# Efficient System Traversal and Property Verification by Exploiting Circuit Locality

**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. Prakash Mohan Peranandam**
aus Dindugal, India

**Tübingen**
**2006**

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

*To Appa and Amma*

# Acknowledgements

Interdependence is a higher value than independence. - Stephen R. Covey.

It is indeed hard to justice and acknowledge the help and contribution of all the people, without whose encouragement and support this thesis would have never seen the light of the day. All those whom I have had the pleasure of interacting with during my course of my education have enriched my life in so many ways and made it so worthwhile. I feel a deep sense of gratitude:

- To my advisors, Dr. Jürgen Ruf, Dr. Thomas Kropf and Prof. Wolfgang Rosenstiel. To the extent that the work in this thesis is worthy of credit, Dr. Jürgen Ruf deserves a large portion of that credit. Since his work is the foundation for almost every thing in it. I sincerely thank him for his patience and his help that gave me the insight of verification and for the knowledge he gave me by regular discussion and comments. Dr. Thomas Kropf, highly energetic and enthusiastic person, without whom I would not have had my motivation for this work. I always admired his skills of putting stuffs in a comprehensive way and have a overall picture, which guided me in the right path. His simple but highly motivating words gave me the confidence to pursue my own ideas. Prof. Rosenstiel, the back bone of our group and my work has always been the pace inducer. Conversation with him has been always a great pleasure and a new opening for ideas. Of course, I thank the funding agency D.F.G. for sponsoring this research work.

- To my colleagues, Dr. Roland Weiss, Pradeep K. Nalla, Djones Lettnin, Michael Schröder, Michael Bensch (Mike). I personally thank Roland for his worthful discussions, comments and his help that made my work more successful. I also thank Michael, Mike and Djones for their general research relevant talks and for being my good company in the faculty and outside that made my life easier. In particular, I thank Pradeep very much for his detailed and intense discussions on subjects, as it really helped me in many ways to improve my work.

- Of course, I have to thank my friends who kept me sane for the whole period of my work. To name some: Ashish, Balaji, Avijith, Leenus, Karthi, Kalyan, Boopathy, Sasi, Supriya and other Indian friends. I owe a special thanks to Stefanie and also to others: Adelhied, Arnold, Marta, Virginia, Natia, Lucia, Alvaro, Max, Kareena, Tanja, Georgio and many others. Last but not the least, my wife Kiruba and brother Karthikeyan.

**Abstract**

Bugs in hardware cost money. Whenever an error creeps into a design, time and money must be spent to locate the problem and correct it. With the growing complexity of digital systems, and the tremendous pressure for early-time-to-market schedules, the need for verification tools that can help designers to catch bugs at an early stage in the design process cannot be overemphasized. Statistically, verification takes nearly 75% of the design time.

Traditional methods of verification are empirical in nature and are based on extensive *simulation* of hand-written or automatically generated diagnostic test vectors. Although, provably effective in early stages of the debugging process, their effectiveness drops quickly along the maturity of the design. Hence, functional verification has problems in catching the corner cases, therefore formal verification complements it in order to cope up with the design growth.

In formal verification efficient search procedures are used to automatically determine whether the state space of a design satisfies an abstract logical specification (properties). In general, reachability analysis is known to be a key component of formal verification. For large classes of properties, verification can be reduced to reachability analysis. The term reachability analysis corresponds to the set of states that can be reached by means of traversal step from a set of defined initial states. The typical problem of such analysis is that after some steps of traversal, the system can not handle the overwhelming state space anymore and hence the memory overflow problem occurs. There is constant work on finding algorithms to reduce the memory requirement bottlenecks. One such approach to reduce the memory requirement is the symbolic representation of the state space, where Binary Decision Diagrams (BDDs) are used as data structures. Therefore, the state space traversal is then carried out symbolically by BDD manipulations.

However, large designs may require many millions of BDD nodes that also often causes memory overflow. This phenomenon is well known as BDD blow-up. There are several proposed solutions to deal with the large memory requirements of BDDs. One such novel proposal is to partition the BDD into two or more pieces and handle them separately for further state space traversal. Although there exists a few algorithms to partition the actual BDD, the lack of optimization in these algorithm are often responsible for the excessive growth of the BDDs resulting in slow verification or memory problems. This thesis deals with intelligent heuristics that optimizes the partitioning. The thesis contributes two heuristics, *MinOverlap* for the full validation approach and *Guiding* for a fast falsification approach.

To validate a design completely which is relatively large, a divide-and-conquer approach is state-of-the-art. It partitions the original BDDs that exceeds a threshold size of node count and deals with the partitions separately. The major problem of this methodology is revisiting of states in different partitions and is known as state space overlap that results in redundant computations. The *MinOverlap* algorithm aims at a reduction of state space overlap of different partitions that minimizes the redundant computation and decreases the verification time. The reduction is achieved automatically by means of collecting the information in the form of influence table of state variables by exploiting the design's transition relation.

Similarly, to verify a property faster, guiding by means of providing hints is state-of-the-art technique. However, this is not fully automatic and requires expert intervention. The *Guiding* algorithm in this thesis aims at steering the state space traversal algorithm in the direction of the target states automatically. It avoids the un-interesting state space saving memory and speeds up the finding of bugs. The guiding is realized by exploiting the property and the transition relation for information on target states, and utilizing them for the efficient guiding and steering the traversal automatically. The experimental results shows substantial speed up of the verification process by these heuristics.

**Zusammenfassung**

Hardwarefehler kosten Geld. Jedes Mal, wenn ein Fehler in einem Design auftaucht, müssen Zeit und Geld verschwendet werden, damit das Problem lokalisiert und korrigiert wird. Bei der wachsenden Komplexität digitaler Systeme und des riesigen early-time-to-market Drucks kann die Notwendigkeit von Verifikationsmitteln, die den Designern helfen, bugs im Frühstadium des Designprozesses festzustellen, genug betont werden. Verifikation benötigt laut Statistiken ungefähr 75% der Designzeit.

Die traditionellen Verifikationsmethoden sind empirischer Natur und stützen sich auf extensive Simulation handgeschriebener oder automatisch erzeugter diagnostischer Testvektoren. Obwohl ihre Effektivität in frühen Stadien des debugging Prozesses nachgewiesen ist, sinkt diese schnell sobald das Design weniger Fehler aufweist. Deswegen hat die funktionelle Verifikation Probleme bei der Feststellung von Randfällen und darum wird sie durch die formale Verifikation ergänzt, um die Designentwicklung zu bewältigen.

Bei der formalen Verifikation werden effiziente Suchprozesse verwendet, um automatisch zu bestimmen, ob der Zustandsraum eines Designs eine abstrakte logische Spezifikation erfüllt (properties). Im Allgemeinen ist die Erreichbarkeitsanalyse als ein Hauptbestandteil der formalen Verifikation bekannt. Für grö$\beta$ere Eigenschaftsklassen kann Verifikation auf Erreichbarkeitsanalyse reduziert werden. Die Erreichbarkeitsanalyse bezieht sich auf die Gruppe der Zustände, die durch einen Schritt aus einer Gruppe definierter Eingangszustände erreicht werden kann. Das typische Problem dieser Vorgehensweise ist, dass das System nach einigen Schritten den überwältigenden Zustandsraum nicht mehr bewältigen kann und folglich ein Speicherüberlauf eintritt. Es wird fortlaufend daran gearbeitet Algorithmen zu finden, um die Speicherengpässe zu reduzieren. Ein solcher Ansatz, die Speicheranforderungen zu reduzieren ist die symbolische Darstellung des Zustandsraums, in der Binary Decision Diagrams (BDD) als Datenstrukturen verwendet werden. Hierbei wird das Traversieren des Zustandsraums symbolisch durch Manipulieren von BDDs ausgeführt.

Gro$\beta$e Designs können jedoch viele Millionen von BDD-Knoten benötigen, was auch häufig einen Speicherüberlauf verursacht. Dieses Phänomen ist als BDD-blow-up bekannt. Es sind mehrere Lösungen vorgeschlagen worden, um die gro$\beta$en Speicheranforderungen von BDDs zu bewältigen. Ein solcher neuer Vorschlag ist, das BDD in zwei oder mehr Teile zu partitionieren und diese separat für die weitere Zustandsraumtraversierung zu bearbeiten. Obwohl Algorithmen zum Partitionieren des eigentlichen BDD existieren, sind diese oft nicht optimiert, und somit für die exzessive Zunahme der BDDs verantwortlich, welches eine langsame Verifikation oder Speicherprobleme nach sich zieht. Die vorliegende Doktorarbeit befasst sich mit intelligenten Heuristiken, die das Partitionieren optimieren. Die Dissertation trägt zwei Heuristiken bei, MinOverlap für den vollständigen Validierungsansatz und Guiding für einen schnellen Falsification Ansatz.

Um ein relativ gro$\beta$es Design vollständig zu validieren, ist eine divide-and-conquer Vorgehensweise Stand der Technik. Sie partitioniert Original-BDDs, deren Knotenzahl einen Schwellwert überschreiten, und bearbeitet die Partitionen

separat. Das Hauptproblem dieser Methodologie ist das wiederholte besuchen von Zuständen in unterschiedlichen Partitionen. Dies ist als Zustandsraumüberdeckung (state space overlap) bekannt und führt zu redundante Berechnungen. Der MinO-verlap - Algorithmus bezweckt eine Reduktion der Zustandsraumüber-deckung unterschiedlicher Partitionen, was die redundante Berechnung minimiert und die Verifikationszeit verringert. Die Reduktion wird automatisch erreicht, indem die Informationen in der Form von Einflusstabellen (influence tables) von Zustandsvariablen durch Auswerten der Transitionsrelation des Designs gesammelt werden.

Gleicherma$\beta$en ist die schnellere Verifikation einer Eigenschaft durch guiding Stand der Technik. Jedoch ist dies nicht vollautomatisch zu bewerkstelligen und benötigt Intervention. Der Guiding-Algorithmus in dieser Doktorarbeit bezweckt, den Zustandsraumtraversierungsalgorithmus automatisch in die Richtung der Zielzustände zu steuern. Er umgeht den uninteressanten Zustandsraum, wodurch Speicher gespart wird, und beschleunigt das Auffinden von Fehlern. Das Guiding wird umgesetzt, indem Informationen der Eigenschaft und der Transitionsrelation über Zielzustände ausgeschöpft und für effizientes Guiding angewendet werden, um die Traversierung automatisch zu steuern. Die experimentellen Ergebnisse zeigen eine beachtliche Beschleunigung des Verifikationsprozesses durch diese Heuristiken.

# Contents

# Chapter 1

# Introduction

This thesis presents new heuristics for the optimization of the formal verification of digital systems. This chapter outlines the motivation of the necessity for verification tools as an aid to the design of digital systems. It provides an overview of formal verification and the background on existing methods and it explains the need of optimization using intelligent heuristics. Finally, there is a note on the scope and the main contribution of the thesis.

## 1.1 Why Verification Tools?

The developments in electronic industry has witnessed continued growth since the first integrated digital circuit until the present where complex circuits are designed and fabricated at a faster rate than ever before. This increase in complexity was predicted in particular by Moore's law [1, 2], which states that the number of transistors on a chip would double every eighteen months. This prediction has roughly withstood the test of time for the last 35 years.

The boom of the electronics industry can be easily understood by our daily life dependency on electronic hardware systems, like the embedded systems inside automobiles, airplanes, cell phones, etc. The use of digital systems in daily living is expected to grow even more. Along with the growing design complexity, the business competitiveness also increased, putting tremendous pressure on early time–to–market. Its ubiquitous use coupled with the demand for rapid design requires that the digital systems function correctly in all possible scenarios.

Particularly safety critical applications like airplanes and automobiles require a heightened level of correctness, since faulty designs could cause total disaster. Apart from such safety critical systems, general electronics goods like MP3 players, DVD players, computers, digital cameras, etc. are also becoming common household items. Hence, these products also require early time–to–market and are expected to be bug–free, else the costs are often very high. The well known Pentium floating point division bug cost the Intel corporation [3] almost half a billion dollars [4], for example. A disk drive problem cost the Toshiba corporation [5] nearly a billion dollars. An earlier detection of these bugs would have saved the corporations from such losses. These real life examples illustrate the

importance of error proof design in today's age of high productivity.

The tools that enable the designers to work at higher levels of abstraction take care of the high productivity demand. The verification tools serve to aid the designer in catching bugs at an early stage in the design process and thus help to meet bug–free design.

## 1.2 Verification Methods

Verification methods for digital systems can be broadly classified as being either empirical or formal. The empirical method has been the traditional methodology used in industry and has only recently been complemented by the formal methodology.

### 1.2.1 Empirical Methods

Empirical verification methods generally verify the design by generating test cases for a design and applying those test cases to a model of the design by means of simulations. Empirical methods do not attempt to prove the correctness of a design, but rather aim to derive a level of confidence that the design is free of any obvious errors. The effectiveness of empirical methods depends directly on the effectiveness of the metrics used to grade the quality of the tests. Several coverage metrics [6] like line coverage, basic block coverage, toggle coverage, state coverage, tag coverage etc. have been proposed, but it is as of yet not clear which metric is the most effective in exposing design errors.

Simulation is the most widely used method for validation of models and is referred to as *functional verification* [7, 8] . The design to be tested is described in some modelling language and is referred to as the design under test (DUT). The design specification is then used to generate input and output test vectors. The generation of test vectors according to the specification and the handling of the output are defined to be the testbench. It can be modelled using the verification language, such as *e* [9] . The stimulus routine modelled in the verification language applies the input vectors to the models. The inputs are propagated through the model by the simulation tool and thereby generating the outputs. A monitor routine checks the output of the DUT against expected outputs for each input test vector. If a mismatch is found, the designer can use debugging tools to trace back and find the source of the problem. The problems arise from either incorrect design or incorrect timing. Once the problem source is identified, the designer can fix it and simulate the new model.

Empirical methods are provably effective in the very early stages of the debugging process, where multiple bugs are present. However their effectiveness drops along with increased design maturity, and requires substantially more time to uncover the hard to reach bugs widely referred as *corner case bugs* . This characteristic flaw has sparked an increased interest in the development of more formal methods.

### 1.2.2 Formal Methods

The term formal methods [10] refers to mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems. The phrase "mathematically rigorous" means that the specifications used in formal methods are well–formed statements that follow mathematical logic. The formal verifications are rigorous deductions in that logic, i.e., each step follows from a rule of inference and thus can be checked by a mechanical process. These techniques use mathematical formulations to verify designs and aim at establishing that an implementation satisfies a specification. The term implementation refers to a model of the design to be verified, while the term specification refers to a more abstract model or a property with respect to which the correctness is to be determined. The value of formal methods is that they provide a means to symbolically examine the entire state space of a digital design (hardware or software) and establish a correctness or safety property that is true for all possible inputs.

A formal model of the specific design with a precisely defined meaning allows one to apply mathematical proof techniques. Such a model can be expressed in a variety of mathematical formalisms. Examples of formalisms at the behavioural level are data flow graphs, process algebras and higher order logics. At the lower levels finite state machines and switch level models are often used. The design can be modelled directly by one of these formalisms. Alternatively a formal model can be constructed from a design description in a hardware description language.

## 1.3 What is Formal Verification?

Formal verification [11, 12] means proving that a property holds in a model of a design. The strength of verification lies in its ability to provide mathematical proof, in contrast to conventional simulation. The empirical results from simulations can tell us only that nothing went wrong on the specific cases. Of course, exhaustive attempts of every possible execution of a system is a valid proof. Formal verification can be regarded as having a similar effect as exhaustive simulation. However, specifying the property to prove and creating an accurate model of the design are difficult to achieve. One can ponder endlessly the philosophical impossibility of proving a system correct: Is the spec correct? Is the model accurate? Is the verifier correct? Is the computer we used to run the verifier correct? In [13] Cohn gives an insightful and readable analysis of the fundamental obstacles in proving a hardware design to be correct. Assuming that the model is indeed representative of the actual design, and the specification is a golden reference model, formal verification techniques can be applied.

In practice, we can choose the property to be one that "conforms exactly to a golden reference model which we assume is correct," otherwise we can simply prove easy-to-state properties that matter to us (e.g., absence of deadlock, interface lines follow a handshake protocol, etc.). Ideally, the model we verify is as close to the actual hardware as possible (e.g., a circuit extracted from the VLSI layout), but for complicated designs, an abstracted model is usually needed to

simplify the verification process. Although, formal verification of all kinds of properties including timing, performance, reliability, etc. is conceivable, most formal hardware verification research has focused on verifying functional correctness, which is also the focus of attention of this thesis.

## 1.4 Why Formal Hardware Verification?

People have researched formal verification of computer hardware and software for decades. Traditionally, the emphasis had been on approaching the idea of proving a system correct. The verification methods, typically a mathematically expressive but computationally undecidable logic with support from a semi automated theorem prover, require considerable time and expertise to verify even fairly simple systems. As a result, practical application has been limited to a few domains, such as security and safety-critical systems, where ethical or legal requirements demand the highest assurance of correctness, regardless of cost. For normal hardware design projects processes such as, hiring or training formal verification experts, delaying a product launch to allow time for formal verification, and reducing product performance or features to simplify formal analysis are all economically unacceptable.

Current industrial interest in formal verification places its emphasis on developing formal verification techniques that explicitly recognize economic demands. The other main focus is on the high design complexity and short design cycles that are straining current validation methods. Bugs cost money, especially the hard-to-find bugs that surface late in the design cycle. These bugs force an extra spin of silicon, delay a product launch, or require a massive product recall. Any technique that finds these bugs earlier is enormously valuable. So, instead of trying to certify correctness, formal verification is used as a powerful debugging tool. If the time and effort invested in formal verification is less than the time and effort saved by uncovering difficult bugs earlier, then formal verification is worthwhile, regardless of whether or not it allows us to make any claims about proving the system correct. This cost benefit comparison favours formal verification techniques that are automatic and easy-to-use, even if they lack theoretical expressiveness. Not coincidentally, this practical debugging emphasis for formal verification developed in parallel with new formal verification algorithms based on Binary decision diagram that offer far greater automation than had previously been possible. Binary decision diagram (BDD) is a data structure for representing a Boolean function . Bryant [14] introduced the BDD in its current form, although the general ideas have been around for quite some time.

## 1.5 Formal Verification Methods

Formal verification methods are often divided into two categories, theorem proving and model checking. A more comprehensive survey of formal verification methods can be found in [15].

### 1.5.1 Theorem Proving

Theorem proving, [16] also referred to as deductive verification, is an approach to verification where the verification problem is described as a theorem in a formal theory. A formal theory consists of, a language in which the formulas are written, a set of axioms and a set of inference rules. The inference rules are syntactic transformation rules for the formulas. With these rules and axioms, theorems can be proven. Hence, the verification by theorem proving makes use of deductive reasoning.

However, theorem proving is a time–consuming process that can be performed only by those who are educated in logical reasoning and have considerable expertise. This lack of automation makes its usage rare and limited to guaranteeing the correctness of safety critical systems and protocols. The prominent theorem proving tools are Isabelle [17], PVS [18, 19] and HOL [20].

The advantage of this method is that it gives one the ability to reason about infinite state systems [21], and enables one to check for complex correctness conditions in such large systems. Furthermore, theorem provers also support powerful techniques, such as proof by induction, and they allow the direct verification of parameterized designs without having to instantiate the parameters. However, there is no bound on the time or memory that may be needed to find a proof.

### 1.5.2 Model Checking

Model checking is a formal technique for property verification and is developed independently in the 1980's by Clarke and Emerson [22] and by Queille and Sifakis [23]. An efficient search procedure is used to check if a given finite state transition system is a model for the specification. The model is represented as a state transition system, which consists of a finite set of states, transitions between states and labels on each state. The state labels are atomic properties that hold true in that state. These atomic properties are expressed as Boolean expression of the state variables in the model. The property to be verified on the model is expressed as a temporal formula based on temporal logic [24]. The temporal formula is formed using state variables and time quantifiers like always or eventually. Hence, model checking is a method to check these properties of a design, where the properties are specified as temporal logic formulas [25].

For finite state models, this method can be fully automated. Model checking lets us verify that a state machine obeys a property, for example, that a one-hot encoded state machine is indeed one-hot encoded, that the machine is always resettable, that every request is eventually acknowledged, etc. Assertions [26, 27] that have been used for simulation can also be used as properties for model checking. The model checker works on the state transition system of the model and the given property and produces a result TRUE if the property holds in the model. If the property does not hold, the checker gives a counter–example to show that the property is violated. This feature of model checking is very helpful in debugging because it provides a ready made test case.

The idea behind model checking can be visualized by unrolling the transition

system. We start with the initial state and form an infinite tree (called the computation tree). Roughly speaking, the idea is to systematically explore the state space of a finite state machine in order to check that the given temporal logic formula holds on the machine. Temporal properties can by graphically visualized on this computation tree. The major problem with model checking is the state space explosion problem. The state transition system grows exponentially with the number of state variables. Therefore, memory for storing the state transition system becomes insufficient as the design size grows.

The invention of model checking was a theoretical breakthrough in the use of temporal logic for formal hardware verification. Temporal logic is a formal way of expressing properties that change over time. There are many different kinds of temporal logic; for brevity, we will only consider a few examples taken from one temporal logic called Computation Tree Logic (CTL) [28] which is one of the most popular logics for formal hardware verification with model checking. The basic idea is that we start with ordinary Boolean logic, and then add special temporal operators for describing future events. For example, in CTL, the operator AX means "for all possible input values, in the next clock cycle", the operator EX means "there exists an input such that in the next clock cycle", the operator AG means "for all possible input values, it will always be true that", the operator EF means "there exists a sequence of input values such that eventually", and so forth. The temporal operators can nest, so for example, AGEF(reset) says that it is always possible to find a path back to reset, and AG(req $\rightarrow$ AFack) says that every request is always eventually followed by an acknowledgment.

Symbolic model checking means using Binary decision diagrams (BDDs) as a data structure in the model checking algorithm. The algorithms used in symbolic model checking are a generalization of the reachability algorithm. The reachability algorithm is basically computing the set of all reachable states from a defined set of initial states. Using symbolic values in the reachability algorithm for the signals one can reach all possible next states from a present state set. One step ahead of this reachability algorithm is referred to as *image computation* and one step backwards is referred to as *pre–image computation*. In symbolic model checking pre–image computations are used to compute the set of all possible states the machine could have been in during the preceding clock cycle. Computing EX $Y_1$ is just a single pre–image computation, and computing EF $Y_1$ is just like the reachability iteration, except that we start with $Y_1$ and iterate with pre–image. The other CTL operators are computed similarly. In practice, symbolic model checking has limits due to the fact that the BDDs become too big.

### 1.5.3 Symbolic Simulation

Symbolic simulation [29] is a combination of conventional logic simulation with the BDDs. The advantage of a conventional logic simulator is accuracy. Detailed timing models, hazards, and oscillatory behaviour can all be simulated. The disadvantage of a conventional logic simulator is that only one simulation vector can be run at a time. In Figure 1.1, we would have had to run four simulations with the inputs equal to 00, 01, 10, and 11 to verify the circuit. A circuit with 20 inputs

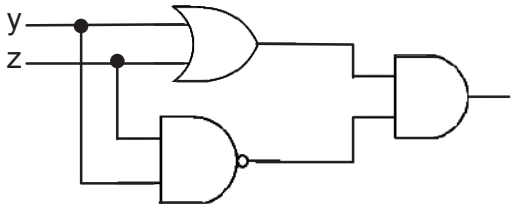


Figure 1.1: A simple XOR example.

would require over a million runs. Symbolic simulation adds two innovations to conventional logic simulation that give the effect of running large numbers of simulation vectors simultaneously. The first innovation is a third logic value $X$ that represents an unknown value. This value is propagated through the circuit just as the 0 and 1 logic values are, although the $X$ is always treated conservatively. For example, $0 \vee X$ is $X$, but $1 \vee X$ is 1, since 1 is a controlling value for OR. Setting an input to $X$ gives the effect of simulating the circuit for both the case where the input was 0 and the case where the input was 1, thereby cutting in half the number of required simulation runs. However, using the value $X$ results in a loss of information.

In Figure 1.1, setting one or both inputs to $X$ yields an $X$ at the output, a useless result for verification. The more important innovation is the introduction of symbolic values, which avoids the information loss from using $X$ values. The basic idea is to set an input to a symbolic value that can be either 0 or 1, rather than to a constant like 0, 1, or $X$. Alternatively, we can think of the symbolic value as remembering whether we assigned a 0 or 1 to a given input. Returning to Figure 1.1, suppose we set primary input $Y$ to 1 and primary input $Z$ to the symbolic value $a$. The symbolic simulator would then calculate that the OR gate will settle to 1 (since 1 OR anything is 1), that the NAND gate will settle to $!a$, and that the AND gate will settle to $!a$. Thus, we have effectively run two simulation vectors ($YZ$ equal to 10 and 11) at once, computing the output as a function of the symbolic values.

To implement this idea, a conventional logic simulator is modified to use BDDs to represent the values on wires as a function of the symbolic values. In practice, the user must trade off using explicit 0s and 1s, the $X$ value, and symbolic values. Setting an input to an explicit value gives conventional logic simulation. Setting an input to $X$ halves the required number of simulation runs, but loses information so the simulation result might not be useful. Setting an input to a symbolic value halves the required number of simulation runs and does not lose information, but makes the BDDs representing the values on the wires larger. Too many symbolic values will make these BDDs too large to build.

## 1.6 The Industries Choice

There are a number of requirements which a formal verification method must meet in order to be valuable in an industrial design environment. Eijk [30] provides a good overview of desirable parameters of a verification tool. In each of

the requirements mentioned below, model checking emerges as the more appropriate match (as compared to theorem proving) in an industrial setting.

- *Error Diagnosis*

  When an error is detected in a design description, the verification method must help the designer in locating the error. It should at least be able to produce a pattern of input stimuli that forms a counterexample for the property being verified.

- *Predictable Performance*

  A formal verification method must be able to handle designs of industrial complexity. The keywords are efficiency and predictability. Since formal verification methods are typically computationally expensive, it is difficult to meet the efficiency requirements. The performance of a formal verification tool must degrade gracefully with increasing design size. Small changes in the design should not have a major negative impact on the performance of the verification method. It is also important that the performance is predictable. Before a specific phase of a design project is started, it should be possible to predict whether a specific verification method will be able to handle the design or not. In this regard, neither theorem proving nor model checking are adequate, but there is growing evidence [31] of in–house model checkers being developed in most advanced semiconductor processor manufacturing companies.

- *Seamless Integration in the Design Flow*

  To make a verification method convenient to use, it is necessary to tightly integrate it into the design environment. It should be possible to use the same description for simulation and formal verification. A verification method should be able to handle the design styles used in the implementation, and also handle the hardware design description languages used to describe the designs and the cell libraries.

- *Automation*

  To minimize the required amount of user guidance, a formal verification method must provide a high degree of automation. Model checking is more amenable to automation than theorem proving, and its application requires no user supervision or expertise in mathematical disciplines such as logic and theorem proving.

  Note that the desire to have methods with a high degree of automation does not mean that a tool should not provide options for the user to guide the verification process. Even for fully automated tools, a small amount of user guidance can sometimes result in a significant increase in performance.

- *Verification Options – Full Validation and Fast Falsification*

  In the early stage of the design process where it is highly possible that the design might fail a specification, the designer or the verifier might require a

fast detection approach for identification of such error cases. The approach that does fast detection of errors is called *Fast Falsification* approach.

If the designer or the verifier intends to check the design for error freeness, then he/she requires an approach that checks the design state space thoroughly. The approach that does a complete search for errors is called *Full Validation* approach.

Therefore, model checking's support for automation, error diagnosis, full validation and fast falsification give it an advantage over theorem proving in an industrial setting.

## 1.7 Scope of the Thesis

This thesis demonstrates the implementation of a heuristic for optimizing a practical automatic verification tool. Verification tools are of many types ranging from explicit model checking to symbolic and SAT based to theorem proving. This thesis deals only with symbolic verification. The symbolic based approach differs in the way it is verifying the design against the specified property. Here we will see a symbolic verification tool that combines the symbolic simulation with the on–the–fly bounded property checking called *SymC* [32] detailed in Chapter 2. However, the applicability of this thesis results is not restricted to *SymC*.

This thesis covers the *SymC* tool that includes the verification algorithm based on simultaneous traversal of the design and the property. Hence, the property conversion to a traversable format, the symbolic state traversal and the terminating conditions have been discussed. The main topic of this thesis is the improvement of the algorithm to increase the capacity of the *SymC* tool by means of dividing the work load into two or more parts and handling them sequentially. The work load here is the design's state space traversal and dividing this into number of smaller pieces is generally referred to be divide–and–conquer approach. The divide–and–conquer approach can be applied for both full validation and fast falsification. The main contributions of the thesis are the intelligent heuristics for dividing the state space and traversing it efficiently in order to handle comparatively larger designs and thus finish the verification process faster.

Therefore, this thesis details the methods and the intelligent heuristics of dividing the state space and its related issues that are not yet addressed in state–of–the–art techniques. Finally, the experimental results proves the potential of the new heuristic in improving the *SymC* tool.

## 1.8 Contribution and Results of the Thesis

This thesis presents a new method to optimize the divide–and–conquer approach. Partitioning the one whole set of state space into a number of pieces and handling each subset sequentially is known as the divide–and–conquer method. In principle, the method of partitioning the state space and handling them can be of two different ways and is referred as *Splitting* and *Windowing*.

- ***Splitting:*** In short, this method of partitioning divides one whole set of states into two subsets and continues with the usual procedures of verification on these subsets sequentially. There are no special restrictions that have to be obeyed in future steps. The division is based on the *Shannon* expansion explained in Chapter 2.

- ***Windowing:*** In short, this method of partitioning divides one whole set of states into a pre–defined number of subsets. Each of the subsets is defined to be a window and has a restriction that has to be obeyed in all future steps. The windowing technique also continues with the usual verification procedures in a breadth first fashion on all subsets. Although, depth first approach is also possible, this thesis deals only with the breadth first. The partitioning by window technique is detailed in Chapter 2.

In both the techniques the problem of work duplication prevails in different forms. In other words, same states are visited in the different subsets, that leads to the redundancy computation. One part of this thesis basically concentrates on an algorithm that aims at minimizing this redundant computation in both the splitting and the windowing techniques and compares the result. Minimization of redundant computation basically improves the full validation approach that is briefed in the section 1.6. The other part of the thesis concentrates on the fast falsification approach (see section 1.6) by providing a heuristic for the splitting technique and comparing the results with the state–of–the–art fast falsification approach by windowing technique.

The heuristics and algorithms that are contributed by this thesis are listed below:

- **Static Overlap reduction – MinOverlap**

  This heuristic as a pre–processing step collects all the locality information from the transition relation. This information is then utilized during the partitioning process in order to minimize the redundant computation. This algorithm is successfully applied to both windowing and splitting techniques. This is an optimization method for the full validation approach. Further, this algorithm adopts a dynamic means of removing the already visited states.

- **Guiding**

  This heuristic is to steer the traversal in the direction of error state by means of some pre–processed information, hence making the whole verification faster by finding the bugs faster. This approach is an optimization for the fast falsification case. This heuristic uses two different methods to guide, one is to utilize the state variables and the other is to utilize the input variables for guiding.

In general the ideas in this thesis are optimization of both full validation and fast falsification approaches by the two partitioning techniques. These algorithms are evaluated on publicly available benchmark circuits from the ISCAS–89 suite

and the IBM benchmark suite. The orders of magnitude of improvement in the results obtained when compared with earlier state–of–the–art schemes are reported.

## 1.9 Overview of the Thesis

Chapter 2 introduces some preliminaries on logic manipulation. In particular, it explains BDDs, their utilization, and the basics of modelling. It also briefs the symbolic state traversal methods and its optimizations. This chapter also covers the temporal logic that is the base of the property specification.

Chapter 3 briefly reviews the related work in the context of partitioning and guiding. It introduces the symbolic bounded property verification tool *SymC*. It also explains the translation of the property to a special automaton called AR-automaton. It also analyzes the unaddressed problems that are covered in this thesis.

Chapter 4 details the heuristics with small examples and the pre–processing information collection and its ordering. Chapter 5 details the static overlap reduction algorithm and the dynamic overlap reduction. It also explains the adaptation of the algorithm in order to apply it to the windowing technique.

Chapter 6 explains the guiding algorithm and the two variants of the guiding approach. Chapter 7 demonstrates the practical usefulness of the new algorithms by means of experimental results obtained over the standard benchmark suits. Finally, the chapter 8 concludes the thesis with summary and possible future work.

# Chapter 2

# Preliminaries

Boolean logic forms the basis for computation in modern binary computer systems. In general any algorithm or any electronic circuit can be represented using Boolean functions . A Boolean function is an expression formed with binary variables and logical operators. Boolean algebra is the basic mathematical tool for reasoning about Boolean functions and hence about digital systems. This chapter introduces some basic definitions of Boolean functions. Binary decision diagrams (BDD), a well known data structure to represent a Boolean function is introduced. This chapter further elaborates on how digital systems can be modelled using BDDs and how they are manipulated to solve verification problems. Moreover, this chapter also discusses verification optimization concepts, which will be elaborated in the state–of–the–art chapter. Finally, there is a discussion of property specifications.

## 2.1 Boolean Function and BDDs

A Boolean function is a mapping from $\mathcal{B}^n \rightarrow \mathcal{B}$ with $n \geq 0$, where $\mathcal{B} = \{0, 1\}$. Each function can be expressed by a formula. A Boolean formula is an expression composed of Boolean variables connected by the Boolean connectors. A Boolean formula is a formula whose range are the discrete values {0,1}. A simple formula consists of one of the constants 0 (denoted as $\perp$) or 1 (denoted as $\top$) or it consists of a variable. Formally,

**Definition 1** *A Boolean formula is defined as a recursive expression with the following grammar:*

$$
\begin{array}{lll}
expr ::= \perp \mid \top \mid (expr) & & \\
\quad \mid \ < variable > & & \\
\quad \mid expr \text{ “} \vee \text{ “ } expr & \textit{(OR operator)} & \\
\quad \mid expr \text{ “} \wedge \text{ “ } expr & \textit{(AND operator)} & \\
\quad \mid \text{“!“ } expr & \textit{(NOT operator)} &
\end{array}
\tag{2.1}
$$

The above mentioned three basic binary operators AND, OR, NOT denoted as $\wedge$, $\vee$ and ! respectively are the basis for other common operations. Some example are shown in Table 2.1.

Table 2.1: Examples of Boolean formulas

| formula | Notation | Definition |
|---|---|---|
| exclusive-or (XOR) | $f \bigoplus g$ | $(f \wedge !g) \vee (!f \wedge g)$ |
| equivalence (XNOR) | $f \equiv g$ | $(f \wedge g) \vee (!f \wedge !g)$ |
| implication | $f \rightarrow g$ | $!f \vee g$ |

If all the variables in a formula are chosen from a set $X$, then the formula is said to be a function over $X$. The support of a formula $f$ is the set of all variables occurring in $f$, and is denoted by $supp(f)$.

Quantification ($\exists$ and $\forall$) and substitution ($\leftarrow$) are the major operation used for manipulation of boolean formulas and are shown in Table 2.2. The substitution denoted as $f(a \leftarrow g)$ means in function all the occurrence of the variable $a$ will be replaced by the variable $g$. The positive cofactor of a Boolean formula $f$ with respect to a variable $a$ is the formula that is obtained by replacing every occurrence of $a$ in $f$ by the constant 1, and is denoted by $f_a$. Similarly, the negative cofactor of $f$ with respect to $a$ is the formula obtained by replacing every occurrence of $a$ by the the constant 0, and is denoted by $f_{!a}$. Therefore, cofactoring of a Boolean function to a variable is assigning a truth value to that variable. From now, if a variable $a$ occurs in a function $f$ as $a$ then it is said to be occurring as positive cofactor in that function. Similarly, if the variable occurs as $!a$ then it is said to be occurring as negative cofactor in that function. The basis of BDD is Shannon expansion and this relies on the use of cofactors. The equation 2.2 holds for every Boolean formula and is known as Shannon expansion.

$$f = (a \wedge f_a) \vee (!a \wedge f_{!a}) \tag{2.2}$$

Cofactors are also used to define some common operations for quantifiers and substitution (Table 2.2).

Table 2.2: Quantifiers and substitution

| formula | Notation | Definition |
|---|---|---|
| existential quantification | $\exists a.f$ | $f_a \vee f_{!a}$ |
| universal quantification | $\forall a.f$ | $f_a \wedge f_{!a}$ |
| substitution | $f\,(a \leftarrow g)$ | $(g \wedge f_a) \vee (!g \wedge f_{!a})$ |

A Boolean formula is said to be a *tautology* if and only if all the interpretations or assignments lead to the constant 1. It is said to be *satisfiable* if at least one of the assignments leads to the constant 1. A *minterm* of a Boolean formula $f$ can be defined as a logical statement that consists of the truth value assignments for conjunction of all variables that belongs to $supp(f)$ such that it satisfies the whole formula. The notation $\|f\|$ dentes the number of minterms of the formula $f$.

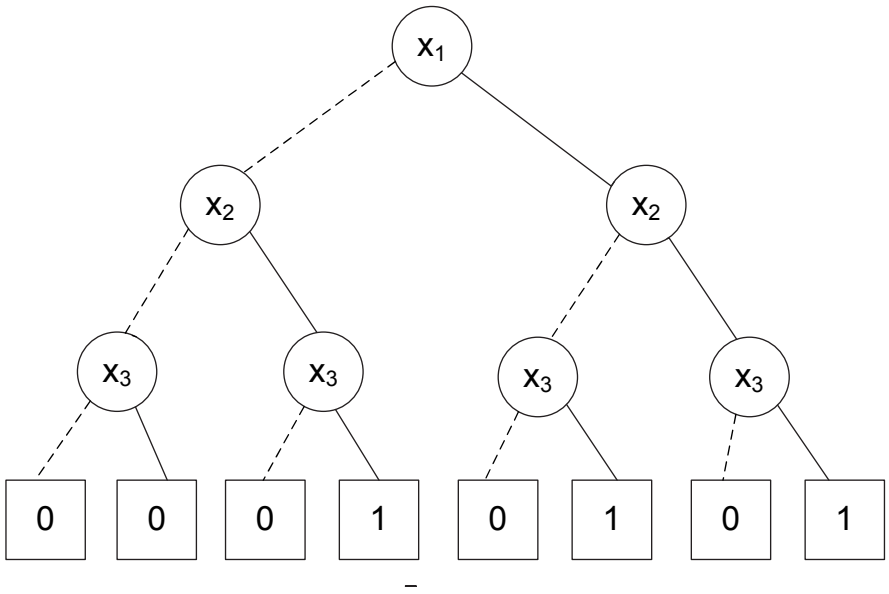


Figure 2.1: Representing Boolean functions

## 2.2 Binary Decision Diagrams

Binary Decision Diagrams (BDD) are a suitable data structure for representing binary formulas. Bryant [33] proposed this representation by imposing restrictions on the representation introduced by Lee [34] and Akers [35] such that the resulting form is canonical. BDDs are often substantially more compact than traditional representations such as truth tables, conjunctive normal form (CNF) and disjunctive normal form (DNF). Moreover, manipulation of BDDs are very efficient, hence, BDDs are used widely for variety of applications.

BDDs represent Boolean formulas as directed acyclic graphs, with internal vertices corresponding to the variables over which the function is defined and terminal vertices labelled by the function values 0 and 1. They form a canonical representation, making testing of functional properties such as satisfiability and equivalence straight forward. As an example, Figure 2.1 illustrates a BDD representation of the formula $f(x_1, x_2, x_3) = (x_1 \vee x_2) \wedge x_3$, for the special case where the graph is actually a tree. Each nonterminal vertex $v$ is labelled by a variable $var(v)$ and has arcs directed toward two children: $lo(v)$ (shown as a dashed line) corresponding to the case where the variable is assigned 0, and $hi(v)$ (shown as a solid line) corresponding to the case where the variable is assigned 1. Each terminal vertex is labelled 0 or 1. For a given assignment to the variables, the value yielded by the formula is determined by tracing a path from the root to a terminal vertex, following the branches indicated by the values assigned to the variables. The formula value is then given by the terminal vertex label. The memory requirements $\mid f \mid$ of the Boolean function $f$ is defined as the number of BDD vertices or nodes.

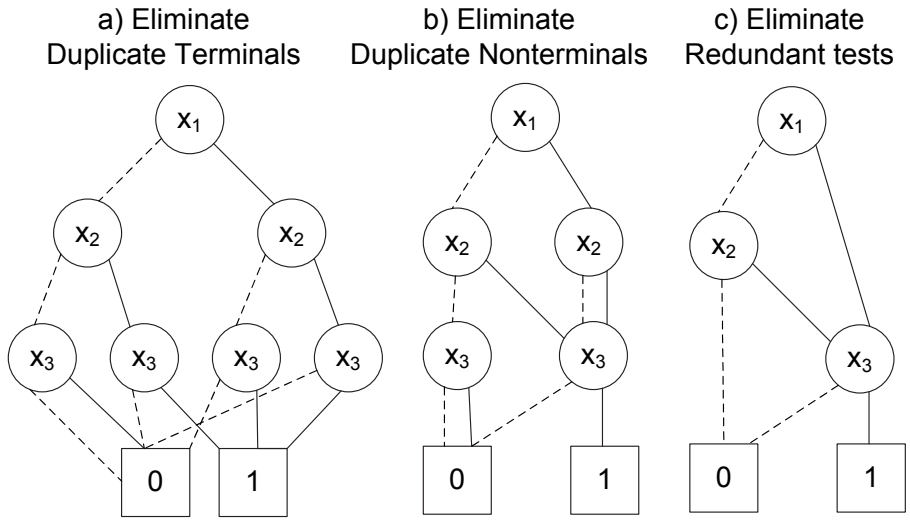


Figure 2.2: Transformation of BDD to ROBDD

### 2.2.1 Ordering and Reduction

An *Ordered* BDD (OBDD) , has a total ordering $<$ over the set of variables. For any vertex $u$, and either of the nonterminal children $v$, their respective variables must be ordered $var(u) < var(v)$. In the decision tree of Figure 2.1, for example, the variables are ordered $x_1 < x_2 < x_3$.

Furthermore, three transformation rules are defined over these graphs that do not alter the function represented, but result in more compact and canonical representations of the functions.

- **Remove Duplicate Terminals:** Eliminate all but one terminal vertex with a given label and redirect all arcs into the eliminated vertices to the remaining one.
- **Remove Duplicate Nonterminals:** If nonterminal vertices $u$ and $v$ have $var(u) = var(v)$, $lo(u) = lo(v)$, and $hi(u) = hi(v)$, then eliminate one of the two vertices and redirect all incoming arcs to the other vertex.
- **Remove Redundant Tests:** If nonterminal vertex $v$ has $lo(v) = hi(v)$, then eliminate $v$ and redirect all incoming arcs to $lo(v)$.

Starting with any BDD satisfying the ordering property, we can reduce its size by repeatedly applying the transformation rules. We use the term ROBDD to refer to a maximally reduced graph that obeys some ordering. For example, Figure 2.2 illustrates the decision tree shown in Figure 2.1. Note that on applying the first transformation, the number of terminal nodes are reduced from eight to two, and then the number of nonterminal vertices are reduced by two after the second transformation. On application of the third transformation rule another two vertices are eliminated. Since we will always be using this data structure in its ordered and reduced form, unless otherwise mentioned, henceforth we will use the term BDD to mean ROBDDs.

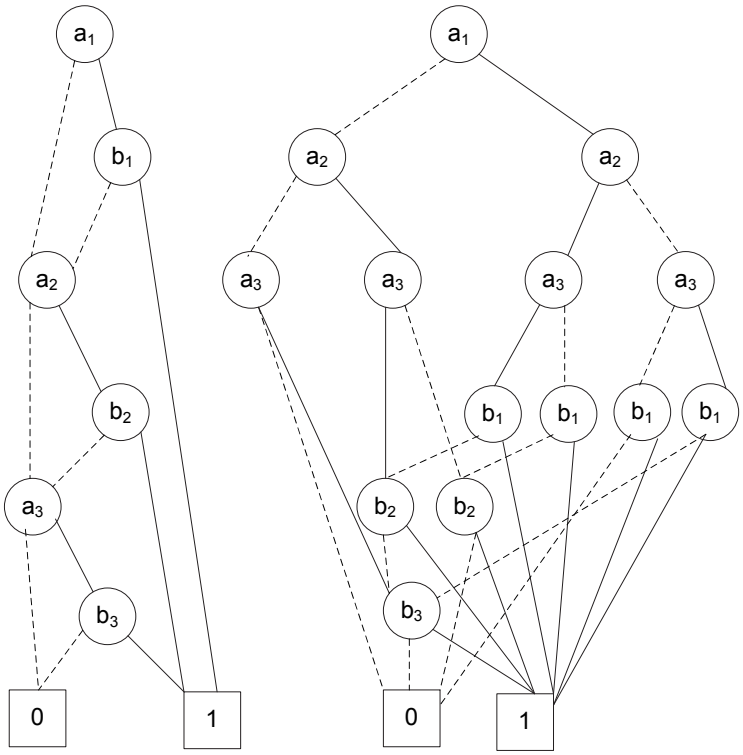


Figure 2.3: Effects of variable ordering

The resulting representation of a function is canonical, *i.e.*, for a given ordering two BDDs for the same function are isomorphic. This property has several important consequences. Functional equivalence can easily be tested. A function is satisfiable if and only if its BDD representation does not correspond to the single terminal vertex labelled 0. The tautological function must have the terminal vertex labelled only with 1 in its BDD representation. If a function is independent of variable $x$, then its BDD representation cannot contain any vertices labelled by x. Thus, once BDD representations of functions have been generated, many functional properties become easily testable.

### 2.2.2 Effects of Variable Ordering

The form and size of the BDD representing a function depends on the variable ordering. For example, Figure 2.3 shows two BDD representations of the function denoted by the Boolean expression $(a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3)$ [36]. For the case on the left, the variables are ordered $a_1 < b_1 < a_2 < b_2 < a_3 < b_3$ yielding a BDD with 8 nodes, while for the case on the right they are ordered $a_1 < a_2 < a_3 < b_1 < b_2 < b_3$ yielding a BDD with 16 nodes.

The difference of a factor of two in the previous example may not appear all that dramatic. However, for the more general case of $f = (a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee \ldots \vee (a_n \wedge b_n)$, it can be proved that the first variable ordering $a_1 < b_1 < a_2 < b_2 < \ldots < a_n < b_n$ yields a BDD with 2(n+1) vertices, whereas the other choice of variable ordering $a_1 < a_2 < \ldots < a_n < b_1 < \ldots < b_n$ yields a BDD with $2^{n+1}$ vertices [37].

For large values of $n$, the difference between the linear growth of the first order and the exponential growth of the second has a dramatic effect on the memory requirements and the efficiency of the manipulation algorithms.

Most applications using BDDs choose some ordering at the beginning and construct graphs for all relevant functions according to this order. This ordering is chosen manually or according to some heuristic guided analysis of the underlying functions in the design. For example, several heuristic methods have been devised that, given a logic gate network, often can derive a good ordering for variables representing the primary inputs [38, 39]. Note that these heuristic do not need to find the best possible ordering. As long as an ordering can be found that avoids an exponential growth, operations on BDDs remain reasonably efficient.

A desirable ordering heuristic referred as dynamic ordering approach is usually supported by BDD package. The dynamic approach changes the ordering during or after the BDD construction on demand only, i.e., only if a certain size limit is exceeded.

## 2.3 Modeling Sequential Hardware with BDDs

In order to be suitable for formal verification, the system needs to be formally modelled. The word *system* in this thesis scope means a synchronous digital system. The model should capture all the functionalities of the system such that correctness can be established. On the other hand, it should also abstract away those details that do not effect the correctness of the system functionalities. For example, when modelling a communication protocol the focus should be on the exchange of messages rather than the content of the messages. Considering the sequential reactive systems, which may need to interact with their environment frequently, they cannot adequately be modelled by their input/output behaviour. Therefore the other important feature that should be captured is *state*. A state is a snapshot description of the system that captures the values of the variables at a particular instant of time. Hence Finite State Machines (FSM) as defined in definition 2 are being used to formally model the sequential hardware.

**Definition 2** *A Finite State Machine (FSM) of Mealy type is defined as a 6-tuple,* $\mathcal{M} = (S, S_0, I, \lambda, T, O)$ *where*

- $S$ *is the set of states,*
- $S_0 \subseteq S$ *is the set of initial states,*
- $I$ *is the input alphabet,*
- $\lambda$ *is the output alphabet,*
- $T \subseteq S \times I \times S$ *is the transition relation between states,*
- $O \subseteq S \times I \times \lambda$ *is the output relation.*

The computation of a reactive system can be defined in terms of the transition relation of the FSM. To capture this intuition about the computation of reactive systems, a state transition graph called *Kripke structure* is being used. A Kripke structure consists of a set of states, a set of transitions between states, and a function that labels each state with a set of atomic propositions that are true in this state. Paths in a Kripke structure model the computations of the system. Although these models are very simple, they are sufficiently expressive to capture those aspects of temporal behaviour (see section 2.4) that are most important for verifying the functional correctness. The Kripke structure [12] can be formally defined as in definition 3.

**Definition 3** *Let $\mathcal{A}$ be a set of atomic propositions. A Kripke structure $\mathcal{K}$ over $A$ is defined as a 4-tuple, $\mathcal{K}$ = (S, $s_0$, T, L) where*

- *S is the set of states,*
- *$S_0 \subseteq S$ is the set of initial states,*
- *$T \subseteq S \times S$ is the transition relation that must be total, that is, for every state $s \in S$ there is a state $s^{'} \in S$ such that T($s, s^{'}$),*
- *L: $S \rightarrow 2^{\mathcal{A}}$ is the function that labels each state with a set of atomic propositions that are true in that state,*

The Kripke structure is a closed system that models all possible behaviours of the system. For a synchronous circuit the states are encoded by Boolean state variables, for example if the set of Boolean state variables is $E = \{e_1, \ldots, e_k\}$ then the set of possible states are $S = \{0, 1\}^k$. A set of states can be represented as a Boolean function $S(E)$ with BDDs. For each state variable $e_i$, there is a piece of combinational logic which determines its next truth value. Let $f_i$ be the next state function computed by this logic for variable $e_i$. Then the transition relation $S \times S$ can be derived by the next state function vector, $f = \{f_1, \ldots, f_k\}$. To represent the transition relation using BDDs, we require a second set of state variables $E^{'} = \{e_1^{'}, \ldots, e_k^{'}\}$ called next state variables. Then the transition relation of synchronous circuit is denoted as T(E,E') and it is defined as

$$\text{T}(E, E^{'}) = t_1(E, e_1^{'}) \wedge \ldots \wedge t_k(E, e_k^{'}) \text{ where}$$
$$t_i(E, e_i^{'}) = (e_i^{'} \equiv f_i(E))$$

This functional representation can be expressed by means of BDDs. This method of representing the formulas and functions using BDDs are referred as the *symbolic representation* . This method of modelling a synchronous circuit as BDD can be illustrated by using an example. The digital circuit in Figure 2.4 is a modulo-8 counter [40]. The set of state variables $E = \{e_1, e_2, e_3\}$ having $2^3$ possible assignments forming that number of *minterms* otherwise the number states. Let us use $E^{'} = \{e_1^{'}, e_2^{'}, e_3^{'}\}$ to denote the next state variables.

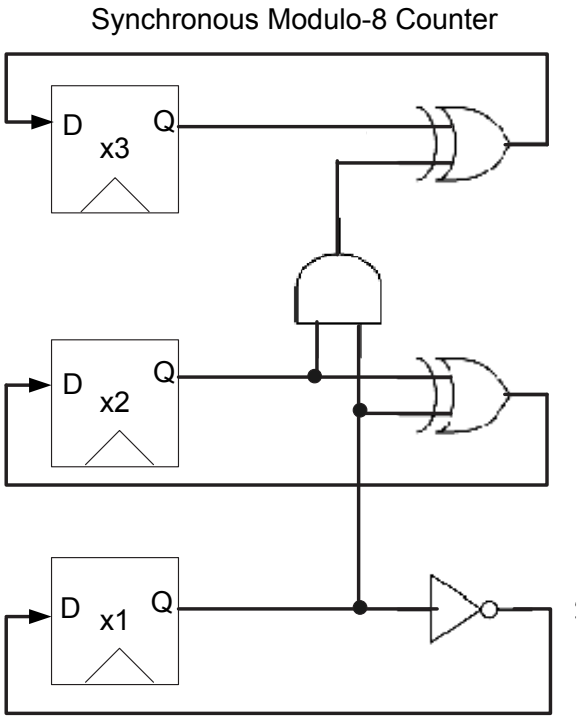


Figure 2.4: Synchronous modulo-8 counter

The next state function vector $f = \{f_1, f_2, f_3\}$ is given by

$$\begin{aligned} f_1 &= !e_1, \\ f_2 &= e_1 \oplus e_2, \\ f_3 &= (e_1 \wedge e_2) \oplus e_3. \end{aligned} \tag{2.3}$$

BDDs for these functions can be easily created. Assuming the initial state is encoded with all the state variables being 0, we get a BDD for the initial state by creating $!e_1 \wedge !e_2 \wedge !e_3$. Given a BDD for the individual next state functions $f_i$, it is straightforward to compute the BDD that represents the individual transition relation $t_i$ as follows:

$$\begin{aligned} t_1(E, e_1^{'}) &= (e_1^{'} \equiv !e_1), \\ t_2(E, e_2^{'}) &= (e_2^{'} \equiv e_1 \oplus e_2), \\ t_3(E, e_3^{'}) &= (e_3^{'} \equiv (e_1 \wedge e_2) \oplus e_3). \end{aligned} \tag{2.4}$$

Given a BDD for the individual transition relation $t_i$, it is straightforward to compute the BDD that represents the whole monolithic transition relation T as follows,

$$T(E, E^{'}) = t_1(E, e_1^{'}) \wedge t_2(E, e_2^{'}) \wedge t_3(E, e_3^{'}).$$

## 2.4 Temporal Logic

Temporal Logic is a formal specification language for use with a symbolic checker. A temporal logic is a logic augmented with temporal modalities to allow the specification of variables valuation orders in time, without having to introduce time explicitly. For example, a temporal logic with the modalities always and eventually will be able to specify the following property: "for all future moments in which $p$ holds there will be a future moment in which $q$ holds". Whereas traditional propositional logics can specify properties relating to the states of systems, a temporal logic is better suited to describe ongoing behaviour of nonterminating and interacting (reactive) systems.

There are two main kinds of temporal logics: linear and branching [41, 42]. In linear temporal logics (LTL) , each moment in time has a unique possible future, while in branching temporal logics, each moment in time may have several possible futures. In general, LTL can express properties of individual executions and the semantics are defined in terms of set of executions. Computation tree is a finitely branching infinite tree in which the traversal starts with the defined initial set of states $S_0$ and the successors are given by the $T$ of the Kripke structure (see Definition 3). The branching temporal logic can express properties of a computation tree, hence the name computation tree logic (CTL). CTL formulas can reason about many executions at once. The CTL semantics are defined in terms of states.

**Definition 4** *Let Vars = $\{a, b, c, \ldots\}$ be a finite set of distinct symbols, called the variable domain. Then the CTL syntax is defined as follows:*

$$\begin{aligned}
&\phi ::= v \mid !\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \\
&\qquad \mid AG\phi \mid AF\phi \mid A\,(\phi U \phi) \mid AX\phi \\
&\qquad \mid EG\phi \mid EF\phi \mid E\,(\phi U \phi) \mid EX\phi \\
&\textit{Where, } v \in \textit{Vars,} \\
&\qquad X \textit{ is neXt time operator,} \\
&\qquad F \textit{ is Eventually (Finally) operator,} \\
&\qquad G \textit{ is Globally operator,} \\
&\qquad U \textit{ is Until operator,} \\
&\qquad A \textit{ is All paths, the Universal quantifier,} \\
&\qquad E \textit{ is Exists a path, the Existential quantifier.}
\end{aligned} \tag{2.5}$$

**Definition 5** *Let $Vars = \{a, b, c, \ldots\}$ be a finite set of distinct symbols, called the vari-*

*able domain. Then the LTL syntax is defined as follows,*

$$\begin{aligned}
&\phi ::= v \mid !\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \\
&\qquad \mid G\phi \mid F\phi \mid \phi U \phi \mid X\phi \\
&\textit{Where, } v \in \textit{Vars,} \\
&\qquad X \textit{ is neXt time operator,} \\
&\qquad F \textit{ is Eventually (Finally) operator,} \\
&\qquad G \textit{ is Globally operator,} \\
&\qquad U \textit{ is Until operator.}
\end{aligned} \tag{2.6}$$

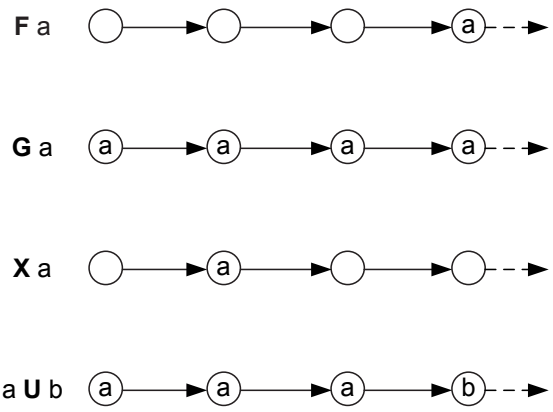


Figure 2.5: Semantics of LTL operators

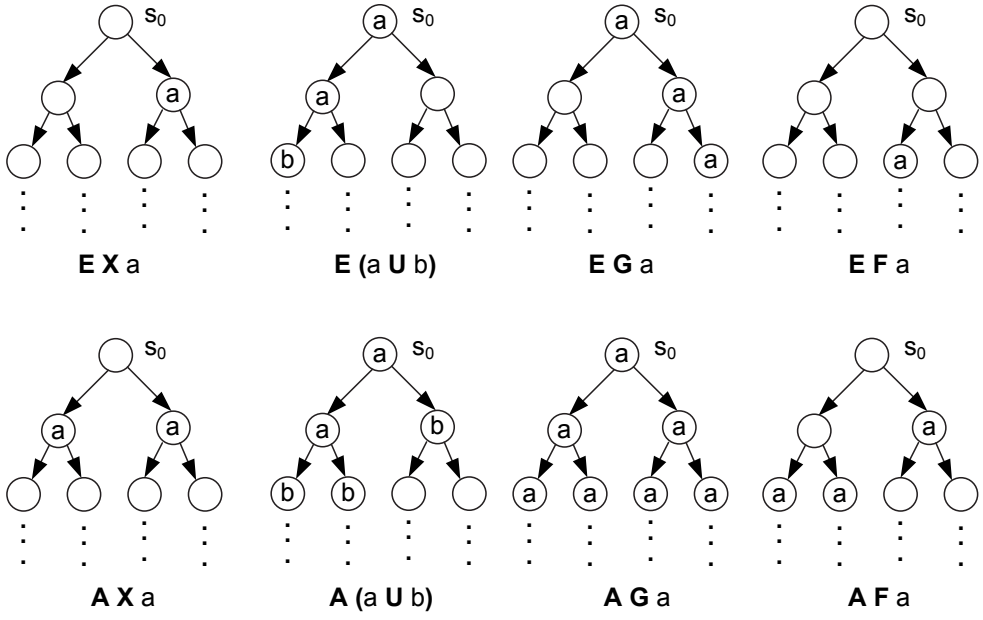


Figure 2.6: Semantics of CTL operators

LTL formulas are therefore interpreted over linear sequences and are regarded as specifying the behaviour of a single computation of a system as shown in see

Figure 2.5. CTL formulas, on the other hand, are interpreted over structures that can be viewed as trees, each describing the behaviour of the possible computations of a nondeterministic system as shown in Figure 2.6. In the sense of expressiveness both CTL and LTL can not be compared [43]. Figure 2.7 shows that CTL and LTL have a overlap of expressiveness, i.e. properties expressible in both logics.

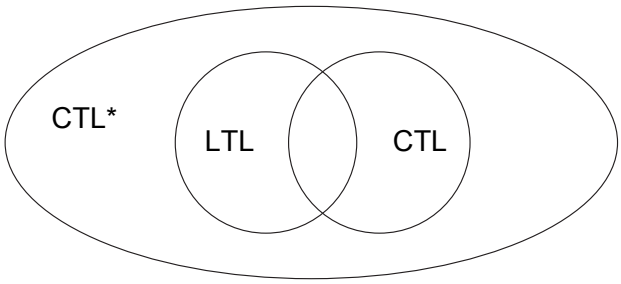


Figure 2.7: Expressiveness of CTL and LTL

For example, invariant properties like "$p$ never holds in any trace" can be expressed in both logics as shown below:

$$\text{AG !p} \quad \text{or} \quad \text{G !p}$$

The reactivity properties like "whenever $p$ happens, eventually $q$ will happen." can also be expressed in both logics as shown below,

$$\text{AG } (p \rightarrow \text{AF } q) \quad \text{or} \quad \text{G } (p \rightarrow \text{F } q)$$

As explained, CTL considers the whole computation tree whereas LTL only considers individual runs. Thus CTL allows to reason about the branching behaviour, considering multiple possible runs at once. For example, the CTL property "AG EF $p$" (reset property) is not expressible in LTL. An other instance is the CTL property "AF AX $p$" that distinguishes the two systems in Figure 2.8, but the LTL property "FX $p$" does not. The directed arrows represents the transitions and the state pointed by the arrow from environment is the initial state of the system shown in Figure 2.8

Even though CTL considers the whole computation tree, its state based semantics is subtly different from LTL. Thus, there are also properties expressible in LTL but not in CTL. For example, the LTL property "FG $p$" is not expressible in

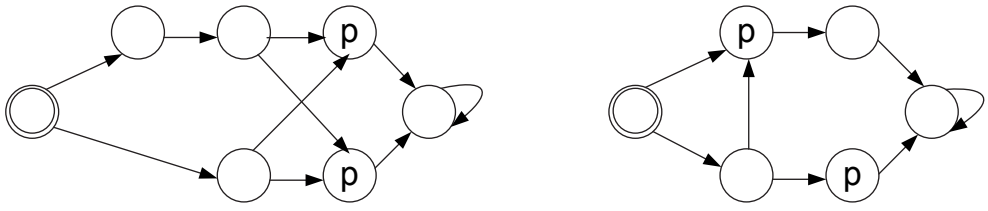


Figure 2.8: Expressiveness of CTL

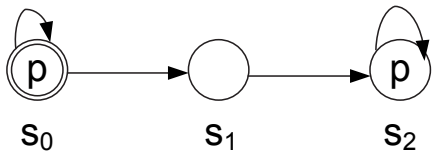


Figure 2.9: Expressiveness of LTL

CTL. The system in Figure 2.9 satisfies the LTL property but not the CTL property "AF AG $p$".

Though there is an overlap, each logic can express properties that the other cannot. Hence the expressiveness of CTL and LTL is incomparable in general. As shown in Figure 2.7, there is a logic called CTL* that combines the expressiveness of both CTL and LTL. However, we will not deal with CTL and CTL* as its out of this thesis scope. In this thesis we deal only with the LTL properties and its extension Finite LTL (FLTL) with the interpretation of CTL* quantification.

The logic FLTL [32] is used as a property description language. FLTL is an enrichment of pure LTL by time bounds which can be annotated to the temporal operators. Furthermore the formulas are interpreted over finite runs.

**Definition 6** *The syntax of FLTL is recursively defined over the variable domain:*

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

*with* $v \in$ *Vars,* $m \in \mathbb{N}$ *and* $n \in \mathbb{N} \cup \{\infty\}$

Since we use a simulation approach, we represent the changes of variables by traces:

**Definition 7** *A trace* $T[n..m]$ $(m \geq n)$ *is a mapping* $T : \{n, \ldots, m\} \rightarrow 2^{\text{Vars}}$*. 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 are extended. These extensions are used to define formally the semantics of FLTL over a three valued logic.

**Definition 8** *(Trace extension) Let* $T[0, m], T'[0, n]$ *be two traces with* $n > m$*.* $T'$ *is called a* trace extension *of* $T$*, if*

$$\text{for all } j \text{ with } 0 \leq j \leq m : T(j) = T'(j) \tag{2.7}$$

FLTL formulas are interpreted over traces. First we define the satisfiability relation over infinite traces:

**Definition 9** *The satisfiability relation* $\models_i \subset (\mathcal{T}^{\infty}, \text{LTL})$ *is defined recursively over the*

*structure of LTL formulas:*

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

Where $a$ is a propositional variable, $f$ is a LTL formula, $X, G, F$ are temporal operators and $m, n, i \in \mathbb{N}$. The standard temporal operators (F, G) are special cases of the timed operators by instantiating $m, n$ with $0$ and $\infty$, respectively. The semantics of FLTL is given by Definition 10.

**Definition 10** *Let $f$ be a LTL formula and $T \in \mathcal{T}^\infty$ be a trace. $T$ is said to satisfy $f$ (i.e. $T \models f$) if $T \models_0 f$.*

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

**Definition 11** *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**.

## 2.5 Symbolic State Traversal

Given the BDD for the initial state and the next state functions of a digital system, symbolic algorithms can compute the successors or predecessors. The successors are a set of next states that can be reached from the initial state in one step and is referred to as *image* computation. The predecessors are also a set of states from where the initial state can be reached in one step and this is referred as *pre-image* computation. This computation can be done repeatedly to collect all the reachable states and hence verification of the correctness of the system with regard to a property can be checked. The next subsection explains why traversal is done symbolically using BDDs.

### 2.5.1 Why Symbolic?

In the original implementation of the model checking algorithm [44], transition relations were represented explicitly. For systems with small number of processes, the number of states was usually fairly small, and the approach was often quite practical. However, the systems with large number of states were too big to handle. In [45] McMillan realized that by using a symbolic representation for the

state transition graphs, much larger systems could be verified. Many tasks in digital systems can be formulated in terms of operations over small, finite domains. By introducing a binary encoding of the elements in these domains, these problems can be further reduced to operations over Boolean values. Using a symbolic representation of Boolean functions, one can express a problem in a very general form. Solving this generalized problem by symbolic Boolean function manipulation then provides the solutions for a large number of specific problem instances. The new symbolic representation was based on Bryant's BDDs [46]. By further improvements in the BDD representation it became possible to verify some examples that had more than $10^{20}$ states [47]. Since then, various refinements of the BDD based techniques by other researchers have pushed the state count to more than $10^{120}$ [48, 49].

### 2.5.2 Image and Pre-image Computation

Verification of the correctness of a system can be achieved by repeatedly computing all the reachable states from the defined initial states, this computation is also called *reachability analysis* . In general *reachability analysis* is known to be a key component of formal verification. The term *reachability analysis* corresponds to the set of states that can be reached by means of traversal steps from a set of defined initial states. Computing the reachable states is done step by step by collecting the successors of the present state set at every step and replacing the present state set by the successors for next step. This one step traversal or successors collection is called *image computation*. In order to formally define the symbolic imge computation, we denote the BDD representing the current set of states $S_c$ by $S_c(E)$, where $E$ = { $e_1$, $e_2$, ..., $e_k$ } is the set of state variables.

$$Image(S_c(E), T) = \exists\, e_1, e_2, \ldots, e_k\, [S_c(E) \wedge T(E, E^{'})] \tag{2.8}$$

The replacement of present state variables by successor state variables is done by substitution:

$$Image(S_c(E), T)\ (e_1^{'}, e_2^{'}, \ldots, e_k^{'} \leftarrow e_1, e_2, \ldots, e_k) \tag{2.9}$$

Similarly, the pre-image computation traverses backwards by collecting the predecessors. The symbolic pre-image computation is defined as

$$Pre - image(S_c(E^{'}), T) = \exists\, e_1^{'}, e_2^{'}, \ldots, e_k^{'}\, [S_c(E^{'}) \wedge T(E^{'}, E)] \tag{2.10}$$

Figure 2.10 delineates the basic fix-point state space computation algorithm. Lines 1 and 2 initialize the variables with the initial state set. Lines from 5 to 8 is the fix-point loop. The new states that are reached by the image computation are added to the variable $fix_{new}$. At one point there will be no more new states added that ends the loop showing the reach of fix-point.

```
fix_new = initialstates;
present = initialstates;

do
  fix_old = fix_new;
  next = Image(present);
  present = next;
  fix_new = fix_old ∪ present;
while ( fix_new ≠ fix_old )
```

Figure 2.10: Pseudo code for fix-point iteration of state space traversal.

# Chapter 3

# State Of The Art

This chapter discusses briefly the progress in the development of the formal methods techniques and their optimizations for verifying complex hardware systems. The main disadvantage of model checking is the state explosion problem. In 1987 McMillan used Bryant's BDDs to represent state transition systems efficiently. The BDD based representation of the state space and the transitions are referred to as a symbolic technique and it increases the size of the systems that could be verified. Other promising approaches to alleviate the state explosion include the exploitation of partial order information [50], localization reduction [51], and semantic minimization [52] to eliminate unnecessary states from a system model. Most of these techniques are used in modern symbolic verification tools, in order to increase the capability of the tool to handle bigger size and make the verification faster and easier.

Although, symbolic and other techniques improved the traditional explicit model checker to handle comparatively large state space, there is still the problem of the state explosion prevailed beyond a limit but this time as a BDD explosion due to its symbolic nature. So the researchers re-engineered and altered the basic algorithm of the traditional model checker to be a symbolic forward traversal tool that verifies the property on the fly. These tools are referred with the name symbolic property checkers [32]. Some of these symbolic checkers are altered not to do the traditional fixed point iteration and hence to avoid the maintenance of the state space history, which in turn decreases the memory requirement of verification. These tools verify the property by different means, one among them is converting the property to an automaton, and traversing it in parallel along with the model and proving the correctness. The interesting point is that the optimizations that were developed for symbolic model checking also perfectly suit and improve the symbolic property checkers.

However, the present industrial large design requirement demands more than what these tools can provide. In other words, the traditional state traversal approach as shown in Figure 3.1 computes the image of the state space by constructing a monolithic transition relation. Construction of one whole monolithic transition relation BDD of a industrial size designs often causes BDD explosion. Moreover, the state space reached also grows very fast creating memory problems. In order to keep up the tools with their memory explosion problem for

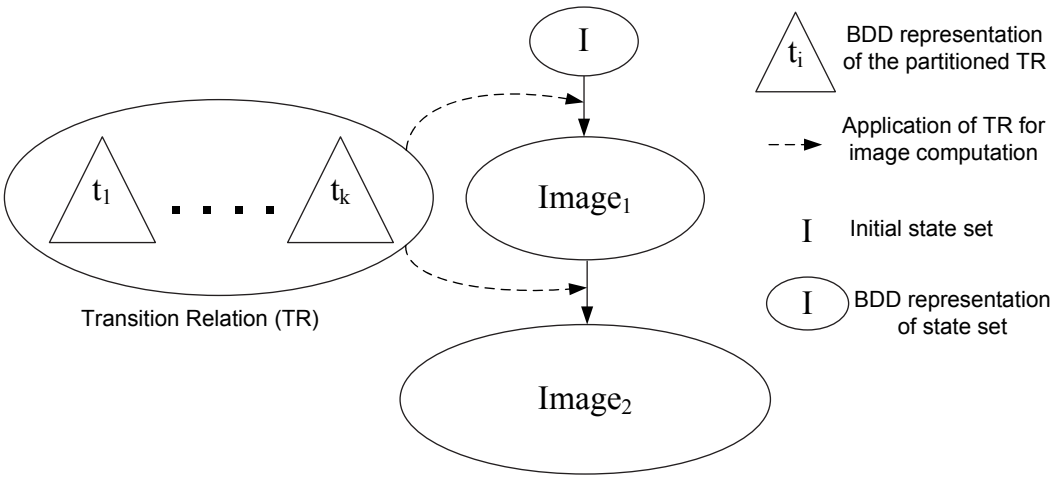


Figure 3.1: Monolithic image computation

large designs number of methodologies were proposed. These methodologies are briefed in the section 3.1. One of these methodologies is "divide–and–conquer" approach which is the main topic of this thesis and hence the sections 3.2 and 3.3 concentrate on the state–of–the–art techniques of this method. The un-addressed problems of the state–of–the–art methods and techniques are discussed in section 3.4. Later in section 3.5 the symbolic verification tool *SymC* is introduced which is used for implementing and experimenting the algorithms that addresses the problems mentioned in the section 3.4.

## 3.1 Improvements and Optimization Methodologies

There have been several improvements to formal techniques, particularly in symbolic verification. The below listed improvements and optimizations are very efficient and enhance the symbolic verification technique to handle much larger designs in comparison to what it could handle before.

1. **Partitioned Transition Relation:**

   Originally, only a monolithic transition relation was used. Unfortunately, for most designs the BDD representing the monolithic $T(E, E^{'})$ itself is often very big. In practice, each $t_i$ can be usually represented by a smaller BDD. However, the size of the BDD representing the entire transition relation may grow as the product of the sizes of the individual parts. This size might be too large to handle and exactly this was the limitation of symbolic verification until then. In [49] the authors introduced the method of keeping the parts separate, which are implicitly conjuncted. This new representation is referred to be the conjunctive partitioned transition relation.

   Image computation, as noted earlier, is the fundamental operation of symbolic verification. For a conjunctive partitioned transition relation, the im-

age computation equation 2.8 is then rewritten as follows:

$$Image(S(E), T) = \exists e_1, \ldots, e_k[S(E) \wedge (t_1(E, e_1^{'}) \wedge \ldots \wedge t_k(E, e_k^{'}))] \quad (3.1)$$

One major difficulty in computing the image without building the monolithic transition relation is that existential quantification does not distribute over conjunction. This problem was overcome by the method called early quantification [53, 54] and this technique is based on two observations. First, circuits exhibit locality, i.e, every $t_i(E, e_i{'})$ usually depends on only a small number of state variables. Second, although the conjunction does not commute with existential quantification, sub-formulas can be moved out of the scope of existential quantification if they do not depend on any of the variables being quantified. These techniques significantly increase the size of circuits that can be verified compared to the previous methods.

2. **Cone of Influence Reduction:**

   Cone of Influence is a very straightforward but effective state reduction technique. This reduction is the universal way of pruning away unrelated variables that increase the state space in vain. Before running a symbolic checker on the model, this technique finds the set of variables that can potentially affect the property, and removes all the other variables from the model. In other words, Cone of Influence reduction is based on the idea that if the property "$p$" refers only to variables $v \subseteq Vars$, then we can reduce the problem of checking that $p$ is true in model $M$ to the problem of checking that $p$ is true in a possibly much simpler model $N$ whose states give values only for a small subset of variables containing $v$.

   The dependency is computed by first taking the variables that directly occur in the property, adding to this set those variables that appear on the right hand side of the assignments to the variables already in the set, and doing this repeatedly until no new variables can be added. Removing this unnecessary variables, the state space of the model is reduced, and hence the BDD nodes required. This enables the model checker to verify designs faster and handle bigger designs.

3. **Divide–and–Conquer:**

   In principle, there are two different techniques of divide–and–conquer approach. One is *Windowing* technique and the other is *Splitting* technique.

   - **Windowing**
     In [55] the authors have defined a method to divide the Boolean state space into pre-defined "$k$" partitions and represent a function over each partition as a separate BDD. Further they showed that these partitions can be exponentially more compact than monolithic BDDs. Based on this partitioned ordered BDDs (POBDDs) further developments were presented which are discussed in depth in the section 3.2.

This method is mainly to avoid or reduce the memory explosion in symbolic model checking by utilizing the partitioned-OBDDs (POBDDs). In [55] the authors introduced POBDDs that are compact, canonical and efficient for manipulation of Boolean functions rather than manipulating one big BDD. Generally, BDDs represent all the state sets and the transition relation in symbolic traversal. This representation can grow large if the set of states to be represented is big and this corresponds directly to more memory requirements. The memory requirements $\mid f \mid$ of a Boolean function $f$ is defined as the number of BDD nodes. In order to reduce the memory requirements one can partition a Boolean function into smaller parts, whose union is the whole set. The balancing condition i.e., maximum size of each partitions are defined over the memory requirement, for example if a Boolean function $f$ represented as BDD is partitioned into two function $f_1$ and $f_2$ then the balancing condition could be specified as $\mid f_i \mid \leq \frac{1}{2} \mid f \mid$. Any BDD can be divided into $n$ partitions and represent it as POBDDs and this is defined as

**Definition 12** *[Boolean function partitioning] [55] Given a Boolean function $f : B^n \rightarrow B$ defined over $k$ inputs $X_k = \{x_1, \ldots, x_k\}$, a POBDD representation $\mathcal{X}_f$ of $f$ is a set of $n$ function pairs $\mathcal{X}_f = \{(r_1, \widetilde{w}_1), \ldots, (r_n, \widetilde{w}_n)\}$ where $r_i : B^k \rightarrow B$ and $\widetilde{w}_i : B^k \rightarrow B$, for $1 \leq i \leq n$ are also defined as $X_k$ and satisfy the following conditions:*

- *$r_i$ and $\widetilde{w}_i$ are represented as BDDs with variable ordering $\pi_i$, for $1 \leq i \leq n$,*
- *$r_1 \vee r_2 \vee \ldots \vee r_n = 1$,*
- *$\widetilde{w}_i = r_i \wedge f$, for $1 \leq i \leq n$.*

The set $\{r_1, \ldots, r_k\}$ is denoted by $R$, and each $r_i$ is called a *window function*. The window function $r_i$ represents a part of the Boolean space over which $f$ is defined. Every pair $(r_i, \widetilde{w}_i)$ represents a partition of the function $f$. The states that make the condition $r_i$ true are the states that are defined to be the *owned* states of the window $\widetilde{w}_i$ and all the other states are defined to be the *non–owned states*. Hence, the window function is a *restriction* that the state space has to hold to be in that window. The traversal of this windowing technique is showed in the Figure 3.2 where the local or restricted image is computed for every window and the non–owned states that are reached from the owned states are computed and distributed. The non-owned states that are reached from the owned states are referred as *cross–over* states. Detailed explanation of the figure is given in section 3.2.1. In this thesis we see a slightly altered version of the partitioning, where $k = 2^m$ and $m \in \mathbb{N}$ partitions are recursively done as shown in the Figure 3.3 and the partitioning are based on the following definition.

**Definition 13** *Given a Boolean function $f : B^n \rightarrow B$, then Boolean function partitioning is defined as*

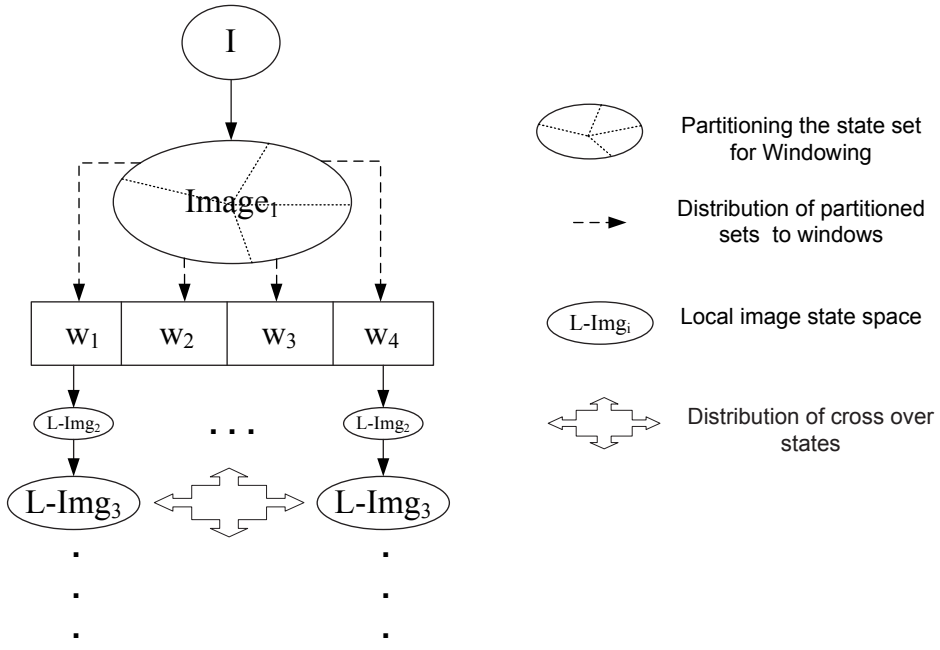


Figure 3.2: Image computation by windowing technique where $k$ = 4

$$f = f_1 \vee f_2 \ \textit{where} \ \ f_1 = v \wedge f_v \ \textit{and} \ \ f_2 = \bar{v} \wedge f_{\bar{v}}$$

The variable $v$ is called the splitting variable. This splitting variable defines the partitioning of the function $f$ into $f_1$ and $f_2$.

The Figure 3.3 shows the recursive partitioning for windows based on the definition 13. In order to relate the Figures 3.3 and 3.2, consider the state space named $S$ in Figure 3.3 and the state space named $Image_1$ in Figure 3.2 are the same for the partitioning step.

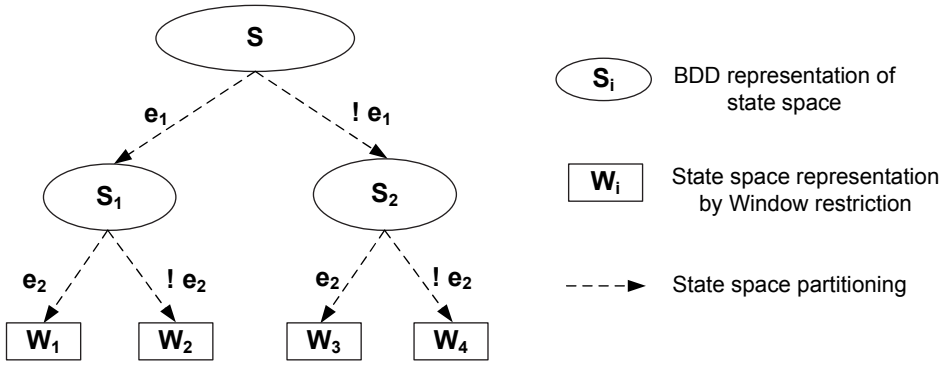


Figure 3.3: Recursive partitioning for windowing technique with $m$ = 2

- **Splitting**

  Splitting is also a POBDD, except that the partitions are always divided into two and the partition is based on the Shannon expansion (see equation 2.2) as shown in the Figure 3.4. The major difference of

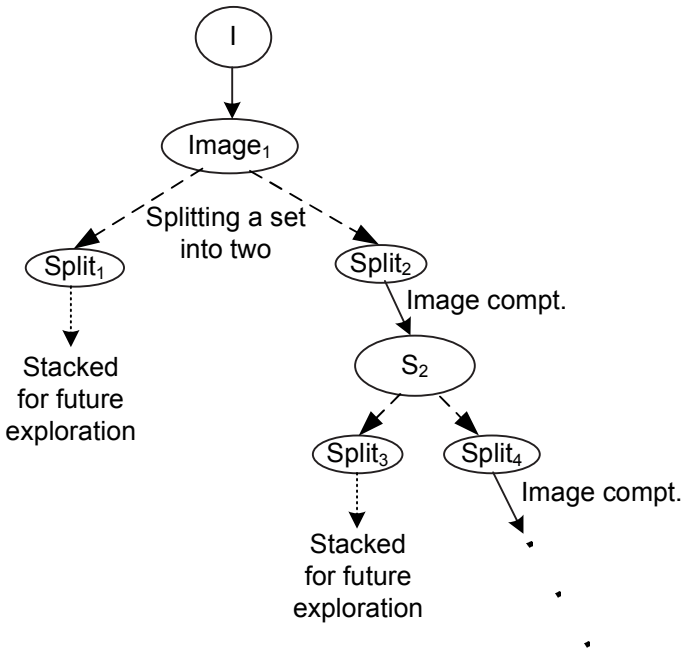


Figure 3.4: Image computation by splitting technique

splitting technique compared to the windowing is that the former only restricts the state space only once while partitioning where as the later applies the restriction in every step from the time of partition. Detailed explanation and comparison is given in section 3.2.1

Although, splitting and windowing techniques are comparable, only the windowing technique was highly concentrated and developed by researchers as the former suffers from the problem of redundant computation. The redundant computation is caused by the reaching already visited states and this phenomenon of revisiting states in different partitions is referred as the state space *overlap*.

However, this problem is avoided in windowing with a trade–off of some overheads. The state–of-the–art developments in windowing techniques are discussed in section 3.2. The section 3.4 addresses the redundant computation or overhead problems and discusses the necessary optimizations and other necessary improvements. Chapters 4 till 7 discusses in detail the new heuristics that addresses these problems and improvements in both splitting and windowing techniques and shows the benefits by experimental results.

4. Approximation and Guiding

   Nevertheless, the above divide–and–conquer technique is quite efficient in representing the function, research has been also in the direction of reducing the unnecessary state space from the model to be verified. This method prunes away some un-interesting state space and guides the traversal based on some heuristics or to approximate the model in order to accelerate the

traversal. However, the approximation and guiding can also be categorized under the divide–and–conquer approach with the assumption that the interesting set of states will be reached before all the partitions are explored. Therefore, in this thesis the approximation and guiding techniques or its combination are seen as divide–and–conquer techniques. The details of the state–of–the–art of this approximation and guiding techniques are discussed in the section 3.3.

## 3.2 Partitioned-BDDs (POBDD)

The idea of partitioning was used to discuss a function representation scheme called partitioned-BDDs in [56], which was then extensively developed in [55]. Partitioning of BDDs is enabled only when the BDD representing the state space reaches a certain threshold value, for example the memory limit. There are two variants of applying this POBDDs, one is partitioning a BDD into a pre-defined or dynamically decided number of parts known as *windowing techniques*(see Figure 3.2), where each part is referred to be a window. Every window is differentiated by unique combination of variables called restriction, and the state space belonging to that window should satisfy that restriction. This leads to the concept of owned and non-owned state space of a window, hence by this concept the state space overlap among the windows can be totally avoided. The second variant is to always partition a BDD into two and follow one while stacking the other called as *splitting* (see Figure 3.4). The splitting technique is dynamic by nature as the partitioned part can be re-splitted again into two if it grows beyond the threshold after some traversal steps. Splitting does not have any restriction as windows, hence there is no owned and non-owned state space to any splits.

The way the windows are handled in the windowing technique can make the technique to be a fast falsification or a full validation approach. In contrast, the partitioning methodology decides whether the splitting technique is a fast falsification or full validation approach. Although both the variants have their own advantages and disadvantages, many criteria are identical and hence tuning them improves both approaches.

### 3.2.1 Sequential POBDD

In [57] the authors presented a technique that is based on the windowing technique and partitioned transition relation. Each partition can have a different variable ordering and it requires only one partition to be in memory at any given time. This technique claims to be effective in total memory utilization by introducing the reachability analysis with the owned and non-owned states of a window. The reachability analysis introduced in this paper suggests a method to restrict the transition relation in order to keep the image state space of a window to be completely within that partition. In other words, transition relations are restricted in order to get only the owned next states. In Figure 3.2 the state space is represented by a BDD called $Image_1$, where it is partitioned into four subsets defined by the

windows $W_1 \ldots W_4$. The local image of a window is denoted by the $L - Image_i$, and the non–owned state space of any window have to be transported to the owner window. This transportation of state space is known to be the state space communication and it is denoted by the four direction arrow. Once the fixed point is reached within that window, then the non owned states are computed and passed or communicated (shown in dotted lines) to the other windows that it belongs by means of secondary storage.

This idea of reachability analysis by partitioning the state space and representing it as POBDDs was then realized in a CTL model checking tool in [58, 59]. These papers proposed the use of *dynamic repartition*, where the number of partitions can vary on the fly. Secondly, these papers also claim by theoretical evidence [60], that dynamic repartition can be exponentially more compact than an approach using the fixed constant number of partitions. The paper also describes a new algorithm for CTL model checking that aims at faster falsification, claiming to be the first algorithm to take full advantage of the ideas of POBDDs at an algorithmic level in the model checking. Thirdly, they devised a practical and competitive strategy to discover a path leading to an erroneous state otherwise called counter example generation. However, BDD approaches have a high sensitivity to parameter configuration, hence this paper also introduced a simple technique to handle this instability in BDD based verification by automatically selecting the best configuration for partitioning the state space from multiple short previews of the reachability computation. The above introduced techniques are studied more in detail and tuned in [61]. The basic problems that are addressed:

- If we must partition, what constitutes the axis of partitioning?
  e.g., what splitting variables should be used for creating windows?
- Is the static partitioning effective or is more partitioning required?
- Does local or temporary blow-up needs partitioning?
- How should the processing of partitions be prioritized?

In the Figure 3.4 the splitting technique is shown. Basically this approach splits the actual BDD into two and continues the image computation process using one BDD while stacking the other for future exploration. The advantage of splitting over windowing is that there are no restrictions on the partitions in future traversal steps as in windows, hence there is no owned and non-owned state. This avoids the expensive communication of state space to the owned ones, although this leads to a draw back of overlap of states in different splits. The following subsections discusses the common factors that are important for both the approaches.

#### 3.2.1.1 Splitting variable selection

The choice of splitting variables is critical to the effectiveness of the partitioning approach. In case of pre-defined partitioning, once a *blowup* is detected the goal is

to create small and relatively balanced $2^n$ partitioning windows (n = 1 for splitting approach) by selecting *n* splitting variables. The splitting variable is selected by means of a cost function, for instance, as described in [57]. For each variable, the cost function takes into account the relative BDD sizes of the positive and negative co-factors with respect to the BDD size of the original one.

#### 3.2.1.2 Dynamic and Local partitioning

Initially, the partitioning is done using one splitting variable. At this point, each new partition is checked to see whether the blow-up has subsided. If not, *dynamic* repartitioning is recursively performed on that partitions. A threshold on maximum number of partitions is kept to prohibit the method to produce exponential number of partitions.

During each step of the image computation, many steps of alternating composition and conjunction are performed. It is often found that the BDD size blow-up occurs during this intermediate steps, which is apparently a temporary phenomenon which eventually subsides by the time the image computation is completed. In such a case the invocation of dynamic repartitioning could create a large number of partitions, whose BDD sizes eventually become very small. Hence, it is advantageous to create these partitions *locally* only for that particular image computation and then recombine them before the end of the image computation. If local partitioning does not reduce the blow-up, then dynamic global repartitioning can be invoked.

The splitting approach is dynamic by nature, i.e, the splitting is done whenever the threshold is reached. Optionally this calling can be prohibited for the same reason of exponential partitions. In this thesis the windowing technique is handled with a pre-defined number of windows and not with the dynamic partitioning method.

#### 3.2.1.3 Partition ordering

The goal of ordering the partitions is to discover error states as early as possible in the state space traversal. The expectation is that the probability of catching an error is higher as more of the state space is covered. So the partitions are characterized in terms of time taken to cover the state space symbolically in that partition. This is measured in terms of a cost for processing the partitions. There are two different metrices that are used for assigning a scheduling cost, namely,

- Density based scheduling : The density , similar to [62], of a partition is defined as the ratio of the number of reachable states discovered in that partition to the size of the BDD representing the reachable states. In the interest of greater and faster state space coverage, it is better to first process partitions with a higher density.

$$Density = \frac{Number\ of\ Minterms}{Number\ of\ BDD\ nodes} \tag{3.2}$$

- Time based scheduling : Another useful metric is the time required for the least fixed point computation within each partition. The trail runs gives an idea of the time taken by the partitions. Therefore it is advantageous to select partitions that takes less time.

  The cost for processing a partition is given by,

$$\frac{Time\ taken\ for\ recent\ fixed\ point\ computation\ of\ a\ partition}{Density\ of\ that\ partition} \text{ (see equation 3.2)} \tag{3.3}$$

For the splitting approach, density based scheduling is done aiming at fast coverage of the state space. The fast coverage of the state space directly means higher chance of finding the error state.

### 3.2.2 Distributed POBDD

Although this distributed environment is out of the scope of this thesis, we briefly discuss the state-of-the-art. Section 3.2.1 explaind the optimization of the windowing techniques, but no parallelization done. The paper [63] presents a scalable method for parallel symbolic reachability analysis on a distributed memory environment where all the windows are handled in parallel. This paper also introduces a new cost function for finding the splitting variable taking the reduction and redundancy factors into account. This cost function aims at the balanced partitions of the state space $S$ among the distributed environment. The cost function is defined as,

**Definition 14** *The* cost *function [63] is defined as,*

$$cost(f, v, \alpha) = \alpha * \frac{Max(|f_v|, |f_{\overline{v}}|)}{|f|} + (1 - \alpha) * \frac{|f_v| + |f_{\overline{v}}|}{|f|} \tag{3.4}$$

*where $f$ is the BDD representing the state space $S$, $v$ is the variable for which cost is calculated and $\alpha$ is the adaptive value that decides the reduction factor.*

The $\frac{Max(|f_v|,|f_{\overline{v}}|}{|f|}$) factor gives an approximate measure to the reduction achieved by the partition. The $\frac{|f_v|+|f_{\overline{v}}|}{|f|}$ factor gives an approximate measure of the amount of sharing of BDD nodes between $f_v$ and $f_{\overline{v}}$ and therefore reflects the redundancy in the partition.

The cost function depends on a choice of $0 \leq \alpha \leq 1$. An $\alpha$ = 0 means that the cost function completely ignores the reduction factor, while $\alpha$ = 1 means that the cost function completely ignores the redundancy factor. The algorithm uses a novel approach in which $\alpha$ is adaptive and its value changes in each application of the partitioning algorithm, so that the following goals are achieved.

- The size of each split is below the given threshold $\delta$.
- Redundancy is kept as small as possible.

The actual algorithm initially attempts to find a BDD variable which only minimizes the redundancy factor ($\alpha$ = 0), while reducing the memory requirements below the threshold i.e., $Max(|f_v|, |f_{\overline{v}}|) \leq |f| - \delta$. If such a partitioning does not exist the algorithm increases $\alpha$ i.e., allows more redundancy, gradually until the condition is achieved. The balancing condition $\delta$ is given as in the equation 3.5 below,

$$\delta = \frac{\mid f \mid}{numberofpartitions} \tag{3.5}$$

The distributive methodology in [64] utilizes the network communication rather than the secondary storage devices in [63] in order to distribute the non-owned states to the corresponding network node. This methodology requires an additional network node called *coordinator* to do the network communication efficiently. The *coordinator* takes care of owned and non-owned state distribution, along with an on-the-fly memory balancing scheme among the network nodes. However, the splitting approach has also been implemented in a distributive environment and presented in [65].

Although this method is quite interesting concerning distributive computation, it is a static allocation method. In other words, it immediately splits the state space into as many partitions as the number of network nodes and does not free them until the verification terminates. Thus, static distributive method occupies all the network nodes throughout the verification time regardless of the actual need. This leads to an inefficient splitting because it partitions a relatively small BDD (early steps of traversal) into many small splits. Moreover, this method does not provide a means to overcome the memory overflow that occurs during an image computation or an exchange operation.

However, these problems are addressed in [66] by means of dynamic allocation and reallocation of network nodes. This paper introduces the concept of coordinators and workers. Each worker can be either active or free. It is initialized with one active worker that runs a symbolic verification algorithm, starting from the set of initial states. During its run, workers are allocated and freed, as needed. In the case of a high work load at one of the co-workers, this algorithm can simply split again. It may also happen that the memory requirement of few workers decreased below a certain threshold, then all of these workers are combined and given to one of them where all other are set free.

This feature provides this algorithm with strength and flexibility, and allows to reduce the splitting complexity. It is important to note that splitting occurs only "as needed", when a worker actually has a memory overflow. Thus the algorithm is "work-efficient", i.e., it exploits to the maximum the resources of the active workers before allocating additional ones. Moreover the paper claims the algorithm can effectively exploit any network size, thus, the larger the available network, the larger the systems that can be verified.

In any case all these windowing based distributive algorithms and advancements have a drawback of synchronization, i.e., the network nodes have to synchronize or wait for other nodes at some time point. In [67] the authors introduced the asynchronous distributive algorithm based on the splitting approach,

and this algorithm comes with the interesting side effect of automatic load balancing among the network nodes.

## 3.3 Accelerated Traversal

Splitting and windowing techniques are one way of approaching the BDD explosion problem. There are other effective methods like *Approximation* and *Guiding* in order to enable this BDD based techniques to cope with the explosion problem. In principle, approximation is the problem of extracting a smaller function from a given function. Guiding is a methodology to find an interesting path and slicing others by means of some heuristics. Basically, in this thesis guiding is used to prioritize the partitions to be followed, or to steer the state space traversal in right directions. This accelerated traversal in general is a key point which could increase the POBDDs capability of finding error states faster also called as fast falsification.

### 3.3.1 Approximation

Although, there is a reasonable number of techniques to optimize the verification process, one of the basic techniques is to utilize the optimization techniques of the under lying data structure, the BDDs. BDD approximation is the problem of deriving from a given BDD an another BDD smaller in size by BDD node count. If $\alpha(f)$ is the BDD produced by the application of approximation algorithm $\alpha$ to the BDD of $f$, then the minterms (states) represented by the approximated BDD is required to be either a subset ($\subset$) or superset ($\supset$) of the minterms represented by the input function, called *under-approximation* or *over-approximation* respectively, i.e,

$$\text{minterm}(\alpha(f)) \subset minterm(f) \text{ (under-approximation)}$$
$$\text{minterm}(\alpha(f)) \supset minterm(f) \text{ (over-approximation)}$$

A natural way to rank different approximations is by their density (see equation 3.2). High density corresponds to a concise representation, i.e., a relatively smaller BDD that can represent more number of states. This high density is of interest for the symbolic verification as smaller the BDD size usually faster the verification process. Moreover, the high density BDDs is not only small but it covers the majority of the state space and therefore it is desirable. In general, given a BDD representing a state space, the under-approximation is much suitable to find the concise representation. In [62] two algorithms for under-approximation have been proposed, namely,

- *Heavy-branch subsetting* : Determines how many minterms are in the function rooted at each internal node, and how many nodes would be eliminated by replacing arcs pointing to it.
- *Short-path subsetting* : It is based on the idea that short paths in a BDD correspond to large implicants of the function and use few nodes.

These two under-approximation techniques can be used in the sequential splitted traversal (without windowing) for faster falsification. This under-approximation provides a means of fast falsification as it generates a relatively concise BDD and in general traversing a smaller BDD is much faster. Moreover, as the BDD represents most of the actual state space, there is a higher chance of reaching the error state faster. However, it is not so attractive for parallel algorithms or for windowing as they do not produce a balanced partitions. Following this in [68] the authors introduced a safe under-approximation algorithm. This approximation can be directly compared to the density based scheduling explained in subsection 3.2.1.3.

### 3.3.2 Guiding

In practice, verification tools are usually more useful to find bugs than for proving properties. However, finding a bug that is hard to locate is difficult as large portion of the state space of the design actually satisfy the specification. Hence, the verification tool devotes much effort verifying correct portions of the design. This kind of behaviour of the verification tool usually leads to the BDD explosion problem, which could be easily avoided if the tool is provided with the knowledge of the state space partitions in which the probability of finding a bug is higher.

This approach has evolved into the idea of *guiding* the state space traversal. In [69] the main subject is about biasing the search towards reaching particular states. There were several heuristics discussed in it and all aim at enhancing the verification tool with the bug finding capability also called guiding. These heuristics basically optimize the verification process by searching the state space that is most likely to contain design flaws. There are four different heuristics proposed and these heuristics have been demonstrated to be very effective in guiding the search to find the violation of properties in designs. However some heuristics work better for some designs and others not. In general the authors out of their experience claim that the *Target enlargement* combined with *Tracks* and *Guideposts* heuristics can consistently find errors much faster than their individual performance or breadth first search. All of them are described below.

However, these guiding ideas are experientially tried on explicit model checking and are on the way to be adapted and used along with the POBDDs.

#### 3.3.2.1 Hamming distance

The first search heuristics uses the Hamming distance, which is defined as the number of bits that are different between two bit vectors [70]. Those states that have the lowest Hamming distance to the error states are explored first. In essence, it is believed the states with very few bits differing from the error states will require very few cycles to reach that target.

#### 3.3.2.2 Target Enlargement

It is an effort to make the set of states bigger that will violate the assertion, called error states, so that it can be found with less searching. The pre-image of the these error states is the set of states that in one cycle can reach an error state. If it is possible to reach a state in the pre-image from a start state, then it is also possible to reach an error state. Each successive pre-image potentially describes an even larger set of states that can reach the error states. The larger target increases the opportunity for the guided search to find a path to the error states and consequently reduces the amount of searching that needs to be done.

#### 3.3.2.3 Tracks

In practice, a subset of the state variables can control most of the behaviour of the design. A track computes a series of pre-images that are approximate of the actual pre-images based on a given set of variables that strongly control the behaviour of the system. Using multiple tracks implicitly conjoined, it may be possible to construct a sufficiently accurate pre-image that aids the guided search. Tracks computation are defined formally as,

$$Track_i^{k+1} = PreImage(T, Proj_{V_i}(Track_i^k))$$

Where $Track$ is one layer of one track, $i$ is the Track number, $V_i$ is the variables in Track $i$, and $k$ is the layer number, or number of cycles away from the largest enlarged target. $Track_i^0$ is the largest enlarged target. In essence, the next layer of a track is the pre-image of the projection of the last layer onto the variables in the track. Because all the variables that are not in the track are projected out, the resulting pre-image is always a superset of the actual pre-image. The projection also reduces the size of the BDD and thus enables computation of more pre-images.

During the guided search, the state's score depends on which layer the state belongs. The score is the least layer number in which all tracks contain the state. The score is the minimum cycle number that satisfies the implicit conjunction of all the tracks. Consequently, this evaluation function greedily chooses the layer number that is closest to the target. States that have the smallest score, or those that are closest to the error states are explored first.

#### 3.3.2.4 Guideposts

In addition to above mentioned automated heuristic, designers can also provide hints to the guided search. The hints, called *Guideposts*, are a series of conditions that the designer believe to be interesting or which are even required preconditions for the property to be violated. A guidepost encodes the number of hints that the search has gone through. The evaluation function of the guideposts is,

$$Score = (T_{Guideposts} - P_{Guideposts}) * M_{Cycles} + Score_{Tracks}$$

Where $T_{Guideposts}$ is the total number of guideposts, $P_{Guideposts}$ is the number of guideposts that the current state and its ancestors have past through. $M_{Cycles}$ is the maximum number of cycles in all the tracks. $Score_{Tracks}$ is the score from the Tracks evaluation function. This evaluation function biases the search towards going through the guideposts and passage of more guideposts should indicate a lower score. And the lower score is deemed to be a state that is closer to the target.

However, in [69] the author addressed the usage of hints or guideposts with the explicit state traversal. In [71], the authors addressed the usage of hints in symbolic traversal and also showed the way to identify good hints. Apart from pointing the way towards the target states, the hints used in symbolic guided traversal try to address the difficulties of image computation. This idea was then extended in [72] to allow nested fixpoints and to use hints to obtain over-approximation.

## 3.4 Unaddressed Problems

In the above sections we have seen recent research and developments in order to tackle the BDD explosion problem. Although the above mentioned methods and techniques are efficient, there are some interesting aspects that have not yet been addressed. These unaddressed factors are the topics of this thesis work. This thesis identifies important tuning and effective changes of the above mentioned works to improve their efficiency. Basically, the alterations of the heuristics are implemented in the variant of property verification tool called *SymC* which is a bounded symbolic property verifier. The aspects that are addressed:

- Reduction of redundant computation caused by overlap of state space
- Property based guiding of the POBDD based traversal to the interesting state

### 3.4.1 Reduction of State Overlap

In the section 3.2, we have discussed the advantage of using the POBDDs for efficient representation and traversal by both splitting and windowing techniques. In the splitting technique the state space is partitioned into two when the BDD representing it grows beyond a threshold limit which is defined as BDD node counts. The traversal will be continued with one of the partition where as the other will be stacked for future traversal if required. The traversal of the stacked state spaces is required in case of full validation of the property and errors are not detected in the earlier traversal. In this situation of full validation, traversal of the stacked state spaces might revisit some states that were already visited in the previous traversal as shown in the Figure 3.5. This phenomenon of revisiting the state space is known as state space overlap and this contributes to the redundant computation.

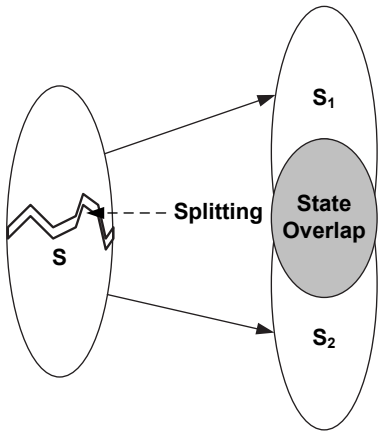


Figure 3.5: State space overlap due to traversal after splitting

The problem of minimizing the state space overlap in the splitting technique has not been addressed straight in any of the state–of–the–art techniques. However, this problem has been addressed indirectly by utilizing the windowing technique which does not allow the state overlap. But this benefit comes along with a trade off, i.e., additional overhead of distribution of non–owned states.

The partitioning of the state space based on windowing can be divided into $n$ number of windows for traversal. Each window is obliged to contain only its owned states and while traversal the non–owned states that are reached are then passed to its respective owner. The transformation or exchange of non–owned states by windows is handled in different styles aiming at an efficient methodology. However, every researcher stated this exchange of non–owned states is a bottleneck of the windowing methodology, independent of whether the windows are handled sequentially or in parallel. Moreover, the naive heuristics of windowing might lead to unbalanced partitions that results in an unequal window size. And if the cross–over states are not distributed at every step and are handled at last then in most cases many windows end up empty due to factor that all owned states reach to non–owned states.

The problem of minimizing the cross–over states and empty windows has not yet been addressed either. Therefore, part of this thesis work concentrates on providing intelligent heuristics for minimizing the state space overlap in the splitting technique. Also an on–the–fly method of partitioning for windowing technique to address the empty windows and cross-over reduction.

### 3.4.2 Guided Splitting

In the section 3.3, we have seen a couple of techniques to accelerate the traversal by means of external guiding (see Figure 3.6) or approximation. The acceleration of the traversal is due to the consideration of only a subset of the actual state space for traversal by introducing the incompleteness. Although, this method is incomplete it is a valid approach if it can reach the interesting state and can be categorized as the fast falsification approach as the traversal is made faster

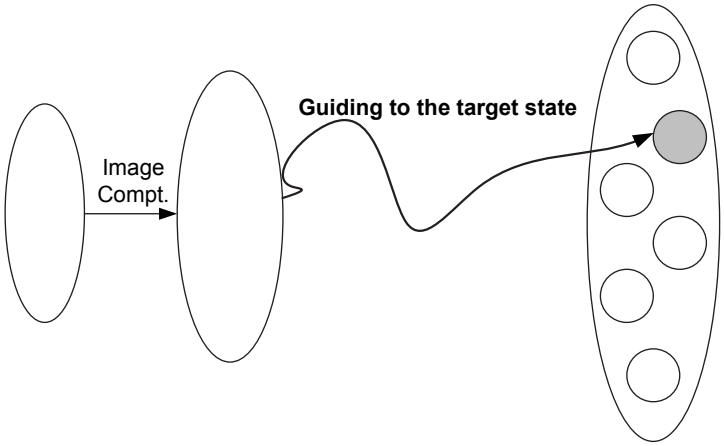


Figure 3.6: Guiding the traversal to the target state

by reducing the size of the state space. However, if this technique can not reach the interesting states then a back tracking mechanism is required to ensure completeness. Most of the techniques are addressing the explicit traversal, although the methodology using "hints" has been upgraded to symbolic traversal. In any case, the methodology of guiding by hints has a bigger disadvantage of requiring human intervention.

Hitherto, the guiding by means of hints needs the verification engineer to have a good knowledge about the design that is being verified in order to accelerate the traversal. And the approximation technique makes the traversal faster by avoiding some state space but it never steers the traversal to the interesting set of states.

There has not been any major work done in this area of guiding the traversal by means of fully automated heuristics. Therefore, the other part of this thesis contributes to this area of guiding by providing a heuristic that intelligently collects the set of important variables from the property and utilizes them for the automatic guiding of traversal. This guiding heuristics are further improved with the combination of approximation.

## 3.5 Symbolic Simulation and Verification

The optimizations and the improvements that were discussed above are incorporated and the unaddressed problems are answered using the symbolic bounded property verification tool called *SymC*. Typically the tool *SymC* takes a design and a couple of properties and proves their correctness. If the proof fails, a counter example may be generated. The major difference in the *SymC* [32] tool compared to model checking tools is that the former do not maintain the history of state space. The reason for avoiding this history is as it could grow beyond a manageable limit, making it a problem for verifying large designs. The history of the state space has to be maintained in order to calculate the fixed point, which is the foundation of the verification algorithm in the model checking where as in *SymC*

the verification is simultaneously while traversing.

Therefore, model checking is the matter of choice for verifying small or average sized modules on block level. In order to extend the model checking to the verification of whole systems or the larger modules researchers re-engineered the basic model checking algorithm aiming at avoiding the huge history. *SymC*, as mentioned, is one such approach where the verification is done by symbolic forward traversal [73].

*SymC* approach is a forward symbolic traversal where the verification is done in parallel by checking for the violation of a property. Although this advantage comes with a compromise of allowing bounded properties, it is not necessarily a disadvantage. Bounded properties are often useful to check the timing information, and makes the property writing exact. This also allows the design to explore only to the necessary point making it possible to verify comparatively larger designs. However, *SymC* also allows the usual LTL properties. That is the violation in case of the temporal operator *Always* or the acceptance in case of temporal operator *Eventually* can be verified.

*SymC* takes a property specification and a system description and translates both inputs into a symbolically simulatable representation. Therefore, the properties are translated to a special automata called AR-automaton. During the symbolic simulation the states of the AR-automaton are observed and *SymC* reports a success or a failure to the user. Trivially, the translation of the design can be avoided by providing the design in a symbolically simulatable representation. Figure 3.7 shows an overview of the *SymC* verification process. The following sections explains the *SymC* approach and it's basic algorithm.

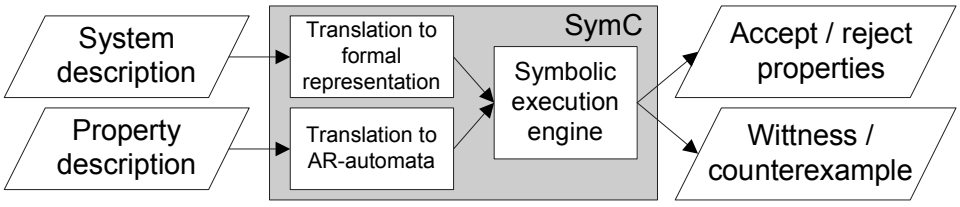


Figure 3.7: Overview of *SymC* verification process

## 3.6 Bounded Property Checker - SymC

*SymC* is a new formal verification technique that combines the property checking and symbolic simulation. It adapts the symbolic simulation to traverse the design and the property in parallel which turns out to be a verification methodology on the fly. Since it uses the same property specifications as simulation based approaches and model checking techniques it can seamlessly be integrated in the verification flow.

One symbolic simulation step corresponds to one image computation of the given state set. The image computation is the process of obtaining all possible next states from the given state set. This is shown in the Figure 3.8.

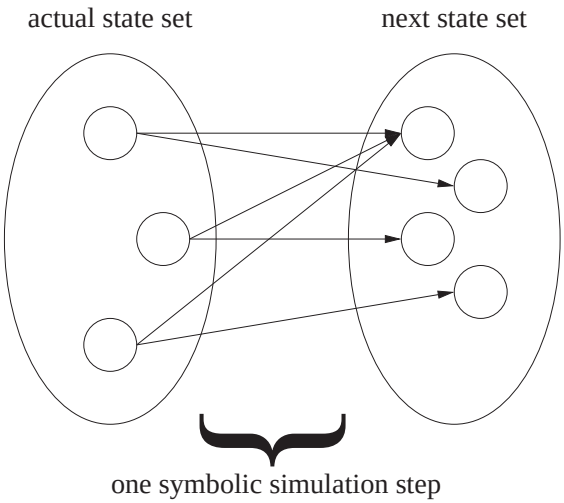


Figure 3.8: Symbolic Simulation step

The properties are given in the linear time temporal logic (LTL, [74]) which has been enriched by timing constraints (FLTL, finite linear time temporal logic) . The temporal logic formulas will be translated to a special finite state machines (AR- automata [75]) which can then be used in the symbolic simulation phase. The design can be modelled directly using the *SymC*'s typical input language. The translation of LTL / FLTL properties to AR-automata is briefed in section 3.7.

The algorithm shown in Figure 3.9 sketches the core of the symbolic simulation engine for one AR-automaton verification. In line 5 the simulation step is initialized and in line 9 the product machine of both the design and the AR-automaton is built. The lines 11 - 26 sketches the main loop of the *SymC* verification process that loops till the bound is reached or till the terminating condition is reached.

Line 15 shows the first step of image computation where only the AR-automaton image is computed, and this is done separately to reduce the intermediate BDD size and for an efficient traversal. The image of the AR-automaton is checked for the termination condition as showed in the lines 17 - 23. For universally quantified property *SymC* reports a failure if there is at least one reject state and reports a acceptance only if all the states are accepted. It is vice versa for the existentially quantified property.

The set "start" is the set of initial states to start with the simulation. *SymC* has different options how to choose this set:

- Starting only from a defined initial state set of the design. In this case, *SymC* checks only those states of the system which are reachable within the bound from the initial state set. Other states are not considered in the verification.
- Starting the simulation in all possible states of the design. This may include states which are probably not reachable from initial state and the algorithm may report a false negative result.
- Starting from known reachable states of the design. This means, the de-

```
// start is the initial state space to start the traversal
// AR − aut is a special automata equivalent to the property
// bound is the number of the simulation steps
symSimulate(in: start, AR − aut, bound)
 t = 0;

 // build product state of the system and the AR-Automaton

 act = start ∧ AR-aut.start;

 while ( t ≤ bound)

   // compute Accept-Reject-Automaton (AR-Automaton) image

   act = image_{AR−aut}(act);

   if (check universally)
     if ( act ∧ AR-aut.reject ≠ false ) reportFailure();
     if ( act ∧ AR-aut.accept = act ) reportAcceptance();

   if (check existentially)
     if ( act ∧ AR-aut.accept ≠ false ) reportAcceptance();
     if ( act ∧ AR-aut.reject = act ) reportFailure();

   act = image_T (act); // image computation
   t = t + 1;
```

Figure 3.9: Symbolic simulation loop of the design and the AR-automaton.

signer specifies an invariant condition defining a set of states which is known reachable. This approach lies in between the first and the second alternative.

The traversal starts with the initialization step where one of the ”start” set of the design is conjuncted with the initial state of the AR-automaton to form the product state. The product state is named as ”act” the actual state set. Then the traversal steps are done by looping the below processes till a termination condition is satisfied or the pre-defined time bound is reached.

The main iteration of *SymC* verification algorithm works in two steps.

- In the first step it computes the successor states (image) of the AR- automata and it checks whether a formula is accepted or rejected, i.e, the termination condition. The acceptance and rejection conditions of an AR-automaton are defined below,

  – For a property that should universally hold, we report a failure if one reject state is reached or success if all current states of an AR-automaton are accept states.

  - For a property that should existentially hold, we report a failure if all current AR-automata states are reject states or success if at least one of the states is an accept state.

  If the termination condition is not satisfied then, *SymC* continues to the second step.

- In the second step of each iteration *SymC* performs one symbolic simulation step on the system under inspection. During the image computation the conjunction of all the transition relation partitions is built on-the-fly to obtain the successor state set. This image computation is accelerated by a variety of standard improvements like early quantification and by using a partitioned transition relation as explained in section 3.1.

The above steps form the basic verification algorithm of *SymC*. Although, this approach helps to verify larger design compared to traditional model checking tool, it also has its own limitations in sense of the BDD explosion and time for verification. These limitations are reduced to some extent by BDD optimization and reduction techniques as explained in section 3.1. But to take the *SymC* tool to next step we have to use the divide–and–conquer approach by adopting *SymC* to the new heuristics that optimizes this approach by means of state overlap reduction and guiding. In the next coming sections we will see the AR-automaton generation and the extension of *SymC* environment to embed the new divide–and–conquer heuristics.

## 3.7 Translation of LTL into AR-automata

The verification algorithm mainly depends on the AR-automaton , as this is the one that leads to the acceptance or rejection of a property. This section addresses the construction of an AR-automaton for a given LTL/FLTL formula [75]. The main translation of temporal logic formulas into AR-automata works bottom-up in 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 atomic AR-automaton (i.e. atomic propositions, signals of the design) as shown in Figure 3.10 accepts a trace if the signal $s$ is true in the current simulation cycle. The doubly circled state represents the initial states. The states labelled with "A" belong to the set $A$ of accepting states and the states labelled with "R" belong to the set $R$ of rejecting states. For the rest of this chapter, we do not distinguish between the state set representation $A$ and the labelling representation "A". The constructed AR-automata are then consecutively composed to more complex AR-automata with respect to the operators in the FLTL formula. The swap operation exchanges the sets $A$ and $R$, i.e., a state labelled with "A" will be labelled

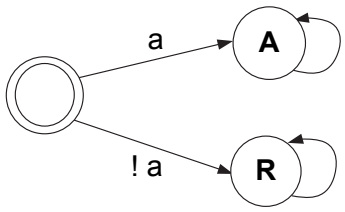


Figure 3.10: Basic AR-automaton

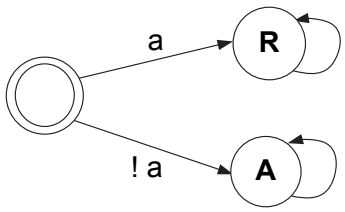


Figure 3.11: Negated formula AR-automaton

with ″R″ and vice versa as shown in Figure 3.11 and this accepts a trace if the signal $!s$ is true. This operation corresponds to the negation in the FLTL formula.

Till now only simple atomic formulas have been shown with their AR-automata equivalent. More complex formulas that are coupled with temporal operators are shown below. Assume a formula with a next time operator ″X″. Figure 3.12 in left shows the plain LTL equivalent AR-automata and the right shows the FLTL equivalent where there is a time bound (ex: $X_{[n]}$).

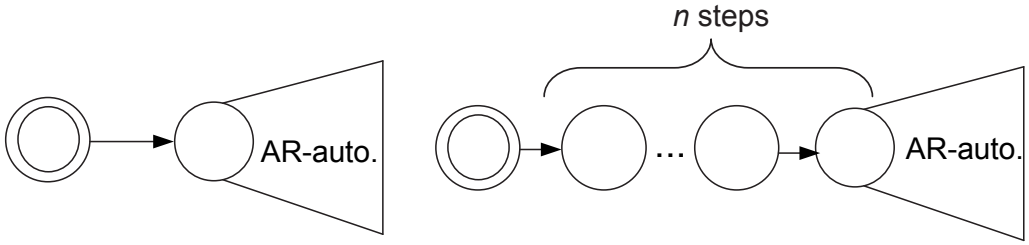


Figure 3.12: left: Next operator $X$ equivalent AR-automaton, right: $X_{[n]}$ equivalent AR-automaton

The conjunction and disjunction operation is done by merging two AR-automaton $\mathcal{A}$ and $\mathcal{B}$ that constructs one new AR-automaton by joining the initial states and appending the remaining transition relations of both AR-automata. The AR-automaton in Figure 3.13 left shows the merged AR-automata for signals $a$ and $b$. This merge operation may generate a non-deterministic AR-automaton.

A major operation during the bottom-up construction is the removal of non-determinism, i.e., the translation of a non-deterministic to a deterministic state machine. The AR-automaton can be treated like standard finite state machines in order to remove the non-determinism by enabling the application of standard algorithms (e.g., sub-set construction [76]). If the sub-set construction is applied for removing non-determinism, the resulting AR-automaton consists of states which

are themselves sets of states of the original AR-automata. These subsets may contain labelled states.

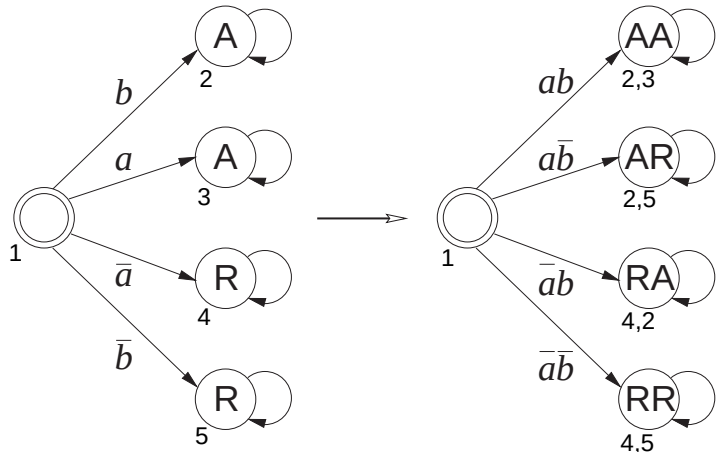


Figure 3.13: left: Original AR-automaton, right: Deterministic AR-automaton

The ″A″ and ″R″ states of the new automaton are defined over the rules called inheritance patterns shown in Figure 3.14. The inheritance patterns are used for constructing the A-set and the R-set of the new automaton. Strong inheritance leads to conjunction and weak inheritance to disjunction. Figure 3.13 right shows the deterministic AR-automata that are merged for the signals $a$ and $b$.

| Inheritance | Strong | Weak |
|---|---|---|
| Accept | A state will be labelled with A if all sub-states are labelled with A | A state will be labelled with A if one of the sub-states is labelled with A |
| Reject | A state will be labelled with R if one of the sub-states is labelled with R | A state will be labelled with R if all the sub-states are labelled with R |

Figure 3.14: Inheritance patterns

The temporal operators ″G″ globally and ″F″ eventually without time bound are constructed by adding all the new transitions that start from the initial state and leading to the initial state for all possible variable combinations as shown in Figure 3.15.

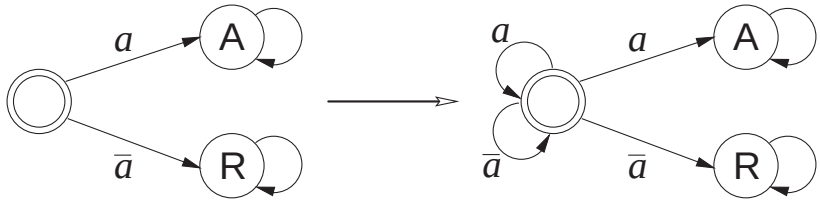


Figure 3.15: left: AR-automaton for variable $a$, right: AR-automaton for formula $F\ a$ or $G\ a$

Adding a chain of new initial states and connecting all of them with the initial state of the original operator results in the construction of the globally respective eventually operator with lower and upper time bounds as shown in Figure 3.16.

In the case of an operator with interval $[n, m]$, $m$ new states should be added, the first $m - n + 1$ states of the chain become the new initial states. In the case of $n = 0$, the initial state of the original AR-automaton remains initial. The next time operator "X" is a special case of this operation where $n = m$, and $n$ number of states are added in chain and only the first state becomes the new initial state. The non-determinism has to be removed after every operation and the type of inheritance which is used in the current construction step is determined by the temporal operator in the FLTL formula. The *globally* operator "G" utilizes the strong inheritance pattern and the "eventually" operator "F" utilizes the weak inheritance pattern (see Figure 3.14). For example, Figure 3.17 left shows the non-deterministic AR-automaton that represents both $F_{[1]}req$ and $G_{[1]}req$. Figure 3.17 right shows the deterministic AR-automaton where the A-set and the R-set defined over the inheritance patterns.

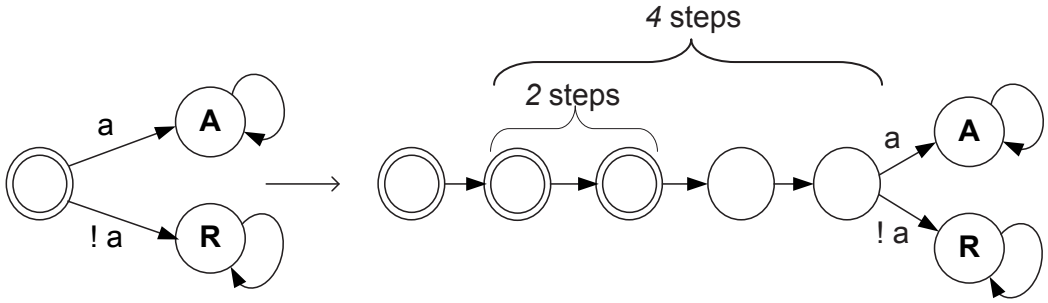


Figure 3.16: left: AR-automaton for variable $a$, right: Automaton for $F$ and $G$ extended by time bound [2,4]

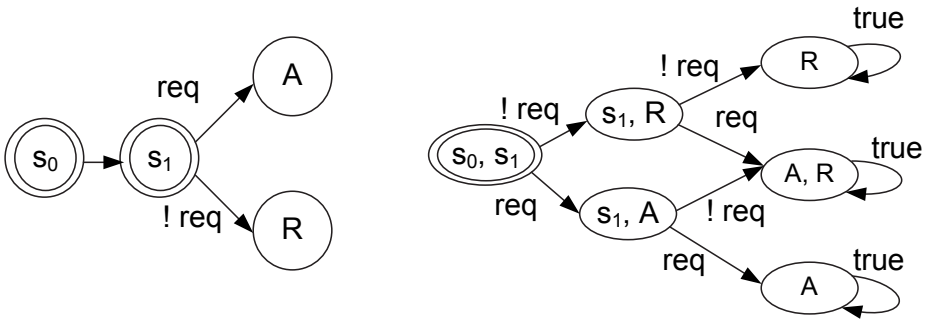


Figure 3.17: Non-deterministic to deterministic AR-automaton

Naive application of the construction rules can easily lead to huge AR-automata. This is in particular true for deeply nested formulas or formulas containing temporal quantifiers with very large time bounds. The basic structure of the generated AR-automaton by the naive construction contains huge number of equivalent states, i.e., bisimilar states. The bisimilar states are handled by the standard merging based on a partitioning algorithm explained in [77]. It starts with

a coarse partition and consecutively applies refinement steps. The algorithm is then repeated until a fix-point is reached. In [75] the authors utilize the standard partitioning algorithm by initially defining three equivalence classes, all accepting states, all rejecting states and the pending states i.e., neither accepting nor rejecting. The reduction is shown in Figure 3.18.

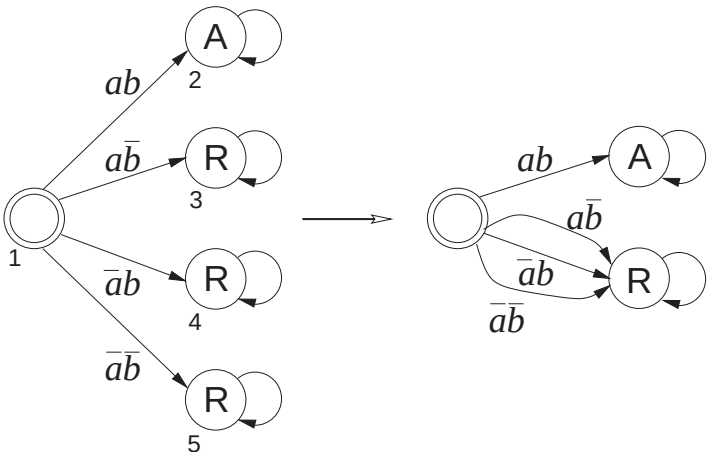


Figure 3.18: left: Original AR-automaton, right: Reduced AR-automaton

The bisimulation reduction dramatically shrinks the state space of an AR-automaton. To avoid an early combinational explosion of the state space, the bisimulation reduction is applied after each determinization step.

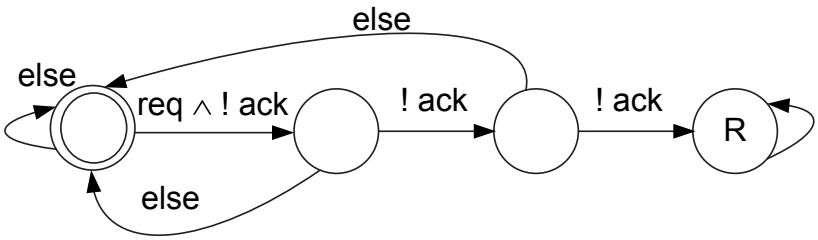


Figure 3.19: AR-automaton for the formula G(req $\rightarrow F_{[2]}$ ack)

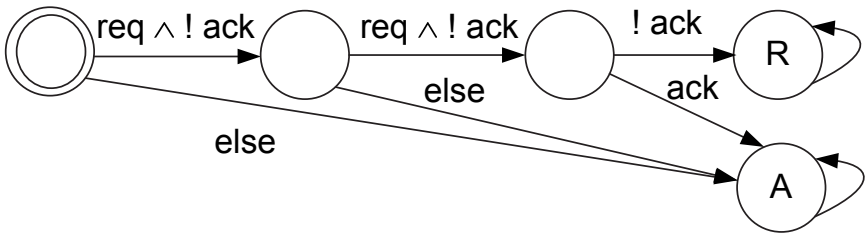


Figure 3.20: AR-automaton for the formula $G_{[1]}$ req $\rightarrow F_{[2]}$ ack

Figure 3.19 and 3.20 show a couple of real formulas and their equivalent AR-automata. The first formula expresses that always if $req$ is true then eventually

within two clocks $ack$ have to be true. The second formula expresses that if always $req$ is true till clock 1, then eventually within two clock cycles $ack$ have to be true.

## 3.8 *SymC* Extensions and Optimizations

The basic version of the tool *SymC* pioneers in verifying relatively larger designs. However, beyond a limit *SymC* has the problem of BDD explosion. Although the problem of BDD explosion can not be totally eradicated, still the methodologies based on POBDD explained in the section 3.2 can improve the situation and keep the memory requirement under control.

The basic verification loop of *SymC* has been adopted with these methodologies in order to further extend the horizon of design size handled by it. The algorithm is been altered in order to adopt both the partitioning techniques i.e., *splitting* and *windowing*. Either of the technique can be optionally selected in the command line option at the start for the verification process. Subsections 3.8.1 and 3.8.2 details the adoption of these techniques.

### 3.8.1 Splitting Technique

The splitting technique is based on the POBDDs and whenever the BDD node count representing the present state set reaches a threshold value, it is partitioned into two sub sets. This process of partitioning a BDD into two is called splitting. The two sub parts of the splitting together represent the whole state space. The splits that are obtained are expected to be smaller and balanced in size, which are then easily traversed one after the other. Thus the splitting technique is referred by the term divide-and-conquer approach. The interesting point of this splitting technique is that one of the splits is traversed first while storing the other. So if the target state is already reached in the first split, then *SymC* saves time and memory by quitting the traversal of the other split as shown in the Figure 3.21. The option of quitting the traversal of a partition or not gives the opportunity of specializing this technique for both *full validation* and *fast falsification*.

Adopting this splitting technique to *SymC* includes the alteration of the termination condition and deciding on the algorithm that partitions the BDD. The important factor of the splitting is to have smaller splits that are easier to traverse. There are different algorithms in order to split a BDD into two. Each of them is based on different heuristics having its own advantage. The following is the list of state–of–the–art partitioning algorithm.

- *VarDisjDecomp* : This variable disjunction decomposition algorithm is provided by the BDD package CUDD [78]. This aims at balanced partition by selecting a best variable depending in its heuristics.

- *BalancedDecomp* : This algorithm is the naive one that partitions the state space into two balanced parts independent of whether it is the smallest pos-

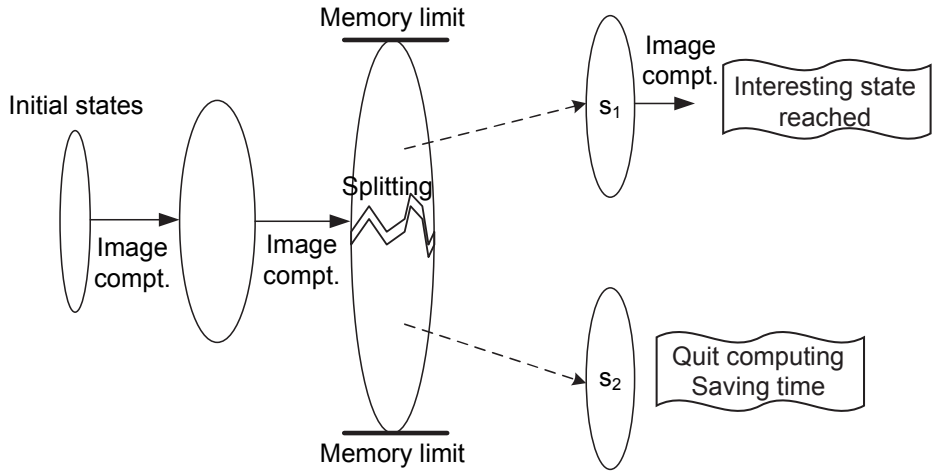


Figure 3.21: Splitting technique

sible. This algorithm is the first one to be implemented in *SymC* to realize the splitting technique.

- *EqualDistDecomp* : This algorithm a variant of the one presented in [63] is aiming not only at balanced partition, but it also considers the reduction and redundancy factors for cost calculation, given in Definition 14. Hence, the BDD partition is both balanced and also the smallest possible.

- *ShortPathSubset* : This algorithm is presented in [62] for under-approximation and provided by the CUDD package. Here in *SymC* it is used for splitting where the resulting partitions are not balanced but one of the splits is comparatively small but representing more state space. This algorithm is specifically useful for the fast falsification approach. This could perform bad in case of validation.

- *HeavyBranchSubset* : This algorithm is also presented in [62] as an alternative under-approximation technique for ShortPathSubset, which is based on a different heuristic. But it also aims at fast falsification by unbalanced splitting and representing more state space. Although aiming at fast falsification note that these under-approximation algorithms does not intelligently guide or steer the traversal to the target states.

The above listed algorithms are the state–of–the–art partitioning algorithms that are available to realize the splitting technique. However, these algorithms do not address the problems mentioned in the section 3.4. Therefore this thesis provids the below listed algorithms addressing those problems.

- *MinOverlap* : This algorithm is one of the main topics of this thesis, where it aims not only for a balanced and small partitions but also for minimal overlap. In other words, this algorithm splits the BDD in a way that in future traversal of those splits have minimal overlapping of states i.e., minimal revisiting of states. This minimal overlap of states reduces the duplicate

of work. Reduction in duplicates makes this algorithm suit better for the full validation in which every partition has to be traversed and this finally reduces the verification time. This algorithm is detailed in chapter 5.

- *Guiding* : This algorithm is the other topic of this thesis, where it aims for fast falsification. Hence, this algorithm will not give a good balancing, but splits and guides the traversal to follow the right one that has the higher chance of reaching the target states. Due to the fact that it partitions in a way that one of the split is smaller and contains more of potential target states, following that guided split could get to the error states faster. This algorithm is detailed in chapter 6.

The algorithm shown in Figure 3.22 sketches the splitting adoption into the *SymC* core simulation engine for one AR-automaton. The lines 10 - 16 in the algorithm check the BDD node count for the threshold value. If true, the current state set $act$ will be splitted into two sets $s_1$ and $s_2$. One of the splits will be stacked (line 14) for future traversal in case of necessity, while the other split is considered as the current state set (line 16) and traversal is continued with it.

The terminating condition of *SymC* is adopted as follows,

- In case of universal correctness, if one of the splits has been reported a failure then the whole verification can be stopped reporting the failure (lines 25 and 26). But if a split is reported a acceptance then the other splits have to be traversed for proving the full acceptance (lines 28 - 31).

- In case of existential correctness, if one of the splits has been reported for acceptance then the whole verification can be stopped reporting the acceptance (lines 35 and 36). But if a split has been reported a rejection or failure, then the other splits have to traversed (lines 38 - 42).

The terminating condtion for the universal and existential correctness of the property is exactly the opposite.

There are few optimizations adopted in *SymC* that increases its capability. For example, if the traversal reaches the state "A" or "R" in the AR-automaton then it loops in the same state. Hence, such states can be removed from the traversal, which reduces the load of the symbolic traversal. This is shown in the line 45 in the algorithm by the call to the function *uninterestingStates()* that returns a BDD representing the uninteresting states.

### 3.8.2 Windowing Technique

The other POBDD technique is *windowing*, where a pre-defined number of windows are partitioned at the start of the simulation or at reaching the threshold size of the BDD. Every window is identified by its unique combination of variables and the actual state space is partitioned and allocated to its respective window. All the window sub parts together represents the whole state space (see Definition 12). In *SymC* the windowing technique is adopted with a pre-defined

```
// start is the initial state space to start the traversal
// AR − aut is a special automata equivalent to the property
// bound is the number of the simulation steps
symSimulate(in: start, AR − aut, bound)
  t = 0;
  // build product state of the system and the AR-Automaton
  act = start ∧ AR-aut.start;
  while ( t ≤ bound ) // Check for the bound to stop
    // Check BDD size for threshold size
    if(act.nodeCount ≥ threshold)
       // s1 and s2 are splitted set
       Split(act, &s1, &s2)
       // stacking one of the split
       stack(s2)
       // traversal of the other split
       act = s1

    // restarting point
    restart:

    // compute AR-Automaton image
    act = image_{AR−aut}(act);

    if (check universally) // if univ. quanti. prop.
      if (act ∧ AR-aut.reject ≠ false)
        reportFailure();
      // acceptance have to be in all splits
        if (act ∧ AR-aut.accept = act)
          if (stack.size == 0) reportAcceptance();
              else
                act = stack.pop();
                goto restart;

    if (check existentially) // if exist. quanti. prop.
      if (act ∧ AR-aut.accept ≠ false)
        reportAcceptance();
      // rejection have to be in all the splits
      if (act ∧ AR-aut.reject = act)
         if (stack.size == 0) reportFailure();
         else
           act = stack.pop();
           goto restart;

    // remove uninteresting states
    act = act ∧ ! unintrestingStates();

    act = image_T (act); // early quant etc.
    t = t + 1; // simulation step counter
```

Figure 3.22: Splitting based *SymC*.

number of windows taken as a command line input. In contrast to the splitting techniques, all the windows are traversed simultaneously, i.e., breadth first, as shown in Figure 3.23.

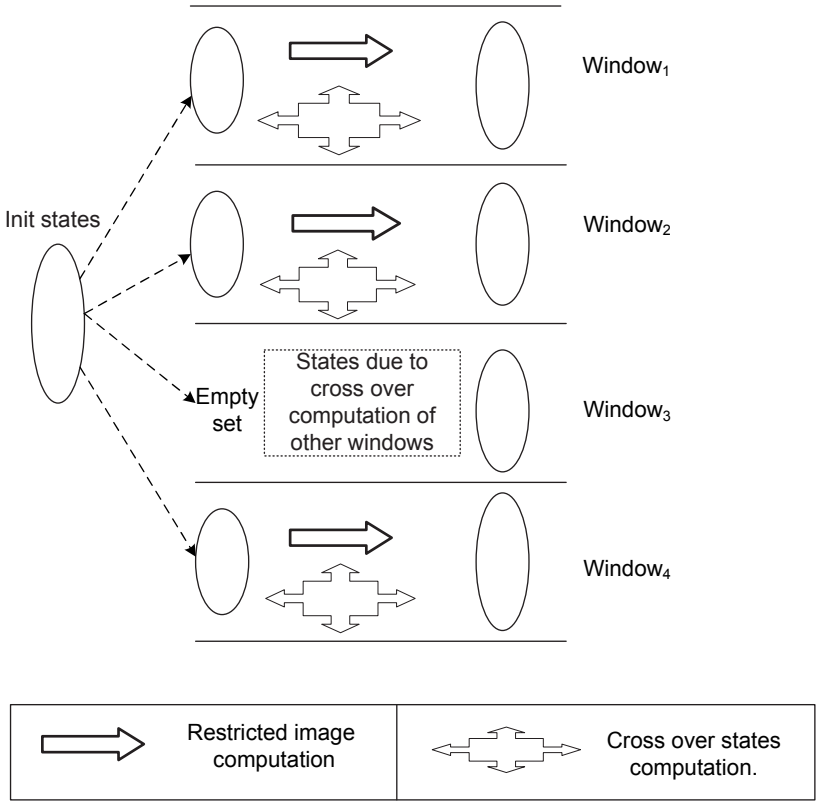


Figure 3.23: Windowing technique

In general the partitioning in windowing is different to the splitting technique, as in the later case the splits are always two whereas the former usually more than 2 windows. Moreover, in the splitting technique partitioning at later stage is possible if the BDD reaches the threshold limit. But in *SymC* the windowing is adopted to have a static number of windows. Due to the fact that all the windows are traversed simultaneously the partitioning algorithm should partition the set into relatively balanced subparts. In the naive form of partitioning there is a higher chance that some of the windows end up empty, i.e., there might be no states that belong to it. However, after some traversal steps these windows might get some states due to the cross over states as shown in Figure 3.23. Trivially, the power of the windowing technique comes to limelight only when every window has some state space to proceed. Because if some windows are empty then it is not of interest as more work load will be on other windows. Particularly, this emptying of windows is a total failure of the windowing technique in case of a distributive environment where many computing power is made idle. Therefore, in the windowing technique the restrictions should be created with some refined heuristics in order to avoid the emptiness of the window.

Due to the fact that the windowing technique has the concept of owned and non-owned states the state overlap problem is avoided. But this advantage comes

with the overhead of communicating the non-owned states to its respective owner and this non-owned states are known as cross-over states. Reduction of the cross-over states at every traversal step is also an important factor to highlight the advantage of this technique, because computing these cross-over states could take lot of time and memory and additional time to be communicated and verified. The overhead of computing and communicating the cross-over states lead to the idea of separating the fast falsification and full validation methods under windowing. Hence, the full validation approach is the one where the cross-over states are computed and communicated at every step of the traversal. The fast falsification approach is where the cross-over states are not computed and communicated till end of the traversal of all windows. This makes the window traversal faster and expecting the target states are reachable in these window limits. In case the target states are only reachable by the cross-over states, the method does not help and therefore has to compute and communicate the cross-over states separately.

Figure 3.24 shows the adoption of the windowing technique into the *SymC* algorithm. The lines 12 and 13 show the initialization of the windows by their respective restrictions and assigning the owned state space. The terminating condition is similar to the splitting technique except the window that is universally accepted or existentially rejected will be cleared as shown in the line 28 and 39. The local image computation of the windows are computed (line 44) and the equation 3.6 defines the local image computation. Once all the windows are finished with owned state computation the non-owned or cross-over states are computed and distributed by the function *CrossOverstates()* (line 48).

$$
\begin{aligned}
&L_{image}(w_i) = image(w_i \wedge r_i^{'}) \\
&\text{Where } w_i \text{ is the window,} \\
&r_i^{'} \text{ is the restriction with next state variable and} \\
&0 \leq \text{i} \leq \text{N-1}
\end{aligned}
\tag{3.6}
$$

The cross over states are computed and disjuncted to their respective windows by the function in line 48. The cross-over states computation from a window $i$ to $j$ is shown in the equation 3.7, where it is then disjuncted with the existing state space in the window $w_j$.

$$
\begin{aligned}
&CrossOverstates(W) = w_j \vee image(w_i \wedge r_j^{'}) \\
&\text{Where W} = w_i, \, 0 \leq \text{i, j} \leq \text{N-1 and i} \neq \text{j}
\end{aligned}
\tag{3.7}
$$

The state of the art techniques suggests to compute the cross over states at later stages, where the aim is to find fast failure. However the *SymC* is implemented with the proper validation approach, where cross over states are computed for every step.

Trivially, the splits and windows could be handled in a distributive environment for better performance. Hence, as an extension to sequential *SymC* a parallel version of it is also implemented. The parallel version of *SymC* is out of this thesis scope, so until otherwise mentioned *SymC* always means the sequential version.

## 3.9 Tutorial for *SymC* Verification

This section gives a short tutorial of the *SymC* tool and its verification process. Consider an example design that is shown in Figure 3.25 left. The design has three states and the transitions are shown as a directed arrows. The initial state is shown by doubly circled notation. The transition from the state $s_3$ can only happen by consuming a Boolean signal $a$, whereas the other transitions can happen at every clock and without any input. Assume that we are interested in verifying a property that the Boolean signal $a$ is consumed eventually within two clocks steps starting from the initial state. The property is represented as a FLTL formula $F_{[2]}a$ and it's corresponding AR-automaton is shown in Figure 3.25 right.

This tutorial is based on the basic *SymC* algorithm shown in Figure 3.9. The product of the initial state of the design and the AR-automaton is constructed as shown in the line 9. The traversal starts at the time point 0 (line 5) and loops till the explicit bound (line 11) is reached or until the termination condition is reached. In the loop, image of the AR-automaton is computed first (line 15) and checked for the terminating condition.

It will enter the *check universally* condition loop (line 17) if the property is specified to be universal property, i.e., the property has to hold in all the traces. In case of an existential property it will enter the *check existentially* condition loop (line 21) i.e., the property has to hold in at least one of the traces. If the terminating condition is not solved then the image of the design is computed (line 25) and the time point or the traversal step is incremented and loops the same again.

If we assume the property to be an universal property, then after 2 time steps the one of the universal terminating conditions will be true. i.e., the $act$ set will contain at least one reject state. Figure 3.26 shows the traversal steps of the product machine. The final set of states are labelled as per the weak inheritance shown in the Figure 3.14 because the property contains an eventual operator. Because we require an universal validation we utilize the strong inheritance on the final set of states to validate the property. Due to the strong inheritance there will be at least one reject state. Therefore, the property will be rejected for universal property (line 18).

However, if we consider the property to be a existential property then the Figure 3.26 is labelled with weak inheritance for both the eventual operator and also for the existential case. Therefore there will be at least one accept state and so the existential terminating condition (line 22) will be true and the whole property will be accepted. In case of rejection or failure of a property, optionally a counter example can be generated by the pre-image computation.

```
// start is the initial state space to start the traversal
// AR − aut is a special automata equivalent to the property
// bound is the number of the simulation steps
symSimulate(in: start, AR − aut, bound)
  t = 0, count = 0;
  // Pre-defined number of windows
  N = 4;
  WindowSplit(); // Partitioning for windows
  // build product state of the system and the AR-Automaton
  act = start ∧ AR-aut.start;
  // initializing windows by its owned state space
  for i = 0 to N-1
          w_i = act * r_i // w_i - window and r_i - restriction
  while ( t ≤ bound )
    while (count < N)
      act = w_count;
      // compute AR-Automaton image
      act = image_{AR−aut}(act);

      if (check universally) // Univ.  prop.
        if (act ∧ AR-aut.reject ≠ false)
          reportFailure();
          // acceptance have to be in all the windows
          if (act ∧ AR-aut.accept = act )
            if (allWindow == accept)//check all partitions
              reportAcceptance();
            else
              w_count = false;
              count++;
              break;
      if (check existentially) // Exist.  prop.
        if (act ∧ AR-aut.accept ≠ false)
          reportAcceptance();
          // rejection have to be in all the windows
          if (act ∧ AR-aut.reject = act )
            if (allWindow == failure)
              reportFailure();
            else
              w_count = false;
              count++;
              break;
      // remove uninteresting states
      act = act ∧ ! unintrestingStates();
      act = L_{image}(act) // local image computaion
      w_count = act;
      count++;
    // cross-over compt.  and dist., W is set all windows
    CrossOverstates(W);
    t = t + 1; // simulation step counter
```

Figure 3.24: Windowing based *SymC*.

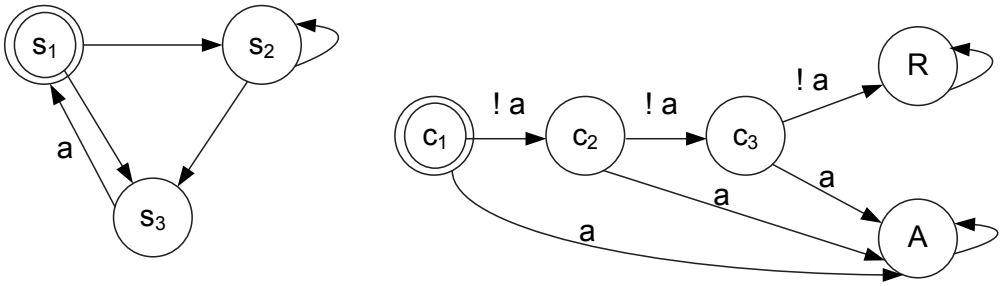


Figure 3.25: left: Design, right: property AR-automaton

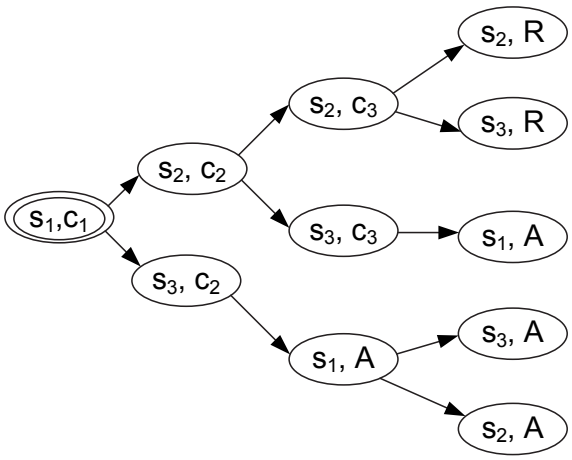


Figure 3.26: Design and AR-automaton traversal

# Chapter 4

# Influence Information and Manipulation

Chapter 3 discussed the state-of-the-art techniques and approaches in the formal verification field. Partitioning the actual BDD into a number of smaller BDDs called POBDDs and handling them separately is one among them. This methodology enables the control over memory requirements of the verification tool, allowing the tool to handle comparatively larger designs. However, in practice this methodology might not scale up, due to a wrong selection of parameters.

This methodology needs some refined and intelligent heuristics to be applied efficiently to address the problems mentioned in section 3.4. Trivially, the aim of POBDD based approach is to minimize the size of the partitioned BDD and also to have balanced partitions as presented in [63], which requires less memory compared to the original BDD. Hitherto, this is the only heuristic that has been applied to realize this partitioning methodology. But partitioning based on BDD size and the balancing factor only, does not explore the whole potential of this methodology, because this POBDD based methodology can be applied in two different verification scenarios, namely *full validation* and *fast falsification*. Both the scenarios require totally different approaches in the method of partitioning. However, the POBDD based verification can be realized by two different techniques called *Splitting* and *Windowing* that is briefed in the section 3.8.

The partitioning heuristics should be based on the scenarios of the verification. For example, in case of the splitting technique the fast falsification does not require a balanced partition whereas for full validation the balancing factor is an important issue. In contrast, the windowing technique requires balanced partitions and non empty windows independent of the verification scenarios.

The following sections discuss in detail the two techniques under both scenarios. The combination of techniques in different scenarios require different heuristics for partitioning. Despite different heuristics, all are based on a special categorization of variables that is referred as the influence factor information. The *influence* factor of variables is basically the static information that can be obtain from the partitioned transition relation. This chapter details the manipulation of partitioned transition relation to collect the influence factors of the variables that are utilized in the heuristics of splitting technique and for the on-the-fly method

of partitioning for the windowing technique.

## 4.1 Intelligent Partitioning

The idea of partitioning is similar in both splitting and windowing techniques. However, there are crucial differences between them. Both techniques differ by the concept of owned and non-owned states and both handle the partitions in a totally different way as explained in the section 3.6. In general, if a property can be decided to be verified under the full validation or fast falsification scenario, then both the partitioning techniques can be optimized for such a verification by means of intelligent partitioning heuristics. This section discusses both the scenarios under splitting and windowing techniques and therefore the goals of the heuristics to be applied. The following details the goals of the full validation and the fast falsification scenarios.

- *Full validation*: Full validation aims at exploring all the traces in order to validate the property and shows it holds in all. The full validation scenario better suits the properties that are universally accepted or existentially rejected. Therefore full validation approach can not offered to do any approximation or restriction as it has to explore all the traces. Hence, in general it is more time consuming compared to fast falsification, but this is the best approach in terms of time for an universal verification where the property holds in all traces.

- *Fast falsification*: The aim of fast falsification is to find at least one trace faster compared to full validation approach that either satisfies the property in case of existential verification or that violates the property in case of universal verification. In other words, this fast falsification approach better suits the properties that are universally rejected or existentially accepted. If the satisfying trace is found faster then the verification process is finished, hence the term fast falsification. The term falsification is due to the fact that existential satisfaction is inverse of the universal violation of the property. In order to find at least one trace faster, fast falsification does not require to explore all the traces and therefore can offered to utilize either under approximation or restriction of actual state space. Of course, in worst case i.e., not finding at least one trace, it has to handle all the traces, which will be time consuming.

The following subsections detail splitting and windowing techniques in both scenarios and briefs the problems that arise.

### 4.1.1 Intelligent Splitting

Splitting is a partitioning technique that is based on the Shannon expansion. Assume, a BDD $f$ representing the actual state space of a design and if $\mid f \mid \geq N_t$

where $N_t$ is the threshold value of the node count limit for partitioning. Then $f$ is partitioned into $f_1$ and $f_2$, as shown in equation 4.1.

$$\begin{aligned} f_1 &= a \wedge f_a \\ f_2 &= !a \wedge f_{!a} \\ f &= f_1 \vee f_2 \end{aligned} \tag{4.1}$$

where $a \in supp(f)$ is the partitioning variable, $f_a$ is the positive cofactor representing part of the state space and $f_{!a}$ is the negative cofactor representing the other part of the state space of $f$ with respect to the variable $a$. Once the partitioning is finished either $f_a$ or $f_{!a}$ is assigned to the actual state space $f$ and continued with traversal while stacking the other. Splitting is called again whenever the actual state set $f$ is crossing the threshold value $N_t$. Depending on the verification scenario, the splitting technique can be optimized and this leads to the intelligent selection of the variable $a$ such that it scales for that scenario.

- Full validation: For full validation of a property in a design, *SymC* is expected to traverse all the partitions. Therefore, the partitions are expected to have relatively balanced sizes so that the they are traversed at ease. Obviously, to minimize the effort of traversing the partitions, revisiting of states has to be avoided or reduced. This phenomenon of revisiting states in different partitions is referred as the *state overlap* among partitions. This state overlap leads to the duplicate computation, which is waste of time and memory and results in degradation of the splitting approach and hence, the tool. Thus the splitting for full validation approach should concentrate on reducing the state space overlap along with relatively balanced partitions.
- Fast falsification: For fast falsification of a property in a design *SymC* is expected to find at least one trace that satisfies the property and that trace should be found faster. This trace can be found by means of reliable *guiding* method that steers the traversal to the target states. Splitting based on guiding requires more interesting states in one of the partitions that could lead to the target state and prioritizing that partition to be followed first. Although guiding increases the possibility of reaching the target states relatively faster, it can be further optimized and made faster when combined with an under-approximation technique. This combination of guiding and under-approximation results in finding the target states faster. Trivially, the other partitions need not to be traversed if the target state is reached in the previous partition, hence, the size balancing between the partitions is of second priority than to the selected variable which covers more interesting states.

### 4.1.2 Intelligent Windowing

Windowing is partitioning technique based on definition 12. In this technique the partitions are referred to as windows. The main intension of the windowing

technique is to partition the state space in a way that every window has some state space at every step of its traversal. Thus in contrast to the splitting technique the balancing condition is important in the windowing technique independent of the scenarios. In this technique the fast falsification and full validation scenarios differ only by the state space considered for the traversal.

- Full validation: For full validation of a property the whole state space has to be considered. Thus the cross-over states have to be computed at every step and distributed to the owners for further traversal. This cross-over state computation and distribution is considered to be the bottleneck and therefore have to be reduced. The partitioning algorithm should be caring about the empty windows as this could degrade the technique.

- Fast falsification: The fast falsification in this technique can be realized in two ways, first by traversing sequentially, i.e., depth wise, all the windows and to stop if the target is reached. The second by traversing all the windows at the same time step, i.e., breadth first, but without computing and distributing the cross-over states. The second approach is utilized in this thesis. The idea of not distributing the cross-over states reduces both the overheads and the additional state space to be traversed and hence the traversal of only the owned states of the windows is faster. In other words, if the target states can be detected by traversing the window without distributing the cross-over states will be much faster compared to the one with cross-over distribution. Trivially, if the target states are not detected by the non-distributed traversal then the cross-over states have to be considered. Hence, if a target state is reachable from the initial owned state space then it is faster to find. However, this also requires a non empty windows and balanced partitions.

We require some sort of information about the design that is verified in order to address the problems and to achieve the required goals of both the techniques in different scenarios. The collection of such information about the design is discussed in the next section. An on-the-fly partitioning approach is introduced in section 4.3 for the windowing technique addressing the problem of balanced partitions that has a possibility of indirect or hidden advantage of less cross-over states.

## 4.2 Influence Factors

In order to address the problems that arise in the splitting technique, we require some defined information of the design that is to be verified. This information should be capable of providing some knowledge that could be utilized in solving or minimizing the problems. The search for such information results in the collection of influence factors which is based on the cone of influence of the partitioned transition relation. The partitioned transition relation is the state-of-the-art technique and was discussed in section 3.1. In [53] Burch et al. introduced the

partitioned transition relation in order to reduce the complexity of BDD based symbolic verification. This partitioned transition relation exhibits a locality information that could be further utilized to improve the formal verification tool. The locality information is collected and ordered to form the influence factor table that are extensively used in order to optimize the splitting and the windowing techniques.

The locality information is collected as a pre-processing step by statically analyzing the design. The locality information manipulation and its corresponding influence factors collection is explained by means of an example. Assume a synchronous circuit design with $m$ number of state variables (E) and $n$ number of input variables (I), then there are $2^m$ possible states that belong to the set of states $S$. The set of states is represented as BDD and is denoted as $S(E)$, where $E = \{e_1, \ldots, e_m\}$ is the set of Boolean state variables. To represent the transition relation using BDDs, we require a second set of states $S'$ encoded by $E' = \{e'_1, \ldots e'_m\}$ called next state variables. Then the BDD representing the transition relation $\mathcal{T}$ is denoted as $\mathcal{T}(E, E')$.

The partitioned transition relation is constructed based on the piece of combinational logic that determines how a state variable $e_i$ is updated. Let $f_i$ be the next state function computed by this logic, then the next state value of the $e_i$ is given by $e'_i = f_i(E)$. This sub equations in turn define the whole transition relation as,

$$\mathcal{T}(E, E') = T_1(E, E') \wedge \ldots \wedge T_m(E, E')$$
$$\text{where } T_i(E, E') = (e'_i \equiv f_i(E)).$$

From now on we denote $T_i(E, E')$ as $T_i$. The transition relation can be expressed as a conjunction of relations and it is represented as a list of parts, which are implicitly conjuncted. This list representation is called the partitioned transition relation. The image computation step with the partitioned transition relation is then improved by a method called *early quantification*. This method is based on an important observation of circuit locality, i.e, only a small number of variables in $E$ is forming the next state function $f_i$, formally, $support(f_i) \subset E$, where $support$ returns the variables involved in the function.

Let us assume the partitioned transition relation shown in the equation 4.2 and discuss the locality information and the influence factor collection. Let us also assume the set of state variables E = { $e_1$, $e_2$, $e_3$} and the input variables I = {$i_1$, $i_2$ } and the transition relation T = { $T_1$, $T_2$, $T_3$ }. From the equation 4.2 one can see that not all the sub transition relation depend on all the state variables and the input variables. The dependency in most cases is a subset of the actual set of state variables and the input variables. This subset dependency of the partitioned transition relation is called the locality information which is collected and ordered in a fashion to form the influence factors. In principle, the influence factor is an ordered collection of all the variables that influence the next state value of a specific state variable.

$$\begin{aligned} T_1 &= (e'_1 \equiv f_1) where f_1 = (\neg e_2 \vee e_3) \vee i_1 \\ T_2 &= (e'_2 \equiv f_2) where f_2 = e_2 \vee (\neg i_2 \vee i_1) \\ T_3 &= (e'_3 \equiv f_3) where f_3 = e_2 \wedge (\neg e_3 \vee i_2) \end{aligned} \tag{4.2}$$

The *influence* factors are different for the input and the state variables. The influence factor collection for the state variables can be calculated repeatedly, i.e., looping the collection for a defined number of steps, whereas for the input variables it is calculated for only once. We have two different factors, first, *lookaheads* denoted by $\mathcal{D}_{\mathsf{S}}^{\uparrow}(e,l)$, is a set containing all the next state variables (equivalent present state variables) that are influenced by the variable $e$ over $l$ steps. Second, *lookback* denoted as $\mathcal{D}_{\mathsf{S}}^{\downarrow}(e^{'},l)$, is a set containing all the state variables that influence the next state variable $e^{'}$ in $l$ steps back. If either of the factor is computed then the other can be obtained. A formal definition of *influence* is given below,

**Definition 15** *Let $l_1, l_2 \in \mathbb{N}$ be the influence limits. For a given FSM $\mathcal{A}$, the influence $\Phi_{l_1,l_2}(e) \in [-1,1]$ of a state variable $e \in E$, with $|E| = m$, is defined as,*

$$\Phi_{l_1,l_2}(e) = \frac{|\mathcal{D}_{S}^{\uparrow}(e,l_1)| - |\mathcal{D}_{S}^{\downarrow}(e,l_2)|}{m}. \tag{4.3}$$

where the factors lookaheads and lookback definitions are formally given by the equations 4.4 and 4.5.

$$\mathcal{D}_{\mathsf{S}}^{\uparrow}(e,1) = \{v_1,\ldots,v_m\} \tag{4.4}$$

where $v_i \in E$, $e \in supp(f_{v_i})$, $v_i' \equiv f_{v_i}$ and $1 \leq \mathrm{i} \leq \mathrm{m}$.

$$\mathcal{D}_{\mathsf{S}}^{\downarrow}(e,1) = supp(f_e) \tag{4.5}$$

where $f_e$ is the next state function of the variable $e$ such that $e' \equiv f_e$.

$$\mathcal{D}_{\mathsf{S}}^{\uparrow}(e,l_1+1) = \mathcal{D}_{\mathsf{S}}^{\uparrow}\,(\mathcal{D}_{\mathsf{S}}^{\uparrow}(e,l_1),1)$$

For $\mathcal{D}_{\mathsf{S}}^{\uparrow}(e,1)$, we collect the corresponding $e_i$ represented by the partitions $T_i$ that contain $e$ and we iterate it for $l_1 > 1$. The implementation of the influence lookaheads is shown in the Figure 4.1. The algorithm picks all the state variables $e \in$ E one by one (line 8) and checks for its presence in the next state function $f_v$ that corresponds to the transition relation partition $t_v$ (line 11). If present in the support set then the present state variable $v$ is inserted in the influence which is equivalent to $v'$ where $t_v = (v' \equiv f_v)$. Completing the loop for all the partitions $t_i$ then depending on the $l$ value it either jumps to line 24 where the lookahead is stored else it loops from the line 16 to 23. In this loop the variables in the influence are again checked for their influence and are appended to the actual influence and this loops $l$ - 1 times.

For $\mathcal{D}_{\mathsf{S}}^{\downarrow}(e,1)$ we count the state variables in the $support$ of $T_i$. This is shown in the Figure 4.2. Although the computation of lookback for more than one step is possible, it is not utilized here in this thesis. The lookback algorithm is shown in Figure 4.2. For all the partitioned transition relation $t_e$ (line 8) the support set $supp(t_e)$ is collected (line 10), and all the variables $v$ in the support set and in E are inserted in the lookback set for the variable $e$ (lines 11 - 12).

In reference to our example the transition relation shown in equation 4.2, the corresponding influence factors are shown in equation 4.6. The lookahead($e_1$)

```
// T is the Partitioned transition relation
// l is the integer value for look ahead
// E is the set of all state variables
// t_e is the next state function of the variable e
// D↑_S(e,l) is the influence that maps string to set of string
look-ahead(in: T, l; out: D↑_S(e,l))
j = 1
for all e ∈ E
  Inf.clear()
  for all t_v ∈ T
    if e ∈ supp(t_v) then
      Inf.insert(v)
      //where t_v = v' ≡ f_v, v' is next state var.  of v
  I_f = Inf
  // e is present state variable name of e'
  while j < l
    for all i_f ∈ I_f
      temp-inf.clear()
      for all v ∈ i_f.second
        temp-inf.insert((I_f.find(v)).second)
      temp-I_f[i_f.first] = temp-inf
    I_f = temp-I_f
    j++
  D↑_S(e,l) = I_f
```

Figure 4.1: Influence lookaheads algorithm.

for one step is a empty set as it does not occur in any of the support set of the transitions. However, the variables $e_2$ and $e_3$ are not empty as it is in the support set of some transitions.

$$\begin{aligned}
&\mathcal{D}^{\uparrow}_{\mathsf{S}}(e_1,1) = \{\} \\
&\mathcal{D}^{\uparrow}_{\mathsf{S}}(e_2,1) = \{e_1,e_2,e_3\} \\
&\mathcal{D}^{\uparrow}_{\mathsf{S}}(e_3,1) = \{e_1,e_3\} \\
&\\
&\mathcal{D}^{\downarrow}_{\mathsf{S}}(e_1,1) = \{e_2,e_3\} \\
&\mathcal{D}^{\downarrow}_{\mathsf{S}}(e_2,1) = \{e_2\} \\
&\mathcal{D}^{\downarrow}_{\mathsf{S}}(e_3,1) = \{e_2,e_3\}
\end{aligned} \tag{4.6}$$

Similar to the lookback factor of the state variables, lookback factor of the input variables can also be collected. It is denoted as $\mathcal{D}^{\downarrow}_{\mathsf{I}}(e,1)$ and used in the fast falsification scenario of the splitting technique and is detailed in chapter 6. The lookback factor of the input variables is a set containing all the input variables that are present in $supp(t_e)$. Formally, the lookback input influence factor is de-

```
// T is the Partitioned transition relation                          1
// E is the set of all state variables                               2
// D↓_S(e,1) is the lookback influence of var e                      3
// support is the support set                                        4
// e is the present state variable that t depend on                  5
// t_e is transition relation of the variable e                      6
look-back(in: T, E; out: D↓_S(e,1) )                                 7
for all t_e ∈ T                                                      8
  Inf.clear()                                                        9
  support = supp(t_e)                                                10
  for all v ∈ support ∧ v ∈ E                                        11
    Inf.insert(v)                                                    12
  D↓_S(e,1) = Inf                                                    13
```

Figure 4.2: Influence lookback algorithm.

```
// T is the Partitioned transition relation                          1
// I is the set of all input variables                               2
// D_i(e) is the lookback influence for input vars.                  3
// support is the support set                                        4
// t_e is transition relation of the state variable e                5
look-back(in: T, I; out:   D_i(e))                                   6
for all t_e ∈ T                                                      7
  Inf.clear()                                                        8
  support = supp(t_e)                                                9
  for all i ∈ I                                                      10
    if i ∈ support                                                   11
      Inf.insert(i)                                                  12
  D_i(e) = Inf                                                       13
```

Figure 4.3: Input influence lookback algorithm.

fined as in the equation 4.7.

$$\mathcal{D}_{\boldsymbol{i}}^{\downarrow}(e,1) = \{i_1, \ldots, i_n\} \tag{4.7}$$

where $n$ is the number of input variables and $i_j \in supp(t_e) \cap I$. The input influence factor is always calculated for 1 step and it is always lookback, hence the notation can be simplified as $\mathcal{D}_{\boldsymbol{i}}(e)$. Figure 4.3 shows the algorithm of input influence collection. The loop takes every partition of the transition relation $t_e$ (line 7) and collects the support set of $t_e$ (line 9). The second loop checks for the presence of any input variable and if yes, then inserts that $i$ in to the set (lines 10 - 12). The input variable influence of our transitions in equation 4.2 is shown in the equation 4.8,

$$\begin{aligned} \mathcal{D}_{\boldsymbol{i}}(e_1) &= \{i_1\} \\ \mathcal{D}_{\boldsymbol{i}}(e_2) &= \{i_1, i_2\} \\ \mathcal{D}_{\boldsymbol{i}}(e_3) &= \{i_2\} \end{aligned} \tag{4.8}$$

This influence factors are extensively used in the algorithms that address the problems of the splitting technique. The influence factors are effectively manipulated in order to find a interesting splitting variable for controlling the state space overlap and also for guiding. These are detailed in sections 5.1 and 6.1 respectively.

The next section 4.3 details the on-the-fly balanced partitioning method for the windowing technique and explains the indirect effect of reduction of cross-over states.

## 4.3 On-The-Fly Partitioning Method for Windowing Technique

Section 4.1.2 pointed out that the windowing technique requires balanced partitions and reduced cross over states. Partitioning using the naive methods create problems of unbalanced partitions and often many empty windows. In this section an on-the-fly partitioning technique is proposed which address the more balanced partition and empty windows. This method has an interesting and indirect side effect, i.e., the possibility of reduced cross-over states among the windows partitioned by this method.

In the usual naive method of pre-defined windowing technique for $2^n$ windows $n$ variables are required and every window is defined by the restriction $r_i$ which is the unique combination of those $n$ variables. Algorithmically, finding such $n$ variables for a balanced distribution is time consuming, hence this thesis utilize a variant of this approach to find the variables on-the-fly while solving the the balancing condition.

On-the-fly balanced distribution can be explained by assuming a set of states represented by the BDD S as shown in Figure 4.4. The first variable is selected in a way that it partitions the set into two balanced parts $S_v$ and $S_{\overline{v}}$ with respect to the variable $v$ based on the Shannon expansion. The balancing condition is defined by $\delta$ as shown in equation 3.5. 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. The difference between the usual naive method to this method is that for $2^n$ windows there can be a maximum of $2^n - 1$ variables involved by the on-the-fly method in contrast to only $n$ variables in naive method. In other words, the naive method always utilizes the same variables for second partitioning, whereas the on-the-fly can use different variables. Figure 4.4 (a) shows the on-the-fly balanced distribution of window partitioning, where there are 3 variables involved in order to form 4 windows. In contrast Figure 4.4 (b) shows the usual method of window partitioning, where there are only 2 variables involved. The different variable utilized for the partitioning is shown in the dotted circles and their combinations form the restriction $r_i$ of the windows, where $1 \leq \mathrm{i} \leq 2^n$ windows.

The on-the-fly window partitioning method, although utilizing a higher number of variables, it still holds the definition of windowing (see definition 12) and makes the distribution efficient and easier. It also produces a balanced partition

among windows and our experiments shows that it hardly creates any empty windows during the traversal.

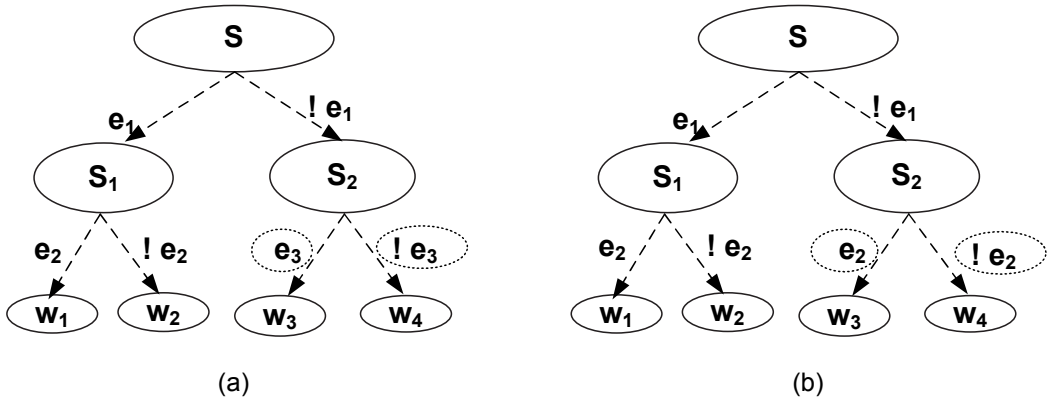


Figure 4.4: (a). On-the-fly window, (b) Usual window partitioning

The interesting side effect of this method is explained with the example shown in Figure 4.4. In general, the unique identification (restrictions) of windows makes it hard to reduce the cross-over states. For example, if 4 windows ($w_1, \ldots, w_4$) are required then a maximum of 2 variables are used to identify the 4 windows by the naive partitioning method. Equation 4.9 shows the unique identification or the restrictions of the 4 windows by the usual method, where the selected variables are $e_1$ and $e_2$. The equation 4.10 shows the restrictions of the 4 windows by the on-the-fly partitioning, where the selected variables are $e_1$, $e_2$ and $e_3$.

$$\begin{aligned} r_1 &= e_1 \wedge e_2 \\ r_2 &= e_1 \wedge !e_2 \\ r_3 &= !e_1 \wedge e_2 \\ r_4 &= !e_1 \wedge !e_2 \end{aligned} \tag{4.9}$$

$$\begin{aligned} r_1 &= e_1 \wedge e_2 \\ r_2 &= e_1 \wedge !e_2 \\ r_3 &= !e_1 \wedge e_3 \\ r_4 &= !e_1 \wedge !e_3 \end{aligned} \tag{4.10}$$

As mentioned earlier the cross-over states of the window $w_n$ are the set of next states that do not satisfy the restriction $r_n$. Formally,

$$Cross\text{-}Over = Image(w_n, T) \wedge !r_n \tag{4.11}$$

In order to completely remove the cross-over states, the restrictions, i.e., the selected partitioning variables have to have the next state function transition depending on itself, as shown in equation 4.12. But this type of transition function

hardly exists in practical design, hence we have to look for an option that might reduce this problem.

$$T_e = (e' \equiv f_e) \text{ where } f_e = e \tag{4.12}$$

Let us compare the usual naive method to the on-the-fly method to prove that the later approach is better for easy distribution and also has the indirect effect of cross-over state reduction.

As we have seen above, the usual method of windowing requires 2 variables to form 4 windows. The restrictions defines the owned state space of the windows. Therefore, given a state, it can be decided to which window it belongs, i.e., which is the owner window of the state. Equation 4.13 shows the owner or the distribution of the state space to windows depending on the truth value of the variable $e_n$. Similarly, the on-the-fly method requires 3 variables to form the 4 windows. Equation 4.14 shows the owner or the distribution of the state space to windows according to the truth value of the variable $e_n$.

Trivially, the distribution of the state space among windows by the usual method is more conservative than the distribution of the on-the-fly method. In other words, except the state variable $e_1$, other variables can be true in 3 different windows by the on-the-fly method, in contrast to only 2 windows by the usual method. This reveals that the on-the-fly method can distribute the state space comparatively easier and faster as the state variables can be true in more windows.

$$\begin{aligned} e_1 &\Leftrightarrow \{w_1, w_2\} \\ !e_1 &\Leftrightarrow \{w_3, w_4\} \\ e_2 &\Leftrightarrow \{w_1, w_3\} \\ !e_2 &\Leftrightarrow \{w_2, w_4\} \end{aligned} \tag{4.13}$$

$$\begin{aligned} e_1 &\Leftrightarrow \{w_1, w_2\} \\ !e_1 &\Leftrightarrow \{w_3, w_4\} \\ e_2 &\Leftrightarrow \{w_1, w_3, w_4\} \\ !e_2 &\Leftrightarrow \{w_2, w_3, w_4\} \\ e_3 &\Leftrightarrow \{w_1, w_2, w_3\} \\ !e_3 &\Leftrightarrow \{w_1, w_2, w_4\} \end{aligned} \tag{4.14}$$

Concerning the indirect effect of reduced cross-over states, the variable $e_1$ is of no interest as it has the same priority level (number of windows) in both the methods, therefore, let us see the next variable $e_2$. Given a state space where the variable $e_2$ is true, then depending on the variable $e_1$, the states should be owned by one of the windows in the set $\{w_1, w_3, w_4\}$, that are partitioned by on-the-fly method. For instance, if state space $S$ = { 001, 010 } is encoded by the state variables E = $\{e_1, e_2, e_3\}$ then window $w_3$ will hold $\{001\}$ and $w_4$ will hold $\{010\}$. If we assume a transition relation in equation 4.15 such that the next state space

$N$ = { 011, 000 } where only the variable $e_2$ will be changed to false, then the set $N$ will be in the window set $\{w_3, w_4\}$ where window $w_3$ will hold { 011 } and $w_4$ will hold { 000 }. In this example there is no cross-over states among windows.

$$\begin{aligned}&\{(001 \rightarrow 011),\\ &(010 \rightarrow 000)\}\end{aligned} \tag{4.15}$$

In contrast, the naive method with the same example will have some cross-over states, i.e., the window $w_4$ will hold the state { 001 } and $w_3$ will hold { 010 }. Assuming the same transition in equation 4.15, then window $w_3$ will hold { 011 } whereas $w_4$ will hold { 000 }. This reveals that there are cross-over states between the windows $w_3$ and $w_4$, where by on-the-fly method there are no cross-over states. This advantage comes as the indirect effect of utilizing more variables in partitioning.

However, there might be cross-over states among the windows in on-the-fly method, but the possibility of automatic reduction does exists, whereas in the usual method this is totally impossible. Therefore, the on-the-fly method is giving some hope of automatic reduction of cross-over states.

# Chapter 5

# Static Overlap Reduction and Dynamic Removal

The divide-and-conquer approach for memory control proved to be a promising approach. However this approach has some vital points that have to be tuned in order to obtain the full benefits of it. Chapter 4 discussed the problems and the methods to tune them. This chapter details further on minimizing the overlap in the splitting technique and also the balanced distribution and cross over state reduction and its implementation details. The implementation of overlap reduction heuristics for splitting is basically adapted to the windowing techniques.

The main problem of POBDD based traversal is the redundant computation, which exists in different forms in both splitting and windowing technique. The minimal overlap algorithm aims at reducing such computation by reducing the state overlap or the cross over states. This is done by the minimal overlap algorithm that selects an intelligent variable for splitting. However, the minimal overlap algorithm can control the state overlap only to some extent. This is further handled by the dynamic overlap removal method. This chapter explains the static and the dynamic minimal overlap algorithms and its implementation.

## 5.1 Influence Factors and Overlap Reduction

The state space overlap of two sets originates from states in different sets having transitions to the same next states. However, in reality partitioning a set into two sets can hardly result in totally disjoint sets of next states, but one can put some effort in selecting the splitting variable $v$ to minimize the overlap. For finding such a good variable $v$, we utilize the influence factor table that is collected as explained in the Section 4.2.

Let us refer to Figure 5.1 in order to see how locality information affects the overlap of sets of states. For example, assume that the set of states $S = \{001, 010, 011, 100, 110, 111\}$ that are encoded by the state variables

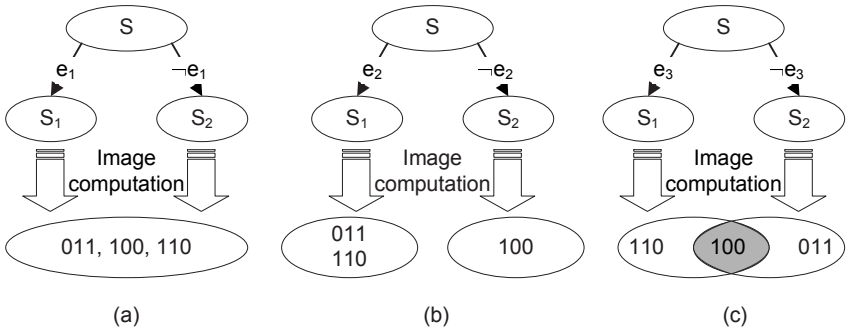


Figure 5.1: Splitting variable and their possible overlap of subsets after an image computation.

$E = \{e_1, e_2, e_3\}$. The partitioned transition relation is given in equation 5.1.

$$\begin{aligned} T_1 &= (e_1' \equiv f_1) where f_1 = \neg e_2 \vee e_3 \\ T_2 &= (e_2' \equiv f_2) where f_2 = e_2 \\ T_3 &= (e_3' \equiv f_3) where f_3 = e_2 \wedge \neg e_3 \end{aligned} \tag{5.1}$$

Let us see with this small example, how the splitting variable could affect the overlap of the two sets. Assume that the set $S$ is splitted into two sets $S_1$ and $S_2$ by the variable $e_1$ as shown in Figure 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 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 $e_2$ as shown in Figure 5.1(b), 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 $\{011, 110\}$ and the image of set $S_2$ will be $\{100\}$. Therefore there is 0% overlap. Finally, if the splitting variable is $e_3$ as shown in Figure 5.1(c), the splitted sets will be $S_1 = \{001, 011, 111\}$ and $S_2 = \{010, 100, 110\}$. 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.

A good splitting variable that reduces the state overlap can be found heuristically by analyzing the locality of the design that is implicitly given by the partitioned transition relation. The algorithm *MinOverlap* pioneers in exploiting this locality or the static information of a partitioned transition relation $\mathcal{T}$ to find a good splitting variable $v$. To understand how the locality or influence factors could help finding the best variable, assume the same example in Figure 5.1(a), where $e_1$ is the splitting variable. The locality information shows that none of the next state functions $f_i$ depend on that variable $e_1$, i.e, $\mathcal{D}_S^{\updownarrow}(e_1, 1) = \{\ \}$ In other words, the variable $e_1$ does not influence any of the next state function (next state variables). Obviously, splitting with $e_1$ does not restrict any of the next state variable $e_i'$ values. Hence this increases the probability of having same values in both splitted sets, eventually leading to a state overlap.

In the other case shown in Figure 5.1(b), $e_2$ is the splitting variable. The locality information result shows that the variable $(e_2)$ influences all the three $f_i$'s i.e., $\mathcal{D}^{\uparrow}_{\mathsf{S}}(e_2, 1) = \{e_1, e_2, e_3\}$. Therefore, the splitting with the variable $e_2$ adds on more restrictions. Hence this restriction constrains the possibilities of the next state variables, and trys to pull the image state space of the splits wide apart resulting in minimal overlap. The locality information, i.e, the *influence factors* (see equation 5.2) is collected in the pre-processing step and represented as tables mapping each variable to its value in descending order as explained in the section 4.2.

$$\begin{aligned} \Phi_{1,1}(e_2) &= 0.66 \\ \Phi_{1,1}(e_3) &= 0 \\ \Phi_{1,1}(e_1) &= -0.66 \end{aligned} \tag{5.2}$$

As explained above, splitting with a high influence variable will lead to fewer cross transitions between the resulting partitions. Our 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 *MinOverlap* algorithm (refer to Figure 5.2 for the pseudo code) picks a optimal state variable for splitting.

## 5.2 Static Overlap Reduction - MinOverlap Algorithm

The above section detailed with an example that the selected splitting variable could control the state space overlap of traversal steps. It also discussed that the high influence factor variables can control the growth of state overlap to some extent. Therefore the minimal overlap algorithm takes the high influence variables into account for the state space splitting.

The minimal overlap algorithm is also referred to as the *static overlap reduction* method, because the overlap reduction is achieved by collecting the variable influence which is a static information from the partitioned transition relation. The next sub sections details the *MinOverlap* algorithm and the dynamic overlap removal. On-the-fly method pseudo algorithm for the windowing technique is discussed at last.

### 5.2.1 MinOverlap Algorithm for Splitting Technique

The minimal overlap (*MinOverlap*) algorithm is called by the main loop of the *SymC* verification process whenever the BDD node count, representing the current state space, crosses the threshold level defined by the user. The minimal overlap algorithm partitions the state space into two parts and following one and stacking the other. However the algorithm aims at reduced overlap, the algorithm itself is motivated for full validation, which means the other stacked parts have to be also traversed at some point. Hence the partitions have to satisfy some balancing condition. Although not equal partitions, but at least the maximum of

the two parts should not be bigger than $\frac{2}{3}^{rd}$ of the actual BDD node count. In other words, the new balancing condition is defined by the equation 5.3,

$$Max(|f_v|, |f_{\overline{v}}|) \leq \delta \tag{5.3}$$

where $\delta = \frac{2}{3} \mid f \mid$ and $f$ is the BDD to be partitioned.

Figure 5.2 shows the abstract implementation details of the minimal overlap algorithm. The state variables are categorized based on their influence and put into different sets (see line 9). The function *getCandidateSet* starts with the set containing variables with a high influence and check them against a balancing condition (see line 11). Alongside, the cost of these variables is computed with the cost function from [63] that consists of a redundancy and a reduction factor. If none of the examined variables satisfied the balancing condition, the variable with minimal cost is selected (see line 15).

```
// S is the current state set                                          1
// Φ is the influence table                                            2
// δ is the memory balance factor                                      3
// α is the weight for the cost function                               4
// S1 and S2 are the resulting partitions                              5
MinOverlap(in: S, Φ, δ, α; out: S1, S2)                                6
  bestVar := Φ.top() // picks the highest influence var.               7
  minCost := cost(S, bestVar, α) // checks for the                     8
balancing condition
  //Below loop collects the set of high influence variables           9
and checks for its cost
  while C = getCandidateSet(Φ) ∧ C ≠ ∅                                10
    for all w ∈ C                                                      11
      if max(|Sw|, |S_w̄|) ≤ δ|S| then                                  12
        v := w; goto do_split                                          13
      else                                                             14
        thisCost := cost(S, w, α)                                      15
        if thisCost < minCost then                                     16
          minCost := thisCost; bestVar := w                            17
  v := bestVar                                                         18
  do_split: S1 := Sv; S2 := S_v̄                                        19
```

Figure 5.2: State set splitting with the *MinOverlap* algorithm.

The main traversal loop gets the two splits $S_1$ and $S_2$ and checks for the balancing condition. If it passes then it stacks the $S_2$ part, and continues with $S_1$. If the balancing condition is not satisfied then the main loop compromises the overlap for the balancing condition. So the main loop updates the influence limit, i.e., considers the next set of lower influence factor variables and calls the *MinOverlap* algorithm again. This time the *getCandidateSet()* function takes the next set of influence factor variables into account. This iteration goes on till the algorithm finds a split that satisfies the balancing condition.

The compromise algorithm is outlined in Figure 5.3. As mentioned above the state variables are categorized into 4 subsets according to their influence factors.

```
// S is the current state set                                    1
// δ is the memory balance factor                                2
// S1 and S2 are the resulting partitions                        3
split(in: S, δ; out: S1, S2)                                     4
  influence = 4                                                  5
  while influence > 0                                            6
    MinOverlap(S, Φ, δ, α; S1, S2)                               7
  if max(|S1|,|S2|) ≤ δ then                                     8
    influence = 0                                                9
  else // updates next set of inf.  vars.                       10
    update(getCandidateSet())                                   11
    influence--                                                 12
```

Figure 5.3: Algorithm for compromising overlap for balancing factor.

The $4^{th}$ subset contains the highest influence variables where as the $1^{st}$ subset contains the lowest influence variables. The *split* algorithm calls the *Minoverlap* algorithm, where the *getCandidateSet()* starts with the highest $4^{th}$ subset. If the balancing condition is not satisfied (line 8) then the function *getCandidateSet()* is informed to take the next level influence factors by the *update()* function (lines 11 and 12) and the loop continues till the balancing condition is satisfied or all the state variables are considered. The balancing condition is based on the cost function in definition 14. However this cost function finds the best variable to partition into two equal halves, i.e., $\delta = \frac{1}{2} \mid S \mid$ balancing condition for two partitions. In *MinOverlap* algorithm equal balancing is a secondary factor, moreover checking for all the $\alpha$ values is a time consuming process. Therefore, for the *MinOverlap* algorithm, an average value of 0.5 for $\alpha$ is used in the cost function and the the balancing condition is defined as $\delta = \frac{2}{3} \mid S \mid$. Although, fixing $\alpha$ to a constant will not result in highly concise BDD partitions, but will save splitting time and produces acceptable conciseness. The reason for compromising the equal distribution $\delta$ is due to the factor that high influence factor variables might not always distribute equally.

### 5.2.2 Dynamic Overlap Reduction for Splitting Technique

The splitting technique based on *MinOverlap* algorithm only controls and minimizes the state overlap of partitions to some extent. Depending on the design this might vary from few traversal steps to tens of steps. This can be seen in the experimental section 7. This activated the option of further removing the overlap dynamically while traversing. Hence, the visited states of a traversed partition is stored at every defined time period and later the traversal of other partitions will check and remove the visited states from its actual state space. Therefore the new partition will have only the new set of states. This simple approach proves to be an excellent time saving method in some designs that tends to have too much of overlap.

In particular, the sequential *SymC* has to traverse the splits one after the other. Moreover, the splits themselves can reach the threshold level and activates the

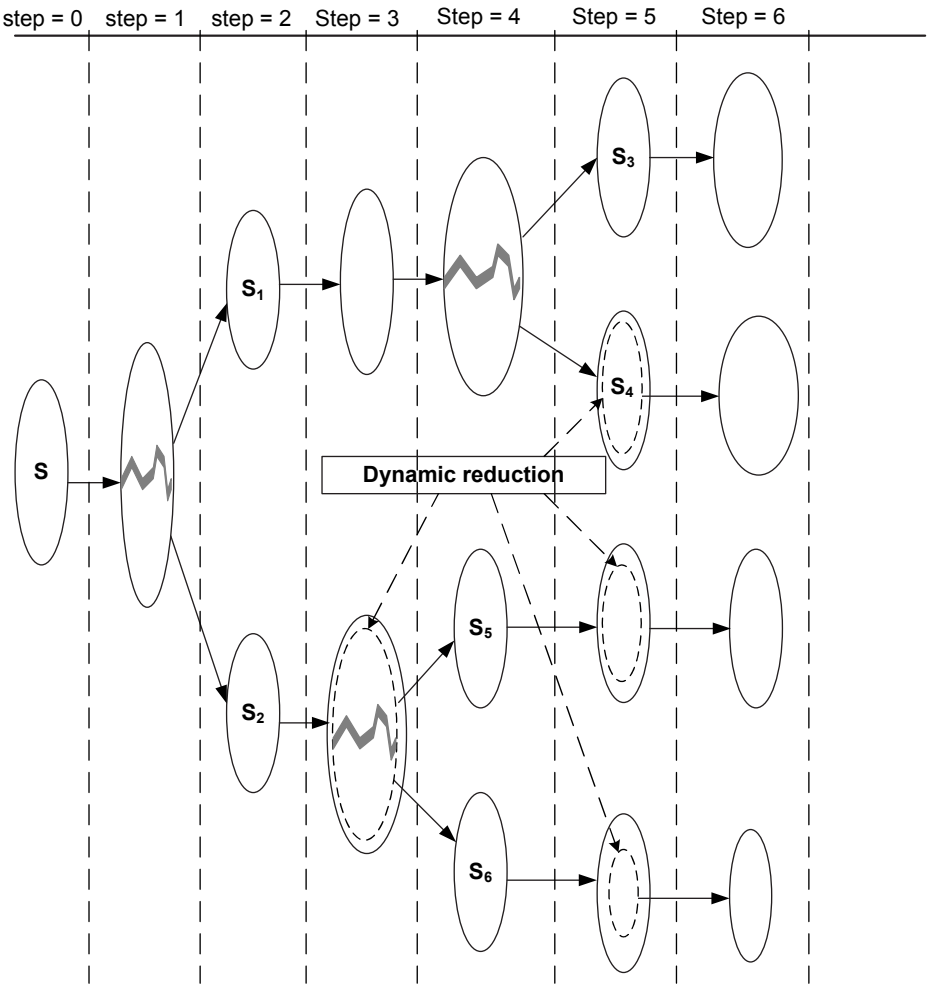


Figure 5.4: Dynamic Overlap removal

splitting process. The pseudo algorithm is shown in Figure 5.5. The dynamic overlap removal approach keeps storing its state space called *visited states* (see lines 8 - 12) for the first split traversal at every defined period of time. The later traversal splits have to remove the visited states and update the visited state space at the defined time limit (see lines 14 - 17).

Figure 5.4 shows the state space traversal, where there is three different splits made at different time point. The first split is at time step 1, and *SymC* follows the split $S_1$ storing the split $S_2$. Let us assume that the dynamic overlap removal time period 2, i.e., every second step of the traversal dynamic removal procedure should be activated. Therefore, the state space at the time step 3 has to be stored and continued. Figure 5.4 shows that the split $S_1$ at time step 4 is crossing the threshold, hence, the second splitting is done resulting in continuing with $S_3$ and stacking $S_4$. The state space of the split $S_3$ will be stored in the visited state space.

Assume, that *SymC* starts the resimulation at time step 6, then split $S_4$ will be considered for the next traversal. Split $S_4$ at time step 5 has to remove its visited state space compared to the split $S_3$ and updates with only unvisited states and continues. Similarly, when split $S_2$ starts its traversal, it has to check at time step

```
// t is the time period for dynamic removal                          1
// cycle is the present simulation time step                         2
// reSim is the resimulation indicator of other splits               3
// split is the time step of first splitting                         4
// Vstates is the map of visited states to the time step             5
// S is the actual state space                                       6
dynamic(in: t, cycle)                                                7
  if !reSim                                                          8
    if split                                                         9
      if cycle == split + t                                          10
        Vstates[cycle] = S                                           11
        split = cycle                                                12
  else                                                               13
    if Vstates.find(cycle) ≠ Vstates.end()                           14
      Overlap = S ∧ Vstates(cycle)                                   15
      S = S ∧ ! Overlap                                              16
      Vstates(cycle) = Vstates(cycle) ∨ S                            17
```

Figure 5.5: Psedo algorithm for *Dynamic* overlap removal.

3. The figure shows that even after removing the visited states still it needs to be partitioned and typically this removal procedure continues at time step 5 for the other split also.

## 5.3 On-the-fly Algorithm for Windowing Technique

The aim of the windowing technique is to have balanced and non empty windows rather than aiming at reduction of cross over states. The on-the-fly method is implemented to adopt the windowing technique and to split the state space into more than two parts.

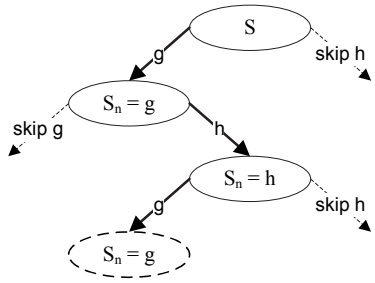


Figure 5.6: Subset tracking for windows

The *on-the-fly* algorithm finds a maximum of $2^k - 1$ variables for $2^k$ splits, which is better for a balanced distribution and non emptiness among the windows as explained in section 4.3. In principle, the on-the-fly method can utilize any of the partitioning algorithms to find the set of partitioning variables, but in this thesis we utilize *MinOverlap* algorithm. The pseudo code for obtaining the

window restriction variables is shown in Figure 5.7. The algorithm identifies the partition and the variable by right shifting the window number. Figure 5.6 pictures the tracking of obtaining the restriction variables for the window $w_5$, for example.

```
// S is the BDD representing the actual state set to be partitioned
// k defines the number of windows to be created i.e., 2^k
// Var is the splitting variable selected by MinOverlap alg.
// encode is the unique window restriction
getSubset(in: S, k)

  for i = 1 ... 2^k
    n = i
    Temp = S;
    encode = True
    for j = 1 ... k
      MinOverlap(Temp, Φ, δ, α; g, h, Var);
      if (n % 2)
        Temp = g;  // skip h for positive bit
        enclode = encode ∧ Var
      else
        Temp = h;          // skip g for negative bit
        enclode = encode ∧ ! Var
      n = n >> 1;              // get next bit
    Window − id[i] = encode
```

Figure 5.7: On-the-fly algorithm for window partitioning.

The lines 11 to 17 in Figure 5.7 shows the collection of variables for windows and this is pictured in Figure 5.6 for a explicit value of $i$ = 5 where $k$ = 3. The binary coding of 5 is ”101” which tracks first the $g$ subset and the after the right shifting tracks to the split $h$ and again to the split $g$. The high influence variables used by the *MinOverlap* algorithm tries to balance the distribution of state space among the windows. Experimental results shows that by this approach window emptiness is greatly reduced more over the maximal influence variables reduce the number of crossover states rather than the minimal influence variables.

# Chapter 6

# Guiding

Chapter 4 discussed the efficiency of POBDD based verification for solving larger problems under two different scenarios, i.e., full validation and fast falsification. Chapter 5 detailed the algorithms for optimizing the full validation approach for splitting technique and discussed an on-the-fly partitioning algorithm for windowing technique. This chapter details on the fast falsification scenario of the splitting technique by means of *guiding* and its types. The fast falsification in the windowing technique is realized by avoiding both computation and distribution of cross-over states and therefore does not require any special optimization algorithm. Hence, fast falsification by windowing is only used for comparing the results of the fast falsification of splitting technique.

Guiding is in principle a special method to traverse only a small subset of traces rather than set of all traces. This small subset of traces are identified to be interesting for a particular verification run by the information collected from the property to be verified. Trivially, this approach of guided splitting fastens the traversal and finds the target states faster. Hence, the guiding in *SymC* is a property based one, where the interesting variables are collected from the property and used in two different flavours as briefed in section 6.1. The different types of guiding in *SymC* are,

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

- Input variable guiding : The actual state set is restricted to take transitions that only accepts the selected input variables.

Both types of guiding basically aim at fast finding of at least one trace that either validate or falsifies the property depending on the quantification. The information used by both type of guiding are extracted from the property that is to be verified. The sections below detail the information collection and the implementation of both types of guiding.

## 6.1 Guiding Heuristics

Naturally, the full validation of a design is an important factor but at some stage of the design process one may be interested to check for errors in a faster way. Although the error can be detected by the full validation approach it might consume relatively more time, therefore we require an alternative approach for fast detection of errors. Such a heuristic should steer the traversal in the direction of errors that would save time and memory of the whole verification process. One of such heuristics is implemented and is called *guiding*. The guiding algorithm basically aims at reaching the target states faster and thus it partitions in a way that one of the splits is a smaller BDD and that split contains a higher percentage of potential target states. The potential target states are those states that will reach the target states after some traversal steps. Generally, small BDD means faster image computation and having the higher probability of reaching the target states in that split directly means finding the target states faster.

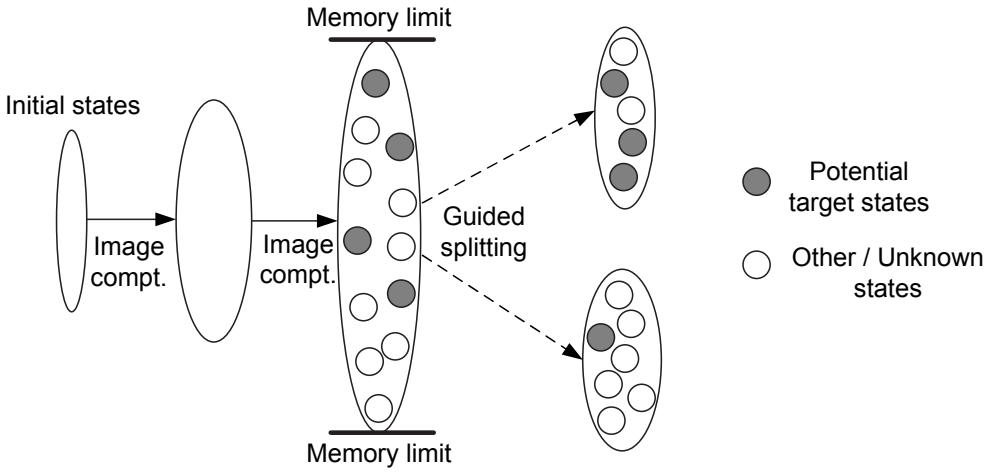


Figure 6.1: Guided splitting

The guiding algorithm is a property based heuristic. The algorithm basically collects the interesting signals that are present in the property that defines the validation of it. Once having those details the guiding algorithm can be implemented in two different flavours. The first one is to use the design's state variables, that are of interest, to partition and the second is to collect and utilize the input signals as hints to steer the traversal. Both approaches guide the traversal depending whether the property is supposed to hold in all traces, i.e., an universal property or if the property should hold in at least one of the traces, i.e., an existential property.

- In case of an universal property the heuristic should guide the traversal to the traces that have higher chance of rejecting the property.
- In case of an existential property the heuristic should guide the traversal in the direction of the traces that have higher chance of accepting the property.

Figure 6.1 portrays the guided splitting, where the grey spots are the potential target states that are identified by the information that is obtained from the property. The splitted partitions are in such a way that one partition has a higher concentration of potential target states and is relatively small in size, i.e., low BDD node count. Therefore, this is realized by first under-approximating the BDD for a low node count that represents more state space and identifying the potential target states in that under-approximated BDD using the collected information.

### 6.1.1 Guiding by State Variable

Guiding based on state variables is basically partitioning the state space into two sets 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. Let us assume the same example from the section 5.1 where S is the set of states and E is the set of state variables that encodes the states in S. The transition relation is shown again by equation 6.2. Let us define a property $P_1$ that is to be verified.

$$P_1 = F_{[1]}e_3 \tag{6.1}$$

$$\begin{aligned} T_1 &= (e_1' \equiv f_1) where f_1 = \neg e_2 \vee e_3 \\ T_2 &= (e_2' \equiv f_2) where f_2 = e_2 \\ T_3 &= (e_3' \equiv f_3) where f_3 = e_2 \wedge \neg e_3 \end{aligned} \tag{6.2}$$

The above property expresses that variable $e_3$ should be eventually true within 1 clock cycle. The information that can be obtained from the property is that the variable $e_3$ is the only interesting variable that could define the property to be valid or not. From the equation $T_3$ in 6.2 we observe that the next state value of the variable $e_3$ is dependent only on the present values of the variables $e_2$ and $e_3$. Hence, the enlarged interesting set of state variables for the guiding is $\mathcal{G}$ = $\{e_2, e_3\}$.

First, let us assume the property is to be proven universally, hence, the guiding should steer the traversal to the traces that do not accept the property. As we know that set $\mathcal{G}$ is the set of variables that decides the $e_3$'s next state value, we partition the state space with only those variables in the set $\mathcal{G}$.

So if the state space is assumed to be S = {010, 100, 110}, and if the set S is partitioned with the variable $e_3$ then one of the split is an empty partition as shown in Figure 6.2. Due to the balancing condition obviously it is not the right choice. Hence the next possible variable $e_2$ is tried that has a better partition with $S_1$ = {100} and $S_2$ = { 010, 110 } as shown in Figure 6.3. The partition is done with one of the interesting variable, but now the question is which one of the splits contains more of the potential error states. To identify the right split to be traversed a cost function (see section 6.3) is implemented. The cost function utilizes the transition relation to decide and selects the right split in order to steer the traversal. The cost function basically constrains the variable that is used for

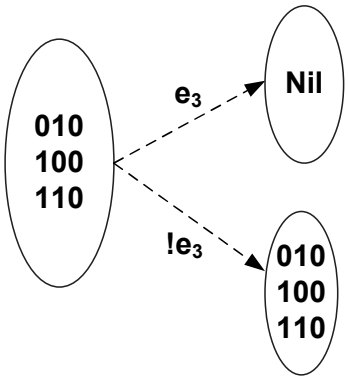


Figure 6.2: Unbalanced guided splitting for universal property

partitioning, which is one of the variable in the set $\mathcal{G}$. Depending on the number of minterms from the resulting constrained BDD, it returns the cofactor of the variable in the transition and hence the split to be followed.

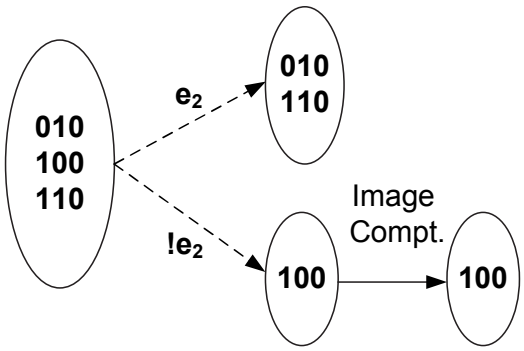


Figure 6.3: Balanced guided splitting for universal property

In our example, constraining the variable $e_2$ in the transition $T_3$ with "1" makes the resulting BDD having one minterm and if constrained with "0" the resulting BDD ends up with no minterm. Hence, the constraining with "1" is much stronger than constraining with "0" for the variable $e_2$ in transition $T_3$. In other words, when the variable $e_2$ is constrained to "0" then the whole transition is empty, i.e., no minterms remain. That unveils that the variable $e_2$ is only present as the positive cofactor, so the negative cofactor of that variable might lead to the value "0" for the variable $e_3$. Therefore we conclude to follow the partition that is "!$e_2$".

Figure 6.3 shows that following the negative cofactor of the variable $e_2$ leads to the failure of the property, giving an idea of the guiding heuristic. Assume the property shown in equation 6.1 should be proven existentially, then guiding is exactly done the same way except the interesting variable set. The interesting variable set $\overline{\mathcal{G}}$ for existential property guiding is obtained as shown in the equation 6.3.

$$\overline{\mathcal{G}} = \mathrm{E} - \mathcal{G} \tag{6.3}$$

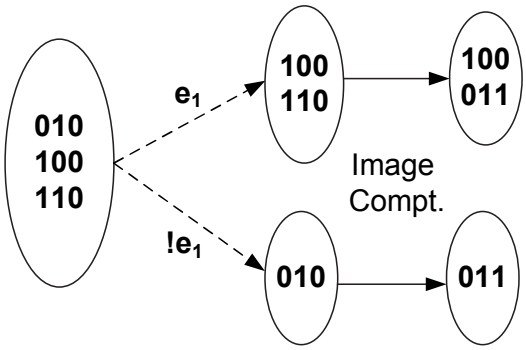


Figure 6.4: Guided splitting for existential property

where, E is the set of state variables of the design.

For the property in equation 6.1 to be proven existentially the interesting variable set for guiding will be $\overline{\mathcal{G}}$ = { $e_1$}. Figure 6.4 shows that both the partitions have a trace that leads to the acceptance of the property.

### 6.1.2 Guiding by Input Variable

Guiding based on input variables is basically restricting the possible next state set to a smaller subset. The restriction is in such a way that the subset has a higher concentration of potential target states. Further traversing this smaller subset is faster and easier to reach the error states. In contrast to the state variable guiding, input variable guiding does not depend on the high influence factor rather it depends only on the variables in the factor $\mathcal{D}_i(e)$.

Let us adopt the transition relation shown in equation 5.1 to the equation shown in the equation 6.4 in order to explain the guiding by input variables. Let us assume the set of input variables I = { $i_1, i_2$ }.

$$\begin{aligned} T_1 &= (e_1' \equiv f_1) where f_1 = (\neg e_2 \vee e_3) \vee i_1 \\ T_2 &= (e_2' \equiv f_2) where f_2 = e_2 \vee (\neg i_2 \vee i_1) \\ T_3 &= (e_3' \equiv f_3) where f_3 = e_2 \wedge (\neg e_3 \vee i_2) \end{aligned} \tag{6.4}$$

To demonstrate the guiding with input variables, let us consider the property $P_2$ in equation 6.5 that expresses that the variable $e_3$ should be true in the next step.

$$P_2 = X e_3 \tag{6.5}$$

$$\begin{aligned} \mathcal{D}_i(e_1) &= \{i_1\} \\ \mathcal{D}_i(e_2) &= \{i_1, i_2\} \\ \mathcal{D}_i(e_3) &= \{i_2\} \end{aligned} \tag{6.6}$$

From equation $T_3$ in 6.4 we observe that the next state value of the variable $e_3$ is partially dependent on the input variable $i_2$. The same is shown in the input

influence collection equation 6.6. Hence, the interesting set of input variable for the guiding is $\mathcal{I}$ = { $i_2$ }.

If we assume the property is to be proved universally, then the guiding should steer the traversal to the traces that do not accept the property. As we know that the set $\mathcal{I}$ is the set of variables that influences the $e_3$'s next state value, we partition by restricting the state space to only those transitions with variables in the set $\mathcal{I}$.

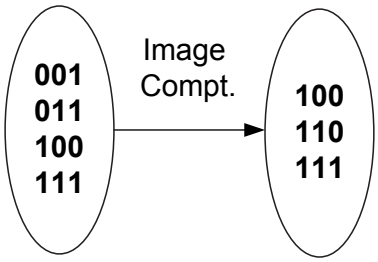


Figure 6.5: Complete next state set after the image computation step

If the state space is assumed to be S = {001, 011, 111, 100} then Figure 6.5 shows the whole set of next states as a result of the image computation. In order to guide using the input variables, the transitions are restricted to one of the cofactors of the input variable $i_2$, so that the next state space is automatically restricted. However, similar to the state variable guiding, the right cofactor restriction is a question. Therefore, the cost function that constrains the input variable in the transition relation is utilized. The cost function decision is based on the minterm count of both positive and negative cofactors of the input variable.

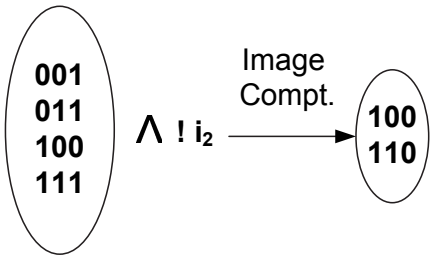


Figure 6.6: Guided by input variable $!i_2$

In our example the input variable $i_2$ is present in its positive form which will be identified by the cost function. Hence, the input variable guiding should restrict the next state space to the negative cofactor of $i_2$. The restriction is done by conjuncting the $!i_2$ to the set S. During the image computation the transitions with the $i_2$ get removed automatically, hence restricting the next state space as shown in Figure 6.6. So restricting the set S with $!i_2$ leads to the next state space $S_1$ = { 110, 100 } and this state space already found a rejection case of our property $P_2$.

If property $P_2$ should be proved existentially, then guiding is exactly done the same way except for the interesting variable set. The interesting variable set $\overline{\mathcal{I}}$ for existential property guiding is obtained as shown in the equation 6.7.

$$\overline{\mathcal{I}} = \mathrm{I} - \mathcal{I} \tag{6.7}$$

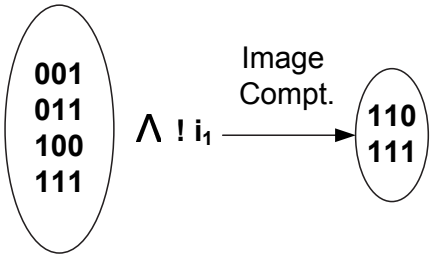


Figure 6.7: Guided by input variable $!i_1$

where, I is the set of input variables of the design.

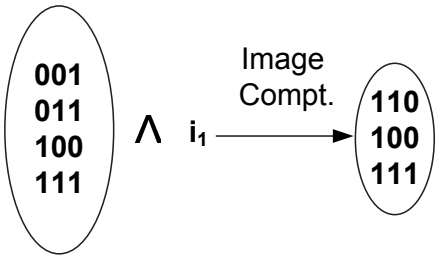


Figure 6.8: Guided by input variable $i_1$

For the property to be proved existentially the interesting variable set for guiding will be $\overline{\mathcal{I}}$ = { $i_1$ }. The Figures 6.7 and 6.8 shows that both restrictions have a trace that leads to the acceptance of the property, hence following any of the restrictions will lead to the acceptance of the property.

## 6.2 Property Based Guiding Information

Properties in *SymC* is specified in FLTL or LTL formulas. The properties are usually of the form,

$$A \rightarrow C$$

where $A$ and $C$ are FLTL/LTL formulas, with $A$ as the assumption and $C$ as the commitment. In some cases the properties contain only the commitment part, i.e., there is no assumptions. It is logical to extract the guiding information from the properties because they are the one that have to be verified against the design. The information that is supposed to be collected are the ones that decides the result of a property whether it is holding or failing in a design. Hence, for guiding only the commitment part of the property is of interest. Because the tool *SymC* is highly optimized for the traversal, the traces that are not satisfying the assumption part of the property will be automatically removed form the traversal. Therefore, whatever the property type is, for guiding all the signals that appears in the commitment part is collected. This thesis deals only with the commitment signal collection and does not cover the handling of the temporal operators of the commitment signals.

The property is taken and manipulated as a string in order to identify the commitment part and collects the set of signals that are involved in that part of the property. The property manipulation for the collection of interesting signals and their influence collection for both state and input variable guiding are shown in Figure 6.9. The commitment signals are identified by the function *getCommitment()* as shown in line 8 that takes the property as input and returns the set of commitment signals. All the interesting signals that are collected are then identified and categorized as a state variable or an output signal of the design. If any of the interesting signal is an output signal then the state variables that are involved in their definition are collected. The tracking down of output signals to a set of state variables is implemented in the function *getDefinition()* (line 12) that takes the output signal name and returns the set of state variables involved.

```
// P is the property to be verified                                    1
// G is the interesting state variables                                2
// I is the interesting input variables                                3
// sst1, sst2, sst3, sst4 are set of string                            4
// st, s are present state variables                                   5
// i is the input variable                                             6
prop_info(in: P; out: G, I)                                            7
commit = getCommitment(P) // collects the commitment part              8
sst1 = decompose(commit) // collects the interesting set of vars.      9
for all st ∈ sst1                                                      10
  if st == output-variable                                             11
    sst2 = getDefinition(st)                                           12
    for all s in sst2                                                  13
      sst3.insert(D↓_S(s,1)) // state vars in supp(t_s)                14
      sst4.insert(D_I(s)) // input vars in supp(t_s)                   15
  else                                                                 16
    sst3.insert(D↓_S(st,1))                                            17
    sst4.insert(D_I(st))                                               18
                                                                       19
G[s] = sst3                                                            20
I[i] = sst4                                                            21
```

Figure 6.9: Guiding variables collection algorithm.

Therefore, the interesting signals are always a set of state variables. This set of variables are the ones that together define the property to be valid or not. Hence, the validation of one property is now decomposed into validation of a set of variables. The variables of this set are the target variables that have to be restricted in order to guide the traversal, but this set of variables is usually small and restriction can not benefit that much. So this set of variables can be improved in order to obtain a broader scope of guiding by collecting the set of variables that influence this small set of target variables. This set of new influencing variables can be either state variables or input variables. The influence variable collection can be seen as a variant of the target enlargement method discussed in the section

3.3. Although this enlargement is done for only one step it is an approximation for more steps.

This enlargement or the collection of influencing variables are obtained using the influence factors for every variable in the interesting set (line 10). Line 11 checks whether the signal is an output signal. If yes then it gets the definition part of that output signal to identify the set of state variables involved (line 12). For every such variables the lookback or the support set of the next state function of that variable is searched for the state and input variables involved and is collected in different sets (lines 13 - 15). If the interesting signal is not a output signal then the support set is searched directly as shown in the lines 16 - 18. The final set of enlarged number of variables are categorized into state variables and input variables and are updated in the set $\mathcal{G}$ and $\mathcal{I}$ respectively (lines 20 and 21).

Obviously, the guiding technique is more efficient if the property contains only a subset of the actual set of state or output variables. For example if the property is specified to check the reachability of a state, then set $\mathcal{G}$ and $\mathcal{I}$ will be the same as the set of state and input variables, respectively. In that case guiding is of no interest as there is no special set of variables. In other words every variable is of interest, leading to no special restriction that could be applied for fast falsification. The next sections details the cost function that steers the traversal, and apply the set of collected interesting variables for fast falsification by the guiding algorithms.

## 6.3 Cost Function for Steering

Guiding in *SymC* is basically composed of two steps. One is to partition the state space into two, with one of the partitions having a higher scope of reaching the target states and second is to identify that partition and traversing it first. The second step of the guiding procedure is called *Steering* and it is implemented by the *Cost()* function. This part of steering is equally important like the part of finding the right set of variables for partitioning. Because, if steering is wrong then the guiding approach will traverse the wrong partition, which trivially increases the time for finding the target states.

$$A = \ !x_1 \ \wedge \ x_2 \ \wedge \ x_3 \tag{6.8}$$

Let us consider an example in order to explain the heuristic behind the *Cost()* function. The aim of the guiding cost function is to find the actual cofactor of a given variable in a given expression and therefore it only returns a '0' or '1'. Assume the expression $A$ as shown in equation 6.8, where $x_1$, $x_2$ and $x_3$ are the variables. If the cost function has to identify the cofactor of the variable $x_1$ in the equation $A$, then the *Cost()* function works as follows: It restricts the variable $x_1$ to its positive cofactor in the expression $A$, therefore the expression $A$ will be reduced to $A = \perp$. The number of minterms of this expression is 0. However, restricting the variable $x_1$ to its negative cofactor the expression $A$ will be reduced to $A = x_2 \wedge x_3$ having 1 minterm. The cofactor restriction that leads to a higher minterm count is the actual cofactor of the variable, i.e., the variable $x_1$ when

restricted to its negative cofactor results in a higher minterm count. Therefore the actual cofactor of the variable $x_1$ is negative in the expression $A$. In other words $!x_1$ is present in the expression $A$.

```
// T is the Partitioned transition relation                          1
// γ set of interesting signals collected from the property          2
// Φ is the set of variable to be checked for its cofactor           3
// Res is an integer variable                                        4
// Pcofactor count of Minterms after +^ve restriction of v           5
// Ncofactor count of Minterms after −^ve restriction of v           6
                                                                     7
Cost(in: T, γ, Φ; out: Res)                                          8
  Pcofactor = 0;                                                     9
  Ncofactor = 0;                                                     10
  for all c ∈ γ                                                      11
    t_c = T.find(c') // identifying the trans. of c                  12
    temp = t_c                                                       13
    for all v ∈ Φ                                                    14
      temp = temp.restrict(v) // +^ve restriction of v               15
    Pcofactor = Pcofactor + temp.CountMinterm()                      16
    temp = t_c                                                       17
    for all v ∈ Φ                                                    18
      temp = temp.restrict(!v) // −^ve restriction of v              19
    Ncofactor = Ncofactor + temp.CountMinterm()                      20
  if Pcofactor >= Ncofactor                                          21
    return 1                                                         22
  else                                                               23
    return 0                                                         24
```

Figure 6.10: Steering algorithm for guiding - *Cost()* function.

Of course, the argument here clearly assumes that the variables in the expression are conjunctively connected. This assumption is not always true in case of the next state function and therefore both best and worst performance of the cost function can not be avoided. However, with the assumption of conjunctive expression, the actual cofactor of a variable in the expression is the cofactor restriction with the highest minterm count.

The *Cost()* function is shown in Figure 6.10. This steering algorithm depends on the interesting set of variables (or tracked down state variables) that are collected from the property. The transition relation of every variable of the interesting set is identified as shown in the lines 11 and 12. Then the set of interesting variables is restricted to its positive cofactor in its corresponding transitions and the number of minterms are counted (line 14 - 16). Again, the original transitions are restricted with the negative cofactor and minterms are counted (line 17 - 20). The counting of minterm is the key idea of guessing the cofactor of the variable in the transitions. For example if the variables are connected as positive cofactor with some other variables, then restricting the variable to be "True" will leave the remaining variables in the function. These remaining variables form a minterm

of the function. If the same is present as a negative cofactor, then restricting it as "True" will result the whole function to be "False", i.e, no minterm if it is conjunctively connected. The cost function returns a 1 (*True*) in case of higher number of minterms for positive cofactor restriction (lines 21 and 22) else returns a 0 (*False*) as shown in the line 22.

## 6.4 Implementation of State Variable Guiding

The state variable guiding as briefed in section 6.1.1 is a splitting technique based on the Shannon expansion,

$$f = (v \wedge f_v) \vee (!v \wedge f_{\overline{v}})$$

where $v$ is the variable that partitions the function. In the state variable guiding the variable $v$ is always from an interesting subset of state variables. The set of interesting state variables basically depends on the property's quantification. If the property is to be verified universally then the set $\mathcal{G}$ (see Figure 6.9 line 2 and 23) is the set of interesting variables and if it should be verified existentially then the set $\overline{\mathcal{G}}$ as defined in equation 6.3 is taken into account. In any case the variable $v$, either from $\mathcal{G}$ or $\overline{\mathcal{G}}$, is selected as defined in the balancing condition. The selected variable $v$ is one of the variables that defines the validation of the property. Therefore, depending on the Boolean connectivity of the selected variable the validity of the property might differ depending on its truth value.

The algorithm for state variable guiding is shown in Figure 6.11. Assume that the variable $v$ is selected from the set $\mathcal{G}$, i.e, an universal property (line 9 -13), then that variable's truth value directly influences the validation of the property. The current state set $S$ will be partitioned into two splits $S_1$ and $S_2$ as shown in the line 25, where $S_1$ is the set that contains only the positive cofactor of the variable $v$ and vice versa for $S_2$. Hence, splitting the set $S$ into two with $v$, increases the chance of the property to be true in one of the splits. Although this totally depends on the Boolean connectivity the best case is assumed. However, guiding is achieved by traversing the split that fails the property by means of steering as shown in the lines 25 - 27. The correct split that fails the property is identified by the *Cost()* function and it is obtained by manipulating the transitions of the corresponding interesting signals and the cofactor of the variable $v$. The *Cost()* function (line 26) is the one that steers the traversal and it is detailed in section 6.3. The cost function identifies the potential target states by considering the transitions of the set of guiding variables. It returns a 1 (*True*) for the selected variable only if the variable is present in its positive cofactor form in the set of transition, hence, exchanging the set to be traversed (line 27). In other words, the guiding signals should not be allowed to be true and therefore we follow the negative restriction of the splitting variable. Because the restricted variable is present in its positive form in the transitions and following the negative cofactor might not allow the interesting set of variables to be true and therefore the traversal leads to the error states.

In contrast, if the variable is selected from the set $\overline{\mathcal{G}}$, i.e., an existential property (line 9 - 13), then that variable's truth value does not influence the validation of the property. Hence, splitting the set $S$ into two with $v$ makes sure that the property will hold in both splits. Traversing either of the splits will eventually lead to at least one trace that satisfies the property.

```
// S is the current state set                                          1
// 𝒢 is the interesting state variables                               2
// E is the set of state variables                                     3
// S1 and S2 are the resulting partitions                              4
// δ is the memory balance factor                                      5
// α is the is the weight for the cost function                        6
guide(in: S, 𝒢, E; out: S1, S2)                                        7
  minCost := cost(S, bestVar, α)                                       8
  if universal property then                                           9
   Φ := 𝒢                                                              10
  else                                                                 11
   𝒢̄ := E - 𝒢                                                          12
   Φ := 𝒢̄                                                             13
                                                                       14
  bestVar := Φ.top()                                                   15
  C := ShortPathSubset(S)                                              16
  for all w ∈ Φ                                                        17
   if max(|Sw|, |S_w̄|) ≤ δ|S| then                                     18
      v := w; goto do_split                                            19
   else                                                                20
      thisCost := Balancing-cost(S, w, α)                              21
      if thisCost < minCost then                                       22
        minCost := thisCost; bestVar := w                              23
  v := bestVar                                                         24
  do_split: S1 := Sv; S2 := S_v̄                                        25
  if Cost(v)                                                           26
    Exchange(S1, S2)                                                   27
 S2 := S2 ∨ (S ∧ !C)                                                   28
```

Figure 6.11: State variable *Guiding* algorithm.

Although the guiding helps the traversal to steer it in a right direction, still the image computation is a major and time consuming step. Hence this algorithm utilizes the under approximation technique "Short Path Subset" in order to obtain the small BDD that represents majority of the states (line 16). This small BDD is then splitted with one of the interesting variables that is selected by the cost function that depends on the memory balance factor. The subset $S_1$ is taken for the further guided traversal, where the other subset is disjuncted with the remaining part of the under approximation (line 28) and stored in the queue as a matter of completion.

## 6.5 Implementation of Input Variable Guiding

The input variable guiding as briefed in section 6.1.2 is not a straightforward splitting technique, but it is a form of a restriction that restricts the next state set. The restricted next state set is only the subset of actual next state set, therefore the input variable guiding is categorized under the splitting technique. This input variable guiding is a variant of the hint based guiding that is discussed in section 3.3.2.4. The vital difference between the hint based guiding and the input variable guiding is that the later is fully automatic, although both approaches utilize the input variables for guiding.

Guiding the traversal by input variables works differently from the state variable guiding. Although both utilize different sets of variables, the input variable guiding utilizes all the variables in the set $\mathcal{I}$ for universal properties and $\overline{\mathcal{I}}$ for the existential properties in contrast to the state variable guiding where exactly one of the best selected variables is used. This is due to the fact that state variable guiding does straight splitting of the state space, where the other restricts the next set of states. The set of interesting input variables depends basically on the property's quantification. If the property is to be verified universally then the set $\mathcal{I}$ is the set of interesting variables and if it should be verified existentially then the set $\overline{\mathcal{I}}$.

```
// S is the current state set                                          1
// I is the guiding input variables                                    2
// I is the set of all input variables                                 3
// S1 and S2 are the resulting                                         4
hint_guide(in: S, I, I; out: S1, S2)                                   5
  if universal property then                                           6
   Φ := I                                                              7
  else                                                                 8
   Ī = I - I                                                           9
   Φ := Ī                                                              10
  C := ShortPathSubset(S)                                              11
  if Cost(Φ)                                                           12
    for all i ∈ Φ                                                      13
      C := C ∧ !i // restriction of current set                        14
  else                                                                 15
    for all i ∈ Φ                                                      16
          C := C ∧ i // restriction of current set                     17
 S1 := C; S2 := S // S is stacked in case of failure of                18
guiding
```

Figure 6.12: Input variable *Guiding* algorithm.

The input variable guiding algorithm is given in Figure 6.12. Depending on the property quantification the guiding variables are taken into account as shown in the lines 7 - 11. Line 12 shows the application of under-approximation to the actual state space for the efficiency that is discussed before. The function *Cost()* in line 13 returns a 0 (*False*) if the input variables are connected as negative in

the transition of the interesting set of variables. This leads to the restriction of transitions with positive cofactor of the input variables. In other words, the set of variables in $\mathcal{I}$ are conjuncted with the current state set (lines 17 - 18). Thus the image computation of that set will not allow the transitions that do not satisfy the condition imposed by the set of conjuncted variables. Therefore, this restriction results in traversing the trace that does not satisfy the property (fast falsification for universal property). If the property is an existential one then the restriction makes sure that the satisfaction trace of the property is appearing in both restrictions as different set of variables are used.

Although the function *Cost()* (section 6.3) works in most cases, some times due to the complexity of the design, this might fail. Line 19 shows that the algorithm stacks the actual state space in case the tool does not find the target state by means of guiding. In this case the original state space is considered with usual traversal. That is, in case of failure of guiding then as a completion of the algorithm it is adopted to quit the guiding and continue with the non-guided traversal.

# Chapter 7

# Experimental Results

This chapter proves the claims that are discussed so far by means of experimental results. All the experiments are conducted on a Sun Blade 1500, SunOS 5.8, with 1GB RAM. The designs that are used in this thesis to prove the static overlap reduction and guiding algorithms are taken from standard benchmark suites, i.e., IBM benchmarks suite [79]and the ISCAS'89 benchmarks [80]. One other design is a holon example [81], where a couple of robots moves work pieces to and among processing units by storing them in buffers.

Although this thesis proposes optimization algorithms that can be exploited with any verficiation tool, the experiments are conducted with the symbolic bounded property verification tool *SymC*. The tool *SymC* utilizes the CUDD BDD package [78]. All experiments are conducted with dynamic variable ordering switched off and compares the optimization algorithms. The standard partitioning algorithms that are compared with the new *MinOverlap* and *Guiding* algorithms are partially from the CUDD package and the others are the state-of-the-art algorithms.

However, it is also important to show that the tool *SymC* without the proposed optimization is comparable and out performs other state-of-the-art tools for some designs. Figure 7.1 lists the runtime comparison of *SymC* and the state-of-the-art tools, where the un-optimized *SymC* could finish the verification of some listed ISCAS'89 designs compared to time out by the other. The first row shows the number of completed simulation steps, the second row lists the time required for the verification and the third lists the verification results, i.e., whether the property is accepted or rejected. The time out is denoted as $T_{out}$ if the tool consumed more than two hours and still did not finish the verification.

| **Verification Tools** | | **s1269** | **s1423** | **s3271** | **s4863** |
|---|---|---|---|---|---|
| **Completed traversal steps** | ***SymC*:** | 2 | 10 | 15 | 1 |
| | **SMV-LTL:** | - | - | - | - |
| **Awaited time in sec** | ***SymC*:** | 349 | 2206 | 2593 | $T_{out}$ |
| | **SMV-LTL:** | $T_{out}$ | $T_{out}$ | $T_{out}$ | $T_{out}$ |
| **Verification Result** | ***SymC*:** | Uni. Rejected | Uni. Rejected | Uni. Rejected | - |
| | **SMV-LTL:** | - | - | - | - |

Figure 7.1: *SymC* versus state-of-the-art tools for universal properties.

The other scalability comparison of *SymC* [73] with state-of-the-art tool is measured with a bus arbiter. The bus arbiter is the benchmark often used in the area

of formal methods [82, 83]. The arbiter combines a priority arbitration with a round robin technique for guaranteeing fairness, i.e., each requesting cell will finally get access to the bus. If the arbiter is very busy and works therfore in the round-robin mode, a request has to be hold at least $2n - 1$ time steps, where $n$ is the number of cells. The property in FLTL is expressed as:

$$G_{[0,2n-1]}req \rightarrow F_{[0,2n-1]}ack \tag{7.1}$$

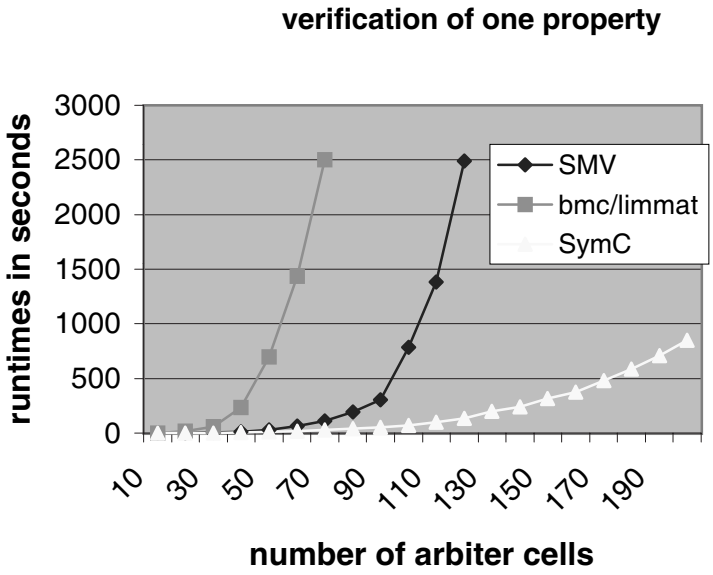


Figure 7.2: *SymC* tool performance compared to state-of-the-art tools with arbiter example.

Figure 7.2 shows the scalability of *SymC* to state-of-the-art tools by comparing the run time (y-axis) to the size of the designs (number of arbiter cells, x-axis). This comparison proves that un-optimized *SymC* is equivalent to state-of-the-art tools in performance and also better in some cases.

The optimized *SymC* experimental results are organized and compared in the following order: Section 7.1 compares the full validation approach, first by the *splitting* technique and later by the *Windowing* technique. In the splitting technique section, the standard and state-of-the-art algorithms are compared to the static overlap reduction *MinOverlap* algorithm for the verification time. The standard partitioning algorithms from CUDD package are the *variable disjunction decomposition* algorithm called *VarDisjDecomp*, and the *under-approximation* algorithm called *ShortPathSubset*. The state-of-the-art algorithms are the *equal balanced partitioning* that considers the reduction and redundancy factors called *EqualDistribution*. The windowing technique compares the usual windowing algorithm with the on-the-fly algorithm. This windowing technique better suits the distributive environment rather than the sequential environment. The distributive environment is out of this thesis scope, therefore the advantage of the on-the-fly approach is compared by means of the number of empty windows and the memory utilization of both algorithms.

Section 7.2 compares the results of the fast falsification approach, which compares the guiding algorithm with the standard algorithms and the fast falsification mode of the windowing technique. In this comparison of fast falsification, the on-the-fly algorithm is used for the windowing, where the cross-over states are not distributed among windows but kept for the second phase of verification.

Section 7.1 details and proves the strength of the static overlap reduction algorithm *MinOverlap* and the *on-the-fly* algorithm. Section 7.2 discusses the results of both the *State* and the *Input* guiding. There are two sets of experiments conducted, one for comparing the guiding algorithms with other standard partitioning algorithms for fast falsification. The second set of experiments is to prove that the guiding algorithm does not degrade so much with fully valid properties. All results are discussed by means of graphs and tables.

## 7.1 Static Overlap Reduction Results - *MinOverlap*

The properties for the IBM benchmarks are obtained from the suite itself and except the 29-batch all others are altered in order to make the property universally rejected, in order to bring the guiding effect to the picture. For ISCAS89 circuits we had no information regarding their behaviour. Therefore, the properties are the reachability of a state at high hamming distance from the initial states. For the circuit *s1423* the properties are utilized from the [84] with a time bound of 14. In general, the pre-processing time is hardly 2% of the total verification time, except 30-batch which requires more than 15%. In the holonic production system, the property specifies the consumption of a work piece.

### 7.1.1 Splitting Results

In this section the static overlap reduction heuristic *MinOverlap* algorithm is compared to a variant of the partitioning heuristic from [63], labelled *EqualDist*, and the variable disjunction decomposition algorithm from the CUDD package [78], labelled as *VarDisj*. Although, there are a number of partitioning algorithms from the CUDD package, *VarDisj* algorithm is the best performing one out of all those algorithms. The *EqualDist* algorithm has been selected as it is the state-of-the-art algorithm that partitions the state space into minimal and equal pieces.

| Design | Number of required traversal steps | Number of traversal steps finished | Awaited time in sec | Memory utilized in MB | Verification Result |
|---|---|---|---|---|---|
| **s1269** | 5 | 2 | 351.92 | 287 | Uni. Rejected |
| **s1423-I** | 10 | 10 | 2206.17 | 753 | Uni. Rejected |
| **s4863** | 4 | 1 | - | 42 | $T_{out}$ |
| **18-batch** | 130 | 45 | - | 77 | $T_{out}$ |
| **19-batch** | 130 | 46 | - | 110 | $T_{out}$ |
| **20-batch** | 130 | 52 | - | 110 | $T_{out}$ |
| **28-batch** | 18 | 17 | 604.8 | 126 | Uni. Rejected |
| **29-batch** | 30 | 20 | - | 172 | $T_{out}$ |
| **30-batch** | 10 | 10 | 899.73 | 214 | Uni. Rejected |
| **nh2-I** | 1500 | 279 | - | 895 | $T_{out}$ |

Figure 7.3: *SymC* tool performance without splitting.

There is a table (see Figure 7.3) that projects *SymC*'s performance without splitting for few of the designs. The table shows time out for 6 out of 10 designs, written as $T_{out}$, i.e. it required more than 2 hours. The remaining 4 designs were proven, but it consumed much longer time than the same verification by *SymC* done with splitting (see Figure 7.5). Moreover, the memory requirement of *SymC* for the verification process without splitting is much higher than with splitting that is compared in the graph shown in the Figure 7.4. This comparison shows that the splitting technique enables the *SymC* tool to handle bigger designs, as it requires less memory and faster verification.

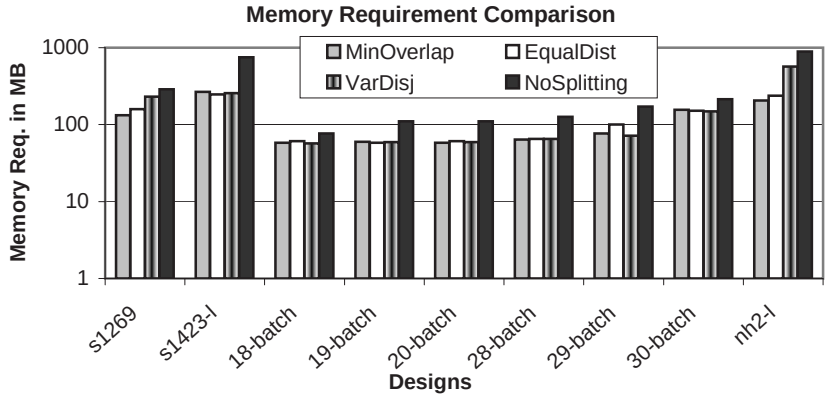


Figure 7.4: Memory requirement of *SymC* tool with and with out splitting

However, the splitting technique improves the *SymC* tool performance, the *MinOverlap* algorithm can optimize the splitting technique further. Figure 7.5 compares the verification time gain by the static reduction of overlap and also the memory requirement of each algorithm. The first, second and ninth column lists the design name, the splitting threshold limit and the verification result respectively, where as the third till eighth column lists the verification run time $(R_t)$ of the algorithms and the memory utilization (Mem.) in MB respectively. Figure 7.6 lists the verification time gain and the graph shown in Figure 7.4 shows that the memory requirement of the *MinOverlap* algorithm is stable and it is always in the lower limit compared to other algorithms in the graph. Despite the fact, for very few cases (for exampl *s3271* in Figure 7.5), the other listed algorithms require relatively lower memory revealing the instability of the approach.

Figure 7.5 lists the results with the dynamic overlap reduction enabled for all algorithms. It clearly shows the gain in the verification time by the *MinOverlap* algorithm. This time gain is due to the reduction of duplicate image computation in the splits caused by the state overlap. This state overlap calculation can be achieved better in the distributive environment and is hard to calculate in a sequential environment. This is due to the fact that the overlap calculation in a sequential environment can only be started after all the stacked partitions are traversed. Moreover, the number of partitions increases over time steps, i.e., upon reaching the threshold value splitting is done repeatedly, making the calculation of average state overlap and fair comparison over time steps impossible.

In general, static overlap reduction is advantageous for most designs, how-

| Design | Thres. | MinOverlap | | EqualDist | | VarDisj | | Result |
|---|---|---|---|---|---|---|---|---|
| | Limit | $R_t$ sec. | **Mem.** MB | $R_t$ sec. | **Mem.** MB | $R_t$ sec. | **Mem.** MB | |
| **s1269** | 5000 | **149.56** | 132 | 161.19 | 159 | $T_{out}$ | 231 | Uni. Rejected |
| **s1423-I** | 50000 | **446.94** | 266 | 490.84 | 246 | 475.58 | 256 | Uni. Rejected |
| **s3271** | 20000 | **4543.25** | 855 | 5935.16 | 601 | 5028.84 | 595 | Uni. Rejected |
| **s4863** | 20000 | **252.99** | 95 | 317.73 | 95 | $T_{out}$ | #1 | Uni. Rejected |
| **04-batch** | 20000 | **3789.14** | 64 | 5417.12 | 69 | 4935.73 | 70 | Uni. Accepted |
| **05-batch** | 20000 | **1249.25** | 58 | 1665.29 | 57 | 1490.92 | 60 | Uni. Accepted |
| **18-batch** | 20000 | **696.01** | 58 | 809.36 | 61 | 921.3 | 57 | Uni. Rejected |
| **19-batch** | 20000 | **1090.21** | 60 | 1176.28 | 58 | 1194.9 | 59 | Uni. Rejected |
| **20-batch** | 20000 | **610.69** | 58 | 809 | 61 | 913.3 | 59 | Uni. Rejected |
| **28-batch** | 20000 | **194.14** | 64 | 269.54 | 65 | 239.6 | 65 | Uni. Rejected |
| **29-batch** | 50000 | **391.44** | 77 | 1541.44 | 100 | 574.99 | 72 | Uni. Rejected |
| **30-batch** | 50000 | **395.72** | 156 | 541.73 | 152 | 490 | 149 | Uni. Rejected |
| **nh2-I** | 50000 | **2753.31** | 206 | 3496.87 | 238 | 12831 | 570 | Uni. Rejected |

Figure 7.5: Comparison of *MinOverlap* with other heuristics.

ever, there are exceptional designs in which static overlap heuristic suffers a minor degradation in the verification time. This is shown in Figure 7.7, where the first column lists the designs and the second, third and fourth column lists the verification time in seconds for *MinOverlap, EqualDist and VarDisj* algorithms respectively. The performance of the heuristic also improves along the depth of the property to be verified. For example, the depth of the property of the design *nh2-II* in Figure 7.7 is 200, where as the depth of the property in the design *nh2-I* shown in the Figure 7.5 is 1500. This design shows that *Minoverlap* gains over the depth of the property for some designs.

However, the degradation can be of two different cause, the first is due to the BDD characteristics. That is, more BDD nodes are required to represent less state space in comparison to represent more state space. Therefore, in some cases while the heuristic tries to keep the state overlap as minimal as possible, the BDD node count increases. This increase directly corresponds to a increase in the image computation time and finally the whole verification time. The authors of [62] have given evidence for the claim of increase in time with BDD node count by discussing the density of the BDDs. This revels that more number of minterms (states in our case) can be represented by relatively smaller BDDs. In any case these results has been published in the paper [85] where distributive *SymC* is the topic.

The second is due to the worst case condition of the heuristic, where the transition relation is mostly connected with disjunctive connectors. Otherwise, it is also possible that the selected set of variables are not satisfying the memory balancing condition of the partition. Therefore, the *MinOverlap* algorithm compromises with the state overlap by adding more variables into the set. However, the *MinOverlap* algorithm degrades only for few designs and it is never the worst performed algorithm and moreover it is always in the toleration level. For example, Figure 7.7 also shows that for the designs *s1512-II and 09-batch* the *VarDisj* performs relatively better. But Figure 7.5 shows that for the deigns *s1269 and s4863 VarDisj* performs the worst and similarly the *EqualDist*. This reveals that the *MinOverlap* algorithm is stable for all the designs in comparison to the other algorithms.

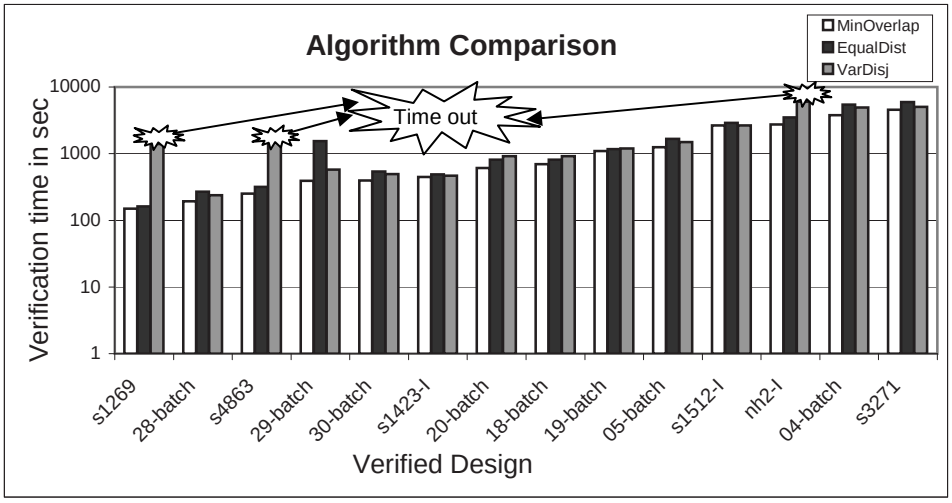


Figure 7.6: *MinOverlap* verification time comparison

| Design | MinOverlap $R_t$ in sec. | EqualDist $R_t$ in sec. | VarDisj $R_t$ in sec. |
|---|---|---|---|
| **s1512-II** | 2962.67 | 3151.28 | **2801.35** |
| **01-batch** | 588.18 | **532.22** | 553.62 |
| **09-batch** | 784.25 | 647.9 | **604.07** |
| **nh2-II** | 375.17 | **290.67** | 304.95 |

Figure 7.7: Degradation of *MinOverlap* algorithm for few cases.

Therefore, the results confirm that the *MinOverlap* algorithm performs better and also stable for most of the cases in terms of both run time and the memory requirement in compared to the other state-of-the-art splitting algorithms.

### 7.1.2 Windowing Results

The windowing technique is the other application of the static overlap reduction algorithm. This algorithm has been adopted for this technique and aimed at minimizing the empty windows and reducing the memory usage. This section lists the results of windowing technique with static overlap reduction technique called on-the-fly windowing and compared to the *Equal-Dist* windowing method. The number of windows are pre-fixed to 8 for both the approaches, however increasing the number of windows in most cases gives better results. The windowing technique generally suits best for the distributive environment of the *SymC* tool rather than the sequential *SymC*. Therefore, this section concentrates on comparing the memory requirement and number of empty windows rather than the verification time of the approach itself.

However, Figure 7.8 compares the on-the-fly static algorithm with the equal distribution method applied in the windowing technique for few of the designs. The first column lists the design name, the second and fourth column lists the verification time of the on-the-fly windowing approach and the usual windowing approach where *EqualDist* algorithm is applied respectively. The third and fifth column lists the number of empty windows created in the on-the-fly and

the usual approach, respectively. The total number of windows is fixed to 8, and the verification is done in the full validation mode, i.e., cross over states are distributed to the owners at every step of the traversal.

The Figure 7.8 compares only few of the designs as in most cases both the methods ended up in $Time_{out}$ case. However, this situation can be handled by increasing the pre-fixed number of windows. The results of this designs shows that the on-the-fly approach is efficient by means of verification time by not allowing the windows to be empty. This situation is highly desired in the distributive environment as all the network nodes will have the job to proceed. However, the cross over states can not be fairly compared, because in the on-the-fly approach there are no empty windows and hence there are relatively a higher number of cross over states compared to the usual method where there are more empty windows.

| Design | MinOverlap Windows | | EqualDist | |
|---|---|---|---|---|
| | $R_t$ sec. | Empty windows | $R_t$ sec. | Empty windows |
| **s1423-I** | **1382.23** | 0 | 1835.65 | 2 |
| **s1512-II** | **2078.69** | 0 | 2480.9 | 0 |
| **01-batch** | **319.5** | 0 | 322.2 | 0 |
| **05-batch** | **713.39** | 0 | 1282.79 | 6 |
| **30-batch** | **1138.61** | 0 | 1554.33 | 2 |

Figure 7.8: Comparison of run time and empty windows by on-the-fly and traditional windowing technique.

Figure 7.9 compares the memory requirement and the number of empty windows for the designs. Both the approaches of the windowing technique in sequential *SymC* do not differ much in terms of verification time, but clearly shows up in the distributive environment. But the distributive verification environment is out of this thesis scope, hence comparing the number of empty windows and the memory requirement of two approaches can give a fair idea of the advantage of the on-the-fly approach. The non-empty number of windows and the low memory utilization is the symbol of a fast and efficient verification. Figure 7.9 reveals that on-the-fly approach requires comparatively less memory and always zero empty windows. The first column lists the design name, where as the second and the fourth column lists the maximum memory requirement of the windows in both on-the-fly and the usual approach respectively. The third and fifth column lists the number of empty windows in both the approaches.

Therefore, the results illustrates that the windowing technique is better tunned and the potential of this approach can be improved by the *On-the-fly* method of partitioning rather than the usual *EqualDist* method.

## 7.2 Guiding Results

This section discusses the results of the fast falsification approach [86], where the properties are disproved faster. The comparison is done between the *guiding* technique and the usual approachs which are the *VarDisj* algorithm and the under-approximation algorithm called *ShortPathSubset* from CUDD package and

| Design | MinOverlap Windows | | EqualDist-Windows | |
|---|---|---|---|---|
| | Max. Memory in MB | No. Empty windows | Max. Memory in MB | No. Empty windows |
| **s1269** | **139** | 0 | 150 | 4 |
| **s1423-I** | **265** | 0 | 405 | 2 |
| **s1512-II** | 82 | 0 | **78** | 0 |
| **01-batch** | **53** | 0 | 57 | 0 |
| **04-batch** | **62** | 0 | 79 | 6 |
| **05-batch** | **60** | 0 | 66 | 6 |
| **19-batch** | **55** | 0 | 62 | 4 |
| **20-batch** | **58** | 0 | 63 | 4 |
| **23-batch** | **48** | 0 | 57 | 4 |
| **30-batch** | **163** | 0 | 192 | 2 |
| **nh2-II** | **481** | 0 | 725 | 0 |

Figure 7.9: Comparison of memory requiremnet and empty windows of the windowing techniques.

the fast falsification mode of the windowing technique. To prove the advantage of the guiding techniques, three different sets of experiments have been conducted.

The first set of experiments compares the time gain of the guiding technique to the other approaches for both the universal and existential properties. The second set of experiments is a fairness comparison that compares all the algorithms with and without the state space under-approximation (*u-approx*). This impartiality comparison is due to the fact that state space *u-approx* is itself a part of the guiding technique and therefore it is important to compare all the algorithms with and without state space *u-approx*. The last set of experiments is for the completeness, to show guiding do not degrade for full validation. Although the guiding algorithm is mainly meant for fast falsification or faster hunting of errors, in case if the design is error free then the guiding algorithm should be able to finish the verification process within the tolerance level of verification time, i.e., double the verification time required by other algorithms. The *SubsetShortPath* is used in our experiments not only as a splitting algorithm but also as a *u-approx* technique.

All the properties of the designs in this section are randomly guessed for universal rejection and existential acceptance. The properties that are used in this section generally expresses the reachability of a set of states. Figure 7.10 and 7.11 list the verification runtime in seconds for different partitioning algorithms for the universal and the existential properties. The least runtime is highlighted in the table and the speedups are explicitly given. Different set of properties are used for universal and existential verification results. The algorithms listed from third to the seventh columns are as follows: *VD* is the variable disjunctive decomposition algorithm provided by CUDD. *Win-8* is the windowing technique with 8 windows. *SP* is the *u-approx* algorithm that is also provided by CUDD, and *IG* is the guiding with input variables and *SG* is the guiding algorithm with state variables. The first column lists the design name and the second column shows the number of flipflops. The last column $S_{Up}$ is the speed up obtained by our guiding algorithms compared to the highest in the table.

The windowing technique in *SymC* is adopted for the fast falsification by avoiding the computation and distribution of cross-over states among windows. However, the results of the designs *s3271*, *23-batch* and *29-batch* shows that fast falsification by windowing technique is not by means of intelligent guiding of

| Design | FFs | VD in sec. | Win-8 in sec. | SP in sec. | IG in sec. | SG in sec. | $S_{Up}$ |
|---|---|---|---|---|---|---|---|
| **s1269** | 37 | >2 hrs | >2 hrs | 3546.0 | **2.5** | >2 hrs | 1418.4 |
| **s1423-I** | 74 | >1 hr | 708.8 | 300.5 | **122.5** | 232.7 | 5.7 |
| **s1423-II** | 74 | 469.6 | 364.9 | 175 | **85.78** | 146.8 | 5.4 |
| **s3271** | 116 | 245.8 | $C_{over}$ | 322 | **26.3** | 282.3 | 12.2 |
| **s4863** | 104 | >1 hr | >1 hr | 1198.2 | **579.6** | 1093.7 | 2.06 |
| **18-batch** | 143 | 921.3 | 1691.6 | 909.5 | 637.6 | **522.2** | 3.2 |
| **19-batch** | 181 | 1194.9 | 3362.4 | 1433.9 | 1206.9 | **846.4** | 3.9 |
| **20-batch** | 148 | 913.3 | 1675.6 | 869.1 | 662.5 | **582.0** | 2.8 |
| **23-batch** | 192 | 205.9 | $C_{over}$ | 177.6 | **73.9** | 109.3 | 2.7 |
| **28-batch** | 109 | 239.6 | 562.3 | 324.1 | 204.8 | **169.1** | 3.3 |
| **29-batch** | 95 | 322.4 | $C_{over}$ | 128.1 | 139.9 | **88.4** | 3.6 |
| **30-batch** | 191 | 493 | 587 | 294 | 284 | **205** | 2.8 |

Figure 7.10: Verification time comparison for *Universal* properties.

| Design | FFs | VD in sec. | Win-8 in sec. | SP in sec. | IG in sec. | SG in sec. | $S_{Up}$ |
|---|---|---|---|---|---|---|---|
| **s1423-I** | 74 | > 1 hr | 806.9 | 336.1 | **199.9** | 331.8 | 4.0 |
| **s1423-II** | 74 | > 1 hr | 689 | 320.1 | **181.9** | 249.3 | 3.7 |
| **10-batch** | 297 | 447.8 | 215.7 | 247 | 202.2 | **201.4** | 2.2 |
| **23-batch** | 192 | 243.6 | 401.1 | 190.1 | **139.2** | 155.8 | 2.8 |
| **28-batch** | 109 | 266.9 | 704.8 | 329.1 | **251.7** | 307.2 | 2.8 |
| **29-batch** | 95 | 569.1 | 600.7 | 257.7 | **175.6** | 190.1 | 3.4 |
| **30-batch** | 191 | 2623.6 | 1810 | 872.9 | **360.8** | 589.6 | 7.2 |

Figure 7.11: Verification time comparison for *Existential* properties.

traversal but it is only by chance if the target state is reached within the window. The lack of intelligence in the fast falsification approach of windowing technique is visible as the target state is not found and can only be reached through cross-over ($C_{over}$) states. In general the results show that the *Input Var.* and *State Var.* guiding is efficient.

To analyze the results we look at the number of input and state variables involved in the guiding process, that are listed in Figure 7.12 and Figure 7.13. The first column *Design* lists the design name, the second through the sixth are as follows: $P_v$ gives the total number of variables involved in the property, $I_n$ lists the total number of input variables, $U_i$ lists the number of input variables utilized for *Input Var.* guiding, $S_n$ lists the total number of state variables, $U_s$ lists the number of state variables utilized for *State Var.* guiding.

| Design | $P_v$ | Input Guiding | | State Guiding | |
|---|---|---|---|---|---|
| | | $I_n$ | $U_i$ | $S_n$ | $U_s$ |
| **s1269** | 24 | 18 | 15 | 37 | 36 |
| **s1423-I & II** | 1 | 17 | 9 | 74 | 27 |
| **s3271** | 43 | 26 | 26 | 116 | 42 |
| **s4863** | 9 | 49 | 33 | 104 | 40 |
| **18-batch** | 40 | 111 | 4 | 143 | 93 |
| **19-batch** | 6 | 175 | 4 | 181 | 131 |
| **20-batch** | 40 | 123 | 5 | 148 | 85 |
| **23-batch** | 7 | 153 | 6 | 192 | 24 |
| **28-batch** | 8 | 35 | 7 | 109 | 30 |
| **29-batch** | 49 | 12 | 2 | 95 | 92 |
| **30-batch** | 5 | 36 | 3 | 191 | 45 |

Figure 7.12: State and Input variables utilization for *Universal* properties.

For *Input Var.* guiding for universal properties the designs *s1269, s3271, s4863* utilizes from 67% to 100% (Figure 7.12) of the total input variables, while for *s1423* it is 41% and for *28 and 29-batch* it is around 18% and for all other design it is less than 5%. For the existential property (Figure 7.13) all the designs perform better with *Input Var.* guiding, although the designs vary in the percentage of utilization of input variables. Therefore, one can generalize that, higher the number of involved input variables, the more efficient the *Input Var.* guiding is for both universal and existential properties.

| Design | $P_v$ | Input Guiding | | State Guiding | |
|---|---|---|---|---|---|
| | | $I_n$ | $U_i$ | $S_n$ | $U_s$ |
| **s1423-I & II** | 1 | 17 | 8 | 74 | 27 |
| **10-batch** | 18 | 125 | 109 | 297 | 9 |
| **23-batch** | 2 | 153 | 153 | 192 | 24 |
| **28-batch** | 8 | 47 | 40 | 109 | 30 |
| **29-batch** | 49 | 10 | 2 | 95 | 13 |
| **30-batch** | 2 | 36 | 36 | 191 | 45 |

Figure 7.13: State and Input variables utilization for *Existential* properties.

The number of state variables involved in *SG* (Figure 7.12) are as follows: The designs *18, 19, 20, 29-batch*, utilizes more than 60% to 90% of state variables for guiding, and the results are better for these designs. The design *s1269* utilizes almost 100% but it degrades due to the fact that there is no special set of variables. In other words, all the state variables are in the interesting variable set thereby degrading the *State Var.* guiding approach to the usual one. The other designs vary from 25% to 40% resulting in moderate time gain. However, as an exception, the design *23-batch* utilizes only 3% (8%) of input (state) variables but still gives a comparatively good result. For the existential property (Figure 7.13) all the designs perform average with *State Var.* guiding, although the designs vary in the percentage of utilization of state variables. This reveals that although guiding in general increases the probability of finding a target state faster it is never 100% accurate, because it totally depends on the design and its transition relation.

Although the guiding algorithm picks up the right path, in most cases the BDD node count is large, resulting in expensive image computation steps. Thus, guiding without under-approximation works only for some designs. In order to make it applicable for wide range of the designs we combine the under- approximation along with guiding which leads to an efficient combination. Our experiments exhibited that both guiding combined with under-approximation proved to be more efficient than the other combination as shown in the Figures 7.14 and 7.15. It is a impartial comparison that compares all the algorithms with and without (*u-approx*). For the *ShortPath* algorithm which itself is an *u-approx* technique it is under-approximated two times. The symbols *VD* stands for *Variable Disjunction Decomposition*, *SP* stands for *Short Path Subset*, *IG* stands for the *Input Variable Guiding* and *SG* stands for *State Variable Guiding*.

The guiding algorithm aims at a fast falsification, i.e., to find the bugs as fast as possible. However, it should also perform better in case of no errors in the design. Therefore we conducted full validation experiments with two examples, *s1512* and *04-batch* (Figure 7.16).

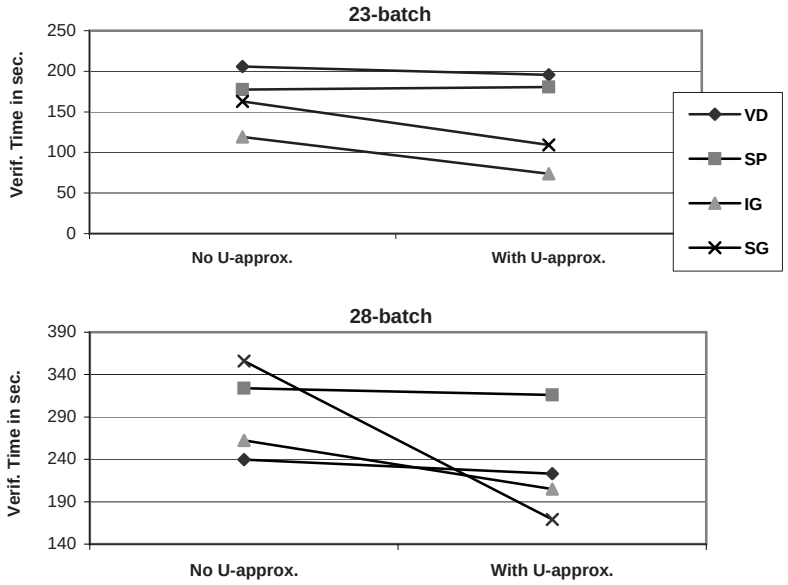


Figure 7.14: Fairness comparison IBM designs.

For the example *s1512* the *Input Var.* guiding takes the maximum time, while *State Var.* guiding performs comparatively the same. For the example *04-batch*, both *Input Var.* and *State Var.* guiding performs better than other algorithm except the windowing. Figure 7.16 shows that although the guiding algorithms is meant for fast falsification it does not degrade so much for full validation.

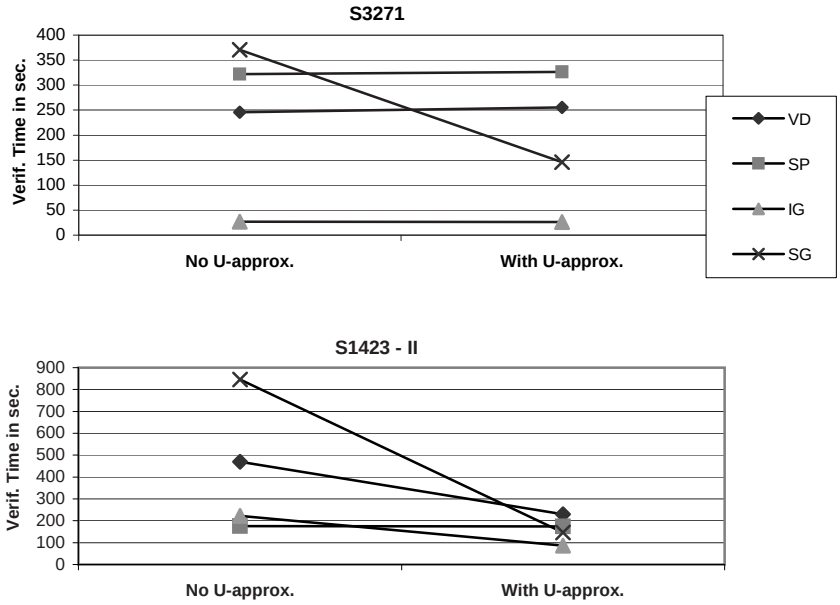


Figure 7.15: Fairness comparison ISCAS designs.

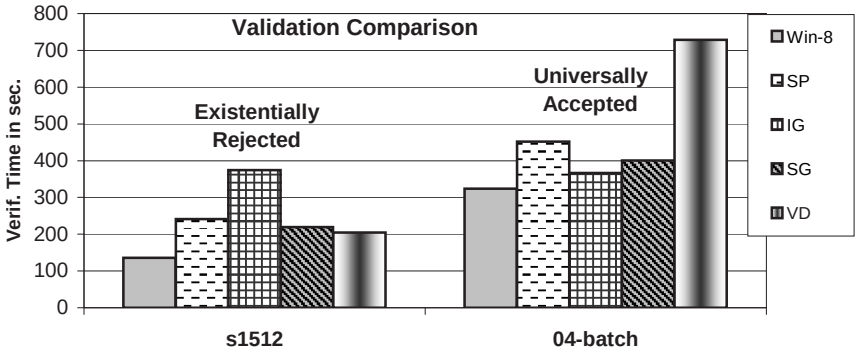


Figure 7.16: Full validation comparison.

# Chapter 8

# Conclusions and Future Work

Today's digital systems are designed by a team of designers and particularly it is verified by a different team of verification engineers. The verification team has to obtain design knowledge from the designers team. However, each individual designer has a very detailed understanding of the working of his or her unit and the way it interfaces with other units in the system. The designer's understanding of the other part of the design is at a much higher level of abstraction.

Understanding a design at various levels of abstraction is perhaps the only way a human mind can deal with the tremendous complexity of today's designs. Therefore, a designer can only be confident about certain local properties relevant to his or her unit. In any case the verification engineers have to verify the whole system and the interface of all the units by collecting properties of units individually and to get a overall picture. Hence, automatic optimization of the formal verification process helps the verification engineers to increase the confidence level and ease their work of getting detailed in the design.

Automatic optimization of the verification process directly requires one or more intelligent heuristics that decide the verification efficiency of a particular design. In order to do this these heuristics pre-process the design and extract some information for verification use. Trivially, these heuristics can be applied in different areas of the verifying process. The state-of-the-art verification process mainly concentrates on the divide-and-conquer approach. Hence, it is appropriate to concentrate on optimizing this approach further.

The key idea of this thesis is optimizing the divide-and-conquer approach. Sets of states that are to be traversed are represented symbolically by BDDs. Along with the traversal the number of states grows and hence the BDD that represents those states also grows. In order to keep the BDD growth in control the BDDs are partitioned into two or more pieces and handled separately. As a result this divide-and-conquer method makes the traversal easy and keeps the memory requirement under control. The optimization is done by an intelligent selection of variables that partitions the state space for efficient traversal or a directed traversal in search of errors for large designs.

It is exactly this phenomenon that is implemented in the *MinOverlap* and *Guiding* algorithm respectively. The *MinOverlap* algorithm is a static state overlap reduction algorithm that reduces the duplication of verification. The *Guiding* algo-

rithm is a property based automatic steering algorithm that guides the traversal to the interesting set of states.

## 8.1 Technical Contributions

In order to make *MinOverlap* and *Guiding* a viable partitioning scheme for both splitting and windowing techniques, the key technical challenges addressed in this thesis :

- An innovative categorization of state variables by the locality of the design. (Chapter 4)
- An efficient pre-processing method that analyzes the transition relation of the design and categorizes the variables according to their influence factor. (Chapter 4)
- An optimal partitioning algorithm that selects the right variable for efficient traversal considering both minimal overlap and the balancing condition of the splits. (Chapter 5)
- An innovative identification of interesting variables that influences the truth value of the property to be verified. (Chapter 6)
- An intelligent partitioning or under-approximation algorithm that selects the right partition to traverse for finding the bug faster. (Chapter 6)

## 8.2 Results

The ideas in this thesis have been evaluated on publicly available benchmark circuits from the ISCAS-89 and IBM suites. The experiments show orders of magnitude improvement in the quality of results obtained when compared with earlier schemes of partitioning. The ideas have also been evaluated on a large multi module *Holonic* design.

Using *MinOverlap* and *Guiding* as underlying partitioning schemes has enabled us to verify the large designs faster and with relatively low memory requirement. This can be automatically handled by the symbolic verification tool. Moreover, these heuristics can be easily incorporated in other verification tools.

## 8.3 Possible Future Work

Like any thesis, although some answers are provided, many more questions are raised. Even as there is some progress in automatically verifying digital systems, there is still a long way to go before any design is automatically verified. In this section, some ways in which this work can be further extended are suggested.

### 8.3.1 Hybrid Influence Calculation

The overlap reduction heuristic that minimizes the overlap of states space of different partitions totally depends on the locality information. The locality information is basically the variable interdependence in the transition relation. The collection of this interdependence is referred to as influence factor and this calculation can be repeated over a number of steps, which is referred as *lookahead* and *lookback* factors. Throughout this thesis, the experiments have been conducted with one step lookahead and one step lookback.

However, this lookback and lookahead can be increased by a few more steps to complement this heuristic. Although this point sounds interesting at first, not all designs behave the same or shows improvements for multiple step calculation. Moreover, the experience shows that the number of steps to be calculated for optimal traversal is different for different designs. Hence, this influence calculation for a design can be further studied and improved to decide the optimal number of steps for the influence calculation.

### 8.3.2 Automatic Dynamic Overlap Removal

This thesis explains the dynamic overlap removal of state space that basically removes the already visited states from further traversal. Although this technique is very useful and optimizes most designs, there are few exceptions where this state space removal increases the BDD node counts. This increase in the BDD node count eventually increases the image computation time and hence the verification time.

This dynamic algorithm can be further optimized by a heuristic that decides whether the removal decreases or increases the BDD node count. And depending on this result the dynamic overlap removal procedure can be activated.

### 8.3.3 Accurate Cost Function

The *Cost()* function that is utilized in the *Guiding* heuristic is based on assumption that the next state function mostly consists of conjunctive connections. Trivially, this assumption is not always true, hence leads to less accuracy. This cost function can be further studied to generalize this assumption further and improve the scope of accuracy.

## 8.4 Discussion

The power of formal methods, like model checking, property checking, is that they can cover all possible outcomes and hence give absolute guarantees of correctness. But that same power is also its limitation. The number of possible outcomes is astronomically large for today's large designs, and formal methods do not scale well to deal with such large problem sizes. Therefore, one of the keys for formal methods is to find a better divide-and-conquer methods. The *MinOverlap*

and *Guiding* algorithms appears to be an effective divide-and-conquer methods to help meet this challenge.

Formal verification with in-built optimization techniques appears to be very fruitful and promising area for further research. Given the rapid increase in the complexity of today's designs, the traditional simulation based empirical approach of validation will have to be necessarily augmented with divide-and-conquer formal methods.

Given the pressures of early-time-to-market and the reluctance of designers to educate themselves on latest formal verification methods, it is imperative that formal verification tools be automated and easy to use. Thus, efficient automatic optimization heuristics have to be incorporated into the formal tools. Even as this thesis helps in this automatic optimization in formal tools by enhancing the verification process of large hardware designs, there is room for improvement and a long way to go before any design is automatically verified.

# Bibliography


[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics Magazine*, vol. 38, pp. 114–117, April 1965.

[2] G.E.Moore, "Lithography and the future of moore's law," in *Proceedings of the SPIE*, vol. 2440, February 1995, pp. 2–17.

[3] *http://www.intel.com*.

[4] D. R. Hof, "Intel takes a bullet - and barely breaks stride," *Business Week*, pp. 38–39, January 1995.

[5] *http://www.toshiba.com*.

[6] A. Piziali, *Functional Verification Coverage Measurement And Analysis*. Kluwer Academic Publishers, 2004.

[7] A. S. Meyer, *Principles of Functional Verification*. Newnes, 2004.

[8] S. Iman and S. Joshi, *The e Hardware Verification Language*. Kluwer Academic Publishers, 2004.

[9] S. Palnitkar, *Verilog HDL*. Prentice Hall, 2003.

[10] J. Wing, "A specifier's introduction to formal methods," *IEEE Computer Magazine*, vol. 23, no. 9, pp. 8–24, September 1990.

[11] C.-J. Seger, "An introduction to formal hardware verification," University of British Columbia, Computer Science Department, Vancouver, Tech. Rep. TR-92-13, 1992.

[12] T. Kropf, *Introduction to Formal Hardware Verification*. Springer-Verlag, 1999.

[13] A. Cohn, "The notion of proof in hardware verification," *Journal of Automated Reasoning*, vol. 5, no. 4, pp. 127–139, 1989.

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

[15] A. Gupta, "Formal Hardware Verification Methods: A Survey," *Journal of Formal Methods in System Design*, vol. 1, pp. 151–238, 1992.

[16] W. Bibel, *Automated Theorem Proving*. Vieweg Verlag, 1987.

[17] L. Paulson, *Isabelle: A Generic Theorem Prover*, ser. Lecture Notes in Computer Science. Springer Verlag, 1994, vol. 828.

[18] D. Cyrluk, "Microprocessor Verification in PVS: A Methodology and Simple Example," Computer Science Laboratory, SRI International, Menlo Park, CA, Tech. Rep. SRI-CSL-93-12, December 1993.

[19] J. Rushby and M. Srivas, "Using PVS to prove some theorems of David Parnas," 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. 162–174.

[20] R. Cardell-Oliver, J. Herbert, and J. Joyce, "The Cambridge HOL system: An introduction to interactive machine-assisted theorem-proving in higher-order logic," Tutorial Notes, 1991.

[21] R. Boyer and J. Moore, "Proof-checking theorem-proving and program verification," *Contemporary Mathematics*, vol. 29, pp. 119–132, 1984.

[22] E. Clarke and E. Emerson, "Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic," in *Workshop on Logics of Programs*, ser. Lecture Notes in Computer Science, D. Kozen, Ed., vol. 131. Yorktown Heights, New York: Springer-Verlag, May 1981, pp. 52–71.

[23] J. Queille and J. Sifakis, "Specification and verification of concurrent systems in CESAR," in *International Symposium in Programming*, 1981.

[24] A.Pnueli, "A temporal logic of concurrent programs," *Theoretical Computer Science*, vol. 13, pp. 45–60, 1981.

[25] A. Pnueli, "The temporal semantics of concurrent programs," in *Symposium on Foundations of Computer Science*, 1977.

[26] H. Foster, A. Krolnik, and D. Lacey, *Assertion-Based Design*. Kluwer Academic Publishers, 2003.

[27] C. N. C. Jr. and H. D. Foster, "Asserstion-based verification," in *Advanced Formal Verification*, R. Drechsler, Ed. Dordrecht, The Netherlands: Kluwer Academic Publishers, 2004, pp. 167–204.

[28] M. Ben-Ari, Z. Manna, and A. Pnueli, "The temporal logic of branching time," in *ACM Symposium on Principles of Programming Languages (POPL)*, 1981, pp. 164–176.

[29] R. Bryant, "A method for hardware verification based on logic simulation," *Journal of the ACM*, vol. 38, no. 2, pp. 299–328, 1991.

[30] C. A. J. Eijk, "Formal methods for the verification of digital circuits," Ph.D. dissertation, Eindhoven University of Technology, DK 8000 Aarhus C, 1997.

[31] A. T. Eiriksson, "The formal design of 1m-gate asics," *Form. Methods Syst. Des.*, vol. 16, no. 1, pp. 7–22, 2000.

[32] J. Ruf, P. M. Peranandam, T. Kropf, and W. Rosenstiel, "Bounded property checking with symbolic simulation," in *Forum on Specification and Design Languages 2003*, 2003.

[33] R. E. Bryant, "Symbolic boolean manipulation with ordered binary-decision diagrams," *ACM Computing Surveys*, vol. 24(3), pp. 293–318, September 1992.

[34] C. Y. Lee, "Representation of switching circuits by binary-decision programs," *Bell System Technical Journal*, vol. 38, pp. 985–999, 1959.

[35] S. Akers, "Binary Decision Diagrams," *IEEE Transactions on Computers*, vol. C-27, no. 6, June 1978.

[36] R. Bryant, "Symbolic boolean manipulation with ordered binary decision diagrams," School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, Tech. Rep. CMU-CS-92-160, July 1992.

[37] G. Govindaraju, "Approximate Symbolic Model Checking Using Overlapping Projections," Ph.D. dissertation, Dept. of Electrical Engineering, Stanford University, Stanford, CA, 2000.

[38] M. Fujita, H. Fujisawa, and N. Kawato, "Evaluations and improvements of a boolean comparision program based on binary decision diagrams," in *International Conference on Computer Aided Design (ICCAD)*, 1988, pp. 2 – 5.

[39] S. Malik, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli, "Logic verification using binary decision diagrams in a logic synthesis environment," in *International Conference on Computer-Aided Design (IC-CAD)*, 1988, pp. 6 – 9.

[40] E. M. Clarke, O. Grumberg, and D. E. Peled, *Model Checking*. The MIT Press, December 1999.

[41] L. Lamport, ""sometime" is sometimes "not never"-on the temporal logic of programs," in *ACM Symposium on Principles of Programming Languages (POPL)*. New York: ACM, 1980, pp. 174–185.

[42] E. Emerson and J. Halpern, ""sometimes" and "not never" revisited: On branching versus linear time temporal logic," *Journal of the ACM*, vol. 33, no. 1, pp. 151–178, January 1986.

[43] O. Bernholtz and O. Grumberg, "Buy one, get one free!!!" Department of Computer Science, The Techion, Haifa 32000, Israel, Tech. Rep., 1994.

[44] E. Clarke, E. Emerson, and A. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic," in *ACM Symposium on Principles of Programming Languages (POPL)*, 1983.

[45] 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.

[46] R. Bryant, "Graph-based algorithms for boolean function manipulation," in *ACM/IEEE Design Automation Conference (DAC)*, 1990.

[47] 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.

[48] J. Burch, E. Clarke, D. Long, K. MacMillan, and D. Dill, "Symbolic model checking for sequential circuit verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 4, pp. 401–424, April 1994.

[49] J. Burch, E. Clarke, and D. Long, "Symbolic model checking with partitioned transition relations," in *International Conference on Very Large Scale Integration (VLSI)*, A. Halaas and P. Denyer, Eds., IFIP Transactions. Edinburgh, Scotland: North-Holland, August 1991, pp. 49–58.

[50] D. Peled, "Combining partial order reductions with on-the-fly model-checking," 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. 377–390.

[51] R. P. Kurshan, *Computer-aided Verification of Coordinating Processes – The Automata-Theoretic Approach*. Princeton Univ. Press, 1994.

[52] W. M. Elseaidy, R. Cleaveland, and J. John W. Baugh, "Modeling and verifying active structural control systems," *Sci. Comput. Program.*, vol. 29, no. 1-2, pp. 99–122, 1997.

[53] 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.

[54] D. Geist and I. Beer, "Efficient model checking by automated ordering of transition relation," 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. 299–310.

[55] 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.

[56] J. Jain, M. Abadir, J. Bitner, D. Fusell, and J. Abraham, "IBDD's: An Efficient Functional Representation for Digital Circuits," in *European Design Automation Conference*. Hamburg: IEEE, March 1992, pp. 440–446.

[57] 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.

[58] S. Iyer, C. Stangier, D. Sahoo, A. Narayan, and J. Jain, "Efficient Symbolic Model Checking using Partitioned - OBDDs," *IWLS - 12th International Workshop on Logic and Synthesis*, 2003.

[59] S. Iyer, D. Sahoo, C. Stangier, A. Narayan, and J. Jain, "Improved symbolic verification using partitioning techniques," in *Conference on Correct Hardware Design and Verification Methods (CHARME)*, 2003.

[60] B. Bollig and I. Wegener, "Partitioned BDDs vs. other BDD models," in *ACM/IEEE International Workshop on Logic Synthesis (IWLS)*, May 1997.

[61] 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.

[62] 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.

[63] 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.

[64] O. Grumberg, T. Heyman, and A. Schuster, "Distributed symbolic model checking for $\mu$-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.

[65] 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.

[66] 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.

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

[68] 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.

[69] 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.

[70] R. W. Hamming, "Error decting and error correcting codes," *Bell System Technical Journal*, vol. 9, pp. 147–160, April 1950.

[71] 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.

[72] 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.

[73] 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.

[74] E. Emerson, "Temporal and Modal Logic," in *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed., vol. B. Amsterdam: Elsevier Science Publishers, 1990, pp. 996–1072.

[75] 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.

[76] M. Rabin and D. Scott, "Finite automata and their decision problems," *IBM Journal of Research and Development*, vol. 3, pp. 115–125, 1959.

[77] R. Milner, *A Calculus of Communicating Systems*, ser. Lecture Notes in Computer Science. New York: Springer-Verlag, 1980, vol. 92.

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

[79] "IBM Formal Verification Benchmark Library," http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/benchmarks.html, IBM Haifa Research Lab, Haifa.

[80] "NCSU. ISCAS'89 Benchmark Information," http://www.visc.vt.edu/~mhsiao/iscas89.html, CAD Benchmarking Lab, North Carolina State University, Raleigh, NC 27695.

[81] J. Ruf, "Formal verification of timing properties on a holonic material transport system," WSI 2000-2, University of Tübingen, Germany, Tech. Rep., 2000.

[82] K. L. McMillan, *Symbolic Model Checking*. Norwell Massachusetts: Kluwer Academic Publishers, 1993.

[83] K. Fisler and P. Kurshan, "Verifying VHDL designs with COSPAN," in *Formal Hardware Verification – Methods and Systems in Comparison*, state of the art report ed., ser. Lecture Notes in Computer Science, T. Kropf, Ed. Springer Verlag, August 1997, vol. 1287, pp. 206–247.

[84] S. Ben-David, T. Heyman, O. Grumberg, and A. Schuster, "Scalable distributed on-the-fly symbolic model checking," in *Formal Methods in Computer-Aided Design, Third International Conference*, ser. Lecture Notes in Computer Science, W. A. Hunt, Jr. and S. D. Johnson, Eds., vol. 1954. Springer, 2000, pp. 390–404.

[85] 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)*, December 2005.

[86] 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.

# Index