*. The windows perform their asynchronous computation, whereas the *static helper* waits for cross over states. The helper also executes the code (line 9-14 in Fig. 6.7) in order to check if the termination condition of the property is reached. If the property is locally validated then it contacts the coordinator if there is some remaining work that it can share with other helpers. This condition is checked using the statement *checkToShareWorkWithOtherHelpers* (line 15). If the property is not proved yet then the image of the system will be computed on its assigned cross over state space.

Upon reaching a reduction time point the function *removeOverlap* (line 20) sends its current state set to the coordinator, which eventually remove the state overlap with respect to the already visited state space by other helpers. Thus *static* as well as *dynamic helpers* iterate through the operations involved in line 9-22 in Fig. 6.7 until they finish the verification task.

Similar to the restriction in case of window network node, the helper performs overlap removal either throughout the verification process or up to n times.

6.2.3 Coordinator Node Algorithm

The *coordinator* algorithm works in four steps:

1. Checks the global property condition. If none of the network nodes proved the property then it continues.
2. Receives all kinds of messages. They include normal window messages, emptied window messages, helper messages and property validated messages from windows and helpers.
3. Processes the window messages i.e., loads each window's window and cross over state space. Later it checks the window's computation pace. If the window runs slower than any one of the helpers, i.e., there exists one helper that has already visited the reduction time point then it converts the status of the window to helper. If the window comes with the validated message then the coordinator also converts the status to helper. In any of these cases it assigns some work to the newly converted helper. This work results from state set sharing with the following helper node that reaches the reduction time point. This time point is not necessarily the same as window's time point. In case there is not much work left then it requests the newly converted helpers to terminate. Otherwise, it informs the window to proceed with window state space traversal.
4. Processes the helper messages i.e., it adds the assignment of unprocessed cross over state sets to each helper. If the helper reaches the time point first

then it gets the assignment first. Let us assume the helper reached the time point t and k *windows* already processed this time point then unprocessed cross over states U can be computed as follows

$$U = (H \bigcup \sum_{i=1}^{k} CO_i) \setminus \sum_{i=1}^{k} W_i$$

Here W_i and CO_i represent the window and cross over state sets of *window* i respectively. H represents the current state set of the helper. After the assignment of set U to the helper the coordinator removes all the cross over and window state sets of that time point.

If the helper is not the first to reach the time point and there exist m nodes that already visited the time point then the helper gets its overlap removed with respect to the visited helpers. The overlap removal can be expressed as follows

$$R = H \setminus \sum_{i=1}^{m} H_i$$

Here each H_i represents the state space of the already visited helper node. H represents the current state set of the processing helper. R represents the trimmed state set. If this is the final helper node to reach the time point then it removes the state space history of that time point. If the result of the overlap removal operation is empty or the helper comes with validated message then the helper shares the work with the following node that reaches the time point.

Thus the coordinator organizes the communication, distributes the work efficiently and removes the overlap between network nodes.

6.3 Counterexample Computation

There are two types of nodes that could discover the target state set in the hybrid approach. Window nodes and helper nodes. As soon as any of those nodes detect the target state set then it informs the coordinator. The coordinator optionally computes the counterexample. The algorithm shown in Fig. 6.8 explicates the counterexample generation algorithm. The coordinator first receives the frontier state sets, i.e., states that were covered during the sequential phase of the distributed algorithm, from the node that detected the target state set (line 5). Then it checks the status of the detected node. If the node is a window node then it is relatively easy to compute the counterexample. It receives the node's parallel phase frontier state sets including the final state set that discovered the error states. After successful receival, it halts other nodes computations and executes the standard counterexample computation code (lines 15-24) by appending parallel state sets to the sequential state sets (line 14). If the node is a helper node

then the coordinator tracks down the complete record information of the node. If the node is a *static helper* then it is as simple as for window nodes to compute the counterexample. Otherwise, the record consists of all information, i.e., at which reduction time step the window node became the *dynamic helper* and with whom it has shared its space. Depending on this information, the coordinator receives the parallel state spaces of all the nodes that were involved in the track record (line 11). It arranges them in chronological order (line 13). This process is similar to the one specified in section 5.2.2, i.e., once the coordinator is ready with the required state space to construct the counterexample then it executes the standard code (line 16-25) [6] to construct a path from the target state to the initial state. The output of the algorithm is a trace in waveform file, which can be viewed using any waveform viewer.

6.4 Window Variable Selection

For the pre-defined 2^n windows the classical method requires exactly n variables to be selected. Algorithmically, finding such n variables for a balanced distribution is time consuming, hence this thesis proposes an alternative method to find the variables *on-the-fly* while solving the balancing condition. The *on-the-fly* approach selects the maximum of $2^n - 1$ variables for 2^n windows. The *on-the-fly* windowing partitioning method is efficient in producing balanced partitions among windows. The *on-the-fly* balancing window variable selection algorithm can be explained by assuming a set of states S. The basis for this algorithm is the parallel state set decomposition algorithm, specified in section 4.1.1. The first variable is selected in a way that it partitions the set into two balanced parts S_v and $S_{\overline{v}}$. Then the two parts are taken separately and searched for the second variable that partitions it according to the balancing condition. This procedure is repeated until 2^n windows are formed. Fig. 6.9 shows the *on-the-fly* balanced distribution of window partitioning, where there are 3 variables involved in obtaining 4 windows. The window state space W_1 can be obtained by restricting state set S with $e_1 \wedge e_2$. Similarly W_2 can be obtained with $e_1 \wedge !e_2$, W_3 can be obtained with $e_2 \wedge e_3$ and W_4 can be obtained with $e_2 \wedge !e_3$. Once all the window network nodes have respective window variables, they restrict the state space with respect to the window that it has at regular time intervals. The naive method requires n variables for 2^n windows and whereas *on-the-fly* method utilizes the maximum of $2^n - 1$ variables for window selection. Using more number of variables for window selection can statically minimize the cross over states during window traversals and is explained in [44].

6.4.1 Partitioning Algorithm

The influencing factors for the windowing technique are balanced partitions and reduced cross over states. Balanced partition means that every window not only

gets a balanced state space at the start of the windowing but also at every traversal step, because badly partitioned windows could get empty images during the traversal steps. The cross over states are a typical problem for the windowing technique. Therefore, the selected window partitioning algorithm should consider reducing the cross over states problem. The *Minimal overlap* algorithm [112] (detailed in Chapter 5) is applied for window partitioning. The algorithm utilizes the high influence cost factors to select the variables. The influence factor for each state variable results from static analysis in the pre-processing phase. The variables are ordered decreasingly by their influence. The author in [44] has shown that the high influence variables result in comparatively good distributions and reduced number of cross over states. The balancing condition and cost function of this partitioning algorithm are fixed to one that is specified in section 5.1.1.

6.5 Conclusion

The scalability problems of the parallelization scheme are cross over states among the partitions and state overlap. The extended hybrid method in this chapter alleviates these problems. The hybrid algorithm is an asynchronous distributed symbolic verification algorithm based on windowing and *dynamic overlap reduction* techniques, suited for full validation and fast falsification [123]. The core algorithm distributes partitions of the state set to computation nodes after reaching a threshold size. The nodes on the cluster machine are employed with two different types of tasks. Some nodes, windows, aim at faster falsification on the basis of the windowing technique. The other type of nodes called helpers are intending for validation on the basis of *dynanmic overlap reduction*, hence the term hybrid. All the cluster nodes asynchronously traverse their local state spaces for both error states detection and reachability of a time bound. Further, the hybrid approach makes the system properly balanced with respect to the node's work load.

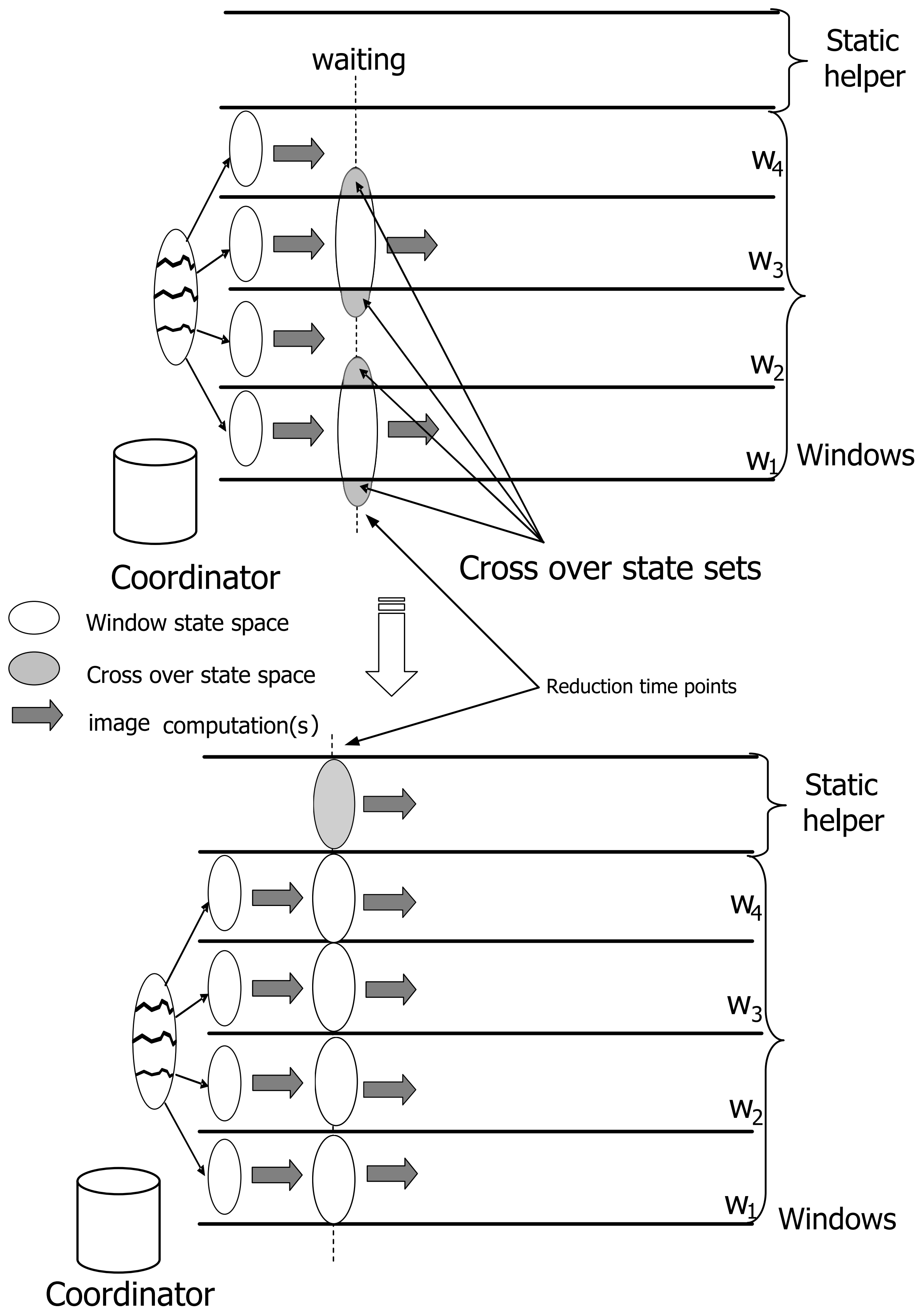

Figure 6.2: Case1: Initial phase of the core algorithm.

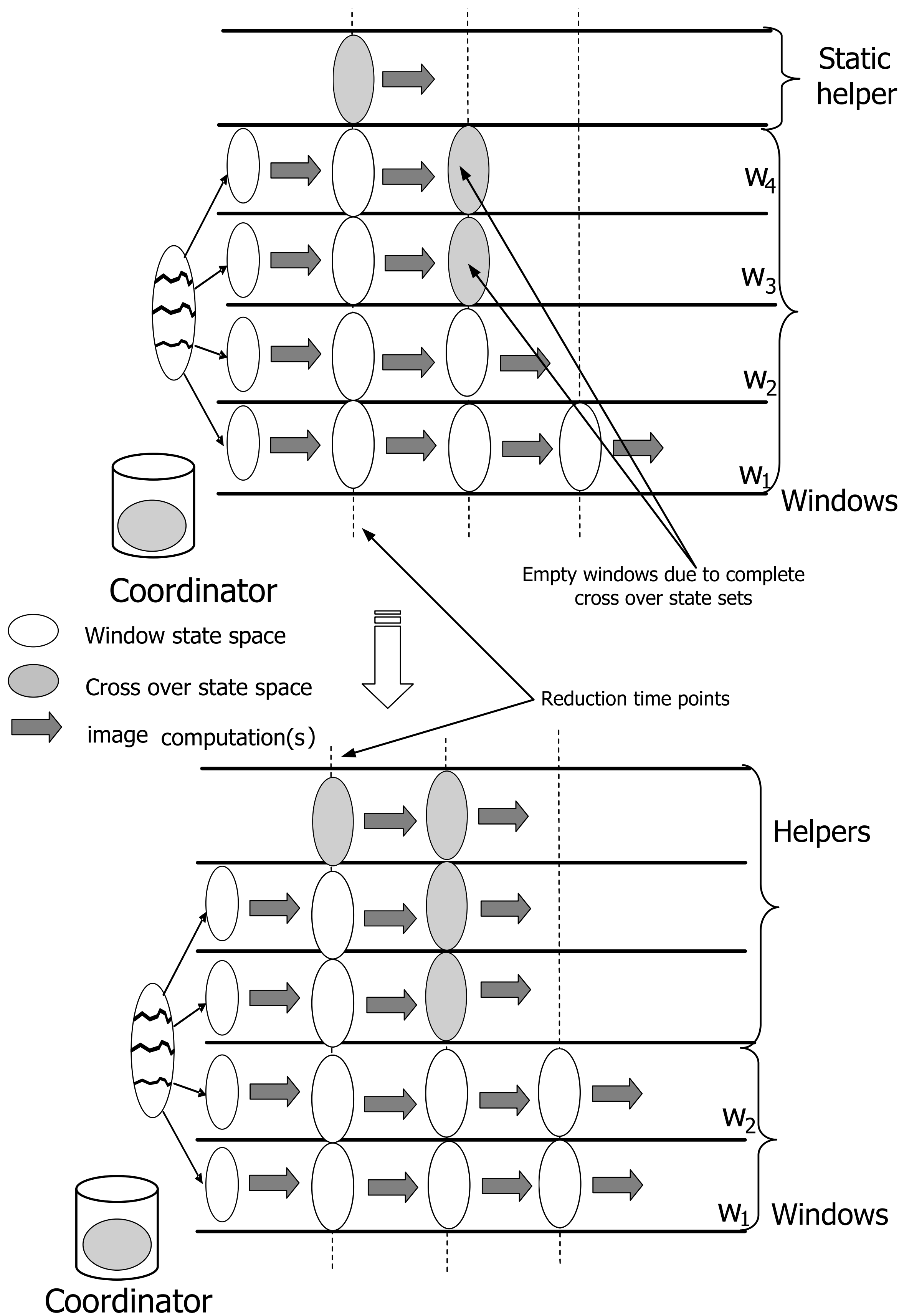

Figure 6.3: Case2: Converting empty windows to helpers.

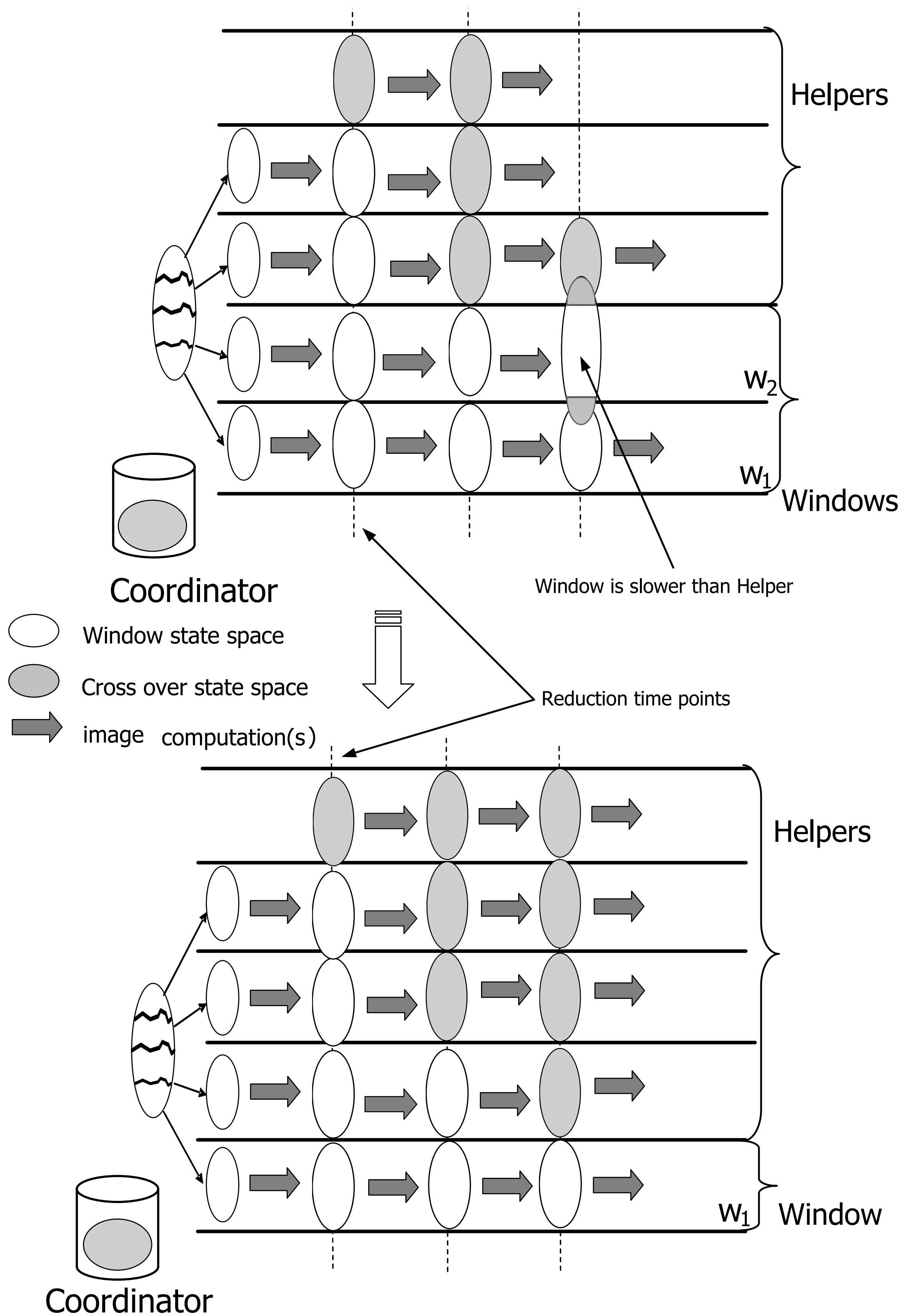

Figure 6.4: Case3: Converting slow windows to helpers.

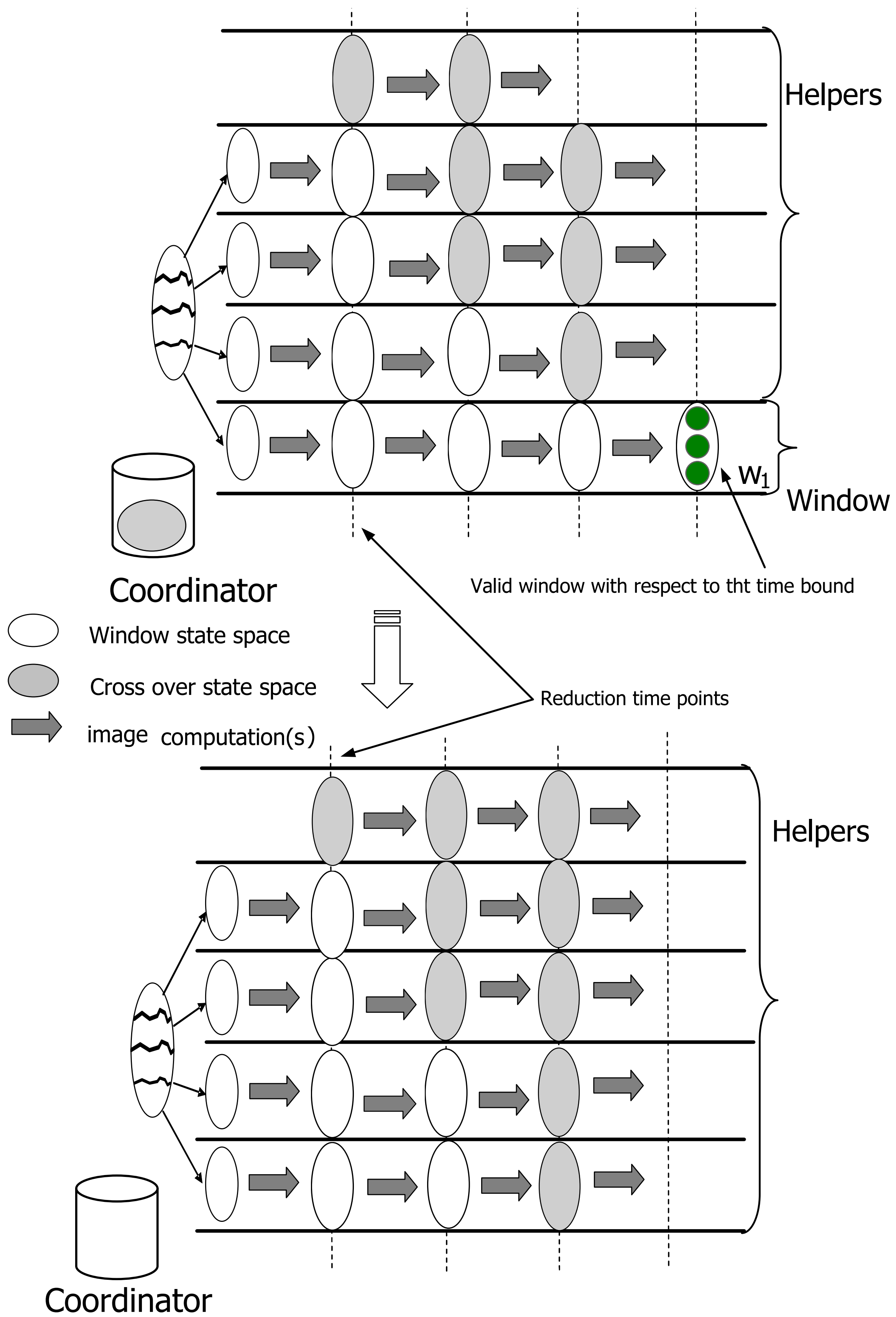

Figure 6.5: Case4: Converting valid windows to helpers.

```
// S_i is the restricted window state set                                     1
// t is the checking time bound                                               2
// p is the period of steps at which restriction is                           3
performed
// n is the restriction limit, 0 indicates continuous                        4
restriction
WindowSimulate(in: S_i, t, p, n)                                              5
  New := N := CO_i :=  false; convertToHelper := false                        6
  restriction_limit := 0; restriction_step := 0                               7
  if n > 0 then tillEnd := false else tillEnd := true                         8
  while iteration < t                                                         9
    checkTerminationConditionGlobally();                                      10
    S_i :=  image_AR(S_i) // AR-automata image comp.                          11
    if( propertyViolation() == true )                                         12
      reportFailure();                                                        13
      contactCoordinator();                                                   14
    checkTerminationConditionLocally();                                       15
    checkToChangeTheStatus()                                                  16
    S_i := image_T(S_i)  // system image comp.                                17
    if(restriction_limit < n) ∨ tillEnd then                                  18
       restriction_step++                                                     19
       if restriction_step = p then                                           20
          // where N represents the next state set                            21
          N  := S_i                                                           22
          S_i := N ∧ w_i // w_i represents the window function                23
          CO_i := N - S_i // CO_i = cross over states                         24
          convertToHelper = sendStateSets(S_i,CO_i,iteration);                25
          if convertTorHelper then                                            26
            New := getNewStateSet(iteration);                                 27
          restriction_step := 0; restriction_limit++;                         28
    iteration++                                                               29
```

Figure 6.6: Window network node algorithm.

```
// t is the checking time bound                                              1
// p is the period of steps at which removal is performed                    2
// n is the removal limit, 0 indicates continuous reduction                  3
// S represents the initial cross over states                                4
HelperSimulate(in: S, t, p, n)                                               5
  reduction_limit := 0; reduction_step := 0                                  6
  If n > 0 then tillEnd := false else tillEnd := true                        7
  while iteration < t                                                        8
    checkTerminationConditionGlobally();                                     9
    S :=  image_AR(S) // AR-automata image comp.                             10
    if( propertyViolation() == true )                                        11
      reportFailure();                                                       12
      contactCoordinator();                                                  13
    checkTerminationConditionLocally();                                      14
    checkToShareWorkWithOtherHelpers();                                      15
    S := image_T(S)  // system image comp.                                   16
    if(reduction_limit < n) ∨ tillEnd then                                   17
      reduction_step++                                                       18
      if reduction_step = p then                                             19
         S := removeOverlap(S,iteration);                                    20
         reduction_step := 0; reduction_limit++                              21
    iteration++                                                              22
```

Figure 6.7: Helper network node algorithm.

```
// T is the target state                                                        1
// rank is the rank of the node that detected the target states                  2
// status is the status of the detected node                                     3
generateCounterExample(in: T, rank, status)                                      4
  S_seq = receiveTheSequentialFrontierStateSets();                               5
  if (status == window_status)                                                   6
     St_par = receiveParallelFrontierStateSetsOfNodes();                         7
  else // Helper status                                                          8
    tRecord = trackTheRpDatabase(rank);                                          9
    receiveParallelFrontierStateSetsOfNodes(tRecord);                            10
  terminateOtherNodes();                                                         11
  if(status == helper_status)                                                    12
    St_par = orderFrontierStateSubsets();                                        13
  appendParallelFrontierStatesIntoSequential();                                  14
  i = St_seq.size() - 1;                                                         15
  S = T ∧ St_seq[i];                                                             16
  S = selectRandomState(S); store(S);                                            17
  while i ≥ 0                                                                    18
    i--;                                                                         19
    // Compute pre-image of AR-Automata                                          20
    S = pre − image_AR(S);                                                       21
    // Compute pre-image of the system                                           22
    S = pre − image_T(S) ∧ St_seq[i];                                            23
    S = selectRandomState(S); store(S);                                          24
```

Figure 6.8: Coordinator node counterexample computation algorithm

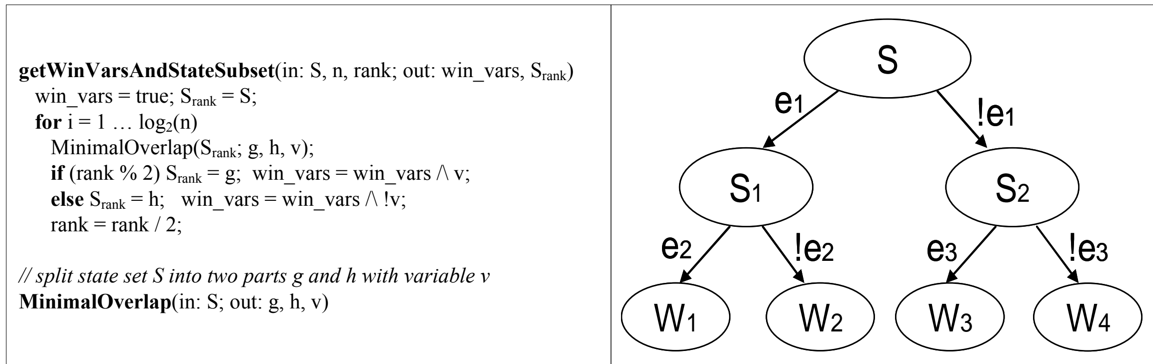

Figure 6.9: The left hand side shows the algorithm for distributing the state set and selecting window variables. The right hand side shows the on-the-fly balanced window partitioning.

Chapter 7

Grid-based Fast Falsification

It is not enough to have many resources; the main thing is to use them well.
- Variant quote of René Descartes

This chapter introduces the new distributive approach using grid. It explains in detail the grid based distribution algorithm for BDD-based bounded property checking, which is best suitable for fast falsification. This approach utilizes intelligent partitioning techniques based on given properties and the transition relation of a design.

7.1 Why Grid

The proposed solutions are to deal with the formal verification of large designs. However, there exist very large industrial designs for which finding an error state or target state is still crucial. Often, the target states are located very deep inside the system (i.e., verification tools need larger time bounds to reach the target states), such that the existing parallel approaches cannot uncover these states due to a limited number of resources available. One way to alleviate the problem of handling big industrial size designs is is to combine and adapt the advantages of state of the art techniques on a grid framework. In other words, the underlying verification algorithm, bounded property checking, is effectively combined with state-of-the-art techniques like under-approximation, guiding and minimal redundant computation algorithms which are then adapted for the grid environment to handle industrial size designs. Moreover, the grid approach pioneers in being asynchronous.

7.1.1 Grid Computing Overview

Grid computing is an emerging technology that enables large scale resource sharing and coordinated problem solving within distributed by providing secure and high-performance mechanisms for discovering and negotiating access to remote

resources. Grid technologies promise to make it possible to share resources on an unprecedented scale. A grid is a type of parallel and distributed system that enables sharing, selection, and consolidation of resources distributed across multiple domains based on their capacity and performance [124]. The grid is a unique environment where massive parallelism is possible by using idle CPU cycles from a large number of computers. Such computers may even be in geographically diverse locations. Since a grid network is not dedicated continuously to a single task, any task scheduled on a grid has to use very little communication or network bandwidth [110]. This makes the grid system different from tightly coupled homogeneous clusters.

7.1.2 Grid Application

The application of grid to verification is relatively new. The proposed approach parallelizes the symbolic state space traversal on a grid framework. In this approach, the main distribution of the state space among the nodes is carried out by a state of the art partitioning algorithm, *Minimal overlap*, which pioneers in minimizing the cross over state transitions among the partitions. Applying this algorithm the communication overhead among the computing nodes can be minimized. The approach also utilizes the state of the art intelligent guiding heuristics [114] based on properties. The *guiding* algorithm aims at fast falsification by steering the traversal. The fast falsification approach tries to identify so-called high priority traces and traverses those traces, with the aim of finding a violation of the property. Grid based bounded property checking for error detection is practical and effective [110, 125].

7.2 Grid-based Distribution Algorithm

The proposed new algorithm exploits the distributed environment with the effective combination of algorithms in order to find bugs faster in large designs, which minimizes the redundant work and guides to the target state. The efficiency of our approach mainly depends on the effective combination of the following:

- Grid computing: As computers acquire more computing capacity the concept of grid computing is becoming prevalent [126].
- Partitioning algorithm: The partitioning algorithm splits the state space into n splits with the intent of minimizing the cross over transitions among the nodes. This in principle decreases the revisiting of states and therefore supports fast traversal.
- Underapproximation algorithm: The underapproximation algorithm enables to follow only the interesting set of paths and avoids the others, thereby increasing the speed of the traversal further.

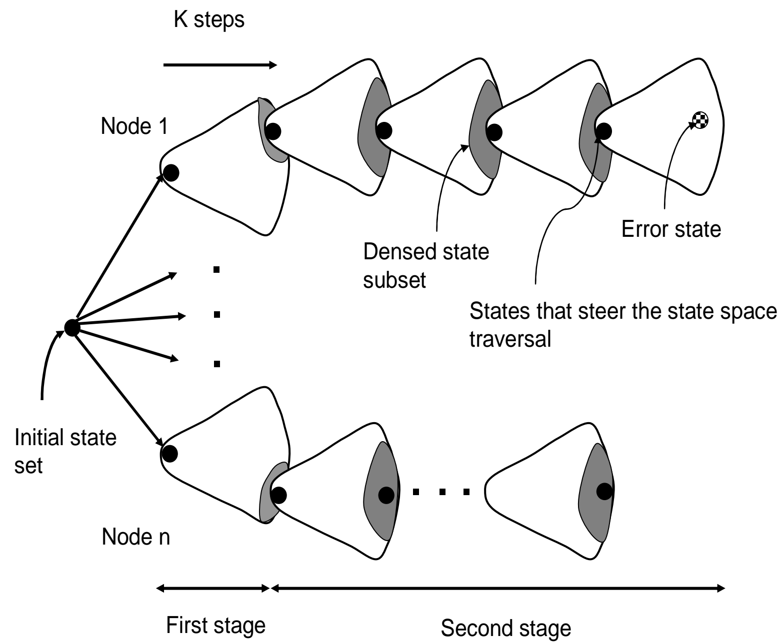

Figure 7.1: Grid based parallelization of the bounded property checking

Fig. 7.1 gives the overall view of the grid approach. As similar to previous approaches, for better memory balancing and also for parallel state set decomposition the number of nodes on parallel environment is restricted to n nodes, where $n = 2^k$ and $k \in \{1, 2, 3, \dots\}$. The core algorithm consists of an initial first stage and a subsequent second stage. In the first stage, all the nodes parse the system description and the property specification and translate them into BDDs. Once the nodes are ready with building the BDDs, they start the traversal in the set of initial states and iteratively compute the frontier set until the current state set reaches a given threshold size. Communication among the loosely coupled systems is relatively slow. Therefore the approach forces all the nodes to do the initial sequential stage on their own and avoid the communication overhead caused by the distribution of the current state set BDD.

The first property checking algorithm for each node continues until the size of the current state set reaches the threshold limit. Then the second stage starts with partitioning the current state set into subsets, where each node on the grid is responsible for taking its disjoint state subset. The initial state set decomposition is performed using the *Minimal overlap* algorithm [112]. In order to minimize the revisiting of states in the partitions, the algorithm heuristically selects a splitting variable v that will partition the states such that they have a minimized number of common next states. Similar to previous approaches, each node iteratively applies the *Minimal overlap* algorithm and splits the current state set into two parts and drops one of the resulting sets. In the end it keeps the subset that belongs to the node identified by its rank.

Once the node has its subset, it proceeds to perform a forward state space traversal in iterative BFS steps. Each node traverses its partitions in parallel, and

this second stage is again threshold controlled. In other words, whenever the size of the BDD representing the current state set of a node reaches the threshold limit then first densed underapproximation [33] is performed on the current state set to extract a large percentage of the state space and then the guiding algorithm is activated for intelligent underapproximation.

As described in section 3.3.1 the density corresponds to a concise representation, i.e., a relatively smaller BDD can represent more number of states. The high density of interest for the verification as smaller the BDD size faster the verification process. The algorithm *Short path subsetting* is used for densed under approximation (line 16 in Fig. 7.2).

The guiding algorithm heuristically restricts the state set to a subset in which there are states that could lead to the target state. There are two different guiding strategies - *State variable guiding* and *Input variable guiding*.

- State variable guiding : The actual state set is partitioned into two parts using one of the interesting state variables that are collected.

- Input variable guiding : The actual state set is restricted with the set of input variables (hints) such that the transitions satisfying those set of variables are allowed and the others are discarded.

In general, the authors in [114] inferred that each guiding algorithm has it's own significance and it is purely depending on the design which guiding algorithm should we follow. Input variable guiding gains if more input variables are involved in the design, but state variable guiding is better if the number of interesting state variables are around 30% to 60% of total state variables in the design. Therefore, both algorithms are adapted for grid approach with even number ranking nodes utilize the *Input variable guiding* (line 20 in Fig. 7.2) and nodes with an odd number ranking utilize the *State variable guiding* (line 18 in Fig. 7.2).

After having performed the underapproximations, all the nodes proceed with forward state space traversal on their restricted subsets until they reach the termination condition. The termination condition for the guiding has been adopted such that, whenever one of the nodes detects the global violation condition or an error state, it initiates abortion of the other nodes by sending a message to all other nodes. Fig. 7.2 delineates each node's main computation loop of the symbolic simulation algorithm. The termination condition has to be checked locally, i.e., only in the current subset (line 8), and globally, which requires communication with all other nodes (line 6). The node that detected the global violation condition optionally compute the counterexample by using the similar algorithm specified in section 4.1.2. In case of no global violation condition is met then the traversal on all nodes continue up to a given time bound, which is either given explicitly by the user (line 2) or implicitly by the property.

```
// S is the set of initial states                                     1
// t is the checking time bound                                       2
symbolicSimulate(in: S, t)                                            3
 parFlagTrigger = false;                                              4
 while iteration < t                                                  5
   checkTerminationConditionGlobally()                                6
   S := image_AR(S) // Compute image of AR-automata.                  7
   checkTerminationConditionLocally(S);                               8
   S := image_T(S) // Compute image of the system.                    9
   if ||S|| > threshold                                               10
     // The condition to trigger the parallel phase                   11
     if parFlagTrigger == false                                       12
       S := getTheCorrespondingStateSubset(S, my_rank);               13
       parFlagTrigger = true;                                         14
     else                                                             15
       S := densedUnderApproximate(S);                                16
       if my_rank % 2 == 0 // Even rank                               17
         S := stateVarGuide(S);                                       18
       else                                                           19
         S := inputVarGuide(S);                                       20
   iteration++;                                                       21
```

Figure 7.2: Main computation loop for state overlap removal.

7.2.1 Underapproximation using Guiding Algorithms

The *guiding* algorithm [114] automatically finds the set of interesting variables by exploiting the property and the transition relation T of a design. This property based state space guiding algorithm can substantially speed up the verification process by picking up the interesting state or the input variables automatically through the influence factors and utilizes them in guiding the state space traversal. The guiding algorithms basically aim at fast finding of target states.

Guiding based on state variables entails partitioning the state space into two by splitting using a selected state variable in such a way that one of the splits has a higher concentration of potential target states and is comparatively smaller. In contrast to state variable guiding the input variable guiding basically restricts the possible next state set to a smaller subset, rather than splitting. This restriction is done in such a way that this subset has a higher concentration of the potential target states. Further traversing this smaller subset is faster and it is easier to reach the error states.

One of the main steps of guiding is to identify the right partition, i.e., the one having more potential target states, to traverse first. This step of the guiding procedure is called *steering*, which is done by the cost function defined as in [114]. The key factor of the cost function is minterm counting to guess the cofactor of the variable in the transitions, which then can be used to identify the right partition to follow.

7.3 Conclusion

Formal verification, in particular finding a bug or a witness, is increasingly important facing the growing complexity of designs. It has proved its potential and is being utilized increasingly in the industry. However, increasing design sizes still leaves verification as the major bottleneck, because the formal techniques do not yet scale to large designs. One approach to further scale up the methodology is to use a grid based distributed platform. This chapter presented a new grid based distributed bounded symbolic verification approach based on effective combination of the intelligent partitioning algorithms that suits best for fast falsification [125]. This grid approach is an asynchronous verification environment which is based on loosely coupled machines that significantly speedup the verification. One of the reasons for the speedup is the minimized communication among the nodes. Although the algorithm has potential to uncover the target states which have the longer counterexample lengths, it cannot support full validation due to underapproximation techniques used by the methodology.

Chapter 8

Parallelization of Black Box Verification

You are never too old to set another goal or to dream a new dream.
- C.S. Lewis

This chapter considers how to perform parallel model checking for *incomplete* designs. First, it details the usual preliminaries required for Black Box verification, i.e., what are the verification goals, under and overapproximated versions of transition relation, difficulties faced by the sequential approaches, etc. Next, it expounds the parallel mixed traversal approach for verifying larger *incomplete* designs. The core algorithm employs some nodes on the cluster computer with forward state space traversal and others are recruited for backward traversal. Later the algorithm checks for intersection between forward and backward state spaces. All the nodes perform under and overapproximation during the traversal using restricted transition relations. Finally, it provides the conclusions for this chapter.

8.1 Black Box Verification

Model checking of *incomplete* designs, i.e., designs which contain unknown parts is becoming prevalent and also attracting industrial community and it has many advantages [74]. Black Box verification enables the use of verification techniques in early stages of the design. Design errors can be already detected when only a partial implementation is at hand - e.g. due to a distribution of the implementation task to several groups of designers. Parts of the implementation, which are not yet finished, are combined into Black Boxes. If the implementation differs from the specification for all possible substitutions of the Black Boxes, a design error is found in the current partial implementation, i.e. to detect an error in the current partial implementation it is necessary to find an assignment of zeros and ones to the primary inputs, which produces erroneous values at the outputs independently from the final implementation of the Black Boxes. Another applica-

tion of Black Box Equivalence Checking is the *abstraction* from "difficult parts" an implementation, which would cause a large peak size in memory consumption during the construction of a canonical form for the implementation. These "difficult parts" of the design can be put into a Black Box and Black Box verification is performed.

8.1.1 Why Parallelization

Despite the developments [74, 75, 76] in the last years, the so-called *state space explosion* still limits the application of Black Box verification for large industrial designs. In addition, it is still a concern in which way to perform traversal in model checking algorithms. Though the authors in [75, 76] stick to backward traversal approaches, for some designs forward traversal outperforms backward traversal. Hence, the effectiveness of the direction of traversal is purely depend on the design as well as the property to be verified. In order to overcome the problems with the direction of traversal and to increase the computational resources available to the verification process both forward and backward traversals can be performed in parallel using networks of computers. Further, traditional modelling of non-deterministic signals for Black Box outputs using conventional model checkers does not provide correct results for *incomplete* designs. Therefore, adjustments were needed to handle approximate methods for providing validity and falsify realizability of Black Box implementations. Especially, the handling of improved construction for $Sat_A(\varphi)$ (described in section 3.1.4) is relatively hard and need to modify the existing image computation algorithm. The preliminaries and state-of-the-art concerning the Black Box verification were explained in detail in sections 2.8 and 3.1.4.

One feasible way to handle the state explosion problem for larger industrial sized designs is to parallelize the Black Box verification. So as an application, this thesis presents a *novel* parallel mixed traversal approach for the verification of larger industrial sized designs. The distributed algorithm even considers the adjustments needed for approximate methods and also handles the improved construction of $Sat_A(\varphi)$.

8.2 Combination of Forward and Backward Traversal on a Distributed Platform

The idea of mixed traversals was introduced by Govindarajulu et al. in [79]. However, the idea was utilized and extended using other state-of-the-art technologies by Cabodi et al. in [81, 82] but only limited to sequential approaches. Fig. 8.1 illustrates such an approach on a parallel platform. It uses $m + n + 1$ number of nodes on a distributed platform, where m and n are the number of nodes representing forward and backward nodes respectively ($m = n = 2^k$ and $k \in \{1, 2, 3, \dots\}$). One extra node (*coordinator*) is responsible for organization and

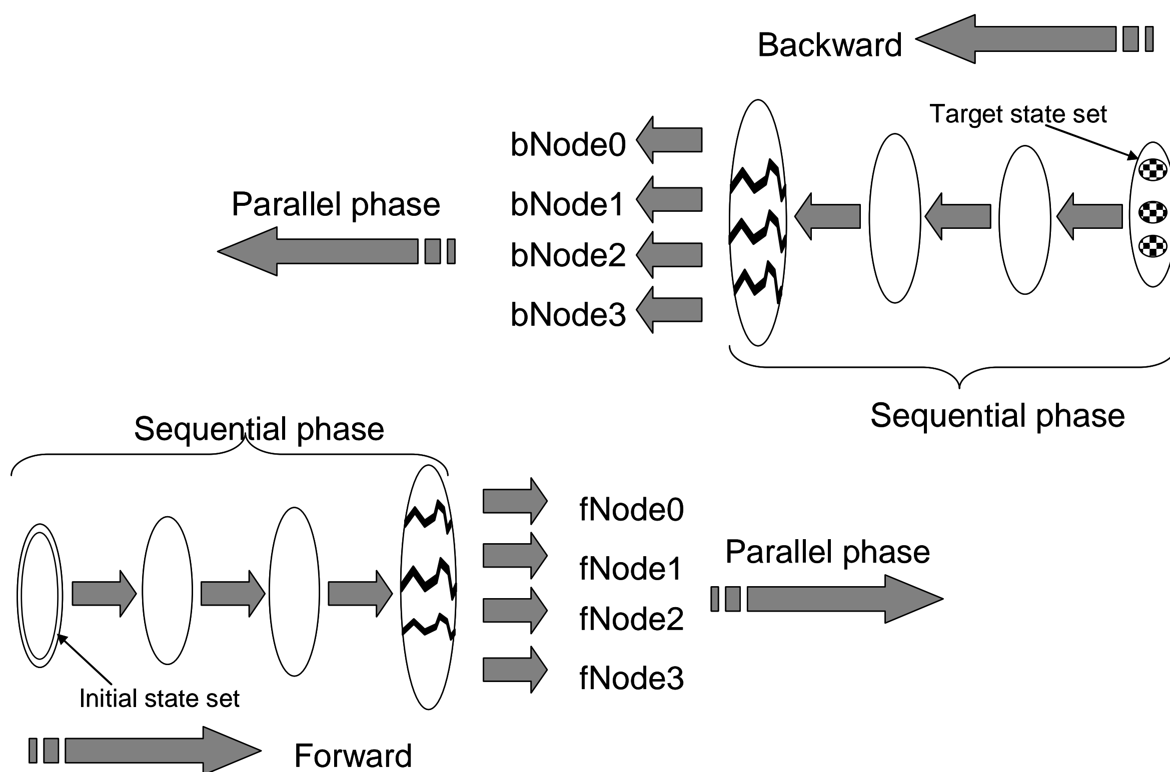

Figure 8.1: Outline of the basic distributed algorithm. After reaching the splitting threshold, the state set is partitioned into subsets and these are distributed on computation nodes for independent traversal on both sides.

checking of the state space intersection between forward and backward nodes. The distributed approach on both sides consists of an initial sequential stage and a subsequent parallel stage. Fig. 8.1 illustrates this process. The coordinator node is not shown in this figure.

In the sequential stage on both sides, one node - known as the master node: starts its symbolic state traversal algorithm, whereas the slaves remain in waiting state. The forward master node begins its traversal on the set of initial states whereas the backward master node begins with the set of target states. The forward node collectively computes successor states of both *possible* and *fixed* states using T. On the other hand, the backward node performs two pre-image computation steps separately, i.e., one for *possible* and the other one for *fixed* transitions. In contrast to the forward node, the backward node uses only the over-approximated version of the transition relation, i.e., TR_E. Upon the successful completion of all image and pre-image computation steps for forward and backward traversals, each node sends its *fixed* and *possible* state sets to the coordinator, which eventually checks if there is an intersect between forward and backward state spaces. The computation of the master on either side continues until the size of the current state set reaches the threshold limit. At this point, similar to the previously described parallel approaches it broadcasts the current state set to all the slaves and indicates them to split. Depending on the node rank each node

splits the current state set and obtains its subset. After all nodes have their respective state subsets the parallel phase of the algorithm begins. During this phase, all slave nodes imitate the sequential steps of their respective master node. The computation loops of all nodes have to be repeated until either a global violation condition is met or a fixed-point condition is reached for both *fixed* and *possible* states. The following subsections explicate each type of network node algorithm.

8.2.1 Backward Node Algorithm

Though the overall algorithm shown in Fig. 8.2 is self explanatory, some of the lines will be explained explicitly. The arguments T and t for the function *bwdSymbolicSimulate* represent the transition relation and the external time limit, respectively. All backward nodes begin computation with this algorithm. In the beginning slaves wait for sure and possible state sets (as shown in line 8), whereas the master node performs the sequential stage alone. Once the size (BDD) of the state set, i.e., either possible or sure reaches the threshold limit then the master node distributes the state sets to all backward slaves (shown in 16). Depending on the node rank each node gets its state subset (shown in lines 9-10 and 17-18 for slave and master nodes respectively). After the state set distribution all nodes perform pre-image computation steps on their state subsets. It is shown in line 22 that the pre-image computation step for *fixed* states uses an overapproximated version of the transition relation TR_E. If we reach a local fixed-point condition, the node will be terminated locally (shown in line 35). The function *checkForGlobalTermination* checks whether an intersection has already been met due to any of the other nodes by asking the *coordinator*. In case of a proved property, the whole process will be terminated. If we do not meet the local or global termination conditions then each node sends its state subsets to the coordinator to update the whole backward database that it is maintaining and asks for checking of intersection with the forward node's state space. In case the intersection is not met then the coordinator sends back the newly reached next possible and sure sets to the backward node. This operation (communication with coordinator) is performed in lines 38 and 39.

```
                                                                         1
bwdSymbolicSimulate(in: T, t)                                            2
  Sure = Possible = newSure = newPoss = bad_states(q⃗′);                  3
  seqPhase = true; sureFixReach = false;                                 4
  possFixReach = tReach = false; possCompute = true;                     5
  // All nodes are in waiting state except Master node                   6
  if(node ≠ bwdMaster)                                                   7
    waitForSureAndPossibleStateSets();                                   8
    Sure = getTheCorrespondingPiece(Sure);                               9
    Possible = getTheCorrespondingPiece(Possible);                       10
  while iteration < t                                                    11
    if(seqPhase && (node == bwdMaster))                                  12
      tReach = (Sure.size() ≥ threshold);                                13
      tReach = tReach || (Possible.size() ≥ threshold);                  14
      if(tReach)                                                         15
        distributeTheStateSetsToBwdSlaves(Sure,Possible);                16
        Sure = getTheCorrespondingPiece(Sure);                           17
        Possible = getTheCorrespondingPiece(Possible);                   18
        seqPhase = false;                                                19
    // Pre-image computation step for sure states                        20
    if(¬sureFixReach)                                                    21
      newSure(q⃗′) = (∃x⃗¬(∃q⃗′(T|e=1(q⃗, x⃗, q⃗′) ∧ ¬Sure(q⃗′))))|q⃗←q⃗′            22
      // Checking local sure fixed-point                                 23
      if(Sure(q⃗′) == newSure(q⃗′))                                       24
        sureFixReach = true;                                             25
        sendBwdSureReachFixedpointMsg(node_rank,iteration);              26
        newSure = ⊥;                                                     27
    // Pre-image computation step for possible states                    28
    if(¬possFixReach && possCompute )                                    29
      newPoss(q⃗′) = (∃x⃗(∃q⃗′(T|e=1(q⃗, x⃗, q⃗′) ∧ Possible(q⃗′))))|q⃗←q⃗′;        30
      // Checking local possible fixed-point                             31
      if(Possible(q⃗′) == newPoss(q⃗′))                                   32
        possFixReach = true;                                             33
    if(possFixReach)                                                     34
      localTermination(); // Validity condition met                      35
    checkForGlobalTermination();                                         36
    // State set update                                                  37
    Sure(q⃗)|′q⃗←q⃗ = bwdSureContactCoordinator(newSure(q⃗′)|q⃗′←q⃗);          38
    Possible(q⃗)|′q⃗←q⃗ = bwdPossContactCoordinator(newPoss(q⃗′)|q⃗′←q⃗);      39
    if(Possible == ⊥)                                                    40
      possCompute = false;                                               41
    iteration++;                                                         42
```

Figure 8.2: Backward node's computation loop.

8.2.2 Forward Node Algorithm

The forward node algorithm is shown in Fig. 8.3. Similar to the backward node algorithm, T and t in Fig. 8.3 represent the transition relation and the external time limit, respectively. The algorithm works similarly to the backward approach except it does one image computation on the whole state space. Later the possible and sure sets will be separated by cofactoring (shown in lines 20-21). If we do not meet the local or global termination conditions then each node sends its state *fixed* and *possible* subsets to the coordinator, which eventually updates the whole forward state space that it is maintaining and asks for checking of intersection with the forward node's state space. In case, the intersection is not met then the coordinator sends back the newly reached next possible and sure sets to the forward node. This operation (communication with coordinator) is performed in line 23 of the algorithm.

```
fwdSymbolicSimulate(in: T, t, node_rank)                                   1
  Sure = Possible = Curr = init_states(q⃗);                                 2
  New = φ; seqPhase = true; tReach = false;                                3
  // All nodes are in waiting state except Master node                    4
  if(node ≠ fwdMaster)                                                     5
    waitForTheStateSet();                                                  6
    Curr = getTheCorrespoinfingPiece(Curr);                                7
  while iteration < t                                                      8
    tReach = (Curr.size() ≥ threshold);                                    9
    if(tReach && seqPhase && (node == fwdMaster))                         10
      distributeTheStateSetToAllFwdNodes(Curr);                           11
      Curr = getTheCorrespoinfingPiece(Curr);                             12
      seqPhase = false;                                                   13
    // Image computation step                                             14
    New(q⃗) = (∃x⃗(∃q⃗(T(q⃗,x⃗,q⃗′) ∧ Curr(q⃗))))|q⃗′←q⃗;                          15
    // Checking Local fixed-point                                         16
    if(New(q⃗) == Curr(q⃗))                                                 17
      localTermination();                                                 18
    checkForGlobalTermination();                                          19
    Sure(q⃗) = New(q⃗)|e=0;                                                 20
    Possible(q⃗) = New(q⃗)|e=1;                                             21
    // State set updation                                                 22
    Curr(q⃗) = fwdContactCoordinator(Sure(q⃗),Possible(q⃗));                 23
    iteration++;                                                          24
```

Figure 8.3: Forward node's computation loop.

8.2.3 Coordinator Node Algorithm

The coordinator's algorithm is shown in Fig. 8.4. It receives all messages from both *forward* and *backward* nodes. This particular operation is shown in line 5. These messages will be processed in chronological order. As a processing operation, the coordinator first receives the state sets from the respective node. For forward nodes lines 10-11 and for backward nodes lines 32-33 show this operation. Second, it updates the databases of all forward and backward nodes. Lines 13-14 show this operation for forward traversal and lines 35-36 for backward traversal. Third, it checks for state set intersection between forward and backward state spaces. Lines 16-18 show this operation for forward traversal and lines 38-40 for backward traversal. In case no intersection is met then it computes the newly reached state sets by removing the visited state set from the original state sets that it has received. Lines 19-20 and 41-42 show this operation for forward and backward nodes respectively. Finally, it sends back the newly reached state sets to the respective node. Lines 47 and 49 show this operation for backward nodes and line 30 for forward nodes. The handling of fixed-points/intersections work as follows:

Intersection of fwdSure and bwdSure: Unrealizability of property proven. In this case, the coordinator terminates the computations of all nodes. Lines 24 and 48 in the coordinator's algorithm show this condition.

fwdSure reaches fixed-point: The coordinator does not terminate any node except fwdSure computation (bwdSure computation is more exact than fwdSure computation. Therefore, it may still be possible to prove unrealizability later). The coordinator does nothing special in this case except the current state set ($Curr$) of each forward node automatically represents possible state set since the $Sure$ state set becomes empty.

bwdSure reaches fixed-point: The unrealizability of the property can not be proven (but validity may still be proven later). The coordinator terminates all the node's bwdSure and fwdSure computations, but does not affect the bwdPoss and fwdPoss computations of backward and forward nodes. Lines 24-27 in the Backward node's algorithm and lines 22-23 in the Coordinator's algorithm show this operation.

Intersection of fwdPoss and bwdPoss: Validity can not be proven (but unrealizability may still be proven later). In this case, the coordinator terminates bwdPoss and fwdPoss computations of all nodes, but does not affect the bwdSure and fwdSure computations of backward and forward nodes. Lines 25-28 and 44-47 in the Coordinator's and lines 40-41 in the Backward node's algorithm show this operation.

fwdPoss or bwdPoss reaches fixed-point: Validity of property proven. In this case the coordinator terminates itself by checking whether all the node terminated their computations locally. Line 51 in the coordinator's algorithm checks this condition.

```
runCoordinator(in: T, node_rank)                                          1
  fwdSure = fwdPoss = init_states(q⃗);bwdSure = bwdPoss = bad_states(q⃗);  2
  bwdNodeCounter = 0;                                                     3
  while (true)                                                            4
   receiveAllMsgsFromFwdAndBwdNodes();                                    5
   forall Msgs                                                            6
     if(Msg == bwdSureFixedMsg)                                           7
       bwdNodeCounter =  bwdNodeCounter + 1;                              8
     if(Msg == fwdMsg)                                                    9
       tmpFwdSure = loadFwdSureStspace(Msg);                              10
       tmpFwdPoss = loadFwdPossStspace(Msg);                              11
       // Update complete forward state space                             12
       fwdSure = fwdSure + tmpFwdSure;                                    13
       fwdPoss = fwdPoss + tmpFwdPoss;                                    14
       // Check for intersection with bwd sure                            15
       if(fwdSure ∧ bwdSure ≠ φ) // Unrealizability proven                16
         terminateAllTheNodes();                                          17
         break;                                                           18
       tmpFwdSure = tmpFwdSure - fwdSure;                                 19
       tmpFwdPoss = tmpFwdPoss - fwdPoss;                                 20
       // All the bwd node's sure state set - Fixed-point                 21
       if(bwdNodeCounter == totalNumberOfBwdNodes)                        22
         tmpFwdSure = ⊥;                                                  23
       // Check for intersection with bwd possible                        24
       if(fwdPoss ∧ bwdPoss ≠ φ)                                          25
         // Validity can't be proven but terminate                        26
         // fwdPoss computation.                                          27
         tmpFwdPoss = ⊥;                                                  28
       tmp = (tmpFwdSure) ∗ ē + (tmpFwdPoss) ∗ e;                         29
       sendNewStateSpaceToFwdNode(tmp);                                   30
     else                                                                 31
       tmpBwdSure = loadBwdSureStspace(Msg);                              32
       tmpBwdPoss = loadBwdPossStspace(Msg);                              33
       // Update complete backward state space                            34
       bwdSure = bwdSure + tmpBwdSure;                                    35
       bwdPoss = bwdPoss + tmpBwdPoss;                                    36
       // Check for intersection with fwd sure                            37
       if(fwdSure ∧ bwdSure ≠ φ) // Unrealizability proven                38
         terminateAllTheNodes();                                          39
         break;                                                           40
       tmpBwdSure = tmpBwdSure - bwdSure;                                 41
       tmpBwdPoss = tmpBwdPoss - bwdPoss;                                 42
       // Check for intersection with fwd possible                        43
       if(fwdPoss ∧ bwdPoss ≠ φ)                                          44
         // Validity can't be proven but terminate                        45
         // bwdPoss computation.                                          46
         sendNewStateSpacesToBwdNode(tmpBwdSure, ⊥);                      47
       else                                                               48
         sendNewStateSpacesToBwdNode(tmpBwdSure,tmpBwdPoss);              49
        // Validity of property proven                                    50
     if(checkAllTheNodesTerminatedLocally)                                51
       break;                                                             52
```


Figure 8.4: Coordinator node's computation loop.

8.3 Conclusion

This chapter detailed a novel parallel algorithm for Black Box verification. The approach uses mixed forward and backward traversal mechanisms. some nodes on the cluster machine are assigned with forward state space traversal and others with backward state space traversal. The state space intersection is searched for unrealizability of a Black Box. Until now there exists no parallel algorithm for Black Box verification, this thesis made the first attempt to generate a novel parallel mixed traversal algorithm. The algorithm has several advantages. It alleviates the problem concerning the direction of the traversal. 2. It avoids the intermediate *state explosion* problem by partitioning the state space and assigns the disjoint subsets to both forward and backward nodes. 3. It allows faster verification of properties and this is due to reduced memory requirements for each individual node on the parallel environment.

Chapter 9

Implementation

An idea that is developed and put into action is more important than an idea that exists only as an idea.
- Edward de Bono

This chapter explains all the implementation details. They will be explained in three different sections. First section explains the ingredients needed for the parallelization. Second section explains in detail the methods requisite to transfer the data between network nodes. First, this it explains in detail the available TPO++ functions for both blocking and non-blocking communications. These functions consider basic and standard template library (STL) data types for transmission. Second, it explicates how we can transfer the user defined data type using TPO++ and as an illustration it details the user defined BDD class. Next, it delineates the techniques to transfer a BDD into a sequence of bytes and a sequence of bytes back to a BDD. These byte sequences are used as a mode of transfer for BDDs. Finally, conclusions are given at the end of this chapter. Final section discusses the ingredients needed to construct the grid framework, i.e., the middle-ware, the grid parallel environment, etc.

9.1 Ingredients Needed for the Parallelization

For the communication of one node with another node in a parallel environment the software *Message Passing Interface* (MPI) [127] is used. MPI is a message passing paradigm that is widely used on certain classes of parallel machines, especially those with distributed memory. MPI includes point-to-point message passing and collective (global) operations, all scoped to a user-specified group of processes. During the years it has become the *de facto* standard for parallel computation. MPI also defines C++ bindings for all its interface functions. These bindings merely provide wrappers around MPI constructs and do not fit well into object-oriented concepts. MPI provides no means for transmitting objects.

In order to overcome the lack of integration of object-oriented concepts in

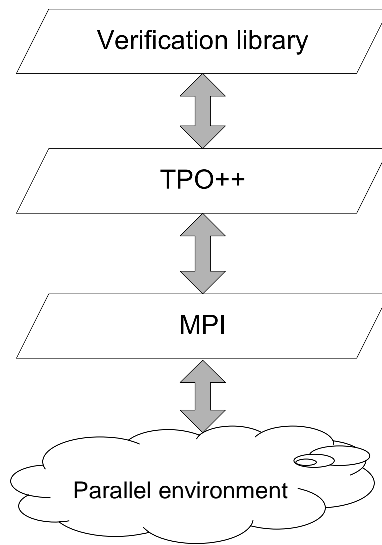

Figure 9.1: Interconnection between different libraries.

MPI, researchers developed TPO++ [128]. TPO++ is an object-oriented message-passing library written in C++ on top of MPI. Its key features are easy transmission of objects, type-safety, MPI-conformity and integration of the C++ Standard Template Library. In order to accomplish BDD transmission, the verification algorithms communicate with TPO++, which eventually contacts with MPI to access the parallel environment. Detailed discussion on asynchronous and synchronous BDD transmission is deferred to chapter 9. Fig. 9.1 shows the layered interconnection between different libraries. The parallel environment in the figure could be any computer cluster.

9.2 Data Transmission

TPO++ uses the initialization function *TPO::init*, which constructs the startup process, which processes the command line arguments and starts the message-passing environment. After initialization, the user can use the global communicator *CommWorld*. *CommWorld* is a predefined object of type *TPO::Communicator* and includes all participating machines. The shut down can be done either automatically on destruction of *CommWorld*, i.e. after the application terminates or explicitly using *TPO::finalize*.

9.2.1 Blocking Communication

TPO++ supports transmitting of predefined C++ data types. When sending a C++ data type, the send call reduces to:

```
int k;
```

```
TPO::CommWorld.send(k,dest_rank,TPO::Tag(110));
```

Where k is the data to send and *dest_rank* represent the recipient rank (where rank represents the identification number of a node). Messages are sent with an accompanying user-defined integer *TPO::Tag*, to assist the receiving process in identifying the message. Thereby, messages can be screened at the receiving end. The message tag can be used optionally. If omitted it defaults to 0. When the function returns, the data has been delivered to the system. On the receiver side, for basic data types receive-call is done as follows:

```
Status status;
status = TPO::CommWorld.recv(k);
```

Where k is the data of type integer. The receiver waits until a message is received from the system. The returned object *status* simplifies the receiver code. The receive methods take two optional parameters, the sender's rank and a message tag for selecting particular messages. If omitted, they default to any sender and any tag respectively.

STL containers can be sent using the same overloaded communicator method. The STL conventions require two iterators specifying begin and end of a container:

```
vector<int> vd;
CommWorld.send(vd.begin(), vd.end(), dest);
```

Similar to the send method for the basic data types, it is optional to use message tags for STL containers. To receive a container, first we need a message specifying the size of the receiving container and later, conforming to this size, the receive method receives the data into a full container. The following example shows the receive code for the STL container:

```
Status st;
st = TPO::CommWorld.recv(cont_size);

vector<int> vd(cont_size);
TPO::CommWorld.recv(vd.begin(), vd.end());
```

9.2.1.1 Non-Blocking Communication

For asynchronous communication, the method *isend* provided by the communicator class is used. The method returns an object of class *Request*. We can wait for the completion of the asynchronous communication using a *wait* method. The following code illustrates the syntax for the *isend* method.

```
Request rt;
rt = TPO::CommWorld.isend(data, dest, TPO::Tag(515));

rt->start();
rt->wait();
delete rt;
```

On the other hand, the receiver uses the method *irecv* to receive the message. Before receiving the actual message the receiver first checks the incoming message using *iprobe* whether it has the relevant characteristics, i.e. what the receiver is looking for. The code for the asynchronous receive looks like as follows:

```
Status st;
Request rt;
st = TPO::CommWorld.iprobe(AnySource,tag_value);
if(st.isValid()) {
  rt = TPO::CommWorld.irecv(value);
  rt->start();
  rt->wait();
}
```

Similar to blocking communication, the methods *isend* and *irecv* are overloaded for asynchronous communication of STL containers.

All the parallel verification approaches described in this thesis use basic data types (C++) for simple communication. For example, if any of the nodes uncovers the target state then it sends a simple integer message to the coordinator, which eventually terminates other nodes computation by sending again a simple integer message. In a heterogeneous network environment, the master node alone is responsible for preprocessing the system model and the property. In such cases, it converts the model and the property into a BDD form and stores them in STL containers. Later it sends these containers to all other network nodes. Therefore, the synchronous and asynchronous communication of simple and STL container data types is frequently used in all parallel approaches.

9.2.2 User-defined BDD Data Type

For the transmission of BDD objects, we have to define the marshalling methods *serialize* and *deserialize*, as part of the class definition. Marshalling is the process of transforming the memory representation of an object to a data format suitable for transmission. The only argument of the marshalling methods is a *Message_data* object, provided by the TPO++ library, used to marshal the object's member data. The *Message_data* class provides *insert* and *extract* methods, whose arguments are all kinds of transmittable types. In a serialize method, *insert* is called repeatedly for every member to prepare the object for transmission. The *Messge_data* object

does not copy the data, but records its memory layout for later transmission. The *extract* method in the deserialize method has to unpack the received message to user-provided memory locations. The code that follows after this paragraph implements the user defined class *BDD* and it's marshalling methods.

```
class BDD {
 public:
  // Header part
  int nnodes; // Number of nodes that are present the BDD
  int nvars;  // Total number of variables of the BDD manager
  int nsuppvars; // Support variables of the BDD
  int rootid; // Index of the BDD root
  vector<int> ids; // Variable IDs
  int ids_size; // Size of the ids vector
  vector<int> permids; // Variable permutations
  int permids_size; // Size of the permids vector

  // Binary
   // List of BDD nodes represented in binary format
  vector<unsigned char> bdata;
  int bdata_size; // Size of the bdata vector

  void serialize(TPO::Message_data &m) const {
   m.insert(nnodes);
   m.insert(nvars);
   m.insert(nsuppvars);
   m.insert(rootid);
   m.insert(ids_size);
   m.insert(ids.begin(),ids.end());
   m.insert(permids_size);
   m.insert(permids.begin(),permids.end());
   m.insert(bdata_size);
   m.insert(bdata.begin(),bdata.end());
  }

  void deserialize(TPO::Message_data &m) {
   m.extract(nnodes);
   m.extract(nvars);
   m.extract(nsuppvars);
   m.extract(rootid);
   m.extract(ids_size);
   ids.resize(ids_size);
   m.extract(ids.begin(),ids.end());
   m.extract(permids_size);
   permids.resize(permids_size);
   m.extract(permids.begin(),permids.end());
```

```
    m.extract(bdata_size);
    bdata.resize(bdata_size);
    m.extract(bdata.begin(),bdata.end());
  }
};
TPO_MARSHALL_DYNAMIC(BDD)
```

The arguments (transmittable types for BDD) used by the *insert* and *extract* methods will be explained later in this chapter. TPO++ supports transmission of user defined data objects only for synchronous methods. But for asynchronous communication, we need to pulverize the data object into a byte stream and store it in containers for later transmission.

9.3 BDD Transmission

The ideas for transferring BDDs are based on the DDDMP package of the CUDD library [58]. Processes periodically exchange BDDs during symbolic simulation and the transmission of BDDs is done in binary form. Two utility functions are used for BDD transmission. *convertTheBddsToBytes* translates a set of BDDs into a sequence of bytes and *constructTheOriginalBDDs* translates the sequence of bytes back to BDDs after the sequence has been transferred. The purpose of *constructTheOriginalBDDs* is to serialize the BDD structure in order for it to be suitable for raw buffer transfers. *convertTheBDDsToBytes* traverses the nodes of each BDD f in Depth First Search (DFS) order. It creates the corresponding sequence from the leaves and follows towards upwards. *constructTheOriginalBDDs* receives byte sequences from start to end and then creates the corresponding BDD nodes one by one as it traverses the data. Shannon expansion is used to create the BDD node from the byte sequences.

9.3.1 BDDs to Sequence of Bytes Conversion

The BDD that needs to be sent over the network is decomposed into two types of data: 1. a *header* in text format; 2. a list of nodes in binary format. The header part will be sent in textual format, whose size is by far dominated by the node list in case of large BDDs (several thousands of BDD nodes). The node list is converted into a binary form. The header part is common for all the BDDs except for each BDD's rootid.

9.3.1.1 Header Data

The sub functions listed under the comment *header part* in Fig. 9.2 construct the header part of the data. For sake of generality and because of dynamic variable ordering both variable IDs and permutations (permids) are included in the

header part. The variable ID holds the name of the variable that labels the node. The variable ID of a variable is a permanent attribute that reflects the order of creation. ID 0 corresponds to the variable created first. The permutation of the i-th variable ID is the relative position of the variable in the ordering. Next, only the variables in the true support of the stored BDDs are included. Information on variables (IDs, permutations) is sorted by IDs, and they are restricted to the true support of the transmitted BDDs. In addition the header part also includes the number of nodes that are present in all BDDs (nnodes), the total number of variables of the BDD manager (nVars) and the indexes of all BDD roots, which allows complemented edges (rootids). All these attributes are shown in the header part of the code in section 9.2.2. The BDD class defined in this code is suitable for a single BDD transmission. However, the code is easily extendable for multiple BDDs by creating a STL container for *rootids*. All other remaining attributes remain the same.

9.3.1.2 Binary Data

The list of nodes of all BDDs are represented in binary format. In binary mode nodes are represented as a sequence of bytes, representing variable index, Then-index, and Else-index in an optimized way. Function *createByteSequence* (line 13) in Fig. 9.2 converts all the BDDs into binary form. It traverses the nodes of each BDD in recursive manner and computes the binary data. The optimization is done under the fact that most of the times the data will be sent redundantly. The redundant data here means, the indexes of the nodes. For example nodes having one of its child nodes as terminal node. In such cases we repeatedly send the index (in integer) representing the terminal node for all the parent nodes unnecessarily. The function *fillLeafNodes* (line 29) in Fig. 9.2 avoids such redundant data for transmission. It represents the integer indexes in absolute or relative mode, where relative means an offset with respect to a Then/Else node information [58]. It selects the best between absolute and relative representations. After finalizing the mode of transmission, the method efficiently converts the values of the indexes, represented in integers, into byte sequences.

9.3.2 BDDs Construction From Sequence of Bytes

The method *constructTheOriginalBDDs* first receives header information and then binary data from the sender. By keeping the header information as a reference, it constructs the original BDDs using the received binary data. The variable match between the BDD manager of the receiver and the received data is optionally based on IDs and permids. In case the mode matches with variable IDs, a BDD is constructed BDD by keeping variable IDs unchanged. As a consequence, the BDD is constructed regardless of the variable order of the reading manager. If the mode matches with variable permids then it allows the variable match according to the position in the ordering (retrieved by array of permutations received

```
// S is the set of BDDs                                          1
convertTheBDDsToBytes(S) {                                       2
  // Creates the header part                                     3
  createNumberOfNodes();                                         4
  createNumberOfVars();                                          5
  createNumberOfTrueSuppVars();                                  6
  createVarIds();                                                7
  createPermIds();                                               8
  createRootIds();                                               9
                                                                 10
  // Create the binary data                                      11
  forAll f ∈ S                                                   12
    createByteSequence (f→rootnode);                             13
}                                                                14
                                                                 15
// f is root node of the BDD                                     16
createByteSequence(f) {                                          17
  // Recursive call for Then edge                                18
  T = f->Then;                                                   19
  createByteSequence(T);                                         20
                                                                 21
  // Recursive call for Else edge                                22
  E = f->Else;                                                   23
  createByteSequence(E);                                         24
                                                                 25
  vf = f->index;  // Variable of the current node                26
  vT = T->index;  // Variable of the Then edge                   27
  vE = E->index;  // Variable of the Else edge                   28
  fillLeafNodes(vf, vT, vE, f, T, E);                            29
}                                                                30
```

Figure 9.2: BDDs to sequence of bytes conversion

from header part). As a consequence, the constructed BDD keeps the ordering as BDD that has been transmitted. Fig. 9.3 delineates the algorithm to construct the original set of BDDs using received header and binary data. The subfunctions described in lines 4-9 receives the header information. The subfunction *receive-LeafNodes* receives regularly a set of bytes and constructs the present node and its Then node and Else node (f, T and E). In addition it computes the variable of the node, its Then variable and its Else variable (vf, vT and vE). With this information, *constructBdd* creates the BDD and points indexes to their respective nodes. The code in lines 17-21 finally creates the original set of BDDs.

For synchronous transmission of BDDs the BDD class described in section 9.2.2 is used. The described specification is just for one BDD but it can be extended for more than one BDD by including *rootids* STL container. All the attributes for this class are filled by calling the method *convertTheBDDsToBytes* before the transmission. TPO++ does not support the asynchronous transmission of

```
// S is the set of BDDs                                              1
constructTheOriginalBDDs(out: S) {                                   2
  // Receives the header part                                        3
  receiveNumberOfNodes();                                            4
  receiveNumberOfVars();                                             5
  receiveNumberOfTrueSuppVars();                                     6
  receiveVarIds();                                                   7
  receivePermIds();                                                  8
  receiveRootIds();                                                  9
                                                                     10
  while iterator <= nnodes                                           11
    receiveLeafNodes(out: vf, vT, vE, f, T, E);                      12
    pnodes[iterator] = constructBdd(in: vf, vT, vE, f, T, E);        13
    iterator++;                                                      14
                                                                     15
  while iterator <= nRoots                                           16
    id = rootids[iterator];                                          17
    f = pnodes[id]                                                   18
    S[iterator] = f;                                                 19
    iterator++;                                                      20
}                                                                    21
```

Figure 9.3: Construction of BDDs from sequence of bytes

user defined objects. Therefore, even the *header* data must be converted the data into binary form and stored in a STL container for transmission.

9.4 The Grid Framework

The grid approach in this thesis uses the grid middle-ware UNICORE (UNiform Interface to COmputer REsources) [129], funded by the German Ministry for Education and Research (BMBF) with Sun Grid Engine (SGE) [130] as batch queuing system. As parallel environment it uses MPICH2 [131] and TPO++. The detailed explanation on methods to transfer the data using TPO++ is deferred to chapter 9.

9.4.1 UNICORE

Fig. 9.4 presents the UNICORE [129] system components and their interaction. For the job submission to any platform of a UNICORE grid the Job Preparation Agent (JPA) is used. The user can also monitor and control jobs through the Job Monitor/Controller (JMC). An Abstract Job Object (AJO) is constructed by the JPA with the definition of a job. With this AJO the JPA contacts the UNICORE Gateway at a selected site.

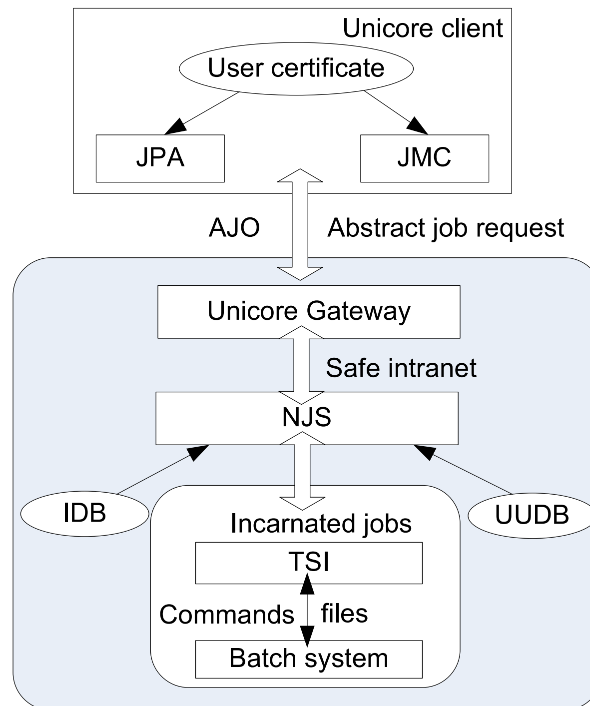

Figure 9.4: UNICORE architecture

The Gateway is a small java-application running at the target site for authenticating users by their X.509 certificate and providing information about available resources. It delegates an AJO to the Network Job Supervisor (NJS) server.

Each target system or cluster of systems is controlled by one NJS which provides resource information from the Incarnation Database (IDB) to the Gateway. The NJS checks the authorization of the user to use the requested resources from the UNICORE User Database (UUDB). The NJS translates the abstract tasks into real batch jobs using the IDB and executes them on the batch subsystem using the Target System Interface (TSI).

By using the graphical interface of UNICORE, the user is able to prepare and modify structured jobs. The UNICORE client is a Java-2 application on a local UNIX/Linux Workstation or Windows PC. This client is used to submit jobs with UNICORE to the grid. As a batch queuing system SGE [130] with parallel environment based on MPICH2 is used.

An overview of the system is shown in Fig. 9.5. The approach uses only one node as master node for all services which means that SGE-Master and all UNICORE services (Gateway, NJS and TSI) will run on one machine. This is possible due to the small size of our system and makes it easier to administrate and change the setup. On each of the node's SGE-Client, MPICH2 and the bounded property checking library have to be installed.

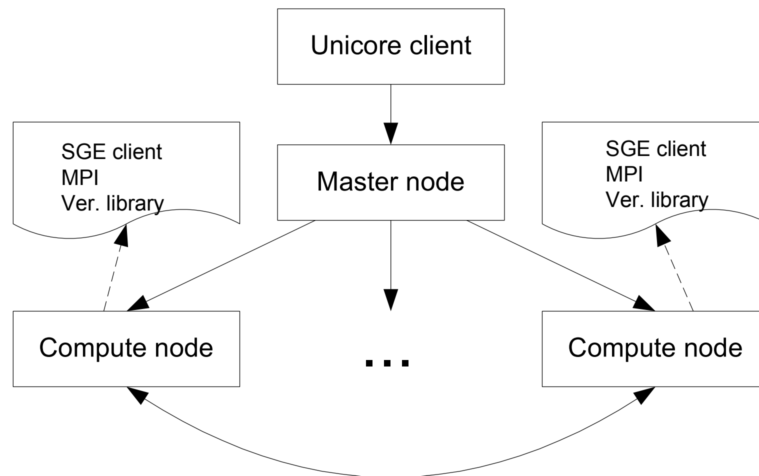

Figure 9.5: Grid approach.

9.5 Conclusion

This chapter presented all the implementation details. First, it has started with the ingredients needed for the parallelization. Then, it has provided the techniques to transfer data between different network nodes using TPO++. TPO++ supports various methods to efficiently transfer the basic data types, STL container and user defined data types. TPO++ supports both synchronous and asynchronous data transfers. However, they are confined only to basic and STL container data types. TPO++ does not support asynchronous communication on user defined data types. This chapter described the ways to communicate BDD functions. BDDs can be transmitted using a binary form by avoiding redundant information. Thereby, an efficient data transmission between network nodes is achieved. Finally, this chapter detailed the grid framework used by the thesis.

Chapter 10

Experimental Results

Theory looks well on paper, but does not amount to anything without practice.
- Henry Wheeler Shaw

Six sets of experiments were performed in this thesis. The first set is to show problems using the sequential approach. The second set of experiments present the results of the basic parallelization approach and describe it's problems with the state overlap. The third set shows the results using the *dynamic overlap reduction* method. The fourth set depicts the results using the hybrid method. This set is divided into four subsets. The first and second subsets are to show that the hybrid approach benefits over the *dynamic overlap reduction* method. The third subset of experiments is to compare the scalability of the hybrid approach using different number of *windows*. The last subset of experiments is to compare the dynamic and the static variable reordering under the hybrid approach. The fifth set of experiments compare a state-of-the-art verification algorithms with the *dynamic overlap reduction* and the hybrid approaches. Finally, one set of experiments was performed with the grid and it is divided into two subsets. The first subset is to show the results with the grid based distributive approach and the second subset is to compare the results of both the grid and the cluster based distributive approaches. All the experiments were stopped after one hour.

10.1 Details of Experiments

10.1.1 Parallel Environment

The experiments were performed on the Kepler cluster at the University of Tübingen. This cluster contains 98 computing nodes, each consisting of a dual 650 MHz Pentium-III processor with 1 Gb of shared memory (512 Mb for each processor). The cluster uses BX motherboards and each has a 32 bit/33 MHz PCI bus. The cluster uses the Linux operating system. The nodes are connected with two networks. Ethernet, 100 Mbit, is used to start and boot nodes, while Myrinet is for

the direct exchange of messages between nodes. The cluster uses a *Score* management software. Score gives the applications direct access to the network interface. In addition, the cluster provides a locking facility that prevents other users from using cluster hosts through the *MessageBoard* server. We can browse the current status of a cluster with the *msgb* client program. The $msgb$ is an application written in Tcl/Tk to display the current status on an X-window display. The users can lock hosts via the *MessageBoard* server.

10.1.2 Implementation Details

For communication among network nodes the software MPI (Message Passing Interface) version 1.2.7 is used. TPO++ version 0.4 is used for transmission of objects and STL data types. A parallel verification library has been written on top of the bounded property checker *SymC* in C++.

10.1.3 Checked Designs

The experiments were conducted on some of the circuits from the ISCAS89 sequential benchmarks [132], IBM benchmarks (batch circuits) [133] and a model of a holonic production system (Holon). The holonic material transport system consists of an input station, three machines, an output station and three automatic transport vehicles, the so-called *holons*. Two of the three machines are for work-piece processing, one is for cleaning. All holons are identical. More details on holonic production system can be found in [134].

10.1.4 Checked Properties

For the ISCAS89 circuits there is no available information regarding their behavior. Therefore the reachability properties of states with a high hamming distance (HHD) from the initial state [82] (see formula A) along with properties from [117] were checked. The hamming distance between two points p_1 and p_2 in the Boolean space, is defined by the number of bits which assume a different value in p_1 and p_2. For example a model with 5 state variables (d_0,d_1,d_2,d_3,d_4) and initial state S = (0,0,0,0,0) then we have one state with hamming distance 5 from S, i.e., (1,1,1,1,1), 6 states at hamming distance $>=$ 4, i.e., (1,1,1,1,1), (0,1,1,1,1),...,(1,1,1,1,0) and so on. In the holonic production system, the consumption of a work piece was checked as property (see formula B). For the IBM benchmarks this thesis checked for reachability of a set of states either by HHD or by states that satisfy a certain condition (see formula C). The formula in C depicts that always for the certain time steps if the condition on the left hand (m30.xx29) is true then eventually the right hand condition should hold in certain time steps. The thesis used both universal and existential properties in the experiments in order to highlight

the approach's ability in handling both full validation and fast falsification scenarios. A sample of our FLTL properties is as follows.

$$\mathsf{G}[b]\ !(s1423.G22\ \&\ s1423.G23\ \&\ \ldots\ \&\ s1423.G95)\ \ (A)$$
$$\mathsf{F}[b]\ OutBuffer.s_consume\ \ (B)$$
$$\mathsf{G}[b]\ (m30.xx29\ \rightarrow \mathsf{F}[d]\ (m30.xx87))\ \ (C)$$

where b, $d > 0$ are explicit time bounds.

10.2 Verification Using a Single Processor

First, I ran some of the larger designs in sequential *SymC* with all relevant optimizations switched on. The sequential algorithm splits the state set repeatedly upon reaching the threshold, whereas the parallel version does it only during state set distribution. The results are available in Fig. 10.1. For all the designs the sequential algorithm cannot complete traversal due to memory overflow or time out problems. The first column in Fig. 10.1 lists the design name. The second column gives the number of flip-flops used in the design. Column three denotes the splitting threshold. The fourth column shows the influence used for *Minimal overlap* splitting. The l_1 and l_2 in Φ_{l_1,l_2} denote the lookahead and lookback influence factors. The *Minimal overlap* algorithm uses these factors for categorizing highly influenced variables and select the best variable for partitioning. The algorithm was detailed in section 5.1.1. The fifth and sixth columns list the time bound specified in the property and maximum peak node count in millions, respectively. The last column shows the overall verification time in seconds. #n or *n denote memory overflow or time out at step n.

Design	FFs	Threshold	Φ_{l_1,l_2}	Time bound	Peak node count	v_t in sec.
s4863	104	20000	$\Phi_{1,1}$	5	4.48	#2
s1512	57	50000	$\Phi_{2,1}$	100	3.80	*80
s1423$_{p1}$	74	50000	$\Phi_{1,0}$	-	13.55	#11
s1423$_{p3}$	74	50000	$\Phi_{1,0}$	12	14.27	*11
04-batch	256	50000	$\Phi_{2,1}$	20	2.59	*15

Figure 10.1: Results for fully optimized sequential *SymC*.

10.3 Distributed Verification

10.3.1 Basic Parallelization

Fig. 10.2 shows the results of basic parallelization [118] approach using different partitioning algorithms. The first column indicates the design names. The second

column gives the number of flip flops used in the design. The column *Threshold* indicates the splitting threshold. The fourth column represents the time bound specified in a property. The fifth column specifies the used state-of-the-art partitioning algorithm. The algorithm *Variable disjunctive decomposition* select the best variable by exploring each variable's positive and negative co-factors (described in section 3.3.1). The algorithm *Eager decomposition* eagerly select the best variable using the cost function described in section 3.3.3. The algorithm *Equal decomposition* (altered version of the slicing heuristic from [97]) considers both reduction and redundancy factors into account for selecting the best splitting variable (described in section 3.3.2). Finally, the algorithm *Minimal overlap* statically analyzes the transition relation of the design and select the best influenced variable for splitting (described in section 5.1.1). Two subcolumns in column sixth specify the verification times in seconds taken by basic approach using 16 and 32 number of nodes. Time out is indicated by * followed by the maximum number of steps n taken by any node. #n denotes memory overflow at step n.

Design	FFs	Threshold	T_b	Partitioning Alg.	v_t in sec.	
					16	32
s1512	57	50000	100	VarDisjDecomp	2146.25	2103.45
				MinimalOverlap	**2135.46**	**2083.24**
				EagerDecomp	2772.98	2750.55
				EqualDecomp	2450.75	2363.23
s4863	104	20000	5	VarDisjDecomp	952.22	813.69
				MinimalOverlap	**714.49**	**579.86**
				EagerDecomp	#2	#2
				EqualDecomp	967.95	851.39
$s1423_{p4}$	74	50000	10	VarDisjDecomp	**215.12**	**203.69**
				MinimalOverlap	243.79	233.02
				EagerDecomp	245.73	224.2
				EqualDecomp	303.73	293.88
s1269	37	5000	10	VarDisjDecomp	**30.34**	**21.74**
				MinimalOverlap	41.79	22.88
				EagerDecomp	36.01	31.67
				EqualDecomp	32.32	22.58
Holon	118	50000	300	VarDisjDecomp	*223	204.06
				MinimalOverlap	*225	**197.38**
				EagerDecomp	*198	*228
				EqualDecomp	*213	208.5

Figure 10.2: Basic parallelization results.

10.3.1.1 Discussion

The runtimes marked with bold text denote the partitioning algorithm yielding the best result. For the ISCAS89 designs *s1512* and *s4863*, in comparison with

sequential approach, where it was not possible to obtain any results, the parallel version completes the verification and obtains the end results. For all the designs (including *Holon* for which the approach using 16 nodes cannot complete traversal due to time out problems) not much of speedup is achieved due to the huge state overlap between the network nodes. For the designs *s1512* and *Holon* Fig. 10.3 compares the resulting overlap when applying different partitioning heuristics on 8 processors. It shows how the overlap evolves over time. The x axis represents the number of time steps after partitioning and y axis represents the normalized overlap O_k (defined in section 4.1.3) among network nodes. Therefore, the overlap must be treated using *dynamic overlap reduction* method. It can be inferred from Fig. 10.2 and Fig. 10.3 that the partitioning algorithm *Minimal overlap* relatively performs better compared to other algorithms but it cannot completely remove the overlap. For design *s1512*, it has been observed that after 20 time steps of state set distribution, there is almost a 100% state overlap among network nodes.

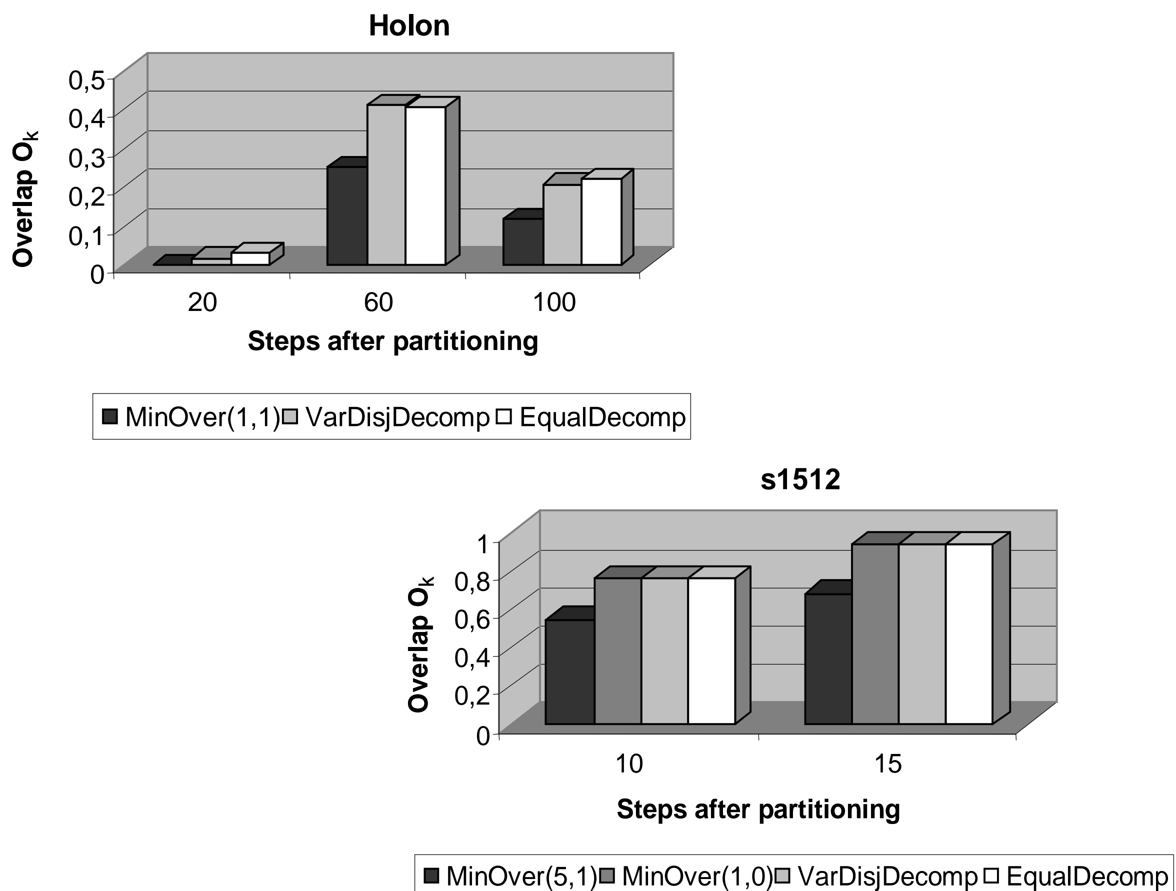

Figure 10.3: Overlap comparison using different partitioning heuritics.

10.3.2 Dynamic Overlap Reduction

Fig. 10.4 shows the results of the distributed approach with *dynamic overlap reduction* [122, 112]. The dynamic overlap reduction algorithm was described in section 5.2. The approach used 32 processors dedicated to the checking algorithm and one processor acting as the coordinator. In these experiments, dynamic overlap reduction is applied throughout the verification process repeatedly every p steps. The first column in Fig. 10.4 indicates the design and the splitting threshold. The second column shows the time period p at which overlap reduction is performed and the influence used in *Minimal overlap*. Both p and Φ_{l_1,l_2} can be determined by the user. The third column lists the time bound specified in the property. Column four lists the time taken by the sequential part and the time step at which the parallel stage starts. Column five shows the maximum peak node count of all the nodes in millions. The last column lists the overall verification time in seconds.

Design	p (Φ_{l_1,l_2})	Time bound	S_t in sec. (step)	P_{nc} in milli.	v_t in sec.
s4863 20000	1 ($\Phi_{1,1}$)	5	1.6 (1)	8.39	587.73
s1512 50000	2 ($\Phi_{2,1}$)	100	108.75 (33)	2.41	508.21
s1423$_{p1}$ 50000	1 ($\Phi_{2,1}$)	-	75.7 (8)	13.2	748
s1423$_{p2}$ 50000	1 ($\Phi_{1,1}$)	-	76.6(8)	1.87	114.25
s1423$_{p3}$ 50000	2 ($\Phi_{2,1}$)	12	75.5 (8)	5.06	1133
Holon 50000	100 ($\Phi_{1,1}$)	1000	86.22 (132)	1.6	230.08
04-batch 50000	2 ($\Phi_{2,1}$)	20	99 (13)	2.3	955

Figure 10.4: Results of the distributed algorithm with dynamic overlap reduction.

10.3.2.1 Discussion

The core algorithm in the parallel approach *dynamic overlap reduction* distributes partitions of the state set to computation nodes after reaching a threshold size. The nodes proceed with their image computation asynchronously. During the traversal each node periodically removes its overlap.

For designs *s1512* and *Holon* clearly Fig. 10.4 shows the advantage of *dynamic overlap reduction* when compared with basic parallelization using state-of-the-art static overlap reduction (*Minimal overlap*) algorithm (depicted in Fig. 10.2). For design *s1512* the *dynamic overlap reduction* outperforms the basic parallelization method by removing the state overlap at every 2 intervals of time steps. For

design *Holon*, the *dynamic overlap reduction* technique has reached a larger time bound (1000) by taking relatively equal amount of verification time, where the basic parallelization using static reduction algorithm could reach 300 time steps (depicted in Fig. 10.2).

For design *s1423* three properties $p1$, $p2$ and $p3$ were considered. Both $p1$ and $p2$ are from [117] and pure LTL properties, hence there is no time bound specified in the property. For both designs *s1423* and *04-batch dynamic overlap reduction* outperforms partitioned sequential approach (which uses *Minimal overlap* algorithm for partitioning) and obtains the end result within time limit.

For designs *s4863*, *s1512* and $s1423_{p1}$, Fig. 10.5 shows the decrease in verification times and memory consumptions with respect to the reduction time steps. After distributing the disjoint state space to each node (beginning of parallel phase), a reduction time step specify at every regular intervals (k) of time steps the overlap removal operation is performed. Fig. 10.5 clearly shows that the delaying in overlap removal reduces the verification system performance. The left hand side of figure shows the time performance and right hand side shows the memory performance (maximum peak node count of all the nodes in millions). The x axis in both figures show the reduction time steps at which the overlap removal is performed. This figure clearly shows the importance of *dynamic overlap reduction*, i.e., delaying in overlap removal can degrade the system performance.

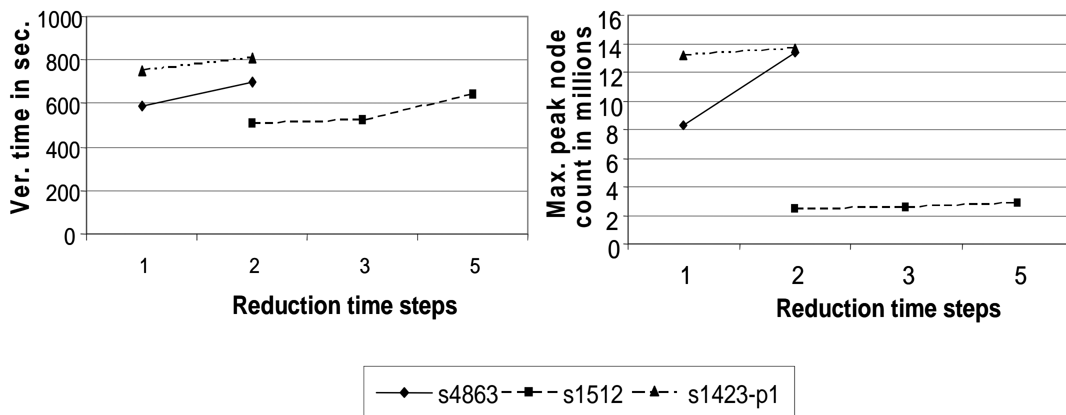

Figure 10.5: Comparison of verification times and memory consumptions with respect to reduction time steps.

Fig. 10.6 depicts the natural load balance graph for the circuit *s1512* with reduction period 2. Only four nodes are shown for clear visibility of the graph. The x axis represents the reduction time points and y axis represents the node's arrival order with respect to reduction time point. As described in section 5.2.1, the node that reaches the reduction time point very late (high node arrival order) can get its overlap removed with respect to the nodes that have already visited the time step. So this late node has not many states in common with the other nodes at this reduction step and therefore contains the smallest subset. This effect in turn eases the node's computation, enabling this node to reach the upcoming reduction time point faster. The load balancing effect can be seen very well when nodes

0 and 24 swap their arrival order during execution. In general, the nodes that deal with a large state space at one time point will be later assigned a small state space and vice versa. Therefore, a natural side effect of dynamic load balcnce among network nodes is achieved.

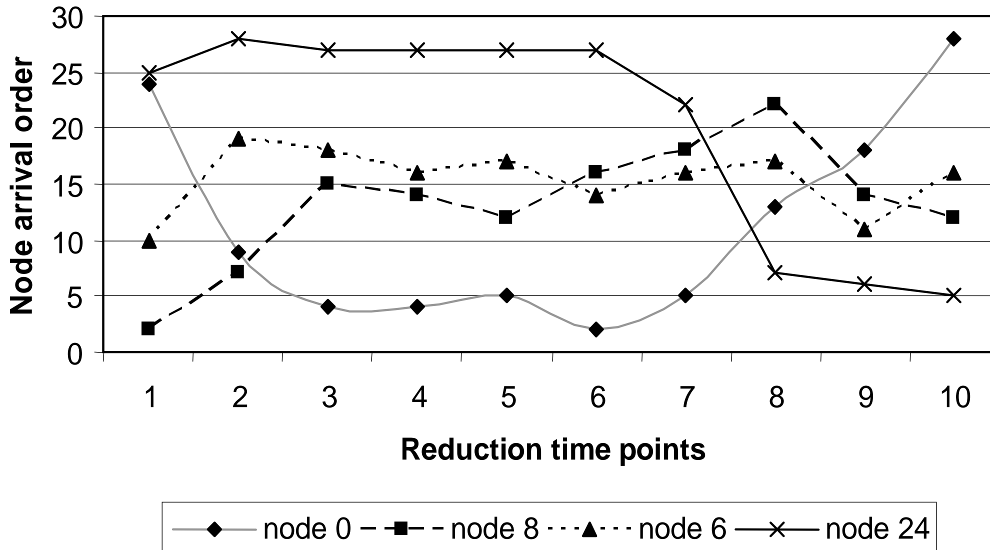

Figure 10.6: Arrival order of nodes at reduction points showing load balancing between the nodes for the design s1512.

10.3.3 Hybrid vs Dynamic Overlap Reduction

Fig. 10.7, Fig. 10.8, Fig. 10.9 and Fig. 10.10 show the comparison results of the *hybrid* distributed approach [123] and *dynamic overlap reduction* distributed approach. The hybrid algorithm was described in section 6.2 and dynamic overlap reduction was described in section 5.2. In contrast to the total number of nodes used by the dynamic overlap reduction method (as indicated in previous section 10.3.2), the hybrid approach uses 32 processors acting as *windows*, one processor as *static helper* and one processor as *coordinator*. In all these experiments, the restriction or removal is applied throughout the verification process repeatedly at every p steps. For fair comparison of the approaches, the same p and threshold limit are utilized and experiments were also performed with dynamic variable reordering disabled in the BDD package. Fig. 10.7 and Fig. 10.8 show comparison results for falsified properties. Time out and memory overflow problems are depicted explicitely in the figures.

10.3.3.1 Discussion

All circuits in Fig. 10.7 and Fig. 10.8 were checked for fast falsification and the error states were found by the *windows* in hybrid approach.

For the designs *s4863* and *29-batch* the results using both approaches are almost equal. This is due to the fact that error states are located close to the splitting point. If bugs are found before reaching the restriction or reduction time point then both approaches are equal.

For design *s1423* two properties $p1$ and $p2$ were considered. The $p1$ is from [117] and a pure LTL property, hence there is no time bound specified in the property. In comparison to dynamic overlap reduction, the hybrid method finds an error state significantly faster. The property $p2$ is a full validation property.

For deeper errors, the method using dynamic overlap reduction takes more memory and verification time compared to the hybrid approach. For designs *18- ,19-* and *30-batch* the dynamic overlap reduction suffers time out problems whereas hybrid method finishes the verification by using less memory.

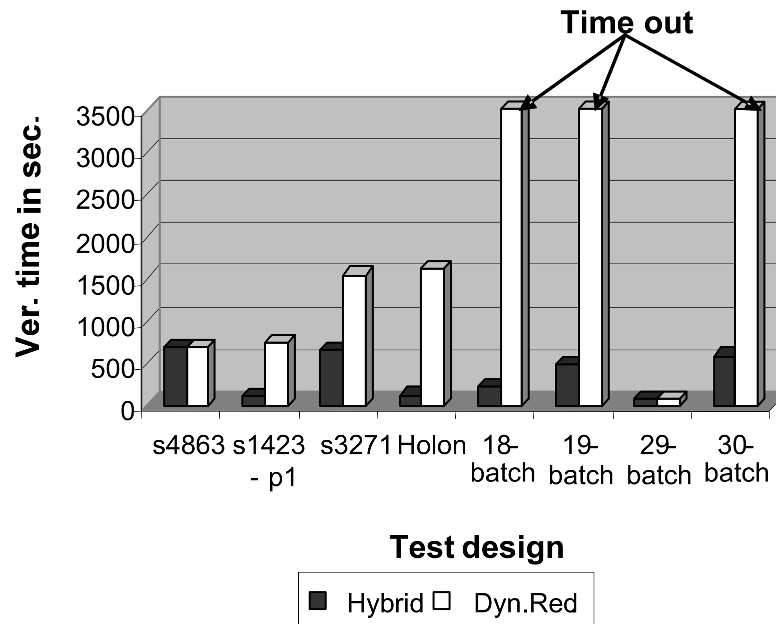

Figure 10.7: Comparison of verification times using hybrid and dynamic overlap reduction approaches for falsified properties.

The designs in Fig. 10.9 and Fig. 10.10 are valid designs with respect to the properties specified. Time out and memory overflow problems are denoted explicitely. For all these designs, hybrid method required less verification time and memory compared to dynamic overlap reduction method except for the design $s1423_{p2}$ due to memory overflow. The *windows* finished their verification job while the static *helper* was computing the image of cross over states. The memory overflow problem occured due to a huge cross over state set. Hence, the *windows* entered the waiting mode in order to get extra work from the *coordinator*.

Fig. 10.11 depicts the nodes work utilization graph for the circuit *04-batch* with restriction period 1. For a clear visibility of the graph only 18 nodes have been chosen, where 16 are *windows* with the ranks shown in x axis, one *static helper* with the rank 32 and one *coordinator* with the rank 33, where rank is a number

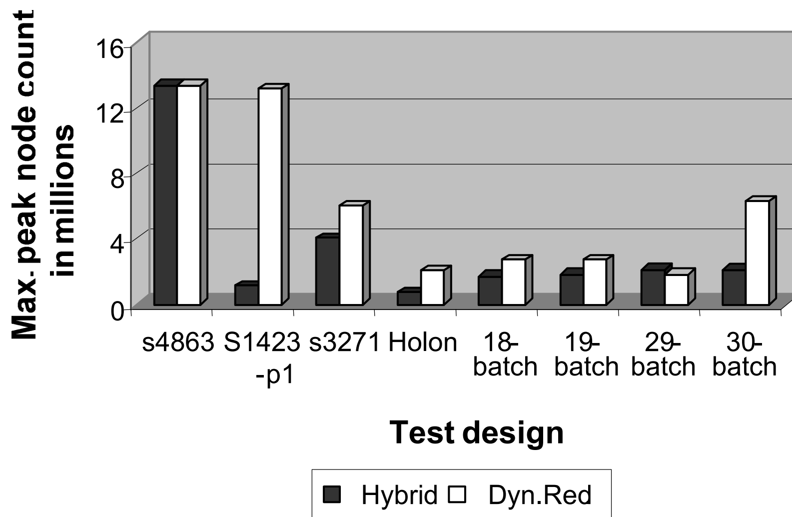

Figure 10.8: Comparison of memory consumptions using hybrid and dynamic overlap reduction approaches for falsified properties.

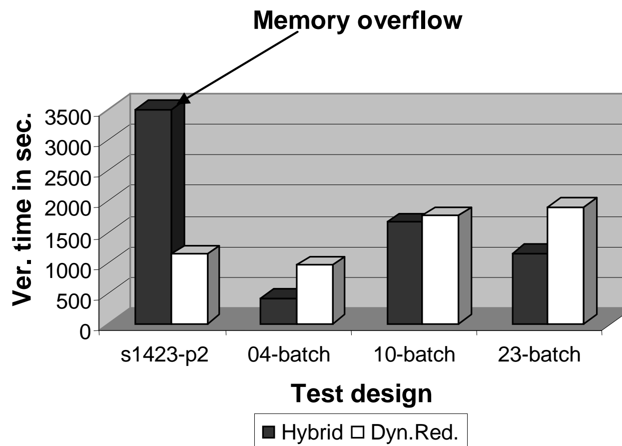

Figure 10.9: Comparison of verification times using hybrid and dynamic overlap reduction approaches for full validation properties.

identifying every node. Each bar in the graph shows each node's sequential time, windowing time and helping time. The *static helper* does not contain window time and the *coordinator* does not contain window time but organizing time as an exception. It is visible in the graph that the helping times of nodes are different. This is decided by the *coordinator* depending on the node work load. Depending on the overall work load the *coordinator* dynamically decides whether to reallocate the work or to terminate the nodes. The node with the rank 1 comparatively took more verification time and the last one to terminate the computation as far as normal nodes are concerned. This is due to more time taken by the final image computation step. Trivially, the *coordinator* has to be the last one to terminate.

Fig. 10.12 depicts the load balance among helper nodes 3, 9, 12, 20 and 32 for the circuit *04-batch* with reduction period 1. It is called *reduction* period since

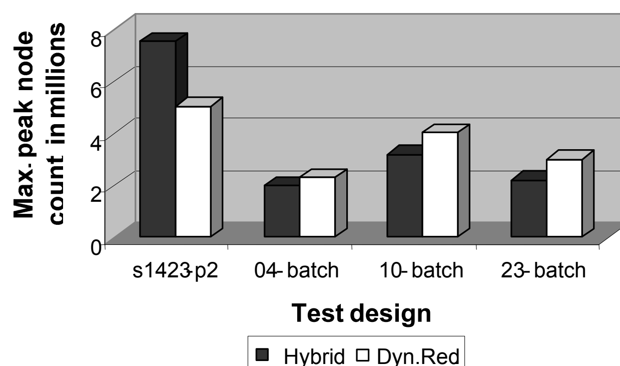

Figure 10.10: Comparison of memory consumptions using hybrid and dynamic overlap reduction approaches for full validation properties.

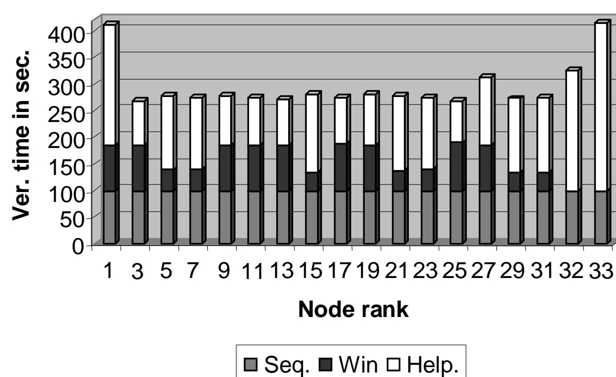

Figure 10.11: Nodes work utilization graph for the design 04-batch.

the *helper* nodes get their overlap removed with respect to all other *helper* and *window* nodes. The node with rank 32 is a static *helper*, the remaining nodes were *windows* in the beginning and dynamically became *helpers*. Only five *helper* nodes are shown for clear visibility. The faster the node arrives, the less the work load and therefore the load balancing effect in the graph can be seen very well as nodes swap their arrival order during execution.

Fig. 10.13 shows the speedup comparison results of 32, 16 and 8 *windows* using the hybrid approach. The designs *s4863*, *23-batch* and *30-batch* with 8 *windows* suffer from memory overflow (indicated by dashed lines) and time out problem (indicated by bars touching the 3500 sec. mark).

10.3.4 Variable Ordering Comparison

Fig. 10.14 shows the comparison results of static and dynamic variable reordering using the hybrid approach. The algorithm *lazy sift* [57],is used for dynamic variable reordering. Same threshold (BDD node count) is used for both the approaches.

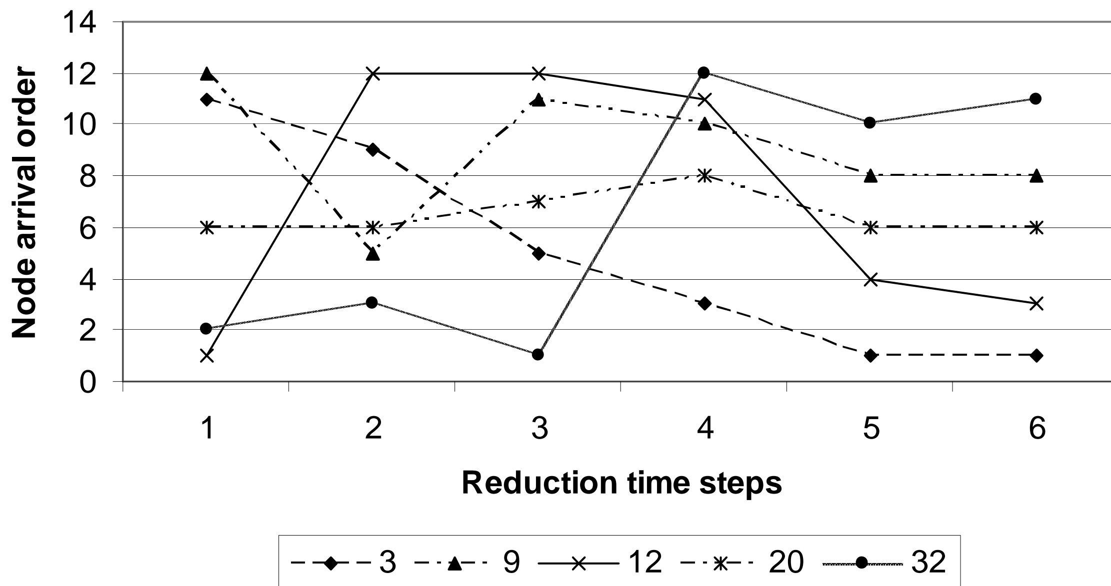

Figure 10.12: Arrival order of helper nodes at reduction points showing load balancing between the helper nodes.

10.3.4.1 Discussion

For the design $s4863$ the method using dynamic variable reordering took less memory and verification time compared to static ordering. The node that computes the sequential phase detected the error state in the dynamic method. This circuit shows the real advantage using dynamic variable reordering. However, for the designs *Holon*, *19-batch* and *30-batch* the method static ordering performs better compared to the dynamic approach. For the designs *19-* and *30-batch* the the approach using dynamic variable ordering did not reach the threshold limit and spends most of the time on searching for the best suitable order. A similar effects have been noticed also in sequential mode of experiments with dynamic variable reordering.

10.4 State-of-the-art Comparison

For comparison with the state-of-the-art parallel algorithms, the designs and properties are categorized into falsification and reachability. Falsification experiments were conducted on Kepler cluster (described in section 10.1.1) and reachable experiments were conducted on our new high performance computing (HPC) cluster.

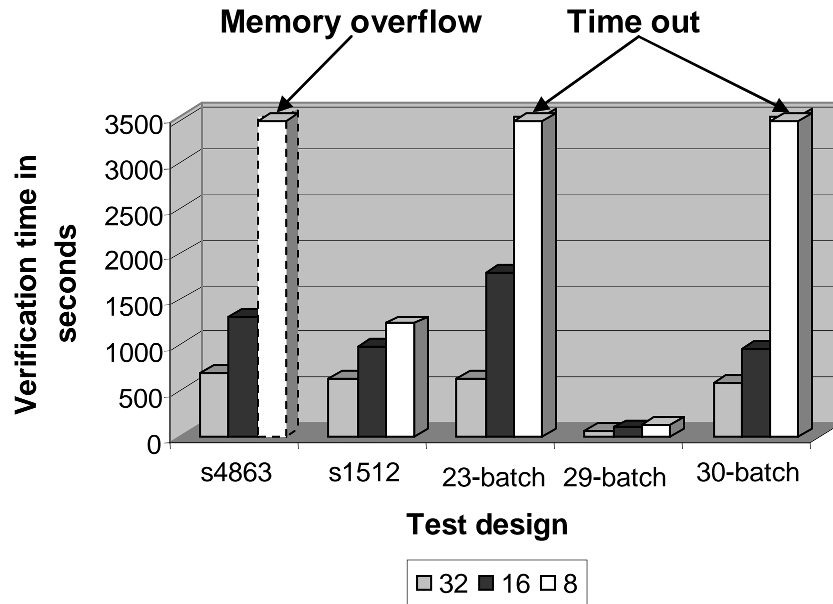

Figure 10.13: Speedup comparison graph.

10.4.1 Falsification

For falsification, thesis compared results for the design *s1423* with properties p_1 and p_2 from [117]. The authors in [117] parallelized the high performance verification tool Rulebase [135] from IBM. In this work they combined the approaches of on-the-fly model checking of RCTL formulas [136] and parallel rechability analysis [97].

In comparison with our parallel setup described in section 10.1.1, the state-of-the-art used a parallel environment with 32 RS6000 machines, each node in the environment consisting of a 225 MHz PowerPC processor with 512 Mb memory. For communication between the network nodes they used a 16 Mbit/second token ring. Though it is unfair to compare results that were achieved on different parallel environments, an attempt was made to state that the verfication times were reduced in a notable amount using the algorithms described in this thesis. Fig. 10.15 shows the verification comparison results for falsified properties for the design *s1423*. Since there is no clear available information on processor's peak performance comparison results, i.e., between PowerPC/225 (Generation G2) and Pentim III/650, it was difficult to normalize the verification times for state-of-the-art comparison.

10.4.1.1 Discussion

Fig. 10.15 visualizes the state-of-the-art comparison results for falsified properties for the large design *s1423*. The first column describes the algorithm utilized. The second column gives the number of nodes utilized by each approach. The third and fourth columns give the sequential time (S_t) and total verification time (V_t)

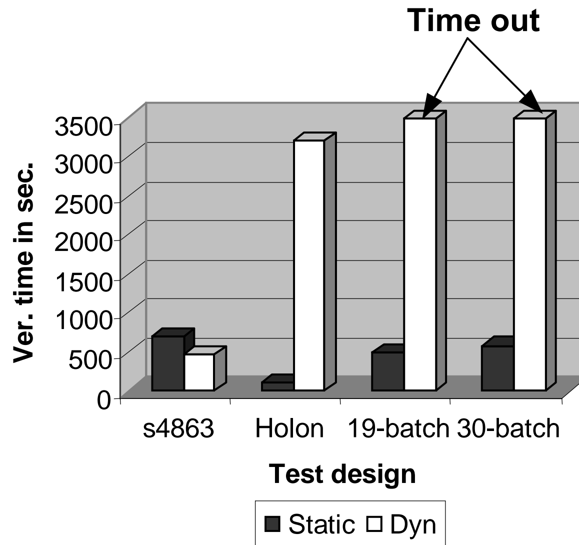

Figure 10.14: Comparison of verification of times using static and dynamic variable orderings.

in seconds, respectively.

Property P1: AG (G729 & G726 → AX[10] (G726))			
Algorithm	Nodes utilized	S_t in sec.	V_t in sec.
State-of-the-art	32	13	16,032
Dyn. overlap red.	32+1	75.5	721
Hybrid method	32+2	75.3	187
Property P2: AG (G729 & G726 → AX[7] (G726))			
State-of-the-art	32	116	521
Dyn. overlap red.	32+1	75.2	164
Hybrid method	32+2	75.6	166

Figure 10.15: State-of-the-art comparison results for falsified properties.

For the property P_1, both hybrid and dynamic overlap reduction methods found an error state significantly faster, even taking different hardware configurations into account. The dynamic overlap reduction achieved a speedup of upto of 22 and *hybrid* method achieved a speedup of upto 85. However, the dynamic overlap reduction method required one extra node acting as a Coordinator and *hybrid* method required two extra nodes one acting as a *Coordinator* and the other acting as a *static helper*.

As memory is directly proportial to the peak node count of the BDD manager, Fig. 10.16 visualizes the memory utilization during breadth first search (BFS) traversal steps of the verification process. Both dynamic overlap reduction and hybrid algorithms utilized the same splitting threshold (BDD nodecount of 50000). Therefore, the memory utilization for the node that computed the sequential

stage has similar characteristics and the lines for both dynamic and hybrid approaches overlap in the sequential stage of the figure. The parallel stage of Fig. 10.16 (8-14 simulation time steps) compares the memory utilization with state-of-the-art parallel algorithm [117]. The memory utilization in this stage represent the average peak node count of all the network nodes (i.e., the sum of peak node count of all the nodes divided by number of nodes in parallel phase). The dynamic overlap reduction took more memory in comparison with state-of-the-art and hybrid approaches but took significantly less time compared with state-of-the-art. The dynamic overlap reduction approach is more suitable for full validation. Therefore, there is not strong restriction on state space during the traversal, hence more memory was required.

Due to window restrictions (each window was restricted with window function that involved 5 variables, i.e., $2^5 = 32$) in hybrid approach, each node in the network took less memory and also perform the state space traversal relatively faster. Therfore, the hybrid approach achieved the end result faster.

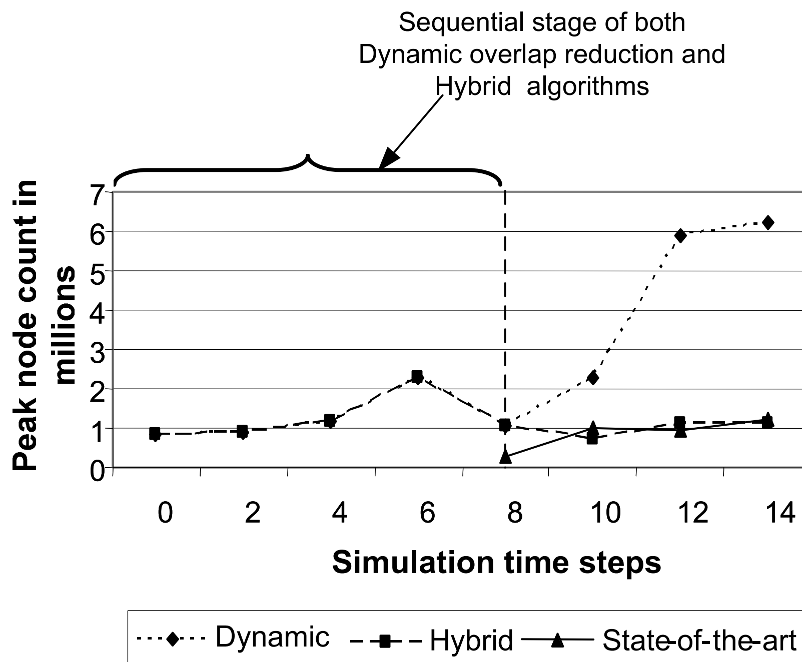

Figure 10.16: Memory utilization during the verification process for the property P_1.

For the property P_2 both hybrid and dynamic overlap reduction techniques took less verification time (speedup of 3.17) compared to the state-of-the-art algorithm. Since the error states were located very close to the splitting point, i.e., even before reaching a reduction point in case of *dynamic overlap reduction* and restriction in case of hybrid approach, the verification results for both the approaches are almost equal.

10.4.2 Reachability

Since non availability of the Kepler cluster at the time of reachability experiments, the further reachability experiments were conducted on our new high performance computing cluster. This cluster contains 40 computing nodes, each consisting of Intel DualCore 2.66 GHz Xeon CPU 5150 (2 CPUs per blade) with 8 Gb of shared memory. For reachability checking, the results for the designs *s1423* and *s3330* with [98] were compared. The authors in [98] presented a distributed algorithm which dynamically allocates and reallocates the processes to tasks in order to recover from local state explosion. They implemented a parallel algorithm in Division [98] , a generic platform for distributed symbolic model checking. Division requires a model checker as an external module. Therefore, they used NuSMV [137] for this purpose. Their parallel environment included 25 PC machines, each consisting of dual 1.7 GHz Pentium 4 processors with 1 Gb memory. The communication between the nodes consisted of a fast Ethernet. For both the designs *s1423* and *s3330* the authors in [98] claimed that they suffered with *memory overflow* at time steps 13 and 3, respectively.

For a fair comparison, maximum of 17 processors were used for dynamic overlap reduction approach. Fig. 10.17 shows the reachability results using dynamic overlap reduction method. The first column in Fig. 10.17 lists the design name. The second column gives the number of flip-flops used in the design. Column three denotes the targeted number (time steps) of reachable state sets from the set of initial states. Two subcolumns in column fourth specify the verification times in seconds taken by dynamic overlap reduction approach using 9 and 17 number of nodes. Both these node sets consisted of one extra node *Coordinator* for organizing the communication and overlap removal. $\#n$ denotes memory overflow at step n. If any of the node requires more than 512 Mb physical memory then the approach is aborted with *memory overflow* condition due to fair comparison with state-of-the-art.

Design	FFs	Steps	v_t in sec.	
			8+1	16+1
s1423	74	13	#13	1636
s3330	132	4	3023	1860

Figure 10.17: Reachability results.

10.4.2.1 Discussion

For both the designs *s1423* and *s3330* the dynamic overlap reduction method started its reachability on certain intial state set and continued until the BFS steps provided in the Fig. 10.17. Both Fig. 10.18 and Fig. 10.19 depict the state-of-the-art comparison with respect to the number of processors utilized in each BFS step for both the designs. For both the designs the state-of-the-art suffered work overflow problem, i.e., a situation where the approach required more than 60 nodes.

For the design *s1423* the dynamic overlap reduction method using 9 nodes suffered memory overflow problem at 13th image computation step. The same method using 17 nodes successfully finished 13 image computation steps. The maximum amount of memory consumed by all the network nodes is 418 Mb, which is within the memory limit (512 Mb per node) used by the state-of-the-art parallel environment. For the design *s3330* the dynamic overlap reduction method using both 9 and 17 computational nodes successfully finished 4 BFS steps with a speedup of 1.6.

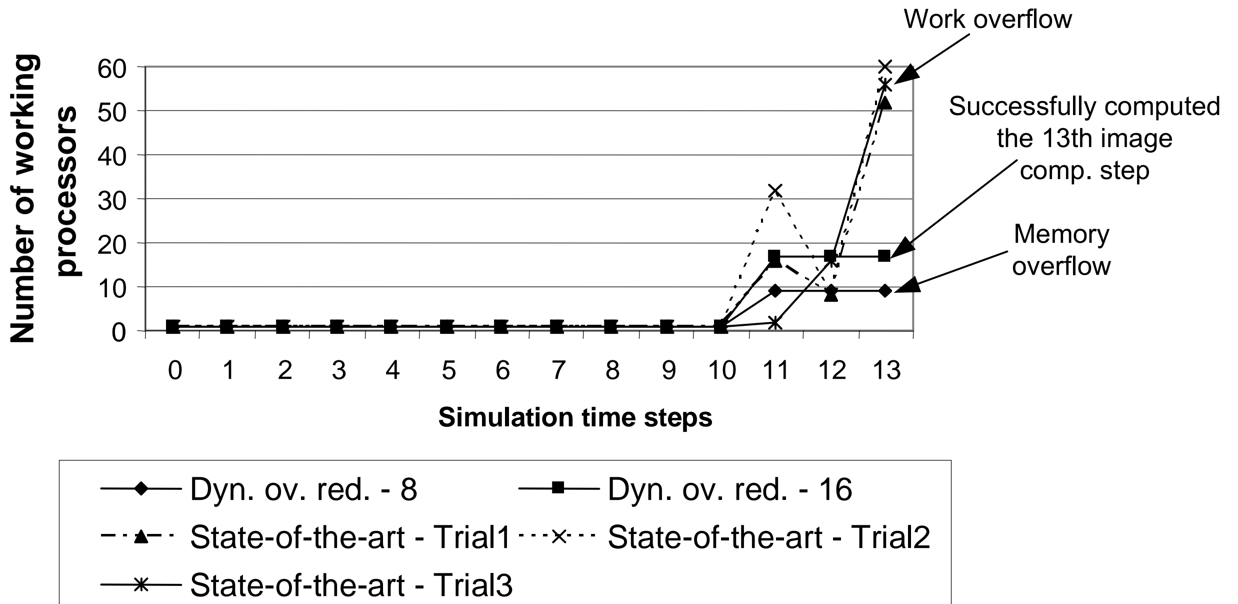

Figure 10.18: State-of-the-art comparison with respect to the number of processors utilized in each BFS step for the design s1423.

10.5 Grid-based Fast Falsification

10.5.1 Grid Configuration

The grid results were performed using three SUN x4100 servers. Each machine consists of two dual core AMD Opteron CPUs with 2.4 Ghz, 4 Gb of RAM and Fedora Core 6 operating system. To simulate different heterogeneous test cases and for the virtualization of the different scenarios XEN [138] was used. One virtual machine, configured with 1 core and 1 Gb of RAM, wass used as master node. On this master node all UNICORE services and the SGE-Master were installed.

For computational nodes 4 virtual machines were configured with each having 1 core with 512 Mb RAM. One machine was configured as a SMP node with 4 cores having 4 Gb RAM, so in total 5 machines were used with 8 cores. As interconnection a normal LAN in different setups from 10 Mb to 1000Mb was used, so that a loosely coupled system like most grids was achieved. High-performance

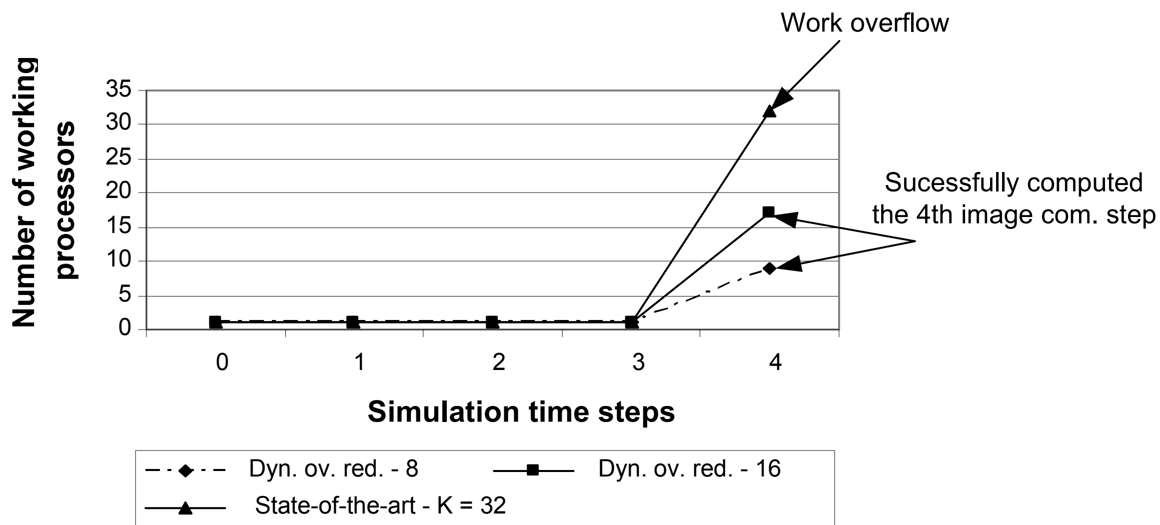

Figure 10.19: State-of-the-art comparison with respect to the number of processors utilized in each BFS step for the design s3330.

interconnections like Myrinet or Infiniband were not used. This approach is very flexible because of the virtualization. With this environment it was possible to test the algorithms in flexible setups.

Fig. 10.20 shows the results of the grid based distributive approach [125]. The first column indicates the design. The second column shows the number of flip flops used in the design. The third column shows the time bound specified in the property. Column four shows the maximum peak node count (in millions) of the node that detected the error state. The fifth column lists the overall verification time in seconds. The last column shows the node's rank that detected the error state and in brackets the guiding algorithm (described in section 3.3.5) it has used is specified. Both Ip and St represent *Input variable guiding* and *State variable guiding*, respectively.

For all the designs verification was finished by using the grid based approach. For the designs *s1423* and *s3271* two properties $p1$ and $p2$ were checked. For design *s1423* both $p1$ and $p2$ are pure LTL properties. It is noticeable in experiments (in the last column) that for some of the designs, nodes that use the *Input variable guiding*(Ip) detected the error state and for the remaining designs nodes that use the *State variable guiding* (St) detected the error state. So both guiding algorithms have their own significance. In all grid based experiments the threshold size as *20000* was used, i.e., the limitation on size of the BDD that represents the state set.

Fig. 10.21 shows the comparison results of the cluster based distributive approach (performed on Kepler cluster) and the new grid based distributive approach. Fig. 10.22 compares the grid and cluster setups. The first column in Fig. 10.22 lists the approach. The second column gives the machine configuration utilized in each approach. For grid approach the machine configuration was decribed elaborately in section 10.5.1. Column three denotes the interconnection utilized in each approach. Finally, the fourth column describes the algorithm uti-

Design	FFs	Time bound	Peak node count	Verification time in sec.	Node's rank & (Algorithm)
s4863	104	5	19	406	6 (Ip)
s1423$_{p1}$	74	-	4	167	1 (St)
s1423$_{p2}$	74	-	8.2	457	3 (St)
s3271$_{p1}$	116	20	4.4	95	6 (Ip)
s3271$_{p2}$	116	12	75	1920	6 (Ip)
Holon	118	1000	1.7	98	3 (St)
s1269	37	5	62	1232	7 (St)
04-batch	256	50	2.1	63	6 (Ip)
19-batch	181	200	2.3	613	5 (St)

Figure 10.20: Results of the grid based distributive approach.

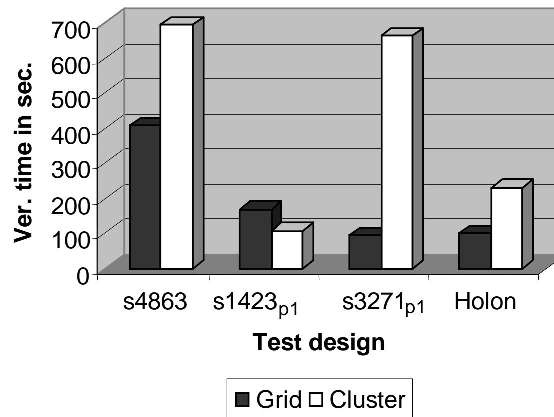

Figure 10.21: Comparison between grid and cluster based distributive approaches.

lized for each approach.

The cluster approach uses the hybrid algorithm described in section 6.2. For all designs verification is finished by using the grid based approach. In most cases the grid approach outperforms the cluster approach due to more computational power of the used nodes and also intellegent underapproximation.

10.5.2 Discussion

For the designs *s4863*, *s3271*$_{p1}$ and *Holon* the grid based approach took less verification time compared to the cluster approach. For the designs *s4863* and *s3271* the cluster approach took less memory due to algorithms used in this approach for reducing the memory consumption. In simple words, the cluster approach gives more priority to reduce the state space by regular restrictions but has more communication overhead.

For the designs *s3271*$_{p2}$ and *s1269*, the results are not available for the cluster

Parallel Env.	Machine configuration	Interconnection	Algorithm used
Grid	Two types of nodes were used 1. Four virtual machines with each having 1 core (from AMD Opteron dual core with 2.4 GHz) with 512 Mb RAM 2. An SMP node with 4 cores (2 dual core AMD Opteron with 2.4 GHz) with 4 Gb of shared RAM **In total the approach used 8 cores**	Normal LAN	Grid based falsification alg.
Kepler cluster	Each node consisting of a dual 650 MHz Pentium-III processor with 1 Gb of shared memory (512 Mb for each processor) **In total the approach used 34 cores**	Myrinet	Hybrid alg.

Figure 10.22: Grid and cluster setup comparison.

approach. But a priori experience on the cluster lets us envisage that there would be a memory overflow with these designs due to memory restriction on each node. At one simulation point, all nodes of the cluster required to have more than 1 Gb of RAM to compute the state space traversal. Whereas in the grid environment the computational nodes possess of enough memory to compute the end result.

Grid based bounded property checking for bug hunting is effective. It is computationally cheap in terms of overhead and an alternative way of parallelizing the bounded property checking. The only data that is transmitted is at the time of termination and no synchronization is required. Such parallelization needs no interdependence at all and therefore it can effectively check the existence of bugs for very large designs for large time bounds.

Chapter 11

Conclusions and Future Work

Originality consists of the achievement of new combinations, and not of the creation of something out of nothing.
- Richard V. Clemence

11.1 Conclusions

Formal verification techniques like symbolic model checking based on BDDs provide exhaustive coverage and they have the calibre to uncover subtle bugs in comparison with traditional simulation. However, for large industrial designs these techniques do not scale well due to the state explosion problem. An alternative option is to use the partitioning approach. The idea is partition the state set into smaller subsets when the representation of the state set size reaches a certain threshold size. Then, the verification tool explores these subsets sequentially. Once the tool reaches a target state, we can save time and space by skipping the exploration of the rest of the partitions. Assume, our design is error free or the target state can be reached only in the final partition. Then we have to explore all the partitions, which is a time consuming process. Therefore, for industrial sized designs the problems still pursue. One optimal solution is to parallelize the formal verification algorithms. The idea of the approach is to partition the state space upon reaching a threshold limit and to assign the traversal of the subsets to network nodes.

However, the current BDD based parallel schemes often suffer from state overlap or duplicate work, cross over states among partitions and synchronosity. For example the basic methodology is fairly simple to construct but the state overlap between network nodes may pursue after few steps of traversal. The *dynamic overlap reduction* is an important technique in enabling verification of larger designs. It uses extra resources like *coordinator* for removing state overlap. In addition, it has the natural side effect of load balancing among network nodes. However, this methodology is best suitable for full validation. Since all the nodes work with their entire state space, the method would require more time for find-

ing error states that have the longer counterexample lengths.

This thesis also presented a hybrid distributed symbolic verification algorithm based on *windowing* and *dynamic overlap reduction* techniques, suited for full validation and fast falsification. The approach is mainly asynchronous and dynamic in nature. The nodes on the cluster machine are employed with two different types of tasks. Some nodes, *windows*, restrict their state space, i.e., to hold the newly computed state space with in its window region and aim at faster falsification. The other type of nodes *helpers* are responsible for traversal of cross over states, i.e., states that are left over by *windows*, with regular reduction of state overlap. The hybrid approach makes the system properly balanced with respect to the node's work load, i.e., nodes having less work at one time step will later be assigned with onerous task. The longevity of ideal nodes lasts very quickly, i.e., the ideal nodes will be reassigned with the work as quickly as possible. The ideal nodes in parallel phase are the nodes that are kept waiting for some reassignment of work. The state space restriction and overlap reduction operations are well controlled by the user. Therefore, the system uses the optimal amount of coordination. In addition, the hybrid method uses an efficient mode (binary) of data transmission between network nodes. The hybrid methodology utilizes the *Minimal overlap* algorithm for state set distribution. The algorithm aims at minimizing the cross over states among the partitions by analyzing the structural information of the design. Further, the hybrid algorithm equally distributes the partitioning effort on all nodes. So, all these factors make the hybrid distributed symbolic algorithm scalable and well suited for fast falsification and full validation.

The proposed solutions are to deal with the formal verification of large designs on a homogeneous cluster environment. However, there exist very large industrial designs for which finding an error state or target state is vital. This thesis also presented a new grid based distributed bounded symbolic verification approach based on effective combination of the intelligent partitioning algorithms that is best suited for fast falsification. The approach is a totally asynchronous. In addition, this thesis presents a novel distributed algorithm for Black Box verification. The approach uses a mixed forward and backward traversal mechanism.

Overall the distributed algorithms have several advantages. They enable the verification of larger models than those with the regular nonparallel version. A sequential version fails for these designs because it often encounters state space explosion early on in the computation, after which it could not make much progress due to memory limitations. However, the reduced memory requirements for the network nodes in the distributed version still allow progress in the traversal process. Therefore, it is able to finish large circuits. The parallel approaches can exploit any network size and their utilization of network resources make them suitable for solving very large verification problems.

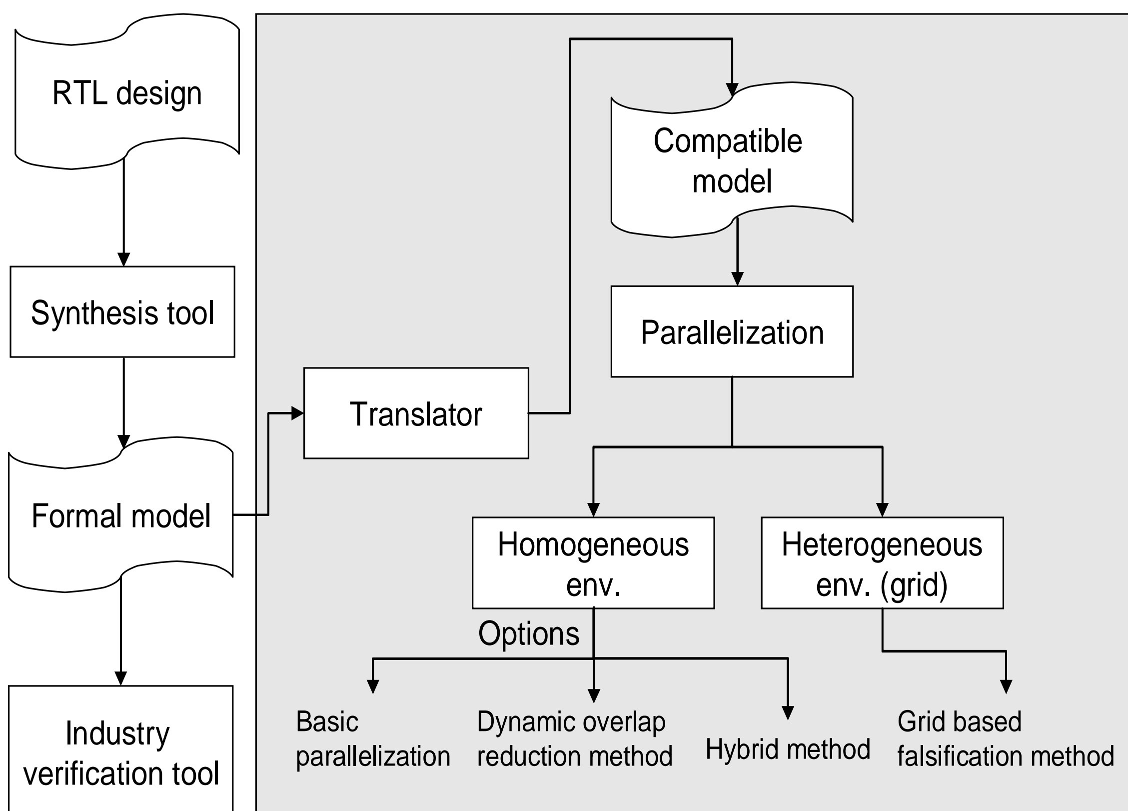

Figure 11.1: Application of tools developed in this thesis into industry.

11.1.1 Benefits to the Industry

This thesis focussed on efficient parallelization of symbolic bounded property checking. Therefore, on top of any BDD based symbolic model checker, without much of alteration, the parallel algorithms can be applied directly. The prominent parallel algorithms in thesis include, the dynamic overlap reduction, the hybrid approach (both *windowing* and *dynamic overlap reduction*), the grid based falsification approach, the parallel partitioning, an efficient data transmission and dynamic load balance between computational nodes. In simplest, we can easily merge the existing tool into industry verification flow by extracting the model that has similar semantics to the model this thesis used. Then, depending on the parallel environment we can apply directly the available parallel tool with different verification algorithms. Fig. 11.1 shows the integration of the developed tools into an industrial flow.

11.2 Future Work

As future work I would like to focus my research in four different directions: 1. Further optimize the hybrid methodology; 2. Options to further explore grid based verification; 3. Parallelization of a Black Box verification; 4. Parallel soft-

ware verification.

Currently, the *window* restriction in hybrid algorithm is performed at every n steps without considering the size of the frontier state set. I would like to apply the restriction only when needed, i.e., if the size of the frontier state set crosses a certain threshold at every defined number of steps then apply the *window* restriction. This trivially can reduce the cross over states computation. Further, I would like to distribute the coordinating effort on more number of nodes.

The initial success of the grid based approach motivates us to further explore on optimized communication for effective handling of the state space among the nodes. Moreover, a dynamic allocation of nodes, when required, would make this approach more competitive. Future research will focus on a hybrid approach in which full validation is also handled along with the fast falsification.

It is conspicuous in this thesis that only algorithms for parallelization of Black Box verification for incomplete designs are presented. In the very near future, I would like to implement and pilot these algorithms on a cluster machine. However, by considering the success stories of distributed verification I can envisage an upbeat outcome.

The verification of complex systems can not be considered only on the hardware module level anymore. The amount of software has increased significantly over the last years and therefore, the verification of embedded software has become a fundamental importance. However, the verification of embedded software is much harder than for the hardware due to its complexity. Therefore, I would like to apply parallel principles in the area of software verification.

Bibliography

[1] J. Edmund M. Clarke, O. Grumberg, and D. A. Peled, *Model checking*. Cambridge, MA, USA: MIT Press, 1999.

[2] "Collection of Bugs at GSI." [Online]. Available: http://www-aix.gsi.de/~giese/swr/ariane5.html

[3] "Collection of Software Bugs at University of München." [Online]. Available: http://www5.in.tum.de/~huckle/bugse.html

[4] B. Wile, J. Goss, and W. Roesner, *Comprehensive functional verification: The complete industry cycle (systems on silicon)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[5] J. Bergeron, *Writing Testbenches: Functional Verification of HDL Models, Second Edition*. Norwell, MA, USA: Kluwer Academic Publishers, 2003.

[6] T. Kropf, *Introduction to formal hardware verification: methods and tools for designing correct circuits and systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1999.

[7] D. W. Hoffmann, J. Ruf, T. Kropf, and W. Rosenstiel, "Simulation meets verification – verifying temporal properties in SystemC," in *Proceedings of the Symposium on Digital Systems Design (DSD)*, Maastricht, The Netherlands, 2000.

[8] K. Chang, W. Tu, Y. Yeh, and S. Kuo, "A simulation-based temporal assertion checker for PSL," in *46th IEEE International Midwest Stmp. on Circuits and Systems*. IEEE Computer Society, 2003, pp. 1528–1531.

[9] R. J. Weiss, J. Ruf, T. Kropf, and W. Rosenstiel, "Efficient and customizable integration of temporal properties into systemc," *Forum on Specification and Design Languages (FDL'05)*, September 2005.

[10] R. Armoni, D. Korchemny, A. Tiemeyer, M. Vardi, and Y. Zbar, "Deterministic dynamic monitors for linear-time assertions," in *Proceedings of the Workshop on Formal Approaches to Testing and Runtime Verification*, vol. 4262 of Lecture Notes in Computer Science. Springer, 2006. [Online]. Available: citeseer.ist.psu.edu/armoni06deterministic.html

[11] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Software verification with BLAST," in *Tenth International Workshop on Model Checking of Software (SPIN), volume 2648 of Lecture Notes in Computer Science*. Springer-Verlag, 2003, pp. 235–239.

[12] "SATABS - predicate abstraction with SAT for ANSI-C." [Online]. Available: http://www.verify.ethz.ch/satabs/

[13] R. Brummayer and A. Biere, "C32SAT: Checking C expressions." [Online]. Available: http://fmv.jku.at/c32sat

[14] E. M. Clarke, "Automatic verification of finite-state concurrent systems," in *Proceedings of the 15th International Conference on Application and Theory of Petri Nets*. London, UK: Springer-Verlag, 1994, p. 1.

[15] E. M. Clarke and I. A. Draghicescu, "Expressibility results for linear-time and branching-time logics," in *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop*. London, UK: Springer-Verlag, 1989, pp. 428–437.

[16] G. J. Holzmann, *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Professional, September 2003.

[17] R. Nalumasu and G. Gopalakrishnan, "PV: An explicit enumeration model-checker," in *FMCAD '98: Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design*. London, UK: Springer-Verlag, 1998, pp. 523–528.

[18] R. E. Bryant, "Symbolic simulation - techniques and applications," in *Design Automation Conference*, 1990, pp. 517–521. [Online]. Available: citeseer.ist.psu.edu/bryant90symbolic.html

[19] A. J. Hu, "Formal hardware verification with BDDs: An introduction," in *IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing (PACRIM)*, October 1997, pp. 677–682.

[20] R. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, August 1986.

[21] J. Burch, E. Clarke, K. L. McMillan, and D. Dill, "Sequential circuit verification using symbolic model checking," in *ACM/IEEE Design Automation Conference (DAC)*, ACM/IEEE. Los Alamitos, CA: IEEE Society Press, June 1990, pp. 46–51.

[22] J. Burch, E. Clarke, and D. Long, "Symbolic model checking for sequential circuit verification," in *International Conference on Very Large Scale Integration (VLSI)*, Edinburgh, Scottland, August 1990, pp. 49–58.

[23] J. Burch, E. Clarke, K. L. McMillan, D. Dill, and L. Hwang, "Symbolic model checking: 10^{20} states and beyond," *Information and Computing*, vol. 98, no. 2, pp. 142–170, June 1992.

[24] O. Coudert, J. C. Madre, and C. Berthet, "Verifying temporal properties of sequential machines without building their state diagrams," in *CAV '90: Proceedings of the 2nd International Workshop on Computer Aided Verification*. London, UK: Springer-Verlag, 1991, pp. 23–32.

[25] K. L. McMillan, "Symbolic model checking: an approach to the state explosion problem," Ph.D. dissertation, Pittsburgh, PA, USA, 1992.

[26] C. Eisner and D. Peled, "Comparing symbolic and explicit model checking of a software system," in *Proceedings of the 9th International SPIN Workshop on Model Checking of Software*. London, UK: Springer-Verlag, 2002, pp. 230–239.

[27] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*. London, UK: Springer-Verlag, 1999, pp. 193–207.

[28] A. Biere, E. Clarke, R. Raimi, and Y. Zhu, "Verifying safety properties of a PowerPC microprocessor using symbolic model checking without BDDs," *Lecture Notes in Computer Science*, vol. 1633, 1999. [Online]. Available: citeseer.ist.psu.edu/biere99verifying.html

[29] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," *Form. Methods Syst. Des.*, vol. 19, no. 1, pp. 7–34, 2001.

[30] M. Davis and H. Putnam, "A computing procedure for quantification theory," *Journal of the ACM*, vol. 7, pp. 201–215, 1960.

[31] G. Cabodi, P. Camurati, and S. Quer, "Can BDDs compete with SAT solvers on bounded model checking?" in *DAC '02: Proceedings of the 39th conference on Design automation*. New York, NY, USA: ACM Press, 2002, pp. 117–122.

[32] G. Parthasarathy, M. K. Iyer, and K.-T. Cheng, "A comparison of BDDs, BMC, and sequential SAT for model checking," in *HLDVT '03: Proceedings of the Eighth IEEE International Workshop on High-Level Design Validation and Test Workshop*. Washington, DC, USA: IEEE Computer Society, 2003, p. 157.

[33] K. Ravi and F. Somenzi, "High-density reachability analysis," in *1995 IEEE/ACM International Conference on CAD*. ACM and IEEE Computer Society Press, 1995, pp. 154–158.

[34] G. Cabodi, P. Camurati, and S. Quer, "Improved reachability analysis of large finite state machines," in *1996 IEEE/ACM International Conference on CAD*. ACM and IEEE Computer Society Press, 1996, pp. 354–360.

[35] A. Narayan, A. J. Isles, J. Jain, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Reachability analysis using partitioned-ROBDDs," in *1997*

IEEE/ACM International Conference on CAD. ACM and IEEE Computer Society Press, 1997, pp. 388–393.

[36] H. Garavel, R. Mateescu, and I. Smarandache, "Parallel state space construction for model-checking," *Lecture Notes in Computer Science*, vol. 2057, 2001. [Online]. Available: citeseer.ist.psu.edu/article/garavel01parallel.html

[37] W. Prasetya, "Formalization of variables access constraints to support compositionality of liveness properties," in *Higher Order Logic Theorem Proving and its Applications*, ser. Lecture Notes in Computer Science, J. Joyce and C.-J. Seger, Eds., vol. 780, University of British Columbia. Vancouver, Canada: Springer-Verlag, published 1994, August 1993, pp. 324–338.

[38] S. Graf, "Verification of distributed cache memory by using abstractions," in *Conference on Computer Aided Verification (CAV)*, ser. Lecture Notes in Computer Science, D. L. Dill, Ed., vol. 818. Stanford, California, USA: Springer-Verlag, June 1994, pp. 207–219.

[39] R. Reetz and T. Kropf, "A flowgraph semantics of VHDL: A basis for hardware verification with VHDL," in *Formal Semantics for VHDL*, ser. The Kluwer international series in engineering and computer science, C. Delgado Kloos and P. Breuer, Eds. Madrid, Spain: Kluwer Academic Publishers, March 1995, vol. 307, ch. 7, pp. 205–238.

[40] A. Olivero, J. Sifakis, and S. Yovine, "Using abstractions for the verification of linear hybrid systems," in *Conference on Computer Aided Verification (CAV)*, ser. Lecture Notes in Computer Science, D. L. Dill, Ed., vol. 818. Stanford, California, USA: Springer-Verlag, June 1994, pp. 81–94.

[41] R. Hojati and R. K. Brayton, "Automatic datapath abstraction in hardware systems," in *Conference on Computer Aided Verification (CAV)*, ser. Lecture Notes in Computer Science, P. Wolper, Ed., vol. 939. Liege, Belgium: Springer Verlag, July 1995, pp. 98–113.

[42] E. Emerson, S. Jha, and D. Peled, "Combining partial order and symmetry reductions," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, E. Brinksma, Ed. Enschede, The Netherlands: Springer Verlag, LNCS 1217, 1997, pp. 19–34.

[43] R. Nalumasu and G. Gopalakrishnan, "A new partial order reduction algorithm for concurrent systems," in *IFIP Conference on Computer Hardware Description Languages and their Applications (CHDL)*, C. Delgado Kloos and E. Cerny, Eds., IFIP WG 10.5. Toledo, Spain: Chapman and Hall, April 1997.

[44] P. M. Peranandam, "Efficient system traversal and property verification by exploring circuit locality," in *Ph.D. Thesis at University of Tübingen, Germany*, 2006.

[45] C. Jard and T. Jéron, "Bounded-memory algorithms for verification on-the-fly," in *CAV '91: Proceedings of the 3rd International Workshop on Computer Aided Verification*. London, UK: Springer-Verlag, 1992, pp. 192–202.

[46] R. Buyya, *High Performance Cluster Computing: Architectures and Systems*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1999.

[47] "FEST - funktionale verifikation von systemen." [Online]. Available: http://www.em.cs.uni-frankfurt.de/cluster/index.php?id=37

[48] E. Moore, "Gedanken-experiments on sequential machines," in *Automata Studies*, C. Shannon and J. MacCarthy, Eds. Princeton, New Jersey: Princeton University Press, 1956, pp. 129–153.

[49] P. P. Chu, *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*. Wiley-IEEE Press, 2006.

[50] G. H. Mealy, "A method for synthesizing sequential circuits," *Bell System Technical Journal*, vol. 34(5), pp. 1045–1079, 1955.

[51] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. 35, no. 8, pp. 677–691, 1986. [Online]. Available: citeseer.ist.psu.edu/bryant86graphbased.html

[52] A. J. Hu, "Formal hardware verification with BDDs: An introduction," in *Pacific Rim Conference on Communications, Computers, and Signal Processing (PACRIM)*. IEEE, 1997, pp. 677–682.

[53] M. Fujita, H. Fujisawa, and N. Kawato, "Evaluation and improvements of boolean comparison method based on binary decision diagrams," in *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE Computer Society Press, 1988, pp. 2–5.

[54] M. Fujita, Y. Matsunaga, and T. Kakuda, "On variable ordering of binary decision diagrams for the application of multi-level logic Synthesis," in *European Design Automation Conference*, IEEE. Amsterdam: IEEE Computer Society Press, February 1991, pp. 50–54.

[55] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," in *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, ACM/IEEE. Santa Clara, California: IEEE Computer Society Press, November 1993, pp. 42–47.

[56] N. Ishura, H. Sawada, and S. Yajima, "Minimization of binary decision diagrams based on exchange of variables," in *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 1991, pp. 472–475.

[57] H. Higuchi and F. Somenzi, "Lazy group sifting for efficient symbolic state traversal of FSMs," in *ICCAD '99: Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design*. Piscataway, NJ, USA: IEEE Press, 1999, pp. 45–49.

[58] F. Somenzi, "CUDD: CU decision diagram package, release 2.4.0," Department of Electrical and Computer Engineering, University of Colorado at Boulder, 2004. [Online]. Available: http://vlsi.colorado.edu/~fabio/CUDD

[59] J. Lind-Nielsen, "Buddy: A binary decision diagram library." [Online]. Available: http://sourceforge.net/projects/buddy

[60] "CMU BDD: The BDD library." [Online]. Available: http://www.cs.cmu.edu/~modelcheck/bdd.html

[61] J. Jain, J. Bitner, D. Fussel, and J. Abraham, "Functional partitioning for verification and related problems," in *Brown/MIT VLSI Conference*, March 1992.

[62] J. Jain, "On analysis of Boolean functions," Ph.D. dissertation, Department of Electrical and Computer Engineering, The University of Texas at Austin, Pittsburgh, PA, 1993.

[63] A. Narayan, J. Jain, M. Fujita, and A. L. Sangiovanni-Vincentelli, "Partitioned ROBDDs - a compact, canonical and efficiently manipulable representation for Boolean functions," in *1996 IEEE/ACM International Conference on CAD*. ACM and IEEE Computer Society Press, 1996, pp. 547–554.

[64] F. Somenzi and R. Bloem, "Efficient Büchi automata from LTL formulae," in *Computer Aided Verification, 12th International Conference*, ser. Lecture Notes in Computer Science, E. A. Emerson and A. P. Sistla, Eds., vol. 1855. Springer Verlag, 2000, pp. 248–263.

[65] J. Ruf, D. W. Hoffmann, T. Kropf, and W. Rosenstiel, "Simulation-guided property checking based on a multi-valued AR-automata," in *Design, Automation and Test in Europe 2001*, W. Nebel and A. Jerraya, Eds. IEEE Press, 2001, pp. 742–748.

[66] J. R. Burch, E. M. Clarke, and D. E. Long, "Representing circuits more efficiently in symbolic model checking," in *28th Conference on Design Automation*. ACM Press, 1991, pp. 403–407.

[67] J. Burch, E. Clarke, D. Long, K. L. McMillan, and D. Dill, "Symbolic model checking for sequential circuit verification," Carnegy Mellon University, Pittsburg, PA 15213, Tech. Rep. CMU-CS-93-211, July 1993.

[68] J. Ruf and P. Peranandam, "Bounded property checking with symbolic simulation," in *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, GI/ITG/GMM Workshop. Shaker Verlag, February 2003, pp. 209–218.

[69] J. Ruf, P. M. Peranandam, T. Kropf, and W. Rosenstiel, "Using symbolic simulation for bounded property checking," *Forum on Specification and Design Languages (FDL'03)*, September 2003.

[70] J. Ruf, D. W. Hoffmann, T. Kropf, and W. Rosenstiel, "Simulation based validation of FLTL formulas in executable system descriptions," in *Forum on Design Languages (FDL 2000)*, R. Seepold, Ed. Tübingen, Germany: Sig.-VHDL and ECSI, September 2000, pp. 311–319.

[71] K. L. McMillan, "Symbolic model checking: An approach to the state explosion problem," Ph.D. dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1992, cMU-CS-92-131.

[72] D. Geist and I. Beer, "Efficient model checking by automated ordering of transition relation," in *Proceedings of the sixth International Conference on Computer-Aided Verification CAV*, David L. Dill, Ed., vol. 818. Standford, California, USA: Springer-Verlag, 1994, pp. 299–310.

[73] R. Hojati, S. C. Krishnan, and R. K. Brayton, "Early quantification and partitioned transition relations," in *ICCD '96: Proceedings of the 1996 International Conference on Computer Design, VLSI in Computers and Processors*. Washington, DC, USA: IEEE Computer Society, 1996, pp. 12–19.

[74] C. Scholl and B. Becker, "Checking equivalence for partial implementations," in *DAC '01: Proceedings of the 38th conference on Design automation*. New York, NY, USA: ACM, 2001, pp. 238–243.

[75] T. Nopper and C. Scholl, "Approximate symbolic model checking for incomplete designs." in *FMCAD*, ser. Lecture Notes in Computer Science, A. J. Hu and A. K. Martin, Eds., vol. 3312. Springer, 2004, pp. 290–305. [Online]. Available: http://dblp.uni-trier.de/db/conf/fmcad/fmcad2004.html#NopperS04

[76] T. Nopper and C. Scholl, "Flexible modeling of unknowns in model checking for incomplete designs," in *Modellierung und Verifikation*, ser. 8. GI/ITG/GMM Workshop, W. Ecker, Ed., 2005.

[77] S. K. Iyer, D. Sahoo, C. Stangier, A. Narayan, and J. Jain, "Improved symbolic verification using partitioning techniques," in *CHARME*, 2003, pp. 410–424.

[78] D. Sahoo, S. K. Iyer, J. Jain, C. Stangier, A. Narayan, D. L. Dill, and E. A. Emerson, "A partitioning methodology for BDD-based verification," in *FMCAD*, 2004, pp. 399–413.

[79] S. G. Govindaraju and D. L. Dill, "Verification by approximate forward and backward reachability," in *ICCAD*, 1998, pp. 366–370.

[80] S. G. Govindaraju, D. L. Dill, A. J. Hu, and M. A. Horowitz, "Approximate reachability with BDDs using overlapping projections," in *Design Automation Conference*, 1998, pp. 451–456. [Online]. Available: citeseer.ist.psu.edu/govindaraju98approximate.html

[81] G. Cabodi, S. Nocco, and S. Quer, "Mixing forward and backward traversals in guided-prioritized bdd-based verification," in *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*. London, UK: Springer-Verlag, 2002, pp. 471–484.

[82] G. Cabodi, S. Nocco, and S. Quer, "Improving SAT-based bounded model checking by means of BDD-based approximate traversals," in *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2003, p. 10898.

[83] S. Barner and O. Grumberg, "Combining symmetry reduction and under-approximation for symbolic model checking," in *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*. London, UK: Springer-Verlag, 2002, pp. 93–106.

[84] T. Nopper, C. Scholl, and B. Becker, "Project FEST: Funktionale verifikation von systemen," 2007. [Online]. Available: http://www.em.cs.uni-frankfurt.de/cluster/index.php?id=37

[85] K. Ravi and F. Somenzi, "Hints to accelerate symbolic traversal," in *CHARME '99: Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. London, UK: Springer-Verlag, 1999, pp. 250–264.

[86] K. Ravi, "Adaptive techniques to improve state space search in formal veri cation," Ph.D. dissertation, University of Colorado, Department of Electrical and Computer Engineering, Colorado, 1999.

[87] R. Bloem, K. Ravi, and F. Somenzi, "Efficient decision procedures for model checking of linear time logic properties," in *CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification*. London, UK: Springer-Verlag, 1999, pp. 222–235.

[88] R. Bloem, K.Ravi, and F.Somenzi, "Symbolic guided search for CTL model checking," in *DAC '00: Proceedings of the 37th conference on Design automation*. New York, NY, USA: ACM Press, 2000, pp. 29–34.

[89] C. H. Yang and D. L. Dill, "Validation with guided search of the state space," in *Design Automation Conference (DAC)*. San Francisco, CA: ACM/IEEE, June 1998, pp. 599–604.

[90] D. Sahoo, J. Jain, S. K. Iyer, D. L. Dill, and E. A. Emerson, "Multi-threaded reachability," in *DAC '05: Proceedings of the 42nd annual conference on Design automation*. New York, NY, USA: ACM Press, 2005, pp. 467–470.

[91] U. Stern and D. L. Dill, "Parallelizing the Murϕ verifier," in *Computer Aided Verification, 9th International Conference*, ser. Lecture Notes in Computer Science, O. Grumberg, Ed., vol. 1254. Springer Verlag, 1997, pp. 256–278.

[92] V. Braberman, A. Olivero, and F. Schapachnik, "Issues in distributed timed model checking: Building Zeus," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 7(1), pp. 4 – 18, February 2005.

[93] F. Lerda and R. Sisto, "Distributed-memory model checking with SPIN," in *Proc. of the 5th International SPIN Workshop*, ser. LNCS, vol. 1680. Springer-Verlag, 1999.

[94] M. Leucker, R. Somla, and M. Weber, "Parallel model checking for LTL, CTL* and L_μ^2," in *Electronic Notes in Theoretical Computer Science*, L. Brim and O. Grumberg, Eds., vol. 89. Elsevier Science Publishers, 2003.

[95] L. Brim, J. Crhova, and K. Yorav, "Using assumptions to distribute CTL model checking," in *1st International Workshop on Parallel and Distributed Methods in VerifiCation*, ser. Electronic Notes in Theoretical Computer Science. Elsevier, 2002.

[96] R. Palmer and G. Gopalakrishnan, "The parallel PVmodel-checker," in *1st International Workshop on Parallel and Distributed Methods in VerifiCation*, ser. Electronic Notes in Theoretical Computer Science. Elsevier, 2002.

[97] T. Heyman, D. Geist, O. Grumberg, and A. Schuster, "Achieving scalability in parallel reachability analysis of very large circuits," in *Computer Aided Verification, 12th International Conference*, ser. Lecture Notes in Computer Science, E. A. Emerson and A. P. Sistla, Eds., vol. 1855. Springer Verlag, 2000, pp. 20–35.

[98] O. Grumberg, T. Heyman, and A. Schuster, "A work-efficient distributed algorithm for reachability analysis," in *Computer Aided Verification, 15th International Conference*, ser. Lecture Notes in Computer Science, W. A. Hunt Jr. and F. Somenzi, Eds., vol. 2725. Springer Verlag, 2003, pp. 54–66.

[99] O. Grumberg, T. Heyman, N. Ifergan, and A. Schuster, "Achieving high-speedups in distributed symbolic reachability analysis through asynchrnous computation," in *Conference on Correct Hardware Design and Verification Methods (CHARME'05)*, October 2005.

[100] M. K. Ganai, A. Gupta, Z. Yang, and P. Ashar, "Efficient distributed SAT and SAT-based distributed bounded model checking," *Int. J. Softw. Tools Technol. Transf.*, vol. 8, no. 4, pp. 387–396, 2006.

[101] M. Böhm and E. Speckenmeyer, "A fast parallel SAT–solver — efficient workload balancing," in *Presented at Third Symposium on Artificial Intellegence and Mathematics*. New York, NY, USA: ACM Press, 1994.

[102] W. Blochinger, "Towards robustness in parallel SAT solving." in *PARCO*, ser. John von Neumann Institute for Computing Series, G. R. Joubert, W. E. Nagel, F. J. Peters, O. G. Plata, P. Tirado, and E. L. Zapata, Eds., vol. 33. Central Institute for Applied Mathematics, Jlich, Germany, 2005, pp. 301–308.

[103] D. Petcu, "Parallel explicit state reachability analysis and state space construction." Proceedings of the international Symposium on Parallel and Distributed Computing (ISPDC'03), 2003.

[104] R. Palmer and G. Gopalakrishnan, "Partial order reduction assisted parallel model checking," in *Proc. Parallel and Distributed Model Checking (PDMC) Workshop*, 2002.

[105] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine, "Kronos: A model-checking tool for real-time systems," in *Proc. 10th International Conference on Computer Aided Verification, Vancouver, Canada*, A. J. Hu and M. Y. Vardi, Eds., vol. 1427. Springer-Verlag, 1998, pp. 546–550. [Online]. Available: citeseer.ist.psu.edu/bozga98kronos.html

[106] H. Zhang, M. P. Bonacina, and J. Hsiang, "PSATO: A distributed propositional prover and its application to quasigroup problems," *Journal of Symbolic Computation*, vol. 21, no. 4, pp. 543–560, 1996. [Online]. Available: citeseer.ist.psu.edu/85349.html

[107] Y. Zhao, S. Malik, M. W. Moskewicz, and C. F. Madigan, "Accelerating boolean satisfiability through application specific processing," in *ISSS*, 2001, pp. 244–249. [Online]. Available: citeseer.ist.psu.edu/zhao01accelerating.html

[108] L. Fix, O. Grumberg, A. Heyman, T. Heyman, and A. Schuster, "Verifying very large industrial circuits using 100 processes and beyond." *Int. J. Found. Comput. Sci.*, vol. 18, no. 1, pp. 45–62, 2007. [Online]. Available: http://dblp.uni-trier.de/db/journals/ijfcs/ijfcs18.html#FixGHHS07

[109] O. Grumberg, T. Heyman, and A. Schuster, "Distributed symbolic model checking for μ-calculus," in *Computer Aided Verification, 13th International Conference*, ser. Lecture Notes in Computer Science, G. Berry, H. Comon, and A. Finkel, Eds., vol. 2102. Springer Verlag, 2001, pp. 350–362.

[110] S. Iyer, J. Jain, D. Sahoo, and E. A. Emerson, "Under-approximation heuristics for grid-based bmc," in *International Workshop on Parallel and Distributed Methods in verifiCation (PDMC'05)*, July 2005.

[111] K. Ravi, K. L. McMillan, T. R. Shiple, and F. Somenzi, "Approximation and decomposition of binary decision diagrams," in *35th Conference on Design Automation*. ACM Press, 1998, pp. 445–450.

[112] P. M. Peranandam, P. K. Nalla, R. J. Weiss, J. Ruf, T. Kropf, and W. Rosenstiel, "Overlap reduction in symbolic system traversal," in *IEEE International High Level Design Validation and Test Workshop 2005 (HLDVT 05)*, November 2005.

[113] S. Orzan, J. van de Pol, and M. V. Espada, "A state space distribution policy based on abstract interpretation," in *3rd International Workshop on Parallel*

and Distributed Methods in VerifiCation, ser. Electronic Notes in Theoretical Computer Science. Elsevier, 2004.

[114] P. M. Peranandam, P. K. Nalla, J. Ruf, R. J. Weiss, T. Kropf, and W. Rosenstiel, "Fast falsification based on symbolic bounded property checking," in *DAC '06: Proceedings of the 43rd annual conference on Design automation*. New York, NY, USA: ACM Press, 2006, pp. 1077–1082.

[115] D. Sahoo, S. K. Iyer, J. Jain, C. Stangier, A. Narayan, D. L. Dill, and E. A. Emerson, "A partitioning methodology for BDD-based verification," in *Formal Methods in Computer-Aided Design, Fifth International Conference*, ser. Lecture Notes in Computer Science, A. J. Hu and A. K. Martin, Eds., vol. 3312. Springer, 2004, pp. 399–413.

[116] P. M. Peranandam, R. J. Weiss, J. Ruf, T. Kropf, and W. Rosenstiel, "Dynamic guiding of bounded property checking," in *IEEE International High Level Design Validation and Test Workshop 2004 (HLDVT 04)*, November 2004.

[117] S. Ben-David, T. Heyman, O. Grumberg, and A. Schuster, "Scalable distributed on-the-fly symbolic model checking," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 4(4), pp. 496–504, August 2003.

[118] P. K. Nalla, R. J. Weiss, J. Ruf, T. Kropf, and W. Rosenstiel, "Parallel bounded property checking with SymC," in *Modellierung und Verifikation*, ser. 8. GI/ITG/GMM Workshop, W. Ecker, Ed., 2005.

[119] T. R. Shiple, R. Hojati, A. L. Sangiovanni-Vincentelli, and R. K. Brayton, "Heuristic minimization of BDDs using don't cares," in *Design Automation Conference*, 1994, pp. 225–231. [Online]. Available: citeseer.ist.psu.edu/shiple94heuristic.html

[120] O. Coudert, C. Berthet, and J. Madre, "Verification of synchronous sequential machines using symbolic execution," in *International Workshop on Automatic Verification Methods for Finite State Systems*, ser. Lecture Notes in Computer Science, vol. 407. Grenoble, France: Springer-Verlag, June 1989, pp. 365–373.

[121] Y. Hong, P. A. Beerel, J. R. Burch, and K. L. McMillan, "Safe BDD minimization using don't cares," in *DAC '97: Proceedings of the 34th annual conference on Design automation*. New York, NY, USA: ACM, 1997, pp. 208–213.

[122] P. K. Nalla, R. J. Weiss, P. Peranandam, J. Ruf, T. Kropf, and W. Rosenstiel, "Distributed symbolic bounded property checking," in *4th International Workshop on Parallel and Distributed Methods in VerifiCation*, ser. Electronic Notes in Theoretical Computer Science. Elsevier, 2005.

[123] P. K. Nalla, P. M. Peranandam, J. Ruf, S. Lämmermann, J. Behrend, R. Weiss, T. Kropf, and W. Rosenstiel, "Fast distributed property checking," in *University Booth at Design Automation Test in Europe (DATE06)*, Munich, Germany, 2006.

[124] "IBM Solutions Grid for Business Partners: Helping IBM Business Partners to Grid-enable applications for the next phase of e-business on demand." [Online]. Available: http://www-304.ibm.com/jct09002c/isv/marketing/emerging/grid_wp.pdf

[125] P. K. Nalla, J. Behrend, P. M. Peranandam, J. Ruf, T. Kropf, and W. Rosenstiel, "Grid based fast falsification for bounded property checking," *Forum on Specification and Design Languages (FDL'07)*, September 2007.

[126] C. I.Foster, *The GRID: Blueprint for a new Computing Infrastructure*. Morgan Kaufmann, 2003.

[127] "MPI Specification." [Online]. Available: http://www.mpi-forum.org/docs/

[128] T. Grundmann, M. Ritt, and W. Rosenstiel, "TPO++: An object-oriented message-passing library in c++," in *2000 International Conference on Parallel Processing*. IEEE Computer Society, 2000, pp. 43–50.

[129] V. Huber, "UNICORE: A grid computing enviroment for distributed and parallel computing," *Proceedings of 6th Internatioanl Conference on Parallel Computing*, 2001.

[130] "Sun SGE Homepage." [Online]. Available: http://gridengine.sunsource.net/

[131] "MPICH2 Homepage." [Online]. Available: http://www-unix.mcs.anl.gov/mpi/mpich2/

[132] "ISCAS89 benchmarks." [Online]. Available: http://www.ece.vt.edu/mhsiao/iscas89.html

[133] "IBM benchmarks." [Online]. Available: http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/benchmarks.html

[134] J. Ruf, R. J. Weiss, T. Kropf, and W. Rosenstiel, "Modeling and formal verification of production automation systems," in *Integration of Software Specification Techniques for Applications in Engineering*, ser. Lecture Notes in Computer Science, E. et. al., Ed. Springer, September 2004, vol. 3147, pp. 541–566.

[135] I. Beer, S. Ben-David, C. Eisner, and A. Landver, "Rulebase: An industry-oriented formal verification tool," in *Design Automation Conference*, 1996, pp. 655–660. [Online]. Available: citeseer.ist.psu.edu/beer96rulebase.html

[136] I. Beer, S. Ben-David, and A. Landver, "On-the-fly model checking of rctl formulas," in *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*. London, UK: Springer-Verlag, 1998, pp. 184–194.

[137] "NuSMV: a new symbolic model checker." [Online]. Available: http://nusmv.irst.itc.it/

[138] XEN, "XEN Homepage." [Online]. Available: http://www.cl.cam.ac.uk/research/srg/netos/xen/

[139] W. Ecker, Ed., *Modellierung und Verifikation*, ser. 8. GI/ITG/GMM Workshop, 2005.

[140] *Proceedings of PDMC 2002*, ser. Electronic Notes in Theoretical Computer Science. Elsevier, 2002.

[141] E. A. Emerson and A. P. Sistla, Eds., *Proceedings of CAV 2000*, ser. Lecture Notes in Computer Science, vol. 1855. Springer Verlag, 2000.

Index

AR-automata, 26

BDD, 14
BDD Minimization, 70
BDD Transmission, 111
Black Box Verification, 30
Blocking Communication, 107
Boolean Functions, 12
Bounded Model Checking, 6

Cofactor, 13
CommWorld, 107
Constrain, 71
Cross over States, 75
CTL, 19
CTL*, 18

Distributed Verification, 8
Dynamic Helper, 77
Dynamic Overlap Reduction, 66

Eager Decomposition, 45
Early Quantification Tree, 29
Empty Window, 77
Enumerative Model Checking, 5
Equivalence Checking, 4
Explicit State Model Checking, 5

Finite State Machine, 14
Fixed transitions, 32
FLTL, 24
Formal Verification, 3

Generalized Cofactor, 13
Generative Disjunctive Decomposition, 44
Grid Computing, 90
Guiding, 46

Hybrid Distributed Algorithm, 77

Image Computation, 22
Influence Lookaheads, 64

Load Balancing, 10, 69
Logic Cone Reduction, 29
LTL, 20

Marshalling, 109
Mealy State Machine, 14
Message Passing Interface (MPI), 106
Minimal Overlap, 45, 63
Minterms, 13, 16
Model Checking, 4
Moore State Machine, 14

Non-Blocking Communication, 108

Partitioned-ROBDDs, 16
Partitioning, 29
Possible transitions, 32
Pre-image Computation, 22

Realizability, 31
Restrict, 71

Satisfiability Solvers, 6
Shannon Expansion, 13
Simulation, 2
State Explosion, 7
State Set Overlap, 59
Static Helper, 77
Support Set, 12
Symbolic Model Checking, 5
Symbolic Simulation, 5
SymC - Bounded Property Verification Tool, 24
Synchronization, 10

Temporal Logic, 18
Theorem Proving, 4
TPO++, 106

Transition Relation, 21

UNICORE, 114
Unrealizability, 103

Validity, 31
Variable Disjunctive Decomposition, 44
Verification, 1

Window States, 75
Windowing, 74

X-simulation, 5